**高性能并行计算结课作业**

1. **矩阵相乘的并行计算实现**

**代码地址：**

矩阵相乘串行C语言程序路径：

/home2/2020317110037/class\_final/matrix/matrix\_serial.c

矩阵相乘MPI并行C语言程序路径：

/home2/2020317110037/class\_final/matrix/matrix\_mpi.c

矩阵相乘MPI+OpenMP混合编程C语言程序路径：

/home2/2020317110037/class\_final/matrix/matrix\_mpiopenmp.c

**实验结果：**

（1）MPI并行，N=2000

|  |  |  |  |
| --- | --- | --- | --- |
| 进程数 | 时间/s | 加速比 | 并行效率 |
| 串行 | 239.32 | 1.00 | 100.0% |
| 2 | 60.10 | 3.98 | 199.1% |
| 4 | 27.81 | 8.60 | 215.1% |
| 6 | 17.40 | 13.75 | 229.2% |
| 8 | 16.84 | 14.2 | 177.6% |

1. MPI+OpenMP混合编程，N=2000,线程数为4

|  |  |  |  |
| --- | --- | --- | --- |
| 进程数 | 时间/s | 加速比 | 并行效率 |
| 串行 | 239.32 | 1.00 | 100.0% |
| 2 | 50.46 | 4.74 | 237.1% |
| 4 | 20.61 | 11.61 | 290.2% |
| 6 | 11.04 | 21.67 | 361.2% |
| 8 | 8.42 | 28.42 | 355.2% |

**实验分析：**

本实验由于N设置为2000000维时，程序运行一段时间后会被系统自动kill，因此只能降低到2000维进行测试。因为3种方法代码思路相似，下面主要分析MPI+OpenMP混合编程C语言程序。#define NUM\_THREADS 4定义线程数为4。MPI\_Init(&argc,&argv)开始MPI并行。=(int\*\*)malloc(N\*sizeof(int\*))定义ABCD四个动态数组。之后for循环下 A[i][j] = rand()%10；B[i][j] = rand()%10;给数组AB赋值。 #pragma omp parallel for reduction(+:x)private (i,j,k)开始OpenMP多线程并行。在for循环中 x+=A[i][k]\*B[k][j]计算矩阵相乘并记录在 C[i][j] = x。通过for循环和 MPI\_Reduce(&C[i][j],&D[i][j],1,MPI \_INT, MPI\_SUM,0,MPI\_COMM\_WORLD)接收消息并记录到D[i][j]中。在计算矩阵相乘的开始和结束用MPI\_Wtime()来记时，并打印出时间。通过上表的实验结果可以看出，随着进程数的增加，MPI并行和MPI+OpenMP混合编程的时间都减少，加速比增加，并行效率先增加后减少，并且MPI+OpenMP混合编程的并行效率要明显高于MPI。

1. **计算数列n=1,2，...，N的加和值**

**代码地址：**

数列加和的串行C语言程序路径：

/home2/2020317110037/class\_final/K\_add/k\_serial.c

数列加和的MPI并行C语言程序路径：

/home2/2020317110037/class\_final/K\_add/k\_mpi.c

数列加和的MPI+OpenMP混合编程C语言程序路径：

/home2/2020317110037/class\_final/K\_add/k\_mpiopenmp.c

**实验结果：**

（1）MPI并行，N=10000000000

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 进程数 | 结果 | 时间/s | 加速比 | 并行效率 |
| 串行 | 1213023695161793535 | 34.040000 | 1.00 | 100.0% |
| 2 | 1213023695161793535 | 17.204044 | 1.97 | 98.9% |
| 4 | 1213023695161793535 | 8.694499 | 3.91 | 97.8% |
| 6 | 1213023695161793535 | 5.794318 | 5.87 | 97.9% |
| 8 | 1213023695161793535 | 4.343361 | 7.83 | 97.9% |

（2）MPI+OpenMP混合编程，N=10000000000,线程数为4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 进程数 | 结果 | 时间/s | 加速比 | 并行效率 |
| 串行 | 1213023695161793535 | 34.040000 | 1.00 | 100.0% |
| 2 | 1213023695161793535 | 17.117211 | 1.98 | 99.4% |
| 4 | 1213023695161793535 | 2.355145 | 14.45 | 361.3% |
| 6 | 1213023695161793535 | 2.193011 | 15.52 | 258.7% |
| 8 | 1213023695161793535 | 1.846772 | 18.43 | 230.4% |

