北京邮电大学软件学院

2019-2020 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：

姓名： 平雅霓 **学号：** 2017211949

指导教师： 卢本捷

日　　期： **2019 年 12 月 1 日**

一、 实验目的

学习并行计算的初步方法。

二、 实验内容

用多种方法完成 pi 的并行计算

三、 实验环境

1. 两台或以上的 windows 或 linux 等。
2. 节点的网络互联。
3. Visual studio 2017

四、 实验要求

1. 记录实验过程，分析试验现象。
2. 记录主要源代码
3. 撰写实验报告.

五、 MPICH 实验步骤

1. 方法 1：面积积分

1.1. 问题描述：

$$\frac{\pi}{4} = \int_0^1 \frac{1}{1+x^2} = \text{argtgx}|(1,0)$$

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。

1.2. 解法：

采用等步长中矩形公式，其中 n 为积分区间数，h = 1/n 为步长，xi = (i + 0.5) h 为积分区间的中点，假设采用 p 个进程同时计算，各自计算其中的一部分，然后再将结果加起来。

此处用到一个函数 MPI_Reduce()，其作用是将通信器里所有的进程的某个变量值相加并赋值到另一个变量上，在此方法的实现上我用它来相加每个进程计算的各自负责的积分区域，其函数实现如下：

```
MPI_METHOD MPI_Reduce(
    // 指向输入数据的指针
    _In_range_(!= , recvbuf) _In_opt_ const void* sendbuf,
    // 指向输出数据的指针，即计算结果存放的地方
    _When_(root != MPI_PROC_NULL, _Out_opt_) void* recvbuf,
    // 数据尺寸，可以进行多个标量或多个向量的规约
    _In_range_(>= , 0) int count,
    // 数据类型
    _In_ MPI_Datatype datatype,
    // 规约操作类型
    _In_ MPI_Op op,
```

```
    // 目标进程号，存放计算结果的进程
    _mpi_coll_rank_(root) int root,
    // 通信子
    _In_ MPI_Comm comm
);
```

1.3. 代码实现

```
double area(int argc, char* argv[]) {
    int i, rank, size, n;
    double x, PI, sum = 0, startwtime, endwtime;
    clock_t t = 0;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    startwtime = MPI_Wtime();                          //记录开始时间

    n = 1000000;
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);      //将n广播出去

    double step = 1.0 / n;                             //划分积分面积
    for (i = rank; i < n; i = i + size) {              //每个进程计算自己的部分
        x = (i+0.5)*step;
        sum += 4 / (1 + x * x);
    }

    MPI_Reduce(&sum, &PI, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);    //汇集所有进程的sum的和pi

    if (rank == 0) {
        PI = step * PI;
        printf("面积法求PI\n");
        printf("计算的PI = %.24f\n真实的PI = 3.14159265358979323846264643\n", PI);
        endwtime = MPI_Wtime();                                          //记录计算结束时间
        printf("时间=%f\n", endwtime - startwtime);                      //计算运算总用时
        double piTrue = 3.14159265358979323846264643;                    //计算误差
        printf("误差为：%.15f\n", piTrue - PI);
    }
    MPI_Finalize();
    return PI;
```

代码实现过程为先初始化进程，n 为积分的划分数，使用 MPI_Bcast 将 n 广播给所有进程，然后按照公式计算每个划分区域的面积，然后采用 MPI_Reduce()函数将所有进程的结果汇总相加，最后输出求出的 PI，并计算运算时间和误差。

1.4. 运行结果

当 n = 1000000，进程数为 1 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 1 pi.exe
面积法求PI
计算的PI = 3.14159265358976425019932 3
真实的PI = 3.14159265358979323846264 3
时间=0.014285
误差为：0.000000000000029
```

当 n = 1000000，进程数为 4 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 4 pi.exe
面积法求PI
计算的PI = 3.14159265358990325012200 6
真实的PI = 3.14159265358979323846264 3
时间=0.006115
误差为：-0.000000000000110
```

