

KiwenLau

Monday, February 9, 2015

Calculate PI using MPI with three different methods

All rights reserved Please keep the author name: KiwenLau and original blog link :
<http://kiwenlau.blogspot.com/2015/02/calculate-pi-using-mpi-with-three.html>

1. Introduction

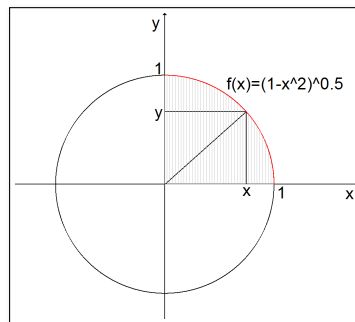
In this blog, I will explain how to calculate PI using MPI. I implement 3 different methods to calculate PI, two of them use Trapezoid rule, another one uses Monte Carlo method. I also compare them with sequential program.

For the experiment, I use Ubuntu 14.04.1 as the Operating system and OpenMPI 1.6.5 as the MPI implementation. And, the CPU has 4 physical cores.

All source codes can be cloned from my GitHub repository:
https://github.com/kiwenlau/MPI_PI

2. Using Trapezoid rule 1

• Analysis



Picture 1 – Trapezoid rule 1

In picture 1, the radius of the circle is 1. So, the area of the circle is π , then the area of the shadow is $\pi/4$. (x, y) is on the circle, so $x^2 + y^2 = 1$, in other word, the function of the red part of the circle is as follows:

$$f(x) = \sqrt{1 - x^2}$$

The area of the shadow is the integral of $f(x)$ from 0 to 1, so:

$$\int_0^1 f(x) dx = \frac{\pi}{4}$$

So:

$$PI = 4 * \int_0^1 f(x) dx = 4 * \int_0^1 \sqrt{1 - x^2} dx$$

So, PI can be calculated trough computing the integral. The approximation of the integral can be easily calculated using trapezoid rule.

• Implementation

Following are some initializations:

```
for (i=rank; i<N; i+=size)
{
    x2=d2*i*i;
    result+=sqrt(1-x2);
}
```

First, each process will calculate a part of the sum:

```
for (i=rank; i<N; i+=size)
{
    x2=d2*i*i;
    result+=sqrt(1-x2);
}
```

KiwenLau



kiwenlau@gmail.com

kiwenlau.com

- kiwenlau.com
- [GitHub](#)

Labels

[Docker](#) [Hadoop](#)
[Ubuntu](#) [VirtualBox](#) [Linux](#)
[Container MPI Mac](#)

Hot Blogs

[Quickly build arbitrary size Hadoop Cluster based on Docker](#)

[Calculate PI using MPI with three different methods](#)

[设置Mac与Virtualbox中的Ubuntu的共享文件夹](#)

[Steps to compile 64-bit Hadoop 2.3.0 under Ubuntu 14.04](#)

[Ubuntu 14.04.01中安装 Virtualbox guest addition](#)

[Linux Container Technology Overview](#)

}

Then, sum up all results using MPI_Reduce:

```
MPI_Reduce(&result, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Last, process 0 will calculate PI

```
pi=4*d*sum;
```

Details can be found in the source code: MPI_PI/ Trapezoid1/mpi_pi.c

• Result

I test the code for 5 times with "np" from 1 to 8. All tests get the same PI.

PI=3.141593

Table 1 shows the execution time.

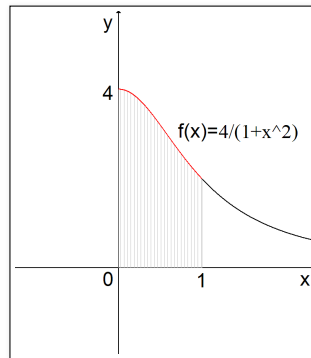
np	Test1	Test2	Test3	Test4	Test5	Average
1	0.138324	0.135576	0.138109	0.13765	0.138454	0.1376226
2	0.068847	0.070167	0.069483	0.068559	0.069432	0.0692976
3	0.05257	0.048384	0.049324	0.051414	0.049129	0.0501642
4	0.037132	0.039768	0.037378	0.04228	0.039799	0.0392714
5	0.060557	0.056813	0.057978	0.058465	0.058904	0.0585434
6	0.052823	0.051535	0.056545	0.050593	0.043815	0.0510622
7	0.06313	0.03907	0.062945	0.062601	0.06349	0.0582472
8	0.044854	0.084504	0.05721	0.059365	0.052478	0.0596822

Table 1 – Trapezoid rule 1 execution time

When "np" equals 4, the number of CPU cores, the code runs the fastest.

3. Using Trapezoid rule 2

• Analysis



Picture 2 – Trapezoid rule 2

In picture 2, the function of the curve is:

$$f(x) = \frac{4}{1+x^2}$$

The area of the shadow is the integral of f(x) from 0 to 1:

$$\int_0^1 f(x) dx = \int_0^1 \frac{4}{1+x^2} dx = \int_0^{\pi/4} \frac{4}{1+\tan^2 \theta} d\theta = \int_0^{\pi/4} 4 d\theta = \pi$$

So:

$$PI = \int_0^1 f(x) dx = \int_0^1 \frac{4}{1+x^2} dx$$

So, PI can be calculated trough computing the integral. The approximation of the integral can be easily calculated using trapezoid rule.

• Implementation

Following are some initializations:

```
#define N 1E7  
#define d 1E-7  
#define d2 1E-14
```

```
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
```

```
MPI_Comm_size (MPI_COMM_WORLD, &size);
```

First, each process will calculate a part of the sum:

```
for (i=rank; i<N; i+=size)
{
    x2=d2*i*i;
    result+=1.0/(1.0+x2);
}
```

Then, sum up all results using MPI_Reduce:

```
MPI_Reduce(&result, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Last, process 0 will calculate PI

```
pi=4*d*sum;
```

Details can be found in the source code: MPI_PI/ Trapezoid2/mpi_pi.c

• Result

I test the code for 5 times with "np" from 1 to 8. All tests get the same PI.

PI=3.141593

Table 2 shows execution time.

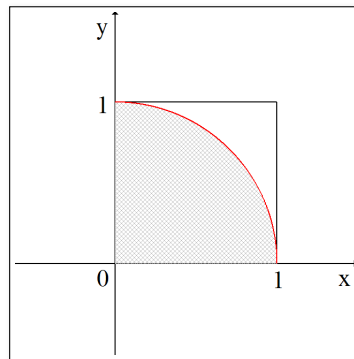
np	Test1	Test2	Test3	Test4	Test5	Average
1	0.13981	0.13833	0.139757	0.13845	0.140403	0.13935
2	0.070451	0.069965	0.070183	0.070751	0.071339	0.0705378
3	0.049448	0.052946	0.051671	0.047731	0.052426	0.0508444
4	0.040811	0.040441	0.03904	0.036853	0.036466	0.0387222
5	0.057964	0.060933	0.060975	0.062817	0.059873	0.0605124
6	0.039066	0.046236	0.051338	0.049921	0.052602	0.0478326
7	0.060012	0.069322	0.06014	0.04282	0.07605	0.0616688
8	0.06458	0.029466	0.054937	0.06015	0.048682	0.051563

Table 2 – Trapezoid rule 2 execution time

When "np" equals 4, the number of CPU cores, the code runs the fastest.

4. Using Monte Carlo method

○ Analysis



Picture 3 – Monte Carlo method

In picture 3, the area of the square is 1 while the area of the shadow is $\pi/4$. If we randomly pick up a large number of points in the square and only count the number of points inside the shadow. Assume total number of points is N while the number of points inside the shadow is M. Then:

$$\frac{M}{N} = \frac{\pi/4}{1} = \frac{\pi}{4}$$

So:

$$PI = \frac{4 * M}{N}$$

• Implementation

Following are some initializations:

```
#define N 1E8
#define d 1E-8
```

```
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
```

```
MPI_Comm_size (MPI_COMM_WORLD, &size);
```

First, each process will calculate a part of the sum:

```

for (i=rank; i<N; i+=size)
{
    x=rand()/(RAND_MAX+1.0);
    y=rand()/(RAND_MAX+1.0);
    if(x*x+y*y<1.0)
        result++;
}

```

Then, sum up all results using MPI_Reduce:

```
MPI_Reduce(&result, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

Last, process 0 will calculate PI

```
pi=4*d*sum;
```

Details can be found in the source code: MPI_PI/ Montecarlo/mpi.c

• Result

I test the code for 5 times with "np" from 1 to 8.

Table 3 shows the value of PI:

np	Test1	Test2	Test3	Test4	Test5	Average
1	3.1418	3.1417	3.1415	3.1416	3.1416	3.1416
2	3.1416	3.1416	3.1416	3.1416	3.1417	3.1416
3	3.1411	3.142	3.1418	3.1414	3.1417	3.1416
4	3.1416	3.142	3.1416	3.1411	3.142	3.1417
5	3.1421	3.142	3.1424	3.1416	3.1414	3.1419
6	3.1413	3.1418	3.1415	3.1417	3.1417	3.1416
7	3.1417	3.1416	3.1415	3.141	3.1416	3.1415
8	3.1411	3.1413	3.1421	3.142	3.1409	3.1415

Table 3 – Monte Carlo method PI value

Table 4 shows execution time:

np	Test1	Test2	Test3	Test4	Test5	Average
1	1.961464	1.967849	1.962504	1.974828	1.965518	1.9664326
2	0.998715	0.980131	0.994609	0.989492	0.986302	0.9898498
3	0.66601	0.667706	0.668889	0.666878	0.665649	0.6670264
4	0.502049	0.503426	0.509393	0.508674	0.49901	0.5045104
5	0.633725	0.615059	0.65566	0.678822	0.558259	0.628305
6	0.665554	0.674438	0.727003	0.587375	0.663797	0.6636334
7	0.582889	0.597576	0.61121	0.657734	0.582026	0.606287
8	0.578283	0.529796	0.585216	0.542904	0.576989	0.5626376

Table 4 – Monte Carlo method execution time

When "np" equals 4, the number of CPU cores, the code runs the fastest.

5. Compare with sequential program

I also implement the sequential program for calculating PI with three different methods.

Details of the sequential program can be found in the source code:

```
MPI_PI/ Trapezoid1/sequential/pi.c
```

```
MPI_PI/ Trapezoid2/sequential/pi.c
```

```
MPI_PI/ Montecarlo/sequential/pi.c
```

I test the code for 5 times.

Table 5 shows execution time:

	Test1	Test2	Test3	Test4	Test5	Average
Trapezoid 1	0.136861	0.138335	0.137984	0.136689	0.137343	0.1374424
Trapezoid 2	0.138011	0.138736	0.138747	0.137833	0.13992	0.1386494
Monte Carlo	1.860186	1.910956	1.860795	1.867981	1.863835	1.8727506

Table 5 – Sequential program execution time

Table 6 shows the speedup of MPI program:

	Sequential	MPI (np=4)	Speedup
Trapezoid 1	0.1374424	0.0392714	3.499809021
Trapezoid 2	0.1386494	0.0387222	3.580617837
Monte Carlo	1.8727506	0.5045104	3.712015847

Table 6 – Compare MPI and sequential program

According to Amdahl's Law, the maximum speedup using 4 processors is 4, so the result shows all the MPI program get a good speedup.

6. MPICH vs OpenMPI

MPI is a standard and it has several implementations, including MPICH and OpenMPI. As I mentioned before, I use OpenMPI 1.6.5 for all the tests. But in fact, I also run all programs using MPICH 3.04 and find some interesting things.

I test the code for 5 times with "np" from 1 to 8 using MPICH 3.04.

Table 7 shows the average execution time using MPICH compared with using OpenMPI:

np	Trapezoid 1		Trapezoid 2		Monte Carlo	
	OpenMPI	MPICH	OpenMPI	MPICH	OpenMPI	MPICH
1	0.1376226	0.0452836	0.13935	0.0392522	1.9664326	1.8396306
2	0.0692976	0.022972	0.0705378	0.0203664	0.9898498	0.9244602
3	0.0501642	0.015644	0.0508444	0.0146466	0.6670264	0.6115258

4	0.0392714	0.0126114	0.0387222	0.0122344	0.5045104	0.4694174
5	0.0585434	0.0259696	0.0605124	0.0342758	0.628305	0.7133318
6	0.0510622	0.0374954	0.0478326	0.0327896	0.6636334	0.694501
7	0.0582472	0.0588336	0.0616688	0.0423222	0.606287	0.647828
8	0.0596822	0.0435324	0.051563	0.0505938	0.5626376	0.6022956

Table 7 – Compare OpenMPI and MPICH

From Table 7, we can see that MPICH almost outperforms OpenMPI in every case. This is very impressive.

Table 8 shows the average execution time of MPICH (np=1), OpenMPI (np=1) and sequential program.

	OpenMPI (np=1)	MPICH (np=1)	Sequential
Trapezoid 1	0.1376226	0.0452836	0.1374424
Trapezoid 2	0.13935	0.0392522	0.1386494
Monte Carlo	1.9664326	1.8396306	1.8727506

Table 8 – Compare OpenMPI, MPICH and sequential program

From Table 8, we can find that MPICH (np=1) even run faster than sequential program! Maybe MPICH has a more efficient compiler, or this is caused by some other reason.

7. Conclusion


In summary, calculating PI using Trapezoid rule is more efficient than using Monte Carlo method. Because plenty of random numbers should be generated when using Monte Carlo method, this will cost lots of time. The communication overhead of calculating PI is very low, so all MPI program show good speedup. MPICH has a better performance than OpenMPI for these programs, this is a interesting phenomenon and worth deep res

发帖者 [Unknown](#) 时间： [1:20 AM](#) 

标签： [MPI](#)

2 comments:

• **Anonymous** [November 14, 2017 at 3:00 PM](#)
thanks a lot. The sequential part is not on git though..
[Reply](#)

 **mohit** [November 23, 2018 at 3:54 AM](#)
ivanka trump hot pictures
[Reply](#)

Enter your comment...

Comment as:

Google Accoun ▼

Publish

Preview

[Newer Post](#)

[Home](#)

[Older Post](#)

Subscribe to: [Post Comments \(Atom\)](#)