Introduction to MPI

Table of Contents

MPI Program Structure

1. MPI is a set of precompiled library routines that the user links with their code.

2. An "MPI" parallel program is a sequential program which has been modified to include calls to MPI routines and conditional statements to adapt the execution of the program to its local context.

3. Each processor executes the same program using local processor id to determine its behavior.

4. MPI distributes the programs to the processors, loads them and initiates execution on each processor.

5. Environment specification and execution initiation is external to MPI

Communication Model

1. A communicator (MPI_Comm) is a collection of processors that can send messages to each other. For basic programs, the only communicator needed is MPI_COMM_WORLD. It is predefined in MPI and consists of all the processors running when program execution begins.
2. Subsets of MPI_COMM_WORLD can be created to partition the processors into smaller communication groups.
3. Message communicators much match between message sender and receiver.
4. Communicators can also be used to determine the number of processors participating in a particular communicator set and the sequence of the processor in the communicator.
5. The processor's location in the communicator sequence is determined by the MPI_Comm_rank function.
6. The total number of processors in the communicator can be determed by the MPI_Comm_size.

Introduction to MPI

MPI_INIT(int *argc, char ***argv)
Initiate a computation.
argc, argv are required only in the C language binding,
where they are the main program's arguments.

MPI_FINALIZE()
Shut down a computation.

MPI_COMM_SIZE(comm, size)
Determine the number of processes in a computation.

IN	comm	communicator (handle)
OUT	size	number of processes in the group of comm (integer)

MPI_COMM_RANK(comm, pid)
Determine the identifier of the current process.

IN	comm	communicator (handle)
OUT	pid	process id in the group of comm (integer)

MPI_SEND(buf, count, datatype, dest, tag, comm)
Send a message.

IN	buf	address of send buffer (choice)
IN	count	number of elements to send (integer ≥ 0)
IN	datatype	datatype of send buffer elements (handle)
IN	dest	process id of destination process (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)

MPI_RECV(buf, count, datatype, source, tag, comm, status)
Receive a message.

OUT	buf	address of receive buffer (choice)
IN	count	size of receive buffer, in elements (integer ≥ 0)
IN	datatype	datatype of receive buffer elements (handle)
IN	source	process id of source process, or **MPI_ANY_SOURCE** (integer)
IN	tag	message tag, or **MPI_ANY_TAG** (integer)
IN	comm	communicator (handle)
OUT	status	status object (status)

Message Properties

1. MPI messages are one dimensional array of items and are the first argument of the send (MPI_Send) and receive (MPI_Recv) functions.

2. Argument to indicate where the array starts, arguments that indicate the number of elements in the array (count) and the type of each element (datatype) are also passed to the MPI functions.

3.The tag and comm arguments are used to differentiate multiple messages originating from the same processor.

4. The status argurment in the receive function stores information about the source, size, and tag of the message. This is useful in cases where the receive is allowed to receive a set of possible sources.

An Parallel Pseudo-Program Using the MPI Library

```
program main
begin
MPI_INIT()                                   //Initiate computation
MPI_COMM_SIZE(MPI_COMM_WORLD, count)//Find # of processes
MPI_COMM_RANK(MPI_COMM_WORLD, myid) //Find my id
print("I am", myid, "of", count)    //Print message
MPI_FINALIZE()                      //Shut down
end
```

Introduction to MPI

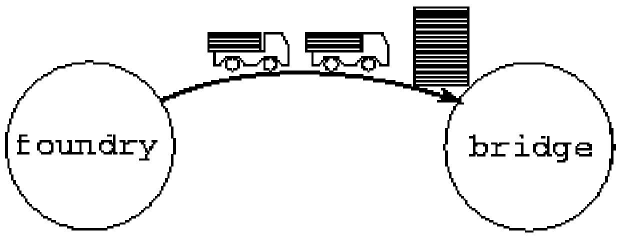
1. If the program on the previous slide is executed by four processes, we will obtain something like the following output.

2. The order in which the output appears is not defined; however, we assume here that the output from individual print statements is not interleaved.

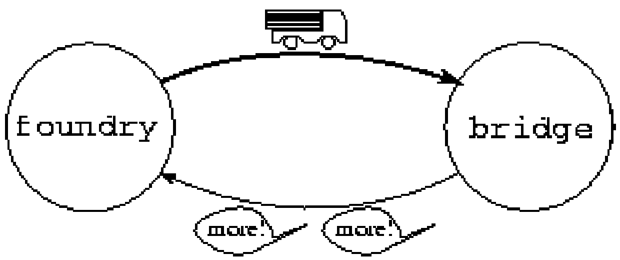
```
I am 1 of 4
I am 3 of 4
I am 0 of 4
I am 2 of 4
```

Foundry - Bridge Process

(a)



(b)