**实验分析：**

本实验采用了串行、MPI并行、MPI+OpenMP混合3种方法。因为3种方法代码思路相似，下面主要分析MPI+OpenMP混合编程C语言程序。#define NUM\_THREADS 4定义线程数为4。MPI\_Init(&argc,&argv)开始MPI并行。#pragma omp parallel for reduction(+:sum)private(i,j,k)开始OpenMP多线程并行。在for循环中计算数列相加j = i+1; k = i+2;sum += (i + j + k)/2;通过MPI\_Reduce(&sum, &result, 1, MPI\_LONG, MPI\_SUM, 0, MPI\_COMM\_WORLD)接收消息并记录到result中。在计算数列加和的开始和结束用MPI\_Wtime()来记时，并打印出时间。通过上表的实验结果可以看出，随着进程数的增加，MPI并行和MPI+OpenMP混合编程的时间都减少，加速比增加，并行效率先增加后减少，并且MPI+OpenMP混合编程的并行效率要明显高于MPI。

1. **所有pi的计算程序**

**代码地址：**

计算pi的串行C语言程序路径：

/home2/2020317110037/class\_final/pi/pi\_serial.c

计算pi的OpenMP并行C语言程序路径：

/home2/2020317110037/class\_final/pi/pi\_function.h

/home2/2020317110037/class\_final/pi/pi\_main.c

计算pi的MPI并行C语言程序路径：

/home2/2020317110037/class\_final/pi/pi\_dot.c

/home2/2020317110037/class\_final/pi/pi\_bcast.c

/home2/2020317110037/class\_final/pi/pi\_gather.c

/home2/2020317110037/class\_final/pi/pi\_reduce.c

计算pi的MPI+OpenMP混合编程C语言程序路径：

/home2/2020317110037/class\_final/pi/pi\_mpi\_openmp.c

**实验结果：**

1. 串行

编译这个C语言程序：$gcc pi\_serial.c -o pi\_serial

改变steps分别为100000、1000000、10000000、100000000、1000000000时计算程序执行时间的统计结果如下表，用clock（）函数和time命令两种方法：

|  |  |  |  |
| --- | --- | --- | --- |
| **NUM\_sTEPS** | **pi** | **clock\_duration** | **USER-time** |
| **100000** | 3.141593 | 0.000000 | 0m0.002s |
| **1000000** | 3.141593 | 0.000000 | 0m0.005s |
| **10000000** | 3.141593 | 0.030000 | 0m0.041s |
| **100000000** | 3.141593 | 0.380000 | 0m0.383s |
| **1000000000** | 3.141593 | 3.810000 | 0m3.810s |

1. OpenMP并行

采用不同OpenMP方案计算pi，改变线程数，统计加速比和并行效率

1.#pragma omp parallel

|  |  |  |  |
| --- | --- | --- | --- |
| 线程数 | 时间/s | 加速比 | 并行效率 |
| 1 | 15.210000 | 1.00 | 100.0% |
| 2 | 16.660519 | 0.92 | 45.95% |
| 4 | 11.650690 | 1.31 | 32.85% |
| 6 | 11.403546 | 1..34 | 22.38% |
| 8 | 9.371057 | 1.63 | 20.42% |
| 10 | 9.404673 | 1.63 | 16.28% |
| 20 | 8.184162 | 1.87 | 9.35% |
| 30 | 8.181843 | 1.87 | 6.24% |
| 40 | 7.178242 | 2.13 | 5.33% |

2.#Padding

|  |  |  |  |
| --- | --- | --- | --- |
| 线程数 | 时间/s | 加速比 | 并行效率 |
| 1 | 15.260000 | 1.00 | 100.0% |
| 2 | 7.813774 | 1.95 | 97.65% |
| 4 | 4.175756 | 3.65 | 91.36% |
| 6 | 2.915064 | 5.23 | 87.25% |
| 8 | 2.259350 | 6.75 | 84.43% |
| 10 | 1.860385 | 8.20 | 82.03% |
| 20 | 1.052782 | 14.49 | 72.45% |
| 30 | 0.972303 | 15.69 | 52.30% |
| 40 | 0.901723 | 16.92 | 42.31% |

