操作系统 课后编程作业

171860607
白晋斌
邮箱 810594956@qq.com

目录

一、实验目的

基于pthread库实现一个多线程程序，采用蒙特卡洛法估算π值（总随机采样次数为10000000次），统计不同的并发线程数量和对应的程序执行时间，结合运行的硬件平台对实验结果进行分析讨论。

二、实验原理

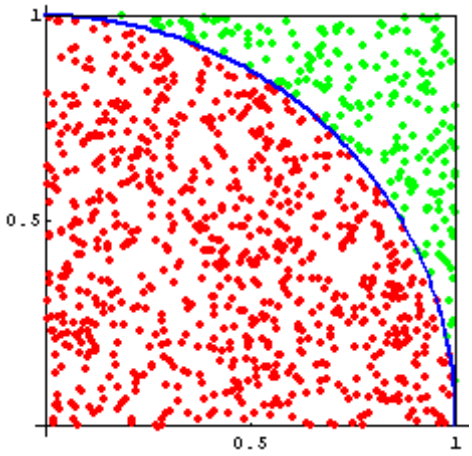
1. 计算原理

在数值积分法中，我们利用求单位圆的1/4的面积来求得Pi/4从而得到Pi。单位圆的1/4面积是一个扇形，它是边长为1单位正方形的一部分，只要能求出扇行面积S1在正方形面积S中占的比例K=S1/S就立即能得到S1，从而得到Pi的值。怎样求出扇形面积在正方形面积中占的比例K呢？一个办法是在正方形中随机投入很多点，使所投的点落在正方形中每一个位置的机会相等看其中有多少个点落在扇形内。将落在扇形内的点数m与所投点的总数n的比m/n作为k的近似值。怎样实现这样的随机投点呢？任何一款计算机语言都有这种功能，能够产生在区间[0，1]内均匀分布的随机数，产生两个这样的随机数x,y,则以(x,y)为坐标的点就是单位正方形内的一点P，它落在正方形内每个位置的机会均等，P落在扇形内的充要条件是x^2+y^2<=1。

设投入的总点数为S1，根据判定条件可计算出落入园内的为S2，则有

$$\frac{\pi}{4} = \mathrm{P} = \frac{\mathrm{S2}}{\mathrm{S1}}$$

由上式即可算出π的值。



2. 并行思想

在上述计算过程中唯一可以并行化的地方是随机数模拟了，假设投入的总点数为kSamplePoints，开启num_threads个线程，这样每个线程平均的去投kSamplePoints/num_threads个点，最后将所有线程的结果求和并计算概率P，这样比串行的效率高很多。

三、实验代码

见附件。

三、统计结果

简单讲如下所示。详细数据见附件excel表格(data.csv，统计了线程数从1到1000，共三次，排除偶然误差)

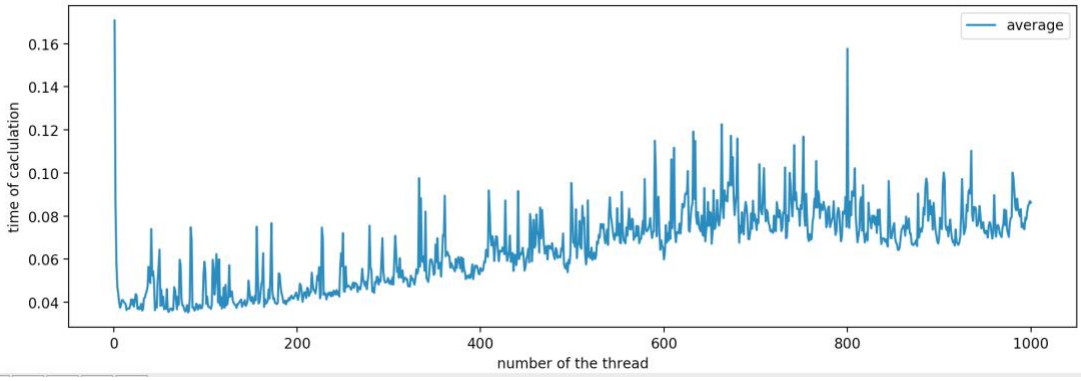
```
parallels@parallels-Parallels-Virtual-Platform:~/Desktop$ gcc mine.c -o mine -lp
thread
parallels@parallels-Parallels-Virtual-Platform:~/Desktop$ ./mine 1000
time:0.167058 s
PI:3.142123
time:0.084466 s
PI:3.141667
time:0.057826 s
PI:3.142306
time:0.046231 s
PI:3.141533
time:0.041678 s
PI:3.141662
time:0.038231 s
PI:3.141440
time:0.034514 s
PI:3.141475
time:0.037136 s
PI:3.141377
time:0.038556 s
PI:3.141247
time:0.036612 s
PI:3.141302
time:0.039647 s
PI:3.141066
time:0.038825 s
PI:3.141528
parallels@parallels-Parallels-Virtual-Platform:~/Desktop$ gcc mine.c -o mine -lp
thread
parallels@parallels-Parallels-Virtual-Platform:~/Desktop$ ./mine 1000
```

