

中山大学计算机学院本科生实验报告

课程名称：超级计算机组成原理

任课教师：吴迪

年级	2019	专业（方向）	计科（超级计算）
学号	18324034	姓名	林天皓
开始日期	2021.4.01	完成日期	2021.4.11

一、实验题目

请使用 pthread 中的 semaphore 计算 π 的值（参考课件 5 page 46）。

计算 π 的方法可以参考课件中的方法，在参考代码中提供了运行所需的主函数。

请同学们下载附件 homework3.zip 将并行的代码补充完整并完成实验报告

二、实验内容

1 实验原理

本次实验主要内容是使用 pthread 进行并行计算编程。Pthread 全称 POSIX Threads，是定义了创建，操作线程的一套 API。主要包括以下四类函数，线程管理，互斥锁，条件变量等。在使用过程中我们常用的是创建线程，加锁解锁来编写并行程序。

通过阅读实验源代码，本次实验计算 π 的通过公式

$$\pi = 4 * [1 - 1/3 + 1/5 - 1/7 + 1/9 - \dots]$$

为了在一次实验中获得多组数据，这里对 main 函数做了一些修改，通过指定不同的线程数，不同的数据量计算得到的数据和所用的时间。

Step1:设定测试数量，申请空间，初始化信号量

```
#define test_case 12
#define thread_case 8
long long thread_tests[thread_case] = {1, 2, 4, 8, 18, 36, 54, 72};
```

```

long long tests[test_case] = {1e3, 1e4, 1e5, 1e6, 1e7, 1e8, 1e9, 1e10, 2e10, 4e10,
8e10, 2e11};
int main(int argc, char *argv[])
{
    for (int j = thread_case - 1; j >= 0; j--)
    {
        thread_count = thread_tests[j];
        printf("=====start a new thread_count test=====\\n");
        fprintf(stdout, "now process thread_count:%ld\\n", thread_count);
        fprintf(stderr, "now process thread_count:%ld\\n", thread_count);
        for (int i = 0; i < test_case; i++)
        {
            long thread; /* Use long in case of a 64-bit system */
            pthread_t *thread_handles;
            double start, finish, elapsed;
            /* please choose terms 'n', and the threads 'thread_count' here. */
            n = tests[i];
            /* You can also get number of threads from command line */
            //Get_args(argc, argv);
            thread_handles = (pthread_t *)malloc(thread_count * sizeof(pthread_t));
            sem_init(&sem, 0, 1);
            sum = 0.0;

```

该步骤包括为 pthread_t 申请空间，初始化信号量，sum 值。

Step2:创建线程，等待线程结束，统计时间，释放空间

```

GET_TIME(start);
for (thread = 0; thread < thread_count; thread++)
    pthread_create(&thread_handles[thread], NULL,
        Thread_sum, (void *)thread);

for (thread = 0; thread < thread_count; thread++)
    pthread_join(thread_handles[thread], NULL);
GET_TIME(finish);
elapsed = finish - start;

sum = 4.0 * sum;
printf("With n = %lld terms,\\n", n);
printf("    Our estimate of pi = %.15f\\n", sum);
printf("The elapsed time is %e seconds\\n", elapsed);
//GET_TIME(start);
//sum = Serial_pi(n);
//GET_TIME(finish);
//elapsed = finish - start;
//printf("    Single thread est  = %.15f\\n", sum);
//printf("The elapsed time is %e seconds\\n", elapsed);

```

```

printf("                pi = %.15f\n", 4.0 * atan(1.0));
sem_destroy(&sem);
free(thread_handles);

```

通过 GET_TIME 宏获取微妙级的系统时间，然后调用 pthread_creat 创建 thread_count 个线程，调用 pthread_join 函数等待所有线程结束后，记录结束时间。由于我们使用的线程数中已经有 1，我们就不再每次使用串行运行一次节省时间。

Step3:每个线程计算 pi 执行函数

```

void *Thread_sum(void *rank)
{
    long my_rank = (long)rank;
    double my_sum = 0.0;
    /*******/
    long long i;
    long long my_n = n / thread_count;
    long long my_first_i = my_n * my_rank;
    long long my_last_i = my_first_i + my_n;
    //printf("rank %ld first=%lld last=%lld\n",my_rank,my_first_i,my_last_i);
    if (my_first_i % 2 == 0)
    {
        for (i = my_first_i; i < my_last_i; i += 2)
            my_sum += 1.0 / (2 * i + 1);
        for (i = my_first_i + 1; i < my_last_i; i += 2)
            my_sum -= 1.0 / (2 * i + 1);
    }
    else
    {
        for (i = my_first_i; i < my_last_i; i += 2)
            my_sum -= 1.0 / (2 * i + 1);
        for (i = my_first_i + 1; i < my_last_i; i += 2)
            my_sum += 1.0 / (2 * i + 1);
    }
    sem_wait(&sem);
    sum += my_sum;
    sem_post(&sem);
    /*******/
    return NULL;
} /* Thread_sum */

