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	**18级计7**	**专业（方向）**	**计科**
学号	18340166	**姓名**	王若琪
Email	wangrq29@mail2.sysu.edu.cn	**完成日期**	2020.10.13

1. 实验目的

（1）通过 Pthreads 实现通用矩阵乘法的并行版本。

（2）实现基于 Pthreads 的数组求和。

① 编写使用多个进程/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，每个线程通过共享单元 global_index 获取 a 数组的下一个未加元素，注意不能在临界段外访问全局下标 global_index 。

② 重写上面的例子，使得各进程可以一次最多提取 10 个连续的数，以组为单位进行求和，从而减少对下标的访问。

（3）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，使用求根公式计算。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。

（4）编写一个 Pthreads 多线程程序来实现基于 monte-carlo 方法的 y=x^2 阴影面积估算。

（5）通过本次实验，熟悉并掌握 Pthreads 编程的几大基本操作和函数的使用。

2. 实验过程和核心代码

2.1 Pthreads 实现并行通用矩阵乘法

Pthreads 实现并行矩阵乘法的主要思想是，生成多个线程，每个线程中执行一部分矩阵 A 的行与矩阵 B 的乘法。

每个线程内运行的函数主要有两个功能，一是确定该线程需要计算的矩阵 A 的范围，即确定 A 的开始行数和结束行数，确定范围部分核心代码如下：

```
//确定每个线程的计算范围
    int my_rank = (long)rank;
    int my_first_M, my_last_M; //矩阵 A 的开始行数和结束行数
    int quotient = M / thread_count;
    int remainder = M % thread_count;
```

```
    int my_n_count;
    if (my_rank < remainder)
    {
        my_n_count = quotient + 1;
        my_first_M = my_rank * my_n_count;
    }
    else
    {
        my_n_count = quotient;
        my_first_M = my_rank * my_n_count + remainder;
    }
    my_last_M = my_first_M + my_n_count;
```

在确定好的范围内进行矩阵乘法的核心代码如下，和之前 GEMM 的过程非常类似，只是起始结束的范围改变了，关键代码如下：

```
    for (int i = my_first_M; i < my_last_M; i++)
    {
        for (int j = 0; j < K; j++)
        {
            matrixC[i][j] = 0;
            for (int l = 0; l < N; l++)
            {
                matrixC[i][j] += matrixA[i][l] * matrixB[l][j];
            }
        }
    }
```

在主线程中，先为每个线程的pthread_t对象分配内存：

```
thread_handles = malloc(thread_count * sizeof(pthread_t)); //为每个线程的pthread_t对象分配内存
```

再生成线程：

```
    for (thread = 0; thread < thread_count; thread++)           //生成线程
    {
        pthread_create(&thread_handles[thread], NULL, gemm, (void *)thread);
    }
```

停止线程：

```
    for (thread = 0; thread < thread_count; thread++) //停止线程
    {
        pthread_join(thread_handles[thread], NULL);
    }
```

最后释放内存：

```
free(thread_handles);
```

这样就完成了并行矩阵乘法的主体部分。

此外，参照课本附的代码，采用宏定义和 gettimeofday() 完成计时部分：

```
#define GET_TIME(now) { \
   struct timeval t; \
   gettimeofday(&t, NULL); \
   now = t.tv_sec + t.tv_usec/1000000.0; \
}
```