3.#pragma omp parallel firstprivate

|  |  |  |  |
| --- | --- | --- | --- |
| 线程数 | 时间/s | 加速比 | 并行效率 |
| 1 | 15.700000 | 1.00 | 100.0% |
| 2 | 7.754945 | 2.02 | 94.95% |
| 4 | 4.133845 | 3.80 | 94.95% |
| 6 | 2.932939 | 5.35 | 89.22% |
| 8 | 2.263445 | 6.94 | 86.70% |
| 10 | 1.859951 | 8.44 | 84.41% |
| 20 | 1.011574 | 15.52 | 77.60% |
| 30 | 0.959011 | 16.37 | 54.57% |
| 40 | 0.902371 | 17.40 | 43.50% |

4.#pragma omp barrier

|  |  |  |  |
| --- | --- | --- | --- |
| 线程数 | 时间/s | 加速比 | 并行效率 |
| 1 | 15.300000 | 1.00 | 100.0% |
| 2 | 7.730937 | 1.98 | 98.95% |
| 4 | 4.224359 | 3.62 | 90.55% |
| 6 | 2.939266 | 5.21 | 86.76% |
| 8 | 2.262906 | 6.76 | 84.52% |
| 10 | 1.850872 | 8.27 | 82.66% |
| 20 | 1.007978 | 15.18 | 75.90% |
| 30 | 0.969828 | 15.78 | 52.60% |
| 40 | 0.901723 | 16.97 | 42.42% |

5.#pragma omp critical

|  |  |  |  |
| --- | --- | --- | --- |
| 线程数 | 时间/s | 加速比 | 并行效率 |
| 1 | 15.350000 | 1.00 | 100.0% |
| 2 | 7.883538 | 1.95 | 97.35% |
| 4 | 4.277045 | 3.59 | 89.72% |
| 6 | 2.943686 | 5.21 | 86.91% |
| 8 | 2.268025 | 6.77 | 84.60% |
| 10 | 1.867960 | 8.22 | 82.18% |
| 20 | 1.023393 | 15.00 | 75.00% |
| 30 | 0.976732 | 15.72 | 52.40% |
| 40 | 0.903846 | 16.98 | 42.46% |

6.#pragma omp atomic

|  |  |  |  |
| --- | --- | --- | --- |
| 线程数 | 时间/s | 加速比 | 并行效率 |
| 1 | 15.370000 | 1.00 | 100.0% |
| 2 | 7.886547 | 1.95 | 97.44% |
| 4 | 4.188002 | 3.67 | 91.75% |
| 6 | 2.912816 | 5.28 | 87.94% |
| 8 | 2.262848 | 6.79 | 84.90% |
| 10 | 1.861620 | 8.26 | 82.56% |
| 20 | 1.011704 | 15.19 | 75.95% |
| 30 | 0.990499 | 15.52 | 51.73% |
| 40 | 0.904637 | 16.99 | 42.47% |

7.#pragma omp parallel for

|  |  |  |  |
| --- | --- | --- | --- |
| 线程数 | 时间/s | 加速比 | 并行效率 |
| 1 | 15.400000 | 1.00 | 100.0% |
| 2 | 7.812360 | 1.97 | 98.56% |
| 4 | 4.276737 | 3.60 | 90.02% |
| 6 | 2.944530 | 5.23 | 87.17% |
| 8 | 2.272089 | 6.78 | 84.72% |
| 10 | 1.856399 | 8.30 | 82.96% |
| 20 | 0.981669 | 15.69 | 78.44% |
| 30 | 0.972498 | 15.84 | 52.80% |
| 40 | 0.901427 | 17.08 | 42.71% |

图1 计算pi程序的加速比随CPU数目增加的变化

图2 计算pi程序的并行效率随CPU数增加

1. MPI并行

1.点对点计算pi

运行命令：$mpirun -np 5 pi\_dot

输出结果：MPI\_dot结果：pi=3.141593

2.MPI\_Bcast计算pi

运行命令：$mpirun -np 5 pi\_bcast

输入N值：10000000

输出结果：MPI\_Bcast结果为 pi=3.141593

3.MPI\_Reduce计算pi

运行命令：$mpirun -np 5 pi\_reduce

输出结果： MPI\_Reduce结果为 pi=3.141593

4.MPI\_Gather计算pi

运行命令：$mpirun -np 5 pi\_gather.c

输出结果： MPI\_gather结果为 pi=3.141593

1. MPI+OpenMP混合

线程数为2，num\_steps=1000000000

|  |  |  |  |
| --- | --- | --- | --- |
| 进程数 | 时间/s | 加速比 | 并行效率 |
| 串行 | 4.430000 | 1.00 | 100.0% |
| 2 | 1.783688 | 2.48 | 124.0% |
| 4 | 0.204099 | 21.70 | 542.5% |
| 6 | 0.179819 | 24.63 | 410.5% |
| 8 | 0.159346 | 27.80 | 347.5% |

