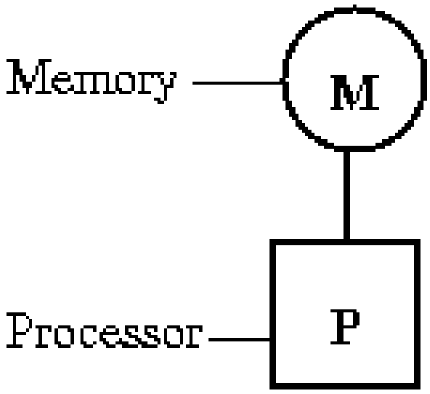
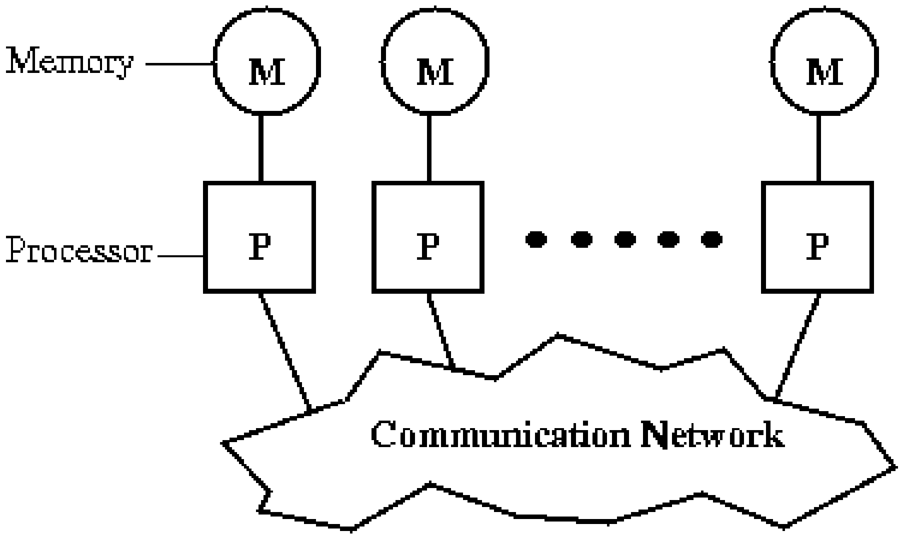
MPI: THE MESSAGE-PASSING INTERFACE

SEQUENTIAL PROGRAMMING PARADIGM:



THE MESSAGE-PASSING PARADIGM:



- Simplicity
- Efficient match to many parallel hardware
- Performance

WHAT IS MPI ?

- A *proposed standard* message-passing library specification
  - message-passing model
  - not a compiler specification
  - not a specific product

- For parallel computers and heterogeneous networks

WHY IS A STANDARD NEEDED ?

- Portability and ease-of-use,
- Provides hardware vendors with a well-defined set of routine to implement efficiently,
- Pre-requisite for development of concurrent software industry,
- Will lead to more widespread use of concurrent computers.

HISTORY

- Broad participation
- Vendors
  - IBM, Intel, TMC, Meiko, Cray, Convex, Ncube
- Library writers
  - PVM, p4, Zipcode, TCGMSG, Chameleon, Express, Linda
- **May 1994**. Final MPI specification released.

WHAT IS IN MPI ?

- Point-to-point message message-passing
- Collective communication
- Support for process groups
- Support for communication contexts
- Support for application topologies
- Profiling interface

PROCESS MODEL AND GROUPS

- Fundamental computational unit is the process. Each process has:
  - an independent thread of control,
  - a separate address space
- MPI processes execute in MIMD style, but:
  - No mechanism for loading code onto processors, or assigning processes to processors
  - No mechanism for creating or destroying processes
- MPI processes belong to groups
  - Process groups can be created and destroyed
  - Membership is static
  - Groups may overlap

COMMUNICATION SCOPE

- In MPI, a process is specified by:
  - a group
  - a rank relative to the group (0, 1, 2, ..., N-1)
- A message label is specified by:
  - a message context
  - a message tag relative to the context
- Groups are used to partition process space
- Contexts are used to partition “message label space”'
- Groups and contexts are bound together to form a **communicator** object. Contexts are not visible at the application level.
- A communicator defines the **scope** of a communication operation