当 n = 1000000，进程数为 8 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 8 pi.exe
面积法求PI
计算的PI = 3.14159265358989037153492 1
真实的PI = 3.14159265358979323846264 3
时间=0.004095
误差为：-0.000000000000097
```

当 n = 10000，进程数为 4 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 4 pi.exe
面积法求PI
计算的PI = 3.14159265442312385374634 7
真实的PI = 3.14159265358979323846264 3
时间=0.000471
误差为：-0.000000000833331
```

当 n = 1000，进程数为 4 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 4 pi.exe
面积法求PI
计算的PI = 3.14159273692312668302406 6
真实的PI = 3.14159265358979323846264 3
时间=0.000422
误差为：-0.000000083333334
```

1.5. 结果比较

测试用例	n	进程数	运行时间	误差
1	1000000	1	0.014285	0.000000000000029
2	1000000	4	0.006115	-0.000000000000110
3	1000000	8	0.004095	-0.000000000000097
4	10000	4	0.000471	-0.000000000833331
5	1000	4	0.000422	-0.000000083333334

通过比较发现：

（1）当 n 不变，进程数增多时，运行时间会减小。

（2）当进程数不变，积分区域划分数越多时，误差会减小。

2. 方法二：幂级数

2.1. 问题描述

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

arctgx= $\int_0^x \frac{dx}{1+x^2} = \int_0^x \sum_{n=0}^{\infty} (-x^2)^n dx$ 交换次序 $= \sum_{n=0}^{\infty} \int_0^x (-x^2)^n dx$

计算积分得：argtgx=$\sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{2n+1}$

所以当 x=1 时：

π /4= $\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$ = 1-1/3+1/5-1/7+……+ (-1)n/2n+1

2.2. 解法

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。

2.3. 代码实现

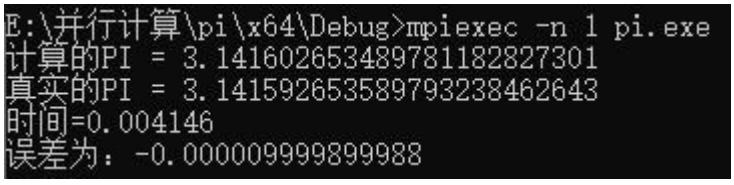
```
//幂级数法求PI
double MiJiShu(int argc, char* argv[]) {
    int done = 0, n, rank, size, i;
    double PI, sum=0; double startwtime, endwtime;
    int namelen; char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    n = 10000;
    startwtime = MPI_Wtime();                    //记录开始时间

    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    for (i = rank + 1; i <= n; i += size)
        sum += pow(-1, i) / (2 * i + 1);

    MPI_Reduce(&sum, &PI, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);        //汇集所有进程的sum的和pi

