**高性能并行计算第7次作业**

姓名：姚虎成 学号：2020317110033

**代码地址：**

点对点计算Pi的C语言程序路径：

/home2/2020317110033/mpi\_pi\_6/mpi\_pi\_point.c

使用MPI\_Reduce计算Pi的C语言路径：

/home2/2020317110033/mpi\_pi\_6/mpi\_pi\_reduce

使用MPI\_Bcast计算Pi的C语言程序路径：

/home2/2020317110033/mpi\_pi\_6/mpi\_pi\_bcast

使用MPI\_Gather计算Pi的C语言程序路径：

/home2/2020317110033/mpi\_pi\_6/mpi\_pi\_bcast

**实验结果：**

本次实验是通过四种MPI的方法进行Pi的计算，分别是点对点通讯的方式、使用MPI\_Reduce、使用MPI\_Bcast、和使用MPI\_Gather四种方法。并比较他们的加速比和并行效率。串行时间=1.670000s。

**表1.1 Point to Point 计算Pi**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 2 | 0.838719 | 99.56% | 1.991130693 |
| 4 | 0.419418 | 99.54% | 3.981709885 |
| 6 | 0.281013 | 99.05% | 5.942790006 |
| 8 | 0.210600 | 99.12% | 7.929720455 |
| 10 | 0.168124 | 99.33% | 9.933137494 |
| 16 | 0.111432 | 93.67% | 14.98676543 |
| 32 | 0.071586 | 72.90% | 23.32862617 |
| 64 | 0.042106 | 61.97% | 39.6622298 |

表1.1是采用点对点计算的Pi的并行效率和加速比，可以看出随着CPU核心的增多，并行效率一开始维持在99%，加速比也接近线性增长，但是随着CPU核心从32到最大值64时，发现并行效率开始下降，加速比提升缓慢，可能原因有两点，一是我们所使用的机器是docker技术虚拟出来的机器，很可能在同一时间段内并没有我们所设定的CPU数目供我们使用。二是随着CPU数目的增加，进程之间的通信也更加频繁，导致通信开销变大。

下表1.2是采用MPI\_Reduce函数计算Pi的并行效率和加速比，规约函数 MPI\_Reduce()，将通信子内各进程的同一个变量参与规约计算。并向指定的进程输出计算结果可以看出，随着CPU数目的增加，并行效率和加速比和采用点对点通讯的表1.1 差别不大，也都呈现出刚开始增加呈线性，后面加速缓慢的现象。

**表1.2 MPI\_Reduce计算Pi**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 2 | 0.837785 | 99.67% | 1.993351516 |
| 4 | 0.421745 | 98.99% | 3.959738705 |
| 6 | 0.281820 | 98.76% | 5.925768221 |
| 8 | 0.211453 | 98.72% | 7.89773614 |
| 10 | 0.168997 | 98.82% | 9.881832222 |
| 16 | 0.107177 | 97.39% | 15.5817013 |
| 32 | 0.057555 | 90.67% | 29.01572409 |
| 64 | 0.054743 | 47.67% | 30.50618344 |

表1.3是使用了MPI\_Bcast计算Pi的并行效率和加速比，MPI\_Bcast()将某个进程的某个变量的值广播到该通信子中所有进程的同名变量中。和之前的表1.1、表1.2大致呈现出相同的趋势。

表1.4使用了MPI\_Gather计算Pi的并行效率和加速比，MPI\_Gather()，将各进程中的向量数据分段聚集到一个进程的大向量中。MPI\_Gather和前三种方法类似，也是当CPU数目增加到32直到更多时，并行效率会显著降低，加速比甚至还出现了减少。可能的原因跟点对点通讯的原因类似，CPU增加到32个核心后，实际并没有32个核心可以用，可能同一时间段还有别的同学在提交任务或者由于docker虚拟出来本来就没有足够多的物理核心可以使用。

**表1.3 MPI\_Bcast计算Pi**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 2 | 0.837946 | 99.65% | 1.992968521 |
| 4 | 0.419583 | 99.50% | 3.980142189 |
| 6 | 0.284769 | 97.74% | 5.864402375 |
| 8 | 0.213245 | 97.89% | 7.831367676 |
| 10 | 0.170511 | 97.94% | 9.794089531 |
| 16 | 0.106804 | 97.73% | 15.6361185 |
| 32 | 0.066340 | 78.67% | 25.17334941 |
| 64 | 0.064152 | 40.67% | 26.03192418 |

**表1.4 MPI\_Gather计算Pi**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 2 | 0.838949 | 99.53% | 1.99058584 |
| 4 | 0.419415 | 99.54% | 3.981736466 |
| 6 | 0.279957 | 99.42% | 5.965201799 |
| 8 | 0.212210 | 98.37% | 7.869563169 |
| 10 | 0.169093 | 98.76% | 9.876221961 |
| 16 | 0.107149 | 97.41% | 15.58577308 |
| 32 | 0.058312 | 89.50% | 28.63904514 |
| 64 | 0.065218 | 40.01% | 25.60642767 |

下图1是随着CPU数目的增加，加速比变化的折线图，可以从折线图中直观的对比四种方法，可以发现，随着CPU数目的增加，四种方法的加速比最开始都是呈线性增加，但当CPU增加到16以及更高时，发现加速比的增加逐渐变缓慢，趋于平缓。像MPI\_Reduce,MPI\_Gather,以及MPI\_Bcast甚至在32核时出现了加速比下降的现象。原因跟所使用的docker机器有关。

