**北京邮电大学软件学院**

**2020-2021学年第一学期实验报告**

**课程名称：** 并行计算

**项目名称：** 圆周率π的计算

**项目完成人：**

**指导教师：** 卢本捷

**日 期： 2020年 11 月 23 日**

1. **实验目的**

学习并行计算的初步方法。

1. **实验内容**

用多种方法完成pi的并行计算

1. **实验环境**
2. 两台或以上的windows 或linux等。
3. 节点的网络互联。
4. Visual studio 2019
5. **实验要求**
6. 记录实验过程，分析试验现象。
7. 记录主要源代码
8. 撰写实验报告.
9. **MPICH 实验步骤**
10. **总体思路**

由于本实验是在并行的条件下使用多种方式计算π的值，所以在程序设计过程中，把每种计算π方法的程序实现封装到一个函数中，然后再使用main()函数依次调用这些函数进行测试与分析。

**main()函数内容如下：**

1. **int** main(**int** argc, **char**\* argv[]) {
2. MPI\_Init(&argc, &argv);
3. MPI\_Comm\_rank(MPI\_COMM\_WORLD, &MyID);
4. MPI\_Comm\_size(MPI\_COMM\_WORLD, &Size);
5. cout << "进程" << MyID << "启动" << endl;
7. **if** (0 == MyID) { StartTime = MPI\_Wtime();   srand((**int**)time(0));}
8. MPI\_Bcast(&N, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);  //将N广播到所有进程中
10. //幂级数积分计算PI值
11. //powerSeries();
13. //蒙特卡洛计算PI值
14. //MonteCarlo();
16. //利用Reduce函数将所有进程的ValueSum累加到root进程(0)的PI变量当中
17. MPI\_Reduce(&ValueSum, &PI, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);
19. **if** (0 == MyID) {
20. EndTime = MPI\_Wtime();
21. auto deviation = PI - Actual\_PI;
22. cout << endl;
23. cout << "计算的PI值为:" << std::setprecision(24) << PI << endl;
24. cout << "实际的PI值为:" << Actual\_PI << endl;
25. cout << "PI的误差为:" << std::setprecision(24) << deviation << endl;
26. cout << "总共用时为:" << EndTime - StartTime << endl;
27. }
28. MPI\_Finalize();
29. **return** 0;
30. }

**main()函数的功能为：**

初始化进程、创建多个进程、主进程使用MPI\_Bcast()函数将计算Pi的次数N广播出去，产生随机数种子；调用使用不同方式实现的计算Pi的函数；使用MPI\_Reduce()函数将每个进程计算的Pi结果累加求和；最后主进程输出计算pi的结果与其他信息；在本次实验中共使用了7种方法实现了Pi的计算。

# 方法一：面积积分

## 问题描述：

对积分进行数值求解即可。

对区间进行划分。简单并行计算。

这是MPICH2中提供的示例方式。

## 函数实现：

采用等宽矩形公式进行积分计算，其中N为积分区间数；rectangleWide = 1.0 / N 为矩形的宽度；

x为积分公式中的x变量；

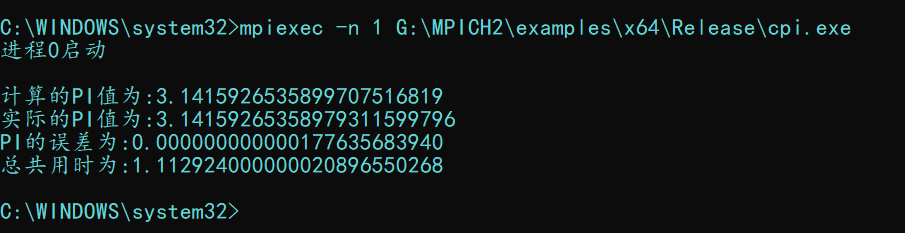
在循环过程中每个进程根据自己的进程ID负责计算一部分pi值，循环时将结果累加求和。

1. //1.面积积分计算PI值
2. **inline** **void** areaIntegral() {
3. **double** X;
4. //计算矩形宽度
5. **double** rectangleWide = 1.0 / (**double**)N;
6. //根据进程ID 计算每个矩形的高度,并求和
7. **for** (**int** i = MyID + 1; i <= N; i += Size) {
8. X = rectangleWide \* ((**double**)i - 0.5);       //第i个矩形x坐标
9. ValueSum = ValueSum + (4 / (1 + X \* X));      //矩形高度累加
10. }
11. //计算总面积 长\*宽
12. ValueSum \*= rectangleWide;
13. }

