

# 中山大学数据科学与计算机学院本科生实验报告

## (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 计科 8 班	专业 (方向)	超算
学号	18340208	姓名	张洪宾
Email	2285075600@qq.com	完成时间	2020 年 10 月 11 号

## 1 实验目的

- 熟悉 pthread 的基本语法, 并深入理解 pthread 模型对 fork-join 模型
- 对矩阵乘法做粒度为线程的并行, 通过不同方式对矩阵乘法做并行, 从不同角度对问题做优化, 逐渐熟悉优化问题的方式
- 理解 mutex、semaphore 和条件变量的使用, 用它们实现同步, 学会如何避免 Data Race
- 熟悉蒙特卡洛方法, 并实现蒙特卡洛方法的并行化

## 2 实验过程、核心代码

### 2.1 实验环境

- 操作系统: macOS Catalina
- C 语言编译器: Apple clang version 11.0.3 (clang-1103.0.32.29)
- Thread model: posix

### 2.2 通过 Pthreads 实现通用矩阵乘法

在上次实验中我们用 MPI 实现了通用矩阵乘法的并行化, 在这里我们采用了类似的方法, 不同之处在于, 之前的程序我们需要考虑进程通信的部分, 在这里我们因为共享内存则不需要

考虑。不过为了程序性能的最优，我考虑了两种方式对程序的局部性的影响，进而对性能的影响。

与上次相同，我们首先应该考虑怎么划分问题。在这里我采用与实验二类似的方法，在有  $p$  个进程的情况下，我们将规模为  $m \times n$  的矩阵 A 划分成  $p$  个规模为  $\frac{m}{p} \times n$  的小矩阵，然后第  $i$  个进程计算第  $i$  个小矩阵和矩阵 B 的乘法。然后就实现了矩阵的并行化。

并行计算的函数如下：

```
1 void* parallel_calculate(void*p){
2     int thread_id = (int)p;
3     for(int i = thread_id * m / thread_count; i < (thread_id + 1) * m /
4         thread_count; i++){
5         for(int j = 0; j < n; j++){
6             for(int x = 0; x < k; x++){
7                 res[i][x] += Matrix_a[i][j] * Matrix_b[j][x];
8             }
9         }
10    }
```

然后我们在 main 函数中输入了线程数量和问题规模后，就可以创建线程来计算了。因为不用考虑通信的问题，问题就变得简单了很多。在这里为了取得效果更好的优化，并且由于我们在这个问题中不会出现 Data Race 的问题，我开启了 O3 优化选项。然后将指令写到 Makefile 中。

Makefile 如下：

```
CC = gcc-9
SRCS = $(wildcard *.c *.h)
TARGET = Q1
.PHONY: all clean
all: $(TARGET)
$(TARGET): $(SRCS)
    $(CC) -w -o $$@ $$^ -O3 -lpthread
clean:
    rm $(TARGET)
```

## 2.3 基于 Pthreads 的数组求和

在这里我们需要建立一个临界区，在这个临界区里，每次都有一个线程可以将数组 a 的下一个未加元素加到全局和中。在这里我采用了信号量的方法来建立临界区。

### 2.3.1 一次只访问一个元素

首先我定义一个全局数组 a，一个全局变量 sum，用于数组求和，又定义了 global\_index，用于索引。然后我又定义了一个信号量，用于确定临界区。每次一个线程进入临界区之前调用 sem\_wait 让其他进程无法进入临界区，然后在临界区中，每次我们只要让当前线程将当前的 a[global\_index] 加入 sum 中即可。

为了保证 pthread 创建出来的线程可以不断地执行，我用了一个 while 循环，直到 global\_index 比数组的大小还要大的时候才终止。

这部分的代码如下：

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/time.h>
4  #include <pthread.h>
5  #include <semaphore.h>
6  #define Size 1000
7
8  int a[Size];
9  sem_t s;
10 int sum = 0;
11 int global_index = 0;
12 int thread_count = 8;
13 void* add(void* p){
14     int my_id = (int)p;
15     while(1){
16         sem_wait(&s);
17         if(global_index < Size){
18             sum += a[global_index];
19             global_index++;
20         }
21         else{
22             sem_post(&s);
23             break;
24         }
25     }
```

```

25     sem_post(&s);
26 }
27 }
28 int main() {
29     sem_init(&s, 0, 1);
30     for(int i = 0; i < 1000; i++){
31         a[i] = i;
32     }
33     pthread_t tid[thread_count];
34     for(int i = 0; i < thread_count; i++){
35         pthread_create(&tid[i], NULL, add, i);
36     }
37
38     for(int i = 0; i < thread_count; i++){
39         pthread_join(tid[i], NULL);
40     }
41     printf("%d\n", sum);
42 }

```

在这里我让数组初始化为  $a[i] = i$ , 于是数组求和的结果应该是  $\sum_{i=0}^{1000} i = 499500$ 。编写 Makefile, 如下:

```

CC = gcc
SRCS = $(wildcard *.c *.h)
TARGET = Q2.1
.PHONY: all clean
all: $(TARGET)
$(TARGET): $(SRCS)
    $(CC) -w -o $@ $^ -lpthread
clean:
    rm $(TARGET)

```

### 2.3.2 一次访问十个元素

在这里为了减小线程切换的开销, 我们一次访问 10 次元素, 在上一问的基础上加上简单的循环就行了。为了达到更好的性能优化, 我对代码做了以下改进: 由于我们要减小对全

局变量 `global_index` 的访问，我们可以将它取出来，然后用 `register` 关键字声明一个变量将 `global_index` 保存起来，让它放进一个寄存器中保存。同样的，为了减少对全局变量 `sum` 的访问，我们可以用一样的方法，在临界区中用 `register` 变量定义一个 `local_sum`，然后在循环体内直接将数组中的元素加到 `local_sum` 中，就可以实现优化了。

最终优化的代码如下：

```
1 void* add(void* p){
2     int my_id = (int)p;
3     while(1){
4         sem_wait(&s);
5         if(global_index < Size){
6             register int local_sum = 0;
7             register int i = global_index;
8             int count = 10;
9             global_index += 10;
10            while(count--){
11                if(i >= Size){
12                    sum += local_sum;
13                    sem_post(&s);
14                    break;
15                }
16                local_sum += a[i++];
17            }
18            sum += local_sum;
19        }
20        else{
21            sem_post(&s);
22            break;
23        }
24        sem_post(&s);
25    }
26 }
```

Makefile 如下：

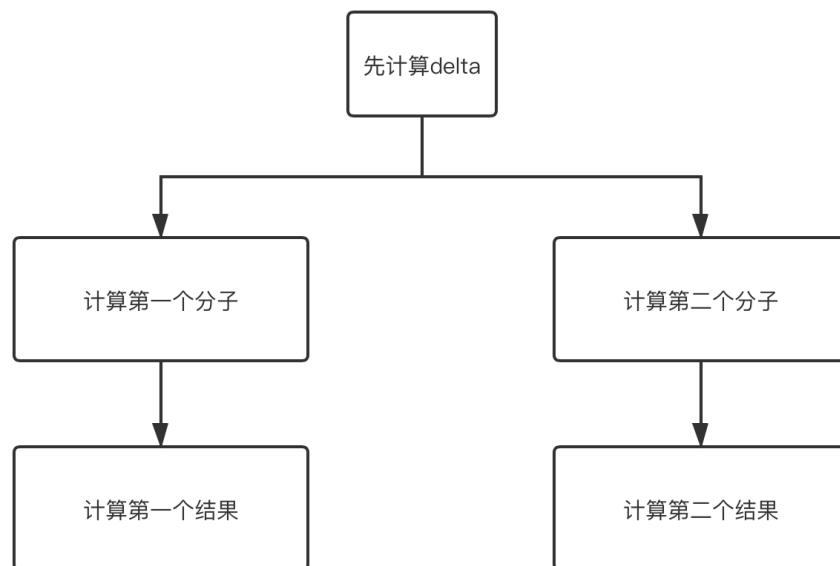
```
CC = gcc
SRCS = $(wildcard *.c *.h)
TARGET = Q2.2
```

```
.PHONY: all      clean
all: $(TARGET)
$(TARGET): $(SRCS)
    $(CC) -w -o $@ $^ -lpthread
clean:
    rm $(TARGET)
```

## 2.4 Pthreads 求解二次方程组的根

在这里题目中要求使用条件变量。在这里最重要的事情是用它实现一个 barrier，用来在生成中间变量前不要执行后续线程。

在这里我将问题分为 3 个步骤，首先计算  $\Delta = \sqrt{b^2 - 4 \times a \times c}$ ，然后再计算两种情况下的分子，分别是  $numerator1 = -b - \Delta$  和  $numerator2 = -b + \Delta$ 。最后计算两种情况下的结果，分别是  $res1 = \frac{numerator1}{2a}$  和  $res2 = \frac{numerator2}{2a}$ 。总共需要五个线程，五个线程前后依赖的关系如下：



在这里有一个问题困扰了我很久。最开始我以为条件变量和信号量类似，于是就没有注意

线程之间执行的顺序，直接在计算结果的线程的头部用了 `pthread_cond_wait` 函数，在计算分子的线程的最后使用了 `pthread_cond_signal` 函数。在运行的过程中就出现了非常玄学的情況，有的时候可以正确计算出结果，有的时候则一直没有输出，终端阻塞住了。于是我就开始 debug，通过调试我发现，如果先执行 `pthread_cond_wait` 后执行 `pthread_cond_signal` 则可以正常算出结果，否则的话，则会出现阻塞现象。

为了找到这个 bug 的原因，我开始 Google。通过查看资料我找到了它和信号量的不同：对于一个条件变量，在等待条件变量被 signal 的线程会进入队列，当 signal 发生的时候，操作系统会选择一个线程出队，当队列为空的时候，操作系统会忽略这个 signal 操作。在我上述代码中，如果先执行了 signal 操作，队列本身就为空，所以这个 signal 操作就被操作系统忽略了，而其他线程执行了 wait 操作后，永远等不到 signal 线程来释放它，就会进入长久的阻塞。而如果先执行 wait 操作后，就可以正常执行。这也是我的程序为什么时而可以输出正确结果时而会阻塞的原因了。所以我们需要保证 wait 操作在 signal 或 broadcast 操作之前完成。

为了解决这个问题，我翻了我们的课本，在教材的 120 页找到了解决该方法：

下面的代码是用条件变量实现路障：

