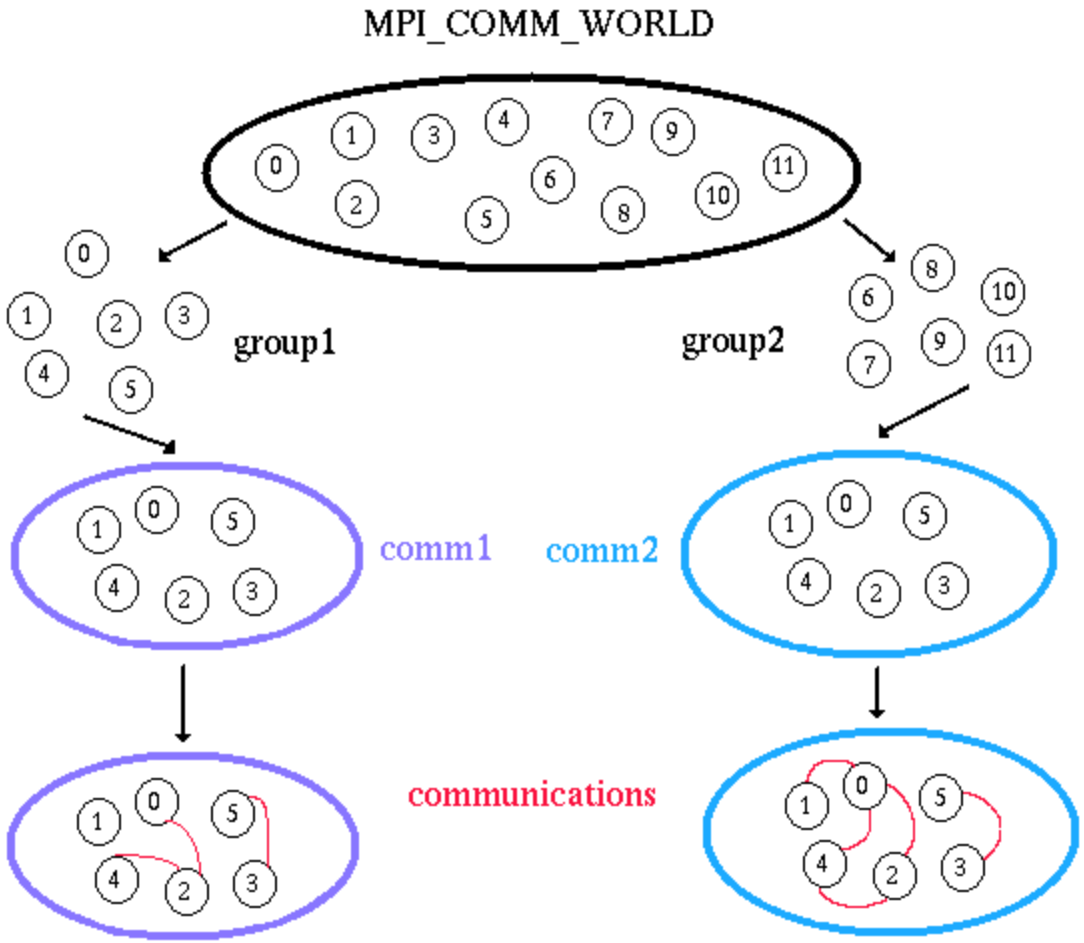
MPI Groups, Communicators and Topologies

Groups and communicators

- In our case studies, we saw examples where collective communication needed to be performed by subsets of the processes in the computation
  - (eg) 2-D version of MVM
- MPI provides routines for
  - defining new process *groups* from subsets of existing process groups like MPI_COMM_WORLD
  - creating a new *communicator* for a new process group
  - performing collective communication within that process group
  - organizing process groups into *virtual topologies*:
    - most common use: permits processes within a group to be given multidimensional numbers rather than the standard 0...n-1 numbering

Groups and communicators



Facts about groups and communicators

- Group:
  - ordered set of processes
  - each process in group has a unique integer id called its rank within that group
  - process can belong to more than one group
    - rank is always relative to a group
  - groups are “opaque objects”
    - use only MPI provided routines for manipulating groups
- Communicators:
  - all communication must specify a communicator
  - from the programming viewpoint, groups and communicators are equivalent
  - why distinguish between groups and communicators?
    - sometimes you want to create two communicators for the same group
    - useful for supporting libraries
  - communicators are also “opaque objects”
- Groups and communicators are dynamic objects and can be created and destroyed during the execution of the program

Typical usage

1. Extract handle of global group from MPI_COMM_WORLD using MPI_Comm_group
2. Form new group as a subset of global group using MPI_Group_incl or MPI_Group_excl
3. Create new communicator for new group using MPI_Comm_create
4. Determine new rank in new communicator using MPI_Comm_rank
5. Conduct communications using any MPI message passing routine
6. When finished, free up new communicator and group (optional) using MPI_Comm_free and MPI_Group_free