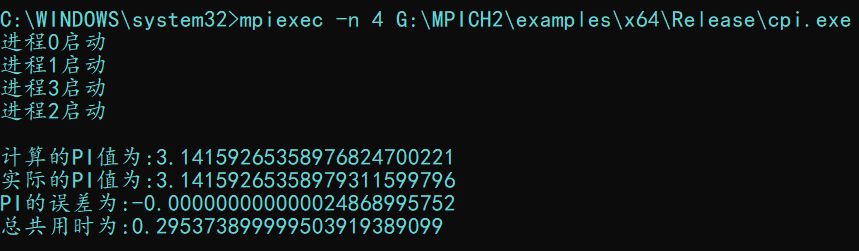
## 运行结果：

**【改变进程数】**

**当N =1×109，进程数为1时**

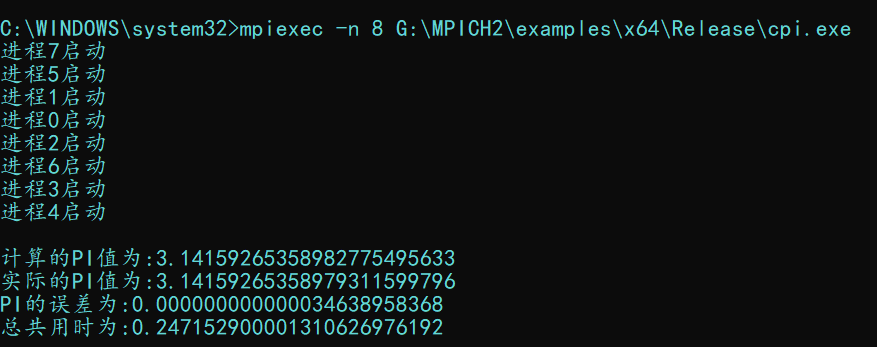


**当N =1×109，进程数为4时**



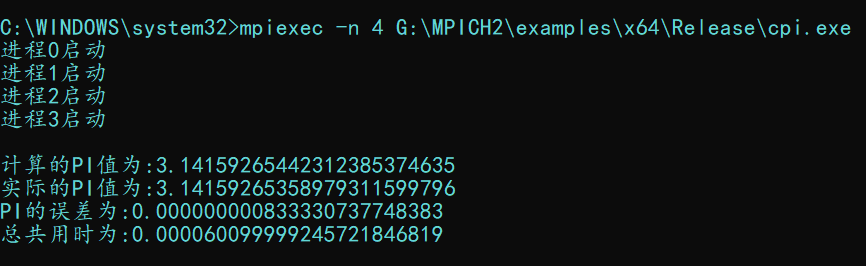
当n = 1000000，进程数为8时

**当N =1×109，进程数为8时**

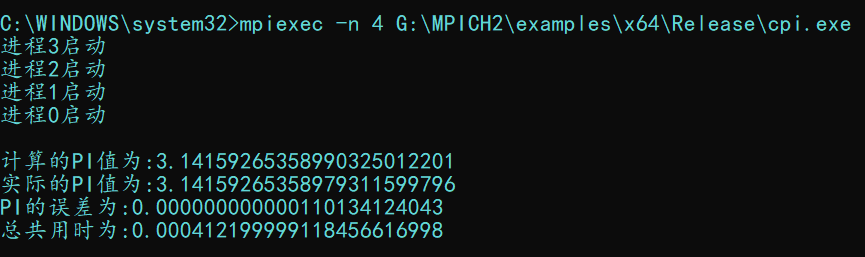


**【改变n的大小】**

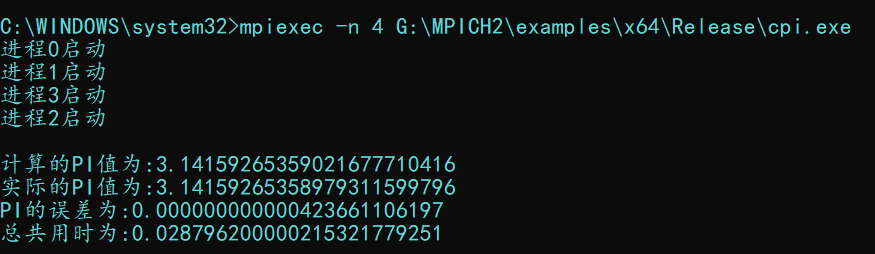
**当N =1×104，进程数为4时**



**当N =1×106，进程数为4时**

****

**当N =1×108，进程数为4时**



## 结果比较：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 测试用例 | N的大小 | 进程数 | 运行时间 | 误差 |
| 1 | **109** | 1 | 1.112924 | 1.7763**×10-13** |
| 2 | **109** | 4 | 0.295373 | 2.4868**×10-14** |
| 3 | **109** | 8 | 0.247152 | 3.4638**×10-14** |
| 4 | **104** | 4 | 0.000060 | 8.3333**×10-10** |
| 5 | **106** | 4 | 0.000412 | 1.1013**×10-13** |
| 6 | **108** | 4 | 0.028796 | 4.2366**×10-14** |

**结果分析：**

1. 当N不变时，进程数翻倍的情况下，运行时间减小，但是每一次减少的时间变少，说明计算的进程数量提高到一定程度时，进程间通信时间占比越高，时间减少幅度放缓。
2. 当进程数不变时，积分区域划分数越多，所用时间越多，误差越小。

# 方法二：幂级数

## 问题描述：

tan（ /4）=1，π/4= arctg（1），对arctan（1）进行幂级数展开即可：

arctgx 计算积分得：argtgx=

所以当x=1 时：

/4= = 1-1/3+1/5-1/7+……+ (-1)n/2n+1

## 函数实现：

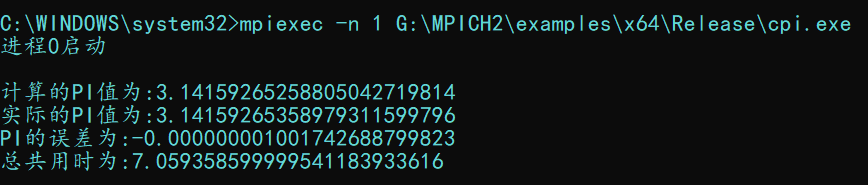
并行计算方法：

1. 事先确定进程数。
2. 主进程确定计算的项数。向各个子进程发送项数。
3. 各个子进程进行自己这部分的累加。
4. 主进程集中。
5. 记录运算时间。
6. **inline** **long** **double** arctg(**int** i, **int** value)
7. {
8. **double** result = pow(-1, (i % 2)) / ((2.0 \* i + 1) \* pow(value, 2 \* i + 1));
9. **return** result;
10. }
12. //2.幂级数(MyID + 1) \* N / Size
13. **inline** **void** powerSeries() {
14. **for** (**long** **double** i = MyID \* N / Size; i < (MyID + 1) \* N / Size; i++)
15. {
16. ValueSum += arctg(i, 1);
17. }
18. ValueSum \*= 4;
19. }