接着我们在pycharm中运行如下代码:

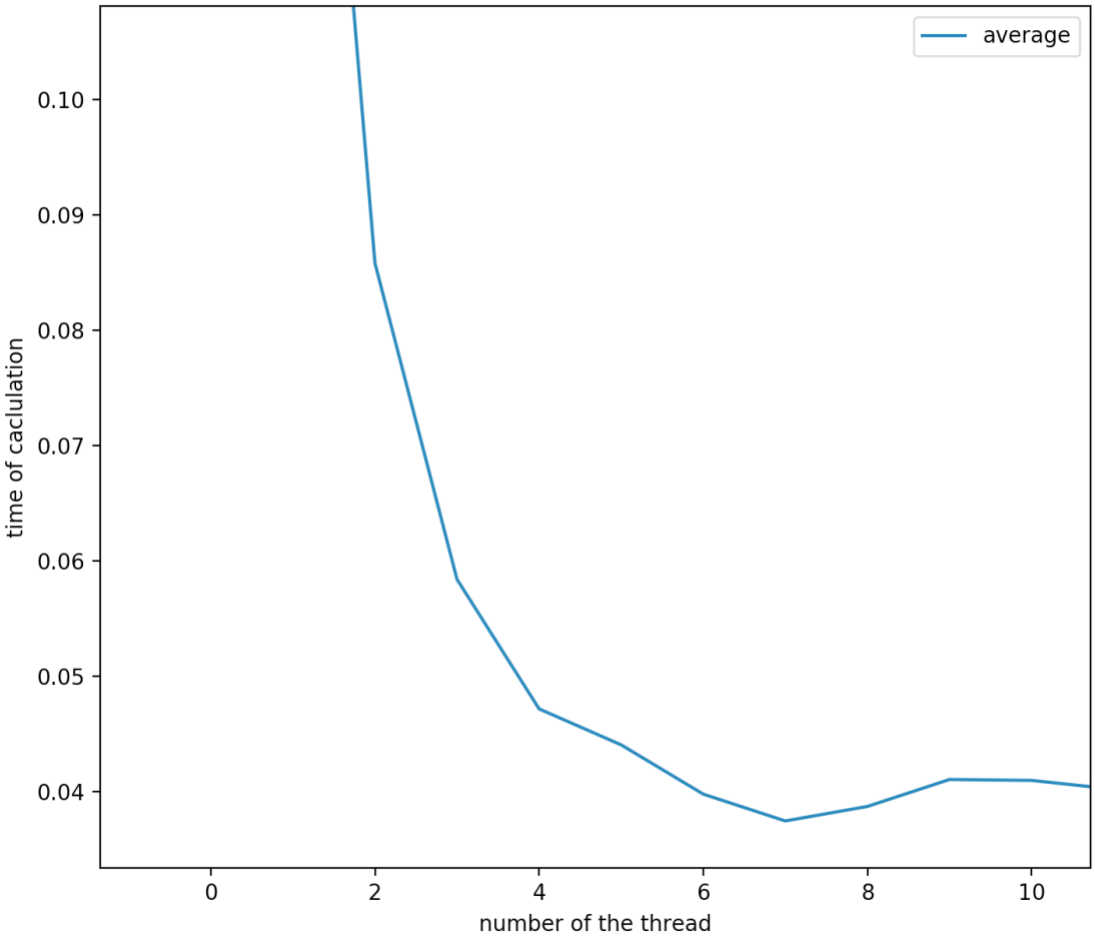
```
import matplotlib.pyplot  as plt
import pandas as pd


plt.figure()
data = pd.read_csv('Y1.csv')
aver=(data['first']+data['second']+data['third'])/3
plt.plot(data['num'], aver,label='average')
plt.xlabel('number of the thread')
plt.ylabel('time of caclulation')
#plt.plot(data['num'],data['data3'],label='first')
#plt.plot(data['num'],data['data4'],label='second')
#plt.plot(data['num'],data['data5'],label='third')
plt.legend()
plt.show()
```

得到如下图片:



对图片进行局部放大，得到如图所示图片。



四、实验结果分析

1.实验结果

由前文图片观察得知，当线程数小于7时，程序运行时间随着线程数的增大而减小；当线程数大于7时，程序运行时间随着线程数的增大而增大。（此时Linux系统cpu为8核core i7）

