**中山大学计算机学院本科生实验报告**

课程名称：超级计算机原理与操作 任课教师：吴迪&黄聃

|  |  |  |  |
| --- | --- | --- | --- |
| 年级 | **2019级** | 专业（方向） | **计算机科学与技术** |
| 学号 | **18329015** | 姓名 | **郝裕玮** |
| 开始日期 | **2021.4.10** | 完成日期 | **2021.4.11** |

1. **实验题目**

使用pthread中的semaphore计算π的值

1. **实验内容**

补充部分代码（即Thread\_sum函数）如下所示（代码分析已全部包含在注释中）：

/\*------------------------------------------------------------------\*/

void\* Thread\_sum(void\* rank) {

   long my\_rank = (long) rank;

   //如果指针类型的大小和表示进程编号的整数类型不同，在编译时就会受到警告

   //在我们使用的机器上，指针类型64位，int类型32位

   //为了避免警告，所以我们用long型替换int型

   double my\_sum = 0.0;//每个线程内部的和（最终要汇总到全局变量sum）

   /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

   double flag;//用于确定公式中每项的正负

   long long i;

   long long group\_num=n/thread\_count;//计算每个线程中需要分配的公式项数（尽可能平均）

   long long group\_first\_i=group\_num\*my\_rank;//计算每个线程中开始累加的第一项

   long long group\_last\_i=group\_first\_i+group\_num;//计算每个线程中需要累加的最后一项

   if(group\_first\_i%2==0){

      flag=1.0;

   }

   else{

      flag=-1.0;

   }

   for(i=group\_first\_i;i<=group\_last\_i-1;i++){//每个线程内部开始累加

      my\_sum+=flag/(2\*i+1);//1/4pi的计算公式

      flag\*=-1;//改变下一项的正负

   }

   sem\_wait(&sem);//阻塞线程，等待获取信号量

   sum+=my\_sum;//获取信号量后可将当前线程的和累加进入全局变量sum

   sem\_post(&sem);//释放信号量

   /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

   return NULL;

}  /\* Thread\_sum \*/

接下来是对其余已完成函数的解析:

宏定义（代码分析已全部包含在注释中）

#define GET\_TIME(now) { \

   struct timeval t; \

   gettimeofday(&t, NULL); \

   now = t.tv\_sec + t.tv\_usec/1000000.0; \

}

//最后一行将us转换为s，统一单位

//该结构体用于计算多线程和串行计算pi的运行时间

全局变量（代码分析已全部包含在注释中）

const int MAX\_THREADS = 1024;//最大线程数

long thread\_count;//线程数量

long long n;//公式项数

double sum;//公式最终的总和（多线程计算pi的估计量）

sem\_t sem;//信号量

主函数（代码分析已全部包含在注释中）

int main(int argc, char\* argv[]) {

   long       thread;  /\* 在64位系统中使用long \*/

   pthread\_t\* thread\_handles;

   //pthread\_t 用于声明线程ID

   double start, finish, elapsed;

   /\* 可选择在这里确定公式项数和线程数量 \*/

   n = 10000;

   thread\_count = 4;

   /\* You can also get number of threads from command line \*/

   //Get\_args(argc, argv);

   thread\_handles = (pthread\_t\*) malloc (thread\_count\*sizeof(pthread\_t));//创建线程数组

   sem\_init(&sem, 0, 1);//初始化信号量

   sum = 0.0;//最终的总和初始化为0

   GET\_TIME(start);//得到多线程运行的开始时间

   for (thread = 0; thread < thread\_count; thread++)

      pthread\_create(&thread\_handles[thread], NULL,Thread\_sum, (void\*)thread);//创建线程

      //第一个参数为指向线程标识符的指针。

      //第二个参数用来设置线程属性。

      //第三个参数是线程运行函数的起始地址。

      //最后一个参数是运行函数的参数。

   for (thread = 0; thread < thread\_count; thread++)

      pthread\_join(thread\_handles[thread], NULL);

