Barrier

- As in all message-passing systems, MPI provides a means of synchronizing processes by stopping each one until they have reached a specific “barrier” call.

Non Blocking Routines

- Non-Blocking send – MPI_Isend(), will return “immediately” even before source location is safe to be altered
- Non-Blocking receive – MPI_Irecv(), will return even if there is no message to accept

Formats

MPI_Isend(buf, count, datatype, dest, tag, comm, request)
MPI_Irecv(buf, count, datatype, source, tag, comm, request)

Completion detected by MPI_Wait() and MPI_Test()

MPI_Wait() waits until the operation has actually completed and will return then,
MPI_Test() returns with a flag set indicating whether operation Completed at that time.

These routines need to know whether the particular operation has Completed which is determined by accessing the request parameter.

Example

To send an integer x for process 0 to process 1 and allow process 0 to Continue

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
If (myrank == 0) {
        int x;
        MPI Isend(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD,
         req1);
        compute();
        MPI_Wait(req1, status);
}
Else if (myrank == 1) {
        int x;
        MPI_Recv(&x, 0, MPI_INT, 1, msgtag, MPI_COMM_WORLD,
       status);
}
```

Example

- int count *buf, source;
- MPI_PROBE(MPI_ANY_SOURCE,0,comm, &status);
- source= status.MPI_Source;
- MPI_GET_COUNT(status, MPI_INT, &count);
- Buf= malloc(count*sizeof(int));
- MPI_RECV(buf,count,MPI_INT,source,0,comm,&status);

- Status.MPI_Source has the source of the message just received.

Modular Programming

- MPI_COMM_DUP(comm,newcomm): A program may create a new context to ensure that communications performed for different purposes are not confused. This mechanism supports sequential composition
  - Newcomm is created for same set of processes but with different context

MPI_COMM_DUP

- MPI_COMM_DUP(comm,newcomm)
  - Create new communicator; same group, new context
- If there are two programs running and need them to have the same processes but performing two different send and receives:

MPI_COMM_DUP

- Same processes perform transposition—transpose A

integer comm,newcomm

MPI_COMM_DUP(comm,newcomm)

Transpose(newcomm,A)

MPI_COMM_FREE(newcomm)

Transpose routine will be defined to use the communicator newcomm in all communication operations. This ensures that communication performed within this routine cannot be confused with communications performed outside

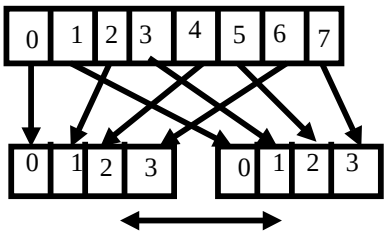
Communicating between processes

- Processes in each group must supply a local intracommunicator that identifies processes involved in that group
- Leader process in each group
- Parent communicator that contains all processes in both groups

Communicating between processes

- MPI_INTERCOMM_CREATE(comm,local_leader,peercomm,remote_leader,tag,intercomm)
  - Comm: local intracommunicator
  - Leader : local leader
  - Peer : parent communicator
  - Remote_leader: other groups leader process
  - Tag: Two groups have a tag that can be used to communicate
  - Intercomm: new intercommunicator

Example




```
MPI_COMM_SIZE(MPI_COMM_WORLD,count)
MPI_COMM_RANK(MPI_COMM_WORLD,myid)
/* Split processes into two groups; odd and even
Numbered*/
MPI_COMM_SPLIT(MPI_COMM_WORLD,
                        myid%2,myid,comm)
/* Determine process id in new group */
MPI_COMM_RANK(comm,newid)
```

If (myid%2 = 0) then

/* Group 0 create intercommunicator and send Message : local leader =0, remote leader = 1, tag = 99*/

MPI_INTERCOMM_CREATE(comm,0,MPI_COMM_WORLD,1,99,intercomm)

MPI_SEND(msg,1,type,newid,0,intercomm)

Else

/* Group 1 create intercommunicator and send Message : local leader =0, remote leader = 1, tag = 99*/

MPI_INTERCOMM_CREATE(comm,0,MPI_COMM_WORLD,0,99,intercomm)

MPI_SEND(msg,1,type,newid,0,intercomm,status)

/* Free communicators */

MPI_COMM_FREE(intercomm)
MPI_COMM_FREE(comm)