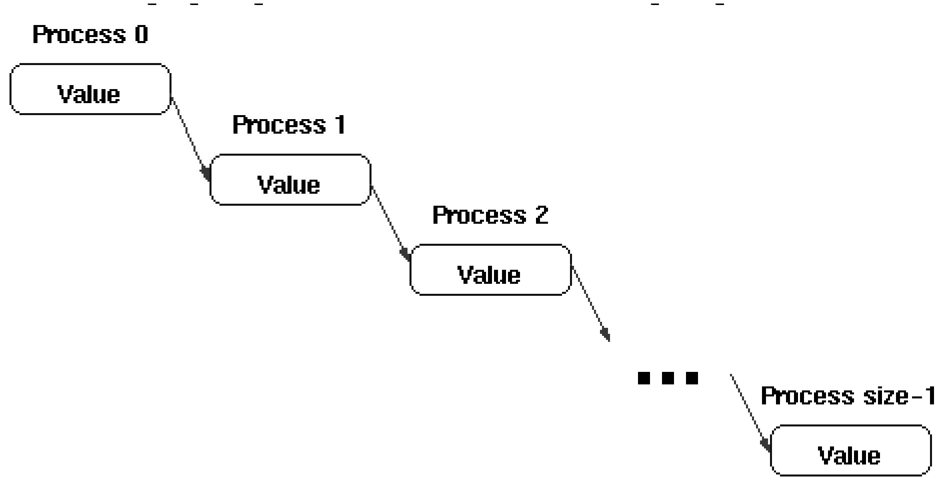
Introduction to MPI

```
program main
begin
  MPI_INIT()                                Initialize
  MPI_COMM_SIZE(MPI_COMM_WORLD, count)
  if count != 2 then exit                   Must be just 2 processes
  MPI_COMM_RANK(MPI_COMM_WORLD, myid)
  if myid = 0 then                          I am process 0:
    foundry(100)                              Execute foundry
  else                                      I am process 1:
    bridge()                                  Execute bridge
  endif
  MPI_FINALIZE()                            Shut down
end

procedure foundry(numgirders)               Code for process 0
begin
  for i = 1 to numgirders                   Send messages
    MPI_SEND(i, 1, MPI_INT, 1, 0, MPI_COMM_WORLD)
  endfor
  i = -1                                    Send shutdown message
  MPI_SEND(i, 1, MPI_INT, 1, 0, MPI_COMM_WORLD)
end

procedure bridge                            Code for process 1
begin
  MPI_RECV(msg, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, status)
  while msg != -1 do                        Receive messages
    use_girder(msg)                         Use message
    MPI_RECV(msg, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, status)
  enddo
end
```

Program 8.1 : MPI implementation of bridge construction problem. This program is designed to be executed by two processes.

Ring Communication

Write a program that takes data from process zero and sends it to all of the other processes by sending it in a ring. That is, process i should receive the data and send it to process i-

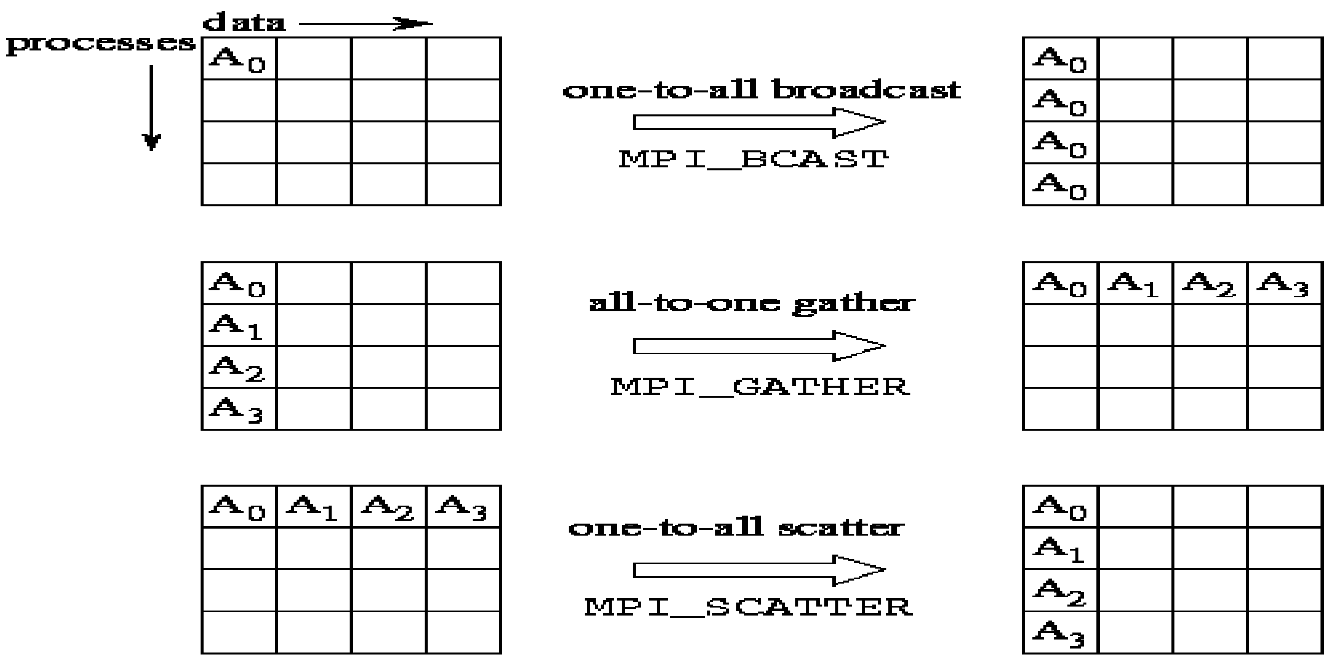


Assume that the data consists of a single integer. Process zero reads the data from the user.