    if (rank == 0)
    {
        printf("计算的PI = %.24f\n真实的PI = 3.14159265358979323846264643\n", (PI + 1) * 4);
        endwtime = MPI_Wtime();                                              //记录结束时间
        printf("时间=%f\n", endwtime - startwtime);                           //计算运算总用时
        double piTrue = 3.14159265358979323846264643;                        //计算误差
        printf("误差为：%.15f\n", piTrue - (PI + 1) * 4);
    }
    MPI_Finalize();
    return (PI + 1) * 4;
}
```

代码实现过程为先初始化进程，n 为多项式的个数，使用 MPI_Bcast 将 n 广播给所有进程，然后按照公式计算进程负责的各自的多项式，然后采用 MPI_Reduce()函数将所有进程的结果汇总相加，最后输出求出的 PI，

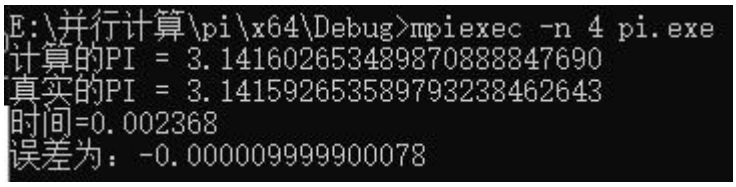
并计算运算时间和误差。

2.4. 运行结果

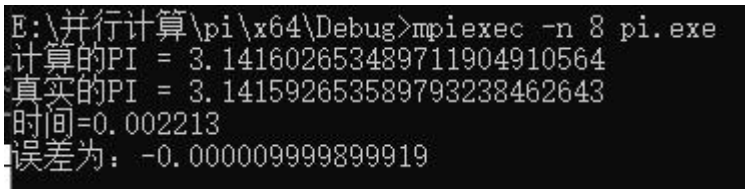
当 n = 100000，进程数为 1 时



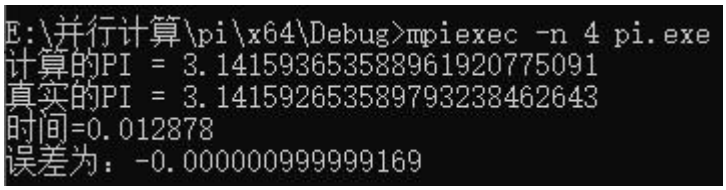
当 n = 100000，进程数为 4 时



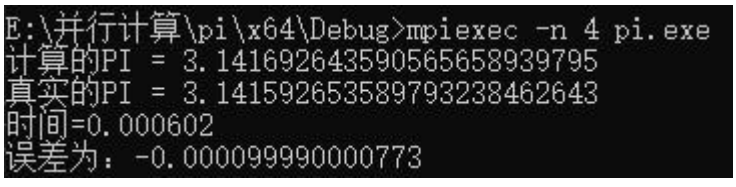
当 n = 100000，进程数为 8 时



当 n = 1000000，进程数为 4 时



当 n = 10000，进程数为 4 时



当 n = 1000，进程数为 4 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 4 pi.exe
计算的PI = 3.142591654339543794094425
真实的PI = 3.141592653589793238462643
时间=0.000641
误差为：-0.000999000749751
```

2.5. 结果比较

测试用例	n	进程数	运行时间	误差
1	100000	1	0.004146	-0.000009999899988
2	100000	4	0.002368	-0.000009999900078
3	100000	8	0.002213	-0.000009999899919
4	1000000	4	0.012878	-0.000000999999169
5	10000	4	0.000602	-0.000099990000773
6	1000	4	0.000641	-0.000999000749751

通过比较发现：

（1） 当 n 不变，进程数增多时，运行时间会减小。

（2） 当进程数不变，多项式数越多时，误差会减小，运行时间也会增大。

3. 方法三：改进的幂级数

3.1. 问题描述

方法一和方法二收敛很慢。要精确到10^{-N} 大致需要计算 2×10^{N} 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φ=arctan(1/5). tanφ =1/5.

tan2φ= 2tanφ/(1-tan2φ) = 5/12.

同理 tan4φ = 120/119. 而 tan（π /4）=1，可见 4φ 与π /4 非常接近。

令 θ =4φ- π /4

所以 tanθ =tan(4φ - π /4) = $\frac{\frac{120}{119}-1}{\frac{120}{119}+1}=\frac{1}{239}$ （根据 tan(A-B) = (tanA-tanB)/(1+tanAtanB)）

θ =arctg (1/239).

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

再利用幂级数展开： =4 $\sum_{n=0}^{\infty}(-1)^n\frac{1}{(2n+1)5^{2n+1}}$ - $\sum_{n=0}^{\infty}(-1)^n\frac{1}{(2n+1)239^{2n+1}}$

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。

3.2. 解法

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。

3.3. 代码实现

```
//改进的幂级数法
double advanced_MiJiShu(int argc, char* argv[]) {
    int done = 0, n, rank, size, i;
    double sum, PI, sum1, sum2;
    double startwtime, endwtime;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    n = 100000;
    startwtime = MPI_Wtime();
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    sum1 = 0.0; sum2 = 0.0;
    for (i = rank; i <= n; i += size) {
        sum1 += 4 * (pow(-1, i) / ((2 * i + 1)*pow(5, 2 * i + 1)));   //左边部分
        sum2 += pow(-1, i) / ((2 * i + 1)*pow(239, 2 * i + 1));       //右边部分
    }
    sum = sum1 - sum2;                          //左边部分减右边部分
    MPI_Reduce(&sum, &PI, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0) {
        printf("计算的PI = %.24f\n真实的PI = 3.14159265358979323846264 3\n", PI * 4);
        endwtime = MPI_Wtime();
        printf("时间=%f\n", endwtime - startwtime);
        double piTrue = 3.14159265358979323846264 3;                    //计算误差
        printf("误差为：%.15f\n", piTrue - PI * 4);
    }
    MPI_Finalize();
    return PI * 4;
}
```

