高级MPI编程技术

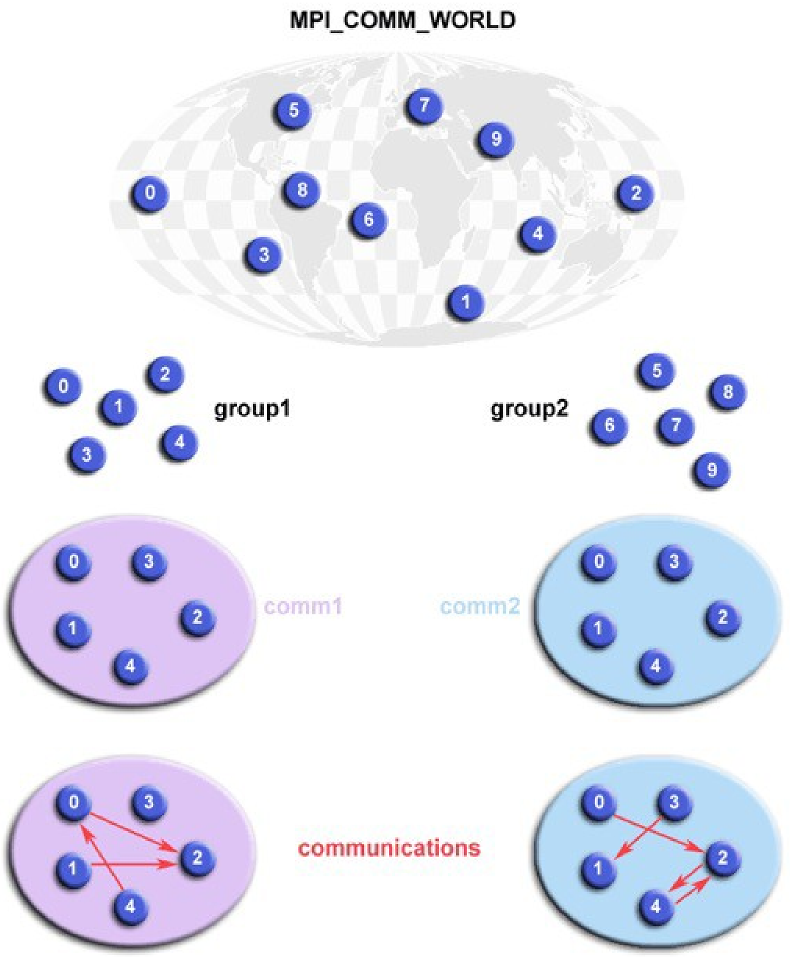
Lecture 10-01: MPI编程——虚拟拓扑

肖俊敏

中国科学院计算技术研究所

简介

- 统一的机制或对象来方便地指示通信上下文，通信进程的组，嵌入抽象进程命名，存储装饰
- **通信上下文**
- **进程组**
- **虚拟拓扑**
- **通信子**



基本概念

- **组**
  - 一个组是进程标志符（后面的进程〕的一个有序集。进程是与实现相关的对象。组内的每个进程与一个整数rank相联系。序列号是连续的并从0开始。组用模糊的组对象来描述，因此不能直接从一个进程到另一个进程传送。可在一个通信子使用组来描述通信空间中的参与者并对这些参与者进行分级
- **上下文**
  - 上下文是通信子所具有的一个特性，它允许对通信空间进行划分。一个上下文所发送的消息不能被另一个上下文所接收。进一步说，允许集合操作独立于挂起的点对点操作。上下文不是显式的MPI对象；它们仅作为通信子实现的一部分而出现
- **组内通信子**
  - 组内通信子将组的概念和上下文的概念结合到一起。为支持指定实现的优化及应用拓扑。MPI通信操作引用通信子来决定点对点和集合操作进行操作的范围和空间

组构造子

- **int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)**
- 输入：comm，通信子（句柄）
- 输出：group， 对应comm（句柄）的组

```
#include "mpi.h"

MPI_Comm  comm_world;
MPI_Group group_world;

comm_world = MPI_COMM_WORLD;
   MPI_Comm_group(comm_world, &group_world);
```

创建一个组（1）

- **int MPI_Group_incl(MPI_Group old_group, int count, int *members, MPI_Group *new_group)**