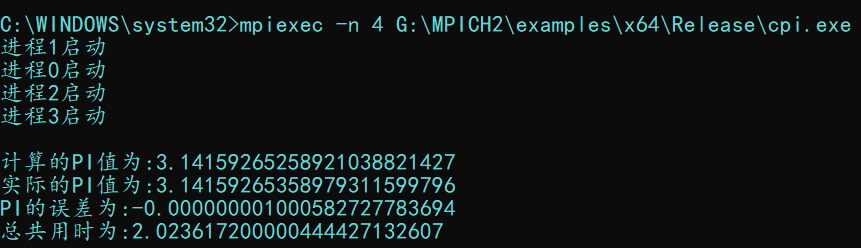
## 运行结果：

**【改变进程数】**

**当N =1×109，进程数为1时**

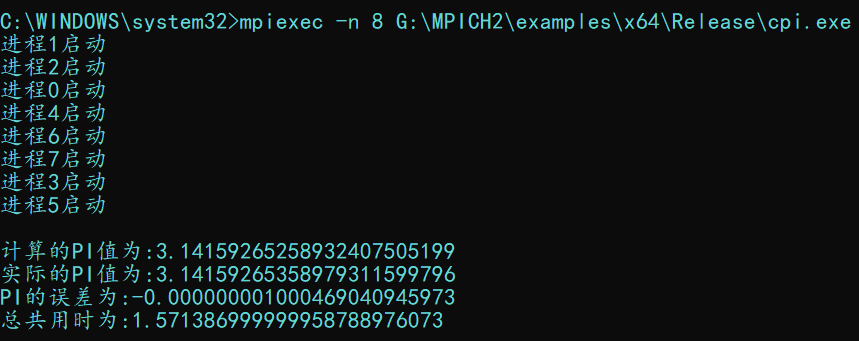


**当N =1×109，进程数为4时**



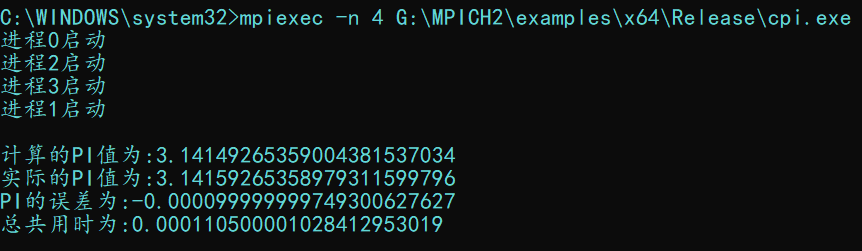
当n = 1000000，进程数为8时

**当N =1×109，进程数为8时**

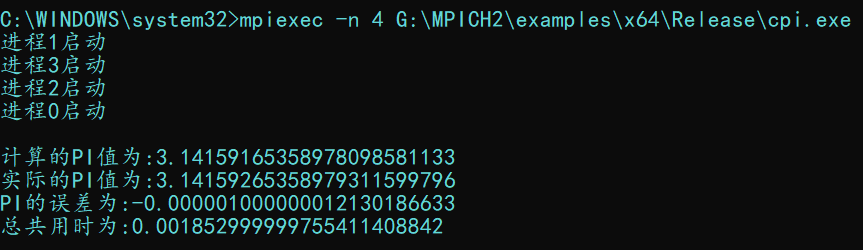


**【改变n的大小】**

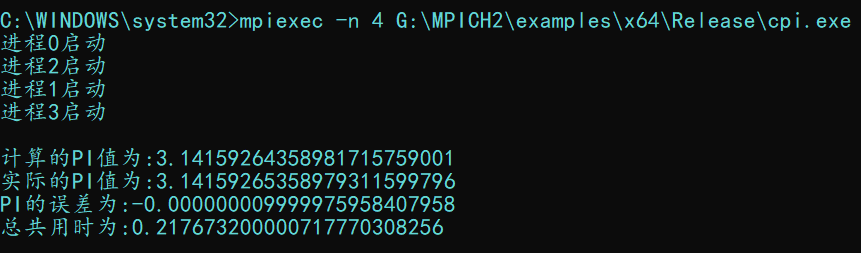
**当N =1×104，进程数为4时**



**当N =1×106，进程数为4时**

****

**当N =1×108，进程数为4时**



## 结果比较：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 测试用例 | N的大小 | 进程数 | 运行时间 | 误差 |
| 1 | **109** | 1 | 7.05935 | -1.0017**×10-9** |
| 2 | **109** | 4 | 2.02361 | -1.0005**×10-9** |
| 3 | **109** | 8 | 1.57138 | -1.0004**×10-9** |
| 4 | **104** | 4 | 0.00011 | -9.9999**×10-4** |
| 5 | **106** | 4 | 0.00185 | -1.0000**×10-6** |
| 6 | **108** | 4 | 0.21767 | -9.9999**×10-9** |

**结果分析：**

1. 当N不变时，进程数翻倍的情况下，运行时间减小，但是每一次减少的时间变少，说明计算的进程数量提高到一定程度时，进程间通信时间占比越高，时间减少幅度放缓。
2. 当进程数不变时，积分区域划分数越多，所用时间越多，误差越小。
3. 与方法一相比，计算精度误差更高。

# 方法三：改进的幂级数

## 问题描述：

以上两种方法收敛很慢。要精确到 大致需要计算2× 项。为提高计算速度采用以下改进的方法：

对于幂级数而言，当x越接近于0时，收敛越快。 上面的例子中，x=1，离0有相当的距离。

令x=1/5. 记=arctan(1/5). tan =1/5.

tan2= 2tan/(1-tan2) = 5/12.

同理tan4 = 120/119. 而tan（ /4）=1，可见 4 与 /4 非常接近。

令 =4- /4

所以tan =tan(4- /4) = （根据 tan(A-B) = (tanA-tanB)/(1+tanAtanB)）

=arctg (1/239).

所以： /4 =4 – = 4 × arctg(1/5)- arctg(1/239)

再利用幂级数展开：=4 -

上述级数收敛的速度非常快。

左边部分：当n=4时，即有1/9×59 < 10-6，而右边收敛更快。

## 函数实现：

并行计算方法：

具体实现方法和方法二差不多，把arctg()的传参值改一下即可。

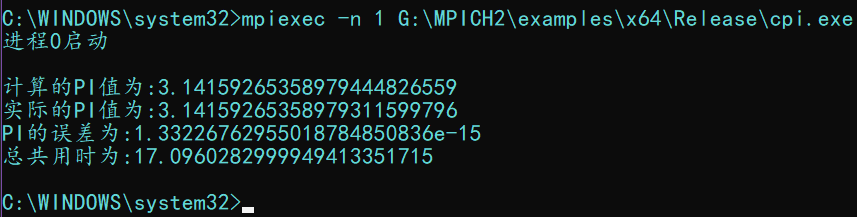
1. **inline** **long** **double** arctg(**int** i, **int** value)
2. {
3. **double** result = pow(-1, (i % 2)) / ((2.0 \* i + 1) \* pow(value, 2 \* i + 1));
4. **return** result;
5. }
7. //3.改进的幂级数
8. **inline** **void** powerSeriesImprove()
9. {
10. **for** (**long** **double** i = MyID \* N / Size; i < (MyID + 1) \* N / Size; i++)
11. ValueSum += 4 \* arctg(i,5) - arctg(i, 239);
13. ValueSum \*= 4;
14. }