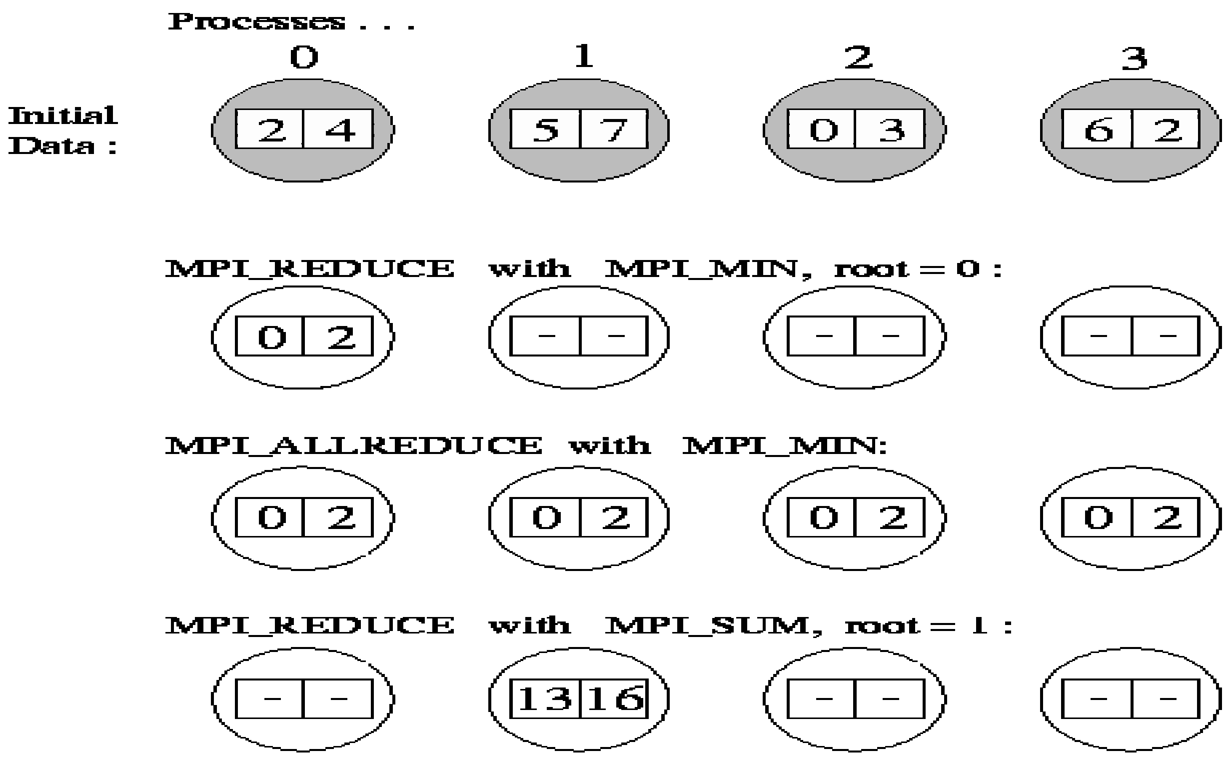
```
#include <stdio.h>
#include "mpi.h"
int main( argc, argv )
int argc;
char **argv;
{
   int rank, value, size;
   MPI_Status status;
   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   MPI_Comm_size( MPI_COMM_WORLD, &size );
```

```
do {
      if (rank == 0) {
         scanf( "%d", &value );
         MPI_Send( &value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD
);
      }
      else {
         MPI_Recv( &value, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD,
                 &status );
         if (rank < size - 1)
            MPI_Send( &value, 1, MPI_INT, rank + 1, 0,
MPI_COMM_WORLD );
      }
      printf( "Process %d got %d\n", rank, value );
   } while (value >= 0);

   MPI_Finalize( );
   return 0;
}
```

