MPI Course Overview

- Introduction and overview of message passing and MPI
- MPI program structure
- Messages
- Point-to-point communication
- Non-blocking communication
- Collective communication
- Miscellaneous (derived datatypes, MPI IO, topologies, MPI2, ...)
- Debugging and Profiling

MPI References

Books written by authors of the MPI standard:

- *Using MPI, Portable Parallel ...*, 2nd edition, Gropp et al, MIT Press, 1999
- *MPI - The Complete Reference, Volume 1, The MPI Core*, 2nd edition, Snir et al, MIT Press, 1999
- *Using MPI-2, Advanced Features of ...*, Gropp et al, MIT Press, 1999
- *MPI - The Complete Reference, Volume 2, The MPI Extensions*, Gropp et al, MIT Press, 1999

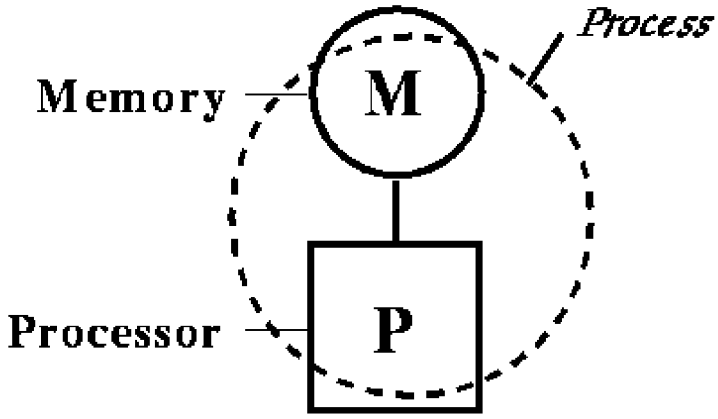
Two freely available MPI libraries that you can download and install:

- LAM - http://www.lam-mpi.org
- MPICH - http://www-unix.mcs.anl.gov/mpi/mpich
- OpenMPI - http://www.open-mpi.org (recommended)

Websites

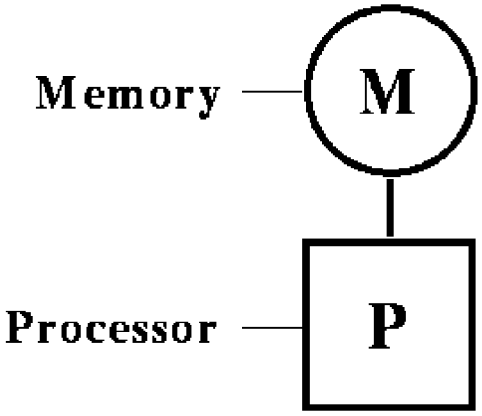
- http://www-unix.mcs.anl.gov/mpi
  - the main MPI web page
- http://www-unix.mcs.anl.gov/mpi/learning.html
  - pointers to teaching material and tutorials
- http://www-unix.mcs.anl.gov/mpi/tutorial/mpiexmpl
  - set of guided exercises
- http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html
  - MPI 1.1 standard in readable form with rationales and advice to users
- http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html
  - MPI 2.0 standard as an addition to the MPI 1.1 standard (less useful)

Sequential Programming Paradigm

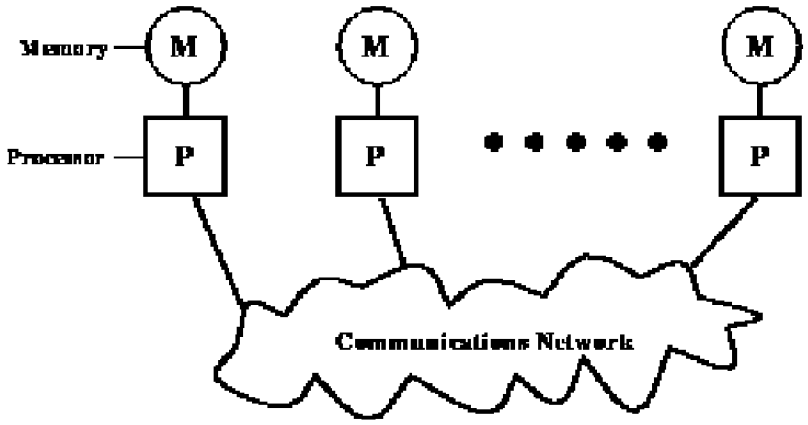


Basic Concepts

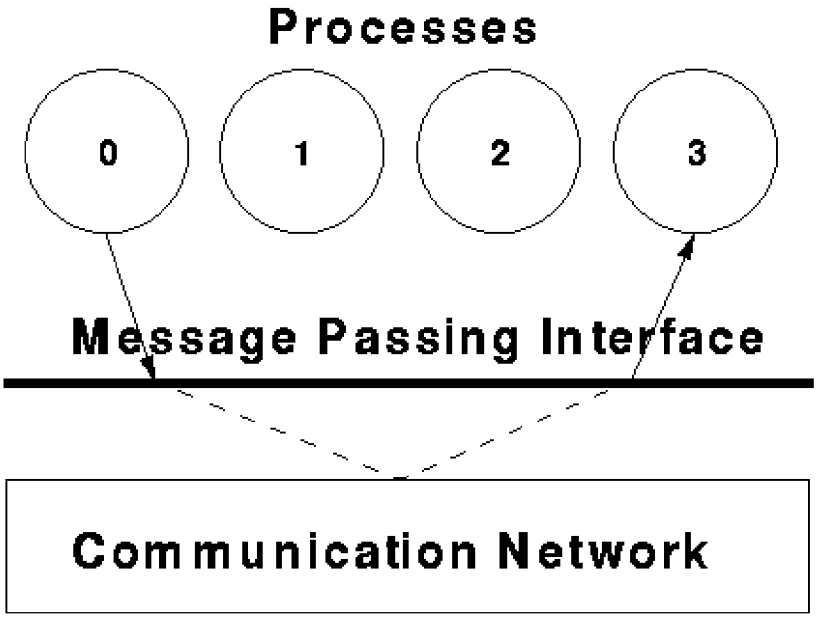
Sequential Computer Architecture Paradigm



Message-Passing Computer Architecture Paradigm



Message-Passing Programming Paradigm



Message-Passing Programming Paradigm (cont'd)

- A single program is run on each processor.
- All variables are private.
- Processes communicate via special subroutine calls - *MPI is just a library!*
- There is no "magic" parallelism.
- Written in a conventional sequential language, i.e. C or Fortran (no special compiler)

Exercise 0: Distributed Memory Message-Passing Concepts

```
 program simple_mp
 integer :: rank, a(0:1), b(0:1), c(5)
 character*64 message

     write(*,*) "Enter rank (0 or 1):"
     read(*,*) rank
! passing a message

     open(unit=7,file="message")
     if ( rank == 0 ) then
         write(7,*) "this is the message"
     else
         read(7,'(a)') message
         write(*,*) "I read ", message
     endif


 ! independent work, results not shared

      if ( rank == 0 ) then
          a(0) = 27
      else
          a(1) = 31
      endif
      write(*,*) " a = ", a

 ! getting rid of the "if"

     b(rank) = rank+2
     write(*,*) " b = ", b

 ! assigning a distributed length 10 vector
      do i = 1,5
          c(i) = i+5*rank
      enddo
      write(*,*) " c = ", c
end program simple_mp
```

What is SPMD?

