

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

批改人：

实验	多线程程序设计	专业（方向）	信息与计算科学
学号	22336049	姓名	陳日康
Email	chenih5@mail2.sysu.edu.cn	完成日期	2024 年 10 月 30 日

1. 实验目的

0. 通过 Pthreads 实现通用矩阵乘法
1. 基于 Pthreads 的数组求和
2. Pthreads 求解二次方程组的根
3. 编写一个多线程程序实现 Monte-carlo 方法

2. 实验过程和核心代码

(0). 通过 Pthreads 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。

(i). 线程函数 matrix_multiplication

```
void *matrix_multiplication(void *thread_id) {
    intptr_t tid = (intptr_t)thread_id;
    int start_row, end_row;
    int base_rows = rowsA / num_threads;
    int extra_rows = rowsA % num_threads;
    int assigned_rows = 0;

    if (tid < extra_rows) {
        assigned_rows = base_rows + 1;
        start_row = tid * assigned_rows;
    }
    else {
        assigned_rows = base_rows;
        start_row = tid * assigned_rows + extra_rows;
    }
    end_row = start_row + assigned_rows;

    for (int i = start_row; i < end_row; i++) {
        for (int j = 0; j < colsB; j++) {
            matC[i][j] = 0.0;
            for (int k = 0; k < colsA; k++) {
                matC[i][j] += matA[i][k] * matB[k][j];
            }
        }
    }
    return NULL;
}
```

每个线程负责计算 matC 中一部分行。通过计算 start_row 和 end_row 来确定每个线程负责的行数。每个线程根据自己的 thread_id 计算分配的起始行和结束行，进行矩阵乘法。

(iii). 创建线程与等待完成

```
for (intptr_t t = 0; t < num_threads; t++) {  
    pthread_create(&threads[t], NULL, matrix_multiplication, (void *)t);  
}  
for (int t = 0; t < num_threads; t++) {  
    pthread_join(threads[t], NULL);  
}
```

使用 `pthread_create` 启动多个线程，每个线程执行 `matrix_multiplication` 函数，计算部分结果。
使用 `pthread_join` 等待所有线程完成。

(1). 基于 Pthreads 的数组求和

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，注意不能在临界区（critical section）外访问共享变量 `global_index`，避免出现 `race condition`。重写上面的例子，使得各线程可以一次最多提取 10 个连续的数组元素，取数组元素策略可以自己定义，可以是随机读取【1-10】个连续的元素，也可以是固定数量的元素，并进行累加求和，从而减少对下标的访问。

(i). 全局变量

```
int a[ARRAY_SIZE];  
int global_index = 0;  
int total_sum = 0;  
pthread_mutex_t mutex;
```

`a[ARRAY_SIZE]` 定义了一个大小为 1000 的全局数组。`global_index` 记录当前已处理的数组下标。`total_sum` 记录所有线程累加的结果。`Mutex` 是互斥锁，用来保证线程在修改共享资源时的同步性。

(ii). `sum_array` 函数

```
void* sum_array(void* threadid) {  
    int local_sum = 0;  
    while (1) {  
        int start_index;  
        int count = 0;  
  
        pthread_mutex_lock(&mutex);  
        start_index = global_index;  
        if (global_index < ARRAY_SIZE) {  
            count = (ARRAY_SIZE - global_index < MAX_ELEMENTS_PER_THREAD) ? (ARRAY_SIZE - global_index) : MAX_ELEMENTS_PER_THREAD;  
            global_index += count;  
        }  
        pthread_mutex_unlock(&mutex);  
  
        if (count == 0) {  
            break;  
        }  
  
        for (int i = 0; i < count; i++) {  
            local_sum += a[start_index + i];  
        }  
    }  
  
    pthread_mutex_lock(&mutex);  
    total_sum += local_sum;  
    pthread_mutex_unlock(&mutex);  
  
    pthread_exit(NULL);  
}
```

`local_sum`: 线程内的局部变量，用于保存当前线程的部分和。`pthread_mutex_lock(&mutex)`和
`pthread_mutex_unlock(&mutex)`: 用来在多个线程间同步访问共享资源（`global_index` 和
`total_sum`），防止数据竞争。

循环中，线程尝试从 `global_index` 开始处理 `MAX_ELEMENTS_PER_THREAD` 个元素，直到数组被
处理完。如果没有剩余元素，则退出循环。最后，局部求和的结果加到全局的 `total_sum` 中。

