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. An "MPI" program is a set of processes, each running in a separate address space and usually on a different processor.

4. Processes communication by sending messages. There are multiple message modes.

5. The processors accessible to an MPI program are normally confined to a single domain or a single common file system.

6. Each process in an MPI program belongs to one or more "communication groups."

7. Each processor has a position, called a rank, in the communication group in which it was originally initiated. A processes rank is a unique identifier for the process in its communication group. Messages are addressed to a processor "rank."

8. Each processor executes the same program using local processor id to determine its behavior. Most MPI programs are structured with some form of central control implemented in the processor with rank "0."

9. MPI distributes the programs to the processors, loads them and initiates execution on each processor. MPI chooses processors upon which to load a program from a list of processors stored in a file, "machines."

10. Environment specification and execution initiation is external to MPI

Computer Science Department MPI machines file

```
aspen.cs.utexas.edu% pwd
/stage/public/share/src/mpi/share
aspen.cs.utexas.edu% more machines.LINUX
yeenoghu
zorkmid
asmodeus
beartrap
bladnoch
bowmore
bruichladdich
bunnahabhain
clynelish
crom
…..
…..
```

```
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)

Communication Model

1. **A communicator (A variable of type 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. The default communicator created by running an MPI program on the CS Linux systems would have the MPI processes on yeenoghu, zorkmid, asmodeus and beartrap as its membership.**

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.**

Communication Model – Continued

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 determined by executing the the MPI_Comm_size.

Message Properties

1. The data of an MPI message is a one dimensional array of items and is specified as the first argument of the send (MPI_Send) and receive (MPI_Recv) functions.

2. There is an argument to indicate where the array starts for a given member of a communicator. Arguments that specify the number of elements in the array (count) and the type of each element (data type) 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 argument 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
```

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

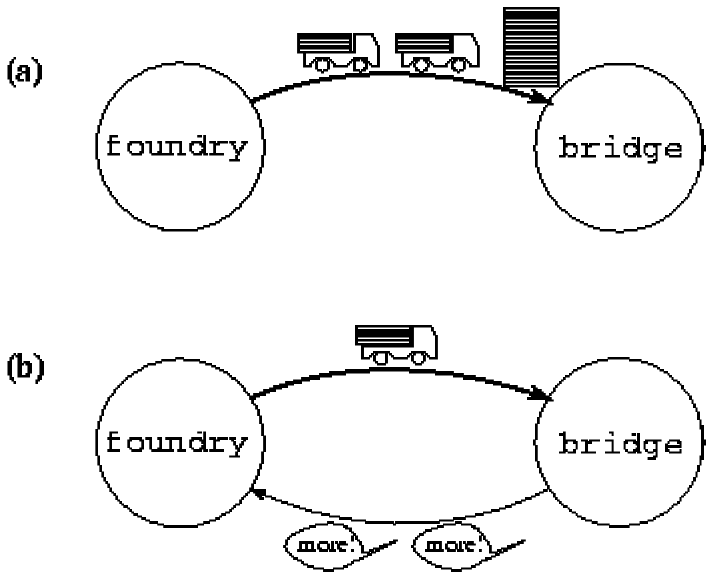
2. The order in which the output appears is not defined; however, we assume here that the output from individual print statements is not interleaved.

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

3. The output from an actual run was:

```
deerpark.cs.utexas.edu% mpirun -np 4
HelloWorld
Hello world from process 0 of 4
Hello world from process 2 of 4
Hello world from process 3 of 4
Hello world from process 1 of 4
```

