Lab4 - Pthreads并行方程求解及蒙特卡洛

实验要求

一、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：

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

输入：a,b,c三个浮点数，其的取值范围均为[-100, 100]

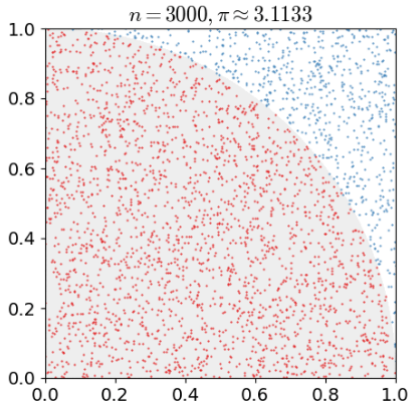
问题描述：使用求根公式并行求解一元二次方程$ax^2+bx+c=0$。

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。

二、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（见上图），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。

实验过程

一、一元二次方程求解

1. 实现思路

中间值：$A = b^2$、$B = 4ac$、$C = \sqrt{A - B}$、$D = 2a$

最终计算：$x_1 = \dfrac{-b + C}{D}$、$x_2 = \dfrac{-b - C}{D}$

2. 代码实现

线程函数：中间量计算，其中计算$C = \sqrt{A - B}$时通过条件变量等待A、B计算完毕

```
void* double_b(void* id) {
    A = b * b;
    pthread_mutex_lock(&mutex);
    flag += 1;
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}

void* four_ac(void* id) {
    B = 4 * a * c;
    pthread_mutex_lock(&mutex);
    flag += 1;
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}
void* sqrt_bac(void* id) {
    while (flag < 2) {}
    C = A - B;
    if (C >= 0) {
        exist = 1;
        C = sqrt(C);
    }
    pthread_exit(NULL);
}
void* two_a(void* id) {
    D = 2 * a;
    pthread_exit(NULL);
}
```

主程序

```
// 创建线程
pthread_create(&threads[0], NULL, double_b, NULL);
pthread_create(&threads[1], NULL, four_ac, NULL);
pthread_create(&threads[2], NULL, sqrt_bac, NULL);
pthread_create(&threads[3], NULL, two_a, NULL);
// 等待所有线程执行完毕
for (int i = 0; i < 4; ++i) {
    pthread_join(threads[i], NULL);
```

```
}
// 最后计算方程的解
if (exist) {
    double x1 = (-b + C) / D;
    double x2 = (-b - C) / D;
    printf("x1 = %f, x2 = %f\n", x1, x2);
}
else {
    printf("no solution existing\n");
}
```

3. 运行结果

```
gcc -g -Wall -o pe pequation.c -lpthread -lm
./pe
```

```
ehpc@61583b2ed2e2:~/data$ gcc -g -Wall -o pe pequation.c -lpthread -lm
ehpc@61583b2ed2e2:~/data$ ./pe
Enter the value of a b c between -100 and 100: 80 100 30
x1 = -0.500000, x2 = -0.750000
Time taken: 0.006731 seconds
ehpc@61583b2ed2e2:~/data$ ./pe
Enter the value of a b c between -100 and 100: 1 2 1
x1 = -1.000000, x2 = -1.000000
Time taken: 0.002107 seconds
ehpc@61583b2ed2e2:~/data$ ./pe
Enter the value of a b c between -100 and 100: 1 1 1
no solution existing
Time taken: 0.002409 seconds
```

4. 性能分析

与串行计算比较：a = 80, b = 100, c =30，统计耗时

并行	串行
0.006812	0.000287
0.003409	0.000238
0.006434	0.000043
0.004306	0.000053

可见并行计算耗时更长，多线程切换的开销大于并行计算的节省耗时

二、蒙特卡洛方法求π的近似值

1. 实现思路

通过数组`x`、`y`存储随机点的坐标，线程平均划分随机点，计算与原点的距离判断是否在四分之一圆内，每个线程累加在圆内的随机点的数量，最后通过互斥量累加到全部随机点在圆内的个数，计算得到π值。

2. 代码实现

线程传递参数

```
struct ThreadData {
    float* x;
    float* y;
    int start;
    int end;
};
```

每个线程统计在圆内的点的数

```
void* partial_count(void* arg) {
    struct ThreadData* data = (struct ThreadData*)arg;
    int tmp = 0;
    for (int i = data->start; i < data->end; i++) {
        if (data->x[i] * data->x[i] + data->y[i] * data->y[i] <= 1) {
            tmp += 1;
        }
    }
    pthread_mutex_lock(&mutex);
    sum += tmp;
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}
```

主程序

```
pthread_t threads[num_threads];
struct ThreadData thread_data[num_threads];
// 线程负责长度
int rows_per_process = n / num_threads;
int remaining_rows = n % num_threads;
float* x = (float*)malloc(n * sizeof(float));
float* y = (float*)malloc(n * sizeof(float));
// 创建随机点
for (int i = 0; i < n; i++) {
    x[i] = (float)rand() / RAND_MAX;
    y[i] = (float)rand() / RAND_MAX;
}
int offset = 0;
// 划分线程负责范围，创建线程
for (int i = 0; i < num_threads; i++) {
    thread_data[i].x = x;
    thread_data[i].y = y;
    thread_data[i].start = offset;
    offset += (i < remaining_rows) ? rows_per_process + 1 : rows_per_process;
    thread_data[i].end = offset;
    pthread_create(&threads[i], NULL, partial_count, (void*)&thread_data[i]);
}
// 等待所有线程完成
for (int i = 0; i < num_threads; i++) {
    pthread_join(threads[i], NULL);
```

```
}
```

3. 运行结果

```
gcc -g -Wall -o pm Montecarlo.c -lpthread
./pm 4
```

```
ehpc@61583b2ed2e2:~/data$ gcc -g -Wall -o pm Montecarlo.c -lpthread
ehpc@61583b2ed2e2:~/data$ ./pm 4
Enter the value of n (1024 - 65536): 65536
estimate pi: 3.138489
Time taken: 0.004860 seconds
```

4. 性能分析

统计不同线程数，不同数据量下的平均耗时，时间单位ms

线程数\统计量	1024	4096	16384	65536
1	0.120	0.157	0.267	0.917
2	0.216	0.260	0.464	1.043
4	0.391	0.430	0.707	1.311
8	0.750	0.819	1.105	1.714
16	1.333	1.429	1.573	2.246

观察数据，随着并行线程数上升，统计程序的耗时没有下降，可见线程调度开销远大于并行计算优势，由于统计数据量的限制在65536，并行计算优势还不明显，但可以观察到并行线程数较大时，随着数据量的翻倍，程序计算时间的翻倍越来越不明显，16线程时不同数据量的耗时较为接近，可以推断在更大数据量时多线程并行能够取得更大优势。