- 新组包括old_group中的count个进程，这count个进程用序列号members[0] ... members[count-1]表示；在new_group中具有序列号 i 的是old_group中具有序列号members[i]的进程。 members中count个元素中的每一个必须是old_group中的有效元素而且所有的元素都必须是不同的，否则程序就是错误的.如果count=0, 则new_group是MPI_GROUP_EMPTY。

```
#include "mpi.h"
MPI_Group group_world, odd_group, even_group;
int i, p, Neven, Nodd, members[8], ierr;

MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_group(MPI_COMM_WORLD, &group_world);
Neven = (p+1)/2;           /* processes of MPI_COMM_WORLD are divided */
Nodd = p - Neven;          /* into odd- and even-numbered groups */
for (i=0; i<Neven; i++) {  /* "members" determines members of even_group */
   members[i] = 2*i;
};
MPI_Group_incl(group_world, Neven, members, &even_group);
```

创建一个组（2）

- **int MPI_Group_excl(MPI_Group group, int count, int *nonmembers,MPI_Group *new_group)**
- 通过从old_group中删除具有序列号nonmembers[0],…, nonmembers[n-1]的进程创建了一组进程new_group。new_group中的进程次序与old_group中的次序相同。 nonmembers中count个元素中的每一个必须是old_group中的有效序列号且所有的元素都必须是不同的;否则,程序就是错误的. 如果count=0，new_group与old_group相同

```
#include "mpi.h"
MPI_Group group_world, odd_group, even_group;
int i, p, Neven, Nodd, nonmembers[8], ierr;

MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_group(MPI_COMM_WORLD, &group_world);
Neven = (p+1)/2;              /* processes of MPI_COMM_WORLD are divided */
Nodd = p - Neven;             /* into odd- and even-numbered groups */
for (i=0; i<Neven; i++) {     /* "nonmembers" are even-numbered procs */
nonmembers[i] = 2*i;
};
MPI_Group_excl(group_world, Neven, nonmembers, &odd_group);
```

组访问子

- **int MPI_Group_rank(MPI_Group group, int *rank)**
- **int MPI_Group_size(MPI_Group group, int *size)**

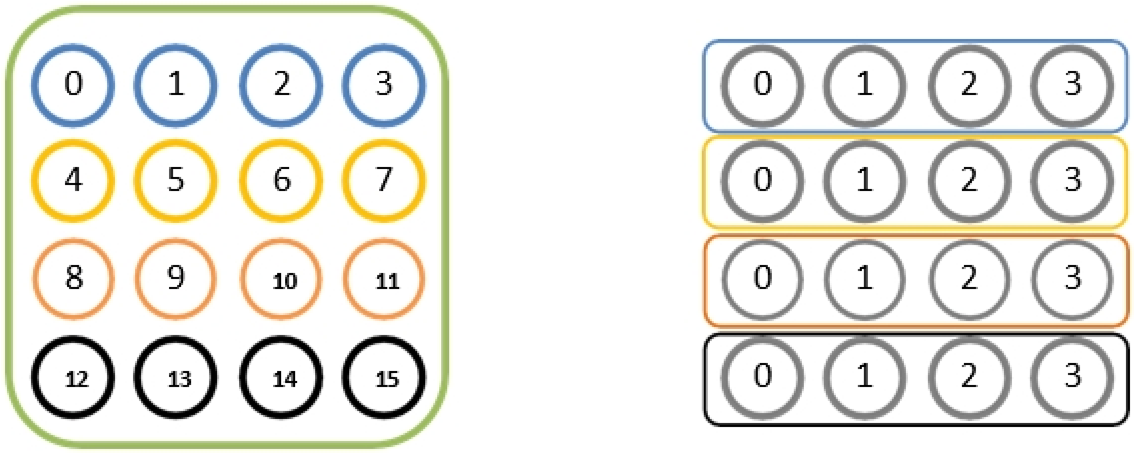
```
#include "mpi.h"
void main(int argc, char *argv[])
{
    int lam, p;
    int Neven, Nodd, members[6], even_rank, odd_rank;
    MPI_Group group_world, even_group, odd_group;
    /* Starts MPI processes ... */
    MPI_Init(&argc, &argv);           /* starts MPI */
    MPI_Comm_rank(MPI_COMM_WORLD, &lam);  /* get current process id */
    MPI_Comm_size(MPI_COMM_WORLD, &p);          /* get number of processes */

    Neven  =  (p  +  1)/2;       /* All processes of MPI_COMM_WORLD are divided */
    Nodd  =  p  -  Neven;        /* into 2 groups, odd- and even-numbered groups */
    members[0] = 0;
    members[1] = 2;
    members[2] = 4;
    MPI_Comm_group(MPI_COMM_WORLD,&group_world);
    MPI_Group_incl(group_world, Neven, members, &even_group);
    MPI_Group_excl(group_world, Neven, members, &odd_group);
```

