中山大学计算机学院本科生实验报告

（2021学年秋季学期）

课程名称：高性能计算程序设计基础　　批改人：

实验	**LAB3**	专业	计算机科学与技术（超算）
学号	19335162	姓名	潘思晗
Email	pansh25@mail2.sysu.edu.cn	完成日期	2021.10.28

一、实验目的

> *1.*构造 *MPI*版本矩阵乘法加速比和并行效率表。
>
> *2.*通过 *Pthreads*实现通用矩阵乘法（*Lab1*）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*。
>
> *3.*编写使用多个进程/线程对数组*a[1000]*求和的简单程序演示
>
> - 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*
> - 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
>
> *4.*编写一个多线程程序来求解二次方程组 $ax^2+bx+c=0$的根，
>
> *5.*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。

二、实验过程和核心代码

0.构造MPI版本矩阵乘法加速比和并行效率表

标准矩阵乘法

加速比S

COMM_SIZE	128 128 128	512 512 512	1024 1024 1024	2048 2048 2048
1	1.00	1.00	1.00	1.00
2	1.00	1.30	1.34	1.55
4	1.00	2.80	2.67	2.00
8	0.03	0.18	0.33	1.00
16	0.00	0.04	0.15	0.12

并行效率$E = S/p$

COMM_SIZE	128 128 128	512 512 512	1024 1024 1024	2048 2048 2048
1	100%	100%	100%	100%
2	50.0%	65.0%	67.0%	77.5%
4	25.0%	70.0%	66.8%	50.0%
8	0.38%	2.25%	4.13%	12.5%
16	0.03%	0.25%	1.0%	0.75%

优化后矩阵乘法

加速比

COMM_SIZE	128 128 128	512 512 512	1024 1024 1024	2048 2048 2048
1	1.00	1.00	1.00	1.00
2	1.00	1.85	1.88	1.19
4	0.35	3.98	3.96	2.23
8	0.02	0.14	0.37	0.68
16	0.00	0.04	0.15	0.27

并行效率

COMM_SIZE	128 128 128	512 512 512	1024 1024 1024	2048 2048 2048
1	100%	100%	100%	100%
2	50.0%	92.5%	94.0%	59.5%
4	8.75%	99.5%	99.0%	50.8%
8	0.25%	1.75%	4.63%	8.5%
16	0.06%	0.25%	1.0%	1.69%

点对点通信在增加进程数量时，并行效率会减小。而以相同的倍率提高问题的规模时，加速效率存在小于，等于和大于原来的加速效率的情况。因此，点对点通信的强可扩展性较差，弱可扩展性会随着问题的规模和进程的数量发生变化。

集合通信在增加进程数量时，有可能存在加速效率不变的情况。对于增加进程数量导致加速效率减小的情况，加速效率存在小于，等于和大于原来的加速效率的情况。因此，集合通信的强可扩展性和弱可扩展性都会随着问题的规模和进程的数量发生变化。

1.通过 Pthreads实现通用矩阵乘法

（1）算法描述

通用矩阵乘法通常定义为: $C = AB$， $C_{m,n} = \sum_{n=1}^{N} A_{m,n}B_{n,k}$

串行版本`GEMM()`函数实现：传入矩阵A、B并为矩阵C申请空间，通过循环嵌套实现矩阵的相乘，最后返回计算结果的指针，如下所示

```
int** GEMM(int** matA,int** matB
{
    int** ans = (int**)malloc(sizeof(int*)*m);
    for(int i=0;i<m;i++){
        ans[i]=(int*)malloc(sizeof(int)*k);
    }
    for(int i=0;i<m;i++){
        for(int j=0;j<k;j++){
            ans[i][j] = 0;
            for(int t=0;t<n;t++){
                ans[i][j] += matA[i][t]*matB[t][j];
```

```
            }
        }
    }
    return ans;
}
```

