姓名：李晖茜
学号：SA22218131

一、实验

0. OpenMP 测试程序

```
pp11@node1:~/mpi_share/SA22218131/openmp$ g++ -o test test.c -fopenmp -lm
pp11@node1:~/mpi_share/SA22218131/openmp$ ./test
```

1. OpenMP 实验内容

指令操作如下：

```
pp11@node1:~$ cd mpi_share/SA22218131/openmp
pp11@node1:~/mpi_share/SA22218131/openmp$ g++ -o test 1_1_1.c -fopenmp -lm
pp11@node1:~/mpi_share/SA22218131/openmp$ ./test
```

1.1.1

```
for i = 2 to 10 do   //循环 1
   for j = 2 to 10
      A[i,j] =  ( A[i-1,j-1] + A[i+1,j+1] )* 0.5;
   endfor
endfor
```

```
#include <stdio.h>
#include "omp.h"

int main(){
   int a[12][12];
    int b[12][12];
    for(int i = 0; i < 12; i++){
        for(int j = 0; j < 12; j++){
            b[i][j] = a[i][j] = i * j;
      }
    }
    for(int i = 2; i <= 10; i++){
        for(int j = 2; j <= 10; j++){
            a[i][j] = (a[i - 1][j - 1] + a[i + 1][j + 1]) * 0.5;
      }
    }
   for(int i = 2; i <= 10; i++){
#pragma omp parallel for
/*     Fork a team of threads      */
      for(int j = 2; j <= 10; j++){
         b[i][j] = (b[i - 1][j - 1] + b[i + 1][j + 1]) * 0.5;
      }
#pragma omp barrier
   }
```

```
    printf("using omp\n");
    for(int i = 2; i <= 10; i++){
         for(int j = 2; j <= 10; j++){
             printf("%d ",a[i][j]);
       }
     }
    printf("\n");

    printf("not using omp\n");
    for(int i = 2; i <= 10; i++){
         for(int j = 2; j <= 10; j++){
             printf("%d ",b[i][j]);
       }
     }
    printf("\n");
    return 0;
}
```

1.1.2

```
for i = 2 to 20 do   // 循环 2
   A[2*i+2] = A[2*i-2] + B[i];
endfor
```

不能并行化，因为存在方向向量为(1)

1.1.3

```
for i = 2 to 20 do  // 循环 3
    if A[i] > 0 then
        B[i] = C[i-1] + 1
    else
        C[i] = B[i] - 1
    endif
endfor
```

```
#include <stdio.h>
#include <stdlib.h>
#include "omp.h"

int main(){
    int A[30], B[30], C[30];
    int A1[30], B1[30], C1[30];
    for(int i=0;i<30;i++){
        A[i] = A[i] = rand()%2;
        B[i] = C[i] = 1;
        B1[i] = C1[i] = 1;
    }
#pragma omp parallel for
    for(int i=2;i<=20;i++){
        if(A[i]>0){
            B[i] = C[i-1] + 1;
```

```
        }
        else{
            C[i] = B[i] - 1;
        }
    }
    for(int i=2;i<=20;i++){
        if(A1[i]>0){
            B1[i] = C1[i-1] + 1;
        }
        else{
            C1[i] = B1[i] - 1;
        }
    }
    for(int i=0;i<30;i++){
        if(B[i]!=B1[i] or C[i]!=C1[i]){
            printf("Can't be paralleled.\n");
        }
    }
}
```

输出 Can't be paralleled.
不能并行化，存在方向向量为(1)

1.2.1

```
for i = 1 to M do   //循环 1  M , N, C 均是常量
   for j = 1 to N
      A[i+1,j+1] =  A[i,j] + C;
   endfor
endfor
```

(1) 给出迭代依赖示意图。
(2) 简述能否逆转外层的 i 循环？能否交换内外循环次序？

```
#include <stdio.h>
#include "omp.h"
#define C 1
#define M 10
#define N 10

int main(){
   int a[20][20];
   int b[20][20];
    for(int i=0;i<20;i++){
      for(int j=0;j<20;j++){
         a[i][j] = b[i][j] = 1;
      }
   }
   for(int i=1;i<=M;i++){
#pragma omp parallel for
/*     Fork a team of threads      */
      for(int j=1;j<=N;j++){
         a[i][j] = a[i+1][j+1] + C;
```

```
      }
#pragma omp barrier
   }
   for(int i=1;i<=M;i++){
      for(int j=1;j<=N;j++){
         b[i][j] = b[i+1][j+1] + C;
      }
   }
   printf("using omp\n");
   for(int i=0;i<10;i++){
      for(int j=0;j<10;j++){
         printf("%d ",a[i][j]);
      }
      printf("\n");
   }
   printf("not using omp\n");
   for(int i=0;i<10;i++){
      for(int j=0;j<10;j++){
         printf("%d ",b[i][j]);
      }
      printf("\n");
   }
   return 0;
}
```

1.2.2

```
for i = 1 to 100 do // 循环 2  N 是常量
    X[i] = Y[i] + 10; // 语句 S1
    for j = 1 to 100 do
        B[j] = A[j, N]; // 语句 S2
        for k = 1 to 100 do
            A[j+1, k] = B[j] + C[j, k]; // 语句 S3
        endfor // loop-k
        Y[i+j] = A[j+1, N]; // 语句 S4
    endfor // loop-j
endfor // loop-i

(1)   给出此循环的语句依赖图。
(2)   尝试向量化/并行化此循环。
```

```
#include <stdio.h>
#include "omp.h"
#define N 100

int main(){
   int X[200] ,Y[200] ,A[200][200] ,B[200] ,C[200][200];
   int X1[200] ,Y1[200] ,A1[200][200] ,B1[200] ,C1[200][200];
    for(int i=0;i<200;i++){
      for(int j=0;j<200;j++){
         A[i][j] = C[i][j] = 1;
         A1[i][j] = C1[i][j] = 1;
      }
```

```
        X[i] = Y[i] = B[i] = 1;
        X1[i] = Y1[i] = B1[i] = 1;
    }
    for(int i=1;i<=100;i++){
        X[i] = Y[i] + 10;
        for(int j=1;j<=100;j++){
            B[j] = A[j][N];
#pragma omp parallel for
            for(int k=1;k<=100;k++){
                A[j+1][k] = B[j] + C[j][k];
            }
#pragma omp barrier
            Y[i+j] = A[j+1][N];
        }
    }
    for(int i=1;i<=100;i++){
        X1[i] = Y1[i] + 10;
        for(int j=1;j<=100;j++){
            B1[j] = A1[j][N];
            for(int k=1;k<=100;k++){
                A1[j+1][k] = B1[j] + C1[j][k];
            }
            Y1[i+j] = A1[j+1][N];
        }
    }
    printf("using omp\n");
    for(int i=1;i<=100;i++){
        printf("%d ",Y[i]);
    }
    printf("\n");
    printf("not using omp\n");
    for(int i=1;i<=100;i++){
        printf("%d ",Y1[i]);
    }
    printf("\n");
    return 0;
}
```