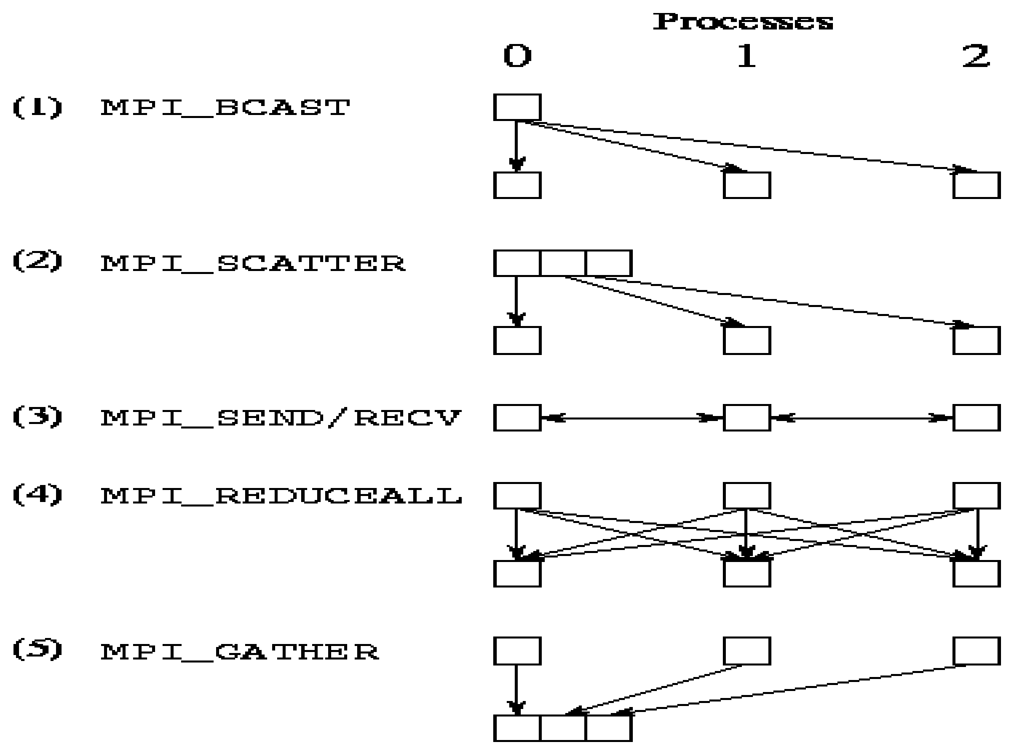
Global Communication Operations



Global Communication Operations



Global Communication Operations



Introduction to MPI

```
MPI_BARRIER(comm)
Global synchronization.
  IN     comm      communicator (handle)

MPI_BCAST(inbuf, incnt, intype, root, comm)
Broadcast data from root to all processes.
  INOUT inbuf     address of input buffer, or output buffer at root (choice)
  IN     incnt     number of elements in input buffer (integer)
  IN     intype    datatype of input buffer elements (handle)
  IN     root      process id of root process (integer)
  IN     comm      communicator (handle)

MPI_GATHER(inbuf, incnt, intype, outbuf, outcnt, outtype,
           root, comm)
MPI_SCATTER(inbuf, incnt, intype, outbuf, outcnt, outtype,
           root, comm)
Collective data movement functions.
  IN     inbuf     address of input buffer (choice)
  IN     incnt     number of elements sent to each (integer)
  IN     intype    datatype of input buffer elements (handle)
  OUT    outbuf    address of output buffer (choice)
  IN     outcnt    number of elements received from each (integer)
  IN     outtype   datatype of output buffer elements (handle)
  IN     root      process id of root process (integer)
  IN     comm      communicator (handle)

MPI_REDUCE(inbuf, outbuf, count, type, op, root, comm)
MPI_ALLREDUCE(inbuf, outbuf, count, type, op, comm)
Collective reduction functions.
  IN     inbuf     address of input buffer (choice)
  OUT    outbuf    address of output buffer (choice)
  IN     count     number of elements in input buffer (integer)
  IN     type      datatype of input buffer elements (handle)
  IN     op        operation; see text for list (handle)
  IN     root      process id of root process (integer)
  IN     comm      communicator (handle)
```

MPI Program for Parallel Implementation of Jacobi iteration for approximating the solution to a linear system of equations.

We solve the Laplace equation in two dimensions with finite differences. Any numerical analysis text will show that iterating

```
while (not converged) {
 for (i,j)
   xnew[i][j] = (x[i+1][j] + x[i-1][j] + x[i][j+1] + x[i][j-1])/4;
 for (i,j)
   x[i][j] = xnew[i][j];
 }
```

will compute an approximation for the solution of Laplace's equation.

Replacement of xnew with the average of the values around it is applied only in the interior; the boundary values are left fixed. In practice, this means that if the mesh is n by n, then the values

x[0][j]
x[n-1][j]
x[i][0]
x[i][n-1]

are left unchanged. These refer to the complete mesh; you'll have to figure out what to do with for the decomposed data structures (xlocal).

Because the values are replaced by averaging around them, these techniques are called relaxation methods.

We wish to compute this approximation in parallel. Write an MPI program to apply this approximation.