（2）核心代码

pthread实现矩阵乘法，很重要的设计思路是确定每个线程thread执行什么任务，如果每个线程执行的任务很复杂，那么代码就会很复杂冗余。通过搜索资料，发现了一种使用pthread进行矩阵乘法的很巧妙的思路：让每个线程计算A的某一行与B的某一列的乘积，那么需要给每个线程传递的参数就是A的行数和B的列数，传递的参数可以很简单。

这种方法可以利用结构体struct轻松完成。

设置结构体parameter，两个成员变量r和c分别代表行数和列数。

```
struct parameter {
    int r,c;
};
```

这样，在线程函数`thread_compute`中，只需传入结构体，进行单行单列相乘即可：

```
void *thread_compute(void *x) {
    struct parameter *data = (parameter*)x;
    double sum = 0;
    for(int i = 0; i < n; i++){
        sum += matA[data->row][i] * matB[i][data->column];
    }
    matC[data->row][data->column] = sum;
    pthread_exit(0);
}
```

在main函数中，首先对矩阵进行初始化：

```
matA = generate_matrix(m,n);
matB = generate_matrix(n,k);
matC = (double**)malloc(sizeof(double*)*m);;
for(int i=0;i<m;i++){
    matC[i] = (double*)malloc(sizeof(double)*k);
}
```

然后对矩阵每行每列进行循环，将当前的行数和列数赋值给结构体x，然后使用`pthread_attr_init()`函数对线程初始化；再创建线程并调用`thread_compute`函数，最后调用`pthread_join()`函数，确保thread的子线程执行完毕后才能执行下一个线程。

```
for(int i = 0; i < m; i++) {
    for(int j = 0; j < k; j++) {
        struct parameter *x = (struct parameter *)
malloc(sizeof(struct parameter));
        x->row = i;
        x->column = j;
        pthread_t t;
        pthread_attr_t attr;
        pthread_attr_init(&attr);
        pthread_create(&t,&attr,thread_compute,x);
        pthread_join(t, NULL);
    }
}
```

完成后标记时间并计算得到运行总时长，最后打印出三个矩阵A、B、C以及完成矩阵乘法所用时间。

2.基于Pthreads的数组求和

2-1 每次获取1个元素

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。

首先声明全局变量

```
pthread_mutex_t mutex;  //互斥锁
int array[1000];        //array数组存放待相加的元素
int threads_num;        //线程数目
int global_index = 0;   //全局索引，共享单元
int sum = 0;            //累加和
```

初始化锁并对待加数组array进行随机赋值

```
clock_t begin, end;
double time1;
threads_num=strtol(argv[1], NULL, 10);
pthread_t thread[threads_num];
pthread_mutex_init(&mutex, NULL);
srand((unsigned)time(0));
for (int i = 0; i < 1000; i++)
{
    array[i] = (int)rand() % 100;
}
```

创建线程并调线程函数`add_array()`进行加和

```
begin = clock();

for (int i=0;i<threads_num;i++){
    for (int j=0;j<1000/threads_num;j++)
        pthread_create(&thread[i], NULL, add_array, NULL);
}
for (int i=0;i<threads_num;i++){
    for (int j=0;j<1000/threads_num;t++)
        pthread_join(thread[i], NULL);
}
pthread_mutex_destroy(&mutex);

end = clock();
```

线程函数`add_array()`如下

```
void* add_array ()
{
    pthread_mutex_lock(&mutex);
    sum += array[global_index];
    global_index++;
    pthread_mutex_unlock(&mutex);
}
```

最后结束计时并打印累加和sum以及运行时间。

2-2 线程10个一组进行求和

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。

主题部分与上面相同，区别在于减少了读取下标的次数，因此main函数中创建的线程数是原来的1/10：

```
begin = clock();

for (int i=0;i<threads_num;i++){
    for (int j=0;j<1000/threads_num/10;j++)
        pthread_create(&thread[i], NULL, add_array, NULL);
}
for (int i=0;i<threads_num;i++){
    for (int j=0;j<1000/threads_num/10;t++)
        pthread_join(thread[i], NULL);
}
pthread_mutex_destroy(&mutex);

end = clock();
```