1.3.1

```
for i = 1 to 100 do   //循环 1
   for j = 1 to 50 do
      A[3*i+2,2*j-1] =  A[5*j,i+3] + 2;
   endfor
endfor
```

（1） 给出满足依赖方向向量(1,1)的迭代依赖对集合的描述。

（2） 找出与迭代（i=11, j=11）相依赖的迭代（m,n）并指出是哪种依赖？

（3） 能否向量化最内层的 j 循环？如不行，简述理由。

```
#include <stdio.h>
#include "omp.h"
#define N 100
```

```
int main(){
   int A[301][301];
   int A1[301][301];
    for(int i=0;i<=300;i++){
      for(int j=0;j<=300;j++){
         A[i][j] = A1[i][j] = 1;
      }
   }
   for(int i=1;i<=100;i++){
#pragma omp parallel for
      for(int j=1;j<=50;j++){
            A[3*i+2][2*j-1] = A[5*j][i+3] + 2;
      }
#pragma omp barrier
   }
   for(int i=1;i<=100;i++){
      for(int j=1;j<=50;j++){
            A1[3*i+2][2*j-1] = A1[5*j][i+3] + 2;
      }
   }
    for(int i=0;i<=300;i++){
      for(int j=0;j<=300;j++){
         if(A[i][j]!=A1[i][j]){
            printf("Can't be paralleled.\n");
            break;
         }
      }
   }
   printf("using omp\n");
    for(int i=0;i<=10;i++){
      for(int j=0;j<=10;j++){
         printf("%d ",A[i][j]);
      }
      printf("\n");
   }
   printf("not using omp\n");
    for(int i=0;i<=10;i++){
      for(int j=0;j<=10;j++){
         printf("%d ",A1[i][j]);
      }
      printf("\n");
   }
   return 0;
}
```

输出Can't be paralleled.
i>16，j>10时，存在方向向量(0,1)的流依赖关系，所以内层循环不能向量化

1.3.2

```
S1: x = y * 2
    for i = 1 to 100 do
S2:   C[i] = B[i] + x
S3:   A[i] = C[i-1] + z
S4:   C[i+1] = A[i] * B[i]
       for j = 1 to 50 do
S5:      D[i,j] = D[i,j-1] + x
       endfor
    endfor
S6: z = y + 4
```

给出上述程序的语句依赖图。

```
#include <stdio.h>
#include "omp.h"

int main(){
    int A[200] ,B[200] ,C[200] ,D[200][200];
    int A1[200] ,B1[200] ,C1[200] ,D1[200][200];
    for(int i=0;i<=200;i++){
      for(int j=0;j<=200;j++){
         D[i][j] = D1[i][j] = 1;
      }
        A[i] = B[i] = C[i] = i;
        A1[i] = B1[i] = C1[i] = i;
   }
   int y = 1;
   int x = y * 2;
   int z = 3;
   for(int i=1;i<=100;i++){
      C[i] = B[i] + x;
      A[i] = C[i-1] + z;
      C[i+1] = A[i] * B[i];
#pragma omp parallel for
      for(int j=1;j<=50;j++){
         D[i][j] = D[i][j-1] + x;
      }
      z = y + 4;
#pragma omp barrier
   }

   int y1 = 1;
   int x1 = y1 * 2;
   int z1 = 3;
   for(int i=1;i<=100;i++){
      C1[i] = B1[i] + x1;
      A1[i] = C1[i-1] + z1;
      C1[i+1] = A1[i] * B1[i];
      for(int j=1;j<=50;j++){
         D1[i][j] = D1[i][j-1] + x1;
      }
      z1 = y1 + 4;
   }

    for(int i=0;i<=200;i++){
```

```
      for(int j=0;j<=200;j++){
         if(D[i][j]!=D1[i][j]){
            printf("Can't be paralleled.\n");
            break;
         }
      }
   }
   printf("using omp\n");
    for(int i=0;i<=10;i++){
      for(int j=0;j<=10;j++){
         printf("%d ",D[i][j]);
      }
      printf("\n");
   }
   printf("not using omp\n");
    for(int i=0;i<=10;i++){
      for(int j=0;j<=10;j++){
         printf("%d ",D1[i][j]);
      }
      printf("\n");
   }
   return 0;
}
```

输出Can't be paralleled.
存在方向向量(0,1)的流依赖关系。

1.4.1

```
for i = 2 to 10 do   //循环 1
   for j = i to 10
      A[i,j] =  ( A[i,j-1] + A[i-1,j] )* 0.5;
   endfor
endfor
```

不能并行化，内层循环存在方向向量(0,1)

1.4.2

```
for i = 1 to 16 do   // 循环 2
   A[i+3] = A[i] + B[i];
endfor
```

```
#include <stdio.h>
#include "omp.h"

int min(int a, int b){
    return a<b?a:b;
}
int main(){
    int A[20] ,B[20];
    int A1[20] ,B1[20];
    for(int i=0;i<=20;i++){
      A[i] = B[i] = 1;
```

```
      A1[i] = B1[i] = 1;
   }
   for(int k=1;k<=16;k+=3){
#pragma omp parallel for
      for(int i=k;i<=min(16,k+2);i++){
         A[i+3] = A[i] + B[i];
      }
#pragma omp barrier
   }
   for(int i=1;i<=16;i++){
      A1[i+3] = A1[i] + B1[i];
   }

   printf("using omp\n");
    for(int i=0;i<=20;i++){
      printf("%d ",A[i]);
   }
   printf("\n");
   printf("not using omp\n");
   for(int i=0;i<=20;i++){
      printf("%d ",A1[i]);
   }
   printf("\n");
   return 0;
}
```

1.4.3

```
for k = 1 to 16 step 5 do   // 循环 3 ,k 的循环步长为 5
   for i = k to min(16,i+4) do //设 min 为求最小值函数
      A[i+3] = A[i] + B[i]
   endfor
endfor
```

