**高性能并行计算结课作业1**

姓名：姚虎成 学号：2020317110033

**代码地址：**

矩阵相乘的串行程序的C语言程序路径：

/home2/2020317110033/class\_end\_1/multi\_mat\_serial.c

矩阵相乘MPI并行计算的C语言程序路径：

/home2/2020317110033/class\_end\_1/ multi\_mat\_mpi.c

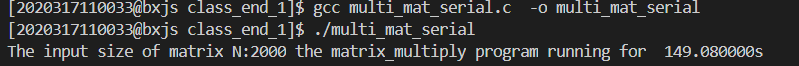
矩阵相乘MPI+OpenMP并行计算的C语言程序路径：

/home2/2020317110033/class\_end\_1/ multi\_mat\_mpi\_omp.c

**实验结果**

1. **串行计算矩阵相乘**

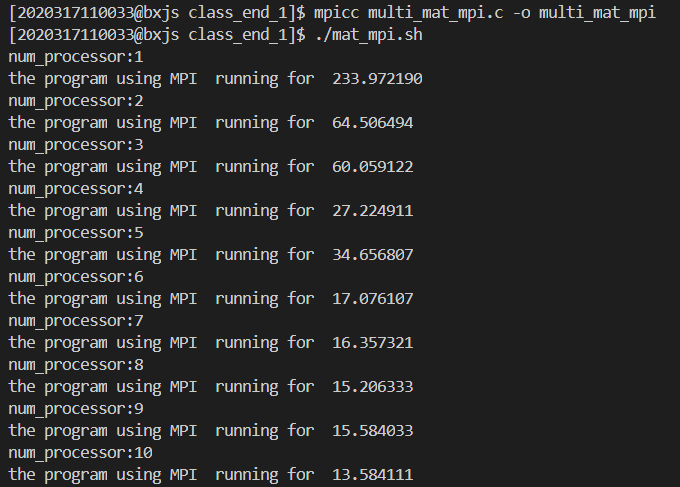
图1是串行的计算矩阵相乘的计算结果，输入数组维度未2000，运行时间为149.080000s，如果维度过大，2万维的数组需要几个小时才能跑出结果，并且中间可能由于其他原因导致任务死掉，无法正常输出结果，所以我们设置矩阵相乘的维度为2000，后面利用MPI以及MPI+OpenMP为便于比较也使用2000维的数组相乘。



**图1 串行计算数组相乘**

**2、MPI计算数组相乘**

利用MPI和OpenMP混合编程计算Pi，N=100000000，在OpenMP中设置线程数为4，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。结果如图2所示。可以看出，随着进程数目的增加，运行时间是逐渐减少然后趋于平衡。



**图2 MPI 计算数组相乘**

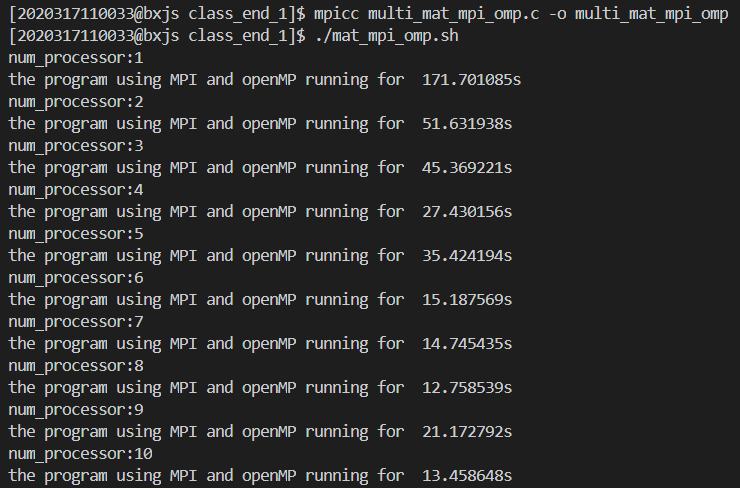
**表1. MPI 计算数组相乘**

|  |  |  |  |
| --- | --- | --- | --- |
| 进程数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 233.972190 | 100.00% | 1.000000 |
| 2 | 64.506494 | 115.55% | 2.311085 |
| 3 | 60.059122 | 82.74% | 2.482221 |
| 4 | 27.224911 | 136.90% | 5.475867 |
| 5 | 34.656807 | 86.03% | 4.301608 |
| 6 | 17.076107 | 145.51% | 8.730327 |
| 7 | 16.357321 | 130.20% | 9.113962 |
| 8 | 15.206333 | 122.55% | 9.803810 |
| 9 | 15.584033 | 106.29% | 9.566202 |
| 10 | 13.584111 | 109.75% | 10.974586 |