在各线程中执行时，设置了`local_sum`存放每10个一组的数字和，再对全局变量 sum 和 global_index 进行更新：

```
void* add_array ()
{
    int local_sum = 0;
    pthread_mutex_lock(&mutex);
    for(int i = 0; i < 10; i++){
        local_sum += array[global_index+i];
    }
    sum += local_sum;
    global_index += 10;
    pthread_mutex_unlock(&mutex);
}
```

最后打印出两种方法的时间，对比结果。

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。

（1）算法描述

选择不同的中间值计算，有很多种不同的算法设计思路。

此处选择使用两个线程分别计算$\triangle=b^2-4ac$和$\sqrt{\triangle}=\sqrt{b^2-4ac}$，这样第二个线程的计算要在第一个线程计算完成后，因此为了保证线程同步问题，使用pthreads的信号量和条件变量来解决。

（2）核心代码

设置全局变量

```
double a, b, c;
double delta,radical;
pthread_mutex_t mutex;
pthread_cond_t cond1,cond2;
```

线程函数

第一个线程计算$\triangle = b^2 - 4ac$

```
void *work_1(void *rank)
{
    pthread_mutex_lock(&mutex);
    delta = b * b - 4 * a * c;
    if (delta < 0)
    {
        printf("The equation has no solution.\n");
        exit(0);
    }

    pthread_cond_signal(&cond1);
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

第11行在计算完成后对条件变量`pthread_cond_signal(&cond1);`加锁。

第二个线程计算分子$\sqrt{\triangle}$

```
void *work_2(void *rank)
{
    pthread_mutex_lock(&mutex);
    while (pthread_cond_wait(&cond1, &mutex) != 0);
    radical = sqrt(delta);
    pthread_cond_signal(&cond2);
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

第4行阻塞线程直到收到cond1的返回结果，解锁。

第6行对条件变量`pthread_cond_signal(&cond1);`加锁，确保主线程的调用顺序不混乱。

> 注意：*pthread_cond_wait()* 用于阻塞当前线程，等待别的线程使用 *pthread_cond_signal()*来唤醒它。 *pthread_cond_wait()* 必须与 *pthread_mutex* 配套使用。*pthread_cond_wait()*函数一进入*wait*状态就会自动 *release mutex*。当其他线程通过 *pthread_cond_signal()*把该线程唤醒，使 *pthread_cond_wait()*通过（返回）时，该线程又自动获得该*mutex*。

主函数

初始化

```
pthread_mutex_init(&mutex,0);
pthread_cond_init(&cond1,NULL);
pthread_cond_init(&cond2,NULL);
```

创建线程

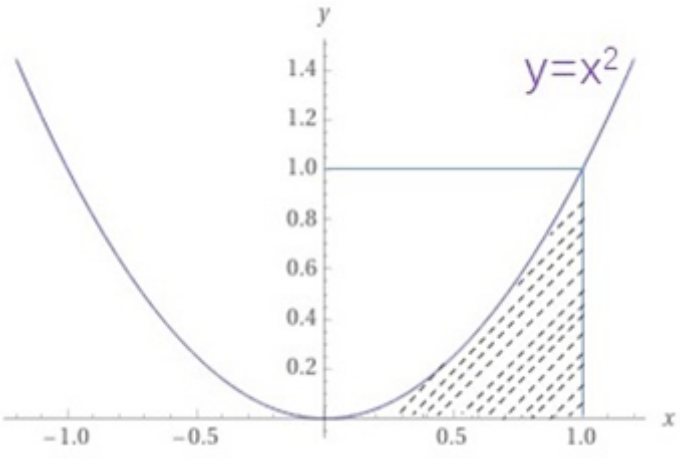
```
pthread_t thread1,thread2;
pthread_create(&thread1, NULL, work_1,(void *)0);
pthread_create(&thread2, NULL, work_2,(void *)1);
```