For convergence testing, compute

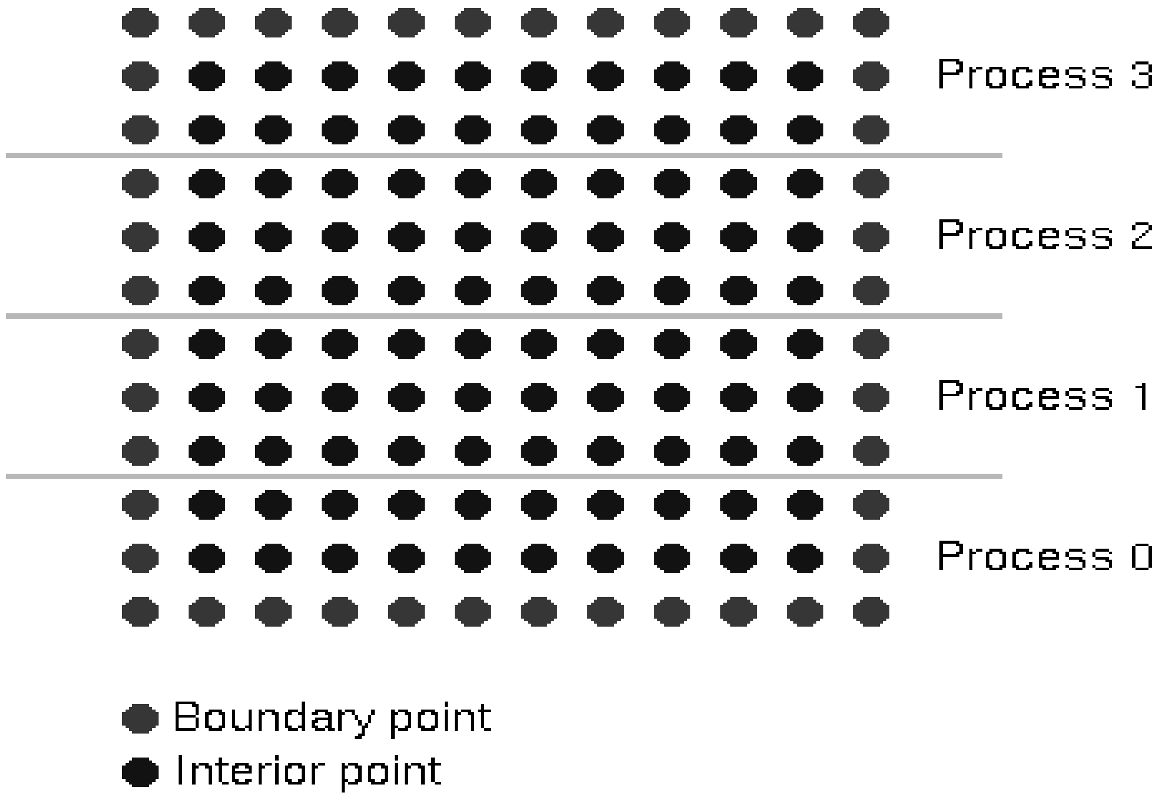
```
diffnorm = 0;
for (i,j)
   diffnorm += (xnew[i][j] - x[i][j]) * (xnew[i][j] - x[i][j]);
diffnorm = sqrt(diffnorm);
```

Use MPI_Allreduce for this. (Why not use MPI_Reduce?)

Process zero will write out the value of diffnorm and the iteration count at each iteration. When diffnorm is less that 1.0e-2, consider the iteration converged. Also, if you reach 100 iterations, exit the loop.

For simplicity, consider a 12 x 12 mesh on 4 processors.

The boundary values are -1 on the top and bottom, and the rank of the process on the side. The interior points have the same value as the rank of the process.


Process 3
Process 2
Process 1
Process 0
Boundary point
Interior point

This is shown below:

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
 3  3  3  3  3  3  3  3  3  3  3  3
 3  3  3  3  3  3  3  3  3  3  3  3
 2  2  2  2  2  2  2  2  2  2  2  2
 2  2  2  2  2  2  2  2  2  2  2  2
 2  2  2  2  2  2  2  2  2  2  2  2
 1  1  1  1  1  1  1  1  1  1  1  1
 1  1  1  1  1  1  1  1  1  1  1  1
 1  1  1  1  1  1  1  1  1  1  1  1
 0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"
/* This example handles a 12 x 12 mesh, on 4 processors only. */
#define maxn 12
int main( argc, argv )
int argc;
char **argv;
{
   int       rank, value, size, errcnt, toterr, i, j, itcnt;
   int       i_first, i_last;
   MPI_Status status;
   double    diffnorm, gdiffnorm;
   double    xlocal[(12/4)+2][12];
   double    xnew[(12/3)+2][12];

   MPI_Init( &argc, &argv );

   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
```

```
MPI_Comm_size( MPI_COMM_WORLD, &size );

  if (size != 4) MPI_Abort( MPI_COMM_WORLD, 1 );

  /* xlocal[][0] is lower ghostpoints, xlocal[][maxn+2] is upper */

  /* Note that top and bottom processes have one less row of interior
     points */
  i_first = 1;
  i_last  = maxn/size;
  if (rank == 0)        i_first++;
  if (rank == size - 1) i_last--;