Foundry - Bridge Process



```
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.

Master Acknowledgement Program

1. Each process finds out about the size of the process pool, its own rank within the pool, and the name of the processor it runs on.
2. Process of rank 0 becomes the master process.
3. The master process broadcasts the name of the processor it runs on to other processes.
4. Each process, including the master process constructs a greating message and sends it to the master process. The master process sends the message to itself.
5. The master process collects the messages and displays them on standard output.
6. This is the way to organise I/O, if only certain processes can write to the screen or to files.

Master Acknowlegement Program

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>
 #define TRUE 1
#define FALSE 0
#define MASTER_RANK 0
main(argc, argv)
int argc;char *argv[];
{   int count, pool_size, my_rank, my_name_length, i_am_the_master =
            FALSE;
char my_name[BUFSIZ], master_name[BUFSIZ],
send_buffer[BUFSIZ],recv_buffer[BUFSIZ];
MPI_Status status;
MPI_Init(&argc, &argv);   MPI_Comm_size(MPI_COMM_WORLD,
            &pool_size);   MPI_Comm_rank(MPI_COMM_WORLD,
            &my_rank);
```

```
MPI_Get_processor_name(my_name, &my_name_length);
if (my_rank == MASTER_RANK)
{     i_am_the_master = TRUE;
      strcpy (master_name, my_name);   }
      MPI_Bcast(master_name, BUFSIZ, MPI_CHAR, MASTER_RANK,
            MPI_COMM_WORLD);
      sprintf(send_buffer, "hello %s, greetings from %s, rank = %d",
            master_name, my_name, my_rank);
MPI_Send (send_buffer, strlen(send_buffer) + 1, MPI_CHAR,
            MASTER_RANK, 0, MPI_COMM_WORLD);
if (i_am_the_master)
{     for (count = 1;
 count <= pool_size; count++)
{       MPI_Recv (recv_buffer, BUFSIZ, MPI_CHAR, MPI_ANY_SOURCE,
            MPI_ANY_TAG, MPI_COMM_WORLD, &status);
       printf ("%s\n", recv_buffer);     }   }
MPI_Finalize();
}
```

This program in English

When you look at an MPI program and try to trace its logic, think of yourself as one of the processors.

And so, you begin execution and the first statement that you encounter is

MPI_Init(&argc, &argv);

What this statement tells you is that *you are not alone*. There are others like you, and all of you comprise a pool of MPI processes. How many there are in that pool altogether?

To find out you issue the command

MPI_Comm_size(MPI_COMM_WORLD, &pool_size);

which, translated into English means:

How many processes there are in the default communicator, which is guaranteed to encompass all processes in the pool, MPI_COMM_WORLD? Please put the answer in the variable pool_size.

When this function returns you know how many colleagues you have. But the next pressing question is: how can you distinguish yourself from the others? Are you all alike? Are you all indistinguishable? When processes are born, each process is born with a different number, much the same as each human is born with different DNA and different fingerprints.

That number is called a *rank number*, and if you are an MPI process you can find out what *your* rank number is by calling function:

MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

The English translation of this call is:

What is my rank number in the default communicator MPI_COMM_WORLD? Please put the answer in the variable my_rank.

A process such as yourself can belong to many communicators. You always belong to MPI_COMM_WORLD, but within the world you can have many sub-worlds, or, let's call it *states*. If you have multiple citizenships, you will also have multiple tax numbers, or multiple social security numbers, that would distinguish you from other citizens of those states. By the same token a process that belongs to many communicators may have different a different rank number in each of them, so when you ask about your rank number you must specify a communicator too.

OK, by now you know how many other processes there are in the pool, and what is your rank number within that pool. You can also find the name of the process*or* that you yourself run on.

You call function:

MPI_Get_processor_name(my_name, &my_name_length);

which translated to English means:

What is the name of the processor that I run on? Please put the name in the variable my_name and put the length of that name in my_name_length.

So far every process in the pool would have performed exactly the same operations. There has been no communication between you guys yet. But now you all check if your rank number is the same as a predefined MASTER_RANK number. Who defines what the MASTER_RANK number is? In this case it is the programmer, the God of MPI processes. But on some systems all processes may go through additional environmental enquiries and check for the existence of a host process or processes which can do I/O, and so on, and then jointly decide on which is going to be the MASTER.

Well, here the MASTER has been annointed by God.

Only one process will discover that he or she is the annointed one. That one process will place TRUE in the i_am_the_master variable. For all other processes that variable will remain FALSE. This one process will laboriously copy its name into the variable master_name. For all other processes that string will remain null.