根据图2计算了并行效率和加速比得到了表1，从表1可以看出，整体趋势是随着进程数目的从1增加到10，并行效率基本都在100%以上，加速比也一直增加。但是当进程数目为3，5时，并行效率在80%波动，其余基本上都是了效率溢出，出现这样的原因可能是由于机器在运行时，不同时间段的运行状态，可能这段时间还有其他任务需要执行，也可能时当进程数为奇数时，每个进程划分任务不均匀导致进度不一样，需要等待的时间延长。

**3、MPI+OpenMP混合编程计算数组相乘**

在实验2中加入了OpenMP模型混合编程，设置线程值为4。编译运行得到以下结果如图3所示。



**图3 MPI+OpenMP 混合编程计算数组相乘**

根据上述MPI和OpenMP混合编程结果计算了并行效率和加速比，结果如下表2所示。

从表2可以看出，随着进程数目的增加，使用MPI+OpenMP混合模型编程的并行效率也不太稳定，但基本上都在100%以上，加速比折线上升。这可能也与进程是奇数个时，也可能时由于机器状态。与MPI结果类似，但加速比总体相比仅使用MPI更大，并行效果更好。

**表2. MPI和OpenMP混合编程 计算数组相乘**

|  |  |  |  |
| --- | --- | --- | --- |
| 进程数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 171.701085 | 100.00% | 1.000000 |
| 2 | 51.631938 | 166.27% | 3.325482 |
| 3 | 45.369221 | 126.15% | 3.784528 |
| 4 | 27.430156 | 156.49% | 6.259574 |
| 5 | 35.424194 | 96.94% | 4.847000 |
| 6 | 15.187569 | 188.42% | 11.305370 |
| 7 | 14.537785 | 168.72% | 11.810677 |
| 8 | 12.758539 | 168.22% | 13.457739 |
| 9 | 21.172792 | 90.11% | 8.109516 |
| 10 | 13.458648 | 127.58% | 12.757677 |

**实验分析：**

通过使用串行方法、MPI、MPI+OpenMP方法计算N=2000维数组相乘的运行时间，实验过程中，N值不能太大，当N=20000维时，串行需要好几个小时，执行时间太长被舍弃。

图4是将使用MPI和使用MPI+OpenMP混合编程的加速比绘制了折线图，可以更加直观的看出，MPI+OpenMP并行效果是比MPI更加优秀。但是两者都呈现折现上升，很可能是由于奇数个进程，导致每个进程的任务量不一样，进程执行时间不一样，耗费在同步上的时间变多，并行效率下降，加速比增加变缓。

**图4 MPI和MPI+OpenMP 加速比随进程数目变化图**

实验发现使用串行方法最慢，使用MPI+OpenMP方法最优，加速效果最好，偶尔出现当进程数为奇数时，并行效率会下降，加速比增速放缓的现象，推测原因可能时由于奇数个，进程任务划分不均匀引起的。

**附录：**

**1、矩阵相乘的串行程序的C语言程序**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include<math.h>

int N=2000;

int \*\*produce\_empty\_mat(int N){

int \*\*empty\_mat;

empty\_mat=(int\*\*)malloc(N\*sizeof(int\*));

int k;

for(k=0;k<N;k++){

empty\_mat[k]=(int\*)malloc(sizeof(int)\*N);

}

return empty\_mat;

}

int\*\* produce\_rand\_mat(int N ){

int \*\*rand\_mat;

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

int k;

for(k=0;k<N;k++){

rand\_mat[k]=(int\*)malloc(sizeof(int)\*N);

}

srand(time(NULL));

int i,j;

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

for(j=0;j<N;j++){

rand\_mat[i][j]= rand()%10;

}

}

return rand\_mat;

}

void free\_mat(int\*\*\* arr, int N){

int i;

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

free ((\*arr)[i]);}

free (\*arr);

}

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

printf("The input size of matrix N:%d\t",N);

int \*\*A=produce\_rand\_mat(N);

int \*\*B=produce\_rand\_mat(N);

int \*\*C=produce\_empty\_mat(N);

int i,j,k,sum;

clock\_t start, end;

start = clock();

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

for(j=0;j<N;j++){

sum=0;

for(k=0;k<N;k++){

sum=sum+A[i][k]\*B[k][j];

}

C[i][j]=sum;

}

}

end = clock();

printf("the matrix\_multiply program running for %.6fs\n", (double)(end-start)/CLOCKS\_PER\_SEC);

free\_mat(&A,N);

free\_mat(&B,N);

free\_mat(&C,N);

return 0;