```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
...
void* ThreadWork(...) {
    ...
    /* Barrier */
    pthread_mutex_lock(&mutex);
    counter++;
    if (counter == thread_count) {
        counter = 0;
        pthread_cond_broadcast(&cond_var);
    } else {
        while (pthread_cond_wait(&cond_var, &mutex) != 0);
    }
    pthread_mutex_unlock(&mutex);
    ...
}
```

教材提供的这种方法巧妙地设置了 barrier，我根据教材的方法编写了如下代码：

```
1 void calculate_sqrt_delta(){
2     double delta = b * b - 4 * a * c;
3     if(delta == 0){
4         flag = 1;
5     }
6     else if(delta < 0) flag = -1;
7     sqrt_delta = sqrt(delta);
8     pthread_mutex_unlock(&mutex1);
9     pthread_mutex_unlock(&mutex2);
```

```

10 }
11 void calculate_numerator_1() {
12     pthread_mutex_lock(&mutex1);
13     numerator1 = -b + sqrt_delta;
14 }
15 void calculate_numerator_2() {
16     pthread_mutex_lock(&mutex2);
17     numerator2 = -b - sqrt_delta;
18 }
19 void calculate_res1() {
20     res1 = numerator1 / (2 * a);
21 }
22 void calculate_res2() {
23     res2 = numerator2 / (2 * a);
24 }
25 void * calculate(void *p) {
26     int tid = (int)p;
27     if(tid == 0) {
28         calculate_sqrt_delta();
29     }
30     if(tid == 1) {
31         calculate_numerator_1();
32     }
33     if(tid == 2) {
34         calculate_numerator_2();
35     }
36     pthread_mutex_lock(&mutex);
37     count++;
38     if(count == target) {
39         pthread_cond_broadcast(&cond);
40     }
41     else {
42         while(pthread_cond_wait(&cond, &mutex) != 0);
43     }
44     pthread_mutex_unlock(&mutex);
45     if(tid == 3) calculate_res1();
46     if(tid == 4) calculate_res2();
47 }

```

其中 calculate 函数就是在 pthread\_create 阶段会被并行执行的函数。在其中使用了教材中实



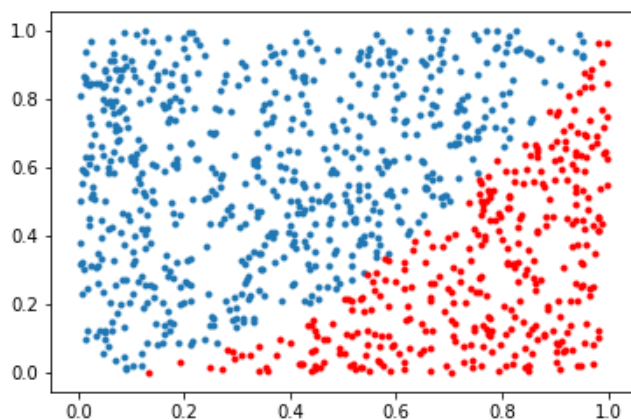
现 barrier 的方法。

同样的，我也为该程序编写了一个 Makefile，如下：