不能并行化，内层循环存在方向向量为(0,1)

1.5.1

```
for i = 1 to 100 do   //循环 1
   A[i] =  A[i] + B[i-1];
   B[i] =  C[i-1] * 2 ;
   C[i] =  1 / B[i] ;
   D[i] =  C[i] * C[i] ;
endfor
```

可以并行化

```
#include <stdio.h>
#include <math.h>
#include "omp.h"

int main(){
    double A[200] ,B[200] ,C[200] ,D[200];
    double A1[200] ,B1[200] ,C1[200] ,D1[200];
```

```
    for(int i=0;i<200;i++){
      A[i] = B[i] = C[i] = D[i] = 1;
      A1[i] = B1[i] = C1[i] = D1[i] = 1;
   }
   for(int i=1;i<=100;i++){
      B[i] = C[i-1] * 2;
      C[i] = 1.0 / B[i];
   }
#pragma omp parallel for
   for(int i=1;i<=100;i++){
      A[i] = A[i] + B[i-1];
   }
#pragma omp parallel for
   for(int i=1;i<=100;i++){
      D[i] = C[i] * C[i];
   }

   for(int i=1;i<=100;i++){
      B1[i] = C1[i-1] * 2;
      C1[i] = 1.0 / B1[i];
      A1[i] = A1[i] + B1[i-1];
      D1[i] = C1[i] * C1[i];
   }

   printf("using omp\n");
    for(int i=1;i<=100;i++){
      printf("%.2lf ",A[i]);
   }
   printf("\n");
   printf("not using omp\n");
   for(int i=1;i<=100;i++){
      printf("%.2lf ",A1[i]);
   }
   printf("\n");
   return 0;
}
```

1.5.2

```
for i = 1 to 999 do   // 循环 2
   A[i] = B[i] + C[i];
   D[i] = ( A[i] + A[ 999-i+1 ] ) / 2 ;
endfor
```

采用圈收缩拆开，可以分别向量化/并行化

```
#include <stdio.h>
#include <math.h>
#include "omp.h"

int main(){
    double A[1000] ,B[1000] ,C[1000] ,D[1000];
    for(int i=0;i<1000;i++){
      A[i] = B[i] = C[i] = D[i] = 1;
   }
```

```
    for(int i=1;i<=999;i++){
        A[i] = B[i] + C[i];
        D[i] = A[i] + A[999 - i + 1];
    }

    printf("not using omp A\n");
    for(int i=1;i<=50;i++){
        printf("%.2lf ",A[i]);
    }
    printf("\n");
    printf("not using omp D\n");
    for(int i=1;i<=50;i++){
        printf("%.2lf ",D[i]);
    }
    printf("\n");
    return 0;
}
```

1.5.3

```
for i = 1 to 100 do   // 循环 3
   for j = 1 to 100 do
      A[3*i+2*j, 2*j] = C[i,j] * 2     ;
      D[i,j]          = A[i-j+6, i+j] ;
   endfor
endfor
```

可以并行化

```
#include <stdio.h>
#include <math.h>
#include "omp.h"

int main(){
    int A[501][501] ,C[501][501] ,D[501][501];
    int A1[501][501] ,C1[501][501] ,D1[501][501];
    for(int i=0;i<500;i++){
      for(int j=0;j<500;j++){
         A[i][j] = C[i][j] = D[i][j] = 1;
         A1[i][j] = C1[i][j] = D1[i][j] = 1;
      }
    }

    for(int i=1;i<=100;i++){
#pragma omp parallel for
      for(int j=1;j<=100;j++){
         A[3*i+2*j][2*j] = C[i][j] * 2;
         if(i-j+6 >= 0){
            D[i][j] = A[i-j+6][i+j];
         }
      }
    }

    for(int i=1;i<=100;i++){
      for(int j=1;j<=100;j++){
         A1[3*i+2*j][2*j] = C1[i][j] * 2;
```

```
            if(i-j+6 >= 0){
                D1[i][j] = A1[i-j+6][i+j];
            }
        }
    }

    for(int i=0;i<500;i++){
        for(int j=0;j<500;j++){
            if(A[i][j] != A1[i][j] || C[i][j] != C1[i][j] || D[i][j] != D1[i]
[j]){
                printf("Can't be paralleled.\n");
            }
        }
    }
    return 0;
}
```

2. MPI 实验内容

编译参考：https://scc.ustc.edu.cn/zlsc/user_doc/html/mpi-application/mpi-application.html

```
pp11@node1:~$ cd mpi_share/SA22218131/mpi
pp11@node1:~/mpi_share/SA22218131/openmp$ mpicc -o test 2_a.c -lm
pp11@node1:~/mpi_share/SA22218131/openmp$ mpirun -n 8 ./test
```

2.a

（1.1）写个将 MPI 进程按其所在节点分组的程序；（1.2）在 1.1 的基础上，写个广播程序，主要思想是：按节点分组后，广播的 root 进程将消息“发送”给各组的“0 号”，再由这些“0”号进程在其小组内执行 MPI_Bcast。

```
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
#define NODES_NUM 2

int main(int argc, char* argv[]){
    int rank, size;
    char processor_name[MPI_MAX_PROCESSOR_NAME];//MPI_MAX_PROCESSOR_NAME：似乎是一
个整数，值为128.
    int nameLen;
    MPI_Status status;

    //MPI_COMM_WORLD：通讯子，“一组可以互发消息的进程集合”
    MPI_Init(&argc, &argv);//初始化
    MPI_Comm_size(MPI_COMM_WORLD, &size); //通信子中总进程数
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);//进程号
    MPI_Get_processor_name(processor_name, &nameLen);//得到处理器的名字
    MPI_Comm splitWorld;//进程分通讯域
    int rank2, size2, color;
    color = processor_name[nameLen-1] - '0';
    MPI_Comm_split(MPI_COMM_WORLD, color, rank, &splitWorld);
    MPI_Barrier(MPI_COMM_WORLD);//跨组的所有成员启动屏障同步
    MPI_Comm_size(splitWorld, &size2);
    MPI_Comm_rank(splitWorld, &rank2);
```

```
    printf("processor_name: %s,rank1:%d,rank2:%d\n", processor_name, rank,
rank2);
    int sendData, recvData, tag;
    recvData = 0;
    if (rank == 0) {
        sendData = 9999;
        for (tag = 1; tag <= NODES_NUM; tag++) {
            //发送函数
            MPI_Send(&sendData, 1, MPI_INT, (tag-1)*(size/NODES_NUM), tag,
MPI_COMM_WORLD);
        }
    }
    if (rank2 == 0) {
        //接收函数
        MPI_Recv(&recvData, 1, MPI_INT, 0, color, MPI_COMM_WORLD, &status);
    }

    MPI_Bcast(&recvData, 1, MPI_INT, 0, splitWorld);//将数据从组的一个成员广播到组的所
有成员
    printf("processor_name: %s,rank: %d, RecvData: %d\n", processor_name, rank2,
recvData);

    MPI_Comm_free(&splitWorld);
    MPI_Finalize();//MPI程序结束
    return 0;
}
```