A SIMPLE MPI PROGRAM

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

int main(argc,argv)
int argc;
char *argv[];
{
  int myrank;      /* Rank of process */
  int numprocs;   /* Number of processes */

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

  printf( "Process %d of %d says Hello!\n",
            myrank, numprocs);

  MPI_Finalize();
}
```

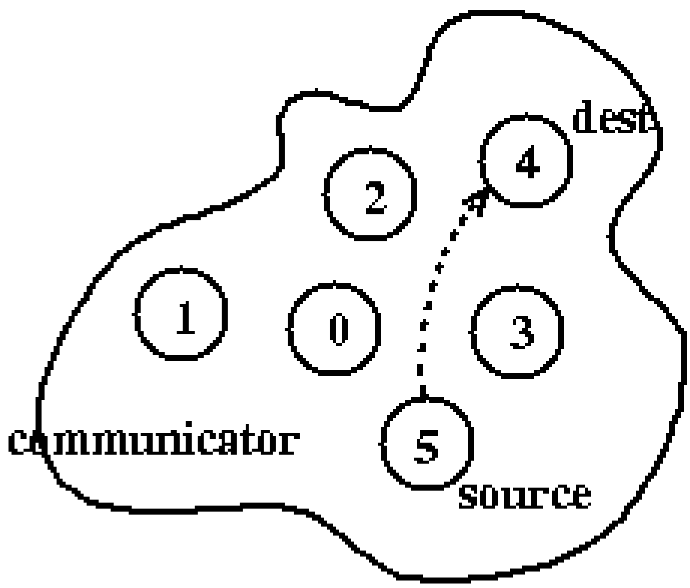
- Must be compiled with C compiler
- Must be linked with the MPI library
- Command to run depends on the MPI implementation

```
hello -np 4

Process 1 of 4 says Hello!
Process 2 of 4 says Hello!
Process 0 of 4 says Hello!
Process 3 of 4 says Hello!
```

POINT-TO-POINT COMMUNICATION

- Always involves exactly two processes (sender-receiver).
- Message selectivity is by rank and message tag.
- Rank and tag are interpreted relative to the scope of the communication.
- The scope is specified by the communicator.
- Rank and tag may be wildcarded.
- The communicator may not be wildcarded



BASIC SEND/RECEIVE

- The standard MPI **send** routine:

```
int MPI_Send( void*            start_of_buffer,
              int              number_of_items,
              MPI_Datatype     datatype,
              int              destination_rank,
              int              message_tag,
              MPI_Comm         communicator)
```

- The standard MPI **receive** routine:

```
int MPI_Recv( void*            start_of_buffer,
              int              max_number_of_items,
              MPI_Datatype     datatype,
              int              source_rank
              int              message_tag,
              MPI_Comm         communicator,
              MPI_Satus*       return_status)
```

- Message buffer is always described by a triple:
  (**address, number_of_items, datatype**).

  – **datatype** may either be *basic* MPI datatype (e.g. MPI_INT, MPI_CHAR, ...) or *derived* MPI type.

  – the corresponding *send* and *receive* datatypes must match

- **source_rank** and **tag** may be wildcarded in the receive routine (MPI_ANY_SOURCE, MPI_ANY_TAG).

- there is no wildcard for the **communicator** (i.e. the actual value must be the same at the corresponding sender and receiver sides).

- The return status object is used after completion of a receive to find the actual ength, source, and tag of the message.

RETURN STATUS OBJECT

- **satus** is a structure.
  - **status.source** gives source process
  - **status.tag** gives the message tag
  - number of elements in message is given by:

```
int MPI_Get_count( MPI_Status      status,
                   MPI_Datatype    datatype,
                   int*            number_of_items)
```

COMMUNICATION COMPLETION

- A send operation is complete if the buffer can safely be reused.
- A receive operation is complete if the buffer is ready for use.
- MPI_Send and MPI_Recv do not return until the communication operation is complete, i.e. they are *blocking* operations.

EXAMPLE : BASIC SEND/RECEIVE

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

int main(argc,argv)
int argc;
char *argv[];
{
  int myrank;                 /* Rank of process     */
  int numprocs;               /* Number of processes */
  int source,dest;            /* Ranks of sender and receiver */
  char message[100];          /* Storage for the message      */
  MPI_Status status;          /* Return status for receive    */

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

  if (myrank != 0)
   { sprintf(message, "Greetings from process %d!",myrank);
     dest = 0;
     /* Use strlen(message)+1 to include `\0` */
     MPI_Send( message, strlen(message)+1, MPI_CHAR,
               dest, 15, MPI_COMM_WORLD);
   } else {
     for (source = 1; source < numprocs; source++)
      { MPI_Recv( message, 100, MPI_CHAR,
                  source, 15, MPI_COMM_WORLD, &status);
        printf("%s\n", message);
      }
   }
  MPI_Finalize();
}
```

SIX FUNCTION MPI

These six function allow you to write many programs:

1. **MPI_Init** (Initialize MPI)
2. **MPI_Comm_size** (Find out how many processes there are)
3. **MPI_Comm_rank** (Find out which process I am)
4. **MPI_Send** (Send a message)
5. **MPI_Recv** (Receive a message)
6. **MPI_Finalize** (Terminate MPI)

BLOCKING BEHAVIOUR

- **Blocking send:**
  - returns when send is locally complete
  - message buffer can be written to after return
- **Non-blocking send:**
  - returns ``immidiately''
  - message buffer should not be written to after return
  - must check for local completion
- **Blocking receive:**
  - returns when receive is locally complete
  - message buffer can be read from after return
- **Non-blocking receive:**
  - returns ``immidiately''
  - message buffer should not be read from after return
  - must check for local completion

NON-BLOCKING SEND/RECEIVE

- The standard non-blocking send routine

```
int MPI_Isend( void*            start_of_buffer,
               int              number_of_items,
               MPI_Datatype     datatype,
               int              destination_rank,
               int              message_tag,
               MPI_Comm         communicator,
               MPI_Request*     request)
```

- The standad non-blocking receive routine:

```
int MPI_Recv( void*            start_of_buffer,
              int              max_number_of_items,
              MPI_Datatype     datatype,
              int              source_rank,
              int              message_tag,
              MPI_Comm         communicator,
              MPI_Request*     request)
```

- Receive does not pass a return status object.

COMPLETION ROUTINES

- Two basic ways of checking on non-blocking sends and receives:

  - call a wait routine that blocks until completion

```
int MPI_Wait( MPI_Request    *request,
              MPI_Status     *status)
```

MPI_Wait blocks until the communication is complete

  - call a test routine that returns a flag to indicate if complete

```
int MPI_Test( MPI_Request    *request,
              int            *flag,
              MPI_Status     *status)
```

MPI_Test returns “immidiately”, and sets the **flag** to true if the communication is complete

- Use of non-blocking and completion routines allow computation and communication to be overlapped.

“UNSAFE” COMMUNICATION

- Standard blocking send and receive are simple, but can be “unsafe”:

Process 0	Process 1
Send(1)	Send(0)
Recv(1)	Recv(0)

Completion depends in general on size of message and amount of system buffering. However, system buffering is not in the scope of MPI.

SOME SOLUTIONS TO THE "UNSAFE" PROBLEM

- Order the operations more carefully:

Process 0	Process 1
Send(1)	Send(0)
Recv(1)	Recv(0)

- Supply receive buffer at same time as send, with MPI_Sendrecv:

Process 0	Process 1
Sendrecv(1,1)	Sendrecv(0,0)

- Use non-blocking operations:

Process 0	Process 1
Isend(1)	Isend(0)
Irecv(1)	Irecv(0)
Waitall	Waitall

- Provide the necessery amount of buffer explicitly and use MPI_Bsend

Process 0	Process 1
Attach_buffer()	Attach_buffer()
Bsend(1)	Bsend(0)
Recv(1)	Recv(0)

COMMUNICATION MODES

MPI provides multiple *modes* for sending messages:

- **S**ynchronous mode (MPI_**S**send): the send does not complete until a matching receive has begun.
- **B**uffered mode (MPI_**B**send): the user supplies the buffer to system for its use:

  int MPI_Buffer_attach(void* buffer, int size)

- **R**eady mode (MPI_**R**send): user guarantees that matching receive has been posted.
  - allows access to fast protocols
  - undefined behaviour if the matching receive is not posted

They differ only in one respect: how completion of the send depends on the receipt of the message.