But all other processes will know that they are not the master, and they will know who the master is, because by now they all know that their rank is *not* MASTER_RANK.

At this stage all processes that are not the master subject themselves to receiving a broadcast from the master. All processes, including yourself (regardless of whether you are the master or not), perform this operation at the same time, and all of them end up with the same message in the variable master_name. This message is the name of the processor the master process runs on. The name has been copied from the variable master_name of the master process and written on variables called master_name that belong to other processes. The MPI machine will have done all that.

This operation is accomplished by calling:

```
MPI_Bcast(master_name, BUFSIZ, MPI_CHAR, MASTER_RANK,
          MPI_COMM_WORLD);
```

In plain English the meaning of this call is as follows:

Copy BUFSIZ data items of type MPI_CHAR from a buffer called master_name that is managed by process whose rank is MASTER_RANK within the MPI_COMM_WORLD communicator, to which I must belong too, to my own buffer also called master_name.

At this stage whether you are a slave process or a master process you are very knowledgeable about your MPI_COMM_WORLD universe. And, if you are a slave process, you are prudent enough to prepare and send a congratulatory message to the master process.

And so first you write the message on your send_buffer:

```
sprintf(send_buffer, "hello %s, greetings from %s, rank = %d", master_name,
          my_name, my_rank);
```

And observe that you write this message even if you are the master. Well there is nothing wrong with congratulating yourself. Some people do it all the time. Having prepared the message you send it to the master process, and if you are the master process you send it to yourself, which is fine too. Some people seldom receive messages from anyone else.

Here is how you will have accomplished this task:

```
MPI_Send (send_buffer, strlen(send_buffer) + 1, MPI_CHAR,
          MASTER_RANK, 0, MPI_COMM_WORLD);
```

In plain English the meaning of this operation is as follows:

Send strlen(send_buffer) + 1 data items (don't forget about the terminating null character, for which function strlen does not account) of type MPI_CHAR, which have been deposited in send_buffer to a process whose rank is MASTER_RANK. Attach a tag 0 to that message (to distinguish it from other messages that the master process may receive from elsewhere, perhaps). The ranking and communication refer to the MPI_COMM_WORLD communicator.

If you are a slave process then this is about all that you are supposed to do in this program, so now you can relax and spin, or go home.

But if you are a master process you have to collect all those messages that have been sent to you and print them on standard output in the *receive* order.

How many messages are you going to receive, master? There will be pool_size messages sent to you from all processes including yourself. So you can just as well enter a for loop and receive all those pool_size messages, knowing, when you count the last one, that your job is done too.

To receive a message you do as follows:

```
MPI_Recv (recv_buffer, BUFSIZ, MPI_CHAR, MPI_ANY_SOURCE,
          MPI_ANY_TAG, MPI_COMM_WORLD, &status);
```

which in plain English means:

Let me receive up to BUFSIZ data items of type MPI_CHAR into my array recv_buffer from any source (MPI_ANY_SOURCE) and with any tag (MPI_ANY_TAG) within the MPI_COMM_WORLD. The status of the received message should be written on structure status.

It is possible to find out a lot about a message *before* you are going to receive it. You can find how long it is, where it comes from, what type are data items inside the message, and so on. But in this case the master process doesn't bother. The logic of the program is simple enough. God, i.e., the programmer, told the master process to receive pool_size messages, so receive them it shall. And it shall it print them on standard output as it receives them.

