What is MPI?

- **A standard message-passing library**
  - p4, NX, PVM, Express, PARMACS are precursors
- **MPI defines a language-independent interface**
  - Not an implementation
- **Bindings are defined for different languages**
  - So far, C and Fortran 77, C++ and F90
  - Java Grande Forum is defining Java bindings
- **Multiple implementations**
  - MPICH is a widely-used portable implementation
  - See http://www.mcs.anl.gov/mpi/

The Six Fundamental MPI routines

- **MPI_Init (argc, argv)**
  - initialize
- **MPI_Comm_rank (comm, rank)**
  - find process label (rank) in group
- **MPI_Comm_size(comm, size)**
  - find total number of processes
- **MPI_Send (sndbuf, count, datatype ,dest, tag, comm)**
  - send a message
- **MPI_Recv (recvbuf, count, datatype, source, tag, comm, status)**
  - receive a message
- **MPI_Finalize( )**
  - End Up

MPI_Init
Environment Management

- **This MUST be called to set up MPI before the invocation of any other MPI routines**

- **int MPI_Init(int *argc, char **argv)**
  - argc and argv are conventional C main routine arguments

MPI_Comm_rank Environment Inquiry

- **This allows you to identify each process by a unique integer called the rank which runs from 0 to N-1 where there are N processes**

- **int MPI_Comm_rank(MPI_Comm comm, int *rank)**
  - comm is an MPI communicator of type MPI_Comm

MPI_Comm_size Environment Inquiry

- **This returns in integer size number of processes in given communicator comm (remember this specifies processor group)**

- **For C: int MPI_Comm_size(MPI_Comm comm,int *size)**
  - where comm, size, mpierr are integers
  - comm is input; size and mpierr returned

Point-to-Point Communication

Send

Mode	**Blocking**	**Nonblocking**
Standard	*mpi_send*	*mpi_isend*
Buffered	*mpi_bsend*	*mpi_ibsend*
Synchronous	*mpi_ssend*	*mpi_issend*
Ready	*mpi_rsend*	*mpi_irsend*

Receive

Blocking	**Nonblocking**
mpi_recv	*mpi_irecv*

Sending a message

int MPI_Send(buf, count, datatype, dest, tag, comm)

void ***buf**	starting address of the data to be sent
int **count**	number of elements to be sent
MPI_Datatype **datatype**	MPI datatype of each element
int **dest**	rank of destination process
int **tag**	message marker (set by user)
MPI_Comm **comm**	MPI communicator of processors involved

Example:

```
MPI_Send(data,500,MPI_FLOAT,6,33,MPI_COMM_WORLD);
```

Receiving a message

int MPI_Recv(buf, count, datatype, source, tag, comm, status)

void ***buf**	starting address of the data to be received
int **count**	number of elements to be received
MPI_Datatype **datatype**	MPI datatype of each element
int **source**	rank of source process
int **tag**	message marker (set by user)
MPI_Comm **comm**	MPI communicator of processors involved
MPI_Status ***status**	status of receiving command

Example:

```
MPI_Send(data,500,MPI_FLOAT,6,33,MPI_COMM_WORLD ,status);
```

Wildcarding

•Receiver can wildcard

•To receive from any source -- MPI_ANY_SOURCE

•To receive with any tag -- MPI_ANY_TAG

•Actual source and tag are returned in the receiver's status parameter

Compilation and Execution

```
kid1# mpicc foo.c –o bar

kid1# mpirun -np 16 bar

kid1# cat <<EOF > kids
kid4
kid5
kid6
kid7
EOF

kid1# mpirun –np 4 –machinefile kids foo
```

```
#include "mpi.h"
main( argc, argv )
int argc;
char **argv;
{
   char message[20];
   int myrank;
   MPI_Status status;
   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
   if (myrank == 0)    /* code for process zero */
   {
      strcpy(message,"Hello, there");
      MPI_Send(message, strlen(message), MPI_CHAR, 1, 99, MPI_COMM_WORLD);
   }
   else                /* code for process one */
   {
      MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
      printf("received :%s:\n", message);
   }
   MPI_Finalize();
}
```