- Single Program, Multiple Data
- Same program runs everywhere.
- Restriction on the general message-passing model.
- Some platforms only support SPMD parallel programs
- General message-passing model can be emulated.

Emulating General Message-Passing with SPMD: Fortran

```
 program
     if (process is to become a controller
process) then
         call CONTROLLER ( /* Arguments */ )
     else
         call WORKER ( /* Arguments */ )
     endif
 end
```

Emulating General Message Passing with SPMD: C

```
   main (int argc, char **argv)
   {
       if (process is to become a controller
  process)
       {
           Controller( /* Arguments */ );
       }
       else
       {
           Worker( /* Arguments */ );
       }
   }
```

Messages

- Messages are packets of data moving between processes.
- The message passing system has to be told the following information:
  - Sending process
  - Source location
  - Data type
  - Data length
  - Receiving process(es)
  - Destination location
  - Size of receive buffer(s)

Access

- A process needs to be connected to a message passing interface.
- A message passing system is similar to:
  - Mail box
  - Phone line
  - fax machine
  - etc.

Addressing

- Messages need to have addresses to be sent to.
- Addresses are similar to:
  - Mail address
  - Phone number
  - fax number
  - etc.

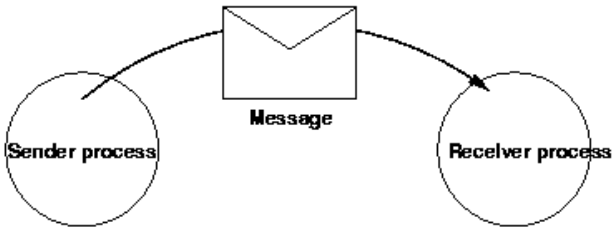
Reception

Receiving process must:

1. participate (cf. have a mailbox it checks, a phone it answers, ..)
2. have capacity to receive (have a big enough mailbox etc)

Point-to-Point Communication

- Simplest form of message passing.
- One process sends a message to another
- Both ends must actively participate
- Sending a point-to-point message requires specifying all the details of the message



Point-to-point communication types

Point-to-point communication concepts:

- Synchronous vs asynchronous
- Blocking vs non-blocking
- Buffer space, reliability, ...

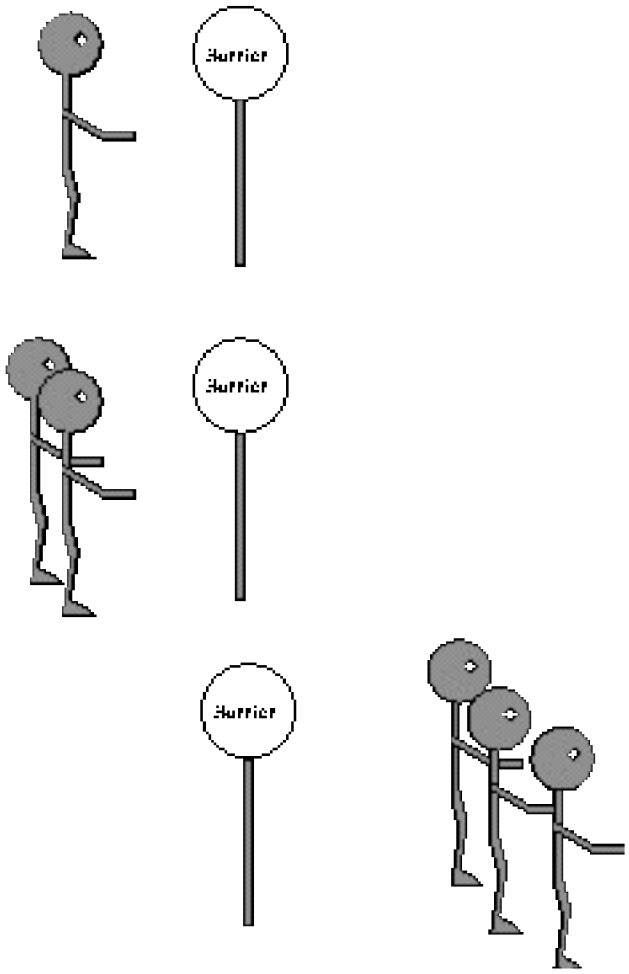
Leads to a myriad of different types of point-to-point communication calls.

Collective communications

- Collective communication routines are higher level routines involving several processes at a time (often all).
- Can be built out of point-to-point communications.

Collective Example: Barrier

Simplest collective communication example is a **barrier**
synchronises participating processes



Message Passing Interface (MPI) standard

MPI is a standard interface for message passing:

- Defined by MPI Forum - 40 vendor and academic/user organizations
- Provides **source-code portability across all systems**
- Allows efficient implementation.
- Provides high-level functionality.
- Supports heterogeneous parallel architectures.
- Evolving - MPI-2 is an addition to MPI-1.

MPI Programs

Header files

Should appear everywhere you call MPI procedures.

C:

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

Fortran:

```
include 'mpif.h'
```

Constants

The header files are full of constants that are used as arguments to MPI procedures.

Examples:

```
!   MPI types for Fortran programs
!
    integer MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION
    integer MPI_COMPLEX, MPI_DOUBLE_COMPLEX, MPI_LOGICAL
!
    parameter (MPI_COMPLEX=23, MPI_DOUBLE_COMPLEX=24, MPI_LOGICAL=25)
    parameter (MPI_REAL=26, MPI_DOUBLE_PRECISION=27, MPI_INTEGER=28)
```

MPI Function Format

C:

```
int error;
error = MPI_Xxxxx (parameter, ...);
```

Fortran:

```
integer ierror
call MPI_XXXXX (parameter, ..., ierror)
```

Generic:

```
MPI_XXXXX (parameter, ...)
```

Initialising MPI

C:

```
int MPI_Init (int *argc, char ***argv)
```

Fortran:

```
subroutine MPI_Init(ierror)
integer ierror
```

Must be the first MPI procedure called.

Handles

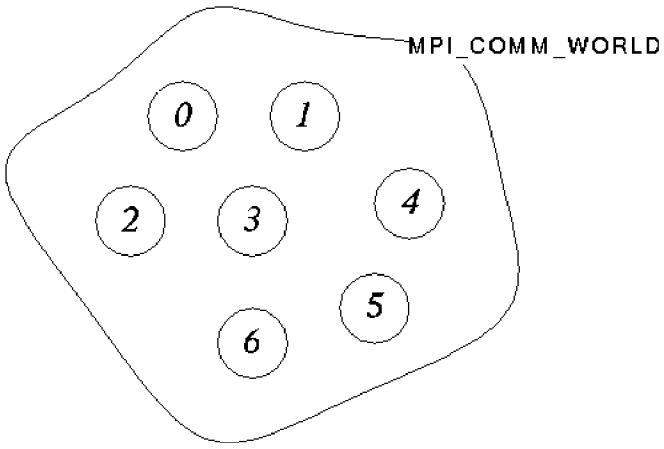
- MPI controls its own internal data structures
- MPI exposes 'handles' to allow programmers to refer to these
- C handles are of defined `typedefs`
- Fortran handles are `integers`.

