MPI Programming Model

- A computation comprises one or more processes that communicate by calling library routines to send and receive messages to other processes

Process Creation and Execution

- A fixed set of processes created at program initialization
- Only static process creation
  - All processes must be defined prior to execution and started together
- One process per processor
- Use SPMD model of computation
- May execute different programs-MPMD

Communicators

- Point to point : send a message from one process to another process
- Collective: global operations (summation/broadcast)
- Probe:asynchronous communication
- Processes have ranks associated with the communicator
- Initially all processes enrolled in a "universe" called MPI_COMM_WORLD, and each process is given a unique rank from 0 to n-1, where there are n processes

Basics

- MPI_INIT (initiate an MPI computation)
  - Must be called before any other MPI function and must be called exactly once per process
- MPI_FINALIZE (Terminate a computation)
  - No MPI function can be called after MPI_FINALIZE
- MPI_COMM_SIZE (Determine number of processes)
- MPI_COMM_RANK (Determine my process id)
- MPI_SEND (send a message)
- MPI_RECV (Receive a message)

More than Basic

```
#include <stdio.h>
#include "mpi.h"
 void main( int argc, char **argv )
 {
   int rank;  /* Rank of process*/
   int size;  /*Number of processes*/
   MPI_Init( &argc, &argv );
   MPI_Comm_size( MPI_COMM_WORLD, &size );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   printf( "Hello world from process %d of %d \n",rank,
    size );
   MPI_Finalize();
 }
```


Using the SPMD Computation Model

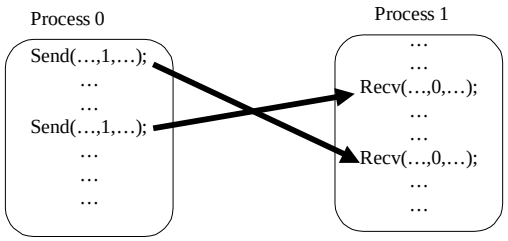
```
Main(int argc, char *argv[])
{
      MPI_INIT(&argc, &argv);
      ....
      ....
      MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
      if (myrank == 0)
            master();
      else
            slave();
      ...
    MPI_Finalize();
}
```

Where master() and slave() are procedures to be executed by the master process and slave process, respectively

Global and Local Variables

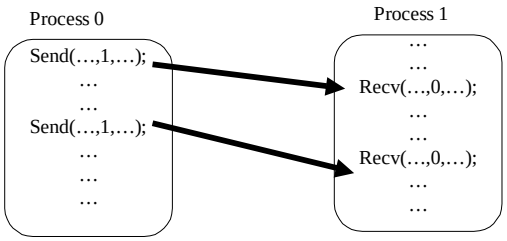
- Any global declarations of variables will be duplicated in each process
- Variables that are not to be duplicated will need to be declared within code only executed by that process

Example



Intended Behavior

Example(Unsafe message passing with libraries



Possible Behavior

Blocking Routines in MPI

- Return when they are locally complete-when the location used to hold the message can be used again or altered without affecting the message being sent
- A blocking send will send the message and return.
- This does not mean that the message has been received, just that the process is free to move on without adversely affecting the message

Point-to-Point Communication

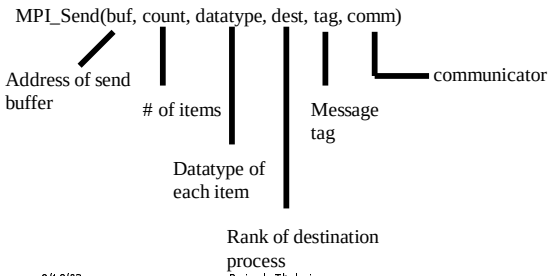
- Message tags are present and wild cards can be used in place of tag (MPI_ANY_TAG) and in place of the source in receive routines (MPI_ANY_SOURCE)
- Datatypes defined in send/receive parameters