```
CC = gcc-9
SRCS = $(wildcard *.c *.h)
TARGET = Q3
.PHONY: all clean
all: $(TARGET)
$(TARGET): $(SRCS)
    $(CC) -w -o $@ $^ -lpthread
clean:
    rm $(TARGET)
```

## 2.5 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

在这里我使用了教材上的 monte-carlo 方法，首先生成一个坐标  $\{(x, y) | 0 \leq x \leq 1, 0 \leq y \leq 1\}$ ，用类似与投靶的方法，可以理解为几何概型，统计落在积分区域的点（下图中红色部分）占点的总数量的比例。当点的数量很多的时候，该比例就接近于所求的阴影部分的面积。



这个方法虽然很简单，但是有一个问题非常地重要，就是随机数的取值方式。最开始我想

要比较朴素地使用 `rand()` 函数来生成随机数，但是它是线程不安全的，在这里使用并不合适。而查看了线程安全的 `rand_r` 的源码后，我发现其实它的映射函数过于简单，可能无法得到比较好的效果。

经过和老师同学讨论后我决定使用 C++ 带有的随机数库。首先用 `std::uniform_real_distribution<double> u(0,1)` 来声明一个 (0,1) 的均匀分布,然后用 `mt19937` 声明随机数引擎 `e`，然后用线程 `tid` 与一个当前时间相关的变量的和做随机数引擎的种子。然后就可以不断使用 `u(e)` 生成在 `[0,1]` 均匀分布的随机数了。

还有一个问题很容易忽略，就是我们在给全局变量，也就是落在阴影中的点的做统计的时候容易出现 `Date_Race`。在这里我最开始使用了 `pthread_mutex` 来保护 `number_in_domain++`，但是发现运行时间非常长，对性能影响很大，于是我又决定在每个线程中使用一个局部变量 `local_sum`，用于记载该进程中已经出现在阴影面积中的点，然后最后再用 `pthread_mutex` 来保护 `number_in_domain += local_sum` 的过程。

关键的代码如下：

```
1 void * monte_carlo(void* p){
2     int64_t tid = reinterpret_cast<int64_t> (p);
3     std::mt19937 e;
4     time_t temp = time(NULL);
5     e.seed(tid + temp % 1000);
6     long long int local_sum = 0;
7     for(int i = tid * number_of_tosses / thread_count; i < (tid + 1) *
8         number_of_tosses / thread_count; i++){
9         double x = u(e);
10        double y = u(e);
11        if(y <= x * x){
12            local_sum += 1;
13        }
14    }
15    pthread_mutex_lock(&lock);
16    number_in_domain += local_sum;
17    pthread_mutex_unlock(&lock);
18 }
19 int main(int argc, char **argv) {
20     pthread_mutex_init(&lock, NULL);
21     pthread_t id[thread_count];
22     for(int i = 0; i < thread_count; i++){
23         pthread_create(&id[i], NULL, monte_carlo, (void*)i);
24     }
```

```

24     for(int i = 0; i < thread_count; i++){
25         pthread_join(id[i], NULL);
26     }
27     pthread_mutex_destroy(&lock);
28     printf("res = %f\n", 1.0 * number_in_domain / number_of_tosses);
29 }

```

与之对应的 Makefile 如下：

```

CC = g++-9

SRCS = $(wildcard *.cpp *.h)

TARGET = pthread_Monte_Carlo

.PHONY: all      clean

all: $(TARGET)

$(TARGET): $(SRCS)
    $(CC) -w -o $@ $^ -lpthread -fpermissive

clean:
    rm $(TARGET)

```

## 3 实验结果

### 3.1 通过 Pthreads 实现通用矩阵乘法

在终端中输入 make，来编译代码以获得可执行文件 Q1。然后输入 ./Q1 可以执行程序，首先需要指定线程的数量，然后再指定 m、n、k 的值，如下：