Non-blocking versions:

MPI_Issend, MPI_Ibsend, MPI_Irsend

Note that any MPI_Recv (or MPI_Irecv) may receive messages sent with any send mode.

FLAVORS OF COMMUNICATION

- For send operation there are:

  4 communication modes

  2 blocking modes

  $\Rightarrow 4 \times 2 = 8$ types of send

SEND mode	Blocking	Non-blocking
Standard	**MPI_Send**	**MPI_Isend**
Buffered	**MPI_Bsend**	**MPI_Ibsend**
Synchronous	**MPI_Ssend**	**MPI_Issend**
Ready	**MPI_Rsend**	**MPI_Irsend**

- For receive operation there are:

  1 communication mode

  2 blocking modes

  $\Rightarrow 1 \times 2 = 2$ types of receive

Mode	Blocking	Non-blocking
Standard	**MPI_Recv**	**MPI_Irecv**

DERIVED DATATYPES

- In MPI, the communication buffer is defined by *a starting address* and a *datatype*. The datatype may be:

  - **Basic type** (e.g. MPI_INT, MPI_CHAR, MPI_LONG, MPI_DOUBLE... etc.

  - **Derived type**

- A general datatype is a sequence of primitive types and integer byte displacements

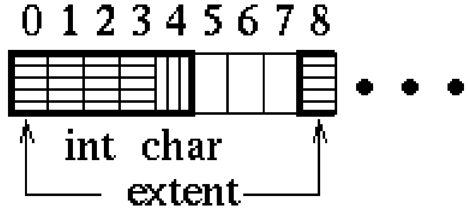
  Typemap = $\{(type_0, disp_0), \ldots, (type_{n-1}, disp_{n-1})\}$

  - E.g.: MPI_INT $\Rightarrow$ (int,0)

  - Displacements are relatives to the starting address of the buffer

- *Extent* of a datatype : the span from the first byte to the last byte, rounded up to satisfy alignment requirements.

  E.g. $\{(int, 0),(char, 4)\} \Rightarrow$ extent: 8 - 0 = 8.




```
int MPI_Type_extent( MPI_Datatype      datatype,
                     int     *extent)
```

CREATING DERIVED DATATYPES

- Derived datatypes are created at run-time. This is done in two stages:

  - **Construct** the datatype. New datatypes are built up from existing ones.

  - **Commit** the datatype. The new datatype is "committed" with a call to:

    int MPI_Type_commit(MPI_Datatype *datatype)

- After being committed, it can be used in any number of communications.

DATATYPE CONSTRUCTORS

- **Contignuous** Replicates a datatype into contignuous locations.

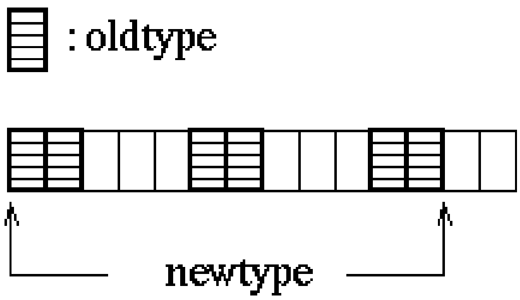
  int MPI_Type_contiguous(int count,
  MPI_Datatype oldtype,
  MPI_Datatype *newtype)

  newtype: concatenation of *count* copies of *oldtype*

- **Vector** Replicates a datatype into locations that consist of equally spaced blocks.

  int MPI_Type_vector(int count,
  int blocklength,
  int stride,
  MPI_Datatype oldtype,
  MPI_Datatype *newtype)

  E.g. count=3, blocklength=2, stride=5 :



DATATYPE CONSTRUCTORS (CONT.)