**实验分析：**

（1）串行

串行是单线程计算pi的方式，由上表比较可以看出，这种方式速度最慢。

（2）OpenMP并行

对于这7种方法，设计了一个头文件function.h用于存放采用不同方案的函数，之后在pi\_main.c文件中通过#include“pi\_function.h”导入，并通过相应的函数名调用不同方案。在pi\_main.c文件中设计了一个if条件判断，用于选择不同的方案；一个for循环，用于改变线程数并批量输出结果，详细信息见附录代码。

1.pragma omp parallel：parallel指令用在一个结构块之前，表示这段代码将被多个线程并行执行，但是由于是共享内存编程，会影响程序的运行速度。

2.Padding：padding采用了结构体填充的方法，避免了多个处理器里面的数据块共享内存。

3.firstprivate：firstprivate子句指定每个线程都有它自己的变量私有副本，并且变量要被继承主线程中的初值，避免了伪共享的发生。

4.barrier：用于并行区内代码的线程同步，所有线程执行到barrier时要停止，直到所有线程都执行到barrier时才继续往下执行。

5.critical：critical制导语句表明域中的代码一次只能执行一个线程，其他线程被阻塞在临界区。

6.atomic：atomic制导语句指定特定的存储单元将被原子更新。

7.pragma omp parallel for：parallel和for语句的结合，用在一个for循环之前，表示for循环的代码将被多个线程并行执行。

通过分析对实验表格的分析绘制了图1和图2两个折线图。图1是计算pi程序的加速比随CPU数目增加的变化，图2是计算pi程序的并行效率随CPU数增加的变化。可以发现方案1直接使用#pragma omp parallel并行域并行化的加速比和并行效率都明显很低。出现这种状况的原因是因为存在伪共享。在多核系统上，每一个处理器都有自己的缓存。计算机体系结构中，必须要保证内存数据的一致性，当某个处理器对属于的它自己的缓存的变量执行更新操作时，如果那个变量所在的块也被其他的核放在了缓存里面，那么就会发生伪共享。要解决伪共享的问题可以采用以上2-7的方案。通过折线图可以看到，随着CPU数目的增加，方案2-7的加速比和并行效率都相似。另外，随着CPU数的增加，并行效率也越来越低。这是因为计算pi和k都是“计算密集型”任务，要进行大量的计算，消耗CPU资源。如果采用并行计算，任务越多，花在任务切换的时间就越多，CPU执行任务的效率就越低。

（3）MPI并行

1.在pi\_dot.c文件中，通过MPI\_Send(&sum,1,MPI\_DOUBLE,0,99,MPI\_COM M\_WORLD)将并行计算的sum值发送给0号进程，再通过MPI\_Recv(&sum,1,M PI\_DOUBL E,source,99,MPI\_COMM\_WORLD, &status)让0号进程接收并计算加和pi+=sum\* step。

2.在pi\_bcast.c文件中，首先在0号进程下通过scanf( )函数输入N值，通过MPI\_Bcast(&N,1,MPI\_INT,0,MPI\_COMM\_ WORLD)将N值发送给所有进程，并行计算后，通过MPI\_Reduce(&pi,&mypi,1,MPI\_DOUBLE,MPI\_SUM,0,MPI\_CO MM\_WORLD)收集结果pi。

3.在pi\_reduce.c文件中，在for循环计算sum后，通过MPI\_Reduce(&su m,&pi,1,MPI\_DOUBLE,MPI\_SUM,0,MPI\_COMM\_WORLD)收集sum值，并计算pi\*step。

4.在pi\_gather.c文件中，在for循环计算buff后，通过MPI\_Gather(&buff,1, MPI\_DOUBLE,sum,1,MPI\_DOUBLE,0,MPI\_COMM\_WORLD)收集buff值并赋给0号进程上的sum，计算pi+ = step\*sum[i]。

