**高性能并行计算结课作业3**

姓名：姚虎成 学号：2020317110033

**代码地址：**

**使用13种方法计算Pi**

1、串行方法计算Pi的C语言程序路径

/home2/2020317110033/serial\_1/pi.c

2、七种OpenMP并行计算方法计算Pi的C语言程序路径

/home2/2020317110033/class\_end\_3/opm\_pi\_7/pi\_parallel\_all.c

/home2/2020317110033/class\_end\_3/opm\_pi\_7/pi\_sh

3、四种MPI并行计算方法计算Pi的C语言程序路径

/home2/2020317110033/class\_end\_3/mpi\_pi\_4/mpi\_pi\_point.c

/home2/2020317110033/class\_end\_3/mpi\_pi\_4/mpi\_pi\_reduce.c

/home2/2020317110033/class\_end\_3/mpi\_pi\_4/mpi\_pi\_bcast.c

/home2/2020317110033/class\_end\_3/mpi\_pi\_4/mpi\_pi\_gather.c

4、MPI+OpenMP并行计算方法计算Pi的C语言程序路径

/home2/2020317110033/class\_end\_3/mpi\_pi\_4/mpi\_openmp\_pi.c

/home2/2020317110033/class\_end\_3/mpi\_pi\_4 /mpi\_pi.sh

**实验结果：**

在计算Pi的C程序时，由于计算时间过短，因此在代码实现过程中将steps扩大至100000000次。下图是计算Pi串行程序的运行结果，结果均保留小数点后六位，Pi=3.141593，串行时间为1.670000s。



表1.1-1.7是采用七种OpenMP并行方法，包括并行域、Padding技术、同步制导语句Critical、Atomic、FirstPrivate、Barrier和For Reduction规约技术，和四种MPI并行计算的方法，分别是MPI\_Send和MPI\_Recv点对点、MPI\_Reduce、MPI\_Bcast、MPI\_Gathers四种方法，最后和一种MPI+OpenMP混合编程，共计12种方法并行计算Pi。

表1.1是使用OpenMP中的并行域编程方法，得到的随着CPU数目的变化，时间，强扩展效率，以及加速比的变化情况。可以看出，随着CPU数目的增加，并行效率以及加速比甚至减少，其结果也不稳定。可能的原因是使用的服务器使用的Docker容器技术，在实际使用CPU数目过程中，可能CPU实际数目并没有那么多，导致并行效率的降低。还有可能是同一时间在服务器运行脚本还有别的同学，导致资源争抢。

**表1.1 并行域计算Pi（Pi=3.141593）**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 1.680021 | 100.00% | 1.000000 |
| 2 | 1.872977 | 44.85% | 0.896979 |
| 3 | 2.778099 | 20.16% | 0.604738 |
| 4 | 3.172086 | 13.24% | 0.529627 |
| 5 | 1.358635 | 24.73% | 1.236551 |
| 6 | 3.576840 | 7.83% | 0.469694 |
| 7 | 2.130868 | 11.26% | 0.788421 |
| 8 | 1.276316 | 16.45% | 1.316305 |
| 9 | 2.042534 | 9.14% | 0.822518 |
| 10 | 2.541952 | 6.61% | 0.660918 |

**表1.2 Padding技术计算Pi（Pi=3.141593）**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 1.679829 | 100.00% | 1.000000 |
| 2 | 0.840302 | 99.95% | 1.999078 |
| 3 | 0.560642 | 99.88% | 2.996260 |
| 4 | 0.420315 | 99.91% | 3.996597 |
| 5 | 0.335990 | 99.99% | 4.999643 |
| 6 | 0.279982 | 100.00% | 5.999780 |
| 7 | 0.240307 | 99.86% | 6.990343 |
| 8 | 0.210169 | 99.91% | 7.992745 |
| 9 | 0.187005 | 99.81% | 8.982811 |
| 10 | 0.168364 | 99.77% | 9.977386 |

表1.2 是采用了OpenMP中Padding技术，利用了一个二维数组来存储积分加和结果，可以看出，随着CPU数目的增加，并行效率基本都维持在99%，CPU数量增多至7个后，并行效率轻微下降，可能是因为通信开销逐渐变大。加速比的提升接近于线性加速比，加速结果表现良好。

