高性能计算程序设计基础 Lab2

实验	Pthreads并行	专业	计算机科学与技术
学号	21311525	姓名	马梓培
Email	mazipei21@gmail.com	完成日期	2024/10/18

1. Pthreads实现矩阵乘法

代码见附件或github.

首先，先进行一些初始化：

```
long thread; pthread_t* thread_handles;

// 初始化互斥锁和条件变量
pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&cond_var, NULL);

A = (float *)malloc(sizeof(float) * N * N);
B = (float *)malloc(sizeof(float) * N * N);
C = (float *)malloc(sizeof(float) * N * N);

initialize_matrix(A, N, 0., 10.);
initialize_matrix(B, N, 0., 10.);

thread_cnt = strtol(argv[1], NULL, 10);
thread_handles = malloc(thread_cnt * sizeof(pthread_t));
avg_rows = N / thread_cnt;
```

我通过将矩阵A按行分块，矩阵B共享来计算矩阵C。
我们知道Pthreads采用的是共享内存的方式来实现。上述的矩阵乘法，并不会有临界区的情况。每个fork的线程都各自处理各自的数据，没有同时访问一块内存的情况。因此，可以直接fork出thread_cnt个线程，每个线程执行各自对应的矩阵乘法。

1.1 pthread_create

```
for (thread = 0; thread < thread_cnt; thread ++) {
   pthread_create(&thread_handles[thread], NULL, matrix_mul, (void *) thread);
}
```

结合pthread_create()定义解释：

```
int pthread_create(
    pthread_t*                thread_p                /* out */,
    const pthread_attr_t*     attr_p                  /* in  */,
    void*                     (*start_routine)(void*) /* in  */,
    void*                     arg_p                   /* in  */);
```

首先，写一个for循环，从0遍历到thread_cnt-1，总共fork出thread_cnt个线程。pthread_create()第一个参数是一个指针，指向我们初始化已经分配好了的pthread_t对象，通过for循环的thread来索引。第二个参数不用，用NULL表示。接下来，*start_routine函数指针，我们传入matrix_mul，表示要并行的函数。而最后一个参数，为每一个线程赋予了唯一的int型参数rank，表示线程的编号。

1.2 pthread_joint

同样，既然有fork线程，我们就需要合并线程。

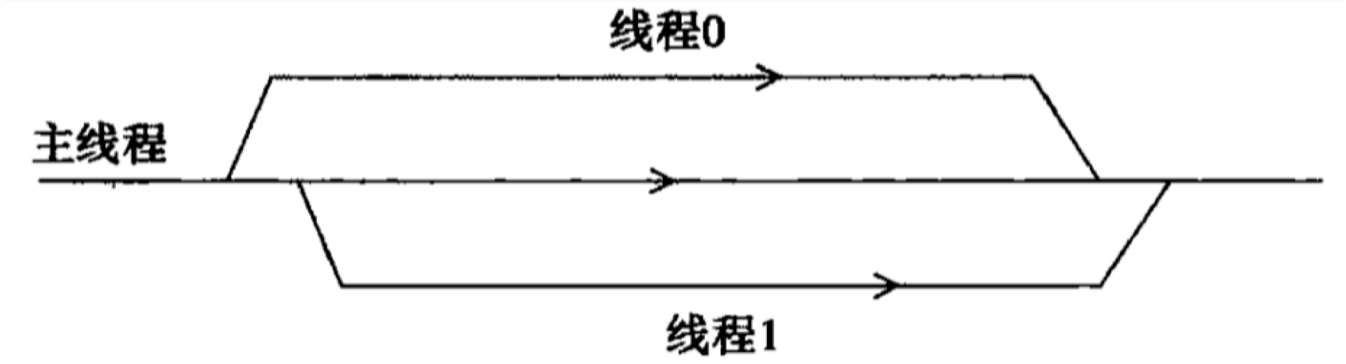


图 4-2　主线程派生与合并两个线程

```
for (thread = 0; thread < thread_cnt; thread ++) {
    pthread_join(thread_handles[thread], NULL);
}
```

结合pthread_join()定义解释：

```
int pthread_join(
    pthread_t       thread        /* in  */,
    void**          ret_val_p     /* out */);
```

