中山大学计算机院本科生实验报告

（2023 学年春季学期）

课程名称：并行程序设计　　**批改人：**

实验	3-Pthreads 并行矩阵乘法与数组求和	专业（方向）	计算机科学与技术
学号	21307082	姓名	赖耿桂
Email	laigg@mail2.sysu.edu.cn	完成日期	2024.4.11

1. 实验目的（200 字以内）

实验一：

使用 Pthreads 多线程实现并行矩阵乘法，分析不同线程数量（1-16）及不同矩阵规模（128-2048）的并行性能，并分析不同数据及任务划分方式的影响；

实验二：

使用 Pthreads 创建多线程，实现并行数组求和，分析不同线程数量（1-16）及数组规模（1M-128M）的并行性能及扩展性，并分析不同聚合方式的影响。

2. 实验过程和核心代码（600 字以内，图文并茂）

实验一（lab3_1.cpp）：

本实验需要额外导入的头文件如下：

```
1.  #include<pthread.h>      // 使用 pthread 库
2.  #include<chrono>         // 用来计时，不用 clock()是因为其计算的是处理器的时间，不是实际时间
```

与 MPI 实验不同的是，此次实验将 3 个矩阵、线程数、矩阵维度、每个线程参与计算的矩阵 A 的行数声明为全局变量：

```
1.  vector<double> A, B, C;     // 矩阵A、B、C
```

```
2.  int M, N, K, num, rows_per_thread;
```

由于使用 pthread_create 需要传递函数指针，故将矩阵乘法封装为一个函数 matrix_multiply：

```
1.  // 执行矩阵乘法
2.  void* matrix_multiply(void* id){
3.      int thread_id = *((int*)id);    // 获取当前线程号
4.      int begin_Row = thread_id * rows_per_thread, end_Row = (thread_id + 1) * rows_per_thread;
5.      // 根据线程号获取当前线程矩阵 A 需要参与运算的起始行数和终止行数
6.
7.      // 进行矩阵乘法
8.      for(int i = begin_Row; i < end_Row; ++i){
9.          for(int k = 0;k < K; ++k){
10.             double temp = 0;
11.             for(int j = 0; j < N; ++j){
12.                 temp += A[i*N + j] * B[j*K + k];
13.             }
14.             C[i*K + k] = temp;
15.         }
16.     }
17.     pthread_exit(NULL);     // 终止线程
18. }
```

在 main 函数中，依然使用命令行参数来获取矩阵维度和线程数：

```
1.  M = atoi(argv[1]);
2.  N = atoi(argv[2]);
3.  K = atoi(argv[3]);
4.  num = atoi(argv[4]);
```

将 num 赋值给一个常量 THREAD_NUM，并计算出每个线程需计算的 A 的行数：

```
1.  const int THREAD_NUM = num;
2.  rows_per_thread = M / THREAD_NUM;   // 计算每个线程中矩阵 A 有多少行要参与运算
```

矩阵的准备工作：

```
1.  // 初始化矩阵 A、B、C 并将 A、B 打印出来
2.  A.resize(M * N);
3.  B.resize(N * K);
4.  C.resize(M * K);
5.
6.  A = random_matrix_generator(M, N);
```

```
    B = random_matrix_generator(N, K);
    printf("Matrix A: \n");
    printMatrix(A, M, N);
    printf("Matrix B: \n");
    printMatrix(B, N, K);
```

接下来是使用 pthread 多线程并行矩阵乘法前的准备工作：

```
    pthread_t handles[THREAD_NUM];  // 为线程分配空间
    int thread_ids[THREAD_NUM];     // 线程号
```