表1.3是采用了OpenMP中同步制导语句中的critical技术，critical制导语句中的代码一次只能执行一个线程，可以解决积分加和问题。

可以看出，随着CPU数目的增加，其加速比也接近于线性加速比，并行效率基本维持在100%附近，加速效果表现优秀。甚至有几次超过了100%，这可能是计算机缓存技术导致其产生了超线性加速，随着CPU数目的进一步增多，其并行效率有轻微下降趋势，这可能是随着CPU数目增多，通信的开销逐渐增多。

**表1.3 Critical技术计算Pi（Pi=3.141593）**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 1.679451 | 100.00% | 1.000000 |
| 2 | 0.839931 | 99.98% | 1.999511 |
| 3 | 0.559814 | 100.00% | 3.000017 |
| 4 | 0.420288 | 99.90% | 3.995953 |
| 5 | 0.335705 | 100.06% | 5.002755 |
| 6 | 0.279893 | 100.01% | 6.000324 |
| 7 | 0.240193 | 99.89% | 6.992083 |
| 8 | 0.210351 | 99.80% | 7.984023 |
| 9 | 0.187047 | 99.76% | 8.978746 |
| 10 | 0.168822 | 99.48% | 9.948067 |

**表1.4** **Atomic 技术计算Pi（Pi=3.141593）**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 1.680018 | 100.00% | 1.000000 |
| 2 | 0.840495 | 99.94% | 1.998845 |
| 3 | 0.560219 | 99.96% | 2.998859 |
| 4 | 0.420162 | 99.96% | 3.998503 |
| 5 | 0.335855 | 100.04% | 5.002213 |
| 6 | 0.279945 | 100.02% | 6.001237 |
| 7 | 0.240089 | 99.96% | 6.997479 |
| 8 | 0.210223 | 99.89% | 7.991597 |
| 9 | 0.187102 | 99.77% | 8.979170 |
| 10 | 0.168198 | 99.88% | 9.988365 |

表1.4是跟第三种方法类似，采用了OpenMP中同步制导语句Atomic语句技术实现并行计算，atomic指导语句指定特定的存储单元将被原子更新，即atomic中的操作会被翻译为原子操作，同一时间只能有一个线程执行他，其他的线程需要等待。用于积分加和。

可以看出，随着CPU数目的增多，其并行效率和加速比变化跟critical类似，并行效率基本稳定100%，加速比接近线性上升。加速结果表现优秀。

表1.5跟前面的Atomic和Critical语句类似，采用了同步制导语句Barrier技术设置障碍进行等待来实现并行计算。 可以看出，使得CPU数目的增多，其并行效率基本稳定在100%，加速比也接近理想的线性加速比，并行速度有明显提升。

**表1.5 Barrier技术计算Pi（Pi=3.141593）**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 1.674773 | 100.00% | 1.000000 |
| 2 | 0.839937 | 99.70% | 1.993927 |
| 3 | 0.559130 | 99.84% | 2.995321 |
| 4 | 0.419086 | 99.91% | 3.996251 |
| 5 | 0.336375 | 99.58% | 4.978893 |
| 6 | 0.279812 | 99.76% | 5.985348 |
| 7 | 0.240012 | 99.68% | 6.977876 |
| 8 | 0.210054 | 99.66% | 7.973064 |
| 9 | 0.186728 | 99.66% | 8.969060 |
| 10 | 0.168184 | 99.58% | 9.957976 |

表1.6采用了OpenMP中的Firstprivate方法，通过Firistprivate（sum）将sum定义成每个线程的私有变量，使得循环并行执行的时候，将sum分给各个线程私有化。

可以看出，使得CPU数目的增多，其并行效率基本稳定在100%，加速比也接近理想的线性加速比，并行速度有明显提升。

**表1.6 Firstprivate技术计算Pi（Pi=3.141593）**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 1.679681 | 100.00% | 1.000000 |
| 2 | 0.840118 | 99.97% | 1.999339 |
| 3 | 0.560381 | 99.91% | 2.997393 |
| 4 | 0.420269 | 99.92% | 3.996679 |
| 5 | 0.335905 | 100.01% | 5.000472 |
| 6 | 0.279870 | 100.03% | 6.001656 |
| 7 | 0.240071 | 99.95% | 6.996607 |
| 8 | 0.210589 | 99.70% | 7.976125 |
| 9 | 0.186926 | 99.84% | 8.985829 |
| 10 | 0.168257 | 99.83% | 9.982839 |