首先，写一个for循环，从0遍历到thread_cnt - 1，总共fork出thread_cnt个线程。pthread_create()第一个参数表明，我们要合并哪个线程。第二个参数表示接受的返回值，我们不需要返回值，写上NULL。

1.3 运行计时

首先，我们需要设置一个barrier让所有进程在同一起跑线，copy自书上：

```
// Barrier
pthread_mutex_lock(&mutex);
counter++;
if (counter == thread_cnt) {
   counter = 0;
   pthread_cond_broadcast(&cond_var);
}
else {
   while (pthread_cond_wait(&cond_var, &mutex) != 0); }
pthread_mutex_unlock(&mutex);
```

计时，start_time 和 end_time 夹住矩阵相乘：

```
clock_gettime(CLOCK_MONOTONIC, &start_time); for (int i = start_rows; i < end_rows;
++i) {    for (int j = 0; j < N; ++j) {
       float sum = 0;
       for (int x = 0; x < N; ++x) {
           sum += *(A + i * N + x) * *(B + x * N + j);        }
       *(C + i * N + j) = sum;
   }
}
clock_gettime(CLOCK_MONOTONIC, &end_time);
```

在同步时间时，需要有互斥锁，因为访问的是同一个共享变量：

```
// 更新全局最大时间
pthread_mutex_lock(&max_time_mutex); // 锁定互斥锁以安全更新
if (elapsed_time > max_elapsed_time) {
   max_elapsed_time = elapsed_time; // 更新最大时间
}
pthread_mutex_unlock(&max_time_mutex); // 解锁互斥锁
```

1.4 通用矩阵乘法实验结果

N = 128

```
❯ ./mm.out 1
Max elapsed time among all threads: 0.013769 seconds.
❯ ./mm.out 2
Max elapsed time among all threads: 0.007233 seconds.
❯ ./mm.out 4
Max elapsed time among all threads: 0.003861 seconds.
❯ ./mm.out 8
Max elapsed time among all threads: 0.001570 seconds.
❯ ./mm.out 16
Max elapsed time among all threads: 0.001096 seconds.
```

N = 256

```
❯ ./mm.out 1
Max elapsed time among all threads: 0.084312 seconds.
❯ ./mm.out 2
Max elapsed time among all threads: 0.050017 seconds.
❯ ./mm.out 4
Max elapsed time among all threads: 0.025685 seconds.
❯ ./mm.out 8
Max elapsed time among all threads: 0.016018 seconds.
❯ ./mm.out 16
Max elapsed time among all threads: 0.008387 seconds.
```

N = 512

```
❯ ./mm.out 1
Max elapsed time among all threads: 0.481108 seconds.
❯ ./mm.out 2
Max elapsed time among all threads: 0.260021 seconds.
❯ ./mm.out 4
Max elapsed time among all threads: 0.146547 seconds.
❯ ./mm.out 8
Max elapsed time among all threads: 0.089492 seconds.
❯ ./mm.out 16
Max elapsed time among all threads: 0.092121 seconds.
```

N = 1024

```
❯ ./mm.out 1
Max elapsed time among all threads: 3.753396 seconds.
❯ ./mm.out 2
Max elapsed time among all threads: 1.961918 seconds.
❯ ./mm.out 4
Max elapsed time among all threads: 1.042281 seconds.
❯ ./mm.out 8
Max elapsed time among all threads: 0.903420 seconds.
❯ ./mm.out 16
Max elapsed time among all threads: 0.849642 seconds.
```

