**高性能并行计算结课作业2**

姓名：姚虎成 学号：2020317110033

**代码地址：**

串行计算数组K项之和的C语言程序路径：

/home2/2020317110033/class\_end\_2/k\_serial

MPI并行计算数组K项之和的C语言程序路径：

/home2/2020317110033/class\_end\_2/k\_mpi.c

/home2/2020317110033/class\_end\_2/k\_mpi.sh

MPI+OpenMP并行计算数组K项之和的C语言程序路径：

/home2/2020317110033/class\_end\_2/k\_mpi\_omp.c

/home2/2020317110033/class\_end\_2/k\_mpi\_omp.sh

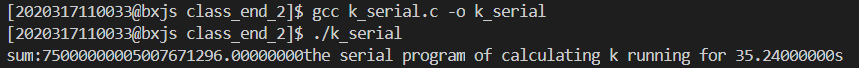
**实验结果：**

**1、串行计算数组K项之和**

通过gcc编译写好的计算数组K项之和的C语言代码，运行结果如下，N的值为1x1010，也就是100亿，包括后面使用MPI以及混合编程时，N值都为100亿。便于比较。

从下图得到求和结果sum=75000000005007671296

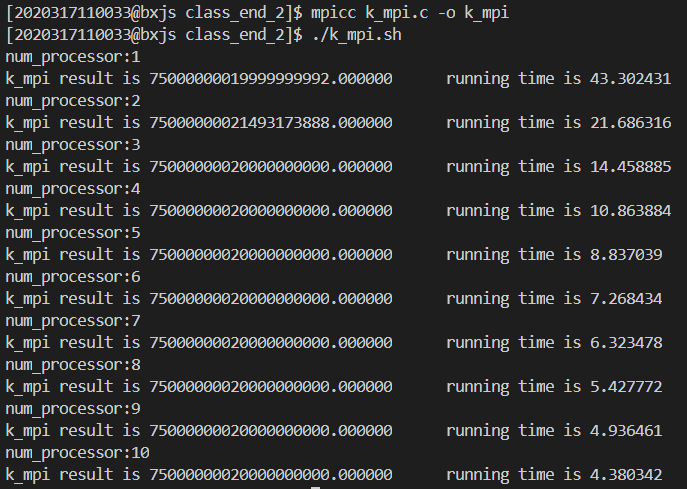
运行时间time=35.24000000s



**2、MPI计算数组K项之和**

使用MPI并行的计算数组K项之和，在开启不同的进程，测试其计算的结果，以及运行时间，采用shell脚本k\_mpi.sh，批量的运行不同参数的mpirun，并输出结果到屏幕。结果如图1所示，使用的N值和串行程序的N值一样。计算结果前两次和后面八次略有区别，和串行计算的结果前10位数都相同，后面有区别，推测可能与sum的double类型有关。

并根据图1做了表1，计算了使用MPI后的并行效率和加速比，计算并行效率和加速比采用的串行时间是当进程为1时的时间，而不是直接使用实验1中计算的串行时间。避免分批次运行由于机器状态不同产生的误差，更好的控制变量。



**图1 MPI输出结果**

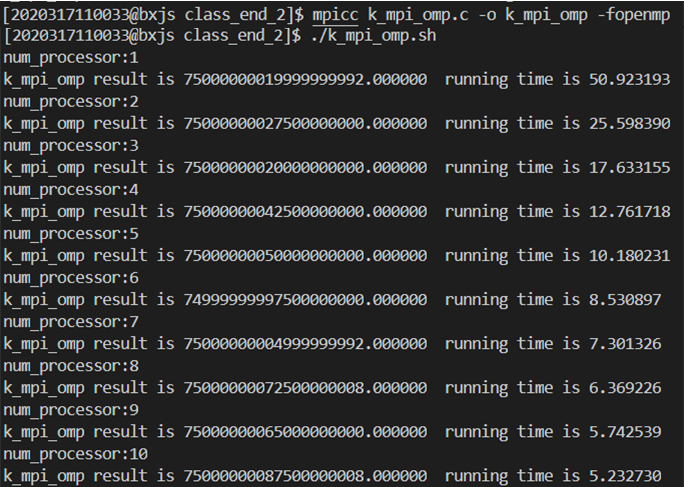
**表1 MPI计算数组K项之和**

|  |  |  |  |
| --- | --- | --- | --- |
| 进程数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 43.302431 | 100.00% | 1.000000 |
| 2 | 21.686316 | 99.84% | 1.996763 |
| 3 | 14.458885 | 99.83% | 2.994867 |
| 4 | 10.863884 | 99.65% | 3.985907 |
| 5 | 8.837039 | 98.00% | 4.900106 |
| 6 | 7.268434 | 99.29% | 5.957601 |
| 7 | 6.323478 | 97.83% | 6.847882 |
| 8 | 5.427772 | 99.72% | 7.977938 |
| 9 | 4.936461 | 97.47% | 8.771958 |
| 10 | 4.380342 | 98.86% | 9.885628 |

从表1可以看出，随着进程数目从1增加到10，并行效率基本维持在100%，但中间当进程数为5，7和9奇数时，其并行效率有波动，推测可能是奇数由于划分到每个进程的任务，未能整除的原因，导致效率下降。加速比也基本呈线性变化。