## 运行结果：

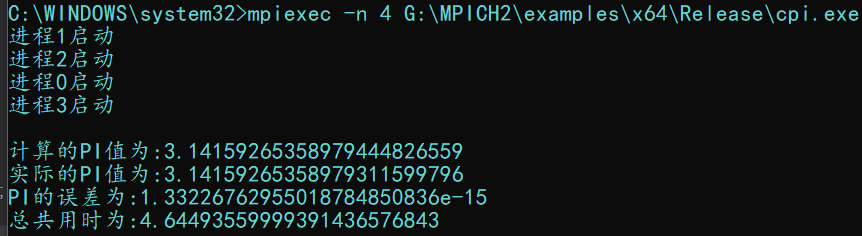
**【改变进程数】**

**由于当N =1×109 时程序计算时间太长，故改为N =1×108**

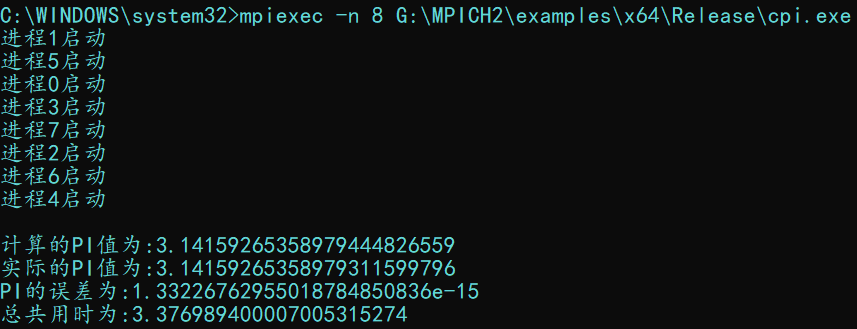
**当N =1×108，进程数为1时**



**当N =1×108，进程数为4时**

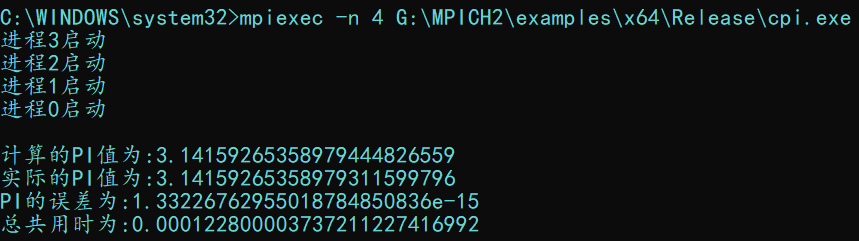


**当N =1×108，进程数为8时**

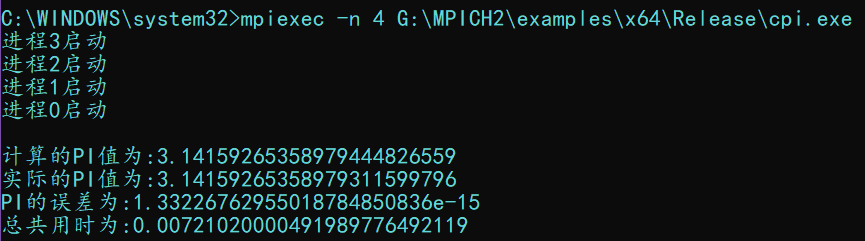


**【改变n的大小】**

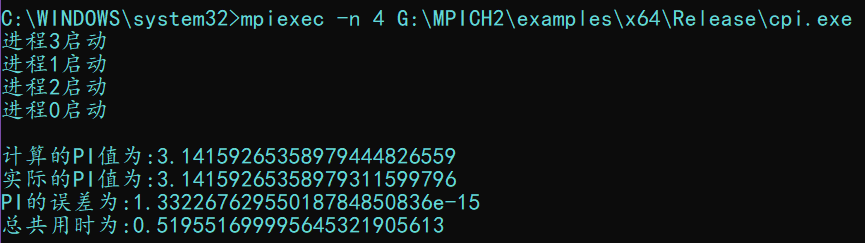
**当N =1×103，进程数为4时**



**当N =1×105，进程数为4时**



**当N =1×107，进程数为4时**



## 结果比较：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 测试用例 | N的大小 | 进程数 | 运行时间 | 误差 |
| 1 | **108** | 1 | 17.0960 | 1.3322**×10-15** |
| 2 | **108** | 4 | 4.64493 | 1.3322**×10-15** |
| 3 | **108** | 8 | 3.37698 | 1.3322**×10-15** |
| 4 | **103** | 4 | 0.00012 | 1.3322**×10-15** |
| 5 | **105** | 4 | 0.00721 | 1.3322**×10-15** |
| 6 | **107** | 4 | 0.51955 | 1.3322**×10-15** |

**结果分析：**

1. 当N不变时，进程数翻倍的情况下，运行时间减小，但是每一次减少的时间变少，说明计算的进程数量提高到一定程度时，进程间通信时间占比越高，时间减少幅度放缓。
2. 当进程数不变时，积分区域划分数越多，所用时间越多，误差越小。
3. 当N大于等于103 时计算误差不再改变，说明计算精度已达到计算机最大值，无法再提高。这也说明改进幂级数的收敛速度明显快于前两种方法。
4. 此外计算的多项数项数越多，计算时间越长。

# 方法四：蒙特卡洛方式

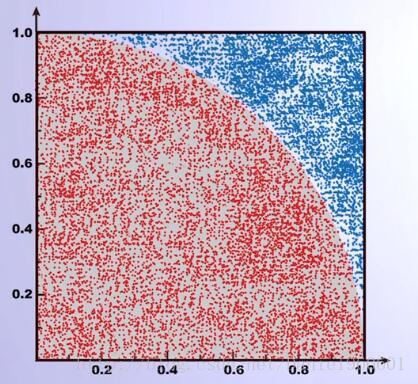
## 问题描述：

1. 各个子进程的工作
2. 使用随机数在正方形内投点
3. 计算落在圆内的点的次数。
4. 计算比值。
5. 主进程收集所有的结果，进行平均。

## 函数实现：

用概率的方式计算pi

构造一个单位正方形和一个单位圆的1/4，往整个区域内随机投入点，根据点到原点的距离判断点是落在1/4的圆内还是在圆外，从而根据落在两个不同区域的点的数目，求出两个区域的比值。如此一来，就可以求出1/4单位圆的面积，从而求出圆周率π，最后再将对个进程计算的结果求和即可。



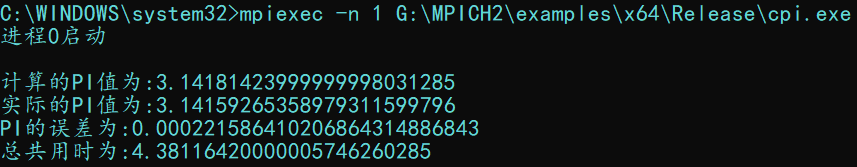
1. //4.蒙特卡洛的方式
2. **inline** **void** MonteCarlo()
3. {
4. **long** **double** randomFrequency = N / Size;
5. **long** **double** count = 0.0;
6. **long** **double** x, y;
8. **for** (**int** i = 0; i < randomFrequency; i++)
9. {
10. x = rand() / **double**(RAND\_MAX);
11. y = rand() / **double**(RAND\_MAX);
12. **if** (sqrt(x \* x + y \* y) <= 1)
13. count++;
14. }
15. ValueSum = 4.0 \* count / (randomFrequency \* Size);
16. }