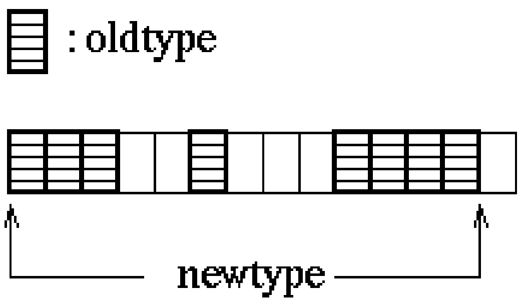
- **Indexed**. Replicates a datatype, taking blocks at fixed offsets. Each block may have different length.

```
int MPI_Type_indexed( int           count,
                      int           *array_of_blocklengths,
                      int           *array_of_displacements,
                      MPI_Datatype  oldtype,
                      MPI_Datatype  *newtype)
```

count=3

array_of_blocklengths=(3,1,4)

array_of_displacements=(0,5,9)



- **Struct**. Generalizes the indexed datatype by allowing each block to be of a different datatype.

```
int MPI_Type_struct( int            count,
                     int            *array_of_blocklengths,
                     MPI_Aint       *array_of_displacements,
                     MPI_Datatype   *array_of_types,
                     MPI_Datatype   *newtype)
```

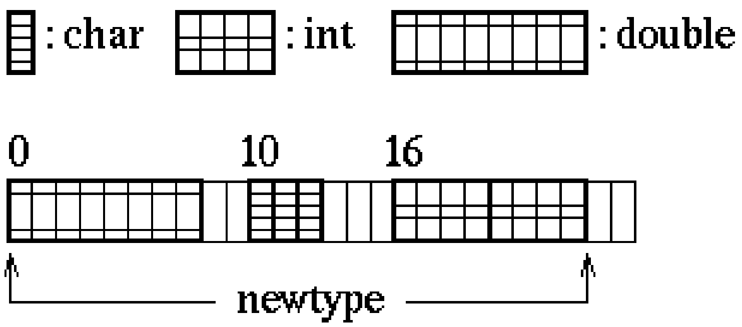
Displacements are given in bytes.

count=3

array_of_blocklengths=(1,3,2)

array_of_displacements=(0,10,16)

array_of_types=(MPI_DOUBLE, MPI_CHAR, MPI_INT)

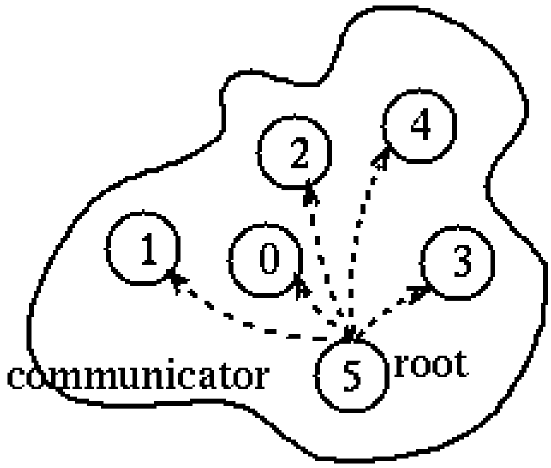


COLLECTIVE COMMUNICATION

- Involves coordinated communication within a group of processes
- No message tags are used
- The amount of data sent must exactly match the amount of data specified by the receiver.
- All collective routines block until they are locally complete
- All members of the group must call the same routine, with matching arguments
- Two broad classes:
  - data movement routines
  - global computation routines