下面是核心过程(需计算该过程的用时)，使用 pthread_create 创建并启动线程，然后使用 pthread_join 汇聚线程并释放资源：

```
    auto start_time = chrono::high_resolution_clock::now();     // 记录开始时间
    for(int i = 0; i < THREAD_NUM; ++i){
        thread_ids[i] = i;
        pthread_create(&handles[i], NULL, matrix_multiply, (void*)&thread_ids[i]);   // 创建并启动线程
    }

    for(int i = 0; i < THREAD_NUM; ++i){
        pthread_join(handles[i], NULL);    // 等待线程结束并回收线程资源
    }

    if(M % THREAD_NUM != 0){
    // 可能 M 未必能被 THREAD_NUM 整除，因此要计算剩余的(M % THREAD_NUM)行
        for(int i = THREAD_NUM * rows_per_thread; i < M; ++i){
            for(int k = 0; k < K; ++k){
                double temp = 0;
                for(int j = 0; j < N; ++j){
                    temp += A[i*N +j] * B[j*K + k];
                }
                C[i*K + k] = temp;
            }
        }
    }

    auto end_time = chrono::high_resolution_clock::now();   // 记录结束时间
    auto using_time = (double)(end_time - start_time).count() / 1e9;
        /*
            使用 chrono::high_resolution_clock::now()算出来的时间差(chrono::high_resolution_clock::duration<double>)是纳
            秒级别的，所以要除以 1e9 也即 10 的 9 次方才能转换成秒
        */
```

最后打印矩阵 C 和所用时间：

```
// 打印矩阵C
printf("Matrix C: \n");
printMatrix(C, M, K);

// 打印使用pthread 多线程实现并行矩阵乘法所用时间
printf("在%d 个线程并行的情况下计算大小为%d*%d 的矩阵 A 和大小为%d*%d 的矩阵 B 所用时间为: %lf s\n", THREAD_NUM,M,N,N,K,using_time);
```

选做（lab3_1_op.cpp）：

对 B 进行划分，即每个线程将矩阵 A 与 B 的几列进行矩阵乘法，思路与对 A 进行划分基本一致，以下仅展示修改部分：

声明一个表示每个线程要参与运算的 B 的列数的全局变量 cols_per_thread：

```
int cols_per_thread;
```

cols_per_thread 的计算如下：

```
cols_per_thread = K / THREAD_NUM;   // 计算每个线程中矩阵 B 有多少列要参与运算
```

修改矩阵乘法函数：

```
void* matrix_multiply(void* id){
    int thread_id = *((int*)id);    // 获取当前线程号
    int begin_Col = thread_id * cols_per_thread, end_Col = (thread_id + 1) * cols_per_thread;
    // 根据线程号获取当前线程矩阵 B 需要参与运算的起始列数和终止列数

    // 进行矩阵乘法
    for(int i = 0; i < M; ++i){
        for(int k = begin_Col;k < end_Col; ++k){
            double temp = 0;
            for(int j = 0; j < N; ++j){
                temp += A[i*N + j] * B[j*K + k];
            }
            C[i*K + k] = temp;
        }
    }
    pthread_exit(NULL);     // 终止线程
```

```
17.  }
```

如果 K 不能被 THREAD_NUM 整除，需要计算余下部分：

```
1.   if(K % THREAD_NUM != 0){
2.           // 可能 M 未必能被 THREAD_NUM 整除，因此要计算剩余的(M % THREAD_NUM)行
3.           for(int i = 0; i < M; ++i){
4.               for(int k = THREAD_NUM * cols_per_thread; k < K; ++k){
5.                   double temp = 0;
6.                   for(int j = 0; j < N; ++j){
7.                       temp += A[i*N +j] * B[j*K + k];
8.                   }
9.                   C[i*K + k] = temp;
10.              }
11.          }
12.      }
```

实验二（lab3_2.cpp）：

需要导入的头文件如下：

```
1.   #include<cstdio>
2.   #include<iostream>  // 接收 string 字符串
3.   #include<cstdlib>
4.   #include<chrono>    // 用来计时，不用 clock()是因为其计算的是处理器的时间，不是实际时间
5.   #include<vector>
6.   #include<cmath>     // 需要使用幂运算
7.   #include<random>    // 用于生成随机数
8.   #include<pthread.h>     // 使用 pthread 库
9.   #include<string>    // 用于截取子串和转换成双精度浮点数
```

定义一个表示一兆的常量，将**要求和的数组、数组的和、互斥锁**和**每个线程参与运算的元素数量**声明为全局变量：

```
1.   #define MEGA (long long)(pow(2, 20))    // 定义 1M(2^20)
2.
3.   vector<double> Array;   // 要求和的数组
4.   double global_sum = 0.0;    // 全局数组总和
5.   pthread_mutex_t Mutex;      // 互斥锁
6.   long long nums_per_thread;      // 每个线程参与计算的元素的数量
```