代码实现过程为先初始化进程，n 为一组相减的多项式的个数，使用 MPI_Bcast 将 n 广播给所有进程，然后按照公式计算分别计算多项式的左部和右部，然后左部减去右部，采用 MPI_Reduce()函数将所有进程的结果汇总相加，最后输出求出的 PI，并计算运算时间和误差。

3.4. 运行结果

当 n = 100000，进程数为 1 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 1 pi.exe
计算的PI = 3.14159265358979400417638 3
真实的PI = 3.14159265358979323846264 3
时间=0.047424
误差为：-0.000000000000001
```

当 n = 100000，进程数为 4 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 4 pi.exe
计算的PI = 3.14159265358979356008717 3
真实的PI = 3.14159265358979323846264 3
时间=0.019164
误差为：-0.000000000000000
```

当 n = 100000，进程数为 8 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 8 pi.exe
计算的PI = 3.14159265358979311599796 3
真实的PI = 3.14159265358979323846264 3
时间=0.018084
误差为：0.000000000000000
```

当 n = 1000000，进程数为 4 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 4 pi.exe
计算的PI = 3.14159265358979356008717 3
真实的PI = 3.14159265358979323846264 3
时间=0.186721
误差为：-0.000000000000000
```

当 n = 10000，进程数为 4 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 4 pi.exe
计算的PI = 3.14159265358979356008717 3
真实的PI = 3.14159265358979323846264 3
时间=0.004078
误差为：-0.000000000000000
```

当 n = 1000，进程数为 4 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 4 pi.exe
计算的PI = 3.14159265358979356008717 3
真实的PI = 3.14159265358979323846264 3
时间=0.000622
误差为：-0.000000000000000
```

3.5. 结果比较

测试用例	n	进程数	运行时间	误差
1	100000	1	0.047424	-0.000000000000001
2	100000	4	0.019164	-0.000000000000000
3	100000	8	0.018084	0.000000000000000
4	1000000	4	0.186721	-0.000000000000000
5	10000	4	0.004078	-0.000000000000000
6	1000	4	0.000622	-0.000000000000000

通过比较发现：

（1） 这种方法的精确度确实比方法一和方法二高。

（2） 当 n 不变，进程数增多时，运行时间会减小，但是我发现因为我的 cpu 有 4 核，当进程数大于 4 的时候，运行时间的减少不再明显。

（3） 当进程数不变，多项式越多时，误差会减小，但是运行时间会增大。

4. 方法四：蒙特卡洛方式

4.1. 问题描述

（1） 各个子进程的工作

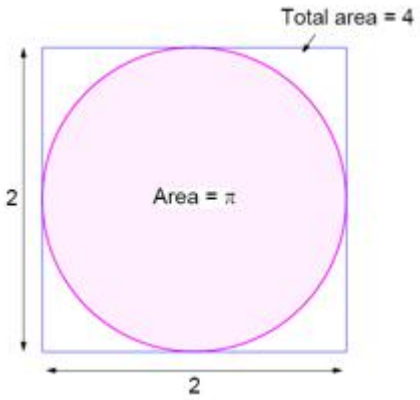
A. 使用随机数在正方形内投点

B. 计算落在圆内的点的次数。

C. 计算比值。

（2） 主进程收集所有的结果，进行平均。

4.2. 解法



$$\frac{\text{Area of circle}}{\text{Area of square}} = \frac{\pi(1)^2}{2 \times 2} = \frac{\pi}{4}$$

