Communicators and Groups

- A group is an ordered set of processes
- Each process in a group is associated with a unique integer rank
- Rank values start at zero and go to N-1, where N is the number of processes in the group
- In MPI, a group is represented within system memory as an object. It is accessible to the programmer only by a "handle"
- A group is always associated with a communicator object

- A communicator encompasses a group of processes that may communicate with each other
- All MPI messages must specify a communicator
- In the simplest sense, the communicator is an extra "tag" that must be included with MPI calls
- Like groups, communicators are represented within system memory as objects and are accessible to the programmer only by "handles"
- For example, the handle for the communicator that comprises all tasks is MPI_COMM_WORLD

Purpose of Group and Communicators

- Allow you to organize tasks, based upon function, into task groups
- Enable Collective Communications operations across a subset of related tasks
- Provide basis for implementing user defined virtual topologies
- Provide for safe communications

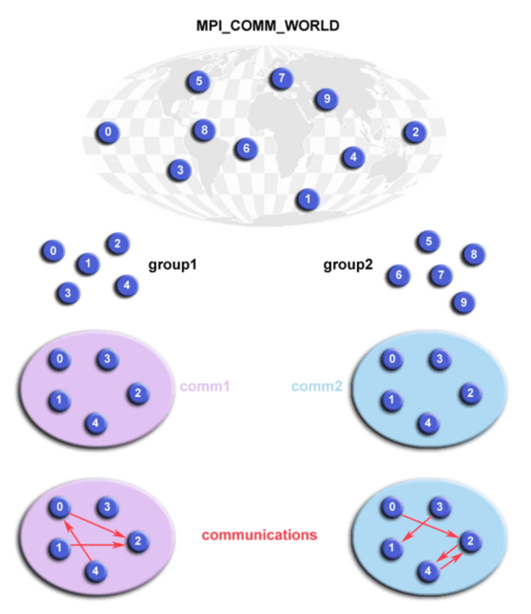
Programming Considerations and Restrictions

- Groups/communicators are dynamic - they can be created and destroyed during program execution
- Processes may be in more than one group/communicator. They will have a unique rank within each group/communicator
- MPI provides over 40 routines related to groups, communicators, and virtual topologies

Usage of Groups and Communicators

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

Groups and Communicators



Groups and Communicators

```
#include "mpi.h"
#include <stdio.h>
#define NPROCS 8

main(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;   // required variables
MPI_Comm    new_comm;   // required variable

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

if (numtasks != NPROCS) {
  printf("Must specify MP_PROCS= %d. Terminating.\n",NPROCS);
  MPI_Finalize();
  exit(0);
  }

sendbuf = rank;

```

Groups and Communicators

```

   // 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 communicati
   MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
   MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);

   // get rank in new group
   MPI_Group_rank (new_group, &new_rank);
   printf("rank= %d newrank= %d recvbuf= %d\n",rank,new_rank,recvbuf);

   MPI_Finalize();
   }
```

Groups and Communicators

Sample program 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 Topology

What Are They?

- In terms of MPI, a virtual topology describes a mapping/ordering of MPI processes into a geometric "shape"
- The two main types of topologies supported by MPI are Cartesian (grid) and Graph
- MPI topologies are virtual - there may be no relation between the physical structure of the parallel machine and the process topology
- Virtual topologies are built upon MPI communicators and groups
- Must be "programmed" by the application developer

Virtual Topology

Why Use Them?

- Convenience
  - Virtual topologies may be useful for applications with specific communication patterns - patterns that match an MPI topology structure
  - For example, a Cartesian topology might prove convenient for an application that requires 4-way nearest neighbor communications for grid based data

Virtual Topology

Why use them?