定义随机生成指定大小的数组的函数 random_array_generator：

```
1.   // 随机生成一个指定大小的数组
2.   vector<double> random_array_generator(long long num){
3.       size_t length = (size_t)num;
```

```
    vector<double> res(length);

     // 初始化随机数生成器
    random_device rd;
    default_random_engine eng(rd());
    uniform_real_distribution<double> distr(0,1); //取0-1的随机浮点数，防止算出来的和太大导致溢出

    for(size_t i = 0; i < length; ++i){
        res[i] = distr(eng);
    }
    return res;
}
```

定义每个线程进行求和的函数 calculate_sum(注意需要**使用互斥锁**来确保**将局部和加到全局和**这个过程准确无误)：

```
// 求和线程
void* calculate_sum(void* id){
    int thread_id = *((int*)id);    // 获取线程号
    long long begin = thread_id * nums_per_thread, end = (thread_id+1) * nums_per_thread;   // 求和起点和终点：[begin,end)
    double local_sum = 0.0;     // 局部和
    for(long long i = begin; i < end; ++i){
        local_sum += Array[i];
    }
    // 使用互斥锁保证将局部和加到全局和
    pthread_mutex_lock(&Mutex);     // 上锁
    global_sum += local_sum;
    pthread_mutex_unlock(&Mutex);   // 解锁
    pthread_exit(NULL);     // 终止线程
}
```

在 main 函数中，首先输入数组长度和线程数：

```
string len;   // 数组长度(未乘1M)
bool flag = false;  //判断是否有输入"M"，若有"M"，则要进行处理乘以1M(2^20)
int num;    // 线程数量
printf("请输入数组长度(数组长度为输入范围在[1M,128M]的整数可以输入具体的整数也可以输入如\"1M、1.25M\"的数)：\n");
cin>>len;
printf("请输入线程数量(线程数量为[1,16]的整数)：\n");
scanf("%d", &num);
const int THREAD_NUM = num;
```

对数组长度进行处理，若有 M 则要截取前面的数字部分并转为 double 类型后再乘以 Mega：

```
// 有"M"
if(len[len.length()-1] == 'M'){
    len.substr(0, len.length()-2);  // 提取前面的数字部分
```

```
        flag = true;    // 将flag改为true，后面需要再乘以1M(2^20)
    }
    double Len = stod(len);   // 将数字部分转换为数字
    if(flag){
        Len *= MEGA;
    }
    long long Length = (long long)Len;
```

计算出每个线程参与计算的数组元素数量：

```
    nums_per_thread = Length / THREAD_NUM;  // 计算出每个线程的要参与运算的元素数量
    Array = random_array_generator(Length);     // 随机生成指定大小的数组
```

使用 pthread 多线程并行数组求和前的准备工作：

```
    pthread_t handles[THREAD_NUM];  // 为线程分配空间
    int pthread_ids[THREAD_NUM];    // 线程号集合
    pthread_mutex_init(&Mutex, NULL);   // 初始化互斥锁
```

接下来是核心过程(需要计时)，和实验一的对应部分基本相同：

```
    auto start_time = chrono::high_resolution_clock::now();     // 记录开始时间
    for(int i = 0; i < THREAD_NUM; ++i){
        pthread_ids[i] = i;
        pthread_create(&handles[i], NULL, calculate_sum, (void*)&pthread_ids[i]);   // 创建并启动线程
    }

    for(int i = 0; i < THREAD_NUM; ++i){
        pthread_join(handles[i], NULL);     // 等待线程结束并回收线程资源
    }

    if(Length % THREAD_NUM != 0){
    // 可能数组长度未必能被THREAD_NUM整除,因此要将对剩余的(Length % THREAD_NUM)进行求和
        for(size_t i = THREAD_NUM * nums_per_thread; i < Array.size(); ++i){
            global_sum += Array[i];
        }
    }
    auto end_time = chrono::high_resolution_clock::now();     // 记录开始时间
    auto using_time = (double)(end_time - start_time).count() / 1e9;
        /*
            使用chrono::high_resolution_clock::now()算出来的时间差(chrono::high_resolution_clock::duration<double>)是纳
            秒级别的，所以要除以1e9也即10的9次方才能转换成秒
        */
```