使用`pthread_cond_wait()`阻塞并等待cond2的解锁，解锁之后，结束线程并计算出最终结果，输出。

```
pthread_mutex_lock(&mutex);
while (pthread_cond_wait(&cond2, &mutex) != 0);
pthread_mutex_unlock(&mutex);

pthread_cond_destroy(&cond1);
pthread_cond_destroy(&cond2);
pthread_mutex_destroy(&mutex);

pthread_join(thread1,NULL);
pthread_join(thread2,NULL);

x1 = (-b+radical)/(2*a);
x2 = (-b-radical)/(2*a);
if (x1 == x2)
    printf("Solution: x1 = x2 = %f\n", x1);
else
    printf("Solution: x1 = %f , x2 = %f\n", x1, x2);
```

4.Pthreads积分求面积

编写一个Pthreads多线程程序来实现基于monte-carlo 方法的$y = x^2$与x轴之间的阴影面积估算：



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。

（1）算法描述

本题的中心是使用蒙特卡洛模拟方法进行定积分的计算。

学习课本知识可知，使用蒙特卡洛方法进行定积分计算，思想如下：

我们先设(X,Y)服从正方形 $\{0 \le x \le 1, 0 \le y \le 1\}$上的均匀分布，则可知$X,Y$分别服从$[0,1]$上的均匀分布，且$X,Y$相互独立。记事件$A = Y \le f(X)$，则A的概率为

$$P(A) = P(Y \le f(X)) = \int_0^1 \int_0^{f(x)} dydx = \int_0^1 f(x)dx = J$$

即定积分J的值就是事件A出现的频率。同时，由伯努利大数定律，我们可以用重复试验中A出现的频率作为p的估计值。即将(X,Y)看成是正方形$\{0 \le x \le 1, 0 \le y \le 1\}$内的随机投点，用随机点落在区域$y \le f(x)$中的频率作为定积分的近似值。

因此我们的算法设计思路如下：

1、首先产生服从$[0,1]$上的均匀分布的$2n$个随机数（n为随机投点个数，可以取很大，如$n=10^4$）并将其配对。

2、对这n对数据$(x_i,y_i),i=1,2,\ldots,n$，记录满足不等式$y_i \le f(x_i)$的个数，这就是事件A发生的频数μ_n，由此可得事件A发生的频率$\frac{\mu_n}{n}$，则$J \approx \frac{\mu_n}{n}$。

（2）核心代码

线程函数

线程`thread`用来计算落在$y=x^2$和x轴之间的随机点的数目，`um_n`代表每个线程中的数目，`num_in_area`代表n次事件中落在目标区域的总数。

```
double function(double x)
{
    double y = x * x;
    return y;
}

void *thread()
{
    long mu_n = 0;
    srand((unsigned)time(0));
    for(int i=0;i<thread_num;i++)
    {
        double random_x = 0+1.0*rand()/RAND_MAX *(1-0);
        double random_y = 0+1.0*rand()/RAND_MAX *(1-0);
        if(random_y<=function(random_x))
            mu_n++;
    }

    pthread_mutex_lock(&mutex);
    num_in_area += mu_n;
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

随机生成$[0,1]$范围内的随机数坐标$(random_x,random_y)$，若落在曲线$y=x^2$下方，则mu_n加1，再累加至num_in_area。

主函数

初始化

```
long number_of_tosses = 1000000000;
int thread_count = 10000;
thread_num = number_of_tosses/thread_count;
pthread_mutex_init(&mutex, NULL);
```

设置随机点的规模为10^9，线程数设置为10000，初始化mutex。

创建线程，调用线程函数`thread`，并计算和打印出最终结果。

