# OpenMP多线程并行计算的分析

16信科1班 201614790222 赵海博

## 问题重述

使用OpenMP和近似计算公式计算圆周率的大小，并分析串行、并行时间的差别以及问题规模对程序运行时间的影响。

## 的数值计算方法

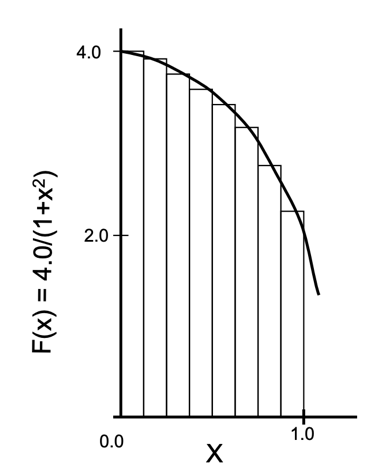
圆周率的每一位、每两位、每三位都符合均匀分布的统计规律。随着计算机的问世以及科学技术发展的需要，的近似值目前精确位数早已突破万亿位[1、2]。计算的过程可以用来检测计算机硬件的可靠性，同时也是用来入门并行计算的最好方式之一。

常用的的数值计算方法包括无穷级数法、蒙特卡罗模拟法与梯形法等，在对于多线程并行方法对的计算过程优化的分析中，为了更大化地体现并行带来的加速比，这里选用算法本身并不最为高效的数值积分方法[3]：

对于一个半径为1的圆在第一象限的曲线积分：

可以得出函数在内的积分为，即：

将积分的值近似为多个矩形的面积和：



由此，的值被近似转化为了N个矩形面积之和。通过数值积分方法即可得到的近似解：

## 单线程计算

### 3.1 迭代次数对计算时间与精度的分析

为了与并行方法进行对比，首先进行单线程计算的测试。测试过程分别取循环次数*num\_steps*为、、。算法[4]及测试结果如下：

*/public/home/201614790222/final\_exam/1(2)-OpenMP-Caculating-PI/serial.c:*

#include <stdio.h>

#include <time.h>

#include <math.h>

typedef long long ll;

#define **realPI** **acos**(-1)

const ll num\_steps = 1e10;

int **main**()

{

**printf**("PI(%.20f) in step\_num=%lld...\n", realPI, num\_steps);

double step, x, pi, sum = 0.0, timetot;

step = 1.0 / (double)num\_steps;

clock\_t start, end;

start = **clock**();

ll I = 0;

for (; i < num\_steps; i++)

{

x = (i + 0.5) \* step;

sum += 4.0 / (1.0 + x \* x);

}

pi = step \* sum;

end = **clock**();

timetot = (double)(end - start) / CLOCKS\_PER\_SEC;

**printf**("Pi=%.20f Running time=%f s\n", pi, timetot);

return 0;

}

计算结果表明：随着循环次数num\_steps的取值（问题规模）增大，计算的精度增加，运算时间增多。

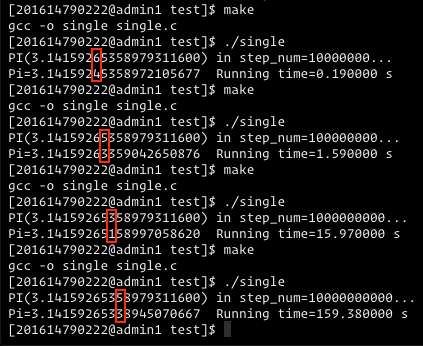


图1 串行程序精度对比过程截图

当循环次数*num\_steps*分别取、、时，计算得到的精度（准确到小数点后位数）对比图如图1所示：

图2 串行模式下精度对比图

### 3.2 串行代码的可并行性分析

串行算法主要耗时部分为数值积分模拟过程的num\_steps次循环：

1 x=0;

2 sum = 0.0;

3 step = 1.0 / (double) num\_steps;

4 for (i=0; i < num\_steps; ++i) {

5 x = (i + 0.5) \* step;

6 sum = sum + 4.0 / (1.0 + x\*x);

7 }

8 pi = step \* sum;

在循环过程（第4-7行）中，算法执行预定数量（num\_steps）次迭代以完成对sum的计算。变量x类似于一个中间变量，它在每次迭代中具有独立的值。在循环结束时，step与sum相乘得到。如果迭代次数num\_steps增加，则的值将更精确。

并行化的最初想法是把num\_steps次循环分配到不同的线程中进行，最后将每个线程得到的sum的值相加得到最终的答案。

## 4 使用OpenMP多线程并行计算

### 4.1 for循环的并行化与多种调度模式

**for循环并行化思想**

假设当前有num个线程（编号为0～num-1）进行num\_steps次迭代任务，我们令第i号线程计算得到的sum存储在sum[i]中，将sum[0]～sum[num-1]的值求和即可得到最终的sum。