(iii). 主函数

```
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("Usage: %s <number_of_threads>\n", argv[0]);
        exit(-1);
    }

    int num_threads = atoi(argv[1]);
    if (num_threads <= 0) {
        printf("Number of threads should be greater than 0.\n");
        exit(-1);
    }

    pthread_t threads[num_threads];
    pthread_mutex_init(&mutex, NULL);

    // Initialize array with i + 1
    for (int i = 0; i < ARRAY_SIZE; i++) {
        a[i] = i + 1;
    }

    // Start timing
    clock_t start_time = clock();

    // Create threads
    for (long t = 0; t < num_threads; t++) {
        int rc = pthread_create(&threads[t], NULL, sum_array, (void*)t);
        if (rc) {
            printf("ERROR: return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }

    // Join threads
    for (int t = 0; t < num_threads; t++) {
        pthread_join(threads[t], NULL);
    }

    // Stop timing
    clock_t end_time = clock();
    double elapsed_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;

    printf("Total sum = %d\n", total_sum);
    printf("Execution time = %f seconds\n", elapsed_time);

    pthread_mutex_destroy(&mutex);
    pthread_exit(NULL);
}
```

初始化互斥锁。用 `i + 1` 初始化数组 `a`，使得数组包含 1 到 1000。使用 `clock` 记录程序开始和结
束时间。循环创建指定数量的线程，每个线程运行 `sum_array` 函数。主线程等待所有创建的线程结
束（`pthread_join`）。销毁互斥锁并输出总和和程序的执行时间。

(2). Pthreads 求解二次方程组的根

编写一个多线程程序来求解二次方程组 $ax^2+bx+c=0$ 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。

(i). 全局变量和条件变量

```
// 定义全局变量和条件变量
std::mutex mtx;
std::condition_variable cv;
int completedThreads = 0;
std::vector<double> roots(2, NAN);
```

`mtx` 定义了一个互斥锁，用于保护线程对共享数据的访问。`cv` 条件变量，用于线程之间的同步，确保主线程等待子线程完成计算。`completedThreads` 记录已完成计算的线程数量。`roots` 保存方程的两个根，初始化为 `NAN` 表示未计算。

(ii). 计算第一个根的函数 `calculateRoot1`

```
void calculateRoot1(double a, double b, double discriminant) {
{
    std::lock_guard<std::mutex> lock(mtx);
    if (discriminant > 0) {
        roots[0] = (-b + sqrt(discriminant)) / (2 * a);
    }
    else if (discriminant == 0) {
        roots[0] = -b / (2 * a);
    }
    completedThreads++;
    cv.notify_one();
}
}
```

使用 `std::lock_guard<std::mutex>` 对互斥锁 `mtx` 加锁，保证多线程访问共享资源（`roots` 和 `completedThreads`）时的安全性。根据判别式 `discriminant` 判断方程的根：`discriminant > 0`：计算第一个根；`discriminant == 0`：计算唯一的根。更新 `completedThreads` 以表示一个线程已完成，然后通过 `cv.notify_one()` 通知等待的线程。

(iii). 等待线程完成的函数 `waitForCompletion`

```
void waitForCompletion() {
    std::unique_lock<std::mutex> lock(mtx);
    cv.wait(lock, [] { return completedThreads == 2; });
}
```

使用 `std::unique_lock` 锁定互斥锁 `mtx`。调用 `cv.wait` 等待，直到 `completedThreads` 等于 2（两个线程都完成了计算）

(iv). 主函数

```
int main() {
    double a, b, c;
    std::cout << "Enter the coefficients of the equation: ";
    std::cin >> a >> b >> c;

    double discriminant = b * b - 4 * a * c;

    // 创建线程来计算两个根
    std::thread t1(calculateRoot1, a, b, discriminant);
    std::thread t2(calculateRoot2, a, b, discriminant);

    // 主线程等待所有计算线程完成
    waitForCompletion();

    // 输出结果
    if (discriminant > 0) {
        std::cout << "The two roots of the equation are: " << roots[0] << " and " << roots[1] << std::endl;
    }
    else if (discriminant == 0) {
        std::cout << "The equation has one real root: " << roots[0] << std::endl;
    }
    else {
        std::cout << "The equation has no real roots." << std::endl;
    }

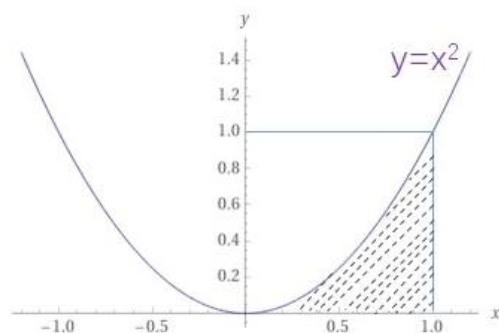
    // 线程回收
    t1.join();
    t2.join();

    return 0;
}
```

