**高性能并行计算第8次作业**

姓名：姚虎成 学号：2020317110033

**代码地址：**

MPI和OpenMP计算Pi混合编程的C语言程序路径：

/home2/2020317110033/mpi\_openmp\_8/mpi\_openmp\_pi.c

寻找最大数的C语言程序路径：

/home2/2020317110033/mpi\_openmp\_8/mat\_max\_serial.c

/home2/2020317110033/mpi\_openmp\_8/mat\_max\_mpi.c

/home2/2020317110033/mpi\_openmp\_8/mat\_max\_mpi\_omp.c

**实验结果：**

**1、MPI和OpenMP计算Pi混合编程**

利用MPI和OpenMP混合编程计算Pi，N=100000000，在OpenMP中设置线程数为2，在MPI编程时设置进程数分别为1，2，4，6，8，10时的计算时间，并计算了加速比和并行效率，计算并行效率的基准值时当进程数为1。结果如表1所示。

从图1可以直观看出，相比于计算Pi的串行时间1.670000s时，当线程数为2，进程数为1时的时间0.385151s，加速效率很高。随着后面进程数目的增多至8时，并行效率一直维持在100%，甚至超过了100%，加速比也一直呈线性变化。加速效果非常好。

**表1. MPI和OpenMP混合编程 计算Pi**

|  |  |  |  |
| --- | --- | --- | --- |
| 进程数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 0.385151 | 100.00% | 1.000000 |
| 2 | 0.191497 | 100.56% | 2.011264 |
| 4 | 0.095718 | 100.60% | 4.023810 |
| 6 | 0.064091 | 100.16% | 6.009440 |
| 8 | 0.047773 | 100.78% | 8.062106 |
| 10 | 0.044331 | 86.88% | 8.688074 |

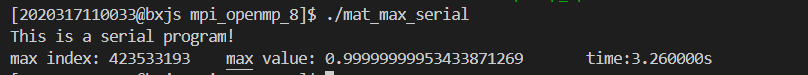
**图1 不同进程数下计算Pi的运行时间和加速比**

但当进程数目增加到10时，加速比和并行效率开始下降，可能的原因是，当计算Pi的问题被分到8个进程时，每个进程可能已经不是满负荷了，所以当进程数目继续增加，每个进程的计算量继续减小，达不到单个进程的满负荷计算时，此时随着进程数目的增多，花在进程之间的通信开销也越来越大，所以出现了并行效率下降的现象。

**2、寻找数组最大最小数**

每个进程随机生成含N个浮点数的数组，使用串行、MPI并行、MPI+OpenMP混合编程并行，找出各进程的全局最大值和该值所在的进程号，以及该进程中最大值所在的数组位置。N为1000000000。

寻找数组最大值串行程序mat\_max\_serial计算结果如下：



寻找数组最大值MPI并行程序mat\_max\_mpi计算结果如下：



寻找数组最大值MPI+OpenMP并行程序mat\_max\_mpi\_omp计算结果如下：



由于每次生成数组都是随机的，所以寻找到的结果也不一样，但是可以比较这三者的运行时间，串行寻找的运行时间是3.260000s，MPI并行寻找的运行时间是1.317383s，MPI+OpenMP并行寻找的运行时间0.928666s。可以比较得出，使用MPI并行编程相对于串行来说可以提升不少，MPI+OpenMP混合编程加速相比仅使用MPI效果更明显。

**实验分析：**

**1、使用MPI+OpenMP混合编程计算Pi**

在使用MPI+OpenMP混合编程模型计算pi时，根据前几次作业的结果，N=100000000时，串行程序计算时间为1.6700000s。仅使用OpenMP开启两个线程时计算时间为0.840000s，所以横向和其他方法比较，使用MPI和OpenMP混合模型进行编程，当MPI为1个进程，OpenMP为两个线程，时间为0.385151s，可以判断，使用MPI+OpenMP混合编程，并行效果相比串行方法、以及仅使用OpneMP和仅使用MPI的都要好。

纵向的比较，我们发现随着MPI进程数目的增加，加速比一直线性增加到8，随后开始下降。并行效率最开始也是一直维持在100%，最后开始下降。造成这种现象的原因可能有两个。首先是随着进程数的增多，问题规模没有变化，每个进程分摊的任务相对来说越来越少，可能到10个进程时，每个进程分配的任务并没有达到其计算能力的峰值。导致每个进程计算能力还有富余。往后继续增加进程数，并行效率便开始下降，线性比也不再呈线性变化。其次，随着进程数目的增加，进程之间的通信开销愈来愈大，这也是导致并行效率下降的原因之一。

**2、寻找数组最大数**

使用串行程序寻找一个包含10亿元素的一维数组中的最大值。我们发现使用MPI+OpenMP和MPI编程并行寻找的效率要比串行高，同时，用MPI+OpenMP混合并行编程要比仅仅使用MPI的效率更高。

**附录：**

1. MPI和OpenMP计算Pi混合编程的C语言代码

#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,start,end;

MPI\_Status status;

MPI\_Init( &argc, &argv );

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

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

start=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);

end=MPI\_Wtime();

if(rank==0) printf("the program using MPI and openMP running for %.6f\n",end-start);