（4）MPI+OpenMP混合

在pi\_mpi\_openmp.c文件中，采用了MPI和OpenMP混合编程，通过define NUM\_THREADS 2定义线程数为2，通过mpirun -np n命令选择进程数分别为2、4、6、8。通过上表可以看出，加速比和并行效率都大大提高了。这是因为每个节点分配多个MPI进程后，每个MPI进程执行多个OPENMP线程。OPENMP部分由于不需要进程间通信，直接通过内存共享方式交换信息，不走网络带宽，所以可以显著减少程序所需通讯的信息。

**附录：**

**第一题matrix**

matrix\_serial.c文件代码

#include <stdio.h>

#include <time.h>

#include<stdlib.h>

#define N 2000

int main(int argc, char \* argv[])

{

    int \*\*A, \*\*B, \*\*C;

    A=(int\*\*)malloc(N\*sizeof(int\*));

    B=(int\*\*)malloc(N\*sizeof(int\*));

    C=(int\*\*)malloc(N\*sizeof(int\*));

    for(int i=0;i<N;i++){

        A[i]=(int\*)malloc(sizeof(int)\*N);

        B[i]=(int\*)malloc(sizeof(int)\*N);

        C[i]=(int\*)malloc(sizeof(int)\*N);

    }

    clock\_t t\_start, t\_end;

    srand(time(NULL));

    for(int i=0; i<N; i++)

    {

        for (int j=0; j<N; j++)

        {

            A[i][j] = rand()%10;

            B[i][j] = rand()%10;

        }

    }

    t\_start = clock();

    for (int i=0; i<N; i++)

    {

        for (int j=0; j<N; j++)

        {

            for(int k=0; k<N; k++)

            {

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

            }

        }

    }

    t\_end = clock();

    double time;

    time = (double)(t\_end - t\_start)/CLOCKS\_PER\_SEC;

    for(int i=0;i<N;i++){

        free(A[i]);

        free(B[i]);

        free(C[i]);

    }

    free(A);free(B);free(C);

    printf("串行时间为：%.2fs\n",time);

    return 0;

}

matrix\_mpi.c文件代码

#include <stdio.h>

#include <time.h>

#include<stdlib.h>

#include<mpi.h>

#define N 2000

int main(int argc, char \* argv[])

{

    double t\_start, t\_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=(int\*\*)malloc(N\*sizeof(int\*));

    B=(int\*\*)malloc(N\*sizeof(int\*));

    C=(int\*\*)malloc(N\*sizeof(int\*));

    D=(int\*\*)malloc(N\*sizeof(int\*));

    for(int i=0;i<N;i++){

        A[i]=(int\*)malloc(sizeof(int)\*N);

        B[i]=(int\*)malloc(sizeof(int)\*N);

        C[i]=(int\*)malloc(sizeof(int)\*N);

        D[i]=(int\*)malloc(sizeof(int)\*N);

    }

    srand(time(NULL));

    for(int i=0; i<N; i++)

    {

        for (int j=0; j<N; j++)

        {

            A[i][j] = rand()%10;

            B[i][j] = rand()%10;

        }

    }

    t\_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);

        }

    }

    t\_end = MPI\_Wtime();

    if(rank==0)

        printf("MPI时间为：%.2fs\n",t\_end-t\_start);

    MPI\_Finalize();

    for(int i=0;i<N;i++){

        free(A[i]);

        free(B[i]);

        free(C[i]);

    }

    free(A);free(B);free(C);

    return 0;

}

matrix\_mpiopenmp.c文件代码

#include <stdio.h>

#include <time.h>

#include<stdlib.h>

#include<mpi.h>

#include<omp.h>

#define NUM\_THREADS 4

#define N 2000

int main(int argc, char \* argv[])

{

    double t\_start, t\_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=(int\*\*)malloc(N\*sizeof(int\*));

    B=(int\*\*)malloc(N\*sizeof(int\*));

    C=(int\*\*)malloc(N\*sizeof(int\*));

    D=(int\*\*)malloc(N\*sizeof(int\*));

    for(int i=0;i<N;i++){

        A[i]=(int\*)malloc(sizeof(int)\*N);

        B[i]=(int\*)malloc(sizeof(int)\*N);