最后编译生成可执行文件，编译指令：

```
gcc -std=c99 -g -o gemm gemm.c -lpthread
```

运行指令：

```
./gemm <number of threads>
```

其他相关细节请见源代码 gemm.c 。

2.2 基于 Pthreads 的数组求和

2.2.1 每个线程通过共享单元 global_index 获取 a 数组的下一个未加元素

实现此问题需要用到临界区思想。首先声明一个全局的互斥量：

```
pthread_mutex_t mutex;
```

再调用函数pthread_mutex_init()进行初始化：

```
pthread_mutex_init(&mutex, NULL);
```

在每个线程的求和函数中进行循环，循环开始时先设置获得临界区的访问权，再进行对 global_index 的判断，如果等于 1000 ，则退出临界区，并跳出循环，如果小于 1000 ，则进行一次加法操作，并将 global_index 加一。此部分关键代码如下：

```
void *array_sum(void *rank)
{
   while (1)
   {
      pthread_mutex_lock(&mutex);//获得临界区的访问权
      if (global_index == NUM)
      {
         pthread_mutex_unlock(&mutex);//退出临界区
         break;
      }
      sum += array[global_index];
      global_index++;
      pthread_mutex_unlock(&mutex);//退出临界区
   }
   return NULL;
}
```

最后编译生成可执行文件，编译指令：

```
gcc -std=c99 -g -o array_sum1 array_sum1.c -lpthread
```

运行指令：

```
./array_sum1 <number of threads>
```

其他相关细节请见源代码 array_sum1.c 。

2.2.2 每线程10个一组进行求和

为了减少对全局下标的访问，使各个进程一次提取 10 个数组元素进行求和。实现此方法时，在各线程中执行时，需要先在临界区内访问 global_index，并更新 global_index，再退出临界区，在临界区外进行 10 个一组的求和，最后再进入临界区，对全局变量 sum 进行更新。此部分关键代码如下：

```
void *array_sum(void *rank)
{
    int temp_index;
    int local_sum;
    while (1)
    {
        local_sum = 0;
        pthread_mutex_lock(&mutex); //获得临界区的访问权
        if (global_index == NUM)
        {
            pthread_mutex_unlock(&mutex); //退出临界区
            break;
        }
        else
        {
            temp_index = global_index;
            global_index += 10;
            pthread_mutex_unlock(&mutex); //退出临界区
        }

        for (int i = 0; i < 10; i++)//在临界区外进行10个一组求和
            local_sum += array[temp_index + i];

        pthread_mutex_lock(&mutex); //获得临界区的访问权
        sum += local_sum;//加到sum里面
        pthread_mutex_unlock(&mutex); //退出临界区
    }
    return NULL;
}
```

最后编译生成可执行文件，编译指令：

```
gcc -std=c99 -g -o array_sum2 array_sum2.c -lpthread
```

运行指令：

```
./array_sum2 <number of threads>
```

其他相关细节请见源代码 array_sum2.c 。

2.3 Pthreads 求解二次方程组的根

本实验的原理在于，用不同的线程计算求根公式的不同中间值，并用条件变量来识别是否执行完每一步，从而控制同步。

我设置了三个线程，第一个线程（0号线程）负责计算出 $delta = b^2 - 4ac$，等计算出来 delta 之后，解锁条件变量 cond_var1，第二个线程（1号线程）开始计算两个不同的分母部分 $denominator1 = -b - \sqrt{delta}$，$denominator2 = -b + \sqrt{delta}$，等算出来两个分母之后，解锁条件变量 cond_var2，第三个线程（2号线程）开始执行，求得最终两个根 x1 和 x2。

第一个线程（0号线程）核心代码如下，主要功能为计算出 delta 的值，如果小于零，说明无解，就退出程序；如果大于零，就用 pthread_cond_signal() 进行对下一步的解锁。

```
void *thread_work_0(void *rank) //求得delta
{
    pthread_mutex_lock(&mutex);
    delta = b * b - 4 * a * c;
    if (delta < 0)
    {
        printf("No solution!\n");
        exit(0);
    }
    pthread_cond_signal(&cond_var1); //解锁，使得1号线程计算可以进行
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

第二个线程（1号线程），先是用 pthread_cond_wait() 阻塞并等待解锁，解锁之后，先计算出两个分母的值，再用 pthread_cond_signal() 进行对下一步的解锁。此部分关键代码如下：