      //函数pthread\_join用来等待一个线程的结束。

      //第一个参数为被等待的线程标识符，第二个参数为一个用户定义的指针，它可以用来存储被等待线程的返回值。

      //这个函数是一个线程阻塞的函数，调用它的线程将一直等待到被等待的线程结束为止

      //当函数返回时，被等待线程的资源被收回。

      //也就是说主线程中要是加了这段代码，就会在该代码所处的位置卡住，直到这个线程执行完毕才会继续往下运行。

   GET\_TIME(finish);//得到多线程运行的结束时间

   elapsed = finish - start;//计算多线程运行时间

   sum = 4.0\*sum;//1/4\*pi\*4=pi（公式计算的是1/4\*pi）

   printf("With n = %lld terms,\n", n);//公式项数

   printf("   Our estimate of pi = %.15f\n", sum);//多线程估计的pi值

   printf("The elapsed time is %e seconds\n", elapsed);//多线程运行时间

   GET\_TIME(start);//得到串行计算的开始时间

   sum = Serial\_pi(n);//串行计算pi，也就是单线程

   GET\_TIME(finish);//得到串行计算的结束时间

   elapsed = finish - start;//计算串行计算的运行时间

   printf("   Single thread est  = %.15f\n", sum);//串行计算估计的pi值

   printf("The elapsed time is %e seconds\n", elapsed);//串行计算运行时间

   printf("                   pi = %.15f\n", 4.0\*atan(1.0));//arctan计算pi值

   sem\_destroy(&sem);//释放信号量

   free(thread\_handles);//释放线程数组

   return 0;

}  /\* main \*/

串行计算pi值（代码分析已全部包含在注释中）

/\*------------------------------------------------------------------

 \* Function:   Serial\_pi

 \* Purpose:    Estimate pi using 1 thread

 \* In arg:     n

 \* Return val: Estimate of pi using n terms of Maclaurin series

 \*/

double Serial\_pi(long long n) {

   double sum = 0.0;

   long long i;

   double factor = 1.0;

   for (i = 0; i < n; i++, factor = -factor) {

      sum += factor/(2\*i+1);//按照公式循环累加每一项

   }

   return 4.0\*sum;//乘4后即可得到pi值

}  /\* Serial\_pi \*/

Get\_args和Usage为命令行编译程序并输入公式项数和线程数，与本题关系不大，不再赘述（因为主函数中已有可以直接修改这两项值的语句）

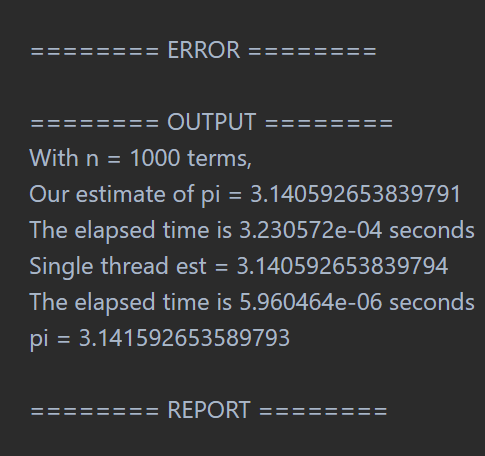
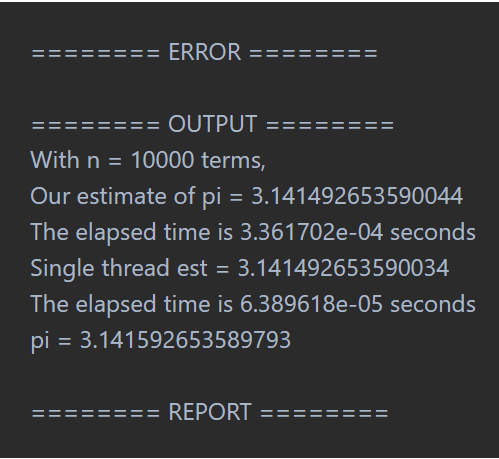
   /\* 可选择在这里确定公式项数和线程数量 \*/