- **Communicators :** used in MPI for all point to point and collective MPI message passing communication
- A communicator is a *communication domain* that defines a set of processes that are allowed to communicate between themselves
- In this way, the communicator domain of the library can be separated from that of a user program
- Each process has a rank within the communicator, an integer from 0 to n-1, where there are n processes

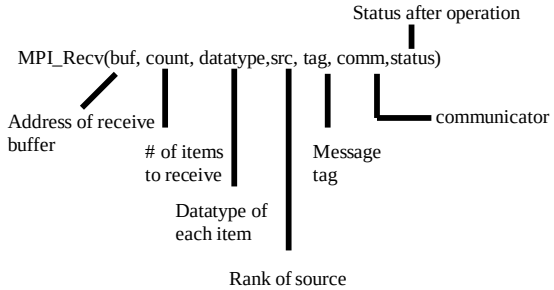
Communicator Types

- **Intracommunicator:** for communicating within a group
- **Intercommunicator:** for communication between groups
- A process has a unique rank in a group (an integer from 0 to m-1 where there are m processes in the group). A process could be a member of more than one group
- **Default intracommunicator:** MPI_COMM_WORLD, exists as the first communicator for all the processes existing in the application. New communicators are created based on existing communicators. A set of MPI routines exists for forming communicators

Send Parameters



Receive Parameters



Example

To send an integer x from process 0 to process 1

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
If (myrank == 0) {
        int x;
        MPI_Send(&x, 1, MPI_INT,1,msgtag, MPI_COMM_WORLD);
} else if (myrank  == 1) {
        int x;
        MPI_Recv(&x, 1, MPI_INT, 0, msgtag,
                MPI_COMM_WORLD, status);
}
```


Solution:Tags

- Distinguish between different messages
- Reduces errors
- MPI_ANY_TAG : may lead to non-determinism (Example 8.2)

Collective Communication

- Involves set of processes as opposed to sender and receiver process.
- Defined by an intra-communicator
- Message tags are not present.

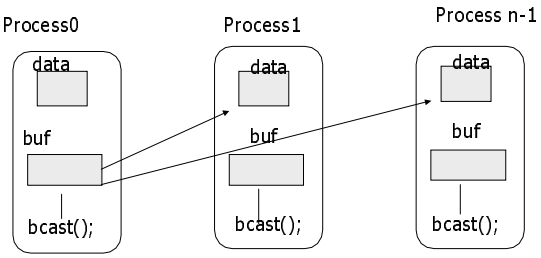
Broadcast and Scatter Routines

- MPI_Bcast() – Broadcast from root to all other processes
- MPI_Gather() – Gather values for group of processes
- MPI_Scatter() – Scatters buffer in parts to group of processes
- MPI_AlltoAll() – Sends data from all processes to all processes

Broadcast routine

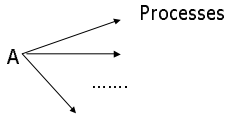
- Broadcast the same message from the root process to the other processes
- Therefore need to identify:
  - The group of processes
  - The root process

Broadcast action does not occur until all the processes have executed their broadcast routine.
Broadcast operation has the effect of synchronizing the processes.



MPI_Bcast() –
Broadcast the same message from root to all other processes

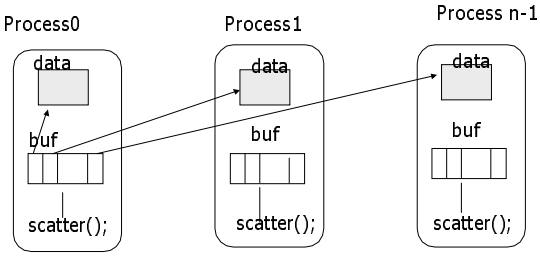
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int source, MPI_Comm Comm)



Sends the data stored in buffer (buf) of process source to all Other processes in the group.

Scatter Operation

- Sending each element of an array of data in the root to separate process.
- The contents of the ith of the array is sent to the ith process.
- As with broadcast need to identify
  - Group of processes
  - The root process



Each process executes the same scatter() routine.

Scatter

Send buffer

Number of elements Sent to each process