Once this point in the program is reached, all processes hit

MPI_Finalize;

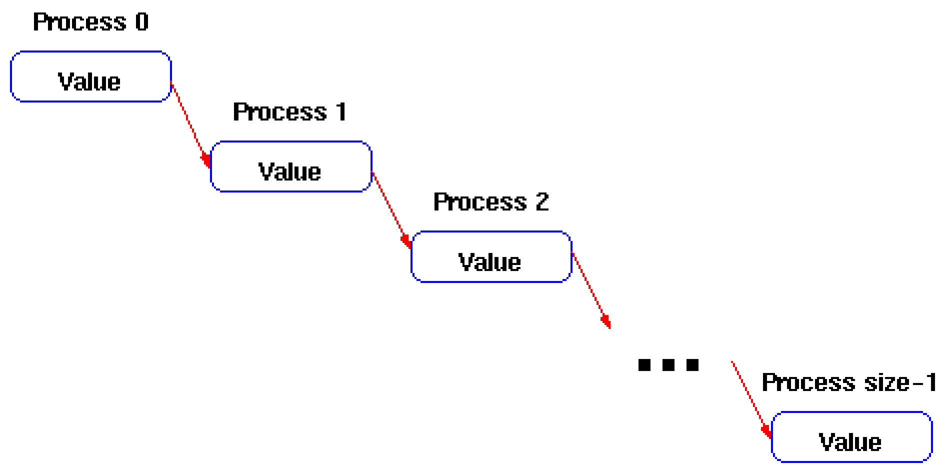
which is the end of the world for them.

If you want to look at more examples in this style, go to

http://beige.ucs.indiana.edu/B673/node120.html

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-



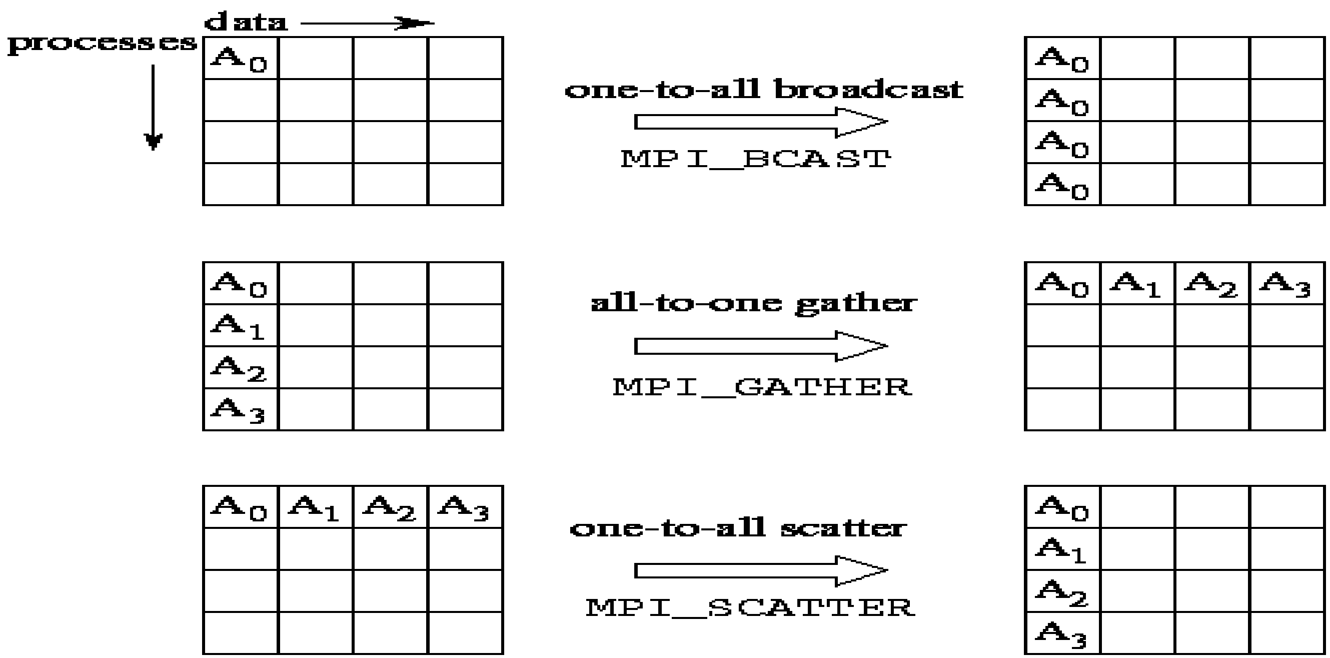
Assume that the data consists of a single integer. Process zero reads the data from the user.

```
#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 );
```