在并行过程中需要防止数据竞争的情况，因此在每一个线程中需要将可能产生冲突的遍历指定为私有(private)

**调度模式**

我们可以通过*#pragma omp parallel for* 开辟for循环中的并行域，而我们分配迭代任务的方式有多种。比如一个for循环假设有10次迭代，使用4个线程去执行，那么哪些线程去执行哪些迭代呢？OpenMP的*schedule*提供了控制迭代的调度和分配，从而适应不同的使用情况，提高性能。

schedule的使用格式为：*schedule (type ,size)*,其中，如果没有指定*size*大小，循环迭代会尽可能平均地分配给每个线程；*type*共四种：static、dynamic、guided、runtime：

• static:“静态”体现在这个分配过程跟实际的运行是无关的，可以从逻辑上推断出哪几次迭代会在哪几个线程上运行。具体而言，对于一个N次迭代，使用M个线程，那么，[0,*size*-1]的*size*次的迭代是在第一个线程上运行，[*size*, *size* + *size* -1]是在第二个线程上运行，依次类推。

• dynamic: 较快的线程抢到更多的任务，没有size参数的情况下，每个线程按先执行完先分配的方式执行1次循环；dynamic也可以设置size参数，size表示每次线程执行完（空闲）的时候给其一次分配的迭代的数量

• guided：采用指导性的启发式自调度方式

• runtime: 表示根据环境变量确定上述调度策略中的某一种，默认也是静态的 (static), 控制schedule环境变量的是*OMP\_SCHEDULE*环境变量

在不对for循环指定调度形式时，默认的调度形式为static：

*/public/home/201614790222/final\_exam/1(2)-OpenMP-Caculating-PI/for.c:*

#pragma omp parallel private(i)

{

int id = **omp\_get\_thread\_num**();

sum[id] = 0.0;

double x;

#pragma omp for schedule(static)

for (i = 0; i < num\_steps; i++)

{

x = (i + 0.5) \* step;

sum[id] += 4.0 / (1.0 + x \* x);

}

}

for (i = 0; i < NUM\_THREADS; i++)

pi += sum[i] \* step;

为了对比不同调度形式对于程序运算速度的影响，这里依次对比了static、dynamic、guided三种调度模式下的运算时间。其中，迭代次数*num\_steps*的取值均为，最大线程数(*NUM\_THREADS*)均为20。不同调度模式指导语句与计算耗时如下：

1. #pragma omp for **schedule**(static)：8.471003 s

2. #pragma omp for **schedule**(static, 2)：6.451545 s

3. #pragma omp for **schedule**(static, 3)：4.045087 s

4. #pragma omp for **schedule**(static, 4)：4.503098 s

5. #pragma omp for **schedule**(static, 5)：6.423467 s

6. #pragma omp for **schedule**(static, 50)：15.335034 s

7. #pragma omp for **schedule**(dynamic)：100.356361 s

8. #pragma omp for **schedule**(guided)：3.690092 s

通过对比，我们发现当for循环的任务调度模式为每次给每个线程分配3次迭代时，算法的效率最高。往往可以在其他算法中拥有更高效率的动态调度(dynamic)策略在此处效率并不高，推测可能是由于循环次数过多造成的。

图3 schedule调度策略对比图

### 4.2 使用critical制导语句开辟临界区

在5.1中算法将sum拆分为了多个子数组，其规模是与参与并行运算的线程数相当的。将sum拆分的原因是避免多个线程并行对sum进行操作时产生数据竞争(Data Race)[5]，然而这种处理办法往往是不具有泛性的，这增加了算法的空间复杂度与代码编写的难度。

OpenMP中可以通过在可能产生数据竞争的地方开辟临界区的方式避免数据竞争，用法为：*#pragma omp critical (name)* (name可省略)。如此可以保证每次只有一个线程进入，需要注意的是critical语句不允许互相嵌套。

*/public/home/201614790222/final\_exam/1(2)-OpenMP-Caculating-PI/critical.c:*

x = 0;

sum = 0.0;

step = 1.0 / (double)num\_steps;

#pragma omp parallel private(i, x, aux) shared(sum)

{

#pragma omp for schedule(static)

for (i = 0; i < num\_steps; i = i + 1)

{

x = (i + 0.5) \* step;

aux = 4.0 / (1.0 + x \* x);

#pragma omp critical

sum = sum + aux;

}

}

pi = step \* sum;

上述代码通过使用critical，保证公有变量sum在进行加法操作时，每次只有一个线程进入，从而避免了数据竞争。因为critical临界区中要求操作的次数要尽量小，因此变量*aux*用于减少临界区内的计算。

