openmp圆周率计算

homework

1. 公式及思路

1.1 无穷级数

使用格雷戈里-莱布尼茨无穷级数，利用以上公式迭代500000次后可得Pi的10位小数：

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} - \frac{1}{15} \cdots$$

同时也等于：

$$\frac{\pi}{2} = \frac{2*2}{1*3} * \frac{4*4}{3*5} * \frac{6*6}{5*7} * \frac{8*8}{7*9} \cdots$$

无穷三次级数：

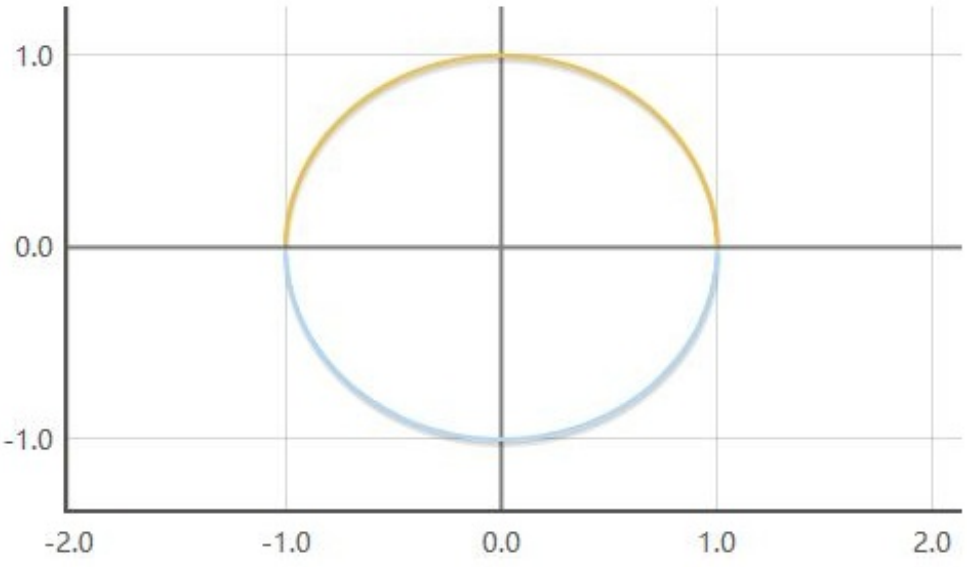
$$\pi = 3 + \sum_{k=0}^{\infty} (-1)^k \frac{4}{(2k+2)(2k+3)(2k+4)}$$

贝利 - 波尔温 - 普劳夫公式：

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right)$$

1.2 蒙洛卡特算法（随机的思想）

> 蒙特·卡罗方法，英文名为Monte Carlo method，也称统计模拟方法，是二十世纪四十年代中期由于科学技术的发展和电子计算机的发明，而被提出的一种以概率统计理论为指导的一类非常重要的数值计算方法。是指使用随机数（或更常见的伪随机数）来解决很多计算问题的方法。与它对应的是确定性算法。蒙特·卡罗方法在金融工程学，宏观经济学，计算物理学（如粒子输运计算、量子热力学计算、空气动力学计算）等领域应用广泛。



首先，我们来建立一个坐标系，以坐标系原点为圆心来画一个单位圆（半径为1的圆）。

这时候，如果我们在和的范围内随机生成点，落在圆内（包括圆上）的概率就等于圆的面积比上圆的外接正方形的面积（红框区域），$P = \frac{S_{圆}}{S_{正方形}} = \frac{\pi}{4}$。

1.3 定积分

> 定积分就是求函数f(X)在区间[a,b]中的图像包围的面积。即由 y=0,x=a,x=b,y=f(X)所围成图形的面积。这个图形称为曲边梯形，特例是曲边三角形。

我们从定积分的计算方法（定义法）入手：

$$\int_a^b f(x)dx = \lim_{n\to\infty} f(x_i)\,\frac{b-a}{n}$$

通过解析几何的知识，平面直角坐标系中的单位圆的方程为： $x^2+y^2=1$

我们可以将图像为半个单位圆（单位圆在x轴上方的部分）的函数表达式写为：
$f(x)=\sqrt{1-x^2}$

由圆的面积公式，我们得到半个单位圆的面积： $S_{半圆}=\frac{1}{2}\pi r^2=\frac{1}{2}\pi$

所以将计算结果乘以2，将会得到圆周率的近似值。

1.4 微积分基本定理

> 牛顿-莱布尼兹公式（Newton-Leibniz formula），通常也被称为微积分基本定理，揭示了定积分与被积函数的原函数或者不定积分之间的联系。
> 牛顿-莱布尼茨公式的内容是一个连续函数在区间 [a，b] 上的定积分等于它的任意一个原函数在区间[a，b]上的增量。牛顿在1666年写的《流数简论》中利用运动学描述了这一公式，1677年,莱布尼茨在一篇手稿中正式提出了这一公式。因为二者最早发现了这一公式，于是命名为牛顿-莱布尼茨公式。
> 牛顿-莱布尼茨公式给定积分提供了一个有效而简便的计算方法，大大简化了定积分的计算过程。