组访问子

```
MPI_Barrier(MPI_COMM_WORLD);
if(lam == 0) {
printf("MPI_Group_incl/excl Usage Example\n");  printf("\n");
printf("Number of processes is %d\n", p);
printf("Number of odd processes is %d\n", Nodd);
printf("Number of even processes is %d\n", Neven);  printf("\n");
printf(" lam            even          odd\n");
}
MPI_Barrier(MPI_COMM_WORLD);

MPI_Group_rank(even_group, &even_rank);
 MPI_Group_rank(odd_group, &odd_rank);
printf("%8d %8d %8d\n",lam, even_rank, odd_rank);

MPI_Finalize();       /* let MPI finish up ... */
}
```

Exercise: GroupSplit

- Explains the use of new communicators and groups
- No programming needed follow the guidelines in the README



创建通信子

- **int MPI_Comm_create(MPI_Comm,MPI_Group group,MPI_Comm *newcomm)**
- 由group所定义的通信组及一个新的上下文创建了一个新的通信子newcomm. comm中的缓冲信息不传播给newcomm. 对于不在group中的进程，本函数返回MPI_COMM_NULL. 如果所有的group参数不都具有同样的值, 或者如果group不是与comm相联的组的子集, 则调用将出错, 注意comm中的所有进程都将执行这一调用,即使他们不属于这一新组. 本调用仅适用于组内通信子。

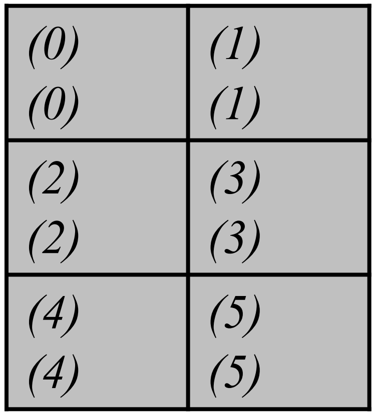
```
#include "mpi.h"
MPI_Comm comm_world, comm_worker;
MPI_Group group_world, group_worker;
int nonmembers=0;

comm_world = MPI_COMM_WORLD;
MPI_Comm_group(comm_world, &group_world);
MPI_Group_excl(group_world, 1, &nonmembers, &group_worker);
                                             /* process 0 not member */

MPI_Comm_create(comm_world, group_worker, &comm_worker);
```

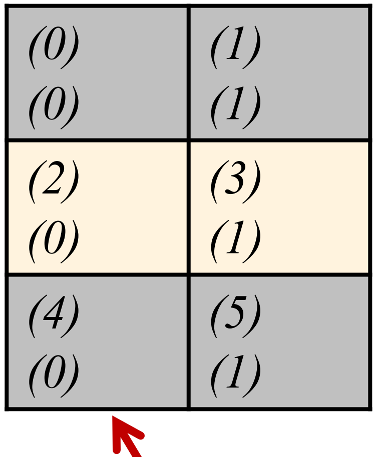
划分通信子

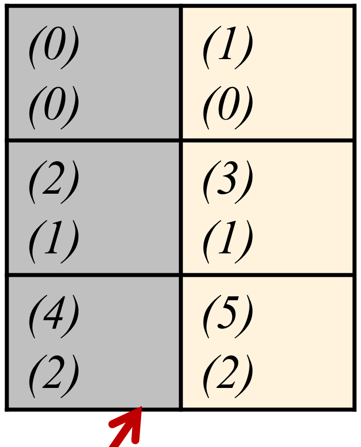
- **int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)**
- 将与comm相关的组划分为若干不相连的子组,每一个子组对应color一个值. 每一个子组包含具有同样color的所有进程. 在每一个子组内, 进程按照参数 key所定义的值的次序进行排列, 并根据它们在旧组中的序列号携带打破后的联系, 为每个子组创建一个新的通信子并在newcomm中返回. 一个进程可能提供color值MPI_UNDEFINED, 在这种情况下,newcomm返回MPI_COMM_NULL. 这是一个集合调用, 但是允许每个进程为color和key提供不同的值.
- 这是一个能将单通信进程组划分成k个子组的功能很强大的机制, 其中k由用户隐含选定(由所有进程上所声明的colors的数目所选定）。每一个结果通信子都是非重迭的。这样一个划分对于定义一个诸如多网格或线性代数等计算的层次是很有用的。