Communicators

- "orthogonal message passing universes"
- human analogy: the mail system is one communicator and the phone system another
- every message "travels" in a communicator (every message passing call has a communicator argument)
- more than just groups of processes - "context"
- very useful for libraries (library messages dont interfere with library users messages)
- referred to by a "handle" (of type `MPI_Comm` in C).

MPI_COMM_WORLD communicator

- `MPI_COMM_WORLD` is the default communicator setup by `MPI_Init()`
- contains all processes
- for today, just use it whereever a communicator is required!
- `MPI_COMM_WORLD` is a handle (look in header file)



Rank

How do you identify different processes?

C:

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

Fortran:

```
subroutine MPI_Comm_rank(comm, rank, ierror)
integer comm, rank, ierror
```

Size

How many processes are contained within a communicator?

C:

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

Fortran:

```
subroutine MPI_Comm_size(comm, size, ierror)
integer comm, size, ierror
```

Exiting MPI

Must be the last MPI procedure called by each process.

C:

```
int MPI_Finalize();
```

Fortran:

```
subroutine MPI_Finalize(ierror)
integer ierror
```

To abort all processes of an MPI job:

C:

```
int MPI_Abort(MPI_Comm comm, int errcode);
```

Fortran:

```
subroutine MPI_Abort(comm, errcode, ierr)
integer comm, errcode, ierr
```

The Four Essentials

```
MPI_Init()
MPI_Comm_size()
MPI_Comm_rank()
 ...
MPI_Finalize()
```

Basic Fortran MPI Program

Fortran:

```
program firstmpi

  ! The most basic MPI Program

  include 'mpif.h'

  integer :: mpierror, mpisize, mpirank

  call MPI_Init(mpierror)
  call MPI_Comm_size(MPI_COMM_WORLD, mpisize, mpierror)
  call MPI_Comm_rank(MPI_COMM_WORLD, mpirank, mpierror)

  ! Do work here

  call MPI_Finalize(mpierror)

end program firstmpi
```

Basic C MPI Program

C:

```
#include <mpi.h>

void main(int argc, char *argv[])
{

/* The most basic MPI Program */

int mpierror, mpisize, mpirank;

mpierror=MPI_Init(&argc, &argv);
mpierror=MPI_Comm_size(MPI_COMM_WORLD, &mpisize);
mpierror=MPI_Comm_rank(MPI_COMM_WORLD, &mpirank);

/* Do work here */

mpierror=MPI_Finalize();

}
```

Using MPI (AC)

On the Altix AC (http://nf.apac.edu.au/facilities/userguide/ac):

The relevant modules to use MPI are loaded for you at login.

Compile:

```
> ifort foo.f -o foo.exe -lmpi
> icc foo.c -o foo.exe -lmpi
```

Execute:

```
> mpirun -np 4 ./foo.exe
```

or

```
> qsub -q express -lncpus=4,walltime=30:00,vmem=400mb -wd
mpirun ./a.out
cntrl-D
```

(Note that on AC you can use either `mpirun` or `prun` as these are both wrappers to the underlying `anumpirun` command.)

Exercise 1: Hello World - the minimal MPI program

1. Write a minimal MPI program which prints "*hello world*" from multiple processes. Compile and run it on 1, 2 and 4 procesors.
2. Add the writing processes rank as part of the output. (Also note the `-t` option for `mpirun`.)
3. Use **Fortran:**

```
integer mpi_log_fd=11
character*10 filenm
...
write(filenm,'("mpilog.",i2.2)') mpirank
open(unit=mpi_log_fd, file=filenm)
```

   or **C:**

```
FILE *mpi_log_fd;
char filenm[10];
...
sprintf(filenm, "mpilog.%2.2d" , mpirank);
mpi_log_fd = fopen(filenm,"w" );
```

   to open a log file for each process and write to it.
   (To use such log files for debugging, it may be necessary to use `flush()` after each `write/fprintf` -- dont use `flush()` except for debugging.)

4. What happens to IO before `MPI_Init()` or after `MPI_Finalize()` on different machines?

[Solution in `hello.f/f90/c`]

Messages

Message Datatypes

- A message contains a number of elements of some particular datatype.
- MPI datatypes:
  - Basic types.
  - Derived types.
- Derived types can be built up from basic types.
- C types are different from Fortran types.
- Each MPI call accepts messages of any datatype - datatype given as an extra argument

MPI Basic Datatypes - C

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double

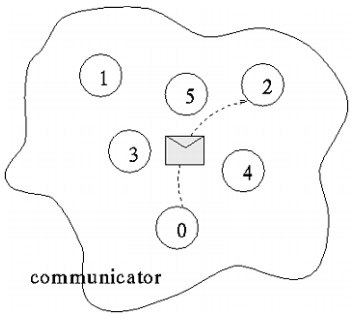
MPI Basic Datatypes - Fortran

MPI Datatypes	Fortran Datatypes
MPI_INTEGER	integer
MPI_REAL	real
MPI_DOUBLE_PRECISION	double precision
MPI_COMPLEX	complex
MPI_LOGICAL	logical
MPI_CHARACTER	character
MPI_BYTE	
MPI_PACKED	

Point-to-Point Communication

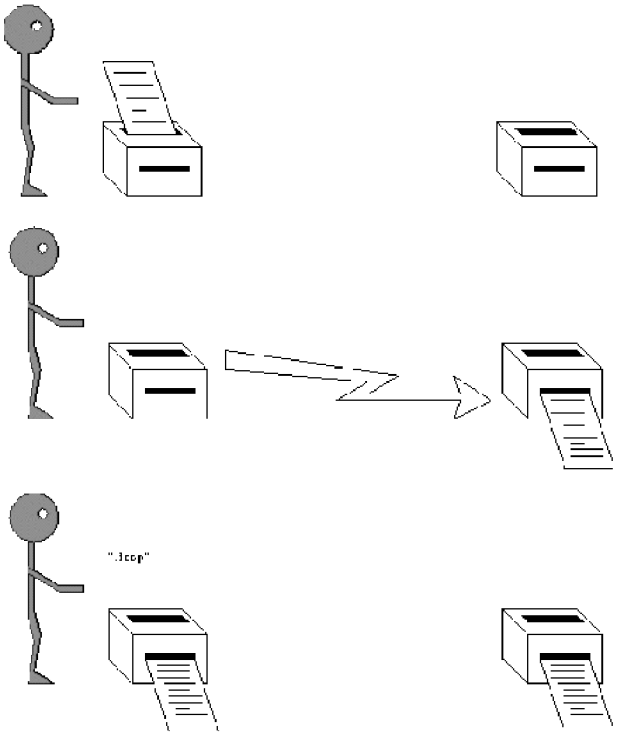
Point-to-Point Communication

- Communication between two processes.
- Source process sends message to destination process.
- Communication takes place within a communicator.
- Destination process is identified by its rank in the communicator.
- Various types of communication



