中山大学计算机院本科生实验报告

（2024 学年春季学期）

课程名称：并行程序设计	批改人:
实验：4	专业（方向）：计算机科学与技术
学号：22336226	姓名：王泓沣
Email：wanghf59@mail2.sysu.edu.cn	完成日期：2025/4/22

1 实验目的

1.1 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。

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。

1.2 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近似值。

输入： 整数，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，落在内切圆内点数，估算的值，及消耗的时间。

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

2 实验过程和核心代码

2.1 一元二次方程求解

使用四个布尔变量代表四个中间值的计算结果：$b^2, 4ac, \Delta, \sqrt{\Delta}$

```
bool ready_b2 = false, ready_4ac = false, ready_disc = false, ready_sqrt
   = false;
```

计算每一个中间值或根的函数如下，包括对共享资源（变量值）加锁、等待、计算、广播、解锁等。

```
void *thread_compute_b2(void *arg) {
    pthread_mutex_lock(&mtx);
    b2 = b * b;
    ready_b2 = true;
    if (ready_4ac) pthread_cond_broadcast(&cv_both);
    pthread_mutex_unlock(&mtx);
    return NULL;
}

void *thread_compute_4ac(void *arg) {
    pthread_mutex_lock(&mtx);
    four_ac = 4.0 * a * c;
    ready_4ac = true;
    if (ready_b2) pthread_cond_broadcast(&cv_both);
    pthread_mutex_unlock(&mtx);
    return NULL;
}

void *thread_compute_disc(void *arg) {
    pthread_mutex_lock(&mtx);
    while (!(ready_b2 && ready_4ac)) pthread_cond_wait(&cv_both, &mtx);
    discriminant = b2 - four_ac;
    ready_disc = true;
    pthread_cond_broadcast(&cv_disc);
    pthread_mutex_unlock(&mtx);
    return NULL;
}

void *thread_compute_sqrt(void *arg) {
    pthread_mutex_lock(&mtx);
    while (!ready_disc) pthread_cond_wait(&cv_disc, &mtx);
    if (discriminant >= 0) sqrt_disc = sqrt(discriminant);
    ready_sqrt = true;
    pthread_cond_broadcast(&cv_sqrt);
    pthread_mutex_unlock(&mtx);
    return NULL;
```

```
}

void *thread_compute_root1(void *arg) {
    pthread_mutex_lock(&mtx);
    while (!ready_sqrt) pthread_cond_wait(&cv_sqrt, &mtx);
    if (discriminant >= 0) root1 = (-b + sqrt_disc) / (2.0 * a);
    pthread_mutex_unlock(&mtx);
    return NULL;
}

void *thread_compute_root2(void *arg) {
    pthread_mutex_lock(&mtx);
    while (!ready_sqrt) pthread_cond_wait(&cv_sqrt, &mtx);
    if (discriminant >= 0) root2 = (-b - sqrt_disc) / (2.0 * a);
    pthread_mutex_unlock(&mtx);
    return NULL;
```

使用 6 个线程计算中间值与两个根

```
    pthread_t t1, t2, t3, t4, t5, t6;
    pthread_create(&t1, NULL, thread_compute_b2, NULL);
    pthread_create(&t2, NULL, thread_compute_4ac, NULL);
    pthread_create(&t3, NULL, thread_compute_disc, NULL);
    pthread_create(&t4, NULL, thread_compute_sqrt, NULL);
    pthread_create(&t5, NULL, thread_compute_root1, NULL);
    pthread_create(&t6, NULL, thread_compute_root2, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    pthread_join(t3, NULL);
    pthread_join(t4, NULL);
    pthread_join(t5, NULL);
    pthread_join(t6, NULL);

    gettimeofday(&t_end, NULL);
    double elapsed = (t_end.tv_sec - t_start.tv_sec)*1e3 + (t_end.tv_usec
        - t_start.tv_usec)*1e-3;
```

2.2 蒙特卡洛方法求 π 的近似值

线程结构体

```
typedef struct {
```

```
    long long points;       // 本线程要生成的随机点数
    unsigned int seed;      // rand_r 的种子
    long long in_circle;    // 本线程统计的落在圆内的点数
} thread_arg_t;
```