```
Q1 — -zsh — 80x24
Last login: Tue Oct 20 21:50:01 on ttys000
zhh@zhanghb-MacBook-Pro ~ % cd Desktop/HPC-lab3/Q1
zhh@zhanghb-MacBook-Pro Q1 % make
gcc-9 -w -o pthread_matrix_multiply pthread_matrix_multiply.c lib.h -lpthread -O3
zhh@zhanghb-MacBook-Pro Q1 % ./pthread_matrix_multiply
Input M:512
Input N:512
Input K:512
Time is:16196µs
zhh@zhanghb-MacBook-Pro Q1 % ./pthread_matrix_multiply
Input M:2048
Input N:2048
Input K:2048
Time is:749573µs
zhh@zhanghb-MacBook-Pro Q1 %
```

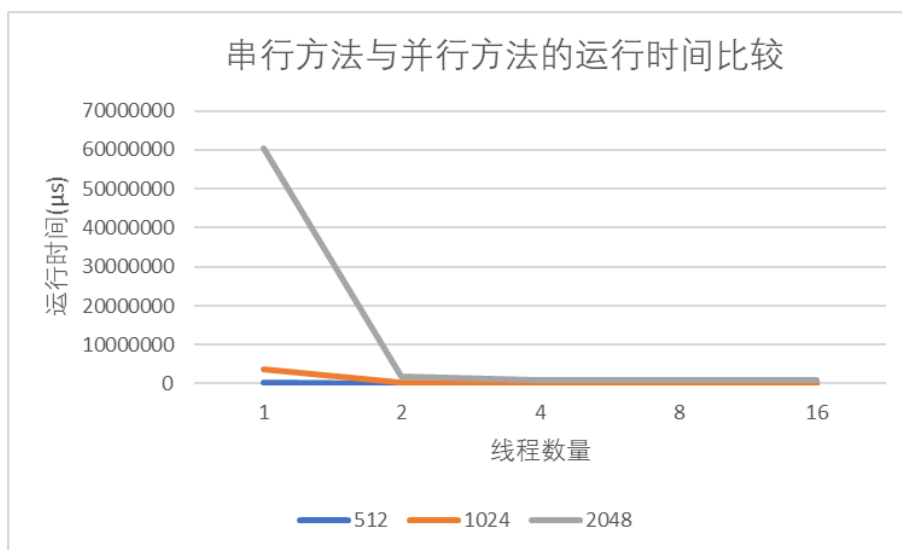
因为该程序是直接由 lab1 用 pthread 并行化得到的,所以可以使用 lab1 的程序与它做对比。严谨起见,我在 lab1 的编译时候也使用了 O3 优化。在这里为了方便起见,问题规模中 m、n、k 取相同的值。

每种线程数量和问题规模的组合我都运行了 5 次,并取平均值记录下来。

问题规模 n \ 线程数量 p	512	1024	2048
1(串行)	238314µs	3562243µs	60578801µs
2	31038µs	233392 µs	1751338µs
4	18273µs	131782 µs	1008356µs
8	15992µs	110614µs	852950µs
16	17202 µs	106716µs	849289µs

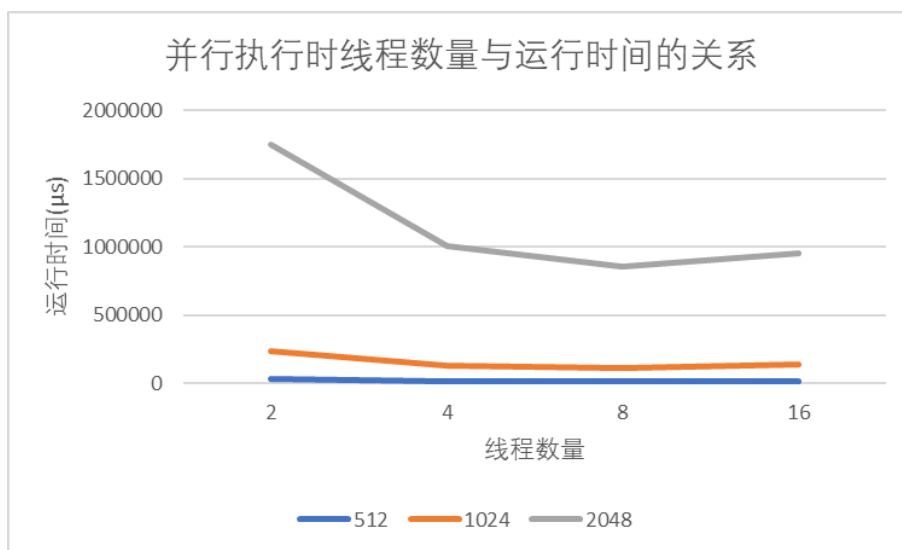
表 1: 不同进程数量下不同规模的问题的计算时间

为了更直观地显示对程序做并行后对性能的优化,我将串行执行的结果和并行执行的结果用折线图表示,如下:



我们可以看到，问题规模比较小的时候，并行计算的优势并不明显，但是当问题规模变得很大的时候，并行计算的优势非常突出。

由于该折线图很难看出什么时候可以得到最佳的线程数量，所以我单独将并程序在不同情况下的表现列举出来，结果如下：



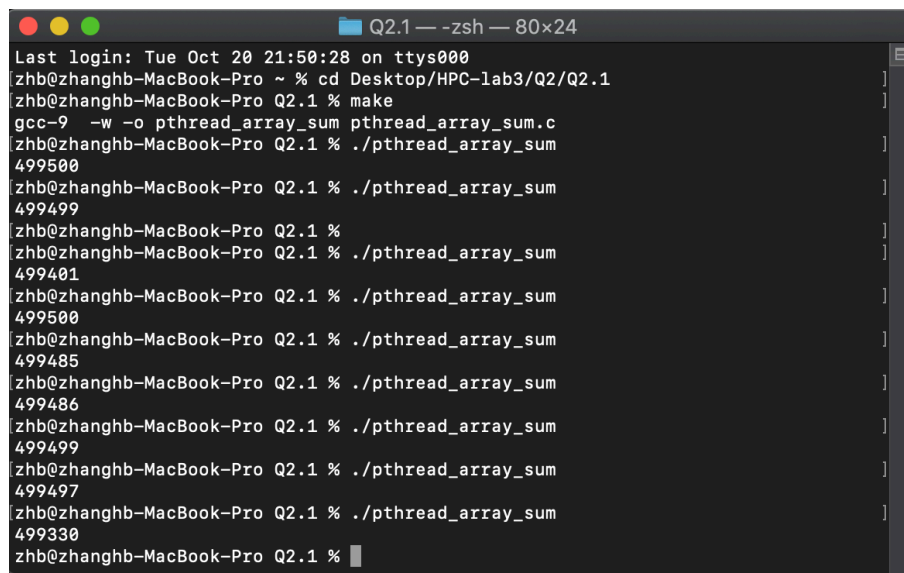
我们可以看出，在线程为 8 之前，线程越多计算时间越短，在线程为 8 之后，线程越多，计算时间不再明显缩短，甚至可能会增加。我查看了 CPU 的情况，该处理器是四核心 8

