中山大学计算机院本科生实验报告

(2024学年秋季学期)

课程名称：高性能计算程序设计

实验	Pthread 共享内存编程	专业(方向)	信息与计算科学
学号	22336313	姓名	郑鸿鑫
Email	zhenghx57@mail2.sysu.edu.cn	完成日期	2024/10/21

1. 实验目的

本实验旨在通过实践操作，深入理解并掌握高性能计算程序设计中的并行计算技术。通过使用Pthreads库实现不同的并行计算任务，以学习如何有效地利用多线程来提高计算效率和处理大规模数据集。

2. 实验过程和核心代码

> **子任务0** 通过Pthread实现通用矩阵乘法
> 通过Pthreads实现通用矩阵乘法（Lab0）的并行版本，Pthreads并行线程从1增加至8，矩阵规模从512增加至2048.
> 问题描述：随机生成$M*N$和$N*K$的两个矩阵A,B,对这两个矩阵做乘法得到矩阵C.

编写代码如下(只展示关键代码)：

```c
// 线程参数结构体
typedef struct {
    int thread_id;
    int start_row;
    int end_row;
} thread_data_t;
// 线程执行函数
void *multiply_matrix(void *arg) {
    thread_data_t *data = (thread_data_t *)arg;
    int i, j, k;
    // 计算属于该线程的部分矩阵乘法
    for (i = data->start_row; i < data->end_row; i++) {
        for (j = 0; j < K; j++) {
            C[i][j] = 0;
            for (k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    pthread_exit(NULL);
}
//main函数关键代码
pthread_t threads[num_threads];
thread_data_t thread_data[num_threads];
int rows_per_thread = M / num_threads; // 每个线程负责的行数
int remaining_rows = M % num_threads; // 处理剩余的行
struct timeval start, end;
gettimeofday(&start, NULL); // 记录开始时间
// 创建线程并分配任务
for (int i = 0; i < num_threads; i++) {
    thread_data[i].thread_id = i;
    thread_data[i].start_row = i * rows_per_thread;
    if (i == num_threads - 1) {
        thread_data[i].end_row = (i + 1) * rows_per_thread + remaining_rows;
    } else {
        thread_data[i].end_row = (i + 1) * rows_per_thread;
    }
    pthread_create(&threads[i], NULL, multiply_matrix, (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);
}
gettimeofday(&end, NULL); // 记录结束时间
// 计算时间差（单位：微秒）
double elapsed_time = (end.tv_sec - start.tv_sec) * 1000.0; // 秒转毫秒
elapsed_time += (end.tv_usec - start.tv_usec) / 1000.0; // 微秒转毫秒
// 打印计算时间
printf("Size: %d, nums of thread: %d , Time taken for matrix multiplication: %lf ms\n",size,num_th
```

说明：线程函数实现了让该线程计算对应矩阵乘法的部分计算工作，其中所需的参数由结构体thread_data_t给出，包括对应部分的起始行和对应部分结束行。

> **子任务1** 使用多个线程对数组$a[1000]$求和的简单程序，演示Pthreads的用法。创建n个线程，每个线程通过共享变量global_index获取a数组的下一个未加元素

编写代码如下(只展示关键代码)：

```
// 线程函数
void* sum_array(void* arg) {
    int sum = 0;
    int i;
    for (i = 0; i < ARRAY_SIZE / num_threads; i++) {
        // 获取互斥锁
        pthread_mutex_lock(&lock);
        // 检查是否还有元素需要处理
        if (global_index < ARRAY_SIZE) {
            int index = global_index++;
            // 释放互斥锁
            pthread_mutex_unlock(&lock);

            sum += a[index];
        } else {
            // 释放互斥锁
            pthread_mutex_unlock(&lock);
            break;
        }
    }
    // 对共享变量加锁，以更新总和
    pthread_mutex_lock(&lock);
    global_sum += sum;
    pthread_mutex_unlock(&lock);
    return NULL;
}
//main函数关键代码
pthread_t threads[num_threads];
int result;
// 初始化互斥锁
pthread_mutex_init(&lock, NULL);
// 初始化数组
for (int i = 0; i < ARRAY_SIZE; i++) {
    a[i] = i;
}
// 创建线程
for (int i = 0; i < num_threads; i++) {
    result = pthread_create(&threads[i], NULL, sum_array, NULL);
    if (result) {
```

```
            printf("Error creating thread\n");
            exit(-1);
        }
    }
    // 等待线程结束
    for (int i = 0; i < num_threads; i++) {
        pthread_join(threads[i], NULL);
    }
    // 输出结果
    printf("Sum: %d\n", global_sum);
    // 销毁互斥锁
    pthread_mutex_destroy(&lock);
    return 0;
```