线程函数：生成随机点并统计落在单位圆内的个数

```
void* thread_func(void *arg) {
    thread_arg_t *t = (thread_arg_t*)arg;
    long long in_cnt = 0;
    for (long long i = 0; i < t->points; ++i) {
        double x = (double)rand_r(&t->seed) / RAND_MAX;
        double y = (double)rand_r(&t->seed) / RAND_MAX;
        if (x*x + y*y <= 1.0) {
            ++in_cnt;
        }
    }
    t->in_circle = in_cnt;
    return NULL;
}
```

与之前的实验类似，将 n 均分至各线程

```
    for (int ni = 0; ni < num_n; ++ni) {
        long long n = n_values[ni];
        for (int t = 0; t < num_options; ++t) {
            int num_threads = thread_options[t];

            // 动态分配线程句柄与参数数组
            pthread_t    *threads = malloc(sizeof(pthread_t) *
               num_threads);
            thread_arg_t *args    = calloc(num_threads, sizeof(
               thread_arg_t));

            // 将 n 均分到各线程，最后一个线程分配余数
            long long base = n / num_threads;
            long long rem  = n % num_threads;
            for (int i = 0; i < num_threads; ++i) {
                args[i].points    = base + (i == num_threads - 1 ? rem :
                   0);
                args[i].seed      = (unsigned int)time(NULL) ^ (i * 0
                   x9e3779b1);
                args[i].in_circle = 0;
            }
```

```
            // 记录开始时间
            struct timeval t_start, t_end;
            gettimeofday(&t_start, NULL);

            // 创建并启动线程
            for (int i = 0; i < num_threads; ++i) {
                pthread_create(&threads[i], NULL, thread_func, &args[i]);
            }
            // 等待线程结束并汇总各线程的统计结果
            long long total_in = 0;
            for (int i = 0; i < num_threads; ++i) {
                pthread_join(threads[i], NULL);
                total_in += args[i].in_circle;
            }

            // 记录结束时间
            gettimeofday(&t_end, NULL);
            double elapsed = (t_end.tv_sec - t_start.tv_sec)
                           + (t_end.tv_usec - t_start.tv_usec) / 1e6;

            // 估算
            double pi_est = 4.0 * (double)total_in / (double)n;

            // 输出一行结果
            printf("%8lld\t%7d\t%7lld\t%10.8f\t%8.6f\n",
                   n, num_threads, total_in, pi_est, elapsed);

            free(threads);
            free(args);
        }
    }
```

3 实验结果

3.1 一元二次方程求解

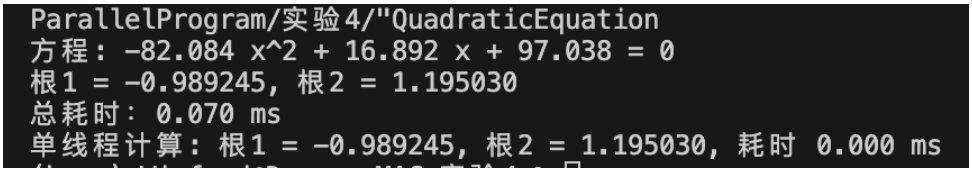

图 1: 一元二次方程实验结果

事实上对于单个方程，多线程的通信开销远大于计算开销，因此耗时较长

3.2 蒙特卡洛方法求 π 的近似值

表 1: Pthreads Monte Carlo 近似计算 π 的性能评估

n	Threads	m	$\hat{\pi}$	Time (s)
1000	1	780	3.12000000	0.000051
	2	762	3.04800000	0.000043
	4	783	3.13200000	0.000068
	8	786	3.14400000	0.000079
	16	803	3.21200000	0.000145
5000	1	3913	3.13040000	0.000106
	2	3897	3.11760000	0.000066
	4	3901	3.12080000	0.000076
	8	3908	3.12640000	0.000083
	16	3912	3.12960000	0.000138
10000	1	7825	3.13000000	0.000194
	2	7803	3.12120000	0.000154
	4	7791	3.11640000	0.000111
	8	7833	3.13320000	0.000109
	16	7848	3.13920000	0.000149
20000	1	15710	3.14200000	0.000347
	2	15704	3.14080000	0.000281
	4	15557	3.11140000	0.000189
	8	15644	3.12880000	0.000178
	16	15739	3.14780000	0.000184
50000	1	39354	3.14832000	0.000895
	2	39307	3.14456000	0.000433
	4	39095	3.12760000	0.000407
	8	39198	3.13584000	0.000374
	16	39209	3.13672000	0.000375

4 实验感想

在计算单个一元二次方程时，单线程肯定更快，实际生产中的常规做法是类似于其他实验，将 n 个一元二次方程平均分配给所有线程。