划分通信子

Iam	0	1	2	3	4	5
irow	0	0	1	1	2	2
jcol	0	1	0	1	0	1







```
/* logical 2D topology with nrow rows and mcol columns */
    irow = Iam/mcol;        /* logical row number */
jcol = mod(Iam, mcol); /* logical column number */
    comm2D = MPI_COMM_WORLD;

MPI_Comm_split(comm2D, irow, jcol, row_comm);
    MPI_Comm_split(comm2D, jcol, irow, col_comm);
```

程序实例（1/3）

```
#include "mpi.h"
void main(int argc, char *argv[])
{
int mcol, irow, jcol, p;
MPI_Comm row_comm, col_comm, comm2D;  int
Iam, row_id, col_id;
int row_group, row_key, map[6];

/* Starts MPI processes ... */
    MPI_Init(&argc, &argv);                /* starts MPI */

    MPI_Comm_rank(MPI_COMM_WORLD, &Iam); /* get current process id */
    MPI_Comm_size(MPI_COMM_WORLD, &p);     /* get number of processes */

    map[0]=2; map[1]=0; map[2]=3; map[3]=1; map[4]=0; map[5]=1;  mcol=2; /* nrow = 3 */
    if(Iam == 0) {  printf("\n");
    printf("Example of MPI_Comm_split Usage\n");
    printf("Split 3x2 grid into 2 different communicators\n");
    printf("which correspond to 3 rows and 2 columns.");  printf("\n");
    printf("Iam          irow          jcol          row-id          col-id\n");
    }
```

程序实例（2/3）

```
/* virtual topology with nrow rows and mcol columns */
    irow = Iam/mcol;   /* row number */
    jcol = Iam%mcol;   /* column number */
    comm2D = MPI_COMM_WORLD;
    MPI_Comm_split(comm2D, irow, jcol, &row_comm);
    MPI_Comm_split(comm2D, jcol, irow, &col_comm);

    MPI_Comm_rank(row_comm, &row_id);
    MPI_Comm_rank(col_comm, &col_id);
    MPI_Barrier(MPI_COMM_WORLD);
    printf("%8d %8d %8d %8d %8d\n",Iam,irow,jcol,row_id,col_id);
    MPI_Barrier(MPI_COMM_WORLD);

    if(Iam == 0) {  printf("\n");
    printf("Next, create more general communicator\n");
     printf("which consists of two groups :\n");
    printf("Rows 1 and 2 belongs to group 1 and row 3 is group 2\n");
    printf("\n");
    }
    /* MPI_Comm_split is more general than MPI_Cart_sub  simple example of 6 processes divided
    into 2 groups; 1st 4 belongs to group 1 and remaining two to group 2 */
    row_group = Iam/4;                         /* this expression by no means general */
    row_key = Iam - row_group*4;               /* group1:0,1,2,3; group2:0,1 */
    MPI_Comm_split(comm2D, row_group, row_key, &row_comm);
    MPI_Comm_rank(row_comm, &row_id);
    printf("%8d %8d\n",Iam,row_id);
    MPI_Barrier(MPI_COMM_WORLD);
```

程序实例（3/3）

```
if(Iam == 0) {
printf("\n");
printf("If two processes have same key, the ranks\n");
printf("of these two processes in the new\n");
printf("communicator will be ordered according'\n");
printf("to their order in the old communicator\n");
printf(" key = map[Iam]; map = (2,0,3,1,0,1)\n");  printf("\n");
}
/* MPI_Comm_split is more general than MPI_Cart_sub  simple example of 6 processes
dirowided into 2 groups;  1st 4 belongs to group 1 and remaining two to group 2 */
row_group = Iam/4; /* this expression by no means general */
row_key = map[Iam];
MPI_Comm_split(comm2D, row_group, row_key, &row_comm);
MPI_Comm_rank(row_comm, &row_id);
MPI_Barrier(MPI_COMM_WORLD);
printf("%8d %8d\n",Iam,row_id);

MPI_Finalize();          /* let MPI finish up ... */
}
```

程序运行结果

Example of MPI_Comm_split Usage Split 3x2 grid into 2 different communicators, which correspond to 3 rows and 2 columns.