在实际测试中，添加critical制导指令后的算法运算时间增大，推测是在此类计算简单的情况下，线程等待进入临界区的时间过多。因此，我们得到启发，并行程序可能反而会使程序的运行时间减慢。

### 4.3 使用Reduction进行规约运算

reduction子句用来对一个或多个参数条目指定一个操作符，每个线程将创建参数条目的一个私有拷贝，在并行域或任务分担域的结束处，将用私有拷贝的值通过指定的运行符运算，原始的参数条目被运算结果的值更新。用法：reduction(operator:list)

*/public/home/201614790222/final\_exam/1(2)-OpenMP-Caculating-PI/reduction.c:*

x = 0;

sum = 0.0;

step = 1.0 / (double)num\_steps;

#pragma omp parallel private(i, x)

{

#pragma omp for reduction(+ : sum) schedule(static)

for (i = 0; i < num\_steps; i = i + 1)

{

x = (i + 0.5) \* step;

sum = sum + 4.0 / (1.0 + x \* x);

}

}

pi = step \* sum;

在上述代码中，变量i与变量x在每个线程中均为私有，通过reduction子句，在循环结束后，sum的值降为每个线程sum值的和。

### 4.4 并行加速比对比

为了对比上述几种并行策略的加速比，我们依次选择并行线程数为3、5、7、9、12，num\_steps均为10^8，schedule模式均为static，对比串行程序、使用for制导指令、使用critical制导指令与使用reduction子句的计算速度。运行过程截图如下：

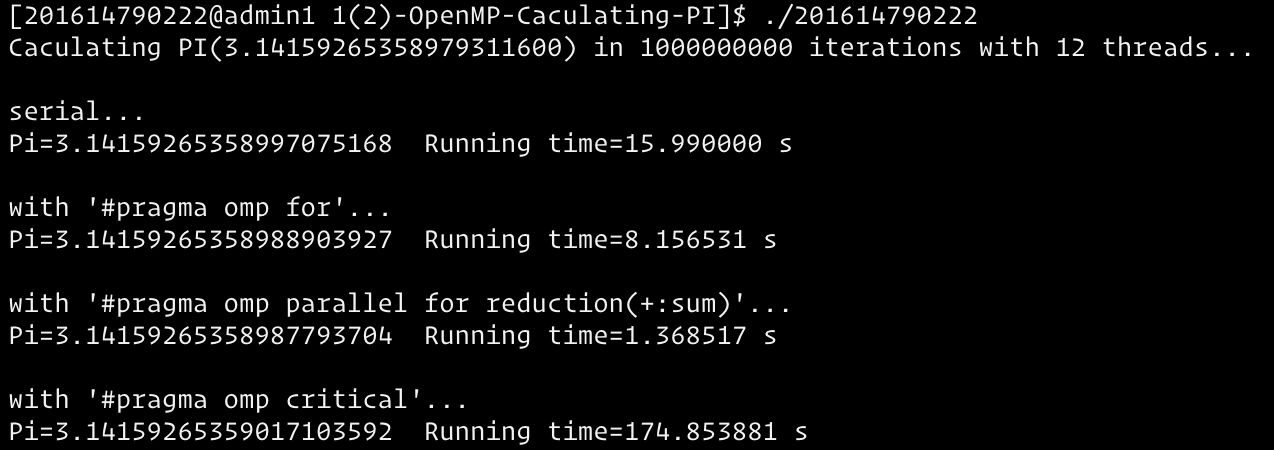


图4 加速比计算过程截图

并行优化理想的加速比是线程的数量，例如，如果应用程序使用4个线程，在这种情况下理想的加速比为4。我们得到的不同并行策略与串行方法和理想情况下的加速比对比图如图5所示：

图5 加速比对比图

分析加速比对比结果，使用reduction子句的for循环最接近理想加速比，而使用了critical制导语句的并行代码加速效果很差，甚至比串行程序减速很多，这是由于在该算法下每个线程的计算任务过于简单，线程等待进入临界区的时间过多造成的。因此，我们在编写并行代码时首先应该分析算法特性，然后再选择合适的并行策略去优化代码。如果选择了不适合的并行方法，并行后的程序运行速度反而可能会减少。

【参考文献】

1. <https://en.wikipedia.org/wiki/Pi>
2. <https://bellard.org/pi/pi2700e9/>
3. [https://extremecomputingtraining.anl.gov//files/2014/01/omp-morning-part1.pdf](https://extremecomputingtraining.anl.gov/files/2014/01/omp-morning-part1.pdf)
4. <https://www.appentra.com/parallel-computation-pi/>
5. <https://scc.ustc.edu.cn/zlsc/sugon/intel/ssadiag_docs/pt_reference/references/sc_omp_anti_dependence.html>