表1.7，使用OpenMP中的for循环配合reduction规约操作实现并行计算加速。相比之前的atomic和critical，加入了reduction（+：sum）将各个线程分开，省去了critical和atomic语句，其并行效率基本稳定在100%，加速比也接近理想的线性加速比。并行速度明显提升。

**表1.7 Reduction技术计算Pi（Pi=3.141593）**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 1.680068 | 100.00% | 1.000000 |
| 2 | 0.840574 | 99.94% | 1.998714 |
| 3 | 0.559865 | 100.03% | 3.000845 |
| 4 | 0.420159 | 99.97% | 3.998648 |
| 5 | 0.336072 | 99.98% | 4.999124 |
| 6 | 0.279975 | 100.01% | 6.000773 |
| 7 | 0.240228 | 99.91% | 6.993637 |
| 8 | 0.210293 | 99.86% | 7.989187 |
| 9 | 0.186931 | 99.86% | 8.987616 |
| 10 | 0.168347 | 99.80% | 9.979780 |

表1.8是利用MPI\_Send和MPI\_Recv点对点通讯技术计算Pi，随着CPU数量的增加，并行效率表现良好，加速比基本上呈线性加速。

**表1.8 MPI\_PointToPoint 计算Pi（Pi=3.141593）**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 1.675684 | 100.00% | 1.000000 |
| 2 | 0.838719 | 99.90% | 1.997909 |
| 3 | 0.542570 | 102.95% | 3.088421 |
| 4 | 0.419418 | 99.88% | 3.995260 |
| 5 | 0.356985 | 93.88% | 4.693990 |
| 6 | 0.281013 | 99.38% | 5.963012 |
| 7 | 0.258645 | 92.55% | 6.478702 |
| 8 | 0.210600 | 99.46% | 7.956714 |
| 9 | 0.186532 | 99.82% | 8.983359 |
| 10 | 0.168124 | 99.67% | 9.966953 |

表1.9 是利用MPI\_Reduce技术计算Pi，随着CPU数量的增加，结果基本和使用点对点通讯技术结果类似。并行效率表现良好，加速比基本上呈线性加速。

**表1.9 MPI\_Reduce 计算Pi（Pi=3.141593）**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 1.684103 | 100.00% | 1.000000 |
| 2 | 0.844447 | 99.72% | 1.994326 |
| 3 | 0.568207 | 98.80% | 2.963890 |
| 4 | 0.430555 | 97.79% | 3.911470 |
| 5 | 0.338602 | 99.47% | 4.973695 |
| 6 | 0.281029 | 99.88% | 5.992631 |
| 7 | 0.244720 | 98.31% | 6.881755 |
| 8 | 0.212095 | 99.25% | 7.940324 |
| 9 | 0.188306 | 99.37% | 8.943438 |
| 10 | 0.172541 | 97.61% | 9.760596 |

表1.10 是利用MPI\_Bcast广播技术计算Pi，随着CPU数量的增加，并行效率表现良好，加速比基本上呈线性加速。并行结果略优于使用MPI\_Reduce和MPI点对点通讯技术的结果，并行效率基本上都在100%之上。

**表1.10 MPI\_Bcast 计算Pi（Pi=3.141593）**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 1.714764 | 100.00% | 1.000000 |
| 2 | 0.841948 | 101.83% | 2.036663 |
| 3 | 0.560712 | 101.94% | 3.058190 |
| 4 | 0.423301 | 101.27% | 4.050933 |
| 5 | 0.340518 | 100.72% | 5.035751 |
| 6 | 0.284368 | 100.50% | 6.030088 |
| 7 | 0.243922 | 100.43% | 7.029969 |
| 8 | 0.217965 | 98.34% | 7.867153 |
| 9 | 0.190253 | 100.15% | 9.013072 |
| 10 | 0.172217 | 99.57% | 9.956996 |

表1.11 是利用MPI\_Gather技术计算Pi，随着CPU数量的增加，并行效率表现良好，加速比基本上呈线性加速，并行结果类似于使用MPI\_Bcast的结果，从并行效率上看，MPI\_Gather 比MPI\_Reduce和MPI点对点技术结果要略优。

**表1.11 MPI\_Gather 计算Pi（Pi=3.141593）**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 1.743484 | 100.00% | 1.000000 |
| 2 | 0.841673 | 103.57% | 2.071451 |
| 3 | 0.569705 | 102.01% | 3.060328 |
| 4 | 0.444497 | 98.06% | 3.922375 |
| 5 | 0.344721 | 101.15% | 5.057667 |
| 6 | 0.290110 | 100.16% | 6.009734 |
| 7 | 0.246074 | 101.22% | 7.085202 |
| 8 | 0.217365 | 100.26% | 8.020997 |
| 9 | 0.197735 | 97.97% | 8.817276 |
| 10 | 0.171584 | 101.61% | 10.161111 |