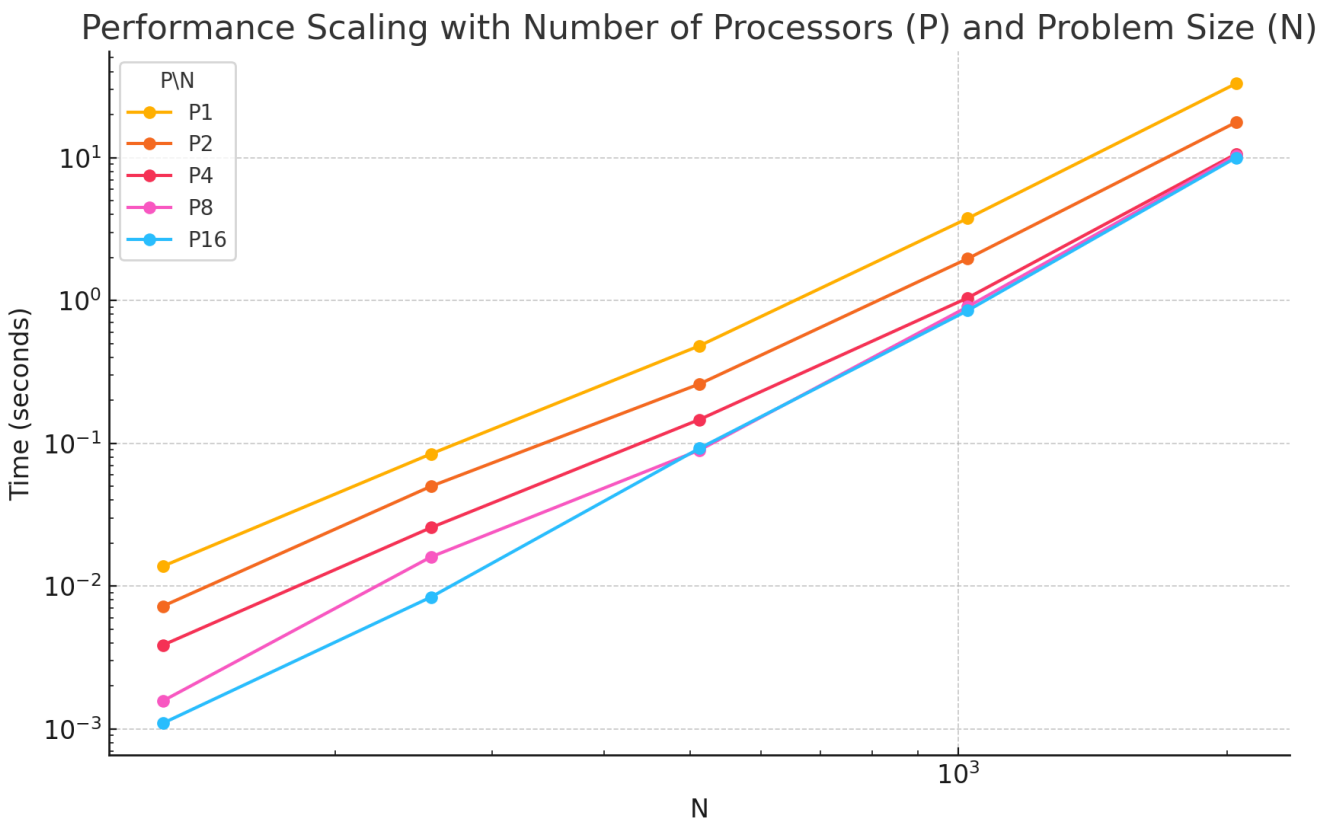
N = 2048

```
❯ ./mm.out 1
Max elapsed time among all threads: 32.983432 seconds.
❯ ./mm.out 2
Max elapsed time among all threads: 17.653807 seconds.
❯ ./mm.out 4
Max elapsed time among all threads: 10.603551 seconds.
❯ ./mm.out 8
Max elapsed time among all threads: 10.331040 seconds.
❯ ./mm.out 16
Max elapsed time among all threads: 9.993411 seconds.
```

绘制成表格

P\ N	128	256	512	1024	2048
1	0.013769	0.084312	0.481108	3.753396	32.983432
2	0.007233	0.050017	0.260021	1.961918	17.653807
4	0.003861	0.025685	0.146547	1.042281	10.603551
8	0.001570	0.016018	0.089492	0.903420	10.331040
16	0.001096	0.008387	0.092121	0.849642	9.993411

(1)



2. Pthread 实现数组求和

2.1 pthread_create和pthread_join

```
for (thread = 0; thread < thread_cnt; thread++) {
    threadData[thread].array = array;
    threadData[thread].start = thread * length_per_thread;
    threadData[thread].end = (thread + 1) * length_per_thread;
    pthread_create(&thread_handles[thread], NULL, sum_array,
(void*)&threadData[thread]);
}
    for (thread = 0; thread < thread_cnt; thread++) {
        pthread_join(thread_handles[thread], NULL);
    }
```