```
srand((unsigned)time(0));
pthread_t *thread_handles = malloc(thread_count *
sizeof(pthread_t));
printf("creat threads ...\n");
for (int i = 0; i < thread_count; i++) {
    pthread_create(&thread_handles[i], NULL, thread, (void *)
NULL);
}
printf("work is over!\n");
for (int i = 0; i < thread_count; i++) {
    pthread_join(thread_handles[i], NULL);
}
pthread_mutex_destroy(&mutex);
free(thread_handles);
printf("Monte Carlo Estimates : %f\n",
(double)num_in_area/(double)number_of_tosses);
printf("Real value of integral : %lf\n",1.0/3);
```

三、实验结果

打印CPU相关信息如下

```
emilylyly@emilylyly-VirtualBox:~$ lscpu
架构:                x86_64
CPU 运行模式:        32-bit, 64-bit
字节序:              Little Endian
CPU:                 1
在线 CPU 列表:       0
每个核的线程数:      1
每个座的核数:        1
座:                  1
NUMA 节点:           1
厂商 ID:             GenuineIntel
CPU 系列:            6
型号:                142
型号名称:            Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz
步进:                11
CPU MHz:             1992.002
BogoMIPS:            3984.00
超管理器厂商:        KVM
虚拟化类型:          完全
L1d 缓存:            32K
L1i 缓存:            32K
L2 缓存:             256K
L3 缓存:             8192K
NUMA 节点0 CPU:      0
标记:                fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
freq pni pclmulqdq monitor ssse3 cx16 pcid sse4_1 sse4_2 x2apic movbe
_l1d arch_capabilities
```

1.通过 **Pthreads**实现通用矩阵乘法

测试时间

编译和运行程序的命令如下：

```
g++  -g pthread_GEMM.cpp -o  pthread_GEMM -lpthread
./pthread_GEMM m n k
```

令矩阵规模从 512 增加到 1024，测试结果如下所示：

```
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab3$ ./pthread_GEMM 512 512 512
THE TIME OF PTHREAD_GEMM: 4.724121 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab3$ ./pthread_GEMM 1024 1024 1024
THE TIME OF PTHREAD_GEMM: 27.687084 s
```

由上述结果可见，由于使用的算法没有进行优化，线程数目很多，开销很大，因此运行速度对比普通通用矩阵乘法并没有什么提升。

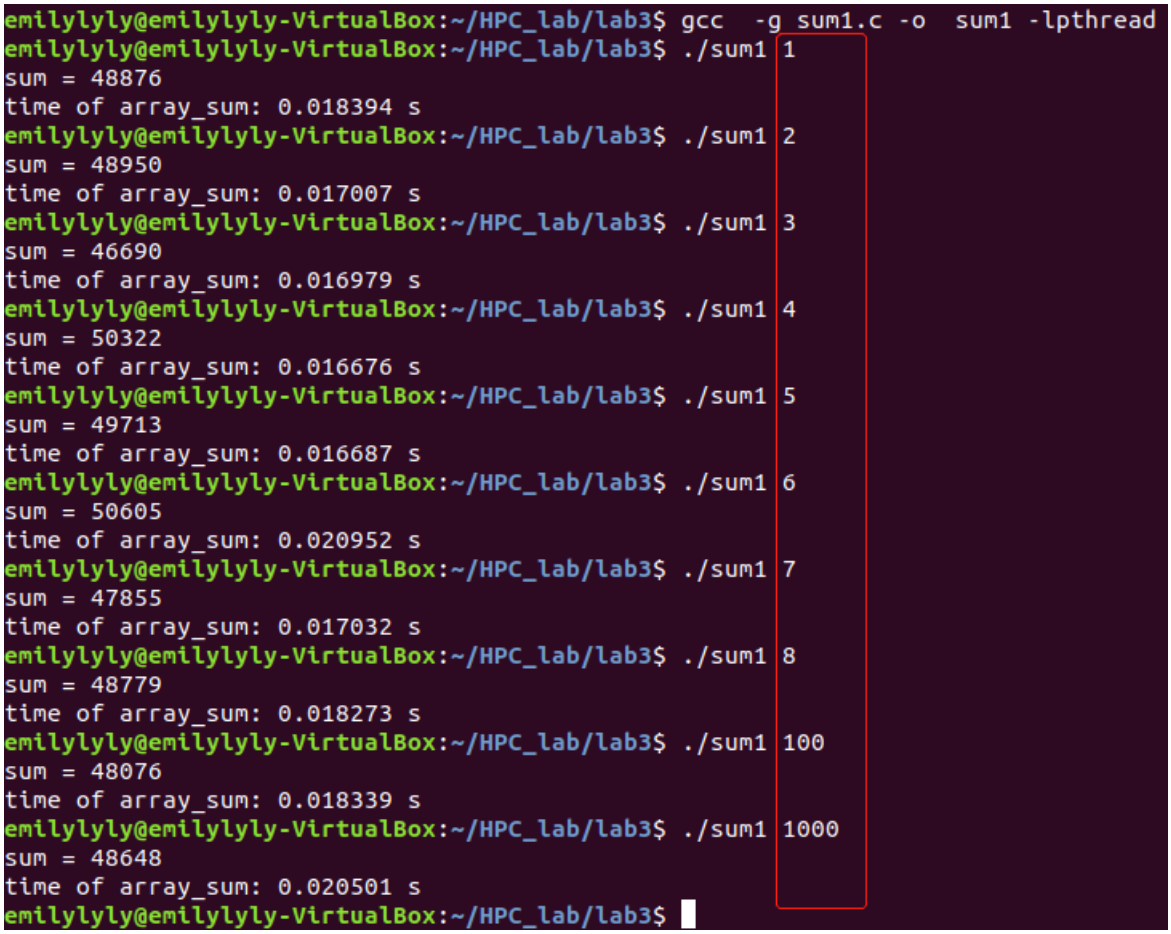
2.基于Pthreads的数组求和

2-1 每次获取1个元素

编译和运行程序的命令如下：