此外，我们做了许多对比实验，例如，设置Linux虚拟机cpu为2核core i7，此时线程数为2时程序运行时间约为最小。具体原因后文分析。

再例如，当总随机采样次数从10000000次增大到100000000次时，PI的精确度略微上升，程序运行时间大概是之前的十倍，线程数为8时刚好程序运行时间最短。

```
parallels@parallels-Parallels-Virtual-Platform:~/Desktop$ ./mine 50
time:1.650898 s
PI:3.141535
time:0.848244 s
PI:3.141517
time:0.569460 s
PI:3.141435
time:0.433663 s
PI:3.141495
time:0.407740 s
PI:3.141593
time:0.371979 s
PI:3.141516
time:0.334198 s
PI:3.141725
time:0.313750 s
PI:3.141669
time:0.363705 s
PI:3.141843
time:0.338516 s
PI:3.141831
time:0.329967 s
PI:3.141831
time:0.331862 s
PI:3.141725
time:0.322416 s
PI:3.141728
time:0.321070 s
PI:3.141665
time:0.331835 s
PI:3.141607
time:0.320448 s
```

2.结果分析

由前文图片，可以看到很明显的从一个线程变成两个线程确实时间减少了一般左右，但是之后增加线程数量对运行时间的减少越来越不明显，甚至一定数目之后开始增加。这是因为我电脑的CPU型号为Intel
酷睿i7四核八线程，所以八线程的程序已经占用了全部的系统资源了。运行过程中的截图如下图所示。

为了证明这个观点，分别是线程数为1、2、3、4、5、6、7、8时的运行CPU记录，可见八个主线程均已满负荷运作，而继续增加线程时，由于计算内容相同，都是同一种类型的计算，只会相同类型的资源而出现抢占，无法达到超线程的目的，所以并不会真正的减少计算时间。

由此可知，如果需要进行大量相同类型的计算（如计算PI），则只需按核的数量创建线程就能充分利用系统资源了。

3.总结概括

总结概括有以下五点：

1. 对比 1e7 个随机点和 1e8 个随机点的时间曲线，我们发现当数据点越多时 PI 的计算越准确，而且由于计算时间的增大，创建线程所需的时间对总时间的影响越小，从而多线程的优势越发明显。

2. 如果需要进行大量相同类型的计算（如计算 PI），则只需按核的数量创建线程就能充分利用系统资源了。

3.随着线程数的增大，运行时需要开辟更多的内存空间，需要额外消耗一部分时间。

4.本实验中的数据规模小于1e8，故即使是采用int型的最大值CPU都可以在很短的时间内计算完毕，而开启线程所用的时间相对于此问题的计算时间来说不能近似忽略，故采用多线程对速度的提升并不理想。

5.在用一些系统函数如rand，要考虑函数自身是否有同步锁，否则将大大拖慢程序运行速度。

五、代码优化

1.rand_r()

通过阅读gcc标准库中随机函数rand的源代码，可以看到这个函数使用同步锁，强制要求该函数的调用串行，即有多个线程调用此函数，必须等其中一个调用完成后，才能再次调用。更确切的说法是标准库提供的函数如果涉及全局变量，那么为了保证线程调用的安全，都必须串行。这是标准C语言库（设计之初没有考虑支持多线程），扩展到多线程环境中的一个非常简单粗暴的防止运行错误的处理。这也导致了直接使用rand()函数达不到高效运行的原因。

故我们采用rand_r函数替代rand达到相关目的。

2.累加计数

```
pthread_mutex_lock(&mutex);
num_in+=this_num;
pthread_mutex_unlock(&mutex);
```

如果每次落点都对全局变量进行累加的话，要不断的 lock 和 unlock，大大拖慢了并行速度，在这里我们选择线程内部申明一个计数变量，累加至该变量，当函数运行将要结束时，将该计数变量加到全局计数变量上，这样一个线程只需要一次 lock 和 unlock。

3.clock

Linux 环境下似乎 clock 函数调用有问题，在这里我们改用 gettimeofday()函数。效果似乎还可以。

六、心得体会

1. undefined reference to 'pthread_create'

Linux下undefined reference to ‘pthread_create’问题解决。

在使用Linux 线程模块时，使用pthread_create 函数。编译命令为 gcc main.c -o test时，会出现如下错误。

```
/tmp/ccIvH3bU.o: In function `main':
main.c:(.text+0x81): undefined reference to `pthread_create'
collect2: error: ld returned 1 exit status
```

问题的原因：pthread不是linux下的默认的库，也就是在链接的时候，无法找到pthread库中该函数的入口地址，于是链接会失败。

解决：在gcc编译的时候，附加要加 -lpthread参数即可解决。

试用如下命令即可编译通过

```
gcc main.c -o test -lpthread
```

2.感悟总结

通过这次实验我学会了很多内容，以前虽然学习过关于线程、进程的概念，但是并没有真正的使用线程编写代码，利用随机数生成PI的多线程思想很简单，但是从零开始的我花了好久才调通多线程的程序，当运行产生正确的输出结果时，心里十分高兴。随后让我绝望的是，随着线程数的增加，运行时间成倍的上升，这与实际矛盾，最后终于知道是由于在线程中生成随机数的原因，修改代码，并行程序快的多了。

通过编写这个小的多线程程序，对计算机的线程、进程等资源的调度有了更加深刻的认识。通过对实验的串行、并行结果的分析，更加清晰的认识到并行化的重要性。