```
void *thread_work_1(void *rank) //求得分母
{
    pthread_mutex_lock(&mutex);
    while (pthread_cond_wait(&cond_var1, &mutex) != 0); //阻塞线程直到被解锁
    denominator1 = (double)-1 * b - sqrt(delta);
    denominator2 = (double)-1 * b + sqrt(delta);
    pthread_cond_signal(&cond_var2); //解锁，使得2号线程计算可以进行
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

第三个线程（2号线程），先是用 pthread_cond_wait() 阻塞并等待解锁，解锁之后，计算出两个最终解，运算结束。此部分关键代码如下：

```
void *thread_work_2(void *rank) //求得最终结果
{
    pthread_mutex_lock(&mutex);
    while (pthread_cond_wait(&cond_var2, &mutex) != 0)
        ; //阻塞线程直到被解锁
    x1 = denominator1 / (2 * a);
    x2 = denominator2 / (2 * a);
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

最后编译生成可执行文件，编译指令：

```
gcc -std=c99 -g -o solve_equation solve_equation.c -lpthread -lm
```

运行指令：

```
./solve_equation
```

其他相关细节请见源代码 solve_equation.c 。

2.4 基于monte-carlo方法的 y=x^2 阴影面积估算

在 [0, 1] 之间随机生成 x ， y，相当于在一个面积为 1 的小方块内随机投掷点，如果判断 $x^2 \leq y$ ，则相当于这个点落在了曲线下方的阴影部分中。生成许多个这样的点，用小方块的面积 1 乘上落在阴影部分的点所占全部点的比例，就得到了阴影部分面积的估算。

先模仿课本例子中的程序设计出随机数生成函数：

```
unsigned my_rand(unsigned *seed_p)
{
    long long z = *seed_p;
    z *= MR_MULTIPLIER;
    z %= MR_MODULUS;
    *seed_p = z;
    return *seed_p;
}
double my_drand(unsigned *seed_p)
{
    unsigned x = my_rand(seed_p);
    double y = x / MR_DIVISOR;
    return y;
}
```

再设计每个线程执行的函数部分，关键代码如下：

```
void *Thread_work(void *rank)
{
    long my_rank = (long)rank;
    long long toss;
    long long local_number_in = 0;
    long long local_tosses = number_of_tosses / thread_count;
    long long start = local_tosses * my_rank;
    long long finish = start + local_tosses;
    double x, y;
    unsigned seed = my_rank + 1;
    for (toss = start; toss < finish; toss++)
    {
        x = 2 * my_drand(&seed) - 1; //随机生成x
        y = 2 * my_drand(&seed) - 1; //随机生成y
        if (x * x <= y)              //判断落在阴影部分
            local_number_in++;
    }
    pthread_mutex_lock(&mutex); //获得临界区的访问权
    number_in += local_number_in;
```

```
    pthread_mutex_unlock(&mutex); //退出临界区
    return NULL;
}
```

最后编译生成可执行文件，编译指令：