```
gcc  -g sum1.c -o  sum1 -lpthread
./sum1 <number of threads>
```

令并行线程不断递增，得到运行结果如下：



2-2 每次10个元素一组求和

```
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab3$ ./sum2 1
time of array_sum, 10 numbers as a group: 0.001865 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab3$ ./sum2 2
time of array_sum, 10 numbers as a group: 0.001763 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab3$ ./sum2 3
time of array_sum, 10 numbers as a group: 0.001712 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab3$ ./sum2 4
time of array_sum, 10 numbers as a group: 0.001640 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab3$ ./sum2 5
time of array_sum, 10 numbers as a group: 0.001542 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab3$ ./sum2 6
time of array_sum, 10 numbers as a group: 0.001607 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab3$ ./sum2 7
time of array_sum, 10 numbers as a group: 0.001614 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab3$ ./sum2 8
time of array_sum, 10 numbers as a group: 0.001561 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab3$ ./sum2 100
time of array_sum, 10 numbers as a group: 0.001807 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab3$ ./sum2 1000
time of array_sum, 10 numbers as a group: 0.000005 s
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab3$
```

对比两次的实验结果可以知道，每个线程每次取10个元素要比每次取1个元素的算法效率有明显的提升。

3.Pthreads求解二次方程组的根

编译和运行程序的命令如下：

```
gcc  -g solve.c -o  solve -lpthread -lm
./solve
```

运行结果如下：

```
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab3$ ./solve
Quadratic equations of one variable: ax^2+bx+c=0
Please enter a, b and c: 1 2 1
Solution: x1 = x2 = -1.000000
```

```
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab3$ ./solve
Quadratic equations of one variable: ax^2+bx+c=0
Please enter a, b and c: 4 -4 1
Solution: x1 = x2 = 0.500000
```

```
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab3$ ./solve
Quadratic equations of one variable: ax^2+bx+c=0
Please enter a, b and c: 3.4 6 2.3
Solution: x1 = -0.562859 , x2 = -1.201846
```

```
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab3$ ./solve
Quadratic equations of one variable: ax^2+bx+c=0
Please enter a, b and c: 6 2 4
The equation has no solution.
```

验证可知，计算结果正确。

4.Pthreads积分求面积

编译和运行程序的命令如下：