```

先获取线程对应的 rank，通过计算总数 n 和 thread_count 得出每个线程所需要计算的数。接下来计算 my_first_i 和 my_last_i。判断 my_first_i 后通过两个 for 循环累加得到局

部求和 my_sum。最后通过信号量获取全局变量的写入权,将局部变量加入全局 sum 中。

至此,我们完成了对实验代码的编写,下面运行测试。

三、实验结果

测试环境说明:

本次实验使用的 CPU 为 Intel(R) Xeon(R) Gold 6150 CPU @ 2.70GHz 18 核心 36 线程

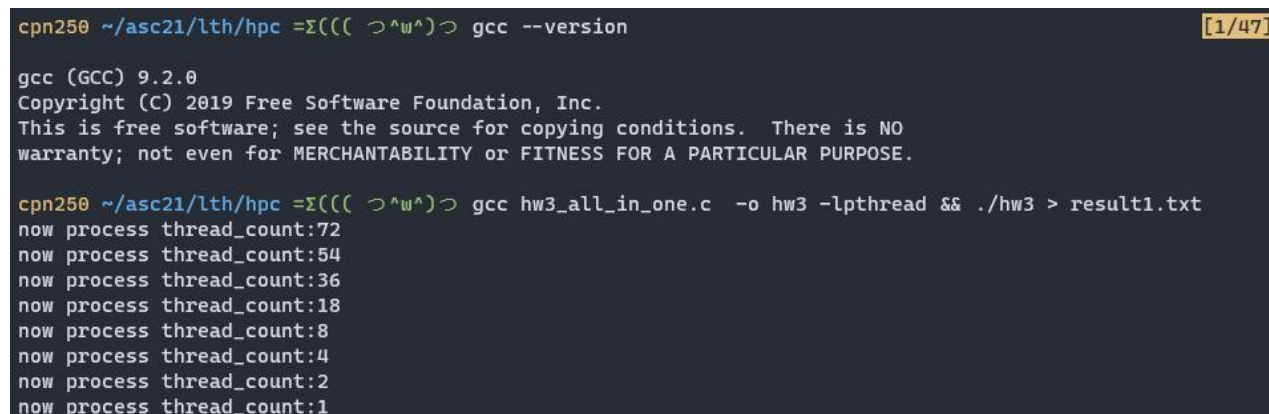
编译器版本为 gcc 9.2.0

本次实验中测试使用了 8 种线程数量,分别为 1, 2, 4, 8, 18, 36, 54, 72.

本次实验中测试使用了 12 种计算数量,分别为 1e3, 1e4, 1e5, 1e6, 1e7, 1e8, 1e9, 1e10, 2e10, 4e10, 8e10, 2e11。

运行测试:

针对上文中的代码,使用 `gcc hw3_all_in_one.c -o hw3 -pthread && ./hw3 >result1.txt` 编译并运行,并将计算结果写入 result1.txt 中。



```
cpn250 ~/asc21/lth/hpc =I((( ㄣ^w^)) gcc --version [1/47]
gcc (GCC) 9.2.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

cpn250 ~/asc21/lth/hpc =I((( ㄣ^w^)) gcc hw3_all_in_one.c -o hw3 -lpthread && ./hw3 > result1.txt
now process thread_count:72
now process thread_count:54
now process thread_count:36
now process thread_count:18
now process thread_count:8
now process thread_count:4
now process thread_count:2
now process thread_count:1
```

图 1-gcc 版本与程序运行

运行中的输出如上,为了方便查看运行状态,我们输出到 stderr 可在终端总查看当前运行进度。运行结束后, `cat result1.txt` 查看运算结果,下图为运算结果节选截图

```

cpn250 ~/asc21/lth/hpc =Σ((( つ^w^)) cat result1.txt
=====start a new thread_count test=====
now process thread_count:72
With n = 1000 terms,
    Our estimate of pi = 3.140524277826286
The elapsed time is 1.744032e-03 seconds
    pi = 3.141592653589793
With n = 10000 terms,
    Our estimate of pi = 3.141492009467665
The elapsed time is 1.499176e-03 seconds
    pi = 3.141592653589793
With n = 100000 terms,
    Our estimate of pi = 3.141582647185670
The elapsed time is 1.461983e-03 seconds
    pi = 3.141592653589793
With n = 1000000 terms,
    Our estimate of pi = 3.141591653525764
The elapsed time is 1.469135e-03 seconds
    pi = 3.141592653589793
With n = 10000000 terms,
    Our estimate of pi = 3.141592553589164
The elapsed time is 2.603054e-03 seconds
    pi = 3.141592653589793
With n = 100000000 terms,
    Our estimate of pi = 3.141592643590398
The elapsed time is 1.921821e-02 seconds

```

图 2-运算结果节选

数据处理:

将数据整理与 excel，并另存为 csv 文件，为了分析获取的时间数据，使用 python 的 matplotlib 库进行绘图展示。使用的绘图代码如下，绘制相应的图需要将对应的代码块取消注释

```

# -*- coding: utf-8 -*-
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
used_time= np.zeros(shape=(8,12))
used_time = pd.read_csv('usetime.csv').to_numpy()
eff = pd.read_csv('eff.csv').to_numpy()
speedup = pd.read_csv('speedup.csv').to_numpy()
com_num=[1000,10000,100000,1000000,10000000,100000000,1000000000,10000000000,20000000000,40000000000,80000000000,200000000000]
thread_num=[1,2,4,8,18,36,54,72]
# ##### show thread ~ time #####
k=12
L=[]
L_TEXT=[]
for i in range(0,12):
    tem_l=plt.plot(thread_num[0:k],used_time[0:k,i])
    L.append(tem_l)

```

```

plt.xlabel('thread num')
plt.ylabel('elapsed time')
L_TEXT.append('n='+str(com_num[i]))

##### show n ~ time #####
# k=12
# L=[]
# L_TEXT=[]
# for i in range(0,8):
#     tem_l=plt.plot(com_num[0:k],used_time[i,0:k])
#     print(used_time[i,0:k])
#     L.append(tem_l)
#     plt.xlabel('n')
#     plt.ylabel('elapsed time')
#     L_TEXT.append('thread_num='+str(thread_num[i]))

##### show n ~ efficiently #####
# k=12
# plt.axes(xscale = "log")
# L=[]
# L_TEXT=[]
# for i in range(0,8):
#     tem_l=plt.plot(com_num[0:k],eff[i,0:k])
#     print(eff[i,0:k])
#     L.append(tem_l)
#     plt.xlabel('n')
#     plt.ylabel('efficiently')
#     L_TEXT.append('thread_num='+str(thread_num[i]))

# ##### show thread ~ efficiently #####
# k=12
# L=[]
# L_TEXT=[]
# for i in range(0,12):
#     tem_l=plt.plot(thread_num[0:k],eff[0:k,i])
#     L.append(tem_l)
#     plt.xlabel('thread numbers')
#     plt.ylabel('efficiently')
#     L_TEXT.append('n='+str(com_num[i]))

##### show n ~ speedup #####
# k=12
# plt.axes(xscale = "log")
# L=[]
# L_TEXT=[]
# for i in range(0,8):

```

```

#     tem_l=plt.plot(com_num[0:k],speedup[i,0:k])
#     print(speedup[i,0:k])
#     L.append(tem_l)
#     plt.xlabel('n')
#     plt.ylabel('speedup')
#     L_TEXT.append('thread_num='+str(thread_num[i]))

# ##### show thread ~ speedup #####
# k=12
# L=[]
# L_TEXT=[]
# for i in range(0,12):
#     tem_l=plt.plot(thread_num[0:k],speedup[0:k,i])
#     L.append(tem_l)
#     plt.xlabel('thread numbers')
#     plt.ylabel('speedup')
#     L_TEXT.append('n='+str(com_num[i]))

plt.legend(L,L_TEXT,loc='upper left',fontsize=8)
plt.show()

```

数据结果与分析：

下表为线程数为 36 时的运算结果

n	Pi
1000	3.140563847277664
10000	3.141492372803838
100000	3.141582650789006
1000000	3.141591653561816
10000000	3.141592553589509
100000000	3.141592643589138
1000000000	3.141592652582249
10000000000	3.141592653479189
20000000000	3.141592653531674
40000000000	3.141592653555964
80000000000	3.141592653571103
200000000000	3.141592653571641

表 1 - pi 值运算结果

运算时间统计:

线程数\运算数量	1000	10000	100000	1000000	10000000	1E+08
1	5.10E-05	7.58E-05	4.99E-04	4.81E-03	4.78E-02	4.78E-01
2	4.70E-05	5.98E-05	2.69E-04	2.43E-03	2.40E-02	2.39E-01
4	7.20E-05	6.51E-05	1.57E-04	1.24E-03	1.20E-02	1.20E-01
8	1.23E-04	9.70E-05	1.33E-04	6.72E-04	6.10E-03	5.99E-02
18	2.06E-04	1.73E-04	1.98E-04	4.47E-04	2.80E-03	2.68E-02
36	7.39E-04	6.19E-04	6.33E-04	7.83E-04	2.88E-03	2.69E-02
54	1.16E-03	1.03E-03	1.02E-03	1.06E-03	2.26E-03	1.81E-02
72	1.81E-03	1.43E-03	1.44E-03	1.40E-03	3.10E-03	2.01E-02

表 2-各线程数各运算数量运行时间统计

线程数\运算数量	1E+09	1E+10	2E+10	4E+10	80000000000	2E+11
1	4.77E+00	4.77E+01	9.53E+01	1.91E+02	3.81E+02	9.54E+02
2	2.39E+00	2.39E+01	4.77E+01	9.55E+01	1.91E+02	4.77E+02
4	1.19E+00	1.19E+01	2.39E+01	4.77E+01	9.55E+01	2.39E+02
8	5.97E-01	5.97E+00	1.19E+01	2.39E+01	4.78E+01	1.19E+02
18	2.67E-01	2.66E+00	5.33E+00	1.06E+01	2.13E+01	5.33E+01
36	1.37E-01	1.37E+00	2.76E+00	5.40E+00	1.08E+01	2.69E+01
54	1.46E-01	1.49E+00	3.07E+00	5.83E+00	1.18E+01	2.90E+01
72	1.52E-01	1.42E+00	2.90E+00	5.83E+00	1.13E+01	2.79E+01