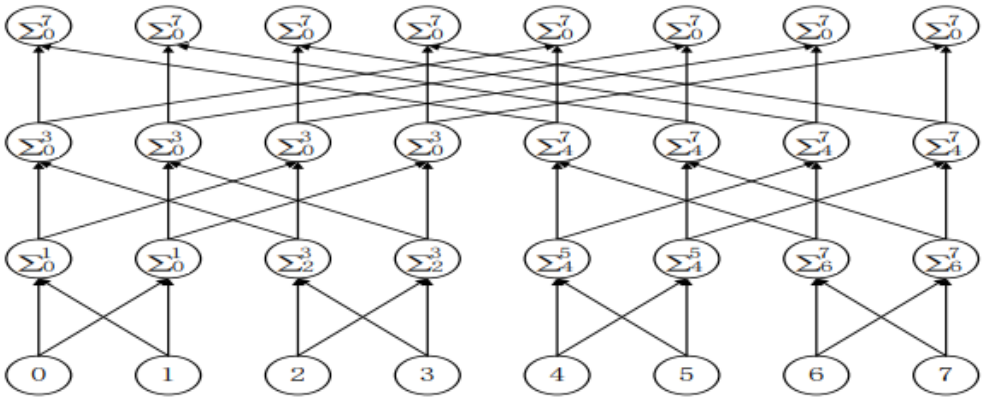
输出:

```
pp11@node1:~/mpi_share/SA22218131/mpi$ mpicc -o test 2_a.c -lm
pp11@node1:~/mpi_share/SA22218131/mpi$ mpirun -n 8 ./test
processor_name: node1,rank1:0,rank2:0
processor_name: node1,rank: 0, RecvData: 9999
processor_name: node1,rank1:1,rank2:1
processor_name: node1,rank: 1, RecvData: 9999
processor_name: node1,rank1:2,rank2:2
processor_name: node1,rank: 2, RecvData: 9999
processor_name: node1,rank1:3,rank2:3
processor_name: node1,rank: 3, RecvData: 9999
processor_name: node1,rank1:4,rank2:4
processor_name: node1,rank: 4, RecvData: 9999
processor_name: node1,rank1:5,rank2:5
processor_name: node1,rank: 5, RecvData: 9999
processor_name: node1,rank1:6,rank2:6
processor_name: node1,rank: 6, RecvData: 9999
processor_name: node1,rank1:7,rank2:7
processor_name: node1,rank: 7, RecvData: 9999
```

2.b

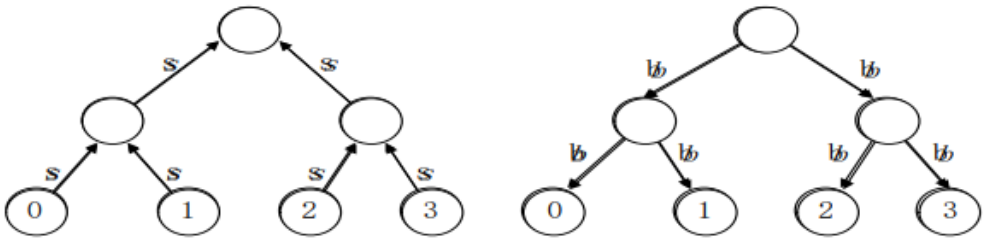
（b） N 个处理器求 N 个数的全和，要求每个处理器均保持全和。

（1）蝶式全和的示意图如下：由于使用了重复计算，共需 logN 步。



给出蝶式全和计算的 MPI 程序实现（设 N 为 2 的幂次方）。

（2）二叉树方式求全和示意图如下：需要 2logN 步。



给出二叉树方式全和计算的 MPI 程序实现。

(1) 蝶式计算

```
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
#include<math.h>

int main(int argc, char* argv[]){
    int rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int data = rank+1;
    int recvdata;
    MPI_Status status;
    printf("process id %d data = %d\n",rank, data);
    int logN = (int)log2(size);
    for(int i = 0; i < logN; i++) {
        int tag = i+1;
        int step = (int)pow(2,i);
```

```
        int dest = rank ^ step;
        MPI_Send(&data, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
        MPI_Recv(&recvdata, 1, MPI_INT, dest, tag, MPI_COMM_WORLD, &status);
        data += recvdata;
    }


    printf("process id %d sum is = %d\n",rank, data);

    MPI_Finalize();
    return 0;
}
```

输出：

```
pp11@node1:~/mpi_share/SA22218131/mpi$ mpirun -n 8 ./test
process id 2 data = 3
process id 0 data = 1
process id 3 data = 4
process id 5 data = 6
process id 7 data = 8
process id 1 data = 2
process id 4 data = 5
process id 6 data = 7
process id 7 sum is = 36
process id 1 sum is = 36
process id 3 sum is = 36
process id 5 sum is = 36
process id 6 sum is = 36
process id 2 sum is = 36
process id 0 sum is = 36
process id 4 sum is = 36
```

（2）二叉树

```
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
#include<math.h>

int main(int argc, char* argv[]){
    int rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int data = rank+1;
    int recvdata;
    MPI_Status status;
    printf("process id %d data = %d\n",rank, data);
    int logN = (int)log2(size);
    for(int i = 0; i < logN; i++) {
        int tag = i+1;
        int step = (int)pow(2,i);
        int dest = rank ^ step;
        MPI_Send(&data, 1, MPI_INT, dest, tag, MPI_COMM_WORLD);
        MPI_Recv(&recvdata, 1, MPI_INT, dest, tag, MPI_COMM_WORLD, &status);
```

```
        data += recvdata;
    }


    printf("process id %d sum is = %d\n",rank, data);

    MPI_Finalize();
    return 0;
}
```

