# Parallel-Programming Task4

刘森元, 21307289

中山大学计算机学院

Codes on https://github.com/Myocardial-infarction-Jerry/Parallel-Programming/tree/main/Task4.

Project built by CMake.

```
> cd Task4
> cmake . && make
> ./PthreadEquation <a> <b> <c>
> ./PthreadMonteCarlo <N>
```

# 1 Environment

11th Gen Intel(R) Core(TM) i7-11700KF @ 3.60GHz

NVIDIA GeForce RTX 3080 Ti O12G

Windows Subsystem for Linux @ Ubuntu 22.04 LTS

# 2 Task

## 2.1 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，根据数据及任务之间的依赖关系，及实验计时，分析其性能。

> **Note**
>
> **一元二次方程：** 为包含一个未知项，且未知项最高次数为 2 的整式方程事，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，其取值范围为 $[-100, 100]$
>
> **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 Pthread 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。

## 2.2 蒙特卡洛方法求 $\pi$ 的近似值

给予 Pthread 编写多线程程序，使用蒙特卡洛方法求圆周率 $\pi$ 近似值。

ⓘ Note

**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 $\pi$ 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 $\pi/4$，可据此来估计 $\pi$ 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 $\pi$ 的值。

**输出：** 总点数 $n$，落在内切圆内点数 $m$，估算的 $\pi$ 值，及消耗的时间 $t$。

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

# 3 Theory

## 3.1 一元二次方程求解

一元二次方程求根公式可分解为

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

故可以将其分配给两个线程计算，其中求根线程依赖于 $\Delta$ 线程。

## 3.2 蒙特卡洛方法求 $\pi$ 的近似值

在计算圆周率 $\pi$ 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 $\pi/4$，可据此来估计 $\pi$ 的值。

# 4 Code

## 4.1 一元二次方程求解

ⓘ Caution

源代码详见 [PthreadEquation.cpp](PthreadEquation.cpp)

```cpp
// 求解 Delta
void *computeDiscriminant(void *arg) {
    QuadraticEquation *eq = (QuadraticEquation *)arg;
    pthread_mutex_lock(&eq->mutex);

    eq->discriminant = eq->b * eq->b - 4 * eq->a * eq->c;
    eq->discriminantComputed = true;

    pthread_cond_signal(&eq->cond);
    pthread_mutex_unlock(&eq->mutex);
    return nullptr;
}

// 求根
```

```cpp
void *computeRoots(void *arg) {
    QuadraticEquation *eq = (QuadraticEquation *)arg;
    pthread_mutex_lock(&eq->mutex);

    while (!eq->discriminantComputed)
        pthread_cond_wait(&eq->cond, &eq->mutex);

    if (eq->discriminant >= 0) {
        eq->root1 = (-eq->b + sqrt(eq->discriminant)) / (2 * eq->a);
        eq->root2 = (-eq->b - sqrt(eq->discriminant)) / (2 * eq->a);
    }
    else {
        double realPart = -eq->b / (2 * eq->a);
        double imagPart = sqrt(-eq->discriminant) / (2 * eq->a);
        eq->root1 = std::complex<double>(realPart, imagPart);
        eq->root2 = std::complex<double>(realPart, -imagPart);
    }

    pthread_mutex_unlock(&eq->mutex);
    return nullptr;
}

int main(int argc, char *argv[]) {
    ...

    pthread_t thread1, thread2;
    pthread_create(&thread1, nullptr, computeDiscriminant, &eq);
    pthread_create(&thread2, nullptr, computeRoots, &eq);

    pthread_join(thread1, nullptr);
    pthread_join(thread2, nullptr);

    ...
}
```