- Communication Efficiency
  - Some hardware architectures may impose penalties for communications between successively distant "nodes"
  - A particular implementation may optimize process mapping based upon the physical characteristics of a given parallel machine
  - The mapping of processes into an MPI virtual topology is dependent upon the MPI implementation, and may be totally ignored

Virtual Topology: Example

A simplified mapping of processes into a Cartesian virtual topology appears below:

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)

Virtual Topology: Example Code

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

main(int argc, char *argv[])  {
int numtasks, rank, source, dest, outbuf, i, tag=1,
   inbuf[4]={MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,MPI_PROC_NULL,},
   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;   // required variable

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

if (numtasks == SIZE) {
   // create cartesian virtual topology, get rank, coordinates, neighbor rank
   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]);

```

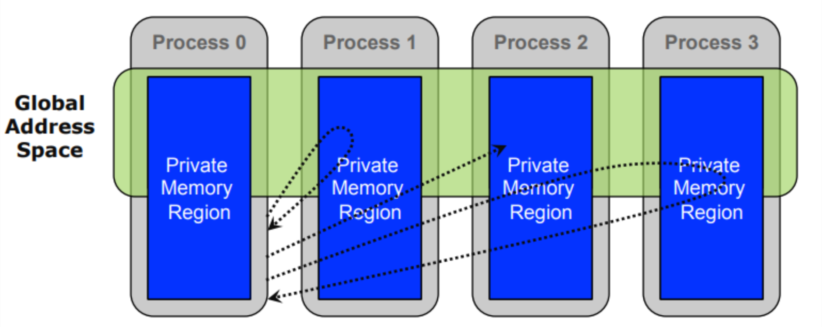
Virtual Topology: Example Code

```
      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;

      // exchange data (rank) with 4 neighbors
      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(8, 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();
   }
```

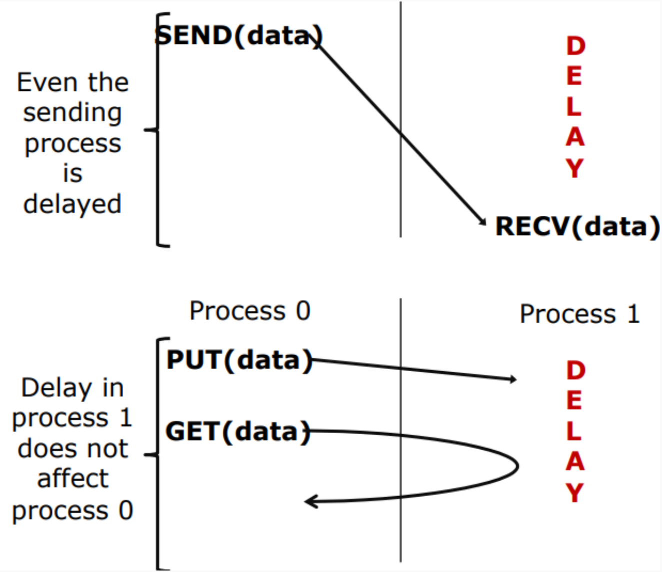
One Sided Communication

- The basic idea of one-sided communication models is to decouple data movement with process synchronization
  - Should be able to move data without requiring that the remote process synchronize
  - Each process exposes a part of its memory to other processes
  - Other processes can directly read from or write to this memory

One Sided Communication



One Sided Communication



Advantages of RMA Operations

- Can do multiple data transfers with a single synchronization operation
- Bypass tag matching
- Some irregular communication patterns can be more economically expressed
- Can be significantly faster than send/receive on systems with hardware support for remote memory access

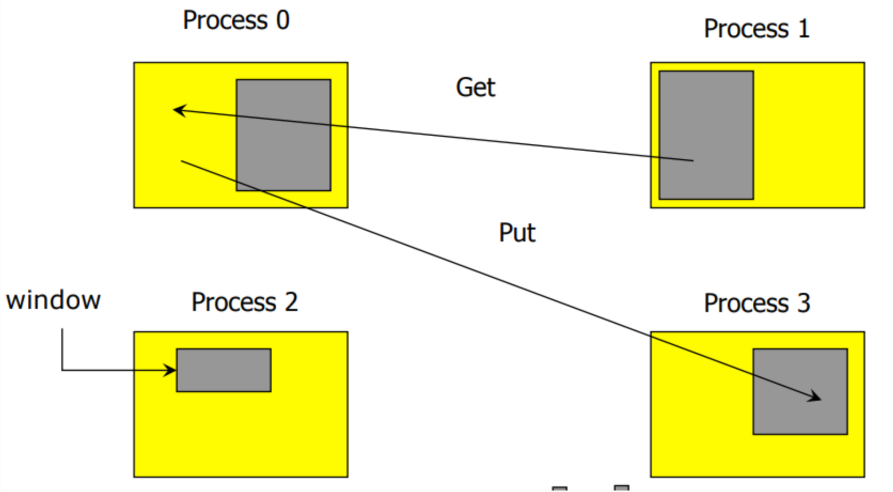
What we need to know in MPI RMA

- How to create remote accessible memory?
- Reading, Writing and Updating remote memory
- Data Synchronization
- Memory Model

Creating Public Memory

- Any memory created by a process is, by default, only locally accessible
- Once the memory is created, the user has to make an explicit MPI call to declare a memory region as remotely accessible
  - MPI terminology for remotely accessible memory is a window
- Once a memory region is declared as remotely accessible, all processes in the window object can read/write data to this memory without explicitly synchronizing with the target process

One Sided Communication



Creating Windows

Use MPI_WIN_CREATE to create windows

- Expose a region of the memory in an RMA window
- Only data exposed in a window can be accessed with RMA ops

int MPI_Win_create(void *base, MPI_int size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)

- base: local data
- size: size of local data
- disp_unit: unit size for displacements
- info: info argument
- win: window object

One Sided Communication