**表1.12 MPI+OpenMP计算Pi（Pi=3.141593）**

|  |  |  |  |
| --- | --- | --- | --- |
| CPU数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 0.367101 | 454.92% | 4.549157 |
| 2 | 0.189096 | 441.57% | 8.831493 |
| 3 | 0.069994 | 795.31% | 23.859188 |
| 4 | 0.054667 | 763.71% | 30.548594 |
| 5 | 0.041576 | 803.35% | 40.167404 |
| 6 | 0.042313 | 657.80% | 39.467776 |
| 7 | 0.036625 | 651.39% | 45.597270 |
| 8 | 0.038439 | 543.07% | 43.445459 |
| 9 | 0.038439 | 482.73% | 43.445459 |
| 10 | 0.029532 | 565.49% | 56.548828 |

表1.12 是利用MPI\_Reduce+OpenMP For Reduction技术计算Pi，设置OpenMP 线程数目为4，可以看出，当CPU数目为1时，加速比就已经达到了4.55，此时虽然只有一个进程，但是开启了四个线程。随着进程数量的增加，线程数目一直为4，并行效率一直保持超高效率，加速比基本上呈超线性加速。但是当进程数目增加到7后，后面继续增加，加速比变化不大。这是因为进程数目增加的同时，由于问题规模并没有增大，导致每个进程分配的任务达不到所在进程的满载状态，进一步导致资源过剩，因此并行效率和加速比提升并不明显。

**图1 十二种并行方法计算Pi**

图1是根据以上12种并行方法，绘制加速比随着进程数目的增加变化的折线图，图中最上面的短虚线MPI+OpenMP方法计算Pi的加速比。中间一块聚在一起没分开的是使用Padding、OMP\_Critical、OMP\_Atomic、OMP\_Barrier、OMP\_Firstprivate、OMP\_Reduction、MPI\_Point、MPI\_Reduce、MPI\_Bcast、MPI\_Gather。最下面的长虚线是并行域方法的加速比变化线。

从上图的结果可以看出，使用MPI+OpenMP混合编程的加速效果远远高于其他方法，使用并行域方法的加速效果远远不如其他方法，原因可能是使用并行域方法进行加速时，可能由于在进行fork-join过程种存在资源（如一些变量）争抢的问题。导致join等待过程时间变长。也可能时同一时间段还有别的同学提交任务，导致计算资源不够。其余10种方法，加速比基本上都是呈线性增加，效果基本一致。中间偶尔有波动，但大体上依然呈线性增长的趋势。

**实验分析：**

在计算Pi的程序中，由于其计算时间过短，将积分中的steps提高至1亿。使用了串行方法、OpenMP中并行域、Padding、Critical、Atomic、Firstprivate、Reduction这七种方法以及MPI编程中的4种方法MPI点对点、MPI\_Reduce、MPI\_Bcast、MPI\_Gather和MPI+OpenMP混合编程方法，共计13种方法对Pi进行计算，并对其中12种并行方法的运行时间，并行效率和加速比做了统计计算。具体脚本见附录。

在实现一次打印从1核到10核不同计算方法的运行时间，我们使用了shell脚本，通过在脚本种设置一个列表，利用shell脚本编程中for循环，进行设置环境变量OMP\_NUM\_THREADS，具体脚本见附录。

对于以上13种方法进行对比，计算Pi的结果小数点后六位基都是正确的，均为3.141593。我们发现利用相比串行计算，除了使用并行域计算会出现并行效率和加速比减少情况，其他方法如使用Atomic、Critical、Barrier、Reduction、Firstprivate和MPI中的点对点、Reduce、Bcast、Gather和MPI+OpenMP方法，并行速度都有明显提升，加速比显著提高，并行效率基本都维持99%。其中，使用MPI+OpenMP混合编程计算Pi效率显著提高。是12种并行计算方法种最快最优的。