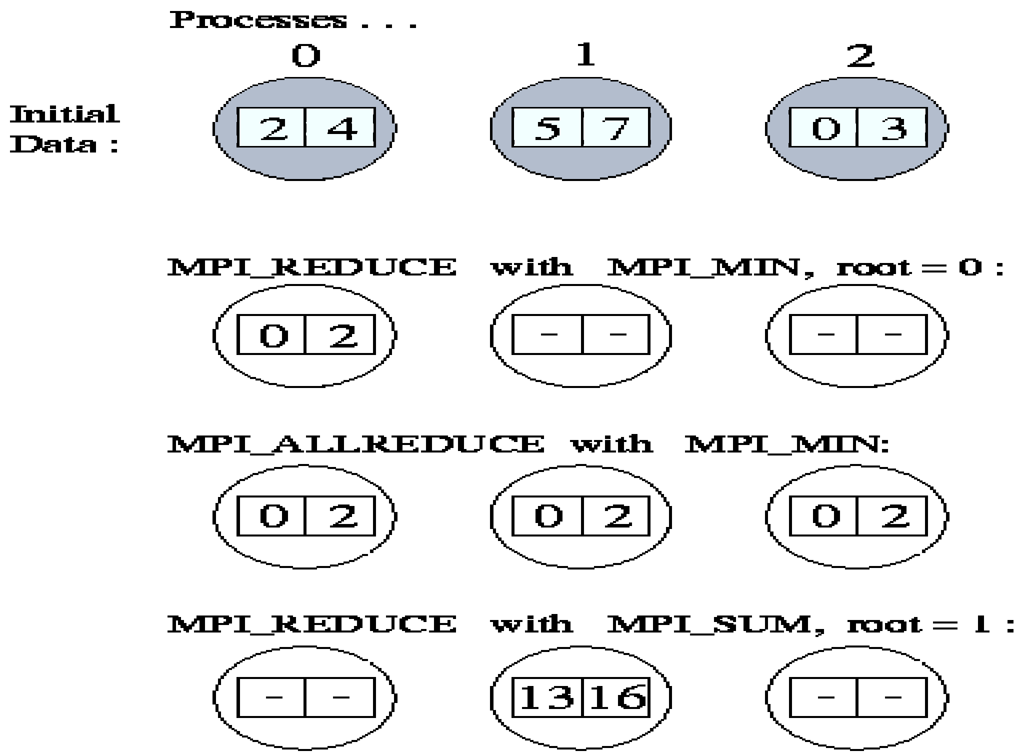
```
do {
      if (rank == 0) {
         scanf( "%d", &value );
         MPI_Send( &value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD
);
      }
      else {
         MPI_Recv( &value, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD,
               &status );
         if (rank < size - 1)
            MPI_Send(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

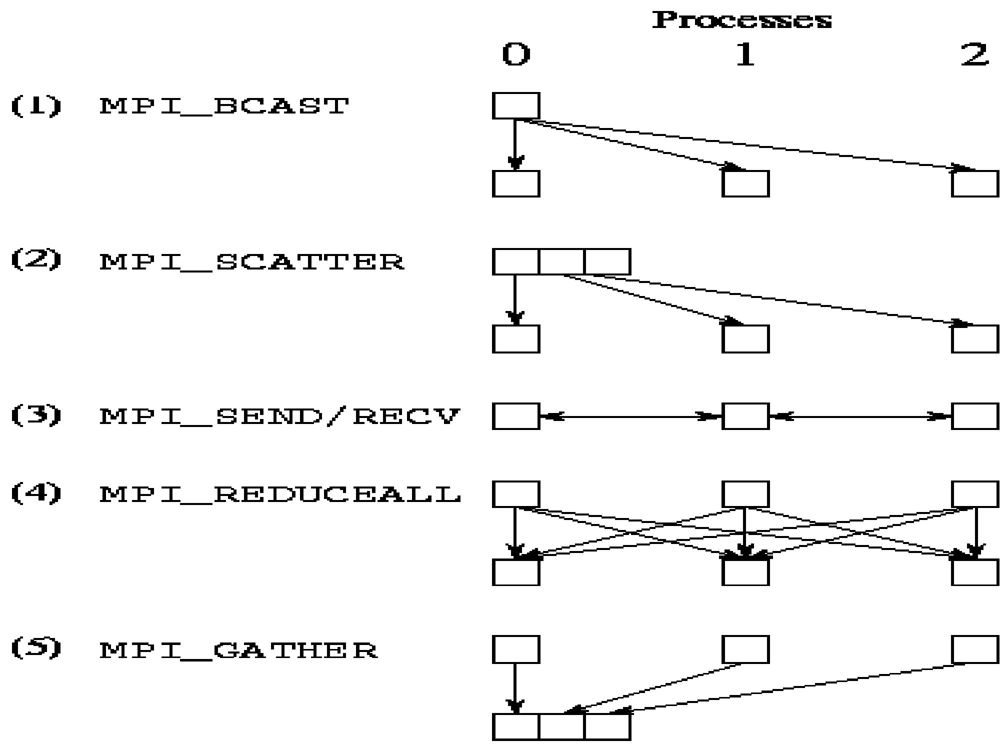
Global Communication Operations



Global Communication Operations



Global Communication Operations