用概率的方式计算 pi

（1） 在 2×2 矩形范围内随机产生一个二维坐标，检测其是否落入圆内。

（2） 累计落入圆内的总数，和所有点的总数。

（3） 其比值即 pi/4

（4） 每个处理器各自计算，求均值即可。

4.3. 代码实现

```
//蒙特卡洛方式
double monte(int argc, char** argv) {
    long long int num_in_cycle=0, num, total_num_in_cycle, each_process_num;
    int rank, size;
    double startwtime, endwtime, PI;

    MPI_Init(NULL, NULL);                          //初始化
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);          //得到进程编号
    MPI_Comm_size(MPI_COMM_WORLD, &size);          //得到进程总数

    num=1000000;                                   //生成点的个数
    startwtime = MPI_Wtime();                      //记录开始时间

    MPI_Bcast(&num, 1, MPI_LONG_LONG, 0, MPI_COMM_WORLD);  //将num_point广播出去
    each_process_num = num / size;                 //每个进程计算的点的个数
    double x, y, distance_squared;
    srand(time(NULL));                             //随机数种子

    for (long long int i = 0; i < each_process_num; i++)
    {
        x = (double)rand() / (double)RAND_MAX;
        x = x * 2 - 1;
        y = (double)rand() / (double)RAND_MAX;
        y = y * 2 - 1;
        distance_squared = x * x + y * y;
        if (distance_squared <= 1)
            num_in_cycle = num_in_cycle + 1;
    }
    MPI_Reduce(&num_in_cycle, &total_num_in_cycle, 1, MPI_LONG_LONG, MPI_SUM, 0, MPI_COMM_WORLD);


    if (rank == 0) {
        PI = (double)*&total_num_in_cycle / (double)num * 4;
        printf("计算的PI = %.24f\n真实的PI = 3.14159265358979323846264643\n", PI);
        endwtime = MPI_Wtime();
        printf("时间=%f\n", endwtime - startwtime);
        double piTrue = 3.14159265358979323846264643;                    //计算误差
        printf("误差为: %.15f\n", piTrue - PI);
    }
    MPI_Finalize();
    return PI;
}
```

代码实现过程为先初始化进程，n 是总共要产生点的个数，使用 MPI_Bcast()将 n 广播给所有进程，然后给每个进程分配点数，生成随机数，计算生成的点是否在圆内，此圆以坐标原点为圆心，点到圆心距离小于 1 则在圆内，记录在圆内点的个数，采用 MPI_Reduce()函数将所有进程的结果汇总相加，然后除以总的点的个数，再乘以 4，即为 PI 的值，最后输出求出的 PI，并计算运算时间和误差。

4.4. 运行结果

当 n = 10000000，进程数为 1 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 1 pi.exe
计算的PI = 3.142243199999999792026983
真实的PI = 3.14159265358979323846264 3
时间=1.618719
误差为：-0.000650546410207
```

当 n = 10000000，进程数为 4 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 4 pi.exe
计算的PI = 3.140670399999999862217237
真实的PI = 3.14159265358979323846264 3
时间=0.581927
误差为：0.000922253589793
```

当 n = 10000000，进程数为 8 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 8 pi.exe
计算的PI = 3.142604799999999976023446
真实的PI = 3.14159265358979323846264 3
时间=0.588578
误差为：-0.001012146410207
```

当 n = 100000000，进程数为 4 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 4 pi.exe
计算的PI = 3.141611520000000101759952
真实的PI = 3.14159265358979323846264 3
时间=6.904933
误差为：-0.000018866410207
```

当 n = 1000000，进程数为 4 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 4 pi.exe
计算的PI = 3.137119999999999908624204
真实的PI = 3.14159265358979323846264 3
时间=0.071311
误差为：0.004472653589793
```

当 n = 100000，进程数为 4 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 4 pi.exe
计算的PI = 3.133760000000000101039177
真实的PI = 3.14159265358979323846264 3
时间=0.012628
误差为：0.007832653589793
```

4.5. 结果比较

测试用例	n	进程数	运行时间	误差
1	10000000	1	1.618719	-0.000650546410207
2	10000000	4	0.581927	0.000922253589793
3	10000000	8	0.588578	-0.001012146410207
4	100000000	4	6.904933	-0.000018866410207
5	1000000	4	0.071311	0.004472653589793
6	100000	4	0.012628	0.007832653589793