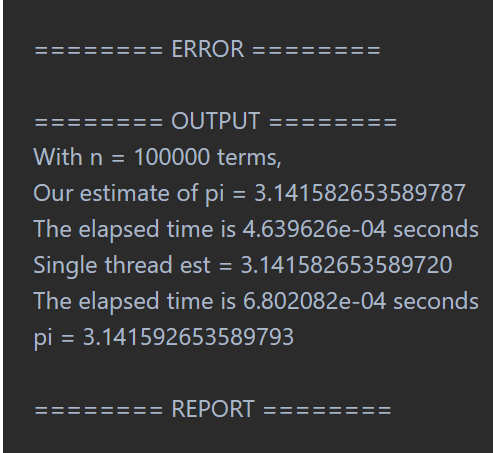
   n = 10000;

   thread\_count = 4;

1. **实验结果**

设置项数分别为1000，10000，100000，线程数不变，均为4：

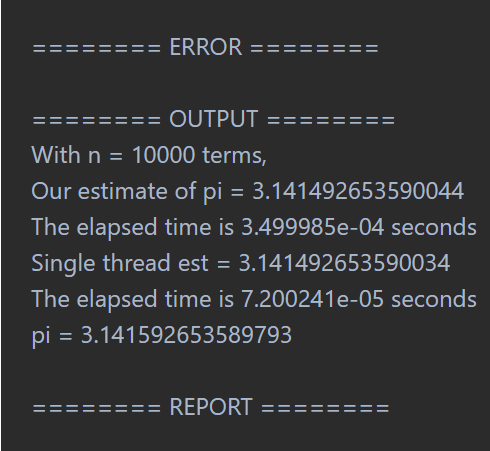
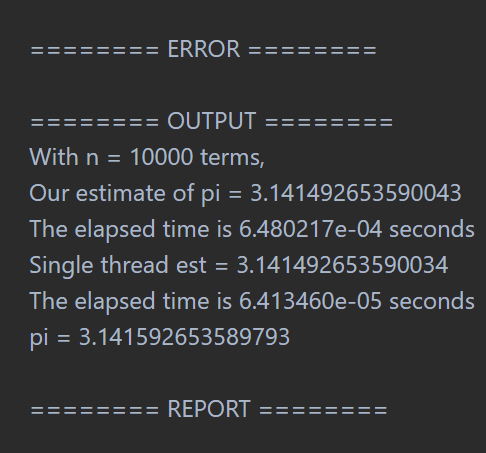
 

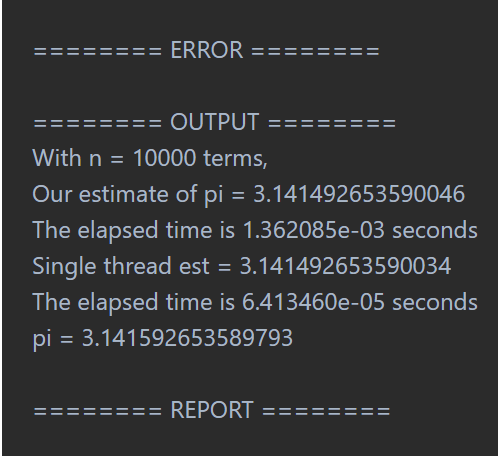


由上述结果可知，项数增加，多线程和串行计算的运行时间均增加，且多线程运行时间的增长速度比串行慢，所以最后多线程运行时间会从比串行慢到比串行快。

且随着项数的增加，多线程和串行计算的估计值也越来越接近pi的真实值（精度越来越高）。

设置线程数分别为4，8，16，项数不变，均为10000：



由上述结果可知，线程数增加，多线程的运行时间增加，串行计算的运行时间基本不变。

同时我们也发现线程数的增加对多线程计算的pi估计值并无影响。