打印数组、求和结果以及所用时间并销毁互斥锁：

```
    printf("数组如下: \n");
```

```
    for(size_t i = 0; i < Array.size();++i){
        printf("%lf ",Array[i]);
    }
    printf("\n");
    printf("在%d 个线程并行的情况下对大小为%s 的数组求和的结果为%lf，所用时间为：%lf s.\n",THREAD_NUM,len.c_str(), global_sum, using_time);
    pthread_mutex_destroy(&Mutex);    // 销毁锁
```

选做(lab3_2_op.cpp)：

思路与实验二基本相同，只是使用一个全局数组 sums(vector<double>类型)来存储每个线程的求和结果，最后在主线程中对 sums 的元素进行求和，以下仅展示修改部分。

定义 sums：

```
    vector<double> sums;    // 用来存储每个线程的求和结果
```

修改线程求和函数(取消互斥锁，将 id 号线程的求和结果存在 sum[id]中)：

```
    // 求和线程
    void* calculate_sum(void* id){
        int thread_id = *((int*)id);    // 获取线程号
        long long begin = thread_id * nums_per_thread, end = (thread_id+1) * nums_per_thread;   // 求和起点和终点：[begin,end)
        double local_sum = 0.0;     // 局部和
        for(long long i = begin; i < end; ++i){
            local_sum += Array[i];
        }
        sums[thread_id] = local_sum;    // 将线程秋和结果存储在 sums 中
        pthread_exit(NULL);     // 终止线程
    }
```

在 main 函数中，对 sums 进行初始化：

```
    sums.resize(THREAD_NUM);
        /*
        初始化 sums，每个线程的求和结果刚好存储在 sums 的对应位置，如 0 号线程的求和结果存储在 sums[0]，
        如果数组长度不能被线程数整除，把剩余部分的求和结果再加入到 sums 末尾即可
        */
```

修改数组长度不能被 THREAD_NUM 整除的情况：

```
    if(Length % THREAD_NUM != 0){
```

```
    // 可能数组长度未必能被 THREAD_NUM 整除，因此要将对剩余的(Length % THREAD_NUM)进行求和
        double temp = 0;
        for(size_t i = THREAD_NUM * nums_per_thread; i < Array.size(); ++i){
            temp += Array[i];
        }
        sums.push_back(temp);
    }
```

对 sums 进行求和：

```
    for(size_t i = 0; i < sums.size(); ++i){
        global_sum += sums[i];
    }
```

3. 实验结果（500 字以内，图文并茂）

实验一：

■ 编译：

```
g++ -g -Wall -o lab3_1 lab3_1.cpp -lpthread
```

■ 运行(这里以两个 256*256 的方阵、4 个进程为例)：

```
./lab3_1 256 256 256 4
```

■ 下为运行示例，仅展示程序正确性：

```
laigg@laigg-VirtualBox:~/codes/parallel/lab3$ g++ -g -Wall -o lab3_1 lab3_1.cpp -lpthread
laigg@laigg-VirtualBox:~/codes/parallel/lab3$ ./lab3_1 4 4 4 2
Matrix A:
1.179236 -1.326176 -3.998677 4.566106
3.987246 2.894828 -3.514136 -1.331624
-1.334542 -3.184554 3.135368 4.647102
-3.334261 4.203861 4.177196 -0.492804
Matrix B:
2.978315 0.784439 2.921765 -4.706354
-3.466776 4.735550 1.530537 -3.198651
4.444679 2.283278 0.050680 -0.461667
-1.615937 2.142486 -3.886849 3.881305
Matrix C:
-17.041687 -4.702408 -16.534728 18.260578
-11.627827 5.959617 21.078162 -31.571016
13.491727 0.987785 -26.676979 33.056431
-5.141691 25.773946 -1.180606 -1.595671
在2个线程并行的情况下计算大小为4*4的矩阵A和大小为4*4的矩阵B所用时间为: 0.000131 s
laigg@laigg-VirtualBox:~/codes/parallel/lab3$
```