首先定义了线程函数以计算局部求和，然后在将其加入到全局变量时需要通过互斥锁来保证对临界区的操作是原始的，所以main函数中定义了一个互斥锁变量来保证不会同时对global_sum进行修改，线程函数会在每一次需要修改全局变量时解锁，在每次修改完成后上锁。

> **子任务2** 编写一个多线程程序来求解二次方程组$ax2+bx+c=0$的根，使用下面的公式，
>
> $$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$
>
> 中间值被不同的线程计算，使用条件变量来识别何时所有的线程都完成了计算

编写代码如下(只展示关键代码)：

```c
// 线程1：计算判别式
void *calculate_discriminant(void *arg) {
    pthread_mutex_lock(&mutex);
    discriminant = b * b - 4 * a * c;
    pthread_mutex_unlock(&mutex);
    // 通知主线程此线程已完成
    pthread_mutex_lock(&mutex);
    threads_completed++;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}
// 线程2：计算根
void *calculate_roots(void *arg) {
    pthread_mutex_lock(&mutex);
    if (discriminant >= 0) {
        root1 = (-b + sqrt(discriminant)) / (2 * a);
        root2 = (-b - sqrt(discriminant)) / (2 * a);
    } else {
        root1 = root2 = 0; // 或者其他表示无实根的值
    }
    pthread_mutex_unlock(&mutex);
    // 通知主线程此线程已完成
    pthread_mutex_lock(&mutex);
    threads_completed++;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}
//main函数关键代码
// 输入一元二次方程的系数
printf("Enter coefficients a, b, and c (for equation ax^2 + bx + c = 0): ");
scanf("%lf %lf %lf", &a, &b, &c);
// 初始化互斥锁和条件变量
pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&cond, NULL);
pthread_t thread1, thread2;
// 创建线程计算判别式
pthread_create(&thread1, NULL, calculate_discriminant, NULL);
```

```
// 创建线程计算根
pthread_create(&thread2, NULL, calculate_roots, NULL);
// 等待两个线程完成
pthread_mutex_lock(&mutex);
while (threads_completed < 2) {
    pthread_cond_wait(&cond, &mutex);
}
pthread_mutex_unlock(&mutex);
// 输出结果
if (discriminant >= 0) {
    printf("Root 1: %.2lf\n", root1);
    printf("Root 2: %.2lf\n", root2);
} else {
    printf("No real roots, discriminant is negative.\n");
}
// 销毁互斥锁和条件变量
pthread_mutex_destroy(&mutex);
pthread_cond_destroy(&cond);
```

说明：我们通过两个线程的并行计算来完成求根的过程，其中线程1用与计算判别式，线程2用于计算根。我们设置了互斥锁来保护共享变量，并且设置了一个计数器来计算已经完成的线程数量。

> **子任务3** 编写一个多线程程序实现蒙特卡洛方法，估算$y = x^2$曲线与x轴之间区域的面积，其中x的范围为$[0, 1]$。

编写代码如下(只展示关键代码)：

```c
// 线程函数
void *monte_carlo_thread(void *arg) {
    int thread_id = *((int *)arg);
    int points_inside = 0;
    int points_generated = 0;
    for (int i = 0; i < NUM_POINTS / NUM_THREADS; i++) {
        double x = rand() / (double)RAND_MAX;
        double y = rand() / (double)RAND_MAX;
        //printf("(%.2f,%.2f)\n",x,y);
        if (y <= x * x) {
            points_inside++;
        }
        points_generated++;
    }
    // 锁定互斥锁以更新全局计数器
    pthread_mutex_lock(&mutex);
    points_inside_curve += points_inside;
    total_points += points_generated;
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}
//main函数
int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];
    srand((unsigned int)time(NULL)); // 初始化随机数生成器
    // 初始化互斥锁
    pthread_mutex_init(&mutex, NULL);
    // 创建线程
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, monte_carlo_thread, (void *)&thread_ids[i]);
    }
    // 等待所有线程完成
    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    // 计算面积
    double area = (double)points_inside_curve / total_points;
```

```
    // 输出结果
    printf("Estimated area under the curve y = x^2 from 0 to 1: %lf\n", area);
    // 销毁互斥锁
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

说明：我们编写线程函数为生成[0,1]x[0,1]上的随机点，每个线程完成自己部分的蒙特卡洛模拟后，利用互斥锁来访访问全局变量points_inside,total_points以保证更新操作的原子性，然后等所有线程模拟完毕后，主线程将所有在$y = x^2$以下的点的数目除以总的随机坐标数，作为积分的估计值。

3. 实验结果

> **子任务0**（详细结果放在压缩包中的Result文件夹）

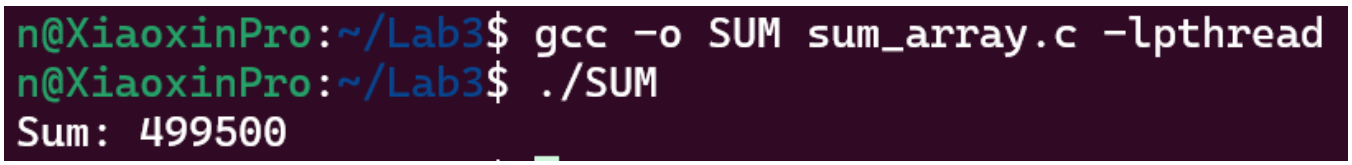
下面给出整理后的数据表格：

线程数 \ 矩阵规模	512	1024	2048
1	500ms	14874ms	176884ms
2	294ms	5969ms	118964ms
4	174ms	3871ms	65328ms
8	132ms	2803ms	42205ms

可以看到在增加并行的线程数时，程序计算的时间会明显变少，随着矩阵规模的增大，计算时间会增大，符合实验预期。

> **子任务1**

如下图所示：



说明：数组的元素初始化为下标，即$a[i] = i$，所以求和的结果为0到999的加和，最终结果应

该是499500，可以看到结果符合预期。

子任务2

我们给出一个简单的例子$2x^2 + 3x + 1 = 0$,这个方程的解为$x_1 = -0.5$，$x_2 = -1$，我们编译运行程序，检查结果：
结果如下图所示:

