中山大学数据科学与计算机院本科生实验报告

(2020 学年秋季学期)

课程名称：**高性能计算程序设计**　　任课教师：**黄聃**　　**批改人：**

年级+班级	2018 级五班	专业（方向）	计算机科学与技术
学号	18340126	姓名	罗仁良
Email	luorliang@mail2.sysu.edu.cn	完成日期	2020 年 10 月 22 日

1 实验目的

1.1 Pthread 实现通用矩阵乘法

1.2 基于 Pthread 的数组求和

1.3 Pthread 求解二次方程的根（利用条件变量）

1.4 Pthreads 多线程实现基于 monte-carlo 方法的$y = x^2$阴影面积估算

2 实验过程和核心代码

2.1 Phtread 实现通用矩阵乘法

2.1.1 矩阵数据存储

矩阵数据存储的存储方式和 MPI 实验中相似，使用一位数组保存矩阵，为提高高速缓存的命中率，部分矩阵采用存储转置矩阵的方式。Phtread 是共享内存的多线程并行，不需要消息传递机制实现数据共享，一定程度减少了通信开销。

2.1.2 任务划分

设矩阵$A \in R^{m \times n}$，矩阵$B = [\ b_1\ \ b_2\ \ b_3 \dots b_k\ ] \in R^{n \times k}$，则

$$A \cdot B = [\ Ab_1\ \ Ab_2\ \ Ab_3 \dots\ \ Ab_k\ ] \in R^{m \times k}$$

- 单位任务：最小的任务是矩阵和一个向量的乘法运算（一个线程至少负责一个矩阵和一个向量相乘的运算）。
- 负载均衡：为保证负载均衡，把矩阵B的列向量平均分配到各个线程之中。

2.1.3 子线程源代码

```
/**   m , n , k , num_threads(运行线程数量)
***   A , B , C   共享数据（矩阵数据）
*** col_count 每个线程得到的任务数量
*** */

int m, n, k, num_threads;
int *A, *B, *C;
int col_count;
```

```
void *matrix_mul_vec(void *rank)
{
    // first  任务开始的列向量索引
    // last   任务结束的列向量索引
    int myrank = *(int *)rank;
    int first, last;
    first = col_count * myrank;
    last = col_count + first;
     // 计算
    for (int i = 0; i < m; i++)
    {
        for (int j = first; j < last; j++)
        {
            int sum = 0;
            for (int l = 0; l < n; l++)
            {
                sum += A[i * n + l] * B[j * n + l];
            }
            C[j * m + i] = sum;
        }
    }

    return NULL;
}
```

2.1.4 主线程

```
     col_count = k / num_threads;              // 根据线程数量划分任务
    pthread_t *thread;
    int* rank;                                 //定义线程序号
    rank = new int[num_threads];
thread = new pthread_t[num_threads];          //子线程

    for (int i = 0; i < num_threads; i++)      // 创建子线程
    {
        rank[i] = i;
        pthread_create(thread + i, NULL, matrix_mul_vec, rank+i);
    }

    for (int i = 0; i < num_threads; i++)      // 等待子线程运行结束
    {
        pthread_join(thread[i], NULL);
    }
```

2.2 基于 Pthread 的数组求和

2.2.1 实现思路

利用一个全局共享的变量 global_index 实现对数组元素的访问，子线程通过 global_index 获得需要计算的数组元素。为保证程序的正确运行，用一个互斥量防止多个线程同时获取同一个数组元素，导致重复计算使得结果出错。同时存储求和结果的变量也是一个全局共享的变量，也需要实现互斥访问。

2.2.2 互斥访问

```
pthread_mutex_lock(&mutex);       // 进临界区加锁
sum += A[global_index];
global_index++;
pthread_mutex_unlock(&mutex);     // 出临界区解锁
```

2.2.3 组访问

原程序线程每次进入临界区只能得到一个元素，进行一次计算。实际上，在每次只计算一个元素的时候，程序相当于串行执行，还额外增加了线程创建销毁，临界区访问的开销，没有达到并行优化的目的。

改进的方式是每次线程进入临界区获取多一点计算任务，使得多个进程都在同时计算，减少对临界区的访问，从而实现并行优化的目的。

```
int size = 10;              // 每次的计算任务
while (global_index < arrSize) {
      // 获取任务
    pthread_mutex_lock(&mutex_index);
    if(global_index + size >= arrSize) size = arrSize-global_index;
    global_index += size;
    pthread_mutex_unlock(&mutex_index);

      // 计算
    int mysum = 0;
    for(int i=0;i<size;i++) mysum += A[global_index-size+i];

      // 汇总结果
    pthread_mutex_lock(&mutex_sum);
    sum += mysum;
    pthread_mutex_unlock(&mutex_sum);
}
```

以上代码分别使用了两个互斥量 mutex_index 和 mutex_sum 分别控制对 global_index 和 sum 变量的访问。

2.3 Phtread 计算一元二次方程的根（条件变量）

根据求根公式

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

令$A = -\frac{b}{2a}$，$B = \frac{\sqrt{b^2-4ac}}{2a}$，$C = A \pm B$.