线程数	矩阵规模				
	128	256	512	1024	2048

1	0.007758s	0.068132s	0.606801s	9.340406s	157.574867s
2	0.005072s	0.033510s	0.317842s	4.683009s	80.714968s
4	0.002324s	0.023914s	0.219638s	3.648411s	52.728892s
8	0.003124s	0.026366s	0.200472s	3.174203s	51.367260s
16	0.003632s	0.030814s	0.193590s	2.899490s	50.634349s

可以看到使用 pthread 多线程并行矩阵乘法的效率比使用 MPI 的效率要高很多。这是因为 pthread 使用轻量级的线程进行通信，其通信开销比较低，创建和销毁开销也更小。除此之外，pthread 通常在共享内存系统中进行并行计算，这意味着线程可以直接访问内存中的数据，而不需要复制或传输数据。

选做：

■ 编译：

```
1.  g++ -g -Wall -o lab3_1_op lab3_1_op.cpp -lpthread
```

■ 运行(这里以两个 256*256 的方阵、4 个进程为例)：

```
1.  ./lab3_1_op 256 256 256 4
```

■ 下为运行示例，仅展示程序正确性：

```
laigg@laigg-VirtualBox:~/codes/parallel/lab3$ g++ -g -Wall -o lab3_1_op lab3_1_op.cpp -lpthread
laigg@laigg-VirtualBox:~/codes/parallel/lab3$ ./lab3_1_op 4 4 4 2
Matrix A:
4.635534 -1.597281 1.832962 -0.490998
4.361741 2.067329 1.973900 4.028786
-3.245580 -1.799218 1.617226 2.932947
4.514831 2.982774 -1.486620 -0.710525
Matrix B:
4.965075 2.013511 -2.945488 -4.963591
0.010269 1.438799 3.310416 -3.445032
-1.178474 -2.360680 0.880615 -3.411557
-1.064087 -0.175036 -4.991409 0.345020
Matrix C:
21.361736 2.794438 -14.876670 -23.928868
15.064426 6.391957 -24.374808 -34.115971
-21.159793 -13.454848 -9.611728 17.802766
24.955104 17.016075 -1.186776 -27.858984
在2个线程并行的情况下计算大小为4*4的矩阵A和大小为4*4的矩阵B所用时间为: 0.000219 s
laigg@laigg-VirtualBox:~/codes/parallel/lab3$
```

线程数	矩阵规模				
	128	256	512	1024	2048

1	0.008103s	0.066320s	0.644357s	10.361447s	157.746503s
2	0.004221s	0.035353s	0.315991s	4.735242s	88.171904s
4	0.003519s	0.026221s	0.213063s	2.345699s	57.231163s
8	0.003736s	0.022326s	0.204245s	2.111639s	55.701782s
16	0.003349s	0.026495s	0.200008s	1.865378s	56.130254s

可以看到进行规模较大的矩阵乘法时，对矩阵 B 进行划分最终的效率和对矩阵 A 进行划分的效率基本相同。这是因为 B 是按列访问，空间局部性较差，虽然每个线程不用访问 B 的所有列，但是在最外层循环需要对 A 的所有行进行访问，循环次数(M*(K/THREAD_NUM)*N，对 A 进行划分每个线程的循环次数为(M/THREAD_NUM)*K*N)并不会减少，虽然在主进程会有除不尽时的循环次数的不同(一个是 M%THREAD_NUM，一个是 K%THREAD_NUM)，但实验中的矩阵规模均可以被 THREAD_NUM 整除，故二者的效率基本相同。

实验二：

■ 编译：

```
1.  g++ -g -Wall -o lab3_2 lab3_2.cpp -lpthread
```

■ 运行：

```
1.  ./lab3_2
```

■ 下为运行示例，仅展示程序正确性：

```
laigg@laigg-VirtualBox:~/codes/parallel/lab3$ g++ -g -Wall -o lab3_2 lab3_2.cpp -lpthread
laigg@laigg-VirtualBox:~/codes/parallel/lab3$ ./lab3_2
请输入数组长度(数组长度为输入范围在[1M,128M]的整数可以输入具体的整数也可以输入如"1M、1.25M"的数):
4
请输入线程数量(线程数量为[1,16]的整数):
2
数组如下:
0.722950 0.806488 0.205295 0.936705
在2个线程并行的情况下对大小为4的数组求和的结果为2.671438，所用时间为: 0.000300 s.
laigg@laigg-VirtualBox:~/codes/parallel/lab3$
```