```
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

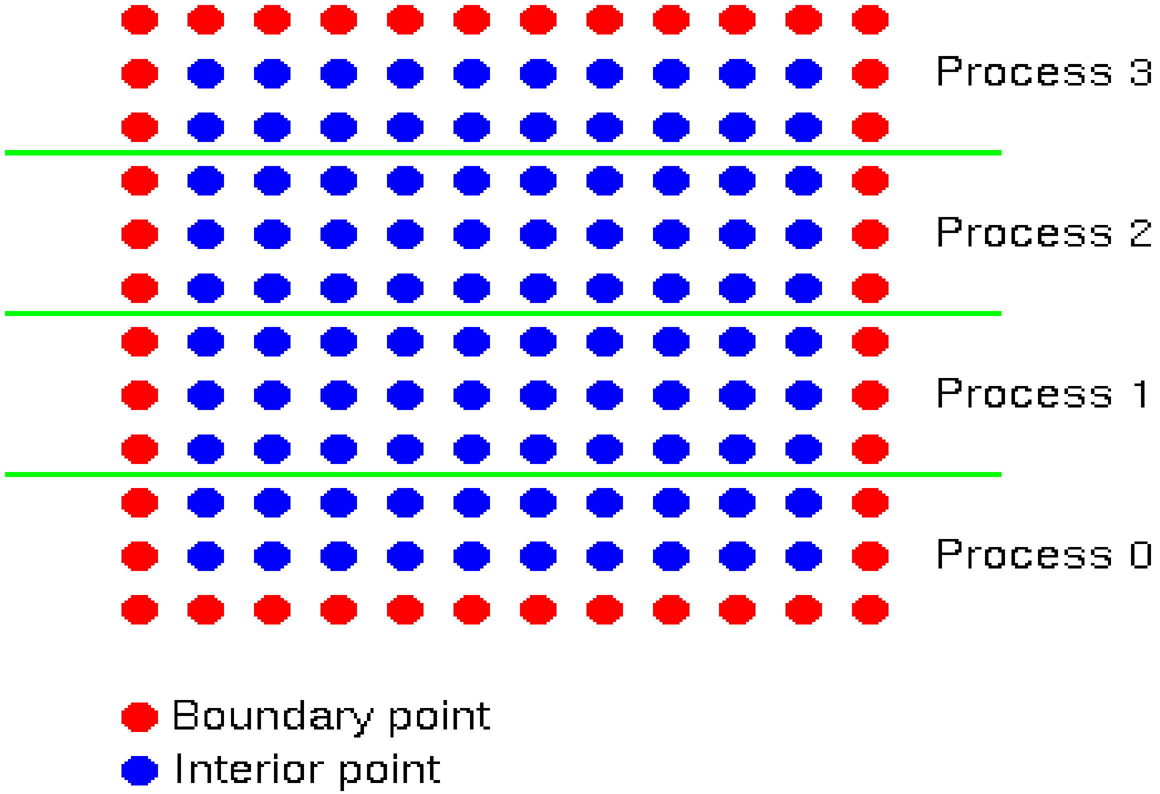
```
diffnorm = 0;
for (i,j)
    diffnorm += (xnew[i][j] - x[i][j]) * (xnew[i][j] - x[i][j]);
diffnorm = sqrt(diffnorm);
```