**图1 CPU-Speedup图**

**实验分析：**

在本次实验中，我们采用了四种MPI并行方法计算Pi，分别是使用点对点通讯，MPI\_Reduce, MPI\_Bcast, MPI\_Gather四种方法，通过比较和计算其并行效率和加速比，可以看出四种方法在CPU核心增加到16核心之前，加速比基本保持线性增加，并行效率也维持在100%左右，但是随着CPU核心增加到32以及64时，发现MPI\_Reduce, MPI\_Bcast, MPI\_Gather三种方法的并行效率和加速比显著下降。其中原因可能是使用的docker技术虚拟出来的集群，实际我们在使用时，可能没有32个以及更多的核心空闲给我们使用，也可能是同一时间段还有别的同学在提交任务，占用了一部分的资源。

**附录：**

1.点对点计算Pi的C语言代码

#include <stdio.h>

#include <mpi.h>

#define N 100000000

double step = 1.0 / N;

int main(int argc, char \*argv[]){

double x,pi=0.0,sum = 0.0;

double step = 1.0 / N;

int id,n,i;

double start,end,speedup;

MPI\_Status status;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &id);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &n);

start=MPI\_Wtime();

for(i=id;i<N; i=i+n){

x =(i-0.5)\*step;

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

}

MPI\_Send(&sum, 1, MPI\_DOUBLE, 0, 99, MPI\_COMM\_WORLD);

if(id==0){

for (i=0;i<n;i++){

MPI\_Recv(&sum, 1, MPI\_DOUBLE, i, 99, MPI\_COMM\_WORLD, &status);

pi+=sum\*step;

}

end=MPI\_Wtime();

double time=end-start;

printf("pi value =%f\t time=%.8f\n",pi,time);

}

MPI\_Finalize();

return 0;

}

2. 使用MPI\_Reduce计算Pi的C语言代码

#include <mpi.h>

#include<stdio.h>

#include<math.h>

#define N 100000000

int main(int argc,char\*argv[]){

double pi,step,x,sum=0.0;

double start =0.0,stop = 0.0;

int i,rank,size;

step=1.0/N;

MPI\_Init(&argc,&argv);

start = MPI\_Wtime();

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD,&size);/

for(i=rank;i<N;i+=size){

x=(i+0.5)\*step;

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

}

MPI\_Reduce(&sum,&pi,1,MPI\_DOUBLE,MPI\_SUM,0,MPI\_COMM\_WORLD);

if(rank==0){

printf("pi is %lf\t",pi\*step);

stop = MPI\_Wtime();

printf("The program calculates pi using reduce running for :%.6fs \n",stop-start);

}

MPI\_Finalize();

return 0;

}

3. 使用MPI\_Bcast计算Pi的C语言代码

#include<stdio.h>

#include<mpi.h>

int N=100000000;

int main(int argc,char \*argv[]){

int my\_rank,num\_procs;

int i;

double sum,step,x,mypi,pi;

double start =0.0,stop = 0.0;

MPI\_Init(&argc,&argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD,&num\_procs);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&my\_rank);

//printf("Process %d of %d\n",my\_rank,num\_procs);

if(my\_rank == 0){

start = MPI\_Wtime();

}

MPI\_Bcast(&N,1,MPI\_INT,0,MPI\_COMM\_WORLD);

sum =0.0;

step = 1.0/N;

for(i = my\_rank;i<N;i+=num\_procs){

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

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

}

mypi = step\*sum;

MPI\_Reduce(&mypi,&pi,1,MPI\_DOUBLE,MPI\_SUM,0,MPI\_COMM\_WORLD);

if(my\_rank == 0){

stop = MPI\_Wtime();

printf("PI is %.8f\t",pi);

printf("The program calculates pi using bcast running for :%.6fs \n",stop-start);

}

MPI\_Finalize();

return 0;

}

4. 使用MPI\_Gather计算Pi的C语言代码

#include<stdio.h>

#include<mpi.h>

#include <stdlib.h>

#define N 100000000

double step=1.0/(double)N;

int main(int argc,char \*argv[]) {

int i, id,num\_procs;

double x,pi = 0.0;

double buff\_sum = 0.0;

double start,stop;

MPI\_Init(&argc,&argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD,&num\_procs);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&id);

double sum[num\_procs];

if(id == 0)

{

start = MPI\_Wtime();

}

for(i =id,sum[id]=0.0;i<N;i+=num\_procs) {

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

buff\_sum += 4.0/(1.0+x\*x);

}

MPI\_Gather(&buff\_sum,1,MPI\_DOUBLE,sum,1,MPI\_DOUBLE,0,MPI\_COMM\_WORLD);

MPI\_Barrier(MPI\_COMM\_WORLD);

if(id == 0) {

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

pi += sum[i]\*step;

stop = MPI\_Wtime();

printf("Pi is = %lf\t",pi);

printf("The program calculates pi using scatter running for :%.6fs \n",stop-start);

}

MPI\_Finalize();

return 0;

}

5．批量生成运行时间的shell脚本

#!/bin/bash

#this script running mpirun for different processors parameter

list="2 4 6 8 10 16 32 64"

cd /home2/2020317110033/mpi\_pi\_6

for num\_processor in $list

do

echo "num\_processor:$num\_processor"

#mpirun -np $num\_processor ./mpi\_pi\_point

#mpirun -np $num\_processor ./mpi\_pi\_reduce

#mpirun -np $num\_processor ./mpi\_pi\_bcast

mpirun -np $num\_processor ./mpi\_pi\_gather

done