从用户获取方程的系数 a 、 b 、 c ，并计算判别式 discriminant 。创建两个线程 $t1$ 和 $t2$ ，分别调用 calculateRoot1 和 calculateRoot2 计算方程的根。调用 waitForCompletion ，主线程等待两个计算线程完成。根据 discriminant 的值输出结果： $\text{discriminant} > 0$ ：方程有两个不同的实根。 $\text{discriminant} == 0$ ：方程有一个实根。否则，方程没有实根。调用 $t1.\text{join}()$ 和 $t2.\text{join}()$ 回收线程，确保主线程在所有子线程结束后退出。

(3). 编写一个多线程程序实现 Monte-carlo 方法

参考课本 137 页 4.2 题和本次实验作业的补充材料。估算 $y=x^2$ 曲线与 x 轴之间区域的面积，其中 x 的范围为 $[0, 1]$ 。



(i).全局变量

```
std::mutex mtx;
long long totalPoints = 0;
long long insideCurvePoints = 0;
```

std::mutex mtx 互斥锁，用于保护多个线程对全局变量的安全访问。totalPoints 表示所有线程处理的总点数。insideCurvePoints 表示落在曲线 $y=x^2$ 下方的点数。

(ii). 线程函数 monteCarloEstimation

```
void monteCarloEstimation(int numPoints) {
    // 使用随机数生成器
    std::mt19937 rng(std::random_device{}());
    std::uniform_real_distribution<double> distribution(0.0, 1.0);

    long long localInsideCurvePoints = 0;

    for (int i = 0; i < numPoints; ++i) {
        double x = distribution(rng);
        double y = distribution(rng);

        // 检查点 (x, y) 是否在 y = x^2 曲线下方
        if (y <= x * x) {
            localInsideCurvePoints++;
        }

        // 线程安全地更新全局变量
        {
            std::lock_guard<std::mutex> lock(mtx);
            insideCurvePoints += localInsideCurvePoints;
            totalPoints += numPoints;
        }
    }
}
```

使用随机数生成器 std::mt19937 和 std::uniform_real_distribution 生成均匀分布的(x,y)坐标 localInsideCurvePoints 作为线程内的局部变量，用来计数在曲线下方的点数，避免频繁访问全局变量。循环生成指定数量的随机点(x, y)，并判断它们是否位于曲线 $y=x^2$ 下方：如果 $y \leq x*x$ ，表示点位于曲线下方。使用 std::lock_guard 锁定互斥锁 mtx，然后将当前线程的局部计数加到全局计数 insideCurvePoints 和 totalPoints 中。使用互斥锁确保多个线程同时更新全局变量时的线程安全。

(iii). 主函数

```
int main(int argc, char* argv[]) {
    if (argc != 3) {
        std::cerr << "Usage: " << argv[0] << " <number_of_threads> <points_per_thread>\n";
        return 1;
    }

    int numThreads = std::atoi(argv[1]);
    int pointsPerThread = std::atoi(argv[2]);

    if (numThreads <= 0 || pointsPerThread <= 0) {
        std::cerr << "Error: Number of threads and points per thread must be positive integers.\n";
        return 1;
    }
}
```

```

std::vector<std::thread> threads;

// 创建线程
for (int i = 0; i < numThreads; ++i) {
    threads.emplace_back(monteCarloEstimation, pointsPerThread);
}

// 等待所有线程完成
for (auto& t : threads) {
    t.join();
}

// 估算面积
double area = static_cast<double>(insideCurvePoints) / totalPoints;

// 输出结果
std::cout << "Estimated area: " << area << std::endl;

return 0;
}