## 运行结果：

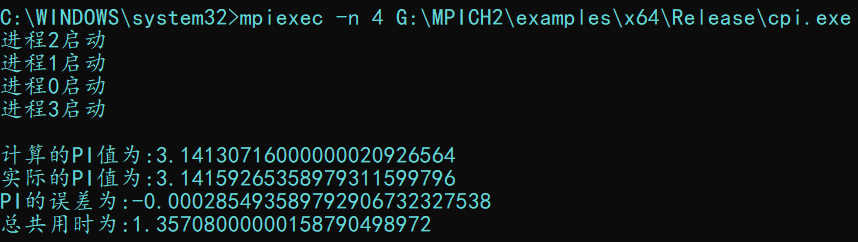
**【改变进程数】**

**由于当N =1×109 时程序计算时间太长，故改为N =1×108**

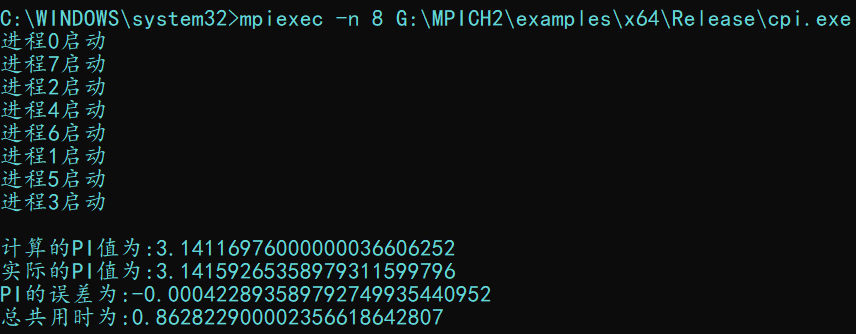
**当N =1×108，进程数为1时**



**当N =1×108，进程数为4时**

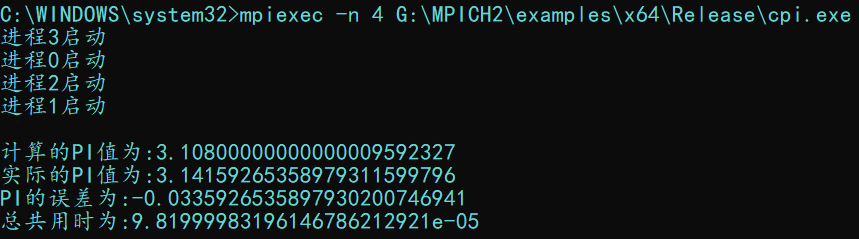


**当N =1×108，进程数为8时**

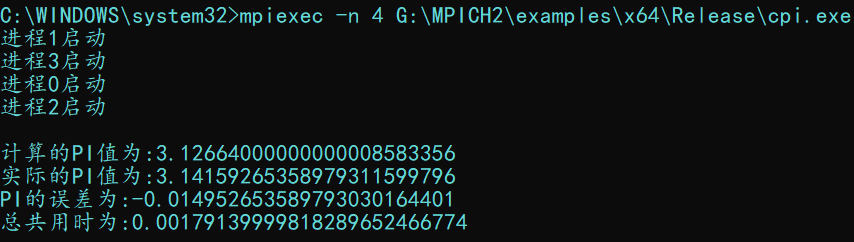


**【改变n的大小】**

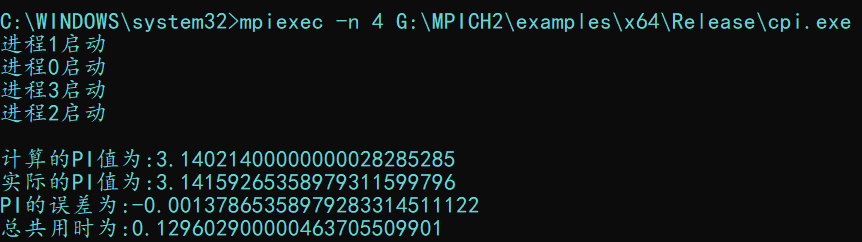
**当N =1×103，进程数为4时**



**当N =1×105，进程数为4时**



**当N =1×107，进程数为4时**



## 结果比较：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 测试用例 | N的大小 | 进程数 | 运行时间 | 误差 |
| 1 | **108** | 1 | 4.38116 | 2.2158**×10-4** |
| 2 | **108** | 4 | 1.35708 | 2.8549**×10-4** |
| 3 | **108** | 8 | 0.86282 | 4.2282**×10-4** |
| 4 | **103** | 4 | 9.819**×10-5** | -3.3359**×10-2** |
| 5 | **105** | 4 | 0.00179 | -1.4952**×10-2** |
| 6 | **107** | 4 | 0.12960 | -1.3786**×10-3** |

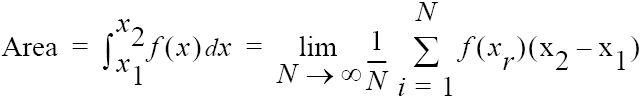
**结果分析：**

1. 使用蒙特卡洛方式计算PI值，其精准度和收敛速度明显低于前三种方法，。
2. 当进程数不变时，随机点的数量越多，所用时间越多，误差越小。
3. 此方法运行时间变化幅度不是很大。

# 方法五：随机积分方式

## 问题描述：

利用公式：



计算

## 函数实现：

将式子转化为上述的形式，计算在0~1之间的随机样本的函数值，最后相加。

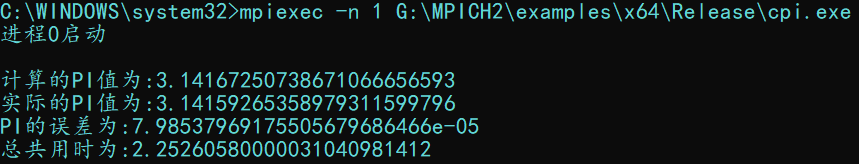
1. //5.随机积分的方法
2. **inline** **void** powerSeriesIm()
3. {
4. **long** **double** randomFrequency = N / Size;
5. **long** **double** X = 0.0;
6. **long** **double** rectangleWide = 1.0 / N;
7. **for** (**int** i = 0; i < randomFrequency; i++)
8. {
9. X = rand() / **double**(RAND\_MAX);  //产生随机数
10. ValueSum = ValueSum + (4.0 / (1.0 + X \* X));//矩形高度累加
11. }
12. // 计算面积 长\*宽
13. ValueSum \*= rectangleWide;
14. }