牛顿-莱布尼兹公式的内容是：

$$\int_a^b f(x)dx = F(b)-F(a)$$

其中f(x)为F(x)的导函数。通过计算得到，上述半圆的函数的不定积分为：

$$F(x) = \frac{sin^{(-1)}(x) + x\sqrt{(1-x)} + x\sqrt{(1+x)}}{2}$$

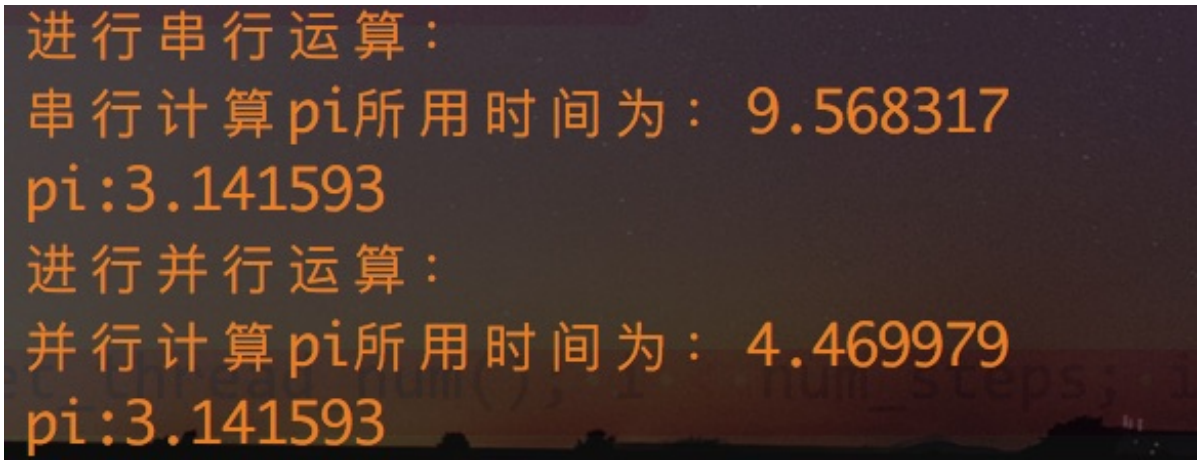
2. 代码实现及结果展示

```
#include <stdio.h>
#include <omp.h>

static long num_steps = 1000000000; //定义所分块数

#define NUM_THREADS 2 //定义所开启的线程数

int main(int argc, char const *argv[])
{
    double start_time, run_time;
    int i = 0;

    //串行-------------------------------------
    printf("进行串行运算：\n");
    double sum = 0.0;
    double step = 1.0 / (double) num_steps;
    double x, pi;

    start_time = omp_get_wtime();
    for (i = 0; i < num_steps; ++i)
    {
        x = (i + 0.5) * step;
        sum += 4.0 / (1.0 + x * x);
    }

    pi = step * sum;
    run_time = omp_get_wtime() - start_time;
    printf("串行计算pi所用时间为：%f\n", run_time);
    printf("pi:%lf \n", pi);

    //并行-------------------------------------
    printf("进行并行运算：\n");
    omp_set_num_threads(NUM_THREADS);//开启线程
    i = 0;
    sum = 0.0;
    x = 0.0, pi = 0.0;
    step = 1.0 / (double) num_steps;
```

```
38.
39.     start_time = omp_get_wtime();
40. #pragma omp parallel sections reduction(+:sum) private(x,i)
41.     {
42.     #pragma omp section
43.         {
44.             for (i = omp_get_thread_num(); i < num_steps; i =
    i+NUM_THREADS)
45.             {
46.                 x = (i + 0.5) * step;
47.                 sum += 4.0 / (1.0 + x * x);
48.             }
49.         }
50.     #pragma omp section
51.         {
52.             for (i = omp_get_thread_num(); i < num_steps; i =
    i+NUM_THREADS)
53.             {
54.                 x = (i + 0.5) * step;
55.                 sum += 4.0 / (1.0 + x * x);
56.             }
57.         }
58.     }
59.     pi = step * sum;
60.     run_time = omp_get_wtime() - start_time;
61.     printf("并行计算pi所用时间为：%f \n", run_time);
62.     printf("pi:%lf \n", pi);
63.
64.
65.     return 0;
66. }
```

结果为：

```
1. 进行串行运算：
2. 串行计算pi所用时间为：9.568317
3. pi:3.141593
4. 进行并行运算：
5. 并行计算pi所用时间为：4.469979
6. pi:3.141593
```



3. 实验总结

利用并行计算对pi的计算进行加速，从实验可以看到加速效果明显，加速比为2.164037，约是之前的两倍。使用并行计算可大副减少运算消耗。