```

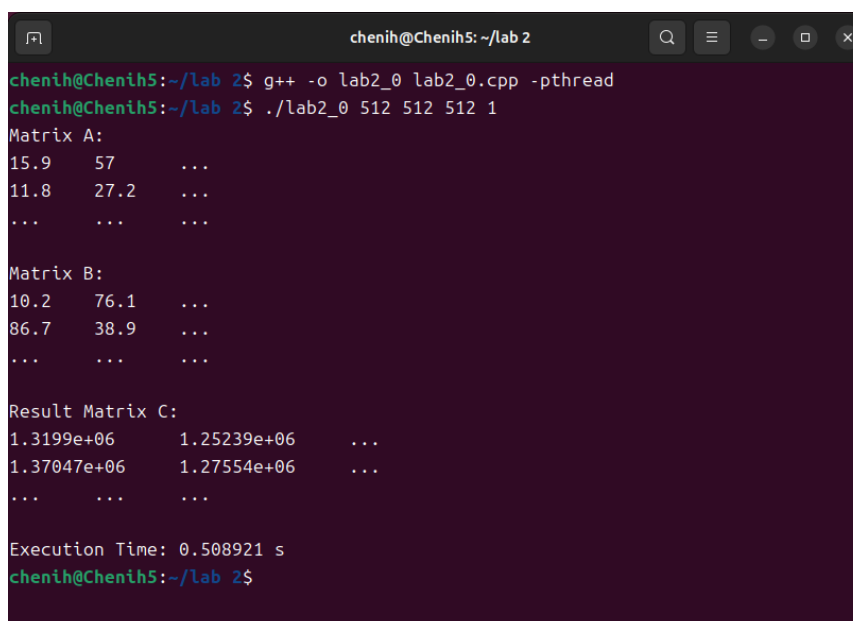
将命令行参数解析为线程数量 `numThreads` 和每个线程处理的点数 `pointsPerThread`，并检查它们是否为正数。使用 `std::vector<std::thread>` 创建线程容器，然后通过循环创建 `numThreads` 个线程，每个线程调用 `monteCarloEstimation` 函数处理 `pointsPerThread` 个点。循环调用 `join` 方法，等待所有线程完成，确保主线程在所有子线程完成计算后继续。计算估算面积：

`area=static_cast<double>(insideCurvePoints) / totalPoints`，表示在曲线下方的点的比例，这个比例就是所求的面积。

3. 实验结果

(0). 通过 Pthreads 实现通用矩阵乘法

打印运行结果：



```

chenih@Chenih5: ~/lab 2
chenih@Chenih5:~/lab 2$ g++ -o lab2_0 lab2_0.cpp -pthread
chenih@Chenih5:~/lab 2$ ./lab2_0 512 512 512 1
Matrix A:
15.9    57    ...
11.8    27.2  ...
...     ...   ...

Matrix B:
10.2    76.1  ...
86.7    38.9  ...
...     ...   ...

Result Matrix C:
1.3199e+06  1.25239e+06  ...
1.37047e+06  1.27554e+06  ...
...         ...         ...

Execution Time: 0.508921 s
chenih@Chenih5:~/lab 2$

```

为简化结果只打印首 2*2 矩阵，其他由省略号替代。由 `print_matrices` 控制是否打印矩阵结果。

512 阶:

```
chenih@Chenih5: ~/lab 2
chenih@Chenih5:~/lab 2$ g++ -o lab2_0 lab2_0.cpp -pthread
chenih@Chenih5:~/lab 2$ ./lab2_0 512 512 512 1
Execution Time: 0.502692 s
chenih@Chenih5:~/lab 2$ ./lab2_0 512 512 512 2
Execution Time: 0.253058 s
chenih@Chenih5:~/lab 2$ ./lab2_0 512 512 512 4
Execution Time: 0.385913 s
chenih@Chenih5:~/lab 2$ ./lab2_0 512 512 512 8
Execution Time: 0.285086 s
```

1024 阶:

```
chenih@Chenih5: ~/lab 2
chenih@Chenih5:~/lab 2$ g++ -o lab2_0 lab2_0.cpp -pthread
chenih@Chenih5:~/lab 2$ ./lab2_0 1024 1024 1024 1
Execution Time: 4.90099 s
chenih@Chenih5:~/lab 2$ ./lab2_0 1024 1024 1024 2
Execution Time: 4.28608 s
chenih@Chenih5:~/lab 2$ ./lab2_0 1024 1024 1024 4
Execution Time: 4.6226 s
chenih@Chenih5:~/lab 2$ ./lab2_0 1024 1024 1024 8
Execution Time: 3.32083 s
```