## 运行结果

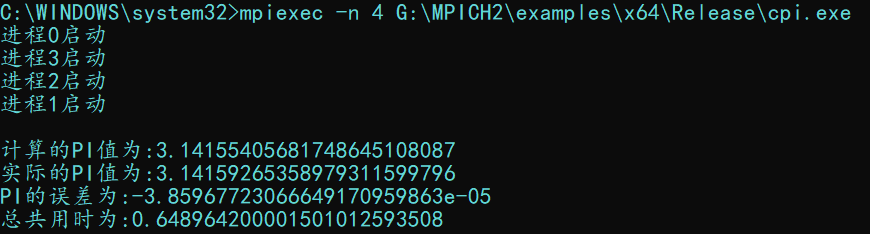
**【改变进程数】**

**由于当N =1×109 时程序计算时间太长，故改为N =1×108**

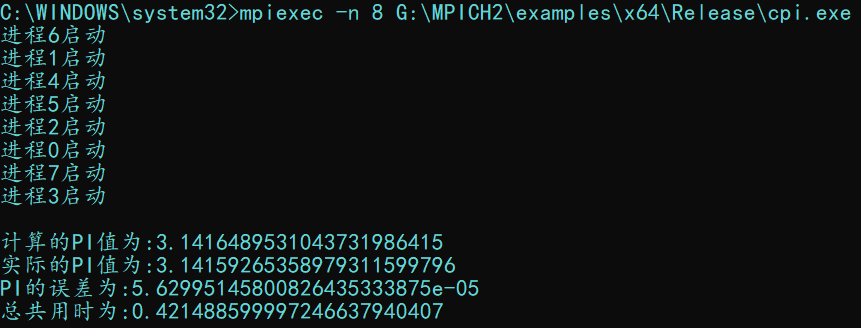
**当N =1×108，进程数为1时**



**当N =1×108，进程数为4时**

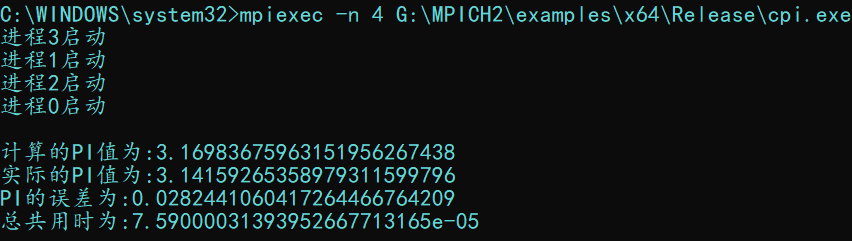


**当N =1×108，进程数为8时**

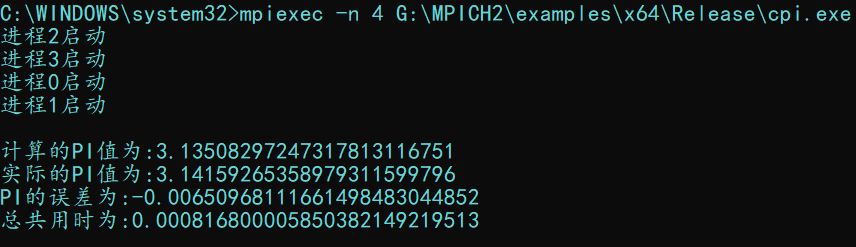


**【改变n的大小】**

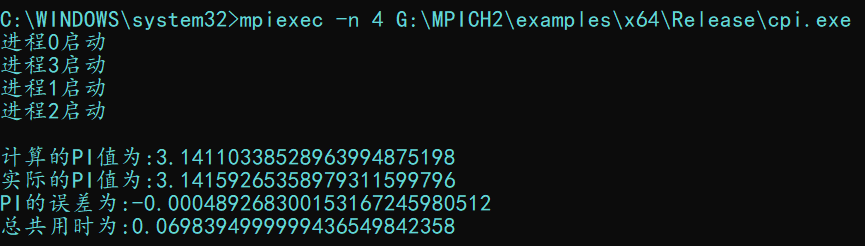
**当N =1×103，进程数为4时**



**当N =1×105，进程数为4时**



**当N =1×107，进程数为4时**



## 结果比较：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 测试用例 | N的大小 | 进程数 | 运行时间 | 误差 |
| 1 | **108** | 1 | 2.25260 | 7.9853**×10-5** |
| 2 | **108** | 4 | 0.6489 | -3.8596**×10-5** |
| 3 | **108** | 8 | 0.4214 | 5.6299**×10-5** |
| 4 | **103** | 4 | 7.59**×10-5** | 2.8244**×10-2** |
| 5 | **105** | 4 | 0.00081 | -6.5096**×10-3** |
| 6 | **107** | 4 | 0.69839 | -4.8926**×10-4** |

**结果分析：**

1. 相比其他方法，这种方法与面积积分相比的精确度差距较大，而且运行时间偏中等。
2. 当n不变，进程数增多时，运行时间会减小。
3. 当进程数不变，产生的随机点越多，误差会减小，运行时间增大。

# 方法六：金田康正公式

## 问题描述：

利用公式：

计算π的值

## 函数实现：

将上述公式转化为代码即可运行。

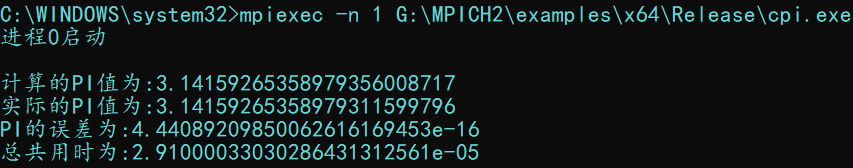
1. //6.金田康正算法
2. **inline** **void** JinTianKangZheng()
3. {
4. **for** (**long** **double** i = MyID \* N / Size; i < (MyID + 1) \* N / Size; i++)
5. {
6. ValueSum += 12 \* arctg((**int**)i, 49) + 32 \* arctg((**int**)i, 57) - 5 \* arctg((**int**)i, 239) + 12 \* arctg((**int**)i, 110443);
7. }
8. ValueSum \*= 4;
9. }

