中山大学计算机院本科生实验报告

（2023 学年春季学期）

课程名称：并行程序设计　　批改人：

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	21307082	姓名	赖耿桂
Email	laigg@mail2.sysu.edu.cn	完成日期	2024.4.20

1. 实验目的（200 字以内）

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。

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，并讨论程序并行性能。

2. 实验过程和核心代码（600 字以内，图文并茂）

实验 1(EquationSolving.cpp)：

根据求根公式，可以通过计算中间变量 b 的平方以及 a×c 来达到使用多线程求解方程的目的。

首先定义一些全局变量：方程的参数 a、b、c，存储 b 的平方和 a×c 的变量，存储 Δ 的变量，用于判断 b 的平方、a×c 和 Δ 是否已经计算好的条件变量，互斥锁，存储方程的根的变量，判断方程是否有解的条件变量。

```
1.  double a, b, c;     // 方程的三个参数a,b,c
2.  double square_B, A_times_C;     // 存储b*b 的结果和a*c 的结果
3.  double delta;   // 存储delta 的计算结果即b*b-4*a*c
4.  bool square_B_is_OK, A_times_C_is_OK;   // 用来判断是否已经计算出中间结果b*b 和a*c 以及最终的delta
5.
6.  pthread_mutex_t mutex;      // 互斥锁
```

```
7.  double x1,x2;   // 方程的两个根
8.  bool solution_exist;    // 记录是否一元二次方程有解
```

定义 3 个线程函数，分别计算 b 的平方、a×c 和 Δ，注意计算 Δ 的线程中要使用互斥锁保护 square_B_is_OK 和 A_times_C_is_OK 的访问和修改，以确保 Δ 的计算无误。

```
1.  // 计算b^2
2.  void* calculate_square_B(void* id){
3.      square_B = b*b;
4.      square_B_is_OK = true;
5.      pthread_exit(NULL);
6.  }
7.
8.  // 计算a*c
9.  void* calculate_A_times_C(void* id){
10.     A_times_C = a*c;
11.     A_times_C_is_OK = true;
12.     pthread_exit(NULL);
13. }
14.
15. // 计算delta
16. void* calculate_delta(void* id){
17.     // 此处要使用互斥锁，保证quare_B_is_OK 和A_times_C_is_OK 的访问和修改是安全的，这样才能保证delta 的计算无误
18.     pthread_mutex_lock(&mutex);
19.     while(!square_B_is_OK || !A_times_C_is_OK){
20.         pthread_mutex_unlock(&mutex);
21.         pthread_mutex_lock(&mutex);
22.     }
23.     delta = square_B - 4*A_times_C;
24.     pthread_mutex_unlock(&mutex);
25.     pthread_exit(NULL);
26. }
```

在 main 函数中，先输入 a、b、c，然后对一些变量进行初始化，做好多线程方程求解的准备工作。

```
    printf("请输入方程 a*x^2+b*x+c=0 的参数 a、b、 c(a、b、c 均为范围为[-100,100]的随机数，由于是一元二次方程，所以 a 不能为 0)：\n");
    scanf("%lf %lf %lf",&a,&b,&c);

    // 对标志量进行初始化
    square_B_is_OK = false;
    A_times_C_is_OK = false;
    solution_exist = false;

    pthread_t handles[3];  // 为线程分配空间
    int thread_ids[3] = {0,1,2};     // 线程号
    pthread_mutex_init(&mutex, NULL);       // 初始化互斥锁
```

接下来是核心部分，创建并启动线程，然后等待线程介绍并回收资源，并计算出所用时间。

```
    auto start_time = chrono::high_resolution_clock::now();     // 开始计时

    // 创建并启动线程
    pthread_create(&handles[0], NULL, calculate_square_B, (void*)&thread_ids[0]);
    pthread_create(&handles[1], NULL, calculate_A_times_C, (void*)&thread_ids[1]);
    pthread_create(&handles[2], NULL, calculate_delta, (void*)&thread_ids[2]);

    for(int i = 0; i < 3; ++i){
        pthread_join(handles[i], NULL);     // 等待线程结束并回收线程资源
    }

    // delta 大于或等于 0，说明方程有解，计算方程的根
    if(delta >= 0){
        x1 = (-1*b - sqrt(delta)) / (2*a);
        x2 = (-1*b + sqrt(delta)) / (2*a);
        solution_exist = true;
    }
    auto end_time = chrono::high_resolution_clock::now();   // 结束计时
    auto using_time = (double)(end_time-start_time).count() ;   // 计算出时间，此处以纳秒为单位，若使用秒为单位，可能会因为精确度不够而显示为 0
```

