

# 中山大学数据科学与计算机学院本科生实验报告

(2020 秋季学期)

课程名称：高性能程序设计 任课老师：黄聃 批改人：

年级 + 班级	18 级计算机	年级 (方向)	计算机科学
学号	18340236	姓名	朱煜
Email	zhuy85@mail2.sysu.edu.cn	完成日期	2020.10.20

## 1 实验目的

### 1.1 通过 Pthreads 实现通用矩阵乘法

通用矩阵乘法 GEMM 通常定义为：

$$C = AB$$
$$C_{m,n} = \sum_{k=1}^N A_{m,k} B_{k,n}$$

输入：M, N, K 三个整数 (512~2048)

问题描述：随机生成 M \* N 和 N \* K 的两个矩阵 A, B, 对这两个矩阵做乘法得到矩阵 C

输出：A, B, C 三个矩阵以及矩阵计算的时间

### 1.2 基于 Pthreads 的数组求和

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。

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。

### 1.3 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。

### 1.4 编程题：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（如图 1）

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。估算  $y = x^2$  曲线与 x 轴之间区域的面积，其中 x 的范围为 [0, 1]。

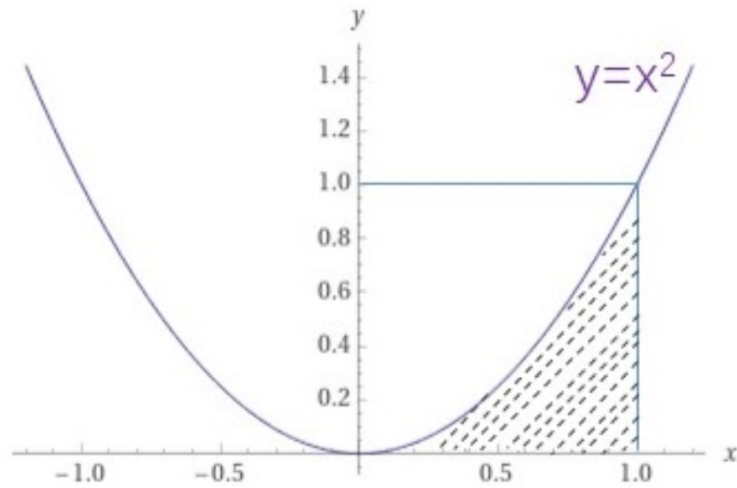


图 1:  $y = x^2$

## 2 实验过程

### 2.1 通过 Pthreads 实现通用矩阵乘法

为了方便数据传输，使用一维数组保存矩阵，矩阵元素使用 int 型，随机初始化矩阵，代码如下

---

```

1 void FillMatrix(int *matrix, int row, int col)
2 {
3     for (int i = 0; i < row; ++i)
4         for (int j = 0; j < col; ++j)
5             matrix[i * col + j] = random(0, 9);
6 }

```

---

根据 GEMM 实现简单的矩阵乘法，代码如下

---

```

1 void MatrixMul(int *A, int *B, int *C, int m, int n, int k)
2 {
3     for (int i = 0; i < m; ++i)
4     {
5         for (int j = 0; j < k; ++j)
6         {
7             int temp = 0;
8             for (int z = 0; z < n; ++z)
9                 temp += A[i * n + z] * B[z * k + j];
10            C[i * k + j] = temp;
11        }
12    }

```

---

将  $A$  矩阵根据线程数按行划分，主线程根据线程号进行划分，每个线程分别进行计算，因为各线程得到的  $C$  矩阵部分不同，因此不存在互斥问题。当线程数为 4 时，矩阵  $A$  的划分方式如图 2