}

**2、矩阵相乘MPI并行计算的C语言程序**

#include<stdio.h>

#include<time.h>

#include<stdlib.h>

#include<mpi.h>

int N=2000;

int \*\*produce\_empty\_mat(int N){

int \*\*empty\_mat;

empty\_mat=(int\*\*)malloc(N\*sizeof(int\*));

int k;

for(k=0;k<N;k++){

empty\_mat[k]=(int\*)malloc(sizeof(int)\*N);

}

return empty\_mat;

}

int\*\* produce\_rand\_mat(int N ){

int \*\*rand\_mat;

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

int k;

for(k=0;k<N;k++){

rand\_mat[k]=(int\*)malloc(sizeof(int)\*N);

}

srand(time(NULL));

int i,j;

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

for(j=0;j<N;j++){

rand\_mat[i][j]= rand()%10;

}

}

return rand\_mat;

}

void free\_mat(int\*\*\* arr, int N){

int i;

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

free ((\*arr)[i]);}

free (\*arr);

}

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

{

double start,end;

int \*\*A,\*\*B,\*\*C,\*\*D;

int rank,np;

MPI\_Status status;

MPI\_Init(&argc, &argv);

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

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

A=produce\_rand\_mat(N);

B=produce\_rand\_mat(N);

C=produce\_empty\_mat(N);

D=produce\_empty\_mat(N);

start=MPI\_Wtime();

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

{

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

{

C[i][j]=0;

for(int k=rank; k<N; k+=np)

{

C[i][j] += A[i][k] \* B[k][j];

}

}

}

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

{

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

{

MPI\_Reduce(&C[i][j], &D[i][j], 1, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

}

}

end=MPI\_Wtime();

if(rank==0)

printf("the program using MPI running for %.6fs\n", end-start);

MPI\_Finalize();

free\_mat(&A,N);

free\_mat(&B,N);

free\_mat(&C,N);

free\_mat(&D,N);

return 0;

}

**3、矩阵相乘MPI+OpenMP并行计算的C语言程序**

#include<stdio.h>

#include<stdlib.h>

#include<mpi.h>

#include<omp.h>

#include<time.h>

int N =2000;

int \*\*produce\_empty\_mat(int N){

int \*\*empty\_mat;

empty\_mat=(int\*\*)malloc(N\*sizeof(int\*));

int k;

for(k=0;k<N;k++){

empty\_mat[k]=(int\*)malloc(sizeof(int)\*N);

}

return empty\_mat;

}

int\*\* produce\_rand\_mat(int N ){

int \*\*rand\_mat;

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

int k;

for(k=0;k<N;k++){

rand\_mat[k]=(int\*)malloc(sizeof(int)\*N);

}

srand(time(NULL));

int i,j;

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

for(j=0;j<N;j++){

rand\_mat[i][j]= rand()%10;

}

}

return rand\_mat;

}

void free\_mat(int\*\*\* arr, int N){

int i;

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

free ((\*arr)[i]);}

free (\*arr);

}

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

{

double start,end;

int \*\*A,\*\*B,\*\*C,\*\*D;

int rank,np;

MPI\_Status status;

MPI\_Init(&argc, &argv);

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

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

A=produce\_rand\_mat(N);

B=produce\_rand\_mat(N);

C=produce\_empty\_mat(N);

D=produce\_empty\_mat(N);

start=MPI\_Wtime();

int i,j,k,temp;

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

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

{

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

{

C[i][j]=0;

temp=0;

for(k=rank; k<N; k+=np)

{

temp+=A[i][k]\*B[k][j];

C[i][j]=temp;

}

}

}

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

{

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

{

MPI\_Reduce(&C[i][j], &D[i][j], 1, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD);

}

}

end=MPI\_Wtime();

if(rank==0)

printf("the program using MPI and openMP running for %.6fs\n", end-start);

MPI\_Finalize();

free\_mat(&A,N);

free\_mat(&B,N);

free\_mat(&C,N);

free\_mat(&D,N);

return 0;

}

**4、批量运行MPI计算矩阵相乘shell脚本**

#!/bin/bash

#this script running mpirun for different processors parameter

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

cd /home2/2020317110033/class\_end\_1

for num\_processor in $list

do

echo "num\_processor:$num\_processor"

mpirun -np $num\_processor ./multi\_mat\_mpi

done

**5、批量运行MPI+OpenMP计算矩阵相乘shell脚本**

#!/bin/bash

#this script running mpirun for different processors parameter

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

cd /home2/2020317110033/class\_end\_1

for num\_processor in $list

do

echo "num\_processor:$num\_processor"

export OMP\_NUM\_THREADS=4

mpirun --bind-to core -np $num\_processor ./multi\_mat\_mpi\_omp

done