Details

- MPI_Group_excl:
  - create a new group by removing certain processes from existing group
  - int MPI_Group_excl(MPI_Group group,int n,int *ranks, MPI_Group *newgroup);
  - ranks: array of size n containing ranks to be excluded
- MPI_Group_incl:
  - create a new group from certain selected processes from existing group
  - int MPI_Group_incl(MPI_Group group,int n,int *ranks, MPI_Group *newgroup);
  - ranks: array of size n containing ranks to be included

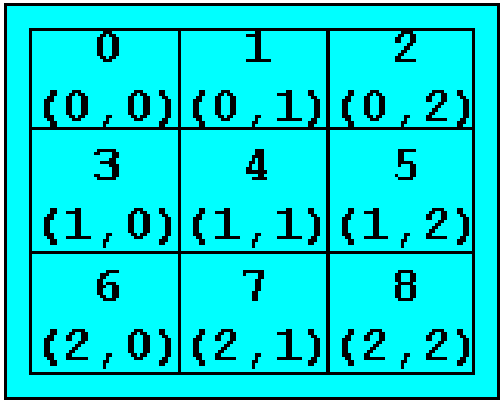
```
main(int argc, char **argv) {
     int me, count, count2;
     void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
    MPI_Group MPI_GROUP_WORLD, grprem;
    MPI_Comm commslave;
    static int ranks[] = {0};
    MPI_Init(&argc, &argv);
    MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    MPI_Group_excl(MPI_GROUP_WORLD, 1, ranks, &grprem);
    MPI_Comm_create(MPI_COMM_WORLD, grprem, &commslave);
    if(me != 0){ /* compute on slave */
        MPI_Reduce(send_buf,recv_buff,count, MPI_INT, MPI_SUM, 1,
                                    commslave);
                }
     /* zero falls through immediately to this reduce, others do later... */
     MPI_Reduce(send_buf2, recv_buff2, count2, MPI_INT, MPI_SUM, 0,
                            MPI_COMM_WORLD);
     MPI_Comm_free(&commslave);
     MPI_Group_free(&MPI_GROUP_WORLD);
     MPI_Group_free(&grprem);
     MPI_Finalize();
}
```

```
#include “mpi.h”
#include <stdio.h>
#define NPROCS 8
int main(argc,argv)
int argc;
char *argv[];  {
int       rank, new_rank, sendbuf, recvbuf, numtasks, ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};
MPI_Group  orig_group, new_group;
MPI_Comm    new_comm;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
sendbuf = rank;

/* Extract the original group handle */
MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

/* Divide tasks into two distinct groups based upon rank */
if (rank < NPROCS/2) {
  MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
  }
else {
  MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);
  }

/* Create new new communicator and then perform collective communications */
MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);

MPI_Group_rank (new_group, &new_rank);
printf("rank= %d newrank= %d recvbuf= %d\n",rank,new_rank,recvbuf);

MPI_Finalize();
}
```

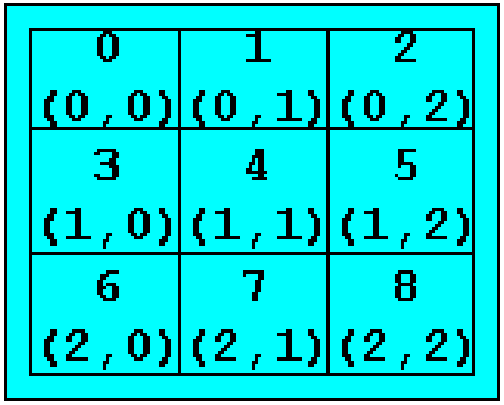
Sample output:

```
rank= 7 newrank= 3 recvbuf= 22
rank= 0 newrank= 0 recvbuf= 6
rank= 1 newrank= 1 recvbuf= 6
rank= 2 newrank= 2 recvbuf= 6
rank= 6 newrank= 2 recvbuf= 22
rank= 3 newrank= 3 recvbuf= 6
rank= 4 newrank= 0 recvbuf= 22
rank= 5 newrank= 1 recvbuf= 22
```