Hello World in C plus MPI

```
#include <stdio.h>
#include <mpi.h>
void main(int argc,char *argv[]) {
   int ierror, rank, size
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if( rank == 0)
      printf ("hello World!\n");
   ierror = MPI_Comm_size(MPI_COMM_WORLD, &size);
   if(ierror != MPI_SUCCESS )
       MPI_Abort(MPI_COMM_WORLD, ierror);
   printf("I am processor %d out of total of %d\n", rank, size);
   MPI_Finalize();
}
```

Collective Communication

Provides standard interfaces to common global operations

- Synchronization
- Communications, i.e. movement of data
- Collective computation

- **A collective operation uses a process group**
  - All processes in group call same operation at (roughly) the same time
  - Groups are constructed “by hand” with MPI group manipulation routines or by using MPI topology-definition routines

- **Message tags not needed (generated internally)**

- **All collective operations are blocking**

Some Collective Communication Operations

- **MPI_BARRIER(comm)** Global Synchronization within a given communicator
- **MPI_BCAST** Global Broadcast
- **MPI_GATHER** Concatenate data from all processors in a communicator into one process
- **MPI_ALLGATHER** puts result of concatenation in all processors
- **MPI_SCATTER** takes data from one processor and scatters over all processors
- **MPI_ALLTOALL** sends data from all processes to all other processes