## 运行结果：

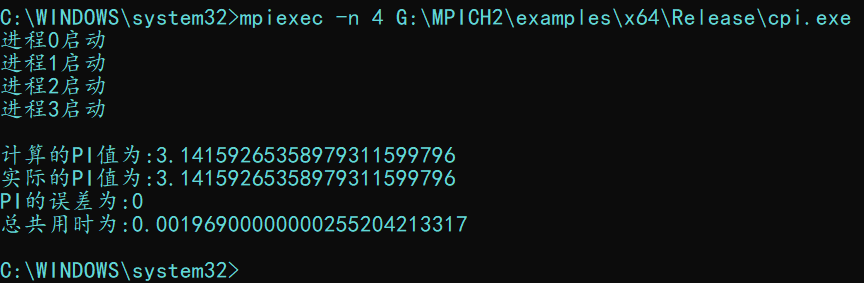
**【改变进程数】**

**由于此方法收敛速度很快，当N=10时，计算的精度就超过了计算机所能计算的范围，故从N =8开始测试**

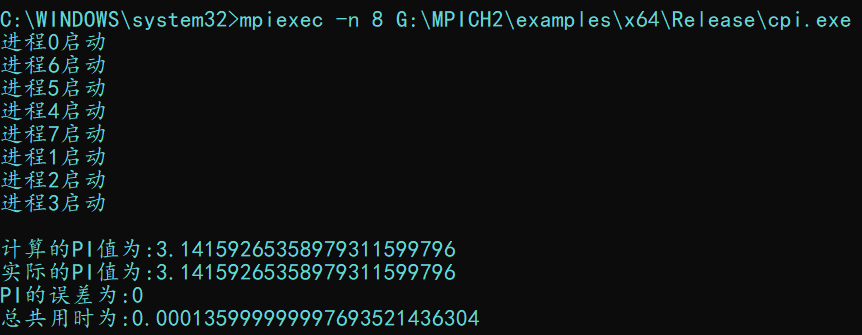
**当N =8，进程数为1时**



**当N =8，进程数为4时**

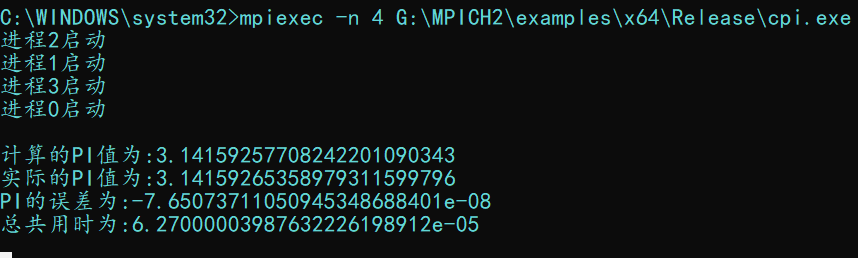


**当N =8，进程数为8时**

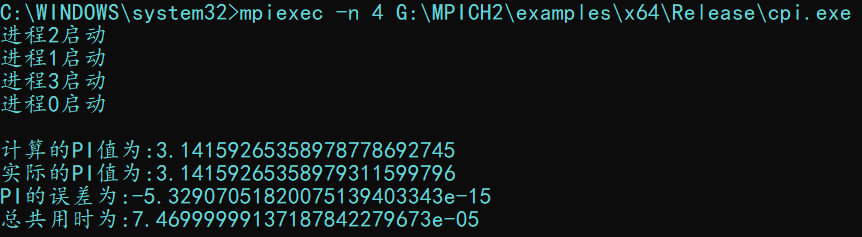


**【改变n的大小】**

**当N =2，进程数为4时**



**当N =4，进程数为4时**



## 结果比较：

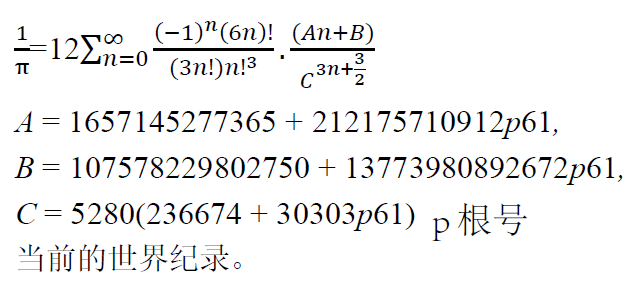
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 测试用例 | N的大小 | 进程数 | 运行时间 | 误差 |
| 1 | **8** | 1 | 2.91**×10-5** | 4.440892**×10-16** |
| 2 | **8** | 4 | 0.00196 | 0 |
| 3 | **8** | 8 | 0.00013 | 0 |
| 4 | **2** | 4 | 6.24**×10-5** | -7.657**×10-8** |
| 5 | **4** | 4 | 7.46**×10-5** | -5.329**×10-15** |

结果分析：

1. 金田康正公式收敛速度超级高，当N>8后计算的精度就超过了计算机所能计算的范围。
2. 此外还出现了误差为0的情况。

# 方法七：Chudnovsky公式

## 问题描述：



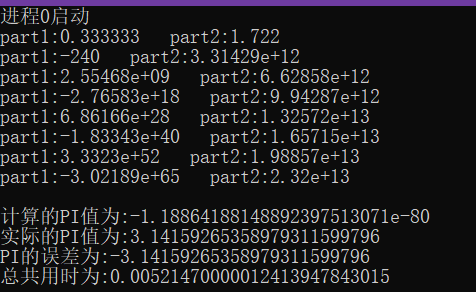
使用此公式进行PI的计算

## 函数实现：

1. //7.Chudnovsky算法
2. **inline** **void** Chudnovsky()
3. {
4. **long** **double** part1 = 0;
5. **long** **double** part2 = 0;
6. **long** **double** P61 = sqrt(61);
7. **long** **double** A = 1657145277365 + 212175710912 \* P61;
8. **long** **double** B = 107578229802750 + 13773980892672 \* P61;
9. **long** **double** C = 5280 \* (236674 + 30303 \* P61);
11. **for** (**int** i = (**int**)(MyID \* N / Size); i < (MyID + 1) \* N / Size; i++)
12. {
13. part1 = pow(-1, (i % 2)) \* fac(6 \* i) / 3.0 \* fac(i) \* pow(fac(i), 3);
14. part2 = A \* i + B / pow(C, 3 \* i + 1.5);
15. cout << "part1:" << part1 << "   part2:" << part2 << endl;
16. ValueSum += part1 \* part2;
17. }
18. ValueSum \*= 12;
19. ValueSum = 1/ValueSum;
20. }

## 运行结果

**由于此公式收敛速度更快，受C++语言精度影响，算出结果往往不准确，故计算结果比较奇怪，无参考性。**



1. **调试心得**

通过本次实验，初步掌握了并行计算的基本思想，并且学会了如何使用mpi库的基础函数，使用多种方法求解pi的值，并且最终将程序运行成功，并对实验结果进行了一系列的分析。

1. **实验源码**
2. #include<math.h>
3. #include<iostream>
4. #include<random>
5. #include <ctime>
6. #include <iomanip>
7. #include "windows.h"
8. #include "mpi.h"
9. **using** std::cout; **using** std::endl;

12. //记录π的值
13. **long** **double** PI = 0.0;
14. //记录每个进程所计算结果的和
15. **long** **double** ValueSum = 0.0;
16. //整个计算PI的开始时间与结束时间
17. **long** **double** StartTime, EndTime;
18. //进程数量,进程ID
19. **int** Size, MyID;
20. //PI计算累加次数,  （可修改用于测试）
21. **long** **double** N = 1e8;
22. //实际PI值
23. **long** **double** Actual\_PI = 3.141592653589793238462643;