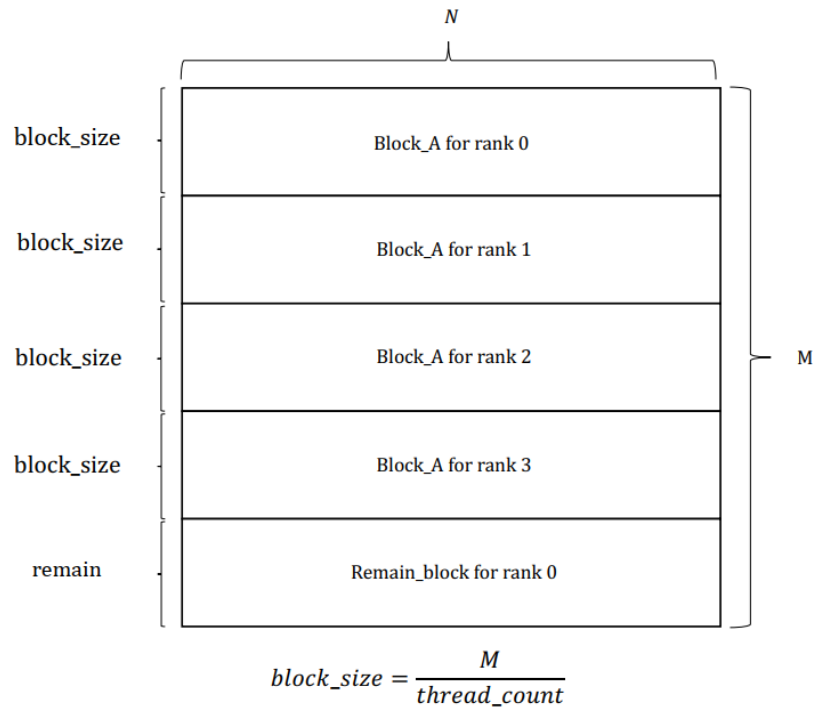


图 2: 矩阵  $A$  的划分

创建线程后，各线程运行 `void *PthMatrixMul(void *rank)` 函数进行各部分矩阵的计算，函数代码如下

---

```

1 void *PthMatrixMul(void *rank)
2 {
3     long my_rank = (long)rank;
4     MatrixMul(A + my_rank * block_size * n, B, C + my_rank * block_size * k,
5               ↪ block_size, n, k);
6     return NULL;
7 }
```

---

各线程计算结束后，主线程计算剩余的矩阵块 **Remain\_block**，得到最终的运算结果  $C$ ，主线程运行的代码如下

---

```

1 // 将矩阵 A 按行划分
2 block_size = m / thread_count;
3 gettimeofday(&start, NULL);
4 // 创建线程运行矩阵计算
```

---

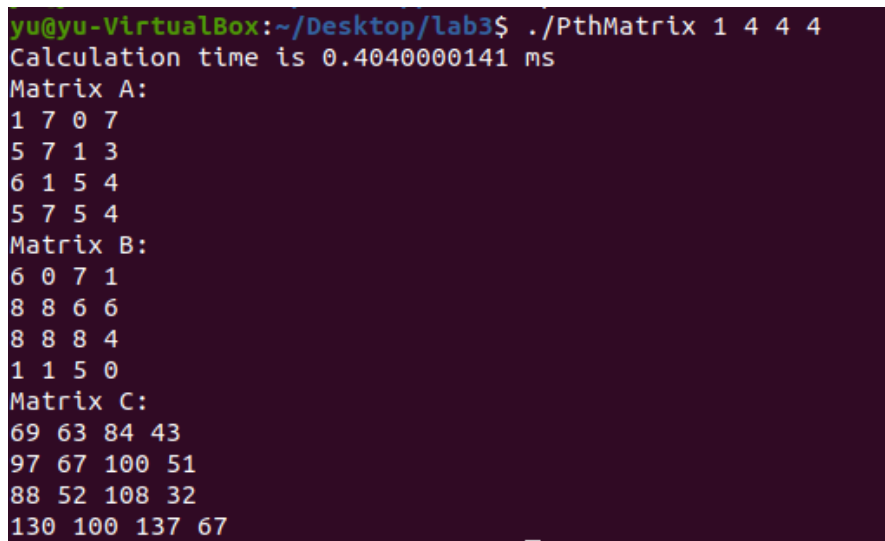
```

5  for (thread = 0; thread < thread_count; thread++)
6      pthread_create(&thread_handles[thread], NULL, PthMatrixMul, (void *)thread);
7  for (thread = 0; thread < thread_count; thread++)
8      pthread_join(thread_handles[thread], NULL);
9  free(thread_handles);
10 // 判断是否划分完 主线程完成剩余部分的计算
11 int remain = m - block_size * thread_count;
12 if (remain > 0)
13     MatrixMul(A + block_size * thread_count * n, B, C + block_size * thread_count * k,
14         ↪ remain, n, k);
14 gettimeofday(&end, NULL);
15 // 计算时间
16 cal_time = (end.tv_sec-start.tv_sec)*1000+(end.tv_usec-start.tv_usec)/1000.0;
17 printf("Calculation time is %.10f ms\n", cal_time);
18 // PrintMatrix(A, B, C, m, n, k);