续表 2-各线程数各运算数量运行时间统计

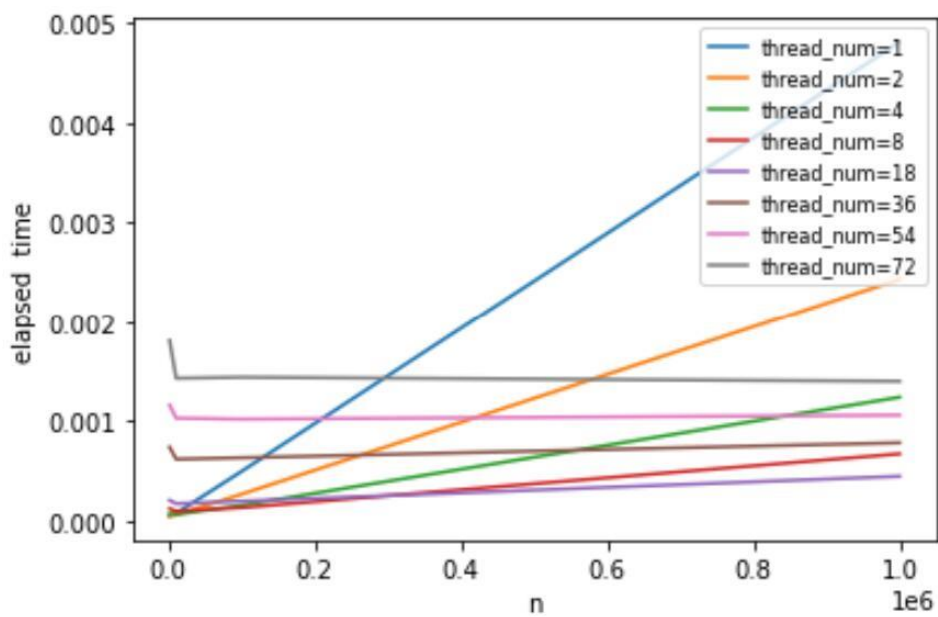


图 3-各线程数运算数量较小时运行时间

在 $n < 10000$ 的情况下，线程数越小速度越快，这是由于线程数越大，最终获取与释放锁

的过程中串行时间越大最终运行时间越大。

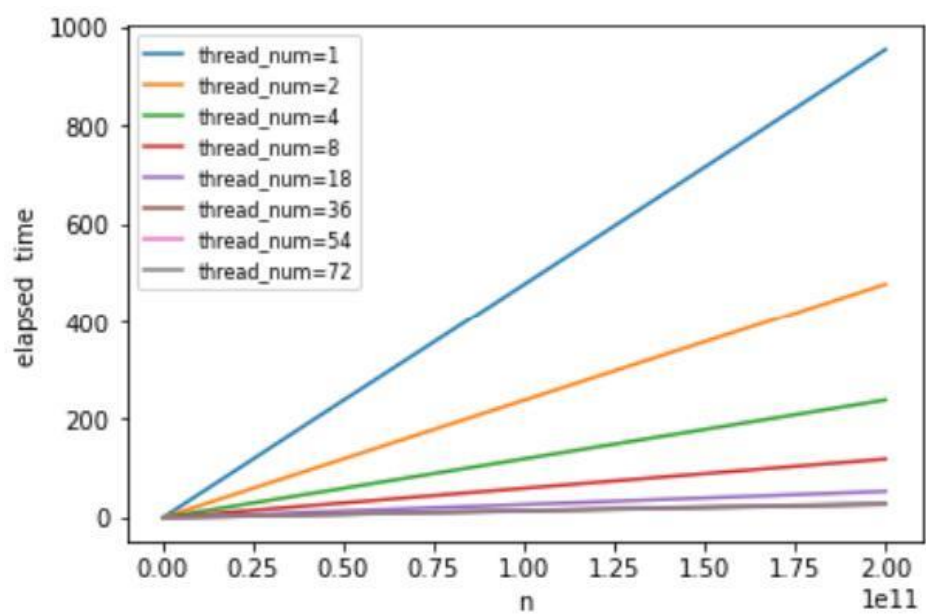


图 4 -各线程数各运算数量运行时间

分析：在 n 足够大时，各个线程数量情况下的运行都近似成线性关系，运行过程中的串行时间随着 n 的增大而减小，运行时间主要受到并行时间影响。

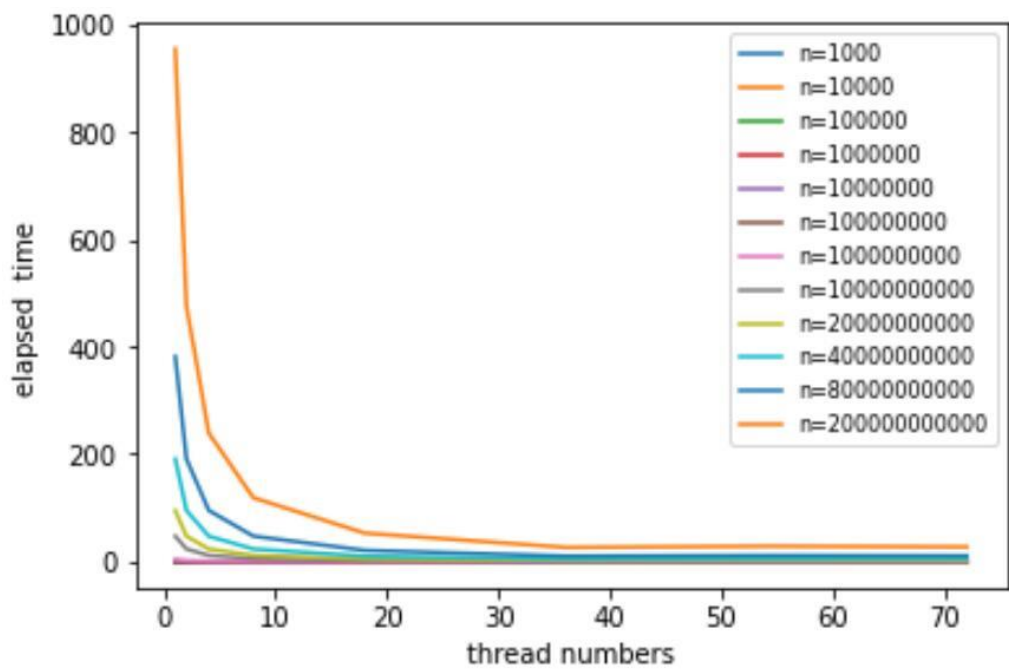


图 5 - 各线程数各运算数量运行时间

分析：由图中观察可得到 n 越大，线程数量越大，总运行时间越小的趋势。

加速比统计

线程数\运算数量	1000	10000	100000	1000000	10000000	1E+08
1	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00
2	1.09E+00	1.27E+00	1.85E+00	1.98E+00	1.99E+00	2.00E+00
4	7.09E-01	1.16E+00	3.17E+00	3.89E+00	3.98E+00	3.99E+00
8	4.15E-01	7.81E-01	3.75E+00	7.16E+00	7.82E+00	7.97E+00
18	2.48E-01	4.39E-01	2.52E+00	1.08E+01	1.70E+01	1.78E+01
36	6.90E-02	1.22E-01	7.88E-01	6.14E+00	1.66E+01	1.78E+01
54	4.39E-02	7.38E-02	4.88E-01	4.54E+00	2.11E+01	2.63E+01
72	2.82E-02	5.31E-02	3.48E-01	3.44E+00	1.54E+01	2.38E+01

表 3- 各线程数各运算数量加速比

线程数\运算数量	1E+09	1E+10	2E+10	4E+10	80000000000	2E+11
1	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00
2	2.00E+00	1.99E+00	2.00E+00	2.00E+00	2.00E+00	2.00E+00
4	3.99E+00	4.00E+00	3.99E+00	3.99E+00	4.00E+00	3.99E+00
8	7.99E+00	7.99E+00	7.98E+00	7.99E+00	7.98E+00	7.99E+00
18	1.79E+01	1.79E+01	1.79E+01	1.80E+01	1.79E+01	1.79E+01
36	3.49E+01	3.49E+01	3.46E+01	3.53E+01	3.52E+01	3.54E+01
54	3.26E+01	3.21E+01	3.10E+01	3.27E+01	3.22E+01	3.29E+01
72	3.15E+01	3.36E+01	3.29E+01	3.27E+01	3.37E+01	3.41E+01