## 4.2 蒙特卡洛方法求 $\pi$ 的近似值

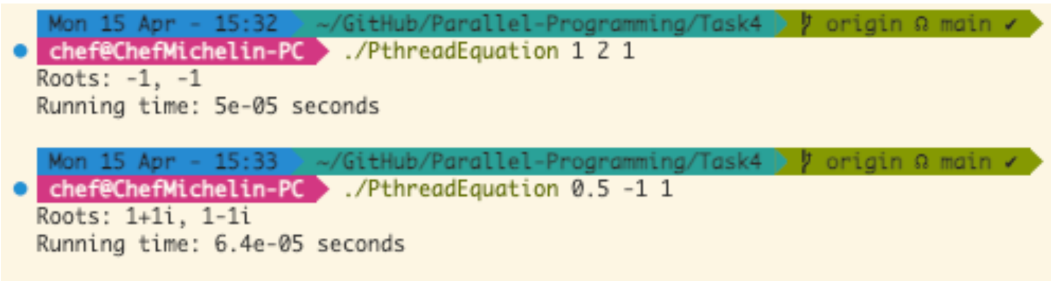
> **Caution**
>
> 源代码详见 PthreadMonteCarlo.cpp

```cpp
void *monteCarlo(void *arg) {
    ThreadData *data = (ThreadData *)arg;
    data->countInCircle = 0;

    unsigned int randState = time(NULL);
    for (int i = 0; i < data->numPoints; ++i) {
        double x = (double)rand_r(&randState) / RAND_MAX * 2 - 1;
        double y = (double)rand_r(&randState) / RAND_MAX * 2 - 1;
        if (x * x + y * y <= 1) {
            ++data->countInCircle;
        }
    }
```

```
    return nullptr;
}

int main(int argc, char *argv[]) {
    ...

    pthread_t threads[NUM_THREADS];
    ThreadData threadData[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; ++i) {
        threadData[i].numPoints = pointsPerThread;
        pthread_create(&threads[i], nullptr, monteCarlo, &threadData[i]);
    }

    int totalInCircle = 0;
    for (int i = 0; i < NUM_THREADS; ++i) {
        pthread_join(threads[i], nullptr);
        totalInCircle += threadData[i].countInCircle;
    }

    ...
}
```

# 5 Result

## 5.1 一元二次方程求解




这段代码的并行性能主要取决于两个线程 `computeDiscriminant` 和 `computeRoots` 的执行时间。这两个线程分别计算二次方程的判别式和根。

代码使用了互斥锁和条件变量来同步两个线程。这些同步操作可能会引入一些开销，尤其是在高并发的情况下。

总的来说，代码的并行性能并不高，因为 `computeRoots` 线程的执行依赖于 `computeDiscriminant` 线程的完成，而且同步操作可能会引入额外的开销。

## 5.2 蒙特卡洛方法求 $\pi$ 的近似值

```
Mon 15 Apr - 15:33  ~/GitHub/Parallel-Programming/Task4  origin main ✓
chef@ChefMichelin-PC  ./PthreadMonteCarlo 1024
Running with 8 threads
Total points in circle: 792
Pi estimate: 3.09375000
Running time: 0.00023000 seconds

Mon 15 Apr - 15:35  ~/GitHub/Parallel-Programming/Task4  origin main ✓
chef@ChefMichelin-PC  ./PthreadMonteCarlo 65536
Running with 8 threads
Total points in circle: 51864
Pi estimate: 3.16552734
Running time: 0.00021500 seconds
```

并行性能主要取决于以下几个因素：

1. **线程数量**：程序使用了 `NUM_THREADS` 个线程来并行生成随机点和计算落在单位圆内的点的数量。增加线程数量可以提高并行性能，但是当线程数量超过处理器核心数量时，性能可能会下降，因为线程切换的开销可能会超过并行计算的收益。
2. **采样点数量**：每个线程都生成了 `numPoints / NUM_THREADS` 个随机点。增加采样点数量可以提高精度，但也会增加计算的时间。
3. **同步开销**：程序使用了 `pthread_join` 函数来等待所有线程完成。这会引入一些同步开销，尤其是在线程数量很大时。
4. **随机数生成**：程序使用了 `rand_r` 函数来生成随机数。这个函数是线程安全的，但是可能比非线程安全的随机数生成函数（如 `rand`）慢一些。

并行性能取决于线程数量、采样点数量、同步开销和随机数生成的速度。