        C[i]=(int\*)malloc(sizeof(int)\*N);

        D[i]=(int\*)malloc(sizeof(int)\*N);

    }

    srand(time(NULL));

    for(int i=0; i<N; i++)

    {

        for (int j=0; j<N; j++)

        {

            A[i][j] = rand()%10;

            B[i][j] = rand()%10;

        }

    }

    t\_start = MPI\_Wtime();

    int i,j,k,x;

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

    for (int i=0; i<N; i++)

    {

        for (int j=0; j<N; j++)

        {

            C[i][j]=0;

            x=0;

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

            {

                x+=A[i][k]\*B[k][j];

                C[i][j] = x;

            }

        }

    }

    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);

        }

    }

    t\_end = MPI\_Wtime();

    if(rank==0)

        printf("MPI+OpenMP时间为：%.2fs\n",t\_end-t\_start);

    MPI\_Finalize();

    for(int i=0;i<N;i++){

        free(A[i]);

        free(B[i]);

        free(C[i]);

    }

    free(A);free(B);free(C);

    return 0;

}

**第二题K\_add**

k\_serial.c文件代码

#include<stdio.h>

#include<time.h>

int main()

{

    long N=10000000000;

    clock\_t start, stop;

    double clock\_Duration;

    long sum=0.0;

    long i, j, k;

    start = clock();

    for (i=1;i<N-1;i++){

        j = i+1; k = i+2;

        sum += (i + j + k)/2;

    }

    stop = clock();

    clock\_Duration =(double)(stop-start)/CLOCKS\_PER\_SEC;

    printf("串行结果：\nN=%lld,K=%.1lld,clock\_Duration=%fs\n",N,sum,clock\_Duration);

    return clock\_Duration;

}

k\_mpi.c文件代码

#include <stdio.h>

#include "mpi.h"

int main(int argc, char \* argv[])

{

    long N = 10000000000;

    int size, rank;

    long result=0;

    double start, stop;

    MPI\_Init(&argc, &argv);

    start = MPI\_Wtime();

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

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

    long i, j, k;

    long sum=0;

    for(i=rank+1; i<N-1; i+=size)

    {

        j = i+1; k = i+2;

        sum += (i + j + k)/2;

    }

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

    stop = MPI\_Wtime();

    if(rank == 0)

    {

        printf("MPI并行结果：\nN=%lld,K=%.1lld,time=%fs\n",N,result,stop-start);

    }

    MPI\_Finalize();

    return 0;

}

k\_mpiopenmp.c文件代码

#include<stdlib.h>

#include <stdio.h>

#include<mpi.h>

#include<omp.h>

#define NUM\_THREADS 4

int main(int argc, char \* argv[])

{

    long N = 10000000000;

    int size, rank;

    long result=0;

    double start, stop;

    MPI\_Init(&argc, &argv);

    start = MPI\_Wtime();

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

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

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

    long sum=0;

    long i, j, k;

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

    for(i=rank+1; i<N-1; i+=size)

    {

        j = i+1; k = i+2;

        sum += (i + j + k)/2;

    }

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

    stop = MPI\_Wtime();

    if(rank == 0)

    {

        printf("MPI+OpenMP并行结果：\nN=%lld,K=%.1lld,time=%fs\n",N,result,stop-start);

    }

    MPI\_Finalize();

    return 0;

}

**第三题pi**

pi\_serial.c文件代码

#include<stdio.h>

#include<time.h>

static long num\_steps = 1000000000;

double step;

clock\_t start, stop;

double clock\_Duration;

int main()

{

 start = clock();

 int i;

 double x,pi,sum=0.0;

 step = 1.0/(double)num\_steps;

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

                x=(i-0.5)\*step;

                sum +=4.0/(1.0+x\*x);

        }

        pi = step\*sum;

        stop = clock();

        clock\_Duration =(double)(stop-start)/CLOCKS\_PER\_SEC;

    printf("num\_steps=%d,pi=%f,clock\_Duration=%f\n",num\_steps,pi,clock\_Duration);

        return 0;

}

pi\_function.h文件代码

#include<omp.h>

#include<stdio.h>

#include<time.h>

#define num\_steps 1000000000

int j;

double pi,clock\_Duration,sum=0.0,start,end;

double step = 1.0/(double)num\_steps;

//串行时间

double serial(){

        clock\_t star, stop;

        star = clock();

        double x;

        int i;