```
n@XiaoxinPro:~/Lab3$ gcc -o ES equation_solution.c -lpthread -lm
n@XiaoxinPro:~/Lab3$ ./ES
Enter coefficients a, b, and c (for equation ax^2 + bx + c = 0): 2 3 1
Root 1: -0.50
Root 2: -1.00
```

可以看到实验结果符合预期，说明通过互斥锁和条件变量可以保证线程之间的同步和对共享变量的原子操作。

子任务3

我们知道对于$y = x^2$在区间$[0, 1]$上的积分

$$\int_0^1 x^2 \, dx$$

可以容易计算得出值为$\frac{1}{3}$，我们运行编写好的多线程函数，使用蒙特卡洛方法来用概率近似这个积分的值，得到结果如下：

```
n@XiaoxinPro:~/Lab3$ gcc -o MC monte_carlo.c -lpthread -lm
n@XiaoxinPro:~/Lab3$ ./MC
Estimated area under the curve y = x^2 from 0 to 1: 0.333068
```

0.333068已经非常接近准确值，有三位有效数字，这是我们在8个线程总共取1000000个点的情况下，于是我们增加随机生成的坐标数量到10000000，编译运行查看结果：

```
n@XiaoxinPro:~/Lab3$ gcc -o MC monte_carlo.c -lpthread -lm
n@XiaoxinPro:~/Lab3$ ./MC
Estimated area under the curve y = x^2 from 0 to 1: 0.333323
```

有效数字达到了4位，我们继续增加总坐标数，并让最终积分值的显示位数增加，结果如下：

```
n@XiaoxinPro:~/Lab3$ gcc -o MC monte_carlo.c -lpthread -lm
n@XiaoxinPro:~/Lab3$ ./MC
Estimated area under the curve y = x^2 from 0 to 1: 0.333332440000
```

在50000000次蒙特卡洛模拟的情况下，可以达到5位的有效数字，而且运行的速度非常迅速，这得益于并行计算的高性能，如果是串行程序，则可能需要在精确性和运行性能方面做出平衡考虑。

4. 实验感想

经过完成这次关于Pthreads的实验，我获得了宝贵的并行编程经验。通过实现通用矩阵乘法、数组求和以及求解二次方程组的根等任务，我深入理解了并行计算的基本概念和实际应用。这些实验不仅加深了我对Pthreads库的理解，还让我体会到了并行化对于提高计算性能的重要性。
在实验中，我学习了如何使用Pthreads来创建和管理线程，以及如何利用互斥锁和条件变量来同步线程。我认识到，尽管并行程序可以显著提高处理速度，但它们也带来了额外的复杂性。例如，线程间的同步和数据共享需要仔细设计，以避免竞争条件和死锁。
此外，我也体会到了串行程序与并行程序之间的差异。串行程序通常更简单、易于理解，但在处理大规模数据集时可能会遇到性能瓶颈。相比之下，虽然并行程序在设计和实现上更具挑战性，但它们能够更有效地利用多核处理器的计算资源，从而在实际应用中提供更快的处理速度,可以兼顾计算的精确度和计算速率。