2048 阶:

```
chenih@Chenih5: ~/lab 2
chenih@Chenih5:~/lab 2$ g++ -o lab2_0 lab2_0.cpp -pthread
chenih@Chenih5:~/lab 2$ ./lab2_0 2048 2048 2048 1
Execution Time: 89.2257 s
chenih@Chenih5:~/lab 2$ ./lab2_0 2048 2048 2048 2
Execution Time: 93.9897 s
chenih@Chenih5:~/lab 2$ ./lab2_0 2048 2048 2048 4
Execution Time: 73.0085 s
chenih@Chenih5:~/lab 2$ ./lab2_0 2048 2048 2048 8
Execution Time: 77.6873 s
```

(1). 基于 Pthreads 的数组求和

```
chenih@Chenih5: ~/lab 2
chenih@Chenih5:~/lab 2$ g++ -o lab2_1 lab2_1.cpp -pthread
chenih@Chenih5:~/lab 2$ ./lab2_1 1
Total sum = 500500
Execution time = 0.000830 seconds
chenih@Chenih5:~/lab 2$ ./lab2_1 2
Total sum = 500500
Execution time = 0.001380 seconds
chenih@Chenih5:~/lab 2$ ./lab2_1 4
Total sum = 500500
Execution time = 0.001670 seconds
chenih@Chenih5:~/lab 2$ ./lab2_1 8
Total sum = 500500
Execution time = 0.005435 seconds
chenih@Chenih5:~/lab 2$
```

可见程序正常运行而且结果正确。

(2). Pthreads 求解二次方程组的根

```
chenih@Chenih5: ~/lab 2
chenih@Chenih5:~/lab 2$ g++ -o lab2_2 lab2_2.cpp -pthread
chenih@Chenih5:~/lab 2$ ./lab2_2 2
Enter the coefficients of the equation: 2 -3 1
The two roots of the equation are: 1 and 0.5
chenih@Chenih5:~/lab 2$ ./lab2_2 4
Enter the coefficients of the equation: 1 -2 1
The equation has one real root: 1
chenih@Chenih5:~/lab 2$ ./lab2_2 8
Enter the coefficients of the equation: 4 2 3
The equation has no real roots.
chenih@Chenih5:~/lab 2$
```

运行程序，可以成功执行。

(3). 编写一个多线程程序实现 Monte-carlo 方法

```
chenih@Chenih5: ~/lab 2
chenih@Chenih5:~/lab 2$ g++ -o lab2_3 lab2_3.cpp -pthread
chenih@Chenih5:~/lab 2$ ./lab2_3 2 10000
Estimated area: 0.33245
chenih@Chenih5:~/lab 2$ ./lab2_3 4 100000
Estimated area: 0.332863
chenih@Chenih5:~/lab 2$ ./lab2_3 8 1000000
Estimated area: 0.333738
chenih@Chenih5:~/lab 2$
```

可见运行结果符合预期，当 x 的范围为 $[0,1]$ 时，曲线 $y=x^2$ 与 x 轴之间区域的面积为 $\frac{1}{3}$ 。

4. 实验感想

在本次实验中，我通过使用 Pthreads 库解决各种多线程编程的应用场景，这次任务让我对多线程的设计与实现有了更全面的理解。在 Pthreads 实现矩阵乘法中我学会了如何将较大的计算任务分解成多个线程并行执行，从而提高计算效率。数组求和任务进一步加深了我对多线程同步机制的认识，特别是在共享资源访问时使用互斥锁来避免数据竞争。在解决二次方程组任务中，我学会了如何利用条件变量来同步线程，从而确保各个线程计算的协调性与正确性。最后通过实现 Monte-Carlo 方法，我体验了如何在统计模拟中使用多线程来提升计算速度。总体来说，这次实验我不仅掌握了 Pthreads 的基础操作，更让我深刻体会到多线程编程在提升程序效率方面的优势，同时也学习到在并行环境下，如何有效地解决同步与共享数据的问题是多线程编程成功的关键。