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

                x=(i-0.5)\*step;

                sum +=4.0/(1.0+x\*x);

        }

        pi = step\*sum;

        stop = clock();

        clock\_Duration =(double)(stop-star)/CLOCKS\_PER\_SEC;

        printf("串行结果：\nnum\_steps=%d,pi=%f,clock\_Duration=%f\n",num\_steps,pi,clock\_Duration);

        return clock\_Duration;

}

//1.#pragma omp parallel

double parallel(int NUM\_THREADS){

        double sum[NUM\_THREADS],t1;

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

        start=omp\_get\_wtime();

        #pragma omp parallel

        {

                double x;

                int i,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);

                }

        }

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

                pi+=sum[j]\*step;

        }

        end=omp\_get\_wtime();

        t1=end-start;

        printf("CPU数为%d个时，parallel结果：\nPi=%f,Time=%f,",NUM\_THREADS,pi,t1);

        return t1;

}

//2.加padding的#pragma omp parallel

double padding(int NUM\_THREADS){

        #define PAD 40

        double sum[NUM\_THREADS][PAD],t2;

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

        start=omp\_get\_wtime();

        #pragma omp parallel

        {

                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);

                }

        }

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

                pi+=sum[j][0]\*step;

        }

        end=omp\_get\_wtime();

        t2=end-start;

        printf("CPU数为%d个时，padding结果：\nPi=%f,Time=%f,",NUM\_THREADS,pi,t2);

        return t2;

}

//3.#pragma omp parallel firstprivate

double firstprivate(int NUM\_THREADS){

        double t3;

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

        start=omp\_get\_wtime();

        #pragma omp parallel firstprivate(sum)

        {

                double x,sum,pi=0.0;

                int i,id;

                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);

                }

                pi += sum\*step;

        }

        end = omp\_get\_wtime();

        t3=end-start;

        printf("CPU数为%d个时，firstprivate结果：\nPi=%f,Time=%f,",NUM\_THREADS,pi,t3);

        return t3;

}

//4.#pragma omp barrier

double barrier(int NUM\_THREADS){

        #define PAD 40

        double sum[NUM\_THREADS][PAD],t4;

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

        start=omp\_get\_wtime();

        #pragma omp parallel

        {

                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);

                }

                #pragma omp barrier

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

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

        }

        end = omp\_get\_wtime();

        t4=end-start;

        printf("CPU数为%d个时，barrier结果：\nPi=%f,Time=%f,",NUM\_THREADS,pi,t4);

        return t4;

}

//5.#pragma omp critical

double critical(int NUM\_THREADS){

        double t5;

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

        start=omp\_get\_wtime();

        #pragma omp parallel

        {

                double x,sum,pi=0.0;

                int i,id;

                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();

        t5=end-start;

        printf("CPU数为%d个时，critical结果：\nPi=%f,Time=%f,",NUM\_THREADS,pi,t5);

        return t5;

}

//6.#pragma omp atomic

double atomic(int NUM\_THREADS){

        double t6;

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

        start=omp\_get\_wtime();

        #pragma omp parallel

        {

                double x,sum,pi=0.0;

                int i,id;

                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();

        t6=end-start;

        printf("CPU数为%d个时，atomic结果：\nPi=%f,Time=%f,",NUM\_THREADS,pi,t6);

        return t6;

}

//7.#pragma omp parallel for

double parallelfor(int NUM\_THREADS){

        double sum=0.0,t7;

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

        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();

        t7=end-start;

        printf("CPU数为%d个时，parallel for结果：\nPi=%f,Time=%f,",NUM\_THREADS,pi,t7);

        return t7;

}

pi\_main.c文件代码

#include<stdio.h>

#include"pi\_function.h"

void main(){

double speedup,efficiency;

printf("请选择一种方法：\n[1]parallel [2]padding [3]firstprivate [4]barrier [5]critical [6]atomic [7]for\n");

int methods;

scanf("%d",&methods);

//串行时间

double T0=0;

T0=serial();

int N[8]={2,4,6,8,10,20,30,40};

for (int a=0;a<8;a++){

    //1.#pragma omp parallel

    if(methods==1){

        double T1=parallel(N[a]);

        speedup=T0/T1;

        efficiency=speedup/N[a]\*100;