```
gcc -std=c99 -g -o monte_carlo monte_carlo.c -lpthread
```

运行指令：

```
./monte_carlo <number of threads> <number of tosses>
```

其他相关细节请见源代码 monte_carlo.c 。

---

3. 实验结果

3.1 Pthread 并行矩阵乘法结果

经编译运行并测试不同规模的矩阵，得到运行时间结果如下表：

线程数\矩阵规模	512 512 512	1024 1024 1024	2048 2048 2048
1	0.529410 s	4.930633 s	96.844620 s
2	0.285578 s	2.651447 s	45.616497 s
3	0.217143 s	1.857635 s	28.722177 s
4	0.177312 s	1.537285 s	22.141498 s
5	0.209230 s	1.880890 s	25.014058 s
6	0.196533 s	1.635584 s	19.744331 s
7	0.199945 s	1.709800 s	19.744331 s
8	0.173432 s	1.851840 s	19.023646 s

由上表可见，在线程数较少的时候，随着线程数的增加，运行时间有明显的减少。但是当线程数增加到一定数目时（4个线程以上），运行时间几乎稳定不变。

3.2 两种数组求和方法运行结果

分别用上文 2.2 中提及的两种方法对有1000个元素的数组进行测试，得到的运行时间如下表：

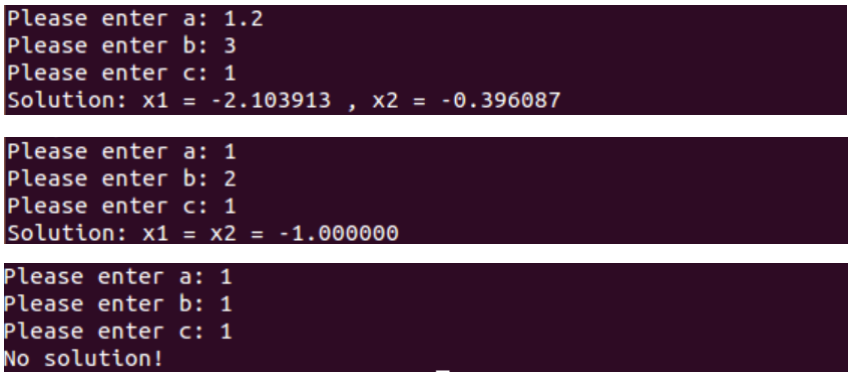
线程数\方法	1. 逐个求和计算	2. 每10个一组计算
1	0.000359 s	0.000320 s
2	0.000683 s	0.000379 s
3	0.000514 s	0.000364 s
4	0.000541 s	0.000451 s
5	0.000636 s	0.000442 s
6	0.000676 s	0.000699 s
7	0.000661 s	0.000773 s
8	0.000818 s	0.000977 s

纵向对比，无论是第一种方法还是第二种方法，总体来说都是当线程数越多，运行时间越长。

横向对比，在线程数较少的时候，每10个一组求和的方法会比逐个求和的方法稍快一些，但在线程数较大的时候，区别差距不是很明显，这可能是因为数组规模还不够大而导致实验结果差别不明显。

3.3 Pthreads 求解二次方程的根

输入几组数据进行测试，可以得到正确结果，如下列截图：



3.4 阴影面积估算结果

测试了不同 number_of_tosses 的结果，如下表：

number_of_tosses	Estimated value
1000	3.060000e-01
10000	3.332000e-01
100000	3.331600e-01
1000000	3.337850e-01
10000000	3.334716e-01
100000000	3.332869e-01

从表格中可以看出，随着 number_of_tosses 的增大，面积估算越来越接近准确值。

---

4. 实验感想

- 虽然在之前其他科目的的课程设计已经有过对 Pthreads 的使用，但本次实验是我第一次系统学习 Pthreads 编程，对它的原理有了更深入的了解，实验过程中也遇到并解决了许多小问题：
- 在做 Pthreads 实现并行矩阵乘法时，一开始我用了 clock() 计时，结果发现当并行线程多了计时反而久了，不符合实际，于是经过搜索网络资料，发现原来 clock() 返回的值是多个核心的时钟滴答数的总和，所以结果会不符合预料。于是我参考课本的资料，新增了一个计时的宏定义，通过使用 gettimeofday 来完成正确计时，就解决了计时的问题。
- 在做基于 Pthreads 的数组求和时，一开始将判断 global_index 的过程写在了临界区外，后来发现这样做不严谨，于是就改到了临界区里面。不仅写操作需要在临界区内，读操作也应该放在临界区内，才能做到不出差错。
- 用蒙特卡洛方法估计面积的实验中，我从课本例题中学到了新的生成真正随机数的方法。
- Pthreads 编程很考验实验中的细致程度，本次实验中我由于不够细致产生过许多bug，今后要多加注意。