输出：

```
pp11@node1:~/mpi_share/SA22218131/mpi$ mpicc -o test 2_b_2.c -lm
pp11@node1:~/mpi_share/SA22218131/mpi$ mpirun -n 8 ./test
process id: 5,data:6
process id: 7,data:8
process id: 1,data:2
process id: 2,data:3
process id: 3,data:4
process id: 6,data:7
process id: 0,data:1
process id: 4,data:5
4 sum is 36
6 sum is 36
7 sum is 36
1 sum is 36
2 sum is 36
0 sum is 36
3 sum is 36
5 sum is 36
```

2.c

（c） 《并行算法实践》单元 V 习题 v-3。给出 FOX 矩阵相乘并行算法的 MPI 实现。

V－3 矩阵相乘的另一种并行算法是 Fox 算法(Fox Algorithm)：将待相乘的矩阵 A 和 B 分成 p 个方块 $A_{i,j}$ 和 $B_{i,j}(0 \leqslant i,j \leqslant \sqrt{p}-1)$，每块大小为 $(n/\sqrt{p}) \times (n/\sqrt{p})$，并将它们分配给 $\sqrt{p} \times \sqrt{p}$ 个处理器 $(P_{0,0}, P_{0,1}, \cdots, P_{\sqrt{p}-1,\sqrt{p}-1})$。开始时处理器 $P_{i,j}$ 存放有块 $A_{i,j}$ 和 $B_{i,j}$，并负责计算块 $C_{i,j}$。然后 Fox 算法执行以下各步 $\sqrt{p}$ 次迭代，即可完成：

① 选中对角块 $A_{i,i}$ 并将其向所在行的 $\sqrt{p}-1$ 个处理器进行一到多播送；

② 各处理器将所收到的 A 阵的块与 B 阵原有的块进行乘－加运算；

③ B 阵的块向上循环 1 步；

④ 如果 $A_{i,j}$ 是本次播送的块，则下次应选块 $A_{i,(j+1) \bmod \sqrt{p}}$ 向同行的 $\sqrt{p}-1$ 个处理器播送，然后转第②步。

请写出 Fox 算法的并行程序，并用适当的编译环境调试运行。

```
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
#include<math.h>
#define N 8//设定N大点，为了让处理器小于N
#define INDEX(i, j, N) (((i)*(N))+(j))
```

```
// 打印矩阵
void print_mat(int *a, int num){
    for(int i = 0; i < num; i++){
        for(int j = 0; j < num; j++){
            printf("%d\t", a[INDEX(i, j, num)]);
            }
            printf("\n");
            }
}

int main(int argc,char *argv[]){
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);//分组
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    int A[N][N];
    int B[N][N];//因为可能会越界我们这里用n+2作为大小
    int C[N][N];
    int T[N][N];
    //我们这里把矩阵分给4个处理器，根号p也就是2，所以每块的大小是4*4，因为实验平台只能支持处理
器的的数量只能为4，不能为9,16
    for(int i=0;i<N;i++){//初始化A，所有线程都生成相同的数据，不用进行广播数据
        for(int j=0;j <N;j++){
            A[i][j]=i+j+1;
            B[i][j]=2*N-i-j+1;
            C[i][j]=0;
        }
    }
    //    for(int i=0;i<N;i++){//初始化A，所有线程都生成相同的数据，不用进行广播数据
    //      for(int j=0;j <N;j++){
    //          for(int k=0;k<N;k++)
    //             C[i][j]+=A[i][k]*B[k][j];
    //      }
    //  }
    int BN=sqrt(size);
    int TN=size;
    int row=rank/BN;
    int line=rank%BN;
    int times=0;
    MPI_Datatype block;
    MPI_Type_vector(N/2, N/2, N, MPI_INT, &block);//定义新的数据类型
    MPI_Type_commit(&block);
    for(int i=0;i<BN;i++){//两次循环
        for(int j=0;j<BN;j++){//广播A块
            if(row==j&&line==(j+times)%BN){//向同行元素广播A
                for(int k=0;k<BN;k++){
                    MPI_Send(&A[row*size]
[line*size],1,block,row*BN+k,0,MPI_COMM_WORLD);//发送A
                }
            }
            if(row==j)      //接受A,放到T中
                MPI_Recv(&T[row*size][line*size],1,block,row*BN+
(j+times)%BN,MPI_ANY_TAG,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        }
        MPI_Barrier(MPI_COMM_WORLD);//等待传送完成
        for(int j=row*TN;j<row*TN+TN;j++){
            for(int k=line*TN;k<line*TN+TN;k++){
```

```
                for(int p=0;p<TN;p++)
                    C[j][k]+=T[j][line*TN+p]*B[row*TN+p][k];//结果放到C中
                //printf("%d:%d=%d*%d:%d+%d\t",rank,C[j][k],T[j][k],B[j]
[k],j,k);
            }
        }
        MPI_Barrier(MPI_COMM_WORLD);//等待计算完成
        //B矩阵的块向上传播一块，这个容易造成死锁
        MPI_Send(&B[row*TN][line*TN],1,block,(row-
1+BN)%BN*BN+line,0,MPI_COMM_WORLD);
        MPI_Recv(&T[row*TN][line*TN],1,block,
(row+1)%BN*BN+line,MPI_ANY_TAG,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        // printf("走过三");
        // MPI_Barrier(MPI_COMM_WORLD);//等待传输结束
        for(int j=row*TN;j<row*TN+TN;j++)
            for(int k=line*TN;k<line*TN+TN;k++)
                B[j][k]=T[j][k];//更新B

        MPI_Barrier(MPI_COMM_WORLD);//等待更新结束
        times++;
    }
    MPI_Barrier(MPI_COMM_WORLD);//等待更新结束
    //开始汇总数据到0号进程
    for(int i=1;i<size;i++){
        if(rank==i)
            MPI_Send(&C[row*TN][line*TN],1,block,0,0,MPI_COMM_WORLD);
        if(rank==0)
            MPI_Recv(&C[i/BN*TN]
[i%BN*TN],1,block,i,MPI_ANY_TAG,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }
    if(rank==0){
        printf("C:\n");
        for(int i=0;i<N;i++){//初始化A，所有线程都生成相同的数据，不用进行广播数据
            for(int j=0;j <N;j++){
                printf("%d \t",C[i][j]);
            }
            printf("\n");
        }
        printf("A:\n");
        print_mat(*A, N);
        printf("B:\n");
        print_mat(*B, N);
    }


    MPI_Finalize();
    return 0;
}
```