26. //1.面积积分计算PI值
27. **inline** **void** areaIntegral() {
28. **long** **double** X;
29. //计算矩形宽度
30. **long** **double** rectangleWide = 1.0 / N;
31. //根据进程ID 计算每个矩形的高度,并求和 myid = 0 分配 第1 5 9 14 ... myid = 2 分配 第2 6 10 15...
32. **for** (**int** i = MyID + 1; i <= N; i += Size) {       //i = myid\* n/size; i<= (myid+1) \* n/size; i++也行.
33. X = rectangleWide \* ((**long** **double**)i - 0.5);       //第i个矩形x坐标
34. ValueSum = ValueSum + (4.0 / (1 + X \* X));      //矩形高度累加
35. }
36. //计算总面积 长\*宽
37. ValueSum \*= rectangleWide;
38. }

41. **inline** **long** **double** arctg(**int** i, **int** value)
42. {
43. **double** result = pow(-1, (i % 2)) / ((2.0 \* i + 1.0) \* pow(value, 2 \* i + 1));
44. **return** result;
45. }
47. //2.幂级数(MyID + 1) \* N / Size
48. **inline** **void** powerSeries() {
49. **for** (**long** **double** i = MyID \* N / Size; i < (MyID + 1) \* N / Size; i++)
50. ValueSum += arctg((**int**)i, 1);
52. ValueSum \*= 4;
53. }

56. //3.改进的幂级数
57. **inline** **void** powerSeriesImprove()
58. {
59. **for** (**long** **double** i = MyID \* N / Size; i < (MyID + 1) \* N / Size; i++)
60. ValueSum += 4 \* arctg((**int**)i,5) - arctg((**int**)i, 239);
62. ValueSum \*= 4;
63. }
65. //4.蒙特卡洛的方式
66. **inline** **void** MonteCarlo()
67. {
68. **long** **double** randomFrequency = N / Size;
69. **long** **double** count = 0.0;
70. **long** **double** x, y;
72. **for** (**int** i = 0; i < randomFrequency; i++)
73. {
74. x = rand() / **double**(RAND\_MAX); //产生随机数0~1
75. y = rand() / **double**(RAND\_MAX);
76. **if** (sqrt(x \* x + y \* y) <= 1)
77. count++;
78. }
79. ValueSum = 4.0 \* count / (randomFrequency \* Size);
80. }
82. //5.随机积分的方法
83. **inline** **void** powerSeriesIm()
84. {
85. **long** **double** randomFrequency = N / Size;
86. **long** **double** X = 0.0;
87. **long** **double** rectangleWide = 1.0 / N;
88. **for** (**int** i = 0; i < randomFrequency; i++)
89. {
90. X = rand() / **double**(RAND\_MAX);  //产生随机数0~1
91. ValueSum = ValueSum + (4.0 / (1.0 + X \* X));//矩形高度累加
92. }
93. // 计算面积 长\*宽
94. ValueSum \*= rectangleWide;
95. }
97. //6.金田康正算法
98. **inline** **void** JinTianKangZheng()
99. {
100. **for** (**long** **double** i = ceil(MyID \* N / Size); i < ceil((MyID + 1) \* N / Size); i++)
101. {
102. ValueSum += 12 \* arctg((**int**)i, 49) + 32 \* arctg((**int**)i, 57) - 5 \* arctg((**int**)i, 239) + 12 \* arctg((**int**)i, 110443);
103. }
104. ValueSum \*= 4;
105. }
107. **inline** **long** **double** fac(**int** n)
108. {
109. **long** **double** sum = 1;
110. **for** (**int** i = 1; i <= n; i++)
111. sum \*= i;
112. **return** sum;
113. }
115. //7.Chudnovsky算法
116. **inline** **void** Chudnovsky()
117. {
118. **long** **double** part1 = 0;
119. **long** **double** part2 = 0;
120. **long** **double** P61 = sqrt(61);
121. **long** **double** A = 1657145277365 + 212175710912 \* P61;
122. **long** **double** B = 107578229802750 + 13773980892672 \* P61;
123. **long** **double** C = 5280 \* (236674 + 30303 \* P61);
125. **for** (**int** i = (**int**)(MyID \* N / Size); i < (MyID + 1) \* N / Size; i++)
126. {
127. part1 = pow(-1, (i % 2)) \* fac(6 \* i) / 3.0 \* fac(i) \* pow(fac(i), 3);
128. part2 = A \* i + B / pow(C, 3 \* i + 1.5);
129. cout << "part1:" << part1 << "   part2:" << part2 << endl;
130. ValueSum += part1 \* part2;
131. }
132. ValueSum \*= 12;
133. ValueSum = 1/ValueSum;
134. }

137. **int** main(**int** argc, **char**\* argv[]) {
138. MPI\_Init(&argc, &argv);
139. MPI\_Comm\_rank(MPI\_COMM\_WORLD, &MyID);
140. MPI\_Comm\_size(MPI\_COMM\_WORLD, &Size);
141. cout << "进程" << MyID << "启动" << endl;
143. **if** (0 == MyID) { StartTime = MPI\_Wtime();   srand((**int**)time(0));}
144. MPI\_Bcast(&N, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);  //将N广播到所有进程中
145. //1.面积积分计算PI值 测试N的范围 1e3~1e7
146. areaIntegral();
148. //2.幂级数积分计算PI值 测试N的范围 1e3~1e7
149. //powerSeries();
151. //3.改进幂级数 测试N的范围 1e3~1e7
152. //powerSeriesImprove()
154. //4.蒙特卡洛计算PI值 测试N的范围 1e3~1e7
155. //MonteCarlo();
157. //5.随机积分的方法 测试N的范围 1e3~1e7
158. //powerSeriesIm();
160. //6.金田康正算法 测试N的范围 2~10
161. //JinTianKangZheng()
163. //7.Chudnovsky算法 测试N的范围 1~5
164. //Chudnovsky();
166. //利用Reduce函数将所有进程的ValueSum累加到root进程(0)的PI变量当中
167. MPI\_Reduce(&ValueSum, &PI, 1, MPI\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

170. **if** (0 == MyID) {
171. EndTime = MPI\_Wtime();
172. auto deviation = PI - Actual\_PI;
173. cout << endl;
174. cout << "计算的PI值为:" << std::setprecision(24) << PI << endl;
175. cout << "实际的PI值为:" << Actual\_PI << endl;
176. cout << "PI的误差为:" << std::setprecision(24) << deviation << endl;
177. cout << "总共用时为:" << EndTime - StartTime << endl;
178. }
179. MPI\_Finalize();
180. **return** 0;
181. }