COLLECTIVE DATA MOVEMENT

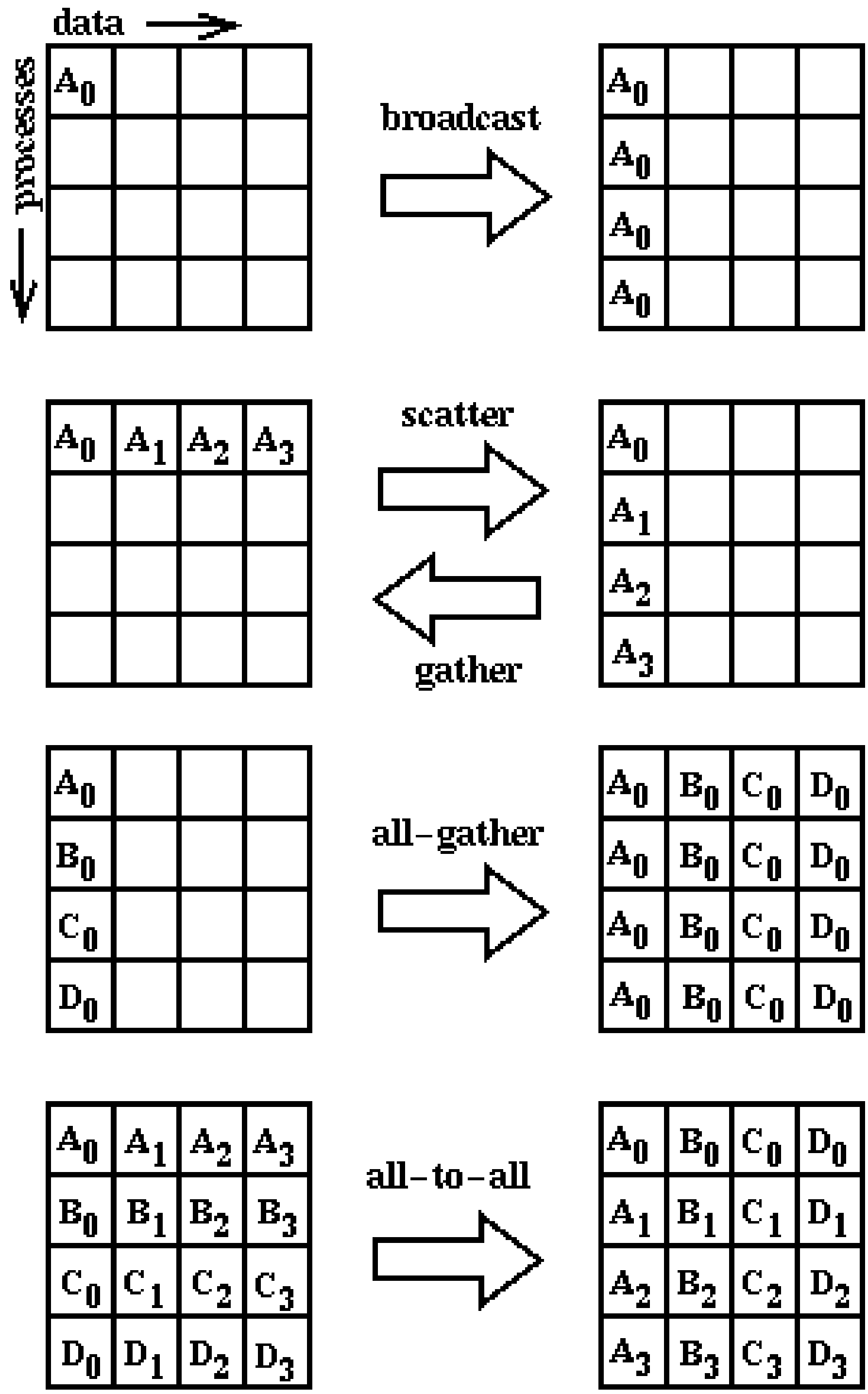
- Simple example: broadcast



int MPI_Bcast(	void*	buf,
	int	count,
	MPI_Datatype	datatype,
	int	root,
	MPI_Comm	comm)

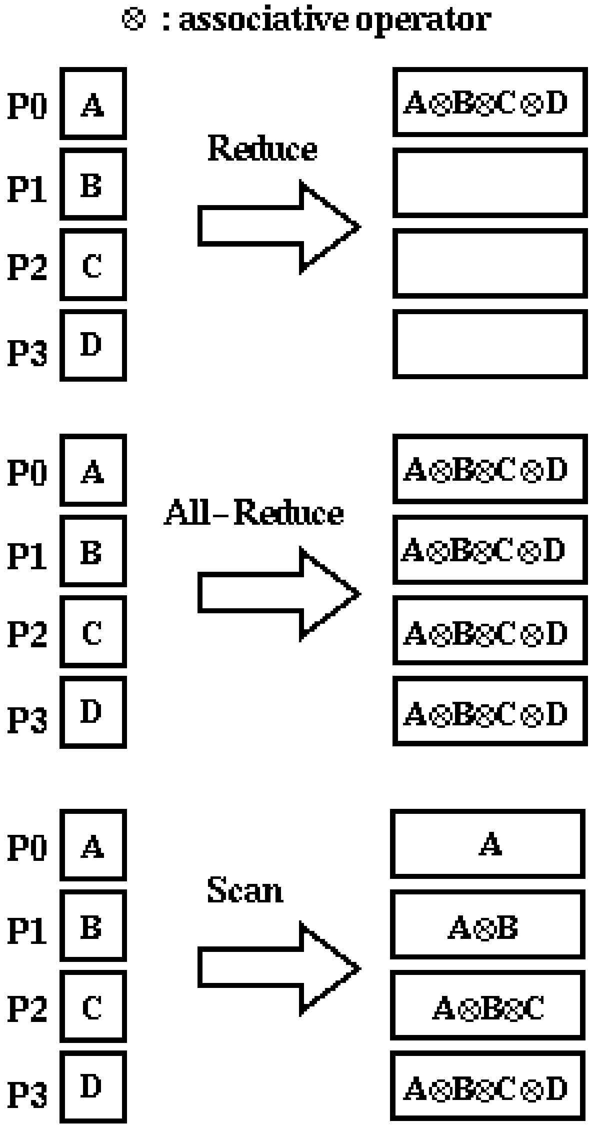
- the contents of root's communication buffer is to be copied to all processes
- comm and root must be the same at all members of the group