Iam	irow	jcol	group	rank
0	0	0	0	0
2	1	0	0	1
3	1	1	1	1
5	2	1	1	2
1	0	1	1	0
4	2	0	0	2

Next, create more general communicator which consists of two groups :
Rows 1 and 2 belongs to group 1 and row 3 is group 2

Iam	new rank
0	0
3	3
2	2
1	1
5	1
4	0

If two processes have same key, the ranks of these two processes in the new communicator will be ordered according to their order in the old communicator key = map(Iam); map = (2,0,3,1,0,1)

Iam	new rank
1	0
0	2
4	0
5	1
3	1
2	3

笛卡尔拓扑

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

- 在许多并行应用程序中，进程的线性排列不能充分地反映进程间在逻辑上的通信模型（通常由基本问题几何和所用的数值算法所决定），进程经常被排列成二维或三维网格形式的拓扑模型，而且，通常用一个图来描述逻辑进程排列，我们指这种逻辑进程排列为“虚拟拓扑”。

笛卡尔构造子

- **int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)**

■ IN comm_old 输入通信子（句柄）

■ IN ndims 笛卡尔网格的维数（整数）

■ IN dims 大小为ndims的整数矩阵，说明了每一维的进程数

■ IN periods 大小为ndims的逻辑矩阵，说明了在每一维上网格是否是周期性的(true)或非周期性的(false)

■ IN reorder 标识数可以重排序(true)或不能(false)（logical）

■ OUT comm_cart 带有新的笛卡尔拓扑的通信子（handle）

程序实例

```
#include "mpi.h"
  MPI_Comm old_comm, new_comm;
  int ndims, reorder, periods[2], dim_size[2];

  old_comm = MPI_COMM_WORLD;
  ndims = 2;          /*  2D matrx/grid */
  dim_size[0] = 3;    /* rows */
  dim_size[1] = 2;    /* columns */
  periods[0] = 1;     /* row periodic (each column forms a ring) */
  periods[1] = 0;     /* columns nonperiodic */
  reorder = 1;        /* allows processes reordered for efficiency */

  MPI_Cart_create(old_comm, ndims, dim_size, periods, reorder, &new_comm);
```

0,0 (0)	0,1 (1)
1,0 (2)	1,1 (3)
2,0 (4)	2,1 (5)

	-1, 0 (4)	-1, 1 (5)	
0, -1 (-1)	0, 0 (0)	0, 1 (1)	0, 2 (-1)
1, -1 (-1)	1, 0 (2)	1, 1 (3)	1, 2 (-1)
2, -1 (-1)	2, 0 (4)	2, 1 (5)	2, 2 (-1)
	3, 0 (0)	3, 1 (1)	

periods(0)=.true.;periods(1)=.false.

	-1, 0 (-1)	-1, 1 (-1)	
0, -1 (1)	0, 0 (0)	0, 1 (1)	0, 2 (0)
1, -1 (3)	1, 0 (2)	1, 1 (3)	1, 2 (2)
2, -1 (5)	2, 0 (4)	2, 1 (5)	2, 2 (4)
	3, 0 (-1)	3, 1 (-1)	

periods(0)=.false.;periods(1)=.true.

辅助函数

- **int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)**
- **int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)**
- comm 带有笛卡尔结构的通信子（句柄）
- coords 说明一个进程在笛卡尔坐标系下的坐标
- rank 被说明进程的标识数（整数）
- maxdims 维度

笛卡尔轮换定位

- **int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)**
  - comm 带有笛卡尔结构的通信子（句柄）
  - direction 轮换的坐标维（整数）
  - disp 偏移（＞0：向上轮换，＜0：向下偏移）（整数）
  - rank_source 源进程的标识数（整数）
  - rank_dest 目标进程的标识数（整数）
- 如果进程拓扑是笛卡尔结构，MPI_SENDRECV操作用于在坐标方向上执行数据轮换。作为输入，MPI_SENDRECV将源进程的标识数作为接受，目标进程的标识数作为发送。如果一个笛卡尔进程组调用函数MPI_CART_SHIFT，那么此函数将上面的标识符提供给调用进程，然后将其传送给MPI_SENDRECV。用户说明坐标方向和步长（正数或负数）。