线程的，可以得出，运行程序最佳的线程数量应该与硬件的情况密切相关。

## 3.2 基于 Pthreads 的数组求和

### 3.2.1 程序正确性分析

在这里我最开始在 macOS 下输入 make 后编译，多次运行 Q1.1 后，结果如下：



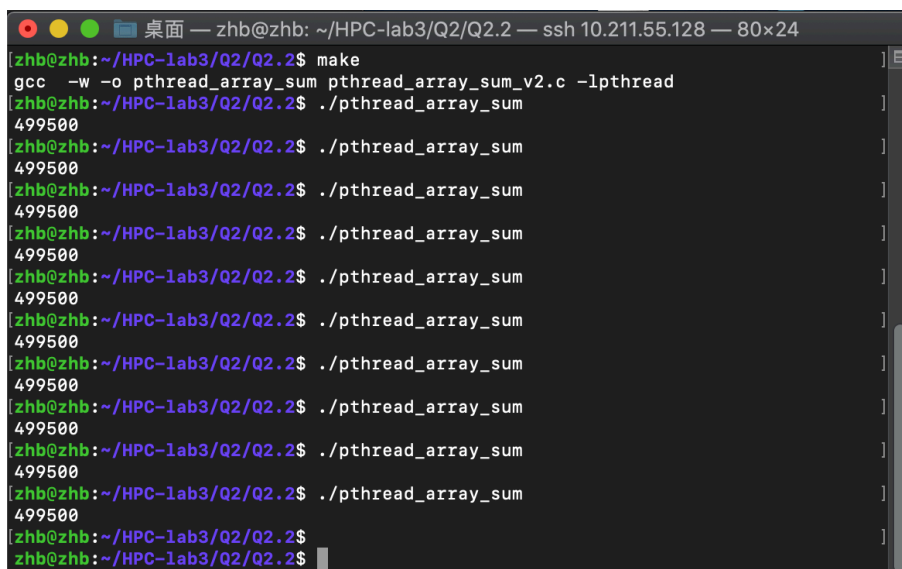
```
Q2.1 — zsh — 80x24
Last login: Tue Oct 20 21:50:28 on ttys000
zhb@zhanghb-MacBook-Pro ~ % cd Desktop/HPC-lab3/Q2/Q2.1
zhb@zhanghb-MacBook-Pro Q2.1 % make
gcc-9 -w -o pthread_array_sum pthread_array_sum.c
zhb@zhanghb-MacBook-Pro Q2.1 % ./pthread_array_sum
499500
zhb@zhanghb-MacBook-Pro Q2.1 % ./pthread_array_sum
499499
zhb@zhanghb-MacBook-Pro Q2.1 % 
zhb@zhanghb-MacBook-Pro Q2.1 % ./pthread_array_sum
499401
zhb@zhanghb-MacBook-Pro Q2.1 % ./pthread_array_sum
499500
zhb@zhanghb-MacBook-Pro Q2.1 % ./pthread_array_sum
499485
zhb@zhanghb-MacBook-Pro Q2.1 % ./pthread_array_sum
499486
zhb@zhanghb-MacBook-Pro Q2.1 % ./pthread_array_sum
499499
zhb@zhanghb-MacBook-Pro Q2.1 % ./pthread_array_sum
499497
zhb@zhanghb-MacBook-Pro Q2.1 % ./pthread_array_sum
499330
zhb@zhanghb-MacBook-Pro Q2.1 %
```

非常明显，出现了 Data Race。我仔细检查了代码，实在想不出可能会出现 Data Race 的地方。于是我打开了 Ubuntu 虚拟机，将这段代码在 Ubuntu 虚拟机中运行了多次：



程序在 macOS 上无法正确运行的原因，因为 macOS 的 POSIX 信号量并未实现无名信号量的机制。

于是我选择在 Ubuntu 下继续这部分的实验。第一部分一次只访问一个元素的正确性已经在上面提到了，在这里我们要运行第二部分的代码。如下：