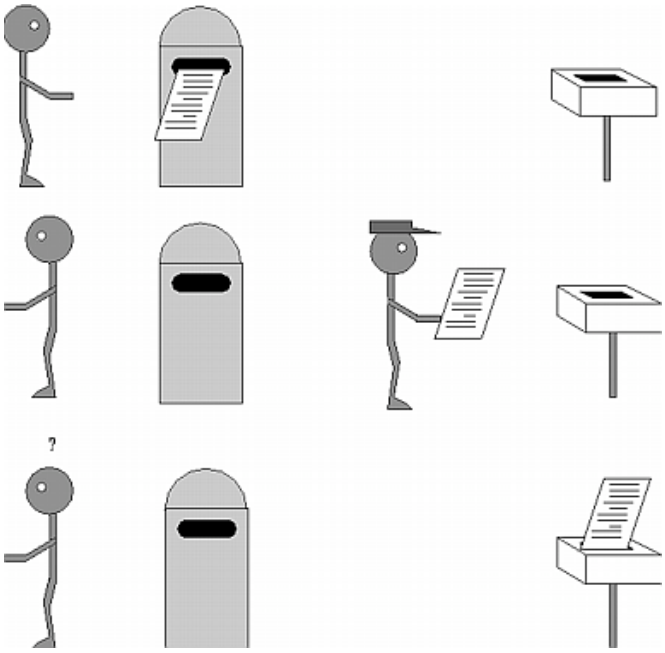
Synchronous Sends

Provide information about the relative execution points of sender and receiver - causes synchronization of the two.



Asynchronous Sends

Sender only know when the message has left.



Synchronous vs Asynchronous

Telephone analogy:

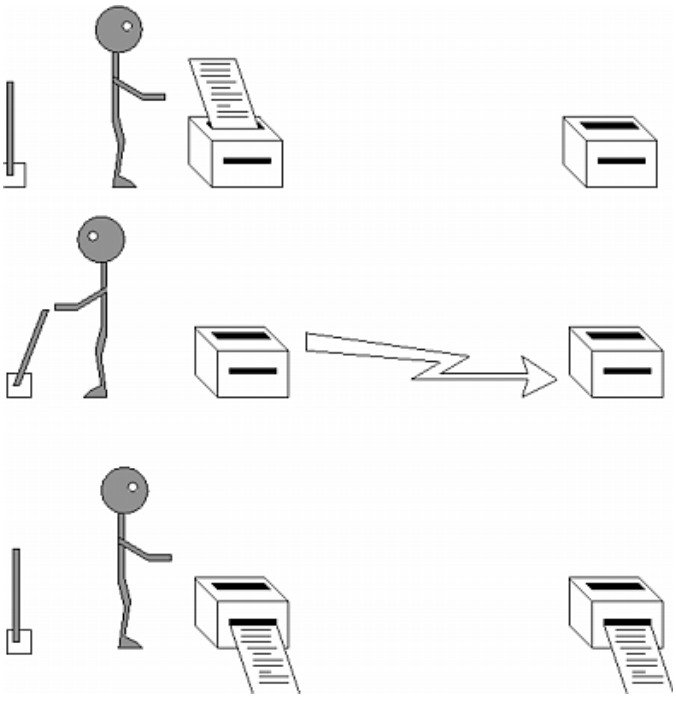
- synchronous - the receiver answers
- asynchronous - the answering machine answers

Blocking Operations

- Relates to when the local operation has completed.
- Only return from the MPI procedure call when the operation (whether it is send or receive) has completed.

Non-Blocking Operations

- Return immediately from MPI procedure (before it has completed).
- Your program does other work while MPI does message-passing operation "in the background".
- At some later time program can test or wait for the completion of the non-blocking operation.



Non-Blocking Operations (cont'd)

- All non-blocking operations should have matching wait operations. Some systems cannot free resources until wait has been called.
- A non-blocking operation immediately followed by a matching wait is equivalent to a blocking operation.
- Non-blocking operations are not the same as sequential subroutine calls as the operation continues after the call has returned.

Non-Blocking Operation Examples

Non-blocking *Send*:
requesting secretary to organise a meeting with colleague:

- usually you check on completion later.
- dont know when or if colleague receives notification.

Non-blocking *Receive*:
hanging your Xmas stocking! You're not allowed to hang around waiting ...

Summary

- "Orthogonal" concepts
  - Synchronous vs asynchronous is a "non-local" issue describing the relative timings of sender and receiver
  - Blocking vs non-blocking is a "local" concept describing completion of operation in either sender or receiver independently
- Both concepts impact performance and semantics

The rest of this section describes various modes of blocking communication

Communication modes

Sender mode	Notes	Synchronous?
Synchronous send	Message goes directly to receiver. Only completes when the receive has begun	*synchronous*
Buffered send	Message is copied in to a "buffer" (provided by the application). Always completes (unless an error occurs), irrespective of receiver.	*asynchronous*
Standard send	Either synchronous or buffered (into a fixed size buffer provided by MPI system)	*both/hybrid*
Ready send	*Assumes* the receiver is ready. Always completes (unless an error occurs), irrespective of whether the receive was ready.	*neither*
Receive	Completes when a message has arrived	

MPI Blocking Sender Modes

Operation	MPI Call
Standard send	`MPI_Send`
Synchronous send	`MPI_Ssend`
Buffered send	`MPI_Bsend`
Ready send	`MPI_Rsend`
Receive	`MPI_Recv</code`

Sending a message

C:

```
int MPI_Ssend(void *buf, int count,
              MPI_Datatype datatype, int dest,
              int tag, MPI_Comm comm)
```

Fortran:

```
 subroutine MPI_Ssend(buf, count, datatype,
dest,
                      tag, comm, ierror)
 <type> buf(*)
  integer count, datatype, dest, tag
  integer comm, ierror
```

Receiving a message

C:

```
int MPI_Recv(void *buf, int count,
             MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm,
             MPI_Status *status);
```

Fortran:

```
subroutine MPI_Recv(buf, count, datatype, source, tag,
                    comm, status, ierror)
 <type> buf(*)
 integer count, datatype, source, tag, comm, ierror
 integer status(MPI_STATUS_SIZE)
```

Synchronous Blocking Message-Passing

- Processes synchronise.
- Sender process specifies the synchronous mode.
- Blocking - both processes wait until the transaction has completed.

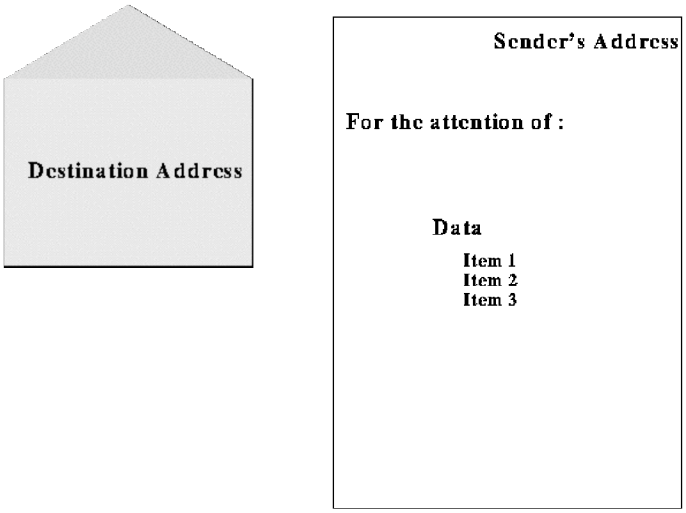
For a communication to succeed:

- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- The communicator must be the same.
- Tags must match.
- Message types must match.
- Receiver's buffer must be large enough.