```
int MPI_Scatter(void *sendbuf, int sendcount,
                MPI_Datatype sendtype, void *recvbuf,
                int recvcount, MPI_Datatype recvtype, int root,
                MPI_Comm comm)
```

Receive buffer

Process *i* receives sendcount contiguous elements of type Senddatatype starting from *i*sendcount* location of sendbuf.

MPI_Scatterv

array

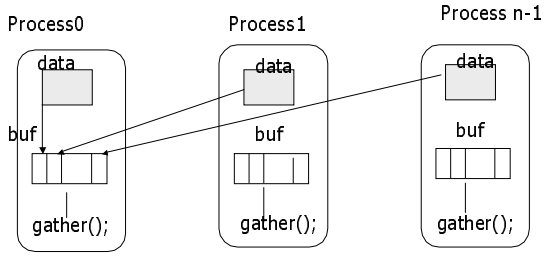
Allows different amount of data to be sent to each process

```
int MPI_Scatterv(void *sendbuf, int *sendcount, int *displs,
                 MPI_Datatype sendtype, void *recvbuf,
                 int recvcount, MPI_Datatype recvtype, int
                 root, MPI_Comm comm)
```

The root process sends sendcount[j] elements to process j
Array displs determines from where in sendbuf these elements
Will be sent from. So, process j, data start from displs[j]

Gather Operation

- One process collects individual v alues from a set of processes.
- Gather is normally used after some computation has been done by these processes
- Gather is the opposite of scatter
- The data from the ith process is
  - received by the root process and
  - placed in the ith location of array set aside to receive the data.



Each process executes the same gather() routine.

Gather

```
int MPI_Gather(void *sendbuf, ine sendcount,
               MPI_Datatype sendtype, void *recvbuf,
               int recvcount, MPI_Datatype recvtype, int root,
               MPI_Comm comm)
```


To gather items from the group of processes into process 0, using

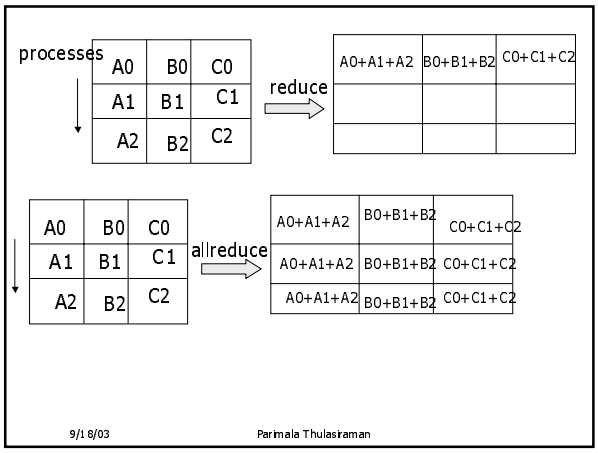
Dynamically allocated memory in the root process,

```
int data[10];
MPI_COMM_rank(MPI_COMM_WORLD,&myrank);
If (myrank == 0) {
    MPI_COMM_size(MPI_COMM_WORLD, &grp_size);
    buf = (int*) malloc(grp_size*10*sizeof(int));
    }
    MPI_Gather(data,10,MPI_INT,buf,grp_size*10,MPI_INT,0,MPI_
       COMM);
```

MPI_Gather() gathers from all processes, including the root.

Reduce operation

- Gather operations can be combined with specified arithmetic or logical operation.
- For example, values could be gather together and added by the root.


processes
A0
B0
C0
A1
B1
C1
A2
B2
C2
reduce
A0+A1+A2
B0+B1+B2
C0+C1+C2
A0
B0
C0
A1
B1
C1
A2
B2
C2
allreduce
A0+A1+A2
B0+B1+B2
C0+C1+C2
A0+A1+A2
B0+B1+B2
C0+C1+C2
A0+A1+A2
B0+B1+B2
C0+C1+C2
9/18/03
Parimala Thulasiraman