```
zhhb@zhhb: ~/HPC-lab3/Q2/Q2.2 — ssh 10.211.55.128 — 80x24
zhhb@zhhb:~/HPC-lab3/Q2/Q2.2$ make
gcc -w -o pthread_array_sum pthread_array_sum_v2.c -lpthread
zhhb@zhhb:~/HPC-lab3/Q2/Q2.2$ ./pthread_array_sum
499500
zhhb@zhhb:~/HPC-lab3/Q2/Q2.2$ ./pthread_array_sum
499500
zhhb@zhhb:~/HPC-lab3/Q2/Q2.2$ ./pthread_array_sum
499500
zhhb@zhhb:~/HPC-lab3/Q2/Q2.2$ ./pthread_array_sum
499500
zhhb@zhhb:~/HPC-lab3/Q2/Q2.2$ ./pthread_array_sum
499500
zhhb@zhhb:~/HPC-lab3/Q2/Q2.2$ ./pthread_array_sum
499500
zhhb@zhhb:~/HPC-lab3/Q2/Q2.2$ ./pthread_array_sum
499500
zhhb@zhhb:~/HPC-lab3/Q2/Q2.2$ ./pthread_array_sum
499500
zhhb@zhhb:~/HPC-lab3/Q2/Q2.2$ ./pthread_array_sum
499500
zhhb@zhhb:~/HPC-lab3/Q2/Q2.2$ ./pthread_array_sum
499500
zhhb@zhhb:~/HPC-lab3/Q2/Q2.2$
```

可以看到第二部分，一次性访问 10 个元素的程序也可以正确运行。

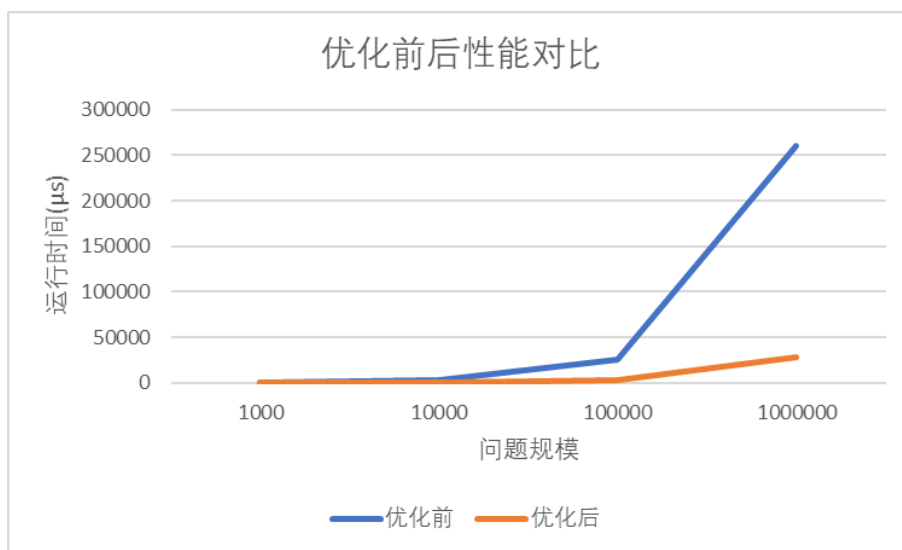
### 3.2.2 程序性能分析

在这里为了突显出优化的效果，我比较了这两种情况下的运行时间。而根据问题 1 观察的结果，我将线程数量设置为 8。在我测试的过程中我发现，当问题规模维持在 1000 的时候，其实二者的运行时间大致相同。为了让优化的效果更加明显，我加大了问题的规模，在每种情况下我都运行 5 次，各种情况下运行的平均时间如下：

问题规模 n	1000	10000	100000	1000000
运行前后				
优化前	351 $\mu s$	2572 $\mu s$	26045 $\mu s$	260217 $\mu s$
优化后	344 $\mu s$	393 $\mu s$	3303 $\mu s$	28313 $\mu s$

表 2: 优化前和优化后的平均运行时间





通过比较不难看出，在问题规模增大的时候，优化效果十分明显。

### 3.3 Pthreads 求解二次方程组的根

在这里运行 make，然后根据我给出的提示输入 a,b,c。在这里我解了几个方程，如下：