与矩阵乘法极为相似，不过多赘述。

2.2 array sum

```
void* sum_array(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    long long sum = 0;
    for (long long i = data->start; i < data->end; i++) {
        sum += data->array[i];
    }

    pthread_mutex_lock(&mutex);
    global_sum += sum;
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

首先，先将自己进程中的A[i]相加求和，得到局部和local_sum。然后降其加到共享变量中。然而，由于共享的缘故，我们必须互斥地访问改变量。因此，需要有一个互斥锁来保证互斥访问。

2.3 实验结果

N = 1M

```
❯ ./as.out 1
Total sum: 49483647 Cost time: 0.001771 seconds
❯ ./as.out 2
Total sum: 49483647 Cost time: 0.000794 seconds
❯ ./as.out 4
Total sum: 49483647 Cost time: 0.000377 seconds
❯ ./as.out 8
Total sum: 49483647 Cost time: 0.000599 seconds
❯ ./as.out 16
Total sum: 49483647 Cost time: 0.000664 seconds
```

N = 2M

```
❯ ./as.out 1
Total sum: 98971657 Cost time: 0.003424 seconds
❯ ./as.out 2
Total sum: 98971657 Cost time: 0.001792 seconds
❯ ./as.out 4
Total sum: 98971657 Cost time: 0.000795 seconds
❯ ./as.out 8
Total sum: 98971657 Cost time: 0.000985 seconds
❯ ./as.out 16
Total sum: 98971657 Cost time: 0.000816 seconds
```

N = 4M

```
❯ ./as.out 1
Total sum: 198025019 Cost time: 0.005634 seconds
❯ ./as.out 2
Total sum: 198025019 Cost time: 0.002740 seconds
❯ ./as.out 4
Total sum: 198025019 Cost time: 0.001475 seconds
❯ ./as.out 8
Total sum: 198025019 Cost time: 0.001373 seconds
❯ ./as.out 16
Total sum: 198025019 Cost time: 0.001206 seconds
```

N = 8M

```
❯ ./as.out 1
Total sum: 396047077 Cost time: 0.008436 seconds
❯ ./as.out 2
Total sum: 396047077 Cost time: 0.004492 seconds
❯ ./as.out 4
Total sum: 396047077 Cost time: 0.002281 seconds
❯ ./as.out 8
Total sum: 396047077 Cost time: 0.001943 seconds
❯ ./as.out 16
Total sum: 396047077 Cost time: 0.001930 seconds
```

N = 16M

```
❯ ./as.out 1
Total sum: 792011723 Cost time: 0.017006 seconds
❯ ./as.out 2
Total sum: 792011723 Cost time: 0.008234 seconds
❯ ./as.out 4
Total sum: 792011723 Cost time: 0.004939 seconds
❯ ./as.out 8
Total sum: 792011723 Cost time: 0.004415 seconds
❯ ./as.out 16
Total sum: 792011723 Cost time: 0.003793 seconds
```

N = 32M