程序实例

```
 /* create Cartesian topology for processes */
    dims[0]  =  nrow;    /* number of rows         */
    dims[1]  =  mcol;    /* number of columns  */
    period[0] = 1;       /* cyclic in this direction */
    period[1] = 0;       /* no cyclic in this direction */
    MPI_Cart_create(MPI_COMM_WORLD, ndim, dims,
period, reorder,  &comm2D);
    MPI_Comm_rank(comm2D, &me) ;
    MPI_Cart_coords(comm2D, me, ndim, coords);

    index =  0;            /* shift along the 1st index (out of 2) */
    displ =  1;            /* shift by  1 */
    MPI_Cart_shift(comm2D, index, displ, &source, &dest1);
```

0,0 (0)	*0,1 (1)*
1,0 (2)	*1,1 (3)*
2,0 (4)	*2,1 (5)*

Rank 2: source 0, destination 4
Rank 1: source 5, destination 3

例子

- 基于16个进程创建4x4的笛卡尔拓扑，每个进程同其邻接4个邻居交换其rank号

程序（1/2）

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 16
#define UP 0
#define DOWN 1
#define LEFT 2
#define RIGHT 3

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

int numtasks, rank, source, dest, outbuf, i, tag=1;
int inbuf[4]={MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL};
int nbrs[4], dims[2]={4,4}, periods[2]={0,0}, reorder=0, coords[2];

MPI_Request reqs[8];
MPI_Status stats[8];
MPI_Comm cartcomm;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
```

程序（2/2）

```
if (numtasks == SIZE) {
 MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &cartcomm);
 MPI_Comm_rank(cartcomm, &rank);
 MPI_Cart_coords(cartcomm, rank, 2, coords);
 MPI_Cart_shift(cartcomm, 0, 1, &nbrs[UP], &nbrs[DOWN]);
 MPI_Cart_shift(cartcomm, 1, 1, &nbrs[LEFT], &nbrs[RIGHT]);

 printf("rank= %d coords= %d %d neighbors(u,d,l,r)= %d %d %d %d\n",
            rank,coords[0],coords[1],nbrs[UP],nbrs[DOWN],nbrs[LEFT],  nbrs[RIGHT]);

 outbuf = rank;
 for (i=0; i<4; i++) {
  dest = nbrs[i];
  Source = nbrs[i];
  MPI_Isend(&outbuf, 1, MPI_INT, dest, tag,  MPI_COMM_WORLD, &reqs[i]);
  MPI_Irecv(&inbuf[i], 1, MPI_INT, source, tag,  MPI_COMM_WORLD, &reqs[i+4]);
 }
 MPI_Waitall(reqs, stats);