续表 3-各线程数各运算数量加速比

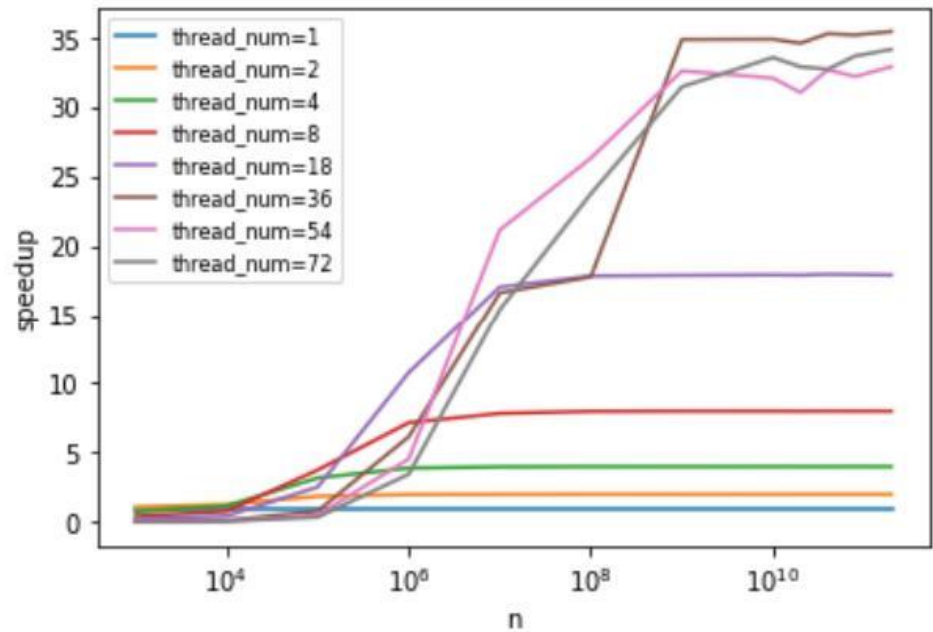


图 6-各线程数各运算数量加速比

分析：由图可得当线程数量没有超过 CPU 线程数量时候，线程数越多，加速比越大。

对于同一线程数，n 越大，加速比越大。

对于同一线程数量，当 n 足够大时，加速比约等于线程数时不再增加。

当线程数量大于 CPU 线程数量时，加速比也不会超过 CPU 线程数量。

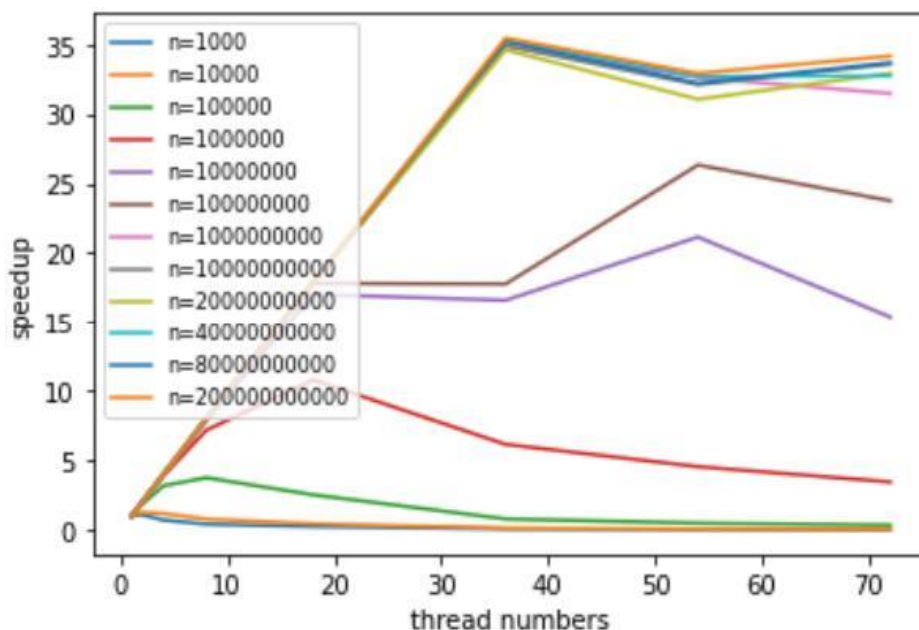


图 7 - 各运算量各运算数量加速比

分析：由图可见当 n 一定时，加速比随着线程数先增加后减少，对于每一个 n 都有一个最优的线程数量，这个数量随着 n 的增加而增加。超过这个线程数量以后，因为过多线程调用的开销，会导致加速比下降。

效率统计：

线程数\运算数量	1000	10000	100000	1000000	10000000	1E+08
1	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00
2	5.43E-01	6.33E-01	9.27E-01	9.89E-01	9.94E-01	1.00E+00
4	1.77E-01	2.91E-01	7.94E-01	9.71E-01	9.94E-01	9.99E-01
8	5.18E-02	9.77E-02	4.69E-01	8.95E-01	9.78E-01	9.97E-01
18	1.38E-02	2.44E-02	1.40E-01	5.98E-01	9.46E-01	9.90E-01
36	1.92E-03	3.40E-03	2.19E-02	1.71E-01	4.61E-01	4.94E-01
54	8.14E-04	1.37E-03	9.05E-03	8.41E-02	3.92E-01	4.87E-01
72	3.91E-04	7.38E-04	4.83E-03	4.78E-02	2.14E-01	3.30E-01

表 4 - 各线程数各运算数量效率

线程数\运算数量	1E+09	1E+10	2E+10	4E+10	80000000000	2E+11
1	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00	1.00E+00
2	9.98E-01	9.97E-01	9.98E-01	9.99E-01	9.99E-01	9.99E-01
4	9.98E-01	9.99E-01	9.98E-01	9.98E-01	9.99E-01	9.98E-01
8	9.98E-01	9.98E-01	9.98E-01	9.98E-01	9.98E-01	9.98E-01
18	9.93E-01	9.95E-01	9.94E-01	9.97E-01	9.96E-01	9.94E-01
36	9.68E-01	9.69E-01	9.61E-01	9.80E-01	9.77E-01	9.84E-01
54	6.04E-01	5.94E-01	5.75E-01	6.05E-01	5.96E-01	6.09E-01
72	4.37E-01	4.66E-01	4.57E-01	4.54E-01	4.68E-01	4.74E-01