输出方程的解的情况并销毁互斥锁。

```
    // 方程有根，打印 2 个根
    if(solution_exist){
        printf("方程的根为：\nx1 = %lf\nx2 = %lf\n", x1, x2);
    }

    // 方程无根
    else{
        printf("方程无解.\n");
    }
    printf("并行多线程求解方程%lfx^2+%lfx+%lf=0 所用时间为：%lfns\n",a,b,c,using_time);

    pthread_mutex_destroy(&mutex);      // 销毁互斥锁
```

由于需要讨论并行性能，因此需要增加串行求解功能并进行计时。

```
    // 下为串行方程求解
    start_time = chrono::high_resolution_clock::now();     // 开始计时
    delta = b*b - 4*a*c;
    if(delta>=0){
        x1 = (-1*b - sqrt(delta)) / (2*a);
        x2 = (-1*b + sqrt(delta)) / (2*a);
        solution_exist = true;
    }
    end_time = chrono::high_resolution_clock::now();   // 结束计时
    using_time = (double)(end_time-start_time).count() ;   // 计算出时间，此处以纳秒为单位，若使用秒为单位，可能会因为精确度不够而显示为 0
    // 方程有根，打印 2 个根
    if(solution_exist){
        printf("方程的根为：\nx1 = %lf\nx2 = %lf\n", x1, x2);
    }
    // 方程无根
    else{
        printf("方程无解.\n");
    }
    printf("串行求解方程%lfx^2+%lfx+%lf=0 所用时间为：%lfns\n",a,b,c,using_time);
```

实验 2(Generator.h+MonteCarlo.cpp)：

根据实验手册可知计算 π 的公式如下：

$$\pi = \frac{4 \times \text{落在内切圆内的点数}}{\text{总点数}}$$

因此关键是统计随机点集中落在最大内切圆的点数。

首先在 Generator.h 中定义生成指定范围内的随机点和指定数量的随机点集的函数。

```
#ifndef GENERATOR_H
#define GENERATOR_H
#include<random>
#include<utility>
using namespace std;

// 生成指定范围内的点
pair<double,double> random_point_generator(){
    // 初始化随机数生成器
    random_device rd;
    default_random_engine eng(rd());
    uniform_real_distribution<double> distr(0, 1);
    return make_pair(distr(eng),distr(eng));
}


// 生成指定大小的归一化后的点集
vector< pair<double,double> > random_points_set_generator(int num){
    vector< pair<double,double> > points;
    points.resize(num);
    for(auto& point : points){
        point = random_point_generator();
    }
    return points;
}
#endif
```

在 MonteCarlo.cpp 中，首先声明一些变量(见注释介绍)

```
int point_num;      // 总点数
int thread_num;     // 线程数
int in_circle_point_num = 0;    // 在内切圆内的点数
```

```
int points_per_thread;      // 每个线程要统计的点数
const pair<double,double> origin = make_pair(0.0,0.0);      // 固定原点
vector< pair<double,double> > points;     // 点集
vector<int> in_circle_points_per_thread;    // 各个线程统计后在内切圆内的点数
double PI;
```

定义线程内统计落在内切圆内的点数的函数。

```
// 线程内统计在内切圆内的点数
void* Count_Points(void* id){
    int thread_id = *((int*)id);      // 获取线程号
    int begin_index = thread_id * points_per_thread, end_index = (thread_id+1) * points_per_thread;

    // 统计的点的索引范围

    // 遍历要统计的所有点
    for(int i = begin_index; i < end_index; ++i){
        double x_distance = points[i].first - origin.first, y_distance = points[i].second - origin.second;
        double square_distance = x_distance*x_distance + y_distance * y_distance;
        // 计算当前点与原点的距离

        if(square_distance <= 1){  // 在内切圆内，圆周上也算进去，小于等于1是因为已经进行了归一化
            in_circle_points_per_thread[thread_id] += 1;
        }
    }
    pthread_exit(NULL);
}
```