线程数	数组规模							
	1M	2M	4M	8M	16M	32M	64M	128M
1	0.003904s	0.006205s	0.009591s	0.024410s	0.042329s	0.076956s	0.172507s	0.313813s

2	0.001886s	0.003678s	0.004896s	0.011866s	0.021580s	0.041688s	0.083179s	0.159479s
4	0.002063s	0.001915s	0.006063s	0.009743s	0.019309s	0.026996s	0.050700s	0.125504s
8	0.001269s	0.002481s	0.004206s	0.007365s	0.014726s	0.021194s	0.040738s	0.081187s
16	0.001790s	0.002382s	0.003889s	0.006956s	0.012170s	0.022309s	0.041640s	0.083597s

总体上来说，随着线程数的增加，求和效率是越来越高的，且随着数组规模的增大，整体上效率也是越来越高。但是由于使用了互斥锁，而其又会导致阻塞，这就使得在线程数增加时反而会增加执行时间，此外，虽然线程的创建和销毁开销较进程会小一些，但是线程数增加带来的创建和销毁开销也会增加，这是线程数增加而执行时间未减少的原因之一。

选做：

■ 编译：

```
1.  g++ -g -Wall -o lab3_2_op lab3_2_op.cpp -lpthread
```

■ 运行：

```
1.  ./lab3_2_op
```

■ 下为运行示例，仅展示程序正确性：

```
laigg@laigg-VirtualBox:~/codes/parallel/lab3$ g++ -g -Wall -o lab3_2_op lab3_2_o
p.cpp -lpthread
laigg@laigg-VirtualBox:~/codes/parallel/lab3$ ./lab3_2_op
请输入数组长度(数组长度为输入范围在[1M,128M]的整数可以输入具体的整数也可以输入如
"1M、1.25M"的数):
8
请输入线程数量(线程数量为[1,16]的整数):
2
数组如下:
0.937620 0.553338 0.906900 0.455632 0.313221 0.888221 0.608325 0.187822
在2个线程并行的情况下对大小为8的数组求和的结果为4.851081，所用时间为: 0.000212 s
```

线程数	数组规模							
	1M	2M	4M	8M	16M	32M	64M	128M
1	0.002971s	0.005272s	0.013513s	0.019634s	0.044871s	0.094512s	0.162943s	0.329822s
2	0.001793s	0.003392s	0.007406s	0.012108s	0.021954s	0.038380s	0.080875s	0.159761s
4	0.002059s	0.001982s	0.005400s	0.010633s	0.019857s	0.028180s	0.040211s	0.088608s
8	0.001886s	0.001562s	0.004436s	0.006868s	0.015131s	0.022904s	0.040658s	0.082807s
16	0.002634s	0.002177s	0.003565s	0.007080s	0.012829s	0.026631s	0.041450s	0.077642s

可以看到不使用互斥锁而使用一个数组来存储各线程的求和结果并对该数组中的元素进行求和的效率和使用互斥锁的效率基本一致。这是因为虽然不会出现因为互斥锁导致阻塞，但是在主线程中多了一步：将各个进程的求和结果再进行求和，其实相当于将各个线程中可能出现的阻塞导致的开销转移到主线程。

4. 实验感想（200 字以内）

通过本次实验，我回顾了此前 OS 课程学习的 pthread 编程，掌握了如何使用 pthread 进行多线程并行矩阵乘法和数组求和。此外通过对比 MPI 并行实验，我也理解了进程通信与线程通信的区别。最后我也尝试并分析了不同的任务划分方式对矩阵乘法效率的影响和不同的聚合方式对数组求和效率的影响。

在本次实验中，我一开始使用 C 的 clock 函数进行计时，导致出现了线程数越多，执行时间不减反增的情况。通过回顾此前理论课关于 clock 函数的内容，明白了 clock 函数只记录 CPU 时间而不包含等待时间，这就导致线程数越多，CPU 时间越多。故改用 chrono 库进行计时（其精度更高且可以计算代码延时）后，解决了该问题。