续表 4 - 各线程数各运算数量效率

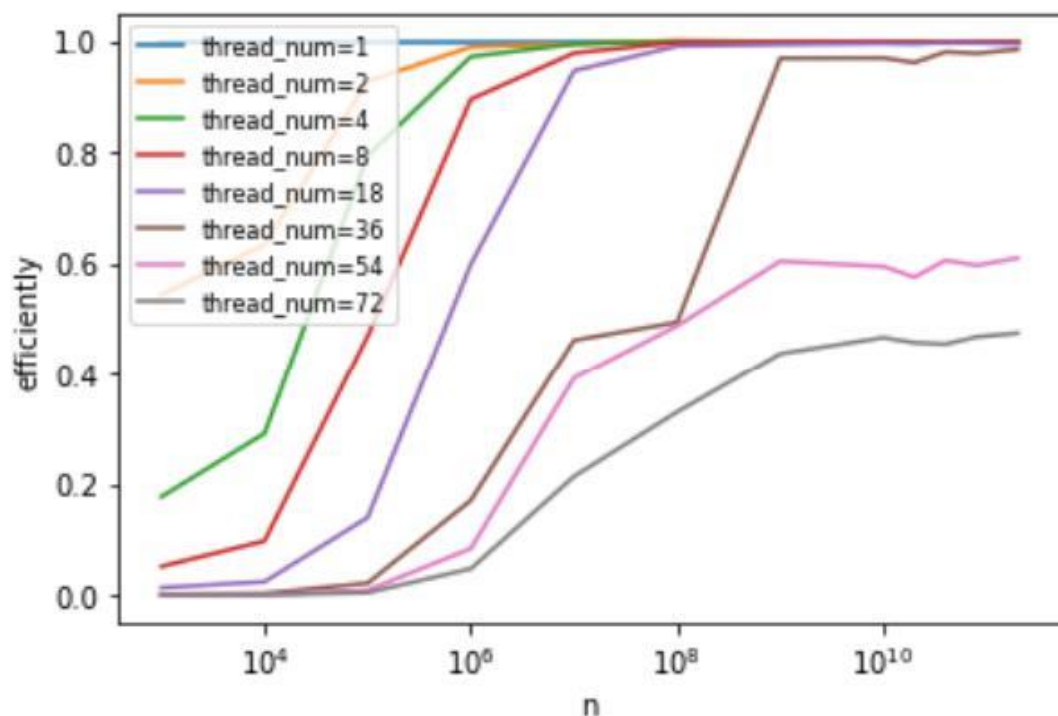


图 8 - 各线程数各运算数量效率

分析：由图可见，当线程数越小时，效率越高。当线程数量小于 CPU 线程数量时，n 足够大时效率近似等于 1.当线程数量大于 CPU 线程数量时，最大效率随着线程数增多而降低。

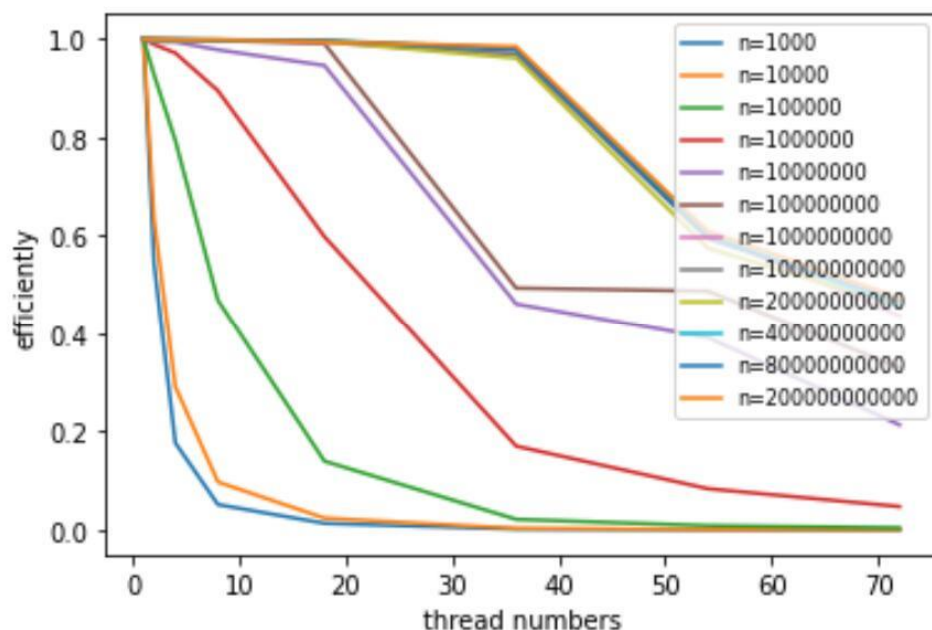


图 9- 运算数量各线程效率

分析：n 越大，效率越高。线程数量越多，效率越低。当线程数量超过 CPU 线程数量时候效率大幅度下降。

实验感想

本次实验是使用 pthread 编程的一次实验，在这次的实验中，学习了使用 pthread_creat 创建多线程程序，使用 pthread_join 完成对线程的等待，同时使用 sem 信号量完成对全局变量的访问添加锁，保证多线程对同一内存的访问不冲突。在基本实验的基础上，进一步测试了其他的实现方法，以及优化运行的代码。