在 main 函数中，首先输入总点数和线程数，然后计算出每个线程总共需要统计的点数，初始化统计各线程计算的落在内切圆内的点数的数组，生成随机点集并为多线程作准备。

```
    printf("请输入总点数和线程数：\n");
    scanf("%d %d",&point_num, &thread_num);

    const int THREAD_NUM = thread_num;
    points_per_thread = point_num / THREAD_NUM;    // 算出每个线程要统计多少个点
    in_circle_points_per_thread.assign(THREAD_NUM+1, 0);      // 初始化各个线程内在内切圆内的点数
    points = random_points_set_generator(point_num);   // 生成在正方形内的随机点集
```

```
8.
9.      pthread_t handles[THREAD_NUM];  // 为线程分配空间
10.     int thread_ids[THREAD_NUM];     // 线程号
```

接下来是核心部分，创建并启动线程，然后将各个线程统计的落在内切圆内的点数加起来后，计算π，再算出所用时间。

```
1.      auto start_time = chrono::high_resolution_clock::now();     // 记录开始时间
2.
3.      for(int i = 0; i < THREAD_NUM; ++i){
4.          thread_ids[i] = i;  // 线程号
5.          pthread_create(&handles[i], NULL, Count_Points, (void*)&thread_ids[i]);     // 创建并启动线
    程
6.      }
7.
8.      for(int i = 0; i < THREAD_NUM; ++i){
9.          pthread_join(handles[i],NULL);   // 等待线程结束并回收线程资源
10.     }
11.
12.     if(point_num % THREAD_NUM != 0){        // 点数不一定能被线程数整除，应统计剩余的
    (point_num%THREAD_NUM)个点
13.         int cnt = 0;
14.         for(int i = THREAD_NUM * points_per_thread; i < point_num; ++i){
15.             double x_distance = points[i].first - origin.first, y_distance = points[i].second - ori
    gin.second;
16.             double square_distance = x_distance*x_distance + y_distance * y_distance;
17.             // 计算当前点与原点的距离
18.
19.             if(square_distance <= 1){  // 在内切圆内，圆周上也算进去
20.                 cnt += 1;
21.             }
22.         }
23.         in_circle_points_per_thread[THREAD_NUM] = cnt;
24.     }
25.
26.     // 将各个线程统计到的在内切圆内的点数加起来
27.     for(auto& num : in_circle_points_per_thread){
28.         in_circle_point_num += num;
29.     }
30.     PI = (double)in_circle_point_num * 4 / (double)point_num;   // 计算π值
31.
32.     auto end_time = chrono::high_resolution_clock::now();   // 记录结束时间
```

```
33.     auto using_time = (double)(end_time - start_time).count() / 1e9;
```

输出相关信息（包括求出的 π 值以及执行时间）。

```
1.    printf("\n 总点数为：%d\n 落在内切圆内的点数为：%d\n 估算的π值为：%lf\n 线程数为：%d\n 所用时间
    为:: %lfs\n",point_num,in_circle_point_num,PI,THREAD_NUM,using_time);
```

3. 实验结果（500 字以内，图文并茂）

实验 1：

■ 编译

```
1.    g++ -g -Wall -o EquationSolving EquationSolving.cpp -lpthread
```

■ 运行

```
1.    ./EquationSolving
```

■ 该实验主要比较的是并行和串行的性能差距，因此下面展示Δ＞0、Δ=0、Δ＜0 3 种情况的运行结果：

1) Δ＞0：

```
laigg@laigg-VirtualBox:~/codes/parallel/lab4$ g++ -g -Wall -o EquationSolving EquationSolving.cpp -lpthread
laigg@laigg-VirtualBox:~/codes/parallel/lab4$ ./EquationSolving
请输入方程 a*x^2+b*x+c=0 的参数a、b、 c(a、b、c均为范围为[-100,100]的随机数，由于是一元二次方程，所以a不能为0):
10 30 20
方程的根为:
x1 = -2.000000
x2 = -1.000000
并行多线程求解方程10.000000x^2+30.000000x+20.000000=0所用时间为: 166652.000000ns
方程的根为:
x1 = -2.000000
x2 = -1.000000
串行求解方程10.000000x^2+30.000000x+20.000000=0所用时间为: 69.000000ns
```