```

---

为验证正确性，使用小的矩阵维度进行矩阵的打印，当矩阵规模大时，为了方便调试，将矩阵打印的取消，运行结果如图 3，则运算正确。



```

yu@yu-VirtualBox:~/Desktop/lab3$ ./PthMatrix 1 4 4 4
Calculation time is 0.4040000141 ms
Matrix A:
1 7 0 7
5 7 1 3
6 1 5 4
5 7 5 4
Matrix B:
6 0 7 1
8 8 6 6
8 8 8 4
1 1 5 0
Matrix C:
69 63 84 43
97 67 100 51
88 52 108 32
130 100 137 67

```

图 3: 4\*4 矩阵运算结果

调节线程数与矩阵规模进行输入，其运算时间结果（单位：ms）如表 1，具体分析见实验结果

## 2.2 基于 Pthreads 的数组求和

为了实现互斥，采用互斥量 `pthread_mutex_t mutex` 实现线程读取全局下标的互斥，使用 `pthread_mutex_lock(&mutex)` 函数进行加锁，使用 `pthread_mutex_unlock(&mutex)` 函数进行解锁，实现临界区，具体求和函数代码如下

表 1: 运算时间与线程数与矩阵规模的关系

线程数 \ 矩阵规模	512*512*512	1024*1024*1024	2048*2048*2048
1	501.76	6707.48	98775.18
2	243.36	2967.13	58112.74
4	134.53	1644.72	35785.62
6	138.12	1884.19	38000.77
8	148.44	1932.82	32888.43

---

```

1 void *PthArgSum(void *rank)
2 {
3     // 各线程循环计算
4     while (true)
5     {
6         // 临界区
7         pthread_mutex_lock(&mutex);
8         if (gloal_index < 1000)
9         {
10             sum += A[gloal_index];
11             gloal_index++;
12             pthread_mutex_unlock(&mutex);
13         }
14         // 下标超过 999 则计算结束
15         else
16         {
17             pthread_mutex_unlock(&mutex);
18             break;
19         }
20     }
21     return NULL;
22 }

```

---

为方便演示，初始化矩阵  $A$  为 0 到 999，使用多线程计算和，*main* 函数代码如下

---

```

1 int main(int argc, char *argv[])
2 {
3     // 初始化互斥量
4     pthread_mutex_init(&mutex, NULL);
5     // 计算从 0 到 999 的和
6     A = new int[1000];

```

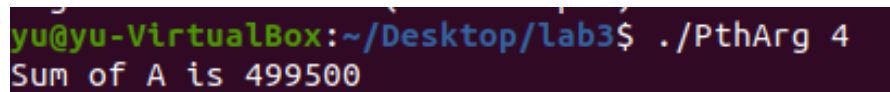
```

7     for (int i = 0; i < 1000; ++i)
8         A[i] = i;
9     long thread;
10    pthread_t *thread_handles;
11    thread_count = strtol(argv[1], NULL, 10);
12    thread_handles = (pthread_t *)malloc(thread_count * sizeof(pthread_t));
13    // 创建线程
14    for (thread = 0; thread < thread_count; thread++)
15        pthread_create(&thread_handles[thread], NULL, PthArgSum, (void *)thread);
16    for (thread = 0; thread < thread_count; thread++)
17        pthread_join(thread_handles[thread], NULL);
18    pthread_mutex_destroy(&mutex);
19    free(thread_handles);
20    printf("Sum of A is %d\n", sum);
21    return 0;
22 }