COLLECTIVE DATA MOVEMENT



Schematic representation of collective data movement in MPI

GLOBAL COMPUTATION



Schematic representation of global computation in MPI

- The function passed to a global computation routine is either:

  - a predefined MPI function, e.g., mpi_sum

  - a user supplied function

APPLICATION TOPOLOGIES

- Mechanism for naming processes in a way that fits the communication pattern better

- Proceses may be configured into one of two topology types:

  – *Cartesian* topologies

  – *General Graph* topologies

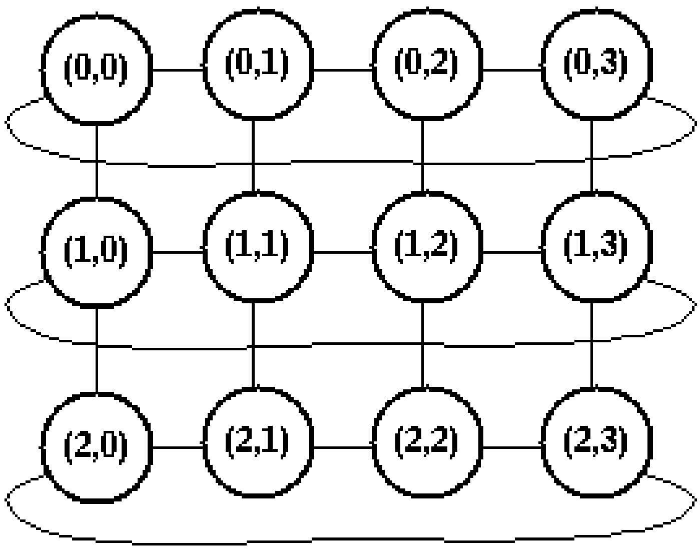
- Creating Cartesian grid topologies:

```
int MPI_Cart_create( MPI_Comm    comm_old,
                     int         ndims,
                     int*        dims,
                     int*        periods,
                     int         reorder,
                     MPI_Comm*   comm_cart)
```

  – ndims=2

  – dims=(4,3)

  – periods=(1,0)



USES OF TOPOLOGIES

- Knowledge of application topology can be used to efficiently assign processes to processors (*reorder* argument).
- MPI provides support for shifting data along a specified dimension of a Cartesian grid.
- MPI provides support for performing collective operations along a specified grid direction.

RANK ←→ COORDINATES

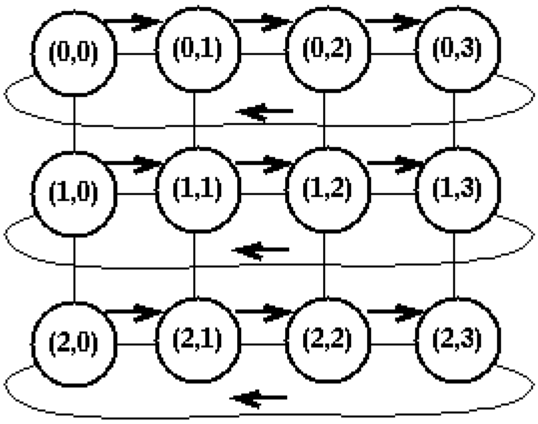
- Mapping of rank to coordinate position in Cartesian topology:

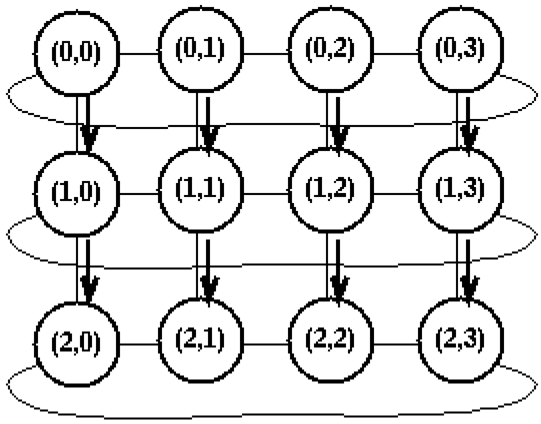
```
int MPI_Cart_coords( MPI_Comm   comm,
                     int        rank,
                     int        maxdims,
                     int*       coords)
```

- Mapping of coordinate position to rank:

```
int MPI_Cart_rank( MPI_Comm   comm,
                   int*       coords,
                   int*       rank)
```

TOPOLOGIES AND DATA SHIFTS





- Ranks of the processes that a process must send to and receive from when performing a shift:

```
int MPI_Cart_shift( MPI_Comm    comm,
                    int         direction,
                    int         disp,
                    int*        rank_source,
                    int*        rank_dest)
```

direction=1
disp=1 $\Rightarrow$ process (x, y) is to send to (x, y+1)
receive from (x, y-1)

- Shift operations are performed using **MPI_Sendrecv**

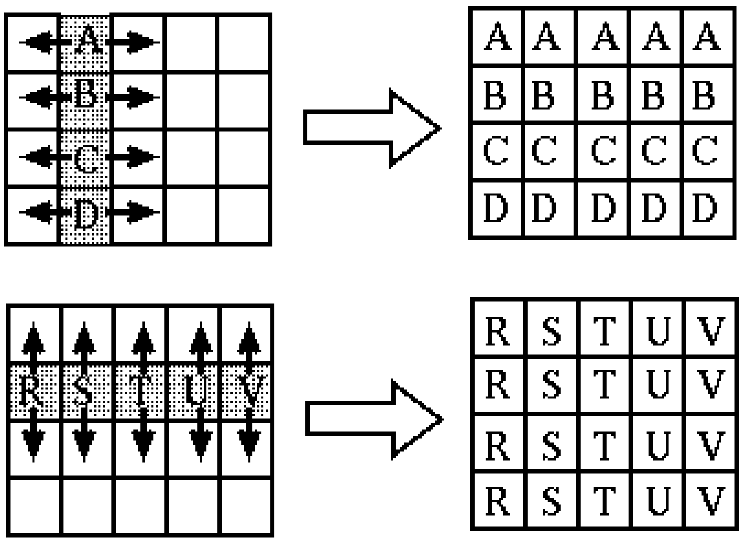
PARTITIONING CARTESIAN TOPOLOGIES

- A Cartesian topology can be partitioned into a set of Cartesian topologies of lower dimension using:

```
int MPI_Cart_sub( MPI_Comm      comm,
                  int*          remain_dims,
                  MPI_Comm*     newcomm)
```

remain_dims[i] is true if the *i*th dimension is retained, and is false if it is discarded.

- Suppose we want to perform a collective communication along a dimension of a Cartesian grid, e.g., a multicast:



  - We can call **MPI_Cart_sub** with *remain_dims*=(false, true) to generate row subcommunicators for first case.

  - If *remain_dims*=(true, false) we get the subcommunicators for the second case.

FINAL COMMENTS

- Additional features of MPI not covered in this tutorial
  - Persistent commmunication
  - Tools for writing libraries
  - Inter-communicators
  - Profiling interface
  - Error handling
- The MPI forum (with old and new participants) has begun a follow-on series of meetings to address advanced features such as:
  - dynamic process management
  - client/server
  - "one-sided" communication (put/get, active messages)
  - real-time extensions
  - language bindings for C++ and Fortran-90
- Several implementations of MPI exist.