**3、MPI计算数组K项之和**

使用MPI+OpenMP混合编程计算数组K项之和，和实验2一样，使用shell脚本k\_mpi\_omp.sh输出不同进程下的mpirun的运行时间。实验结果如图2所示。从下面的结果可以看出，运行结果每次都不太一样，但每次差别并不是很大，推测也是由于sum的类型原因引起的。

****

**图2 MPI+OpenMP混合编程结果**

同样的，根据结果计算了使用MPI+OpenMP混合编程的并行效率和加速比，OpenMP的线程数目设置为四个，并做了表2。

从表1、2综合比较，随着进程数目的增加，MPI+OpenMP方法并行效率在99%上下波动，和仅使用MPI相比，使用OpenMP+MPI混合编程的优势并不明显，并没有很大提升。测试过几次都是这样。与结课作业1中计算数组相乘的结果不同，推测可能是线程之间的并行没有生效。

**表2 MPI+OpenMP计算数组K项之和**

|  |  |  |  |
| --- | --- | --- | --- |
| 进程数目 | 时间（s） | 并行效率 | 加速比 |
| 1 | 50.923193 | 100% | 1.000000 |
| 2 | 25.416790 | 100.18% | 2.003526 |
| 3 | 17.633155 | 96.26% | 2.887923 |
| 4 | 12.761718 | 99.76% | 3.990309 |
| 5 | 10.180231 | 100.04% | 5.002165 |
| 6 | 8.530897 | 99.49% | 5.969266 |
| 7 | 7.301326 | 99.64% | 6.974513 |
| 8 | 6.369226 | 99.94% | 7.995193 |
| 9 | 5.742539 | 98.53% | 8.867714 |
| 10 | 5.232730 | 97.32% | 9.731668 |

**实验分析：**

在计算数组K项之和结果中，使用了三种方法进行计算，包括串行计算，MPI并行计算，MPI+OpenMP并行计算，可以看出，后面两种并行方法相对于串行方法提升效率快很多，但是MPI和MPI+OpenMP两种方法在本次实验中，并没有体现出太大的区别，推测可能原因是此种方法编写的代码并没有完全实现线程级的并行，或者是线程级的并行效果不好。但总的来说，使用MPI以及MPI+OpenMP方法并行效率基本维持99%，加速比也一直线性变化。并行效果表现良好。

从计算的准确性来看的话，在一定程度上，三种方法计算的都比较准确，至少保证了前10位数字的正确性，但是10位之后的数字会略有差异，可能与求和的sum类型是double有关，可能是计算机计算double类型的由于数组过大引起的精度丢失。

下图3是将利用MPI编程和MPI+OpenMP混合编程模型得到的加速比随着进程数目的增加折线图，可以从图中看出，使用MPI和使用MPI+OpenMP两种方法的并行效果基本一致，MPI+OpenMP混合编程相比于MPI并没有太大的优势。

**图3 MPI 和MPI+OpenMP 加速比随进程数目变化**

**附录：**

1、串行计算数组K项之和的C语言程序

#include<stdio.h>

#include<omp.h>

#include <time.h>

static long N =10000000000 ;

//serial k

void k\_serial(){

long i,j,k;

double sum=0.0;

clock\_t start,end;

start=clock();

for(i = 1 ; i < N ; i++){

sum+=3\*(i+1)/2;

}

end=clock();

printf("sum:%.8f",sum);

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

}

void main(){

k\_serial();

}

2、MPI并行计算数组K项之和的C语言程序

#include <stdio.h>

#include "mpi.h"

long N = 10000000000;

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

{

long double x,result=0.0;

int np,rank;

double start,end;

MPI\_Init(&argc, &argv);

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

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

start= MPI\_Wtime();

for(long i=rank;i<=N;i+=np)

{

x += 3\*(i + 1)/2;

}

MPI\_Reduce(&x, &result, 1, MPI\_LONG\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

end = MPI\_Wtime();

double time=end-start;

if(rank == 0)

{

printf("k\_mpi result is %Lf\t running time is %lf\n", result,time);

}

MPI\_Finalize();

return 0;

}

3、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\_2

for num\_processor in $list

do

echo "num\_processor:$num\_processor"

mpirun -np $num\_processor ./k\_mpi

done

4、MPI+OpenMP并行计算数组K项之和的C语言

#include <stdio.h>

#include "mpi.h"

#include <omp.h>

#define NUM\_THREADS 4

long N = 10000000000;

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

{

long double low,high,x=0.0,result=0.0;

int np,rank;

double start,end;

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

MPI\_Init(&argc, &argv);

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

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

start= MPI\_Wtime();

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

long block=N/np;

low=id+rank\*block;

high=(rank+1)\*block;

#pragma parallel omp for reduction(+:x) private(i)

for(long i=low;i<=high;i++)

{

x += 3\*(i + 1)/2;

}

MPI\_Reduce(&x, &result, 1, MPI\_LONG\_DOUBLE, MPI\_SUM, 0, MPI\_COMM\_WORLD);

end = MPI\_Wtime();

double time=end-start;

if(rank == 0)

{

printf("k\_mpi\_omp result is %Lf\t running time is %lf\n", result,time);

}

MPI\_Finalize();

return 0;

}

5、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\_2

for num\_processor in $list

do

echo "num\_processor:$num\_processor"

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

done