  /* Fill the data as specified */
  for (i=1; i<=maxn/size; i++)
     for (j=0; j<maxn; j++)
         xlocal[i][j] = rank;
  for (j=0; j<maxn; j++) {
     xlocal[i_first-1][j] = -1;
     xlocal[i_last+1][j] = -1;
  }
```

```
itcnt = 0;
  do {
    /* Send up unless I'm at the top, then receive from below */
    /* Note the use of xlocal[i] for &xlocal[i][0] */
    if (rank < size - 1)
      MPI_Send( xlocal[maxn/size], maxn, MPI_DOUBLE, rank + 1, 0,
               MPI_COMM_WORLD );
    if (rank > 0)
      MPI_Recv( xlocal[0], maxn, MPI_DOUBLE, rank - 1, 0,
               MPI_COMM_WORLD, &status );
    /* Send down unless I'm at the bottom */
    if (rank > 0)
      MPI_Send( xlocal[1], maxn, MPI_DOUBLE, rank - 1, 1,
               MPI_COMM_WORLD );
    if (rank < size - 1)
      MPI_Recv( xlocal[maxn/size+1], maxn, MPI_DOUBLE, rank + 1, 1,
               MPI_COMM_WORLD, &status );
```

```
/* Compute new values (but not on boundary) */
    itcnt ++;
    diffnorm = 0.0;
    for (i=i_first; i<=i_last; i++)
       for (j=1; j<maxn-1; j++) {
          xnew[i][j] = (xlocal[i][j+1] + xlocal[i][j-1] +
                   xlocal[i+1][j] + xlocal[i-1][j]) / 4.0;
          diffnorm += (xnew[i][j] - xlocal[i][j]) *
                  (xnew[i][j] - xlocal[i][j]);
       }
    /* Only transfer the interior points */
    for (i=i_first; i<=i_last; i++)
       for (j=1; j<maxn-1; j++)
          xlocal[i][j] = xnew[i][j];
```

```
MPI_Allreduce( &diffnorm, &gdiffnorm, 1, MPI_DOUBLE, MPI_SUM,
               MPI_COMM_WORLD );
     gdiffnorm = sqrt( gdiffnorm );
     if (rank == 0) printf( "At iteration %d, diff is %e\n", itcnt,
                    gdiffnorm );
   } while (gdiffnorm > 1.0e-2 && itcnt < 100);