输出：

```
pp11@node1:~/mpi_share/SA22218131/mpi$ mpicc -o test 2_c.c -lm
pp11@node1:~/mpi_share/SA22218131/mpi$ mpirun -n 4 ./test
C:
444     408     372     336     300     264     228     192
552     508     464     420     376     332     288     244
660     608     556     504     452     400     348     296
768     708     648     588     528     468     408     348
876     808     740     672     604     536     468     400
984     908     832     756     680     604     528     452
1092    1008    924     840     756     672     588     504
1200    1108    1016    924     832     740     648     556
A:
1       2       3       4       5       6       7       8
2       3       4       5       6       7       8       9
3       4       5       6       7       8       9       10
4       5       6       7       8       9       10      11
5       6       7       8       9       10      11      12
6       7       8       9       10      11      12      13
7       8       9       10      11      12      13      14
8       9       10      11      12      13      14      15
B:
17      16      15      14      13      12      11      10
16      15      14      13      12      11      10      9
15      14      13      12      11      10      9       8
14      13      12      11      10      9       8       7
13      12      11      10      9       8       7       6
12      11      10      9       8       7       6       5
11      10      9       8       7       6       5       4
10      9       8       7       6       5       4       3
```

2.d

（d） 参数服务器系统的 MPI 模拟。

设系统中总计有 N 个进程，其中 P 个进程作为参数*服务器进程*，而 Q 个进程作为*工作进程*（N = P + Q， 且 0 < P << Q）。工作进程和服务器进程的互动过程如下：

1. 第 i 个工作进程首先产生一个随机数，发送给第 i%P 个参数服务器进程。然后等待并接收它对应的参数服务器进程发送更新后的数值，之后，再产生随机数，再发送……。
2. 每个参数服务器进程等待并接收来自它对应的所有工作进程的数据，在此之后，经通信，使所有的参数服务器获得所有工作进程发送数据的平均值。
3. 每个参数服务器发送该平均值给它对应的所有工作进程，然后再等待……。

试给出上述互动过程的 MPI 程序实现。

```
#include<stdio.h>
#include<stdlib.h>
```

```
#include<mpi.h>
#include<math.h>
#include<time.h>
#define N 3

int main(int argc,char *argv[]){
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);//分组
    MPI_Comm_size(MPI_COMM_WORLD,&size);

    MPI_Comm severworld;
    MPI_Group tempgroup,severgroup;
    MPI_Comm_group(MPI_COMM_WORLD,&tempgroup);//创建无通讯能力组
    int ranks[N]={0,1,2};
    MPI_Group_incl(tempgroup,3,ranks,&severgroup);//选取前三个进程为服务器
    MPI_Comm_create(MPI_COMM_WORLD,severgroup,&severworld);//赋予通讯能力
    for(int i=0;i<10;i++){//进行10次消息传递
        if(rank<N){//服务器进程
            int left=size-3;//算出还有多少进程
            int local_sum=0;//当前保存数据
            int count=0;
            for(int j=rank+N;j<size;j+=N){//获取数据
                int tempnum;

 MPI_Recv(&tempnum,1,MPI_INT,j,MPI_ANY_TAG,MPI_COMM_WORLD,MPI_STATUS_IGNORE);//从工作进程获取随机数
                local_sum+=tempnum;
                count++;
            }
            MPI_Barrier(severworld); //等待拿到所有数字
            int ave=local_sum/count;//获得当前服务器的平均数
            int res=ave;//存放最终平均数
            for(int j=0;j<N;j++){//发给其他人并且拿到平均数
                int tempnum;
                if(j!=rank){//不是自己就发送再接收
                    MPI_Send(&ave,1,MPI_INT,j,0,MPI_COMM_WORLD);

 MPI_Recv(&tempnum,1,MPI_INT,j,MPI_ANY_TAG,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
                    res+=tempnum;
                }
            }
            MPI_Barrier(severworld);//等待收集完成
            res=res/(size-3);
            printf("rank%dgetres%d\n",rank,res);
            //向工作进程发送平均数
            for(int j=rank+N;j<size;j+=N){
                MPI_Send(&res,1,MPI_INT,j,0,MPI_COMM_WORLD);
            }
        }
        else{//工作进程
            srand((unsigned)time(NULL));
            int random = rand()%1000+1;//生成1000以内随机数
            int pos=rank-N;//得到当前工作进程序号
            int rp=pos%N;//拿到接收的服务器进程序号
            MPI_Send(&random,1,MPI_INT,rp,0,MPI_COMM_WORLD);//发送
            int res;
```

```
 MPI_Recv(&res,1,MPI_INT,rp,MPI_ANY_TAG,MPI_COMM_WORLD,MPI_STATUS_IGNORE);//拿到结果
            printf("rank%d get res %d\n",rank,res);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 0;
}
```

输出：

```
pp11@node1:~/mpi_share/SA22218131/mpi$ mpicc -o test 2_d.c -lm
pp11@node1:~/mpi_share/SA22218131/mpi$ mpirun -n 8 ./test
rank0getres174
rank1getres174
rank1getres174
rank2getres174
rank2getres174
rank3getres174
rank3getres174
rank4getres174
rank4getres174
rank5getres174
rank5getres174
rank6getres174
rank6getres174
rank7getres174
rank7getres174
rank7getres174
rank0getres174
rank0getres174
rank1getres174
rank2getres174
rank3getres174
rank4getres174
rank5getres174
rank6getres174
rank7getres174
rank7getres174
rank0getres174
```

2.e

（e） 矩阵 A 和 B 均为 N*N 的双精度数矩阵，有 P 个处理器。针对以下程序片段，分别采用按行块连续划分以及棋盘式划分方式，给出相应的 MPI 并行实现。

```
for(i=1; i<N-1; i++)
    for( j=1;j<N-1; j++)
        B[i][j] = (A[i-1][j] + A[i][j+1] + A[i+1][j] + A[i][j-1]) / 4.0
```