Virtual topologies



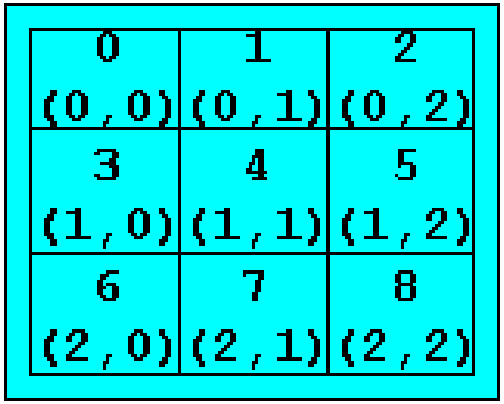
- Current view of group/communicator: one dimensional numbering
- In many algorithms, it is convenient to view the processes as being arranged in a grid of some number of dimensions
- One approach: perform the translation from grid co-ordinates to 1D numbers yourself
  - Hard to write the program
  - Even harder to read the program
- MPI solution:
  - routines for letting you view processes as if they were organized in grid of some dimension
  - MPI also permits you to organize processes logically into general graphs but we will not worry about this
- Key routines:
  - MPI_Cart_Create: create a grid from an existing 1D communicator
    - int MPI_Cart_create (MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)
    - periods: array of booleans to specify wrap-around in each dimension
    - reorder: can ranks be reordered or not? Ignored by most MPI implementations
  - MPI_Cart_get: my grid coordinates
  - MPI_Cart_rank: grid coordinates 1D rank
  - MPI_Cart_shift: offset in some grid dimension 1D rank of sender and receiver




```
#include <mpi.h>
#include <stdio.h>
#define NP 3
main(int argc, char **argv){
/* Grid setup for 3x3 Grid */
……..
/* MPI Cartesian Grid information */
int ivdim[2] = {NP,NP}, ivper[2]={1,1};
int ivdimx[2], ivperx[2], mygrid[2];
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &mype);
/* Create Cartesian Grid and extract information */
MPI_Cart_create(MPI_COMM_WORLD,2,ivdim ,ivper, 0,&igcomm);
MPI_Cart_get( igcomm,2,ivdimx,ivperx, mygrid);
MPI_Cart_shift( igcomm,1,1, &isrca,&idesa); //column offset of +1
myrow=mygrid[0]; mycol=mygrid[1];
printf("A (%d) [%d,%d]: %2d ->%2d\n",mype,myrow,mycol,isrca,idesa);
MPI_Finalize();
}
```

Output

mype	[myrow, mycol]	src	dst
0	[0,0]	2	1
1	[0,1]	0	2
2	[0,2]	1	0
3	[1,0]	5	4
4	[1,1]	3	5
5	[1,2]	4	3
6	[2,0]	8	7
7	[2,1]	6	8
8	[2,2]	7	6



General splitting of communicators

- MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)

- This is a very powerful routine for splitting a communicator into some number of smaller communicators (depending on color) and assigning ranks within the new communicators
- Will be useful for 2D MVM:
  - split MPI_COMM_WORLD twice, once to define row communicators and once to define column communicators
  - use row & column indices in Cartesian topology as colors for calls to Comm_split

Example 5.7 Assume that a collective call to MPI_COMM_SPLIT is executed in a 10 element group, with the arguments listed in the table below.

rank	0	1	2	3	4	5	6	7	8	9
process	a	b	c	d	e	f	g	h	i	j
color	0	⊥	3	0	3	0	0	5	3	⊥
key	3	1	2	5	1	1	1	2	1	0

The call generates three new communication domains: the first with group {f,g,a,d}, the second with group {e,i,c}, and the third with singleton group {h}. The processes b and j do not participate in any of the newly created communication domains, and are returned a null communicator handle.

A call to MPI_COMM_CREATE(comm, group, newcomm) is equivalent to a call to MPI_COMM_SPLIT(comm, color, key, newcomm), where all members of group provide color = 0 and key = rank in group, and all processes that are not members of group provide color = MPI_UNDEFINED. The function MPI_COMM_SPLIT allows more general partitioning of a group into one or more subgroups with optional reordering.

Advice to users. This is an extremely powerful mechanism for dividing a single communicating group of processes into k subgroups, with k chosen implicitly by the user (by the number of colors asserted over all the processes). Each resulting communication domain will be unique and their associated groups are non-overlapping. Such a division could be useful for defining a hierarchy of computations, such as for multigrid, or linear algebra.

Multiple calls to MPI_COMM_SPLIT can be used to overcome the requirement that any call have no overlap of the resulting communicators (each process is of only one color per call). In this way, multiple overlapping communication structures can be created.

Note that, for a fixed color, the keys need not be unique. It is MPI_COMM_SPLIT's responsibility to sort processes in ascending order according to this key, and to

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

main(int argc, char **argv)
{
   MPI_Comm row_comm, col_comm;
   int myrank, size, P=4, Q=3, p, q;

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &myrank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);

   /* Determine row and column position */
   p = myrank / Q;
   q = myrank % Q; /* pick a row-major mapping */

   /* Split comm into row and column comms */
   MPI_Comm_split(MPI_COMM_WORLD, p, q, &row_comm);
   /* color by row, rank by column */
   MPI_Comm_split(MPI_COMM_WORLD, q, p, &col_comm);
   /* color by column, rank by row */

   printf("[%d]:My coordinates are (%d,%d)\n",myrank,p,q);
   MPI_Finalize();
}
```