   MPI_Finalize( );
   return 0;
}
```

Asynchronous Communication Operations

MPI_IPROBE(source, tag, comm, flag, status)
Poll for a pending message.

IN	source	id of source process, or MPI_ANY_SOURCE (integer)
IN	tag	message tag, or MPI_ANY_TAG (integer)
IN	comm	communicator (handle)
OUT	flag	(logical/Boolean)
OUT	status	status object (status)

MPI_PROBE(source, tag, comm, status)
Return when message is pending.

IN	source	id of source process, or MPI_ANY_SOURCE (integer)
IN	tag	message tag, or MPI_ANY_TAG (integer)
IN	comm	communicator (handle)
OUT	status	status object (status)

MPI_GET_COUNT(status, datatype, count)
Determine size of a message.

IN	status	status variable from receive (status)
IN	datatype	datatype of receive buffer elements (handle)
OUT	count	number of data elements in message (integer)

Creating Communication Groups

MPI_COMM_DUP(comm, newcomm)
Create new communicator: same group, new context.

IN	comm	communicator (handle)
OUT	newcomm	communicator (handle)

MPI_COMM_SPLIT(comm, color, key, newcomm)
Partition group into disjoint subgroups.

IN	comm	communicator (handle)
IN	color	subgroup control (integer)
IN	key	process id control (integer)
OUT	newcomm	communicator (handle)

MPI_INTERCOMM_CREATE(comm, leader, peer, rleader, tag, inter)
Create an intercommunicator.

IN	comm	local intracommunicator (handle)
IN	leader	local leader (integer)
IN	peer	peer intracommunicator (handle)
IN	rleader	process id of remote leader in peer (integer)
IN	tag	tag for communicator set up (integer)
OUT	inter	new intercommunicator (handle)

MPI_COMM_FREE(comm)
Destroy a communicator.

IN	comm	communicator (handle)

Communication Groups

`A call of the form`

`MPI_COMM_SPLIT(comm, color, key, newcomm)` **creates one or more new communicators.**

It must be executed by each process in the process group associated with `comm.`

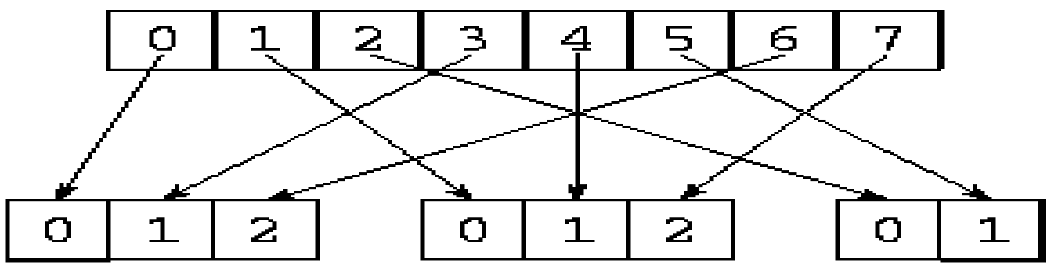
`A new communicator is created for each unique value of color other than the defined constant MPI_UNDEFINED.`

Each new communicator comprises those processes that specified its value of color in the MPI_COMM_SPLIT call. These processes are assigned identifiers within the new communicator starting from zero, with order determined by the value of key or, in the event of ties, by the identifier in the old communicator. Thus, a call of the form MPI_COMM_SPLIT(comm, 0, 0, newcomm) in which all processes specify the same color and key, is equivalent to a call MPI_COMM_DUP(comm, newcomm)

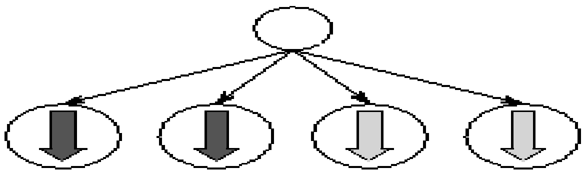
The following code creates three new communicators if `comm contains at least three processes.`

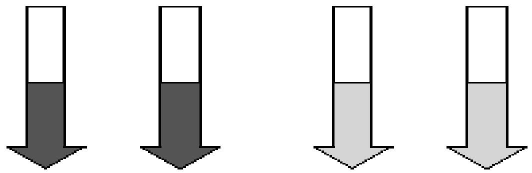
```
MPI_Comm comm, newcomm;
int myid, color;
MPI_Comm_rank(comm, &myid);
color = myid%3;
MPI_Comm_split(comm, color, myid, &newcomm);
```

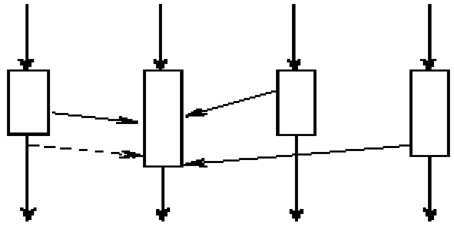
For example, if `comm contains eight processes, then processes 0, 3, and 6 form a new communicator of size three, as do processes 1, 4, and 7, while processes 2 and 5 form a new communicator of size two.`

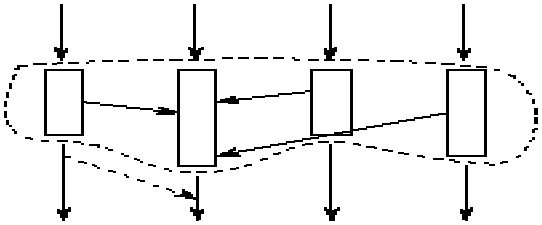


Task Model versus Process Model

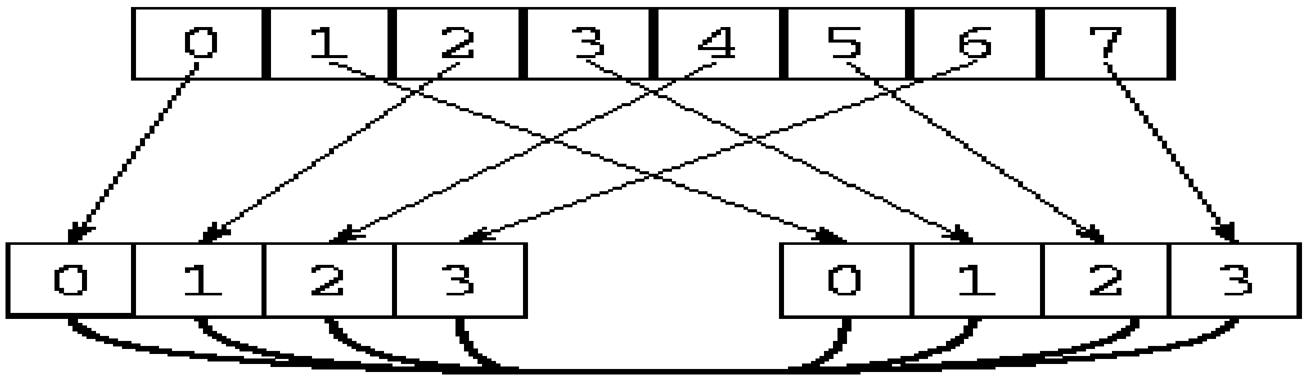








Communication Pattern for Program on Next Slide