MPI\_Finalize();

}

1. 寻找最大数的C语言程序
2. 串行计算寻找1000000000维数组中最大数

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

long N= 1000000000;

void serial\_find(double \*rand\_mat){

clock\_t begin, end;

begin = clock();

long index; double max=0.0;

for (long i=0; i<N; i++)

if (rand\_mat[i] > max)

{

max = rand\_mat[i];

index = i;

}

end = clock();

printf("This is a serial program!\n");

printf(" max index: %lld\t max value: %.20f\t running time:%.6fs\n",

index, max, (double)(end - begin)/CLOCKS\_PER\_SEC);

}

int main()

{

double \*rand\_mat;

rand\_mat = (double \*)malloc(N \* sizeof (double));

srand(time(0));

for (int i=0; i<N; i++)

{

rand\_mat[i] = rand() \* (1.0/ RAND\_MAX);

}

serial\_find(rand\_mat);

return 0;

}

1. MPI并行计算寻找1000000000维度数组最大数

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include "mpi.h"

#include "omp.h"

#define NUM\_THREADS 4

long N = 1000000000;

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

{

typedef struct {double value; int index;} bingo;

bingo in, out;

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

double t\_start, t\_end;

MPI\_Init(&argc, &argv);

int np, rank;

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

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

if (N % np != 0)

{

printf("array length should be divisible by num process");

MPI\_Abort(MPI\_COMM\_WORLD, 1);

}

int length = N/np;

double \*A;

A = (double \*)malloc(length \* sizeof(double));

srand(time(0));

for (int i=0; i<length; i++)

{

A[i] = rand() \* (1.0/ RAND\_MAX);

}

t\_start = MPI\_Wtime();

double max = 0.0;

int thread\_len = length / NUM\_THREADS;

int t\_index[NUM\_THREADS];

double t\_value[NUM\_THREADS];

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

#pragma omp parallel

{

for (int i=id\*length; i<(id+1) \* length; i++)

{

if(A[i] > max)

{

max = A[i];

t\_index[id] = i;

t\_value[id] = max;

}

}

}

int all\_index = 0; max=0.0;

for(int i=0; i<NUM\_THREADS; i++)

{

if (t\_value[i] > max)

{

max = t\_value[i];

in.value = max;

in.index = (rank + 1) \*length + t\_index[i];

}

}

MPI\_Reduce(&in, &out, 1, MPI\_DOUBLE\_INT, MPI\_MAXLOC, 0, MPI\_COMM\_WORLD);

t\_end = MPI\_Wtime();

if (rank == 0)

{

printf(" max process: %d\t max index: %d\t max value:%.11f\t time:%.11fs\n",

out.index/length, out.index%length, out.value, t\_end-t\_start);

}

free(A);

MPI\_Finalize();

return 0;

}

1. MPI+OpenMP混合编程寻找1000000000维数组最大数

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include "mpi.h"

#include "omp.h"

#define NUM\_THREADS 4

long N = 1000000000;

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

{

// structure

typedef struct {double value; int index;} bingo;

bingo in, out;

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

double t\_start, t\_end;

MPI\_Init(&argc, &argv);

int np, rank;

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

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

if (N % np != 0)

{

printf("array length should be divisible by num process");

MPI\_Abort(MPI\_COMM\_WORLD, 1);

}

int length = N/np;

double \*A;

A = (double \*)malloc(length \* sizeof(double));

srand(time(0));

for (int i=0; i<length; i++)

{

A[i] = rand() \* (1.0/ RAND\_MAX); // [0,1]

}

t\_start = MPI\_Wtime();

double max = 0.0;

int thread\_len = length / NUM\_THREADS;

int t\_index[NUM\_THREADS];

double t\_value[NUM\_THREADS];

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

#pragma omp parallel

{

for (int i=id\*length; i<(id+1) \* length; i++)

{

if(A[i] > max)

{

max = A[i];

t\_index[id] = i;

t\_value[id] = max;

}

}

}

int all\_index = 0; max=0.0;

for(int i=0; i<NUM\_THREADS; i++)

{

if (t\_value[i] > max)

{

max = t\_value[i];

in.value = max;

in.index = (rank + 1) \*length + t\_index[i];

}

}

MPI\_Reduce(&in, &out, 1, MPI\_DOUBLE\_INT, MPI\_MAXLOC, 0, MPI\_COMM\_WORLD);

t\_end = MPI\_Wtime();

if (rank == 0)

{

printf("max process: %d\t max index: %d\t max value:%.11f\t time:%.11fs\n",

out.index/length, out.index%length, out.value, t\_end-t\_start);

}

free(A);

MPI\_Finalize();

return 0;

}

3、批量运行的shell脚本  
#!/bin/bash

#this script running mpirun for different processors parameter

list="1 2 4 6 8 10 16 32 "

cd /home2/2020317110033/mpi\_openmp\_8

for num\_processor in $list

do

echo "num\_processor:$num\_processor"

export OMP\_NUM\_THREADS=2

mpirun --bind-to core -np $num\_processor ./mpi\_openmp\_pi

done