```
❯ ./as.out 1
Total sum: 1584318480 Cost time: 0.035475 seconds
❯ ./as.out 2
Total sum: 1584318480 Cost time: 0.017141 seconds
❯ ./as.out 4
Total sum: 1584318480 Cost time: 0.009986 seconds
❯ ./as.out 8
Total sum: 1584318480 Cost time: 0.007589 seconds
❯ ./as.out 16
Total sum: 1584318480 Cost time: 0.007616 seconds
```

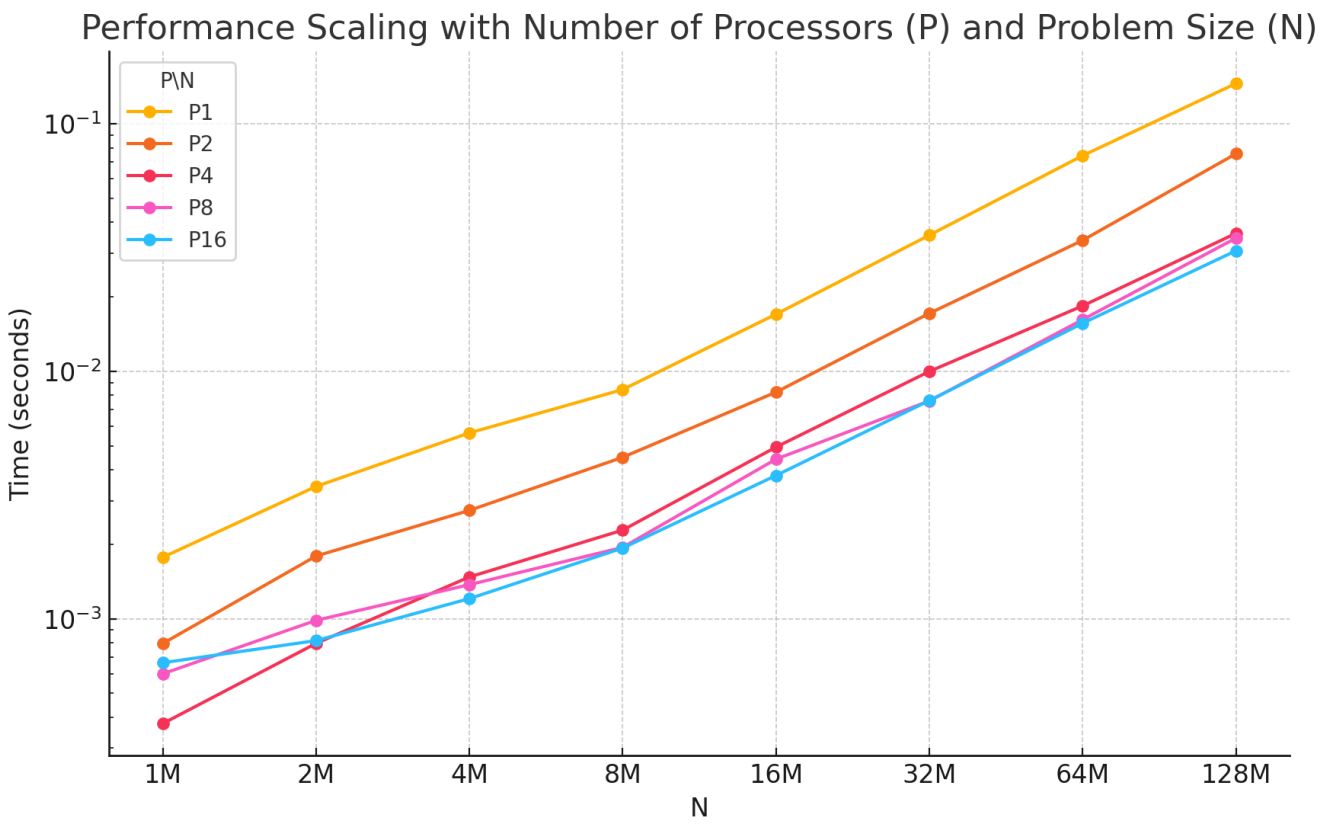
N = 64M

```
❯ ./as.out 1
Total sum: 3168123411 Cost time: 0.074379 seconds
❯ ./as.out 2
Total sum: 3168123411 Cost time: 0.033816 seconds
❯ ./as.out 4
Total sum: 3168123411 Cost time: 0.018382 seconds
❯ ./as.out 8
Total sum: 3168123411 Cost time: 0.016177 seconds
❯ ./as.out 16
Total sum: 3168123411 Cost time: 0.015637 seconds
```

N = 128M

```
❯ ./as.out 1 Total sum: 6336129725 Cost time: 0.145537 seconds
❯ ./as.out 2 Total sum: 6336129725 Cost time: 0.075687 seconds
❯ ./as.out 4 Total sum: 6336129725 Cost time: 0.036094 seconds
❯ ./as.out 8 Total sum: 6336129725 Cost time: 0.034524 seconds
❯ ./as.out 16 Total sum: 6336129725 Cost time: 0.030697 seconds
```

P\N	1M	2M	4M	8M	16M	32M	64M	128M
1	0.001771	0.003424	0.005634	0.008436	0.017006	0.035475	0.074379	0.145537
2	0.000794	0.001792	0.002740	0.004492	0.008234	0.017141	0.033816	0.075687
4	0.000377	0.000795	0.001475	0.002281	0.004939	0.009986	0.018382	0.036094
8	0.000599	0.000985	0.001373	0.001943	0.004415	0.007589	0.016177	0.034524
16	0.000664	0.000816	0.001206	0.001930	0.003793	0.007616	0.015637	0.030697



3. Pthreads求解二次方程组的根

3.1 初始化方程系数

用 rand()随机生成 0 到 100 的随机数来得到方程各个项的系数：