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.

Communication Envelope



Communication Envelope Information

- Envelope information is returned from `MPI_Recv` as `status`
- Information includes:
  - Source: `status.MPI_SOURCE` or `status(MPI_SOURCE)`
  - Tag: `status.MPI_TAG` or `status(MPI_TAG)`
  - Count: call `MPI_Get_count()`

Received Message Count

C:

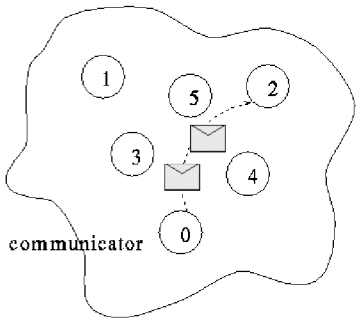
```
int MPI_Get_count (MPI_Status *status,
                   MPI_Datatype datatype,
                   int *count);
```

Fortran:

```
subroutine MPI_Get_count (status, datatype,
                          count, ierror)
integer status(MPI_STATUS_SIZE), datatype
integer count, ierror
```

Message Order Preservation

- Matching messages do not overtake each other - they are received in the order sent even with wildcarding
- This is true even for non-synchronous sends.
- Non-identical messages can be received out of order (use tag, source rank etc)



Exercise 2: Ping pong

1. Write a simple program to use `MPI_Send()` and `MPI_Recv()` to pass data between two processes.
2. Try using `MPI_SSend()` with messages going in both directions simultaneously .

```
Proc 0:         Proc 1:
MPI_Ssend()     MPI_Ssend()
MPI_Recv()      MPI_Recv()
```

   Now try it with `MPI_Send()` - by increasing the message size you should be able to determine the system buffer size.

3. Modify your program to repeatedly pass a message back and forth. Such a program is generally called *pingpong*.
4. Use `MPI_Wtime()` to time the data transfers for different size messages. The timer probably does not have great resolution so for short messages you will need to time multiple transfers. Try comparing the times for `MPI_Send()`, `MPI_Bsend()` and `MPI_Ssend()`.
5. Try using `MPI_Bsend()`. Try just replacing `MPI_Send()` by `MPI_Bsend()` -- does it work? Now use `MPI_Buffer_attach(buffer, size)` to provide buffer space.

[Solution in `pingpong.f/f90/c`]

Timers

C:

```
double MPI_Wtime(void);
```

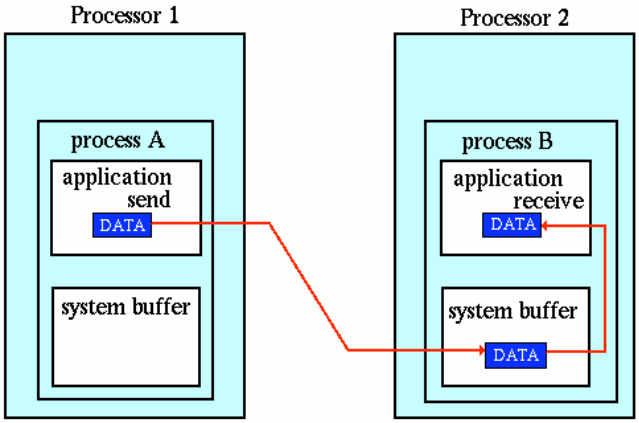
Fortran:

```
double precision MPI_Wtime()
```

- Time is measured in seconds.
- Time to perform a task is measured by consulting the timer before and after.
- Modify your program to measure its execution time and print it out.

Buffers

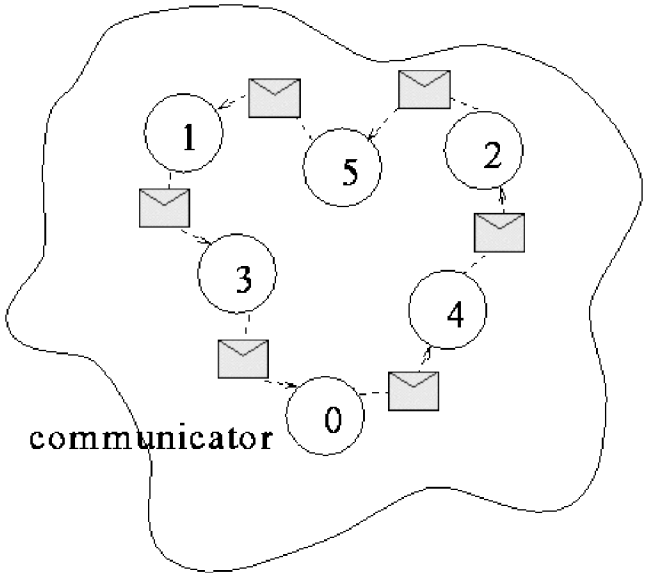
- Just memory in your processes
- Term used for three different concepts:
    1. Your programs variables or arrays (or part thereof) that are used as the "message space" arguments to any MPI routine passing a message,
       eg. `MPI_Send( a, ....)`
    2. The buffer space specifically allocated for `MPI_Bsend()` using `MPI_Buffer_attach()`. An "external" buffer space you have to control.
    3. The buffer space provided by the system



Path of a message buffered at the receiving process

Non-Blocking Communications

Deadlock

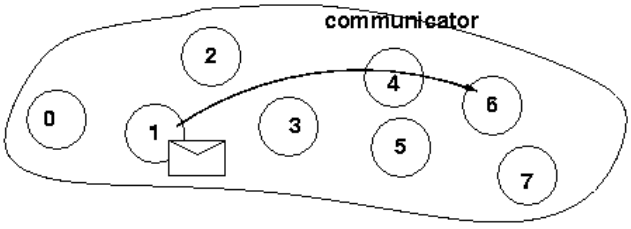


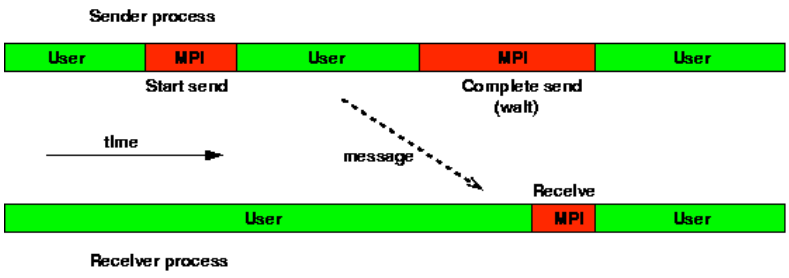
Non-Blocking Communications

Separate communication into three phases:

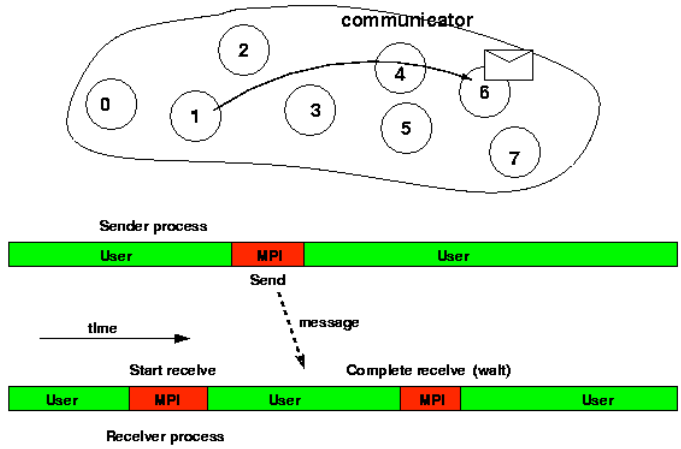
- Initiate non-blocking communication.
- Do some work (perhaps involving other communications?)
- Wait for non-blocking communication to complete.

Non-Blocking Send





Non-Blocking Receive



Handles used for Non-blocking Communication

- datatype - same as for blocking (`MPI_Datatype` or `integer`)
- communicator - same as for blocking (`MPI_Comm` or `integer`)
- request - `MPI_Request` or `integer`
- A request handle is allocated when a communication is initiated.

Non-blocking Synchronous Send

C:

```
MPI_Request request;
MPI_Issend(buf, count, datatype, dest, tag,
           comm, &request);
MPI_Wait(&request, &status);
```

Fortran:

```
integer request
call MPI_Issend(buf, count, datatype, dest, tag,
                comm, request, ierror)
call MPI_Wait(request, status, ierror)
```

Non-blocking Receive

C:

```
MPI_Request request;
MPI_Irecv(buf, count, datatype, src, tag,
          comm, &request);
MPI_Wait(&request, &status);
```

Fortran:

```
integer request
call MPI_Irecv(buf, count, datatype, src, tag,
               comm, request, ierror)
call MPI_Wait(request, status, ierror)
```

Blocking and Non-Blocking

- Send and receive can be blocking or non-blocking.
- A blocking send can be used with a non-blocking receive, and vice-versa.
- Non-blocking sends can use any mode - synchronous, buffered, standard, or ready.
- Synchronous mode affects completion, not initiation.

Non-Blocking Restrictions

- Cannot read or write buffer while non-blocking request (either send or receive) is outstanding
- State of data in buffer is indeterminate until request is completed
- Beware of using temporary storage deallocation for message buffers, eg. Fortran90 array sections and C and Fortran variables.

Communication Modes

Non-Blocking Operation	MPI Call
Standard send	`MPI_Isend`
Synchronous send	`MPI_Issend`
Buffered Send	`MPI_Ibsend`
Ready send	`MPI_Irsend`
Receive	`MPI_Irecv`

Completion

Waiting versus Testing.

C:

```
int flag;
MPI_Request request;
MPI_Status status;
MPI_Wait(&request, &status);
MPI_Test(&request, &flag, &status)
```

Fortran:

```
logical flag
MPI_Wait(request, status, ierror)
MPI_Test(request, flag, status, ierror)
```

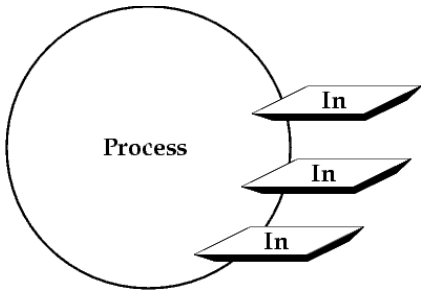
`flag` is a boolean which is true when the operation has completed.

Multiple Communications

- Test or wait for completion of one message.
- Test or wait for completion of all messages.
- Test or wait for completion of as many messages as possible.
- Arguments become arrays of handles.
- `MPI_Wait_all()`, `MPI_Test_some()`, ...

Testing Multiple Non-Blocking Communications

- multiple outstanding receives
- buffers waiting to accept messages from matching senders.
- MPI calls to test/wait on all/some of these requests.



Exercise 3: Non-blocking communication

1. Construct a set of processes in a ring (so that 0 passes to 1 passes to ... n-2 passes n-1 passes to 0). Have each processor pass its rank to it's neighbour and keep passing each message it receives until it gets it' s own rank back. Let each processor keep a sum of the messages it receives and print the sum out when done. Use non-blocking communication to ensure safety .

   [Solution in `ring.f/f90/c`]

2. Write your own all-to-all broadcast using non-blocking communication. Have process i
   - send $i + j$ to process j for each j
   - receive all the messages sent to it
   - iteratively test for completion of any it's sends printing the number done each time until they are all done.
   - write out the messages received

   [Solution in `alltoall.f/f90/c`]

ring.f90

```
program ring
  include "mpif.h"
  integer :: ierror, rank, size, left, right, other, &
           & sum, i, request
  integer, dimension(MPI_STATUS_SIZE) :: send_status, &
                                       & recv_status
  call MPI_Init(ierror)
  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)
  call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)

  right = modulo(rank + 1, size)
  left = modulo(rank - 1, size)
  sum = 0

  do i = 0, size-1
     call MPI_Issend(rank, 1, MPI_INTEGER, right, 1, &
                   & MPI_COMM_WORLD, request, ierror)
     call MPI_Recv(other, 1, MPI_INTEGER, left, 1, &
                 & MPI_COMM_WORLD, recv_status, ierror)
     call MPI_Wait(request, send_status, ierror)
     sum = sum + other
     rank = other
  enddo

  write(*, '("PE",i1,": Sum = ",i4)') rank, sum
     call MPI_Finalize(ierror)
end program ring
```

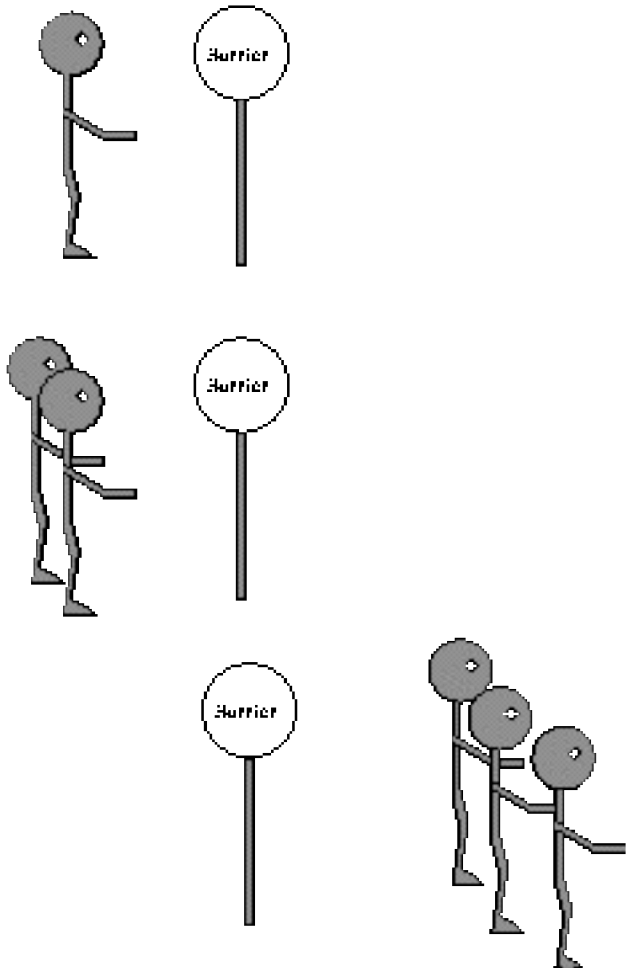
Collective Communications

Collective Communication

- Communications involving a group of processes.
- Called by all processes in a communicator.
- Examples:
  - Barrier synchronisation
  - Broadcast, scatter, gather.
  - Global sum, global maximum, etc.