1.使用其他同步方法

除了使用 sem 信号量加锁以外，我们还可以使用 pthread api 中的 mutex 锁或者使用 busy_wait 的方式保证不冲突。代码存放在打包中，此处不再展示。

使用 mutex 方法和信号量对比如下：

使用功能	Mutex 实现	Semaphore 实现
定义锁	pthread_mutex_t mutex;	sem_t sem;
初始化锁	pthread_mutex_init(&mutex, NULL);	sem_init(&sem, 0, 1);
加锁	pthread_mutex_lock(&mutex);	sem_wait(&sem);
解锁	pthread_mutex_unlock(&mutex);	sem_post(&sem);
销毁锁	pthread_mutex_destroy(&mutex);	sem_destroy(&sem);

表 5 -mutex 与 Semaphore 使用锁的差异

使用 busy wait 方法

定义一个 flag 直到判断等于 my_rank 循环，不仅可以保证对 my_sum 的访问不会冲突，还可以保证各个线程加法的顺序。对 sum 加的部分代码如下。

```
while (flag != my_rank); //busywait
sum += my_sum;
flag = (flag + 1) % thread_count;
```

2.误差情况分析

使用上文中先计算奇数情况下的数据，再计算偶数情况下的数据的时候发现，对于 n 较大，线程数较少的情况下，存在相对较大的浮点精度误差

```
With n = 200000000000 terms,
Our estimate of pi = 3.141592653609107
The elapsed time is 5.327290e+01 seconds
pi = 3.141592653589793
```

图 10 - 18 线程 n=2e11 时误差相对较小

```
With n = 200000000000 terms,
Our estimate of pi = 3.141592652226271
The elapsed time is 9.535076e+02 seconds
pi = 3.141592653589793
```

图 11- 单线程 n=2e11 时误差相对较大

上图为 18 线程对比单线程当 $n=2000000000000$ 的情况下的误差，单线程的误差较 18 线程误差大了 10 倍。因此，使用交替进行加减顺序的误差更小。

3.性能优化，使用循环展开

针对其中运算的部分，频繁的判断是否满足跳转的条件造成了时间浪费，所以针对循环的部分，使用循环展开，一次循环做 8 次运算，减少条件判断在指令执行中的占比。编写如下进行循环展开的代码。

```
void *Thread_sum(void *rank)
{
    long my_rank = (long)rank;
    double my_sum = 0.0;
    /*****
    long long i;
    long long my_n = n / thread_count;
    long long my_first_i = my_n * my_rank;
    long long my_last_i = my_first_i + my_n;
    //printf("rank %ld first=%lld last=%lld\n",my_rank,my_first_i,my_last_i);
    double factor;
    if (my_first_i % 2 == 0)
        factor = 1;
    else
        factor = -1;
    for (i = my_first_i; i+7 < my_last_i; i += 8)
    {
        my_sum += factor / (2 * i + 1);
        my_sum -= factor / (2 * i + 3);
        my_sum += factor / (2 * i + 5);
        my_sum -= factor / (2 * i + 7);
        my_sum += factor / (2 * i + 9);
        my_sum -= factor / (2 * i + 11);
        my_sum += factor / (2 * i + 13);
        my_sum -= factor / (2 * i + 15);
    }
    for(;i<my_last_i;i++,factor=-factor){
        my_sum += factor / (2 * i + 1);
    }
    //printf("my_rank=%ld my_sum= %.6lf\n",my_rank,my_sum);
    sem_wait(&sem);
    sum += my_sum;
    sem_post(&sem);
    *****/
}
```

```
return NULL;
} /* Thread_sum */
```

该程序在 $n=200000000000$ 时，各个线程运行时间如下

线程数量	优化前用时/s	循环展开用时/s	用时占比%
1	9.54E+02	8.10E+02	84.9%
2	4.77E+02	4.05E+02	85%
4	2.39E+02	2.03E+02	84.8%
8	1.19E+02	1.01E+02	85.2%
18	5.33E+01	4.52E+01	84.8%
36	2.69E+01	2.30E+01	85.5%
54	2.90E+01	2.51E+01	86.4%
72	2.79E+01	2.39E+01	85.7%

表 6- 环展开前对比循环展开后运行时间

从上表中可见，加入了循环展开之后，我们的程序性能提高了，运行时间减少了约 15%。

```
With n = 200000000000 terms,
Our estimate of pi = 3.141592653583344
The elapsed time is 2.391151e+01 seconds
pi = 3.141592653589793
```

图 12 -修改算法后精度提高

同时使用了原始的交替相加之后，出现的舍入误差更少，计算精度也得到了提升。