```
srand(static_cast<unsigned>(time(0)));
a = static_cast<double>(rand()) / (static_cast<double>(RAND_MAX / 201)) - 100;
b = static_cast<double>(rand()) / (static_cast<double>(RAND_MAX / 201)) - 100;
c = static_cast<double>(rand()) / (static_cast<double>(RAND_MAX / 201)) - 100;
```

3.2 创建线程执行任务

根据求解公式，一个方程需要通过四个线程来分别计算各个中间量。函数 pth_solveA 和 pth_solveB 分别计算 b*b 和 (-4)*a*c，且 A 与 B 之间没有依赖关系。在计算 b*b-4ac 之前（symbol＜2），需要确保 b*b 和 -4ac 已经计算完成，因为 C 依赖于 A 和 B 的结果。函数 pth_solveC 中，solveable 用来判断方程是否有解，当 C≥0 时，将 solveable 设为 1，表明方程存在实数解。pth_solveD 函数则负责计算 D，且它与 A、B、C 之间没有依赖。

```
// A = b * b
void *pth_solveA(void* rank) {
    A = b * b; pthread_mutex_lock(&lxmutex);
    symbol++; pthread_mutex_unlock(&lxmutex);
    return NULL;
```

```
}
// B = (-4) * a * c
void *pth_solveB(void* rank) {
    B = (-4) * a* c; pthread_mutex_lock(&lxmutex);
    symbol++; pthread_mutex_unlock(&lxmutex);
    return NULL;
}
// C = sqrt(C)
void *pth_solveC(void* rank) {
    while (symbol < 2) {
        C = A - B; if (C >= 0) {
            solveable = 1;
            C = sqrt(C);
        }
    }
    return NULL;
}

// D = 2 * a
void *pth_solveD(void* rank) {
    D = 2 * a;
    return NULL;
}
```

3.3 根据中间结果求解

根据 solveable，若等于 1，说明方程有解，按照公式计算即可。若不等于 1，则方程无解。代码如下，

```
if (solveable == 1) {
    double x1 = (-b + C) / D;
    double x2 = (-b - C) / D;
    printf("x1 = %lf, x2 = %lf", x1, x2);
}
else {
    printf("There is no solution");
}
```

4. Monte-carlo 积分

本报告将介绍如何使用 Pthread 多线程实现蒙特卡罗方法来求解曲线 $y = x^2$ 与x轴之间的区域面积，并讨论代码中的重要思路

4.1 初始化

在编写多线程蒙特卡罗积分时，我们首先需要进行一些必要的初始化，包括创建线程、初始化互斥锁等资源。以下是初始化的代码：

```
long thread;
pthread_t* thread_handles;

// 初始化互斥锁
pthread_mutex_init(&mutex, NULL);

// 分配内存用于线程和数据
thread_handles = malloc(thread_cnt * sizeof(pthread_t));
```

`pthread_mutex_init(&mutex, NULL)` 初始化一个互斥锁，用于在线程间同步关键数据的访问。
`pthread_t* thread_handles` 用于存储每个线程的句柄，thread_cnt 表示我们将创建的线程数量。
在此实现中，每个线程负责处理部分计算，并使用互斥锁来确保共享变量的安全访问。

4.2 线程创建与 pthread_create

为了并行执行蒙特卡罗模拟，每个线程需要处理一定数量的随机点。我们通过 pthread_create 函数来创建线程并启动并行执行。代码如下：