```
int main(int argc, char ** argv)
{
    int *a;    MPI_Win win;

    MPI_Init(&argc, &argv);

    /* create private memory */
    MPI_Alloc_mem(1000*sizeof(int), MPI_INFO_NULL, &a);
    /* use private memory like you normally would */
    a[0] = 1;  a[1] = 2;

    /* collectively declare memory as remotely accessible */
    MPI_Win_create(a, 1000*sizeof(int), sizeof(int),
          MPI_INFO_NULL, MPI_COMM_WORLD, &win);

    /* Array 'a' is now accessibly by all processes in
     * MPI_COMM_WORLD */

    MPI_Win_free(&win);
    MPI_Free_mem(a);
    MPI_Finalize(); return 0;
}
```

Window Allocate

Use MPI_WIN_ALLOCATE

- Create a remotely accessible memory region in an RMA window

int MPI_Win_allocate(MPI_Aint size, int disp_unit,MPI_Info info,MPI_Comm comm, void *baseptr, MPI_Win *win)

Window allocate example

```
int main(int argc, char ** argv)
{
    int *a;    MPI_Win win;

    MPI_Init(&argc, &argv);

    /* collectively create remote accessible memory in a window */
    MPI_Win_allocate(1000*sizeof(int), sizeof(int), MPI_INFO_NULL,
                     MPI_COMM_WORLD, &a, &win);

    /* Array 'a' is now accessible from all processes in
     * MPI_COMM_WORLD */

    MPI_Win_free(&win);

    MPI_Finalize(); return 0;
}
```

Data Movement between windows

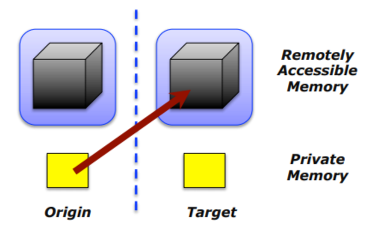
MPI provides ability to read, write and atomically modify data in remotely accessible memory regions

- MPI_GET
- MPI_PUT

MPI_PUT: Put data in a window

MPI_Put(void *origin_addr, int origin_count, MPI_Datatype origin_dtype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_dtype, MPI_Win win)

- Move data from origin to target
- Separate data description triples for origin and target



MPI_GET: Get data from a window

MPI_Get(void *origin_addr, int origin_count, MPI_Datatype origin_dtype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_dtype, MPI_Win win)

- Move data to origin, from target