```

Q3 — -zsh — 80x24
Last login: Tue Oct 20 22:06:01 on ttys000
zhib@zhanghb-MacBook-Pro ~ % cd Desktop/HPC-lab3/Q3
zhib@zhanghb-MacBook-Pro Q3 % make
gcc-9 -w -o pthread_equation pthread_equation.c
zhib@zhanghb-MacBook-Pro Q3 % ./pthread_equation
Please input a, b, c.
1 -2 1
x1 = x2 = 1.000000
zhib@zhanghb-MacBook-Pro Q3 % ./pthread_equation
Please input a, b, c.
1 -7 12
x1 = 4.000000, x2 = 3.000000
zhib@zhanghb-MacBook-Pro Q3 % ./pthread_equation
Please input a, b, c.
1 -11 30
x1 = 6.000000, x2 = 5.000000
zhib@zhanghb-MacBook-Pro Q3 % ./pthread_equation
Please input a, b, c.
1 -15 56
x1 = 8.000000, x2 = 7.000000
zhib@zhanghb-MacBook-Pro Q3 %

```

第一个样例即： $x^2 + 2 * x + 1 = 0$  的解是  $x_1 = x_2 = -1$ 。后面的例子以此类推。

### 3.4 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

用数学的方法计算所求面积  $S = \int_0^1 x^2 = \frac{1}{3} = 0.333333333...$ , 所以我们结果的期望是  $\frac{1}{3}$ 。

在这里我修改了接口, 可以在运行命令的时候指定随机点的总数量。执行 makefile 后, 可以用 `./pthread_Monte_Carlo {问题规模}` 来指定随机点的数量。如下:

```
zhib@zhanghb-MacBook-Pro Q4 % make
g++-9 -w -o pthread_Monte_Carlo pthread_Monte_Carlo.cpp -lpthread -fpermissive
zhib@zhanghb-MacBook-Pro Q4 % ./pthread_Monte_Carlo 100
res = 0.260000
zhib@zhanghb-MacBook-Pro Q4 % ./pthread_Monte_Carlo 1000
res = 0.342000
zhib@zhanghb-MacBook-Pro Q4 % ./pthread_Monte_Carlo 10000
res = 0.337300
zhib@zhanghb-MacBook-Pro Q4 % ./pthread_Monte_Carlo 100000
res = 0.334110
zhib@zhanghb-MacBook-Pro Q4 % ./pthread_Monte_Carlo 1000000
res = 0.333946
zhib@zhanghb-MacBook-Pro Q4 % ./pthread_Monte_Carlo 10000000
res = 0.333338
zhib@zhanghb-MacBook-Pro Q4 % ./pthread_Monte_Carlo 100000000
res = 0.333338
zhib@zhanghb-MacBook-Pro Q4 % ./pthread_Monte_Carlo 1000000000
res = 0.333348
zhib@zhanghb-MacBook-Pro Q4 %
```

不难看到, 在问题规模比较大的时候, 结果是比较接近于我们的估计的。

因为我在随机数的种子中加入了与时间相关的变量 `time(NULL) % 100`, 所以我每次运行的结果都会不一样。这样子就可以更加充分地观察在各个随机点的运行精度了。

在这里我在各种问题规模下都运行了 5 次, 对于有  $i$  个点的情况, 用  $x_{ij}$  表示在该情况下第  $j$  次运行的结果, 用  $\delta_i$  来表示该情况下五次运行结果与我们的期望  $\frac{1}{3}$  的方差, 用于观察精度与点的数量的关系, 计算方式如下:

$$\delta_i = \frac{1}{5}(\sum_{j=0}^5(x_{ij} - \frac{1}{3})^2)$$

随机点的总数量 $i$	100	1000	10000	100000	1000000	10000000
方差 $\delta_i$	8.23889e-05	6.68889e-06	1.35265e-05	4.47673e-07	8.08822e-09	1.18128e-09

表 3: 在各个情况下运行 5 次后计算得到的方差

很容易观察到, 当问题规模非常大的时候, 方差会变得非常小。在此时, 误差基本上可以被忽略, 也就是说, 只要我们取的随机点的数量足够多, 就可以计算出正确的结果。

在我做这个题目的过程中，我也仔细查阅了很多相关资料。然后我发现，事实上，Monte-Carlo 方法的误差分析有比较严谨的 $\text{数学推理}$ 。可以证明，Monte-Carlo 方法的误差以随机点总数增加的倍率的平方根减小。换句话说，要让精度提高 10 倍，我们需要将我们取出的随机点的数量增加 100 倍。在实际的实验观察中，我的结果也大致与该结论相符。

## 4 实验感想

- 共享内存编程模型不用考虑发送和接收数据的问题，相比 MPI 的并行会简单不少
- 为了优化性能，尽可能减少临界区的出现，比如在一个可以用锁保护的修改共享变量的操作可以分解为先修改局部变量，再根据局部变量修改共享变量，这样子就减少了互斥发生的次数。
- 可以用 `register` 等关键字来减少不必要的访存。
- 因为 macOS 的系统的问题了解了有名信号量和无名信号量，更加注重代码的可迁移性。
- O3 优化可以大大提升程序的性能，但是要注意用 `volatile` 等关键字，避免出现死锁。虽然提交的代码中没有体现出现这个情况，但是我在做这个实验的时候尝试使用了忙等待并且开启 O3 优化的时候，忘记加 `volatile` 导致程序进入死锁。
- 条件变量的实现和信号量不一样，在实验中因为这个问题让我卡顿了很久。
- monte-carlo 方法看似简单，但是背后的数学原理也十分严谨，也十分有趣。