```
for (thread = 0; thread < thread_cnt; thread ++) {
   pthread_create(&thread_handles[thread], NULL, monte_carlo_thread, (void *)
thread);
}
```

`pthread_create` 函数用于创建线程。其参数分别为：
thread_p: 指向我们要创建的线程句柄的指针。
attr_p: 线程的属性，通常设为 NULL 表示使用默认设置。
start_routine: 线程执行的函数，这里是 monte_carlo_thread。
arg_p: 传递给线程函数的参数。我们传入了当前线程的索引 thread，用于标识每个线程。
每个线程将开始并行计算一定数量的随机点，检查它们是否落在曲线下方。

4.3 线程合并与 pthread_join

在创建并运行线程后，主程序需要等待所有线程执行完成。这是通过 `pthread_join` 实现的：

```
for (thread = 0; thread < thread_cnt; thread ++) {
    pthread_join(thread_handles[thread], NULL);
}
```

`pthread_join` 函数用于等待线程执行结束。第一个参数为要等待的线程句柄，第二个参数用于接收线程的返回值，在本例中设为 NULL 因为不需要返回值。

合并线程的过程确保所有并行计算完成后，主线程才能继续执行并计算最终的蒙特卡罗结果。

4.4 运行计时

在多线程程序中，为了衡量并行计算的效率，我们通常需要计时。这可以通过 clock_gettime 函数获取开始和结束时间，并计算每个线程的执行时间。

```
clock_gettime(CLOCK_MONOTONIC, &start_time);

for (int i = start_rows; i < end_rows; ++i) {
    for (int j = 0; j < N; ++j) {
        float sum = 0;
        for (int x = 0; x < N; ++x) {
            sum += *(A + i * N + x) * *(B + x * N + j);
        }
        *(C + i * N + j) = sum;
    }
}

clock_gettime(CLOCK_MONOTONIC, &end_time);
```

4.5 同步与互斥锁

当多个线程需要访问同一个共享变量（例如累积蒙特卡罗积分的结果或更新最大耗时）时，必须使用互斥锁来保护这些访问，避免数据竞争。

```
pthread_mutex_lock(&max_time_mutex); // 锁定互斥锁
if (elapsed_time > max_elapsed_time) {
   max_elapsed_time = elapsed_time; // 更新最大时间
}
pthread_mutex_unlock(&max_time_mutex); // 解锁互斥锁
```

通过使用多线程和蒙特卡罗方法，我们能够并行化求解曲线 $y = x^2$ 和 x 轴之间的面积的计算。Pthread 的互斥锁用于同步线程之间的共享数据访问。通过 pthread_create 和 pthread_join，我们能够创建并合并线程，并通过计时来评估程序的性能。

5. 实验感想

在完成高性能计算实验Lab2后，我对Pthreads的并行编程有了更深的理解。通过本次实验，我主要学习了如何使用Pthreads实现矩阵乘法、数组求和以及二次方程组的求解，并初步掌握了Monte Carlo积分的多线程实现。

首先，在矩阵乘法的实验中，我将矩阵A按行分块，矩阵B共享，这样每个线程独立处理自己的一部分数据，避免了线程间的资源竞争。这使得我理解了Pthreads共享内存的特点，并且在设计多线程程序时需要特别注意数据的访问冲突问题。通过pthread_create和pthread_join函数的使用，我成功地实现了并行化计算，并通过实验验证了多线程可以显著提升计算效率，尤其是在处理较大规模的矩阵时，线程数越多，运行时间缩短越明显。

在数组求和的实验中，我同样感受到了并行化的优势。在进行局部和与全局和的计算时，由于涉及到共享变量的更新，因此必须使用互斥锁来保证线程安全。这让我认识到，虽然并行化能提高计算速度，但同时也需要在设计时考虑线程同步和互斥锁的使用，以避免数据竞争的问题。

此外，在求解二次方程组时，我学会了如何将复杂的计算任务拆分成多个子任务，通过多线程并行执行各个中间量的计算。这让我理解了如何在不依赖的计算之间实现并行化，从而加快计算速度。

最后，使用Monte Carlo方法进行积分计算的实验让我进一步体会到并行计算的强大。通过创建多个线程处理大量的随机点，并使用互斥锁同步共享数据的访问，我成功地加速了积分结果的计算。这次实验不仅巩固了我对Pthreads并行编程的理解，也让我意识到在多线程编程中，合理的线程管理和同步机制至关重要。

总体来说，本次实验让我更深入地理解了Pthreads编程模型及其在高性能计算中的应用，并在实际操作中积累了宝贵的经验。