2.3.1 任务划分

实验过程中使用了三个线程（线程 1，线程 2，线程 3）分别计算 A，B，C 的值，其中 C 的值需要等待 A、B 的值被计算出之后才能开始计算。

2.3.2 条件变量

下面以线程 1 完成计算 A 通知线程 3 的过程为例，简要分析实验过程中条件变量配合互斥量使用的方式。

线程 1

```
pthread_mutex_lock(&mutex1);
targ * t = (targ *) argv;
*(t->result) = - t->b / (2 * t->a); // A = *(t->result)
r1_flag++;
pthread_cond_signal(&cond1);
pthread_mutex_unlock(&mutex1);
```

线程 1 进入临界区，首先计算 A 的值，再通过函数 pthread_cond_signal(&cond1)；通知线程 3，A 已完成计算，可以访问。其中 r1_flag 用于标记 A 是否完成计算，初始化为 0。

线程 3

```
pthread_mutex_lock(&mutex1);
if(r1_flag <= 0)        pthread_cond_wait(&cond1,&mutex1);
double a = *(r->a);
pthread_mutex_unlock(&mutex1);
```

线程 3 进入临界区，通过标记值判断 A 是否计算完成，若完成，则直接访问即可；若未完成，调用 pthread_cond_wait(&cond1,&mutex1)，阻塞在条件 cond1 等待线程 1 完成计算，通知线程继续执行。

2.4 Pthreads 多线程实现基于 monte-carlo 方法的$y = x^2$阴影面积估算

2.4.1 任务划分

任务划分比较简单，基于线程数和总的随机点数平均分配即可。

2.4.2 临界区

类似于多线程数组求和，每个线程需要把自己的结果汇总到一起，hit_sum 是一个全局变量，在汇总结果求和的时候需要互斥访问。

```
pthread_mutex_lock(&mutex);
hit_sum += hit;
pthread_mutex_unlock(&mutex);
```

3 实验结果

3.1 通用矩阵相乘

时间（s） 矩阵大小 / 线程数量	512	1024	2048
1	0.304051	0.2445574	19.46263
2	0.152557	0.1213752	9.753112
4	0.078314	0.612525	4.958428
8	0.071244	0.458197	3.603394

对比之前的 MPI 的实验结果，使用 Pthread 的计算结果好快很多，间接可以得出通信开销在并行程序处理中是一个很大的开销。从表格的结果可以看出，1-2-4 的线程数量和计算时间提速成正比关系，但是 4 线程到 8 线程不符这一规律。该结果的主要原因是 CPU 的核心和线程数限制，我在 CPU 只支持 6 线程，8 线程程序运行时，8 个线程并没有实现同一时间运行，只是通过调度实现了并发而已所以加速比没有相应提高。

```
 lab3  → ./mat_thread 2048 2048 2048 1
m n k : 2048 2048 2048
time: 19462.630000 ms
 lab3  → ./mat_thread 2048 2048 2048 2
m n k : 2048 2048 2048
time: 9753.112000 ms
 lab3  → ./mat_thread 2048 2048 2048 4
m n k : 2048 2048 2048
time: 4958.428000 ms
 lab3  → ./mat_thread 2048 2048 2048 8
m n k : 2048 2048 2048
time: 3603.394000 ms
```

注：矩阵计算结果以文件的形式保存在 MatrixA，MatrixB，MatrixC 中。

3.2 数组求和

3.2.1 单个求和

线程数量	1	2	4	8
时间（ms）	0.169	0.269	0.392	0.977

从表中结果可以发现，线程数量越多，计算时间越长，似乎和预期不符。实际上前文有分析，在每次只取一个元素计算的情况下，程序相当于时串行运行，还增加了额外的线程创建销毁调度等开销，所以导致时间更长。

3.2.2 分组求和

数组大小 1000

线程数量	1	2	4	8
时间（ms）	0.190	0.240	0.454	0.797

从表中数据仍然可以发现线程数量越多，计算时间越长，与预期不符。分析可能的原因时每次计算 10 个数组元素还是太少，线程多数时间是在等待进入临界区获取任务。从结果可以看到，分组求和还是要快于单个求和的情况。

数组大小 1000 分组大小 100

线程数量	1	2	4	8
时间（ms）	0.156	0.193	0.400	0.763

从表中数据可以看到提高分组大小有优化的作用，但还是不太理想，进一步分析可能数据集太小，无法体现并行的优势，计算时间小与线程的额外开销。

数组大小10^7 分组大小 1000

线程数量	1	2	4	8
时间（ms）	18.846	11.227	7.487	7.092

可以看到提高数据集和分组大小后，并行优化的效果就比较明显。

3.3 Pthread 求解二次方程的根（利用条件变量）

```
 lab3  → ./solve
input a , b , c: 1 2 1
the solution is :
x1 : -1
x2 : -1
 lab3  → ./solve
input a , b , c: 5 17 9
the solution is :
x1 : -0.655969
x2 : -2.74403
```

3.4 Monte-carlo 方法

```
 lab3  → ./monte 1000 10
result:0.332000
 lab3  → ./monte 10000 10
result:0.336800
 lab3  → ./monte 100000 10
result:0.333410
 lab3  → ./monte 1000000 10
result:0.333709
 lab3  → ./monte 10000000 10
result:0.333251
 lab3  → ./monte 100000000 10
result:0.333342
```

从结果可以看到，随机点越多，结果越接近理论值（1/3）。

4 实验感想

经过本次实验，基本掌握了利用 pthread 实现多线程编程的方式方法。熟悉掌握了有关控制互斥访问相关的操作，例如互斥量、条件变量等。通过矩阵相乘的实验，进一步体会到共享内存式的并行编程通信开销小的优势。在数组求和实验过程中，加深了对并行程序设计中，线程创建、调度、销毁等操作带来的额外开销，对程序运行速度的影响。