```
gcc  -g integral.c -o  integral -lpthread -lm
./integral
```

运行结果如下：

由上述结果可知，在模拟次数为10^9时，计算结果与实际值的偏差已经不大，扩大规模则精确度更高。

四、实验感想

这次实验使用Pthreads对矩阵乘法做了改变，还学习了使用多线程解决积分、解方程、数组求和等问题。在实验过程中也遇到了许多值得思考的问题：

1.数组求和的代码在用g++编译时会有如下错误：

```
error: invalid conversion from ‘void* (*)()’ to ‘void* (*)(void*)’ [-fpermissive]
```

值得注意的是在gcc编译时不会出错，但是用g++就会有问题，究其原因就是C语言编译器允许隐含性的将一个通用指针转换为任意类型的指针，而C＋＋不允许。

解决方法是改用C语言修正，成功解决，之后两个程序也都选择用C语言了。

2.遇到解方程迟迟没有输出的问题。

使用如下命令，调用LLVM工具检测冲突

```
clang solve.c -fsanitize=thread -fPIE -pie -g -o solve && ./solve
```

```
emilylyly@emilylyly-VirtualBox:~/HPC_lab/lab3$ clang solve.c -fsanitize=thread -fPIE
-pie -g -o solve && ./solve
Quadratic equations of one variable: ax^2+bx+c=0
Please enter a, b and c: 1 -2 1
==================
WARNING: ThreadSanitizer: data race (pid=2383)
  Write of size 4 at 0x5635568699b8 by thread T2 (mutexes: write M0):
    #0 work_2 /home/emilylyly/HPC_lab/lab3/solve.c:38 (solve+0xb95fd)

  Previous read of size 4 at 0x5635568699b8 by thread T1:
    #0 work_1 /home/emilylyly/HPC_lab/lab3/solve.c:25 (solve+0xb9547)

  Location is global '<null>' at 0x000000000000 (solve+0x0000010cb9b8)

  Mutex M0 (0x5635568699c0) created at:
    #0 pthread_mutex_lock ??:? (solve+0x347d7)
    #1 work_1 /home/emilylyly/HPC_lab/lab3/solve.c:14 (solve+0xb941b)

  Thread T2 (tid=2386, running) created by main thread at:
    #0 pthread_create ??:? (solve+0x27d06)
    #1 main /home/emilylyly/HPC_lab/lab3/solve.c:57 (solve+0xb977c)

  Thread T1 (tid=2385, finished) created by main thread at:
    #0 pthread_create ??:? (solve+0x27d06)
    #1 main /home/emilylyly/HPC_lab/lab3/solve.c:56 (solve+0xb975a)

SUMMARY: ThreadSanitizer: data race /home/emilylyly/HPC_lab/lab3/solve.c:38 in work_2
==================
```

分析上述结果，首行的WARNING 指出了报错原因data race ，即该程序发生了数据竞争。后面指出了对同一全局变量进行读写的代码位置，是程序第38行和第25行，最后是被修改的全局变量的相关地址以及两个进程相关代码位置。

修改相应位置的代码后，成功解决。

除此之外，生成随机数的方法也值得研究，一开始选用的方法导致了不少问题，经过资料的搜索修改了随机数的生成方法，解决了问题。要记住的是srand语句播种不能放在循环体内，由于循环比较快，会导致srand播种每次都来不及，播种就一直不停的被初始化，导致不能批量生成随机数。

总体而言，通过这次实验，我对于Pthreads程序的编写有了新的理解，也学到了很多。

五、参考资料

1.https://blog.csdn.net/acdreamers/article/details/44978591

2.https://cosx.org/2010/03/monte-carlo-method-to-compute-integration

3.https://blog.csdn.net/zb1593496558/article/details/79993181

4.https://blog.csdn.net/qq_36763031/article/details/118524028

5.https://www.cnblogs.com/Spiro-K/p/6378576.html