```
      integer comm, intercomm, ierr, status(MPI_STATUS_SIZE)
c     For simplicity, we require an even number of processes
      call MPI_COMM_SIZE(MPI_COMM_WORLD, count, ierr)
      if(mod(count,2) .ne. 0) stop
c     Split processes into two groups: odd and even numbered
      call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
      call MPI_COMM_SPLIT(MPI_COMM_WORLD, mod(myid,2), myid,
     $                    comm, ierr)
c     Determine process id in new group
      call MPI_COMM_RANK(comm, newid, ierr)
      if(mod(myid,2) .eq. 0) then
c        Group 0: create intercommunicator and send message
c        Arguments: 0=local leader; 1=remote leader; 99=tag
         call MPI_INTERCOMM_CREATE(comm, 0, MPI_COMM_WORLD, 1, 99,
     $                    intercomm, ierr)
         call MPI_SEND(msg, 1, type, newid, 0, intercomm, ierr)
      else
c        Group 1: create intercommunicator and receive message
c        Note that remote leader has id 0 in MPI_COMM_WORLD
         call MPI_INTERCOMM_CREATE(comm, 0, MPI_COMM_WORLD, 0, 99,
     $                    intercomm, ierr)
         call MPI_RECV(msg, 1, type, newid, 0, intercomm,
     $                    status, ierr)
      endif
c     Free communicators created during this operation
      call MPI_COMM_FREE(intercomm, ierr)
      call MPI_COMM_FREE(comm, ierr)
```

Program 8.7 : An MPI program illustrating creation and use of an intercommunicator.

MPI Data Type Creation Operations

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)
Construct datatype from contiguous elements.

IN	count	number of elements (integer ≥ 0)
IN	oldtype	input datatype (handle)
OUT	newtype	output datatype (handle)

MPI_TYPE_VECTOR(count, blocklen, stride, oldtype, newtype)
Construct datatype from blocks separated by stride.

IN	count	number of elements (integer ≥ 0)
IN	blocklen	elements in a block (integer ≥ 0)
IN	stride	elements between start of each block (integer)
IN	oldtype	input datatype (handle)
OUT	newtype	output datatype (handle)

MPI_TYPE_INDEXED(count, blocklens, indices, oldtype, newtype)
Construct datatype with variable indices and sizes.

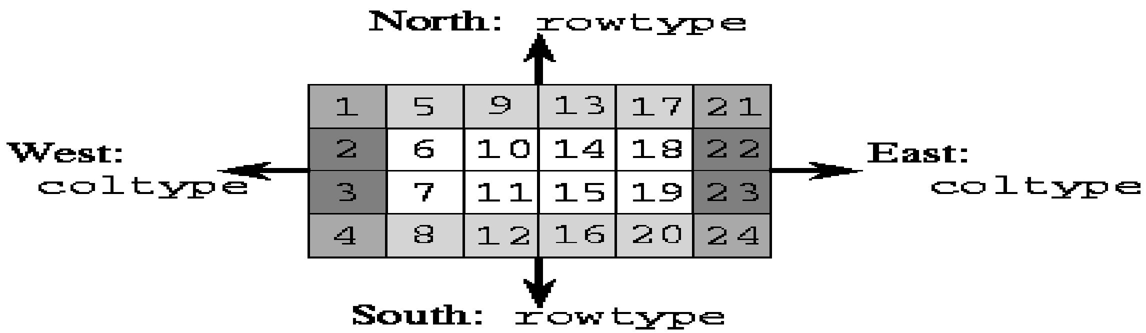
IN	count	number of blocks (integer ≥ 0)
IN	blocklens	elements in each block (array of integer ≥ 0)
IN	indices	displacements for each block (array of integer)
IN	oldtype	input datatype (handle)
OUT	newtype	output datatype (handle)

MPI_TYPE_COMMIT(type)
Commit datatype so that it can be used in communication.

INOUT	type	datatype to be committed (handle)

MPI_TYPE_FREE(type)
Free a derived datatype.

INOUT	type	datatype to be freed (handle)


North: rowtype
West: coltype
East: coltype
South: rowtype
1 5 9 13 17 21
2 6 10 14 18 22
3 7 11 15 19 23
4 8 12 16 20 24

```
      integer coltype, rowtype, comm, ierr
C   The derived type coltype is 4 contiguous reals.
      call MPI_TYPE_CONTIGUOUS(4, MPI_REAL, coltype, ierr)
      call MPI_TYPE_COMMIT(coltype, ierr)
C   The derived type rowtype is 6 reals, located 4 apart.
      call MPI_TYPE_VECTOR(6, 1, 4, MPI_REAL, rowtype, ierr)
      call MPI_TYPE_COMMIT(rowtype, ierr)
      ...
      call MPI_SEND(array(1,1), 1, coltype, west, 0, comm, ierr)
      call MPI_SEND(array(1,6), 1, coltype, east, 0, comm, ierr)
      call MPI_SEND(array(1,1), 1, rowtype, north, 0, comm, ierr)
      call MPI_SEND(array(4,1), 1, rowtype, south, 0, comm, ierr)
      ...
      call MPI_TYPE_FREE(rowtype, ierr)
      call MPI_TYPE_FREE(coltype, ierr)
```

Program 8.8 : Using derived types to communicate a finite difference stencil. The variables west, east, north, and south refer to the process's neighbors.