（1）按行块连续划分

```
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
#include<math.h>
#include<time.h>
#define N 8
```

```c
int main(int argc,char *argv[]){
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);//分组
    MPI_Comm_size(MPI_COMM_WORLD,&size);

    double A[N+2][N+2];
    double B[N+2][N+2];//因为可能会越界我们这里用n+2作为大小
    for(int i=0;i<N+2;i++){//初始化A，所有线程都生成相同的数据，不用进行广播数据
        for(int j=0;j <N+2;j++){
            A[i][j]=i+j;
        }
    }
    int count=N/size;
    int left=N%size;
    if(rank<left)
        count++;//count表明了对应处理器要进行多少行的计算
    for(int i=0;i<count;i++){//计算数据
        int index=rank+i*size;//对应第几行
        for(int j=1;j<N-1;j++)
            B[index][j]=(A[index-1][j]+A[index][j+1]+A[index+1][j]+A[index][j-
1])/4.0;
    }
    MPI_Barrier(MPI_COMM_WORLD);
    //将数据都传递给0号线程
    for(int i=0;i<N;i++){
        if(i%size==rank)//如果不是0向0发送数据
            MPI_Send(&B[i][1],N,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
        if(rank==0)//如果是0就接受数据
            MPI_Recv(&B[i]
[1],N,MPI_DOUBLE,i%size,MPI_ANY_TAG,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }
    MPI_Barrier(MPI_COMM_WORLD);
    if(rank==0){
        for(int i=1;i<N-1;i++){
            for(int j=1;j <N-1;j++){
                printf("%.2lf \t",B[i][j]);
            }
            printf("\n");
        }
    }

    MPI_Finalize();
    return 0;
}
```

输出：

```
pp11@node1:~/mpi_share/SA22218131/mpi$ mpicc -o test 2_e_1.c -lm
pp11@node1:~/mpi_share/SA22218131/mpi$ mpirun -n 4 ./test
2.00    3.00    4.00    5.00    6.00    7.00
3.00    4.00    5.00    6.00    7.00    8.00
4.00    5.00    6.00    7.00    8.00    9.00
5.00    6.00    7.00    8.00    9.00    10.00
6.00    7.00    8.00    9.00    10.00   11.00
7.00    8.00    9.00    10.00   11.00   12.00
```

（2）棋盘式划分

```c
#include<stdio.h>
#include<stdlib.h>
#include<mpi.h>
#include<math.h>
#include<time.h>
#define N 8

int main(int argc,char *argv[]){
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);//分组
    MPI_Comm_size(MPI_COMM_WORLD,&size);

    double A[N+2][N+2];
    double B[N+2][N+2];//因为可能会越界我们这里用n+2作为大小
    for(int i=0;i<N+2;i++){//初始化A，所有线程都生成相同的数据，不用进行广播数据
        for(int j=0;j <N+2;j++){
            A[i][j]=i+j;
        }
    }
    int BN=sqrt(size);
    int h=N/sqrt(size);
    int w=N/sqrt(size);//定义高宽，我们假设可以整除
    int row=rank/BN;
    int line=rank/BN;

    MPI_Datatype block;
    MPI_Type_vector(h, w, N+2, MPI_DOUBLE, &block);//定义新的数据类型
    MPI_Type_commit(&block);
    for(int i=row*h;i<row*h+h;i++){//计算数据
        for(int j=line*w;j<line*w+w;j++){
            if(i>0&&j>0)
                B[i][j]=(A[i-1][j]+A[i][j+1]+A[i+1][j]+A[i][j-1])/4.0;
            else if(i==0&&j==0)
                B[i][j]=(A[i][j+1]+A[i+1][j])/2.0;
            else if(i==0)
                B[i][j]=(A[i][j+1]+A[i+1][j]+A[i][j-1])/3.0;
            else if(j==0)
                B[i][j]=(A[i-1][j]+A[i][j+1]+A[i+1][j])/3.0;
        }
    }

    MPI_Barrier(MPI_COMM_WORLD);//等待计算结束
    for(int i=1;i<size;i++){
        if(i==rank)//如果不是0向0发送数据
        MPI_Send(&B[row*h][line*w],1,block,0,0,MPI_COMM_WORLD);
        if(rank==0)//如果是0就接受数据
        MPI_Recv(&B[i/BN*h]
[i%BN*w],1,block,i,MPI_ANY_TAG,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }
    if(rank==0){
        for(int i=1;i<N-1;i++){
            for(int j=1;j <N-1;j++){
                printf("%.2lf \t",B[i][j]);
            }
            printf("\n");
        }
    }
```

```
    MPI_Finalize();
    return 0;
}
```

输出：

```
pp11@node1:~/mpi_share/SA22218131/mpi$ mpicc -o test 2_e_2.c -lm
pp11@node1:~/mpi_share/SA22218131/mpi$ mpirun -n 4 ./test
2.00    3.00    4.00    1.33    2.00    3.00
3.00    4.00    5.00    2.33    3.00    4.00
4.00    5.00    6.00    3.33    4.00    5.00
9.00    10.00   11.00   8.00    9.00    10.00
10.00   11.00   12.00   9.00    10.00   11.00
11.00   12.00   13.00   10.00   11.00   12.00
```

二、个人实验

结合你自己研究方向，设计一个采用并行计算来求解的问题。简要描述你的问题，并从 PCAM 角度来分析你的并行设计方案，给出你的并行实现及加速性能分析与评测（可以选择 MPI 或 OpenMP 或混合并行）。

问题：使用MPI实现快速排序

方案：

快速排序的基本思想是在待排序的n个记录中任取一个记录（通常取第一个记录）作为基准，把该记录放入适当位置后，数据序列被此记录划分成两部分，分别是比基准小和比基准大的记录；然后再对基准两边的序列用同样的策略，分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

利用多进程设计进行并行快速排序，需要log2(n)个进程，其中0号进程进行初始分块，调用partition()函数将数组分成两部分，将后一半调用MPI_Send()将数据发送到 $2^{(n-1)}$ 线程；随后将进程分成对应的发送进程和接收进程两部分，最后，对每个进程进行串行快速排序；从各进程收集结果，完成并行排序。

将需要用到的辅助函数写在`quickSort.h`，主函数写在`sort.c`。

编译：

```
pp11@node1:~$ cd mpi_share/SA22218131/personal
pp11@node1:~/mpi_share/SA22218131/openmp$ mpicc -o sort sort.c -lm
pp11@node1:~/mpi_share/SA22218131/openmp$ mpirun -n 8 ./sort
```