 printf("rank= %d     inbuf(u,d,l,r)= %d %d %d %d\n",
            rank,inbuf[UP],inbuf[DOWN],inbuf[LEFT],inbuf[RIGHT]);
}
else
 printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}
```

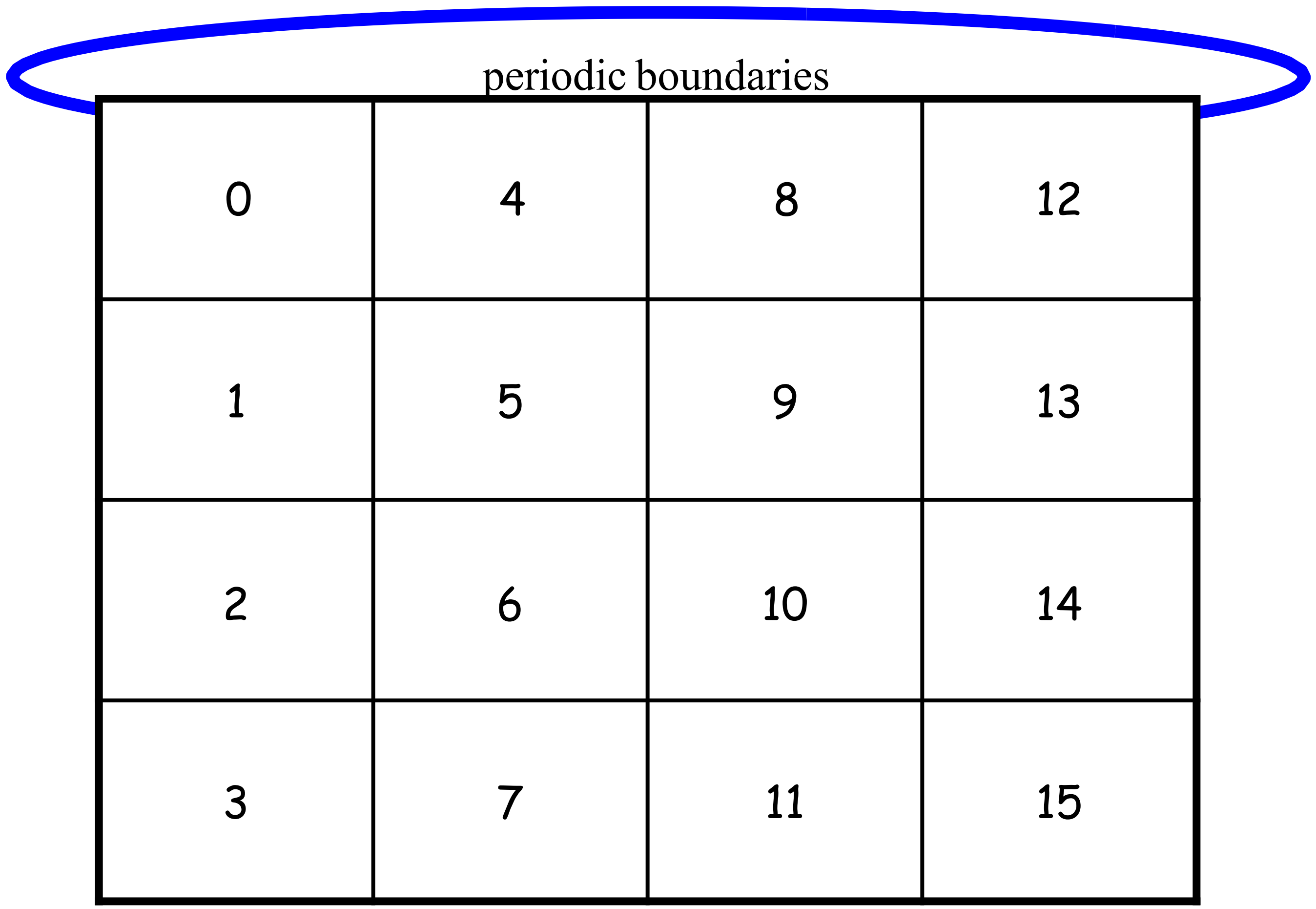
程序运行结果

```
rank=   0 coords=   0 0 neighbors(u,d,l,r)=  -1 4 -1 1
rank=   0                inbuf(u,d,l,r)= -1 4 -1 1
rank=   8 coords=   2 0 neighbors(u,d,l,r)=    4 12 -1 9
rank=   8                inbuf(u,d,l,r)=   4 12 -1 9
rank=   1 coords=   0 1 neighbors(u,d,l,r)=  -1 5  0 2
rank=   1                inbuf(u,d,l,r)= -1 5 0 2
rank=  13 coords=   3 1 neighbors(u,d,l,r)=     9 -1 12 14
rank=  13                inbuf(u,d,l,r)=  9 -1 12 14
…
…
rank=   3 coords=   0 3 neighbors(u,d,l,r)=  -1 7  2 -1
rank=   3                inbuf(u,d,l,r)= -1 7 2 -1
rank=  11 coords=   2 3 neighbors(u,d,l,r)=     7 15 10 -1
rank=  11                inbuf(u,d,l,r)=  7 15 10 -1
rank=  10 coords=    2 2 neighbors(u,d,l,r)=     6 14 9 11
rank=  10                inbuf(u,d,l,r)=  6 14 9 11
rank=   9 coords=   2 1 neighbors(u,d,l,r)=     5 13 8 10
rank=   9                inbuf(u,d,l,r)=   5 13 8 10
```

Exercise: MPI_Cart

- Do I have to make a map of MPI processes myself?
- You can use MPI_Cart to create a domain decomposition for you.
- Exercise sets up 2D domain with one
  - periodic and
  - non-periodic boundary condition

Exercise: MPI_Cart



THANKS