```

---

运行结果如图 4



```

yu@yu-VirtualBox:~/Desktop/lab3$ ./PthArg 4
Sum of A is 499500

```

图 4: 数组求和运行结果

修改 *PthArgSum* 函数，减少下标索引次数，修改后的代码如下

```

1 void *PthArgSum(void *rank)
2 {
3     // 各线程循环计算
4     while (true)
5     {
6         // 临界区
7         pthread_mutex_lock(&mutex);
8         if (gloal_index < 1000)
9         {
10            // 一次取十个 减少下标索引次数
11            for (int i = 0; i < 10; ++i)
12                sum += A[gloal_index + i];
13            gloal_index += 10;
14            pthread_mutex_unlock(&mutex);
15        }
16        // 下标超过 999 则计算结束

```

```

17     else
18     {
19         pthread_mutex_unlock(&mutex);
20         break;
21     }
22 }
23 return NULL;
24 }

```

---

## 2.3 Pthreads 求解二次方程组的根

实验采用四个线程来计算二次方程的根，为实现线程之间的同步，采用了信号量实现线程之间顺序进行，使用到的函数有 `sem_post()` 与 `sem_wait()`；而最后采用条件变量确定所有线程计算完毕，使用到的函数有 `pthread_cond_wait()` 与 `pthread_cond_signal()`。以上变量的定义如下

```

1  int thread_count;
2  double a, b, c;
3  // delta = b*b-4*a*c
4  double delta;
5  // root = sqrt(delta)
6  double root;
7  // 最终结果
8  double x1;
9  double x2;
10
11 // 判断 signal 是否发送过 防止阻塞
12 int count = 0;
13 // 信号量 实现线程同步
14 sem_t sem_delta;
15 sem_t sem_root;
16 // 条件变量 确定是否完成计算
17 pthread_mutex_t mutex_finish;
18 pthread_cond_t cond_finish;

```

---

线程计算根的函数 `PthRoot()` 代码如下

```

1 void *PthRoot(void *rank)
2 {
3     long my_rank = (long)rank;
4     // 0 号线程计算 delta
5     // sem_delta 实现 0 号线程与 1 号线程的同步

```

```

6     if (my_rank == 0)
7     {
8         delta = b * b - 4 * a * c;
9         sem_post(&sem_delta);
10    }
11    // 1 号线程计算 root
12    // sem_root 实现 0 号线程与 1 号线程的同步
13    else if (my_rank == 1)
14    {
15        sem_wait(&sem_delta);
16        root = sqrt(delta);
17        sem_post(&sem_root);
18    }
19    // 2 号线程计算根
20    // 条件变量判断计算结束
21    else if (my_rank == 2)
22    {
23        sem_wait(&sem_root);
24        x1 = (-b + root) / (2 * a);
25        x2 = (-b - root) / (2 * a);
26        // printf("%lf", root);
27        pthread_mutex_lock(&mutex_finish);
28        pthread_cond_signal(&cond_finish);
29        // 已经发过信号唤醒
30        count++;
31        pthread_mutex_unlock(&mutex_finish);
32    }
33    // 3 号进程打印"Calculate Finished"
34    else
35    {
36        pthread_mutex_lock(&mutex_finish);
37        // 判断是否 signal 过
38        if (count == 0)
39            pthread_cond_wait(&cond_finish, &mutex_finish);
40        printf("Calculate Finished\n");
41        pthread_mutex_unlock(&mutex_finish);
42    }
43    return NULL;
44 }

```

---

程序编译运行，计算方程  $x^2 + 2x + 1 = 0$  的解，结果如图 5



```
yu@yu-VirtualBox:~/Desktop/lab3$ ./PthRoot 1 2 1
Calculate Finished
Roots of ax^2+bx+c = 0 are x1 = -1.000000 and x2 = -1.000000
```

图 5: 求根程序运行结果

## 2.4 编程题：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（如图）

monte-carlo 方法采用随机数的方法，用频率代替概率得到阴影面积的估算。通过输入生成随机数的个数，即投掷的“飞镖”数来进行实验，多线程中，将投掷数量平均分给多个线程同时进行投掷，得到投掷在阴影面积内的总次数再在主线程中进行运算得到结果，monte-carlo 实现的函数代码如下，投掷 input 数量的“飞镖”数，返回在阴影面积的“飞镖”数。

---

```
1 long long int MonteCarlo(long long int input)
2 {
3     long long int ret = 0;
4     // 投掷"飞镖"
5     for (long long int i = 0; i < input; ++i)
6     {
7         double x = rand() / (double)(RAND_MAX);
8         double y = rand() / (double)(RAND_MAX);
9         // 判断是否在区域内
10        if (y < x * x)
11            ret++;
12    }
13    return ret;
14 }
```