2) Δ=0：

```
laigg@laigg-VirtualBox:~/codes/parallel/lab4$ ./EquationSolving
请输入方程 a*x^2+b*x+c=0 的参数a、b、 c(a、b、c均为范围为[-100,100]的随机数，由于是一元二次方程，所以a不能为0):
25 50 25
方程的根为:
x1 = -1.000000
x2 = -1.000000
并行多线程求解方程25.000000x^2+50.000000x+25.000000=0所用时间为: 150853.000000ns
方程的根为:
x1 = -1.000000
x2 = -1.000000
串行求解方程25.000000x^2+50.000000x+25.000000=0所用时间为: 64.000000ns
```

3) Δ<0：

```
laigg@laigg-VirtualBox:~/codes/parallel/lab4$ ./EquationSolving
请输入方程 a*x^2+b*x+c=0 的参数a、b、 c(a、b、c均为范围为[-100,100]的随机数，由于是一元二次方程，所以a不能为0):
15 15 15
方程无解.
并行多线程求解方程15.000000x^2+15.000000x+15.000000=0所用时间为: 172055.000000ns
方程无解.
串行求解方程15.000000x^2+15.000000x+15.000000=0所用时间为: 48.000000ns
```

可以看到并行多线程的性能远不如串行性能，可能有以下原因：

①该问题的关键是求解Δ，这是一个小规模问题，使用并行多线程计算Δ提升的性能还不足以弥补多线程带来的通信开销。

②在求解Δ的线程中使用了互斥锁来保证Δ的正确性，会造成线程阻塞，进而增加执行时间。

实验2:

■ 编译

```
1.  g++ -g -Wall -o MonteCarlo MonteCarlo.cpp -lpthread
```

■ 运行

```
1.  ./MonteCarlo
```

■ 下为运行示例，仅展示程序正确性：

```
laigg@laigg-VirtualBox:~/codes/parallel/lab4$ g++ -g -Wall -o MonteCarlo MonteCarlo.cpp -lpthread
laigg@laigg-VirtualBox:~/codes/parallel/lab4$ ./MonteCarlo
请输入总点数和线程数:
1024 2

总点数为: 1024
落在内切圆内的点数为: 799
估算的π值为: 3.121094
线程数为: 2
所用时间为: 0.000173s
```

不同线程下不同总点数的运行时间：

线程数	总点数						
	1024	2048	4096	8192	16384	32768	65536
1	0.000187s	0.000235s	0.000299s	0.000334s	0.000428s	0.000607s	0.000993s
2	0.000143s	0.000162s	0.000228s	0.000337s	0.000361s	0.000566s	0.000780s
4	0.000165s	0.000190s	0.000287s	0.000350s	0.000435s	0.000468s	0.000744s
8	0.000240s	0.000258s	0.000314s	0.000324s	0.000533s	0.000435s	0.000736s
16	0.000389s	0.000380s	0.000469s	0.000583s	0.000769s	0.000535s	0.000700s

可以看到对于总点数较少时(1024~16384)，除了从 1 个线程提升至 2 个线程时，性能稍微有所提高之外，再继续增加线程会导致运行时间增加、性能有所下降。很可能是因为问题规模太小，使用并行多线程是性能提高不大，反而是通信开销导致的性能下降影响更大。

4. 实验感想（200 字以内）

在实验一的计算 Δ 的线程中，一开始我直接在 while 循环中计算 Δ 且未加互斥锁进行保护，导致 Δ 的值不准确，经过分析，可能是未加互斥锁保护 square_B_is_OK 和 A_times_C_is_OK 的访问和修改(可能会出现这种情况：square_B_is_OK 为 true 但 A_times_C_is_OK 为 false，计算 Δ，然后在下一次循环开始前线程 calculate_A_times_C 算出 a×c 使 A_times_C_is_OK 为 true，导致 Δ 的值不准确)。因此后面增加了互斥锁并将 Δ 的计算移出 while 循环后解决问题。这给我很大的启发，在并行编程中，条件变量的识别不能想当然，要严格监控并进行保护。