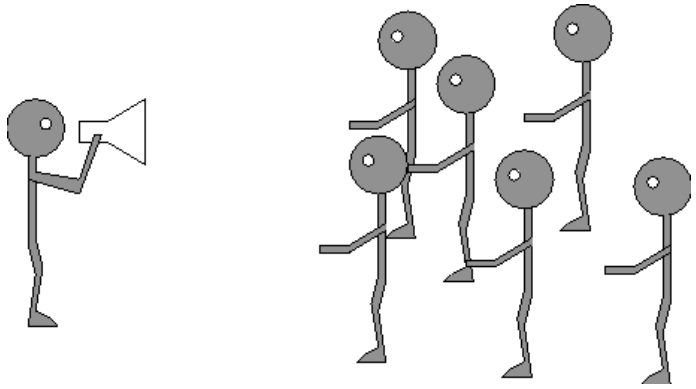
Barrier

Synchronise participating processes



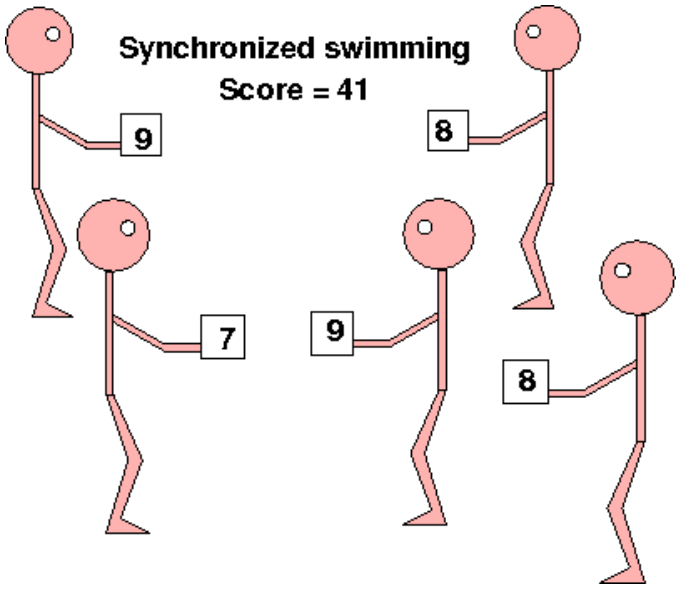
Broadcast

A one-to-all communication.



Reduction Operations

Combine data from several processes to produce a "reduced"
result the same size as each contribution, eg. SUM



Characteristics of Collective Communication

- Collective action over a communicator
- All processes must communicate
- Synchronisation may or may not occur
- All collective operations are blocking.
- No tags.
- Receive buffers must be exactly the right size

Collective communications

- Collective communication routines are higher level routines involving several processes at a time (often all).
- Can be built out of point-to-point communications.

Barrier Synchronisation

C:

```
int MPI_Barrier(MPI_Comm comm);
```

Fortran:

```
subroutine MPI_Barrier(comm, ierror)
integer comm, ierror
```

Broadcast

C:

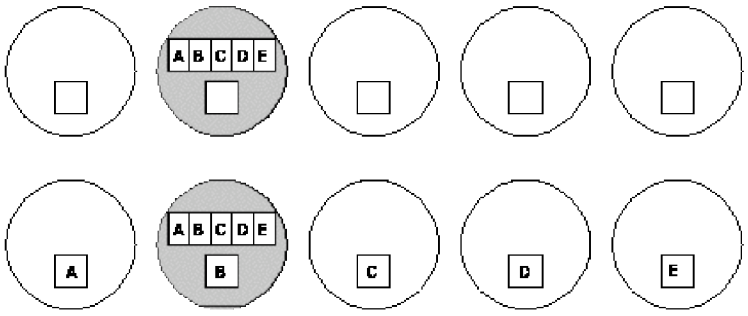
```
int MPI_Bcast(void *buffer, int count,
              MPI_Datatype datatype,
              int root, MPI_Comm comm);
```

Fortran:

```
subroutine MPI_Bcast (buffer, count, datatype,
                      root, comm, ierror)
<type> buffer(*)
integer count, datatype, root, comm, ierror
```

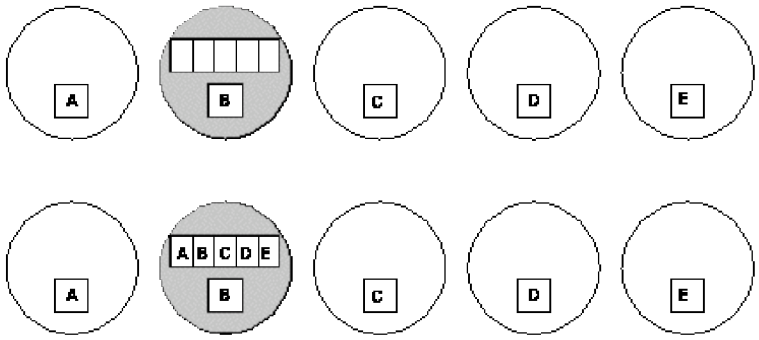
`root` is the rank of the broadcaster.

Scatter




```
MPI_Scatter(sendbuf, sendcount, sendtype,
            recvbuf, recvcount, recvtype, root, comm)
```

Gather




```
MPI_GATHER(sendbuf, sendcount, sendtype,
           recvbuf, recvcount, recvtype, root, comm)
```

Global Reduction Operations

- Used to compute a result involving data distributed over a group of processes.
- Examples:
  - global sum or product
  - global maximum or minimum
  - global user-defined operation

C:

```
int MPI_Reduce(void *inbuf, void *outbuf, int count,
               MPI_Datatype datatype, MPI_Op op,
               int root, MPI_Comm comm);
```

Fortran:

```
subroutine MPI_Reduce (inbuf, outbuf, count, datatype,
                       op, root, comm, ierror)
<type> inbuf(*), outbuf(*)
integer count, datatype, op, root, comm, ierror
```

Example of Global Reduction

Integer global sum

C:

```
MPI_Reduce(&x, &result, 1, MPI_INT, MPI_SUM,
           3, MPI_COMM_WORLD)
```

Fortran:

```
call MPI_Reduce(x, result, 1, MPI_INTEGER,
MPI_SUM,
                3, MPI_COMM_WORLD, ierror)
```