---

各线程投掷得到结果后，需要将结果累加到全局和中，此时需要使用互斥量进行临界区的保护，实现代码如下

---

```
1 void *PthMonteCarlo(void *rank)
2 {
3     long long temp = MonteCarlo(number_size);
4     // sum 的临界区
5     pthread_mutex_lock(&mutex);
6     sum += temp;
7     pthread_mutex_unlock(&mutex);
8     return NULL;
9 }
```

---

运行程序，输入线程数为 4，投掷次数为 1e8，得到结果如图 6。

```
yu@yu-VirtualBox:~/Desktop/lab3$ ./PthMC 4 100000000
Result:0.333147
```

图 6: Monte-Carlo 估算结果

### 3 实验结果

#### 3.1 通过 Pthreads 实现通用矩阵乘法

**Pthreads** 实现矩阵乘法的运算时间与矩阵的规模关系如图 7，运算时间随着矩阵规模大致成线性增长，而当矩阵规模达到 2048 左右时，运算时间增长了几倍。这是由于矩阵规模增大超过缓存界限，使得多线程在同时进行计算时，L3 缓存发生了大量 miss，增加了大量的矩阵读取时间，影响了运行效率。由于采用的虚拟机环境为 4 核 4 线程，因此在线程数大于 4 时，并不能充分地进行并行，而频繁的线程切换花费了时间，提高了运算时间。

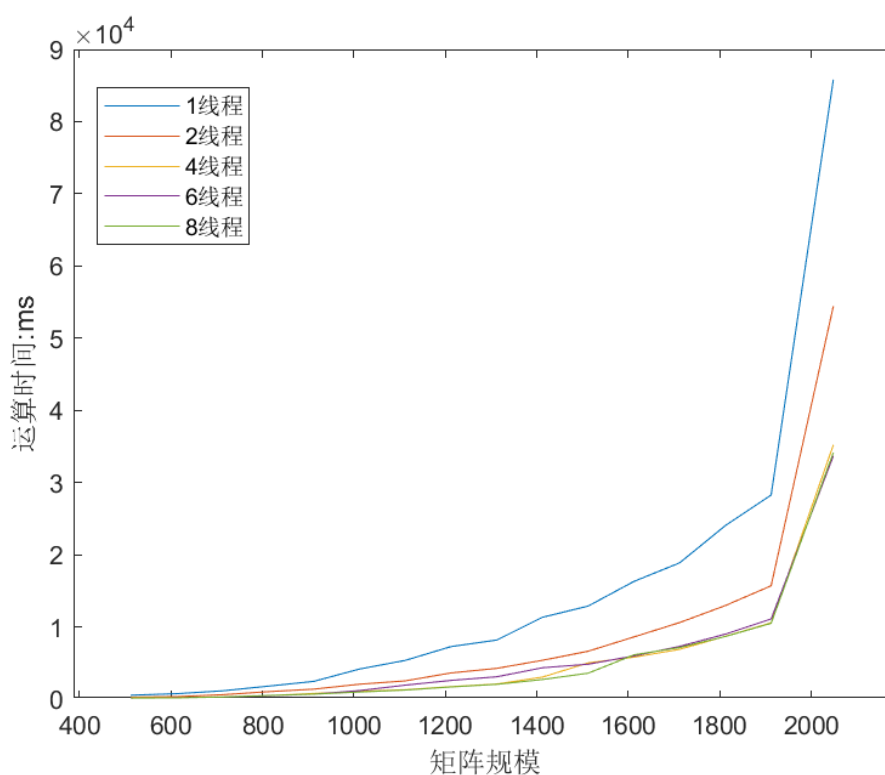


图 7: 矩阵乘法的运算时间与矩阵的规模关系

**Pthreads** 实现矩阵乘法的加速比如图 8，在矩阵规模为  $512 \times 512 \times 512$  时，近似达到了线性加速比。而当矩阵规模增大，进程数的增加会提高缓存的 miss，降低了运算效率，因此加速比增长缓慢。由于计算的上限仍为 4 进程，在进程数大于 4 时出现的加速比下降现象原因可能是进程之间的频繁切换花费了时间。