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.`

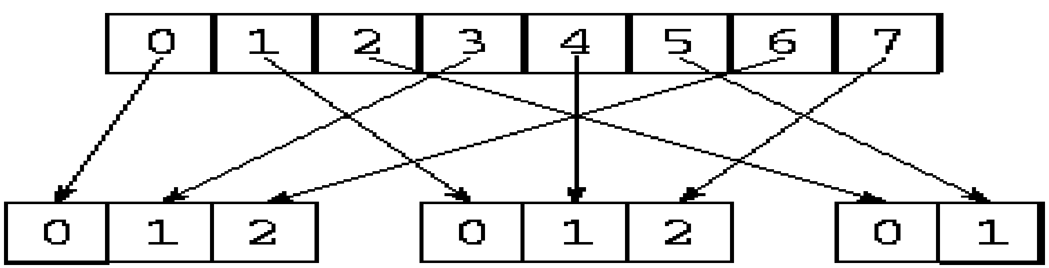
Communicators from Partitioning

- 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)`

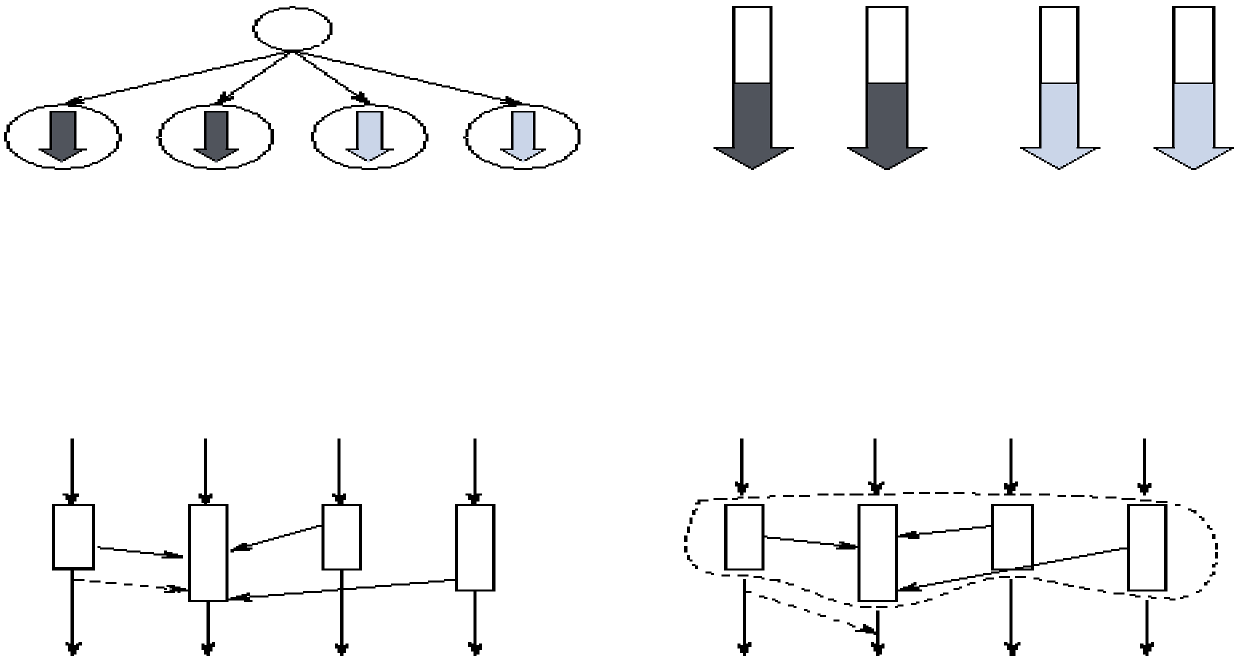
The following code creates three new communicators if `comm contains at least three processes.`