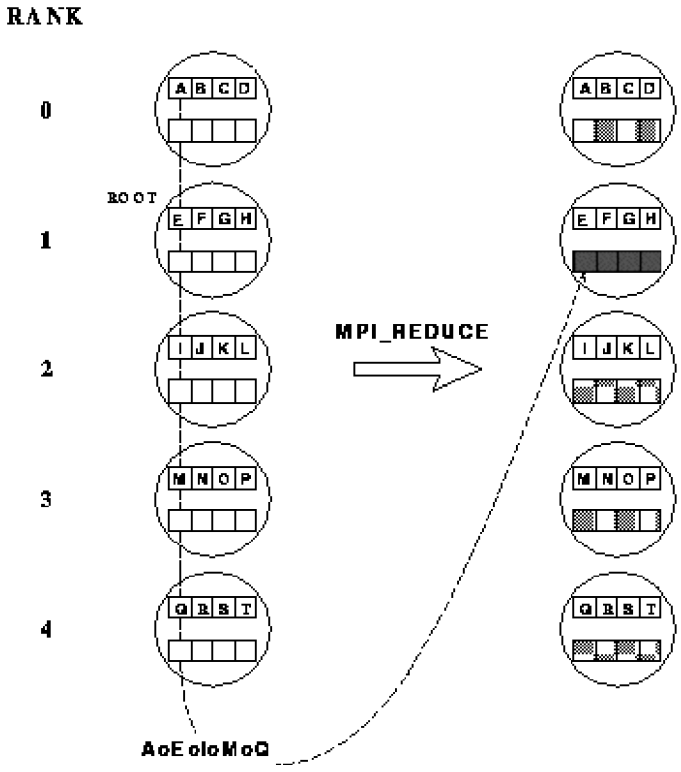
Sum of all the `x` values is placed in `result`.

The result is only placed there on process 3 (the root process).

Predefined Reduction Operations

MPI Name	Function
`MPI_MAX`	Maximum
`MPI_MIN`	Minimum
`MPI_SUM`	Sum
`MPI_PROD`	Product
`MPI_LAND`	Logical AND
`MPI_BAND`	Bitwise AND
`MPI_LOR`	Logical OR
`MPI_BOR`	Bitwise OR
`MPI_LXOR`	Logical exclusive OR
`MPI_BXOR`	Bitwise exclusive OR
`MPI_MAXLOC`	Maximum and location
`MPI_MINLOC`	Minimum and location

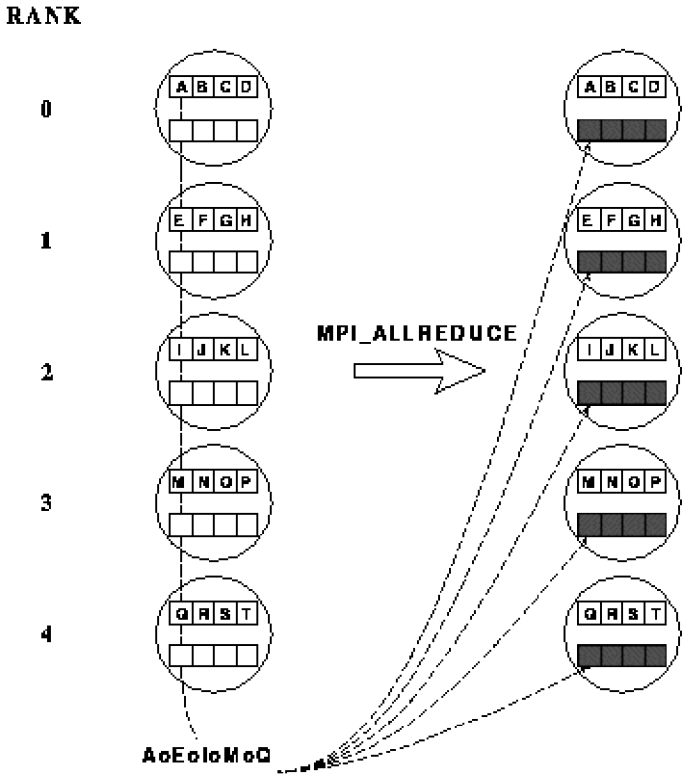
MPI_REDUCE



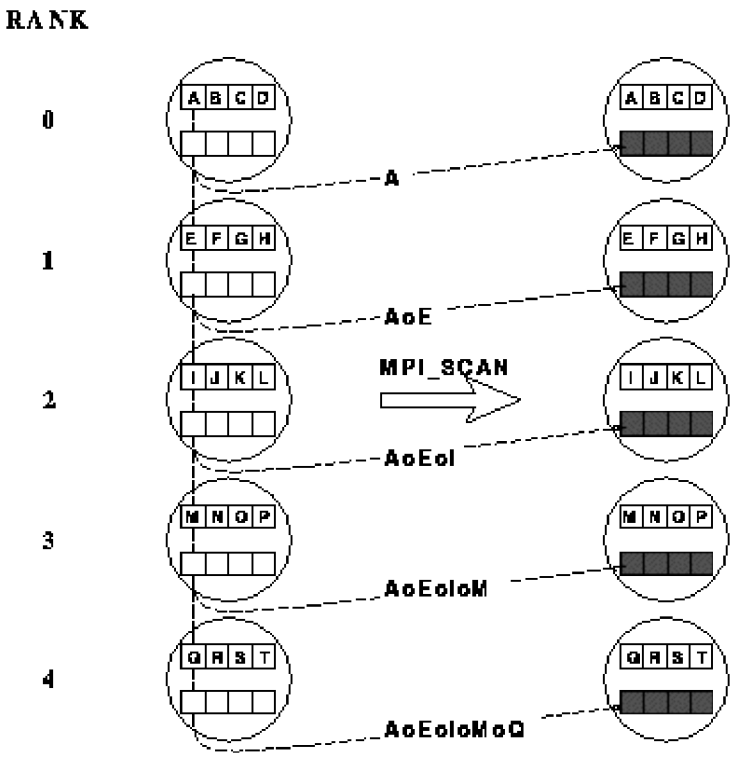
Variants of MPI_REDUCE

- `MPI_Allreduce` - no root process
- `MPI_Reduce_scatter` - result is scattered
- `MPI_Scan` - "parallel prefix"

MPI_ALLREDUCE



MPI_SCAN



User-Defined Reduction Operators

Reducing using an arbitrary operator, n

C: function of type MPI_User_function

```
void my_operator (void *invec, void *inoutvec,
                  int *len, MPI_Datatype *datatype)
```

Fortran: function of type

```
function my_operator (invec(*), inoutvec(*),
                      len, datatype)
    <type> invec(len), inoutvec(len)
    integer len, datatype
```

Reduction Operator Functions

- Operator function for n must act as:

```
for (i = 1 to len)
inoutvec(i) = inoutvec(i)
n invec(i)
```

- Operator n need not commute

Registering a User-Defined Reduction Operator

Operator handles have type MPI_Op or INTEGER

C:

```
int MPI_Op_create (MPI_User_function *function,
                   int commute, MPI_Op *op)
```