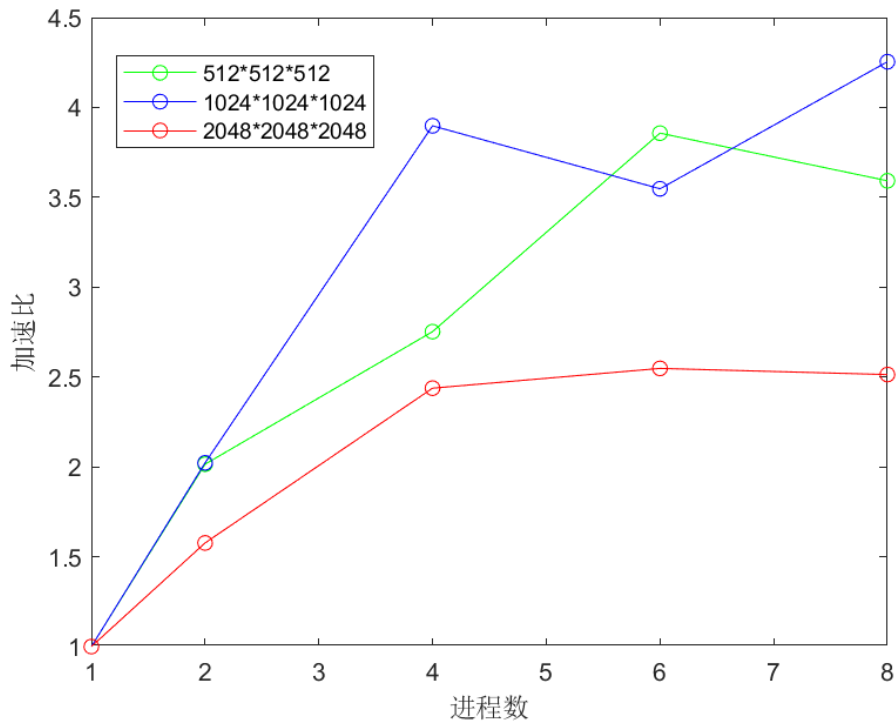


图 8: 矩阵乘法的加速比

### 3.2 基于 Pthreads 的数组求和

由于数组  $A$  的元素个数为 1000，串行求和的计算时间大致在 0.4 0.8ms，而由于存在临界区，每次只有一个线程可进入临界区，其余线程在临界区前等待，因此采用多线程求和的效果与串行求和运算时间基本没有区别。

当采用组方式求和时，由于减少了线程进入临界区的次数，也因此降低了各线程在临界区前等待的时间，因此多线程求和的效果有所提升，比较结果如图 9，其中 **PthArg** 为改进前，**PthArg2** 为改进后，分别运行 4 次，可见 **PthArg2** 的平均运算时间少于 **PthArg** 的平均运算时间。

```

yu@yu-VirtualBox:~/Desktop/lab3$ ./PthArg 4
Sum of A is 499500
Calculation time is 0.6200000048 ms
yu@yu-VirtualBox:~/Desktop/lab3$ ./PthArg 4
Sum of A is 499500
Calculation time is 0.5920000076 ms
yu@yu-VirtualBox:~/Desktop/lab3$ ./PthArg 4
Sum of A is 499500
Calculation time is 0.4440000057 ms
yu@yu-VirtualBox:~/Desktop/lab3$ ./PthArg 4
Sum of A is 499500
Calculation time is 0.6729999781 ms
yu@yu-VirtualBox:~/Desktop/lab3$ ./PthArg2 4
Sum of A is 499500
Calculation time is 0.3589999974 ms
yu@yu-VirtualBox:~/Desktop/lab3$ ./PthArg2 4
Sum of A is 499500
Calculation time is 0.3700000048 ms
yu@yu-VirtualBox:~/Desktop/lab3$ ./PthArg2 4
Sum of A is 499500
Calculation time is 0.3210000098 ms
yu@yu-VirtualBox:~/Desktop/lab3$ ./PthArg2 4
Sum of A is 499500
Calculation time is 0.2939999998 ms