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

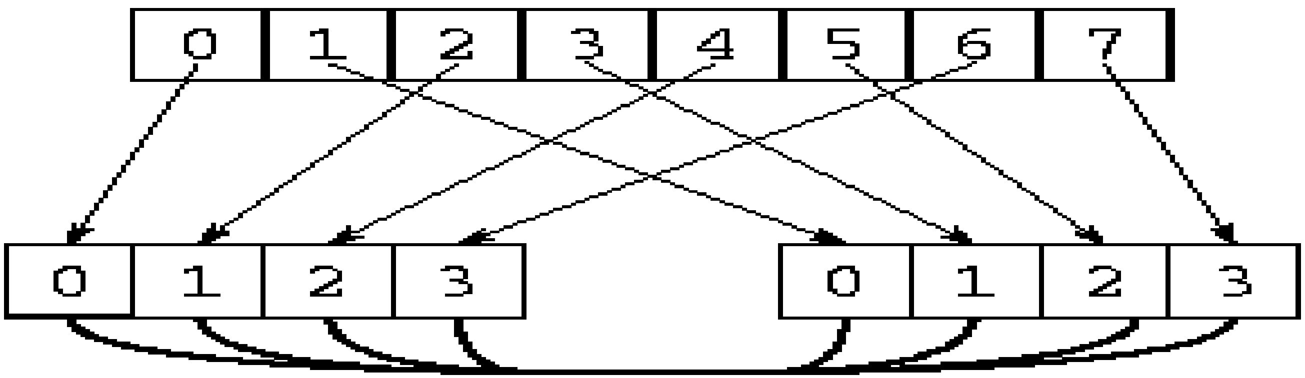
For example, if `comm contains eight processes, then processes 0, 3, and 6 form a new communicator of size three, as do processes 1, 4, and 7, while processes 2 and 5 form a new communicator of size two.`



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

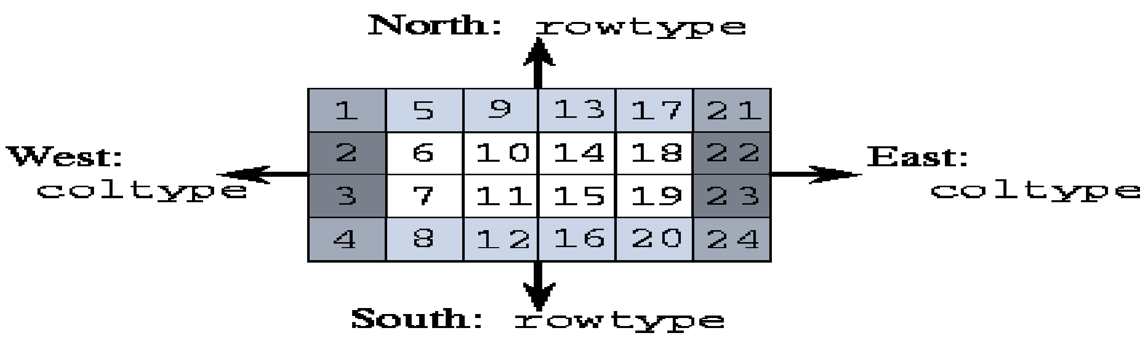
```
MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)
Construct datatype from contiguous elements.
  IN     count      number of elements (integer ≥0)
  IN     oldtype    input datatype (handle)
  OUT    newtype    output datatype (handle)

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.