通过比较发现：

（1） 这种方法的精确度不算很高。

（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，产生的点越多，误差会减小，但是运行时间会增大。

5. 方法五：随机积分法

5.1. 问题描述

利用公式：

$$\text{Area} = \int_{x_1}^{x_2} f(x)\,dx = \lim_{N \to \infty} \frac{1}{N} \sum_{i=1}^{N} f(x_r)(x_2 - x_1)$$

计算 $\pi = 4\int_0^1 \frac{1}{1+x^2}$

5.2. 解法

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。

5.3. 代码实现

```
double sample(int argc, char* argv[]) {
    int i, rank, size, n;
    double x, PI, sum = 0, startwtime, endwtime;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    n = 1000000;
    startwtime = MPI_Wtime();                                   //记录开始时间
    double step = 1.0 / n;
    srand((int)time(0));
    for (i = rank; i < n; i = i + size) {
        x = rand() / RAND_MAX;                                  //生成 0-1 的随机数
        sum += 4 / (1 + x * x);
    }
    MPI_Reduce(&sum, &PI, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0) {
        PI = step * PI;
        printf("计算的PI = %.24f\n真实的PI = 3.14159265358979323846264643\n", PI);
        endwtime = MPI_Wtime();                                 //记录结束时间
        printf("时间=%f\n", endwtime - startwtime);             //计算运行时间
        double piTrue = 3.14159265358979323846264643;
        printf("误差为：%.15f\n", piTrue - PI);                 //计算误差
    }
    MPI_Finalize();
    return PI;
}
```

代码实现过程为先初始化进程，n 是总共要产生随机点的个数，使用 MPI_Bcast()将 n 广播给所有进程，然后给每个进程分配点数，生成 0~1 的随机数，将随机数的函数值相加，采用 MPI_Reduce()函数将所有进程的结果汇总相加，，最后输出 PI，并计算运算时间和误差。

5.4. 运行结果

当 n = 1000000000，进程数为 1 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 1 pi.exe
计算的PI = 3.14160692579141631597 3947
真实的PI = 3.14159265358979323846 2643
时间=83.932939
误差为：-0.000014272201623
```

当 n = 1000000000，进程数为 4 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 4 pi.exe
计算的PI = 3.14159403219691180098 0388
真实的PI = 3.14159265358979323846 2643
时间=38.094131
误差为：-0.000001378607119
```

当 n = 1000000000，进程数为 8 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 8 pi.exe
计算的PI = 3.14167141151399365028 8512
真实的PI = 3.14159265358979323846 2643
时间=34.574037
误差为：-0.000078757924201
```

当 n = 100000000，进程数为 4 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 4 pi.exe
计算的PI = 3.14153564822758557539 8280
真实的PI = 3.14159265358979323846 2643
时间=3.056664
误差为：0.000057005362208
```

当 n = 10000000，进程数为 4 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 4 pi.exe
计算的PI = 3.14049809931806755969 3191
真实的PI = 3.14159265358979323846 2643
时间=0.288682
误差为：0.001094554271726
```

当 n = 1000000，进程数为 4 时

```
E:\并行计算\pi\x64\Debug>mpiexec -n 4 pi.exe
计算的PI = 3.14269729673389397461 8468
真实的PI = 3.14159265358979323846 2643
时间=0.035802
误差为：-0.001104643144101
```

5.5. 结果比较

测试用例	n	进程数	运行时间	误差
1	1000000000	1	83.932939	-0.000014272201623
2	1000000000	4	38.094131	-0.000001378607119
3	1000000000	8	34.574037	-0.000078757924201
4	100000000	4	3.056664	0.000057005362208
5	10000000	4	0.288682	0.001094554271726
6	1000000	4	0.035802	-0.001104643144101

通过比较发现：

（1） 相比其他方法，这种方法的精确度不算很高，而且运行时间较长。

（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，但是运行时间会大幅度增大。

六、 调试心得

通过本次实验，我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。