```

图 9: 数组求和运算比较

### 3.3 Pthreads 求解二次方程组的根

在使用多线程求解二次方程的根时,需要控制各线程执行的顺序。实验中采用了简单的信号量来协调 0 号线程(计算  $\delta$ ),1 号线程(计算  $\sqrt{\delta}$ ),2 号线程(求根),而计算结束后使用条件变量来确定前 3 个线程已经完成计算,最后由 3 号线程判断计算结束。实现过程中存在一个小问题,则 `pthread_cond_signal` 函数为非阻塞的,即即使没有线程在 `pthread_cond_wait` 上等待, `pthread_cond_signal` 函数仍会成功返回,因此在使用条件变量的时候要确保在 `pthread_cond_wait` 上等待的线程可以被正确唤醒。

### 3.4 编程题

本次实验中采用多线程进行 **monte-carlo** 方法估算,估算面积随投掷数量的关系如图 10,在大致为  $1e5$  时, **monte-carlo** 方法估算的值已趋于稳定 使用积分方法计算实际结果,与估算结果近似,可验

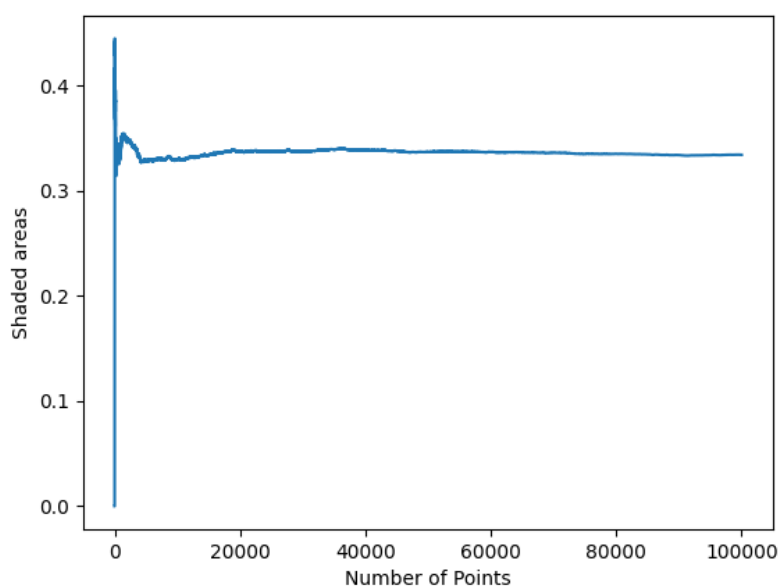


图 10: monte-carlo 估算阴影面积

证得 **monte-carlo** 方法正确。

$$\int_0^1 x^2 dx = \frac{1}{3} x^3 \Big|_0^1 = \frac{1}{3}$$

## 4 实验感想

- 本次实验来看,基于 **Pthreads** 实现的矩阵乘法依赖于全局变量,由于减少了线程之间数据的传输,因此代码较为简单。而与之之前的矩阵乘法相同,线程在运行过程中会共享相同的缓存,多线程在运行过程中因为时序的问题,导致内存优化更加困难,但为了不出现像实验中矩阵规模在 2000 左右出现的性能锐减,保证多线程的高效工作,多线程需要更加完善的内存优化策略。
- 本次实验为了实现线程的同步,使用了互斥量、信号量与条件变量三种实现同步的机制,其中互斥量多用于临界区的保护,而信号量多用于协调线程之间的工作,而条件变量在多种线程等待同一事件时

有着更好的效果,但 `pthread_cond_signal` 函数的特殊性需要特别注意,无论是否唤醒进程,该函数都会正常返回。在实验中简单地使用一个标识来判断是否已经唤醒过,防止线程在 `pthread_cond_wait` 函数阻塞,这充分说明了多线程协调工作的重要性,这需要程序员对线程执行的了解。

- 多线程技术很好地利用了利用 CPU 的资源,如果只有一个线程,多个任务的时候必须等着上一个任务完成才能进行,多线程则不用等待可在主线程执行的同时执行其他任务。在数组求和实验中,充分体现了减少线程等待时间对程序运行的优化。在多线程时,要充分地利利用多线程的 CPU 时间,减少不必要地等待,尽量利用多线程的特性,以免出现多线程负加速的效果。