像OpenMP中的Critical、Atomic、Firstprivate、Redduction方法以及MPI中的Reduce、Bcast、Gather进行并行计算时，会偶尔出现并行效率突然减少，以及加速比增加缓慢的情况，原因可能有三：首先可能是由于同一时间段还有别的同学正在提交任务。还有可能是由于集群是使用Docker虚拟技术，导致实际使用时，并没有那么多CPU分配给我提交的任务使用，有一定的等待时间。最后可能是随着CPU数量的增加，系统之间通信开销变大，导致加速比不能完全线性增加。

**附录：**

代码1、七种方法并行计算Pi值的C语言程序代码

#include<stdio.h>

#include<omp.h>

#include <time.h>

#define PAD 10

static long num\_steps=100000000;

double step;

//0

//computing pi serially

void pi\_serial(){

int i;

double x, pi, sum = 0.0;

clock\_t start, end;

start = clock();

step = 1.0/(double) num\_steps;

for (i=1;i<= num\_steps; i++){

x = (i-0.5)\*step;

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

}

pi = step \* sum;

end = clock();

printf("the serial program of calculating pi running for %.8fs\n", (double)(end-start)/CLOCKS\_PER\_SEC);

}

//01

//computing pi parallel

void pi\_parallel(){

int NUM\_THREADS = omp\_get\_max\_threads();

int i;

double x,pi,sum[NUM\_THREADS],start,end;

step=1.0/(double)num\_steps;

start=omp\_get\_wtime();

#pragma omp parallel

{

int i;

double x;

int id;

id=omp\_get\_thread\_num();

for(i=id,sum[id]=0.0;i<num\_steps;i=i+NUM\_THREADS){

x=(i-0.5)\*step;

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

}

} end=omp\_get\_wtime();

for(i=0,pi=0.0;i<NUM\_THREADS;i++) pi += sum[i]\*step;

//printf("the true value of pi is %f\n ",pi);

printf("the parallel program running for %.8g s\n ",end-start);

}

//02

//computing pi parallel using padding

void pi\_parallel\_pad(){

int i;

int NUM\_THREADS = omp\_get\_max\_threads();

double x,pi,sum[NUM\_THREADS][PAD],start,end;

step = 1.0/(double)num\_steps;

start=omp\_get\_wtime();

omp\_set\_num\_threads(NUM\_THREADS);

#pragma omp parallel private(i)

{

double x;

int i,id; id = omp\_get\_thread\_num();

for (i=id, sum[id][0]=0.0;i< num\_steps; i=i+NUM\_THREADS){

x = (i-0.5)\*step;

sum[id][0] += 4.0/(1.0+x\*x);

}

} end=omp\_get\_wtime();

for(i=0, pi=0.0;i<NUM\_THREADS;i++){

pi += sum[i][0]\*step;

}

//printf("the padding true value of pi is %f\n ",pi);

printf("the padding parallel program running for %.8g s\n ",end-start);

}

//03

//critical

void pi\_critical(){

int NUM\_THREADS = omp\_get\_max\_threads();

double pi,start,end;

step=1.0/(double)num\_steps;

start=omp\_get\_wtime();

#pragma omp parallel

{

double x,sum;

int id,i; id=omp\_get\_thread\_num();

for(i=id,sum=0.0;i<num\_steps;i=i+NUM\_THREADS){

x=(i+0.5)\*step;

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

}

#pragma omp critical

pi+=sum\*step;

} end=omp\_get\_wtime();

//printf("the true value of pi using critical is %f\n ",pi);

printf("the parallel program using critical running for %.8g s\n ",end-start);

}

//04

//atomic

void pi\_atomic(){

int NUM\_THREADS = omp\_get\_max\_threads();

double pi,start,end;

step=1.0/(double)num\_steps;

start=omp\_get\_wtime();

#pragma omp parallel

{

double x,sum;

int id,i; id=omp\_get\_thread\_num();

for(i=id,sum=0.0;i<num\_steps;i=i+NUM\_THREADS){

x=(i+0.5)\*step;

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

}

#pragma omp atomic

pi+=sum\*step;

} end=omp\_get\_wtime();

//printf("the true value of pi using atomic is %f\n ",pi);

printf("the parallel program using atomic running for %.8g s\n ",end-start);

}

//05

//barrier

void pi\_barrier(){

int i;

int NUM\_THREADS = omp\_get\_max\_threads();

double x,pi,sum[NUM\_THREADS][PAD],start,end;

step = 1.0/(double)num\_steps;

start=omp\_get\_wtime();

omp\_set\_num\_threads(NUM\_THREADS);

#pragma omp parallel private(i)