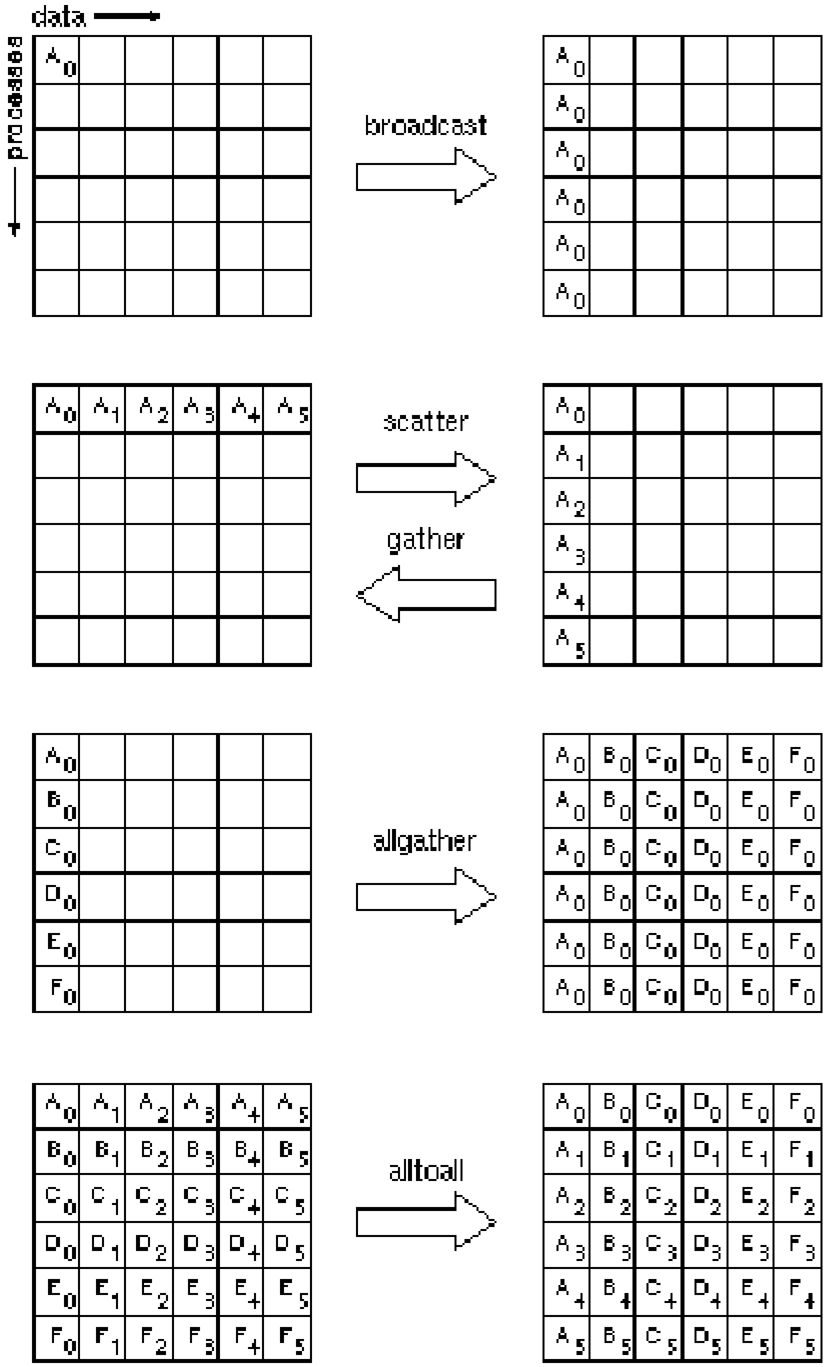
- •MPI_BCAST

- •MPI_SCATTER

- •MPI_GATHER

- •MPI_ALLGATHER

- •MPI_ALLTOALL


data
processes
A_0
broadcast
A_0 A_0 A_0 A_0 A_0 A_0
A_0 A_1 A_2 A_3 A_4 A_5
scatter
gather
A_0 A_1 A_2 A_3 A_4 A_5
A_0 B_0 C_0 D_0 E_0 F_0
allgather
A_0 B_0 C_0 D_0 E_0 F_0
A_0 A_1 A_2 A_3 A_4 A_5
B_0 B_1 B_2 B_3 B_4 B_5
C_0 C_1 C_2 C_3 C_4 C_5
D_0 D_1 D_2 D_3 D_4 D_5
E_0 E_1 E_2 E_3 E_4 E_5
F_0 F_1 F_2 F_3 F_4 F_5
alltoall
A_0 B_0 C_0 D_0 E_0 F_0
A_1 B_1 C_1 D_1 E_1 F_1
A_2 B_2 C_2 D_2 E_2 F_2
A_3 B_3 C_3 D_3 E_3 F_3
A_4 B_4 C_4 D_4 E_4 F_4
A_5 B_5 C_5 D_5 E_5 F_5

MPI Communicator

- Programmer view: group of processes that are allowed to communicate with each other

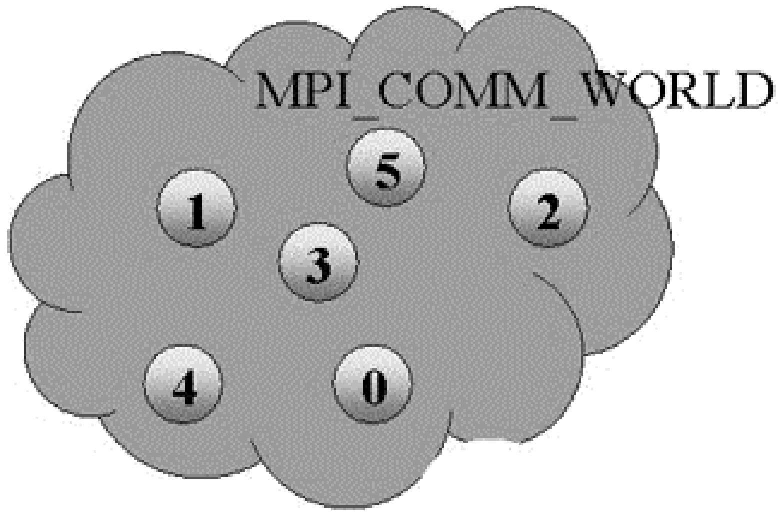
- All MPI communication calls have a communicator argument

- Most often use MPI_COMM_WORLD

- Defined by MPI_Init

- It is all your processors...

MPI_COMM_WORLD
communicator



Rank

- Process ID number within the communicator

- Starting with zero

- Routines:

```
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

- Used to specify source and destination of messages

Size

- How many processes are contained within a communicator?

```
MPI_Comm_size(MPI_Comm comm,int *size)
```

Bones.c

```
#include<mpi.h>
void main(int argc, char *argv[]) {
   int rank,size;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Comm_size(MPI_COMM_WORLD,&size);
   /* ... your code here ...*/
   MPI_Finalize();
}
```

COURSES HOMEPAGES

Parallel
http://www.cslab.ece.ntua.gr/courses/PPS/

Distributed
http://www.cslab.ece.ntua.gr/courses/PPS/