实现：

```
#include <stdio.h>
#include "mpi.h"
#include <time.h>
#include <stdlib.h>
#include "quickSort.h"
#define n 10000000

double whi_sta,whi_end;

//并行排序
void para_quickSort(int *data,int sta,int end,int whi_m,int id,int now_id){
    //printf("1\n");
    int whi_r,whi_j,whi_i;
    int MyLength = -1;
    int *tmp;
```

```
    MPI_Status status;
    //可剩处理器为0，进行串行快速排序排序
    if(whi_m == 0){
        whi_sta = MPI_Wtime();
        if(now_id == id){
            quickSort(data,sta,end);
        }
        return ;
    }
    //由当前处理器进行分块
    if(now_id == id){
        whi_r = partition(data,sta,end);
        MyLength = end - whi_r;
        MPI_Send(&MyLength,1,MPI_INT,id+exp2(whi_m-1),now_id,MPI_COMM_WORLD);
        if(MyLength != 0){
    MPI_Send(data+whi_r+1,MyLength,MPI_INT,id+exp2(whi_m-
1),now_id,MPI_COMM_WORLD);
        }
    }
    if(now_id == id+exp2(whi_m-1)){
        MPI_Recv(&MyLength,1,MPI_INT,id,id,MPI_COMM_WORLD,&status);

        if(MyLength != 0){
            tmp = (int*)malloc(sizeof(int)*MyLength);

            MPI_Recv(tmp,MyLength,MPI_INT,id,id,MPI_COMM_WORLD,&status);
        }
    }
    whi_j = whi_r-1-sta;
    MPI_Bcast(&whi_j,1,MPI_INT,id,MPI_COMM_WORLD);
    if(whi_j > 0){
        para_quickSort(data,sta,whi_r-1,whi_m-1,id,now_id);
    }
    whi_j = MyLength;
    MPI_Bcast(&whi_j,1,MPI_INT,id,MPI_COMM_WORLD);
    if(whi_j > 0){
        para_quickSort(tmp,0,MyLength-1,whi_m-1,id+exp2(whi_m-1),now_id);
    }
    if(now_id == id+exp2(whi_m-1)&&MyLength != 0){
        MPI_Send(tmp,MyLength,MPI_INT,id,id+exp2(whi_m-1),MPI_COMM_WORLD);
    }
    if((now_id == id) && (MyLength != 0))
        MPI_Recv(data+whi_r+1,MyLength,MPI_INT,id+exp2(whi_m-1),id+exp2(whi_m-
1),MPI_COMM_WORLD,&status);
}

int main(int argc,char *argv[])
{
    int *data1,*data2;
    int now_id,sum_id;
    int whi_m,whi_r;
    int whi_i,whi_j;

    MPI_Status status;
    //启动mpi
    MPI_Init(&argc,&argv);
    //确定自己的进程标志符now_id
    MPI_Comm_rank(MPI_COMM_WORLD,&now_id);
```

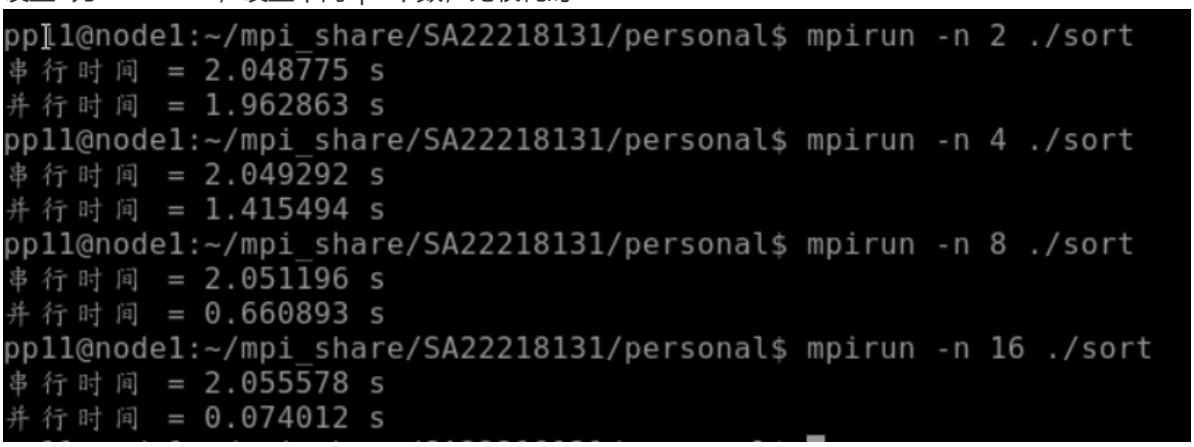
```
    //组内进程数是sum_id
    MPI_Comm_size(MPI_COMM_WORLD,&sum_id);
    double whi_n;
    if(now_id == 0){
        //初始化串行和并行所需排序的随机数组
        whi_n = n;
        data1 = (int*)malloc(sizeof(int)*whi_n);
        data2 = (int*)malloc(sizeof(int)*whi_n);
        rands(data1,n);
        rands(data2,n);
        //打印初始数组
        // printf("---------\n");
        // printf("The original array data1: \n");
        // print(data1,n);
        // printf("The original array data2: \n");
        // print(data2,n);
        // printf("---------\n");
    }
    whi_m = log2(sum_id);
MPI_Bcast(&whi_n,1,MPI_INT,0,MPI_COMM_WORLD);

    //并行排序
    para_quickSort(data1,0,whi_n-1,whi_m,0,now_id);
    whi_end = MPI_Wtime();
    if(now_id == 0){
        double sta_time2 = MPI_Wtime();
        quickSort(data2,0,n-1);
        double end_time2 = MPI_Wtime();
        // printf("----------\n");
        // printf("The final array data1 : \n");
        // print(data1,n);
        // printf("The final array data2 : \n");
        // print(data2,n);
        // printf("----------\n");
        printf("串行时间 = %f s\n",end_time2-sta_time2);
        printf("并行时间 = %f s\n",whi_end-whi_sta);
    }
    MPI_Finalize();
    return 0;
}
```

结果：

设置n为10，对data1并行排序，对data2串行排序，输出排序结果和耗时，发现数据较少时串行耗时小。

```
pp11@node1:~/mpi_share/SA22218131/personal$ mpirun -n 8 ./sort
---------
The original array data1:
83      86      77      15      93      35      86      92      49      21
The original array data2:
62      27      90      59      63      26      40      26      72      36
---------
----------
The final array data1 :
15      21      35      49      77      83      86      86      92      93
The final array data2 :
26      26      27      36      40      59      62      63      72      90
----------
串行时间 = 0.000001 s
并行时间 = 0.000233 s
```

设置n为10000000，设置不同cpu个数，比较耗时：



可见并行效率明显高于串行。