    }

//2.加padding的#pragma omp parallel

    else if(methods==2){

        double T2=padding(N[a]);

        speedup=T0/T2;

        efficiency=speedup/N[a]\*100;

    }

//3.#pragma omp parallel firstprivate

    else if(methods==3){

        double T3=firstprivate(N[a]);

        speedup=T0/T3;

        efficiency=speedup/N[a]\*100;

    }

    //4.#pragma omp barrier

    else if(methods==4){

        double T4=barrier(N[a]);

        speedup=T0/T4;

        efficiency=speedup/N[a]\*100;

    }

    //5.#pragma omp critical

    else if(methods==5){

        double T5=critical(N[a]);

        speedup=T0/T5;

        efficiency=speedup/N[a]\*100;

    }

    //6.#pragma omp atomic

    else if(methods==6){

        double T6=atomic(N[a]);

        speedup=T0/T6;

        efficiency=speedup/N[a]\*100;

    }

    //7.#pragma omp parallel for

    else if(methods==7){

        double T7=parallelfor(N[a]);

        speedup=T0/T7;

        efficiency=speedup/N[a]\*100;

    }

    //输入错误，请重新输入

    else{

        printf("输入错误，请重新输入");

    }

    printf("speedup=%.2f,efficiency=%.2f%%\n",speedup,efficiency);

}

}

pi\_dot.c文件代码

#include <stdio.h>

#include <mpi.h>

#include<time.h>

#define N 10000000

double main(int argc, char \*argv[]){

    double x,pi=0.0,sum = 0.0;

    double step = 1.0 / N;

    int id,n;

    double start,end;

    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(int 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 (int source=0;source<n;source++){

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

            pi+=sum\*step;

        }

    end=MPI\_Wtime();

    double T=end-start;

    printf("MPI\_dot结果 pi=%f\n",pi);

    }

    MPI\_Finalize();

    return 0;

}

pi\_bcast.c文件代码

#include<stdio.h>

#include<mpi.h>

double main(int argc, char \*argv[]){

    int id,n,i,N=0;

    double sum,step,x,pi,mypi;

    MPI\_Init(&argc, &argv);

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

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

    if(id==0){

        scanf("%d",&N);

    }

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

    step=1.0/N;

    sum=0.0;

    for(int i=id;i<N; i=i+n){

        x =(i-0.5)\*step;

        sum+=4.0/(1.0+x\*x);

    }

    pi+=sum\*step;

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

    if(id==0){

    printf("MPI\_Bcast结果为 pi=%f\n",mypi);

    }

    MPI\_Finalize();

    return 0;

}

pi\_gather.c文件代码

#include<stdio.h>

#include<mpi.h>

#include <stdlib.h>

#define N 10000000

double step=1.0/(double)N;

int main(int argc,char \*argv[]) {

    MPI\_Init(&argc,&argv);

    int i, id, num\_procs;

    double x,pi = 0.0;

    double buff = 0.0;

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

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

    double sum[num\_procs];

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

        x =step\*((double)i+0.5);

        buff += 4.0/(1.0+x\*x);

    }

    MPI\_Gather(&buff,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;

        printf("MPI\_Gather的结果为：pi=%f\n",pi);

    }

    MPI\_Finalize();

    return 0;

}

pi\_reduce.c文件代码

#include "mpi.h"

#include<stdio.h>

#include<math.h>

#define N 10000000

int main(int argc,char\*argv[]){

    double sum=0.0,pi,step,x=0.0;

    int i,rank,size;

    step=1.0/N;

    MPI\_Init(&argc,&argv);

    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("MPI\_Reduce结果为 pi=%f\n",pi\*step);

    }

    MPI\_Finalize();

    return 0;

}

pi\_mpi\_openmp.c文件代码

#include "mpi.h"

#include "omp.h"

#include <math.h>

#include <stdio.h>

#define num\_steps 1000000000

#define NUM\_THREADS 2

double main(int argc,char\* argv[]){

    int rank, nproc;

    int i,low,up;

    double sum = 0.0, pi, step, x,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();

    step = 1.0/num\_steps; low = rank\*(num\_steps / nproc); up = low + num\_steps/nproc - 1;

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

        for (i=low;i<up; i++){

            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("MPI+OpenMP混合结果：pi= %f ",pi\*step);

    t1=MPI\_Wtime();

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

    MPI\_Finalize();

}