{

double x;

int i,id; id = omp\_get\_thread\_num();

for (i=id, sum[id][0]=0.0;i< num\_steps; i=i+NUM\_THREADS){

x = (i-0.5)\*step;

sum[id][0] += 4.0/(1.0+x\*x);

}

} end=omp\_get\_wtime();

#pragma omp barrier

for(i=0, pi=0.0;i<NUM\_THREADS;i++){

pi += sum[i][0]\*step;

}

//printf("the padding true value of pi is %f\n ",pi);

printf("the parallel program using barrier running for %.8g s\n ",end-start);

}

//06

//firstprivate

void pi\_firstprivate(){

int NUM\_THREADS = omp\_get\_max\_threads();

double pi=0.0,start,end,sum=0.0;

step = 1.0/(double)num\_steps;

start = omp\_get\_wtime();

#pragma omp parallel firstprivate(sum)

{

double x;

int i,id; id = omp\_get\_thread\_num();

for(i=id;i<num\_steps;i=i+NUM\_THREADS){

x=(i-0.5)\*step;

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

}

pi+=sum\*step;

}end=omp\_get\_wtime();

//printf("the true value of pi using firstprivate is %f\n ",pi);

printf("the parallel program using firstprivate running for %.8g s\n ",end-start);

}

//07

//for reduction

void pi\_reduction(){

int NUM\_THREADS = omp\_get\_max\_threads();

int i;

double pi,sum=0.0,start,end;

step = 1.0/(double)num\_steps;

start=omp\_get\_wtime();

#pragma omp parallel

{

double x;

int i;

#pragma omp for reduction(+:sum)

for (i=1;i< num\_steps;i++){

x = (i-0.5)\*step;

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

}

pi = sum\*step;

}end=omp\_get\_wtime();

//printf("the true value of pi using retuction is %f\n ",pi);

printf("the parallel program using reduction running for %.8g s\n ",end-start);

}

void main(){

pi\_serial();

pi\_parallel();

pi\_parallel\_pad();

pi\_critical();

pi\_atomic();

pi\_barrier();

pi\_firstprivate();

pi\_reduction();

}

2、批量计算不同线程的shell脚本

#!/bin/bash

#this script export OMP\_NUM\_THREADS.

list="1 2 3 4 5 6 7 8 9 10"

cd /home2/2020317110033/openmp\_4

for num\_threads in $list

do

echo "NUM\_THREADS:$num\_threads "

export OMP\_NUM\_THREADS=$num\_threads

./pi\_parallel\_all

Done

3、四种MPI并行计算方法计算Pi的C语言程序路径

1）MPI\_PointToPoint

#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)MPI\_Reduce

#include"mpi.h"

#include<stdio.h>

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

{

int rank,size,data,dataCollect;

MPI\_Init(&argc,&argv);

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

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

printf("rank:%d of size:%d\n",rank,size);

int a[6][6];

int b[6][6];

int i,j;

for( i=0;i<6;i++)

for(j=0;j<6;j++)

{

a[i][j]=rank;

b[i][j]=size;

}

MPI\_Reduce(a,b,1,MPI\_INT,MPI\_SUM,0,MPI\_COMM\_WORLD);

if(0==rank)

{

for( i=0;i<6;i++)

{

for( j=0;j<6;j++)

{

fprintf(stderr,"%d ",b[i][j]);

}

fprintf(stderr,"\n");

}

}

MPI\_Finalize();

return 1;

}

3)MPI\_Bcast

#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)MPI\_Gather

#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;

}

4、MPI+OpenMP并行计算方法计算Pi的C语言程序路径

#include "mpi.h"

#include "omp.h"

#include <math.h>

#include <stdio.h>

#define N 100000000

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

int rank, nproc;

int i,low,up;

double local = 0.0, pi, w, temp,t0,t1,t2;

MPI\_Status status;

MPI\_Init( &argc, &argv );

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

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

t0=MPI\_Wtime();

w = 1.0/N; low = rank\*(N / nproc); up = low + N/nproc - 1;

#pragma omp parallel for reduction(+:local) private(temp,i)

for (i=low;i<up; i++){ temp = (i+0.5)\*w;

local = local + 4.0/(1.0+temp\*temp); }

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

if(rank==0) printf("pi = %.20f\t",pi\*w);

t1=MPI\_Wtime();

if(rank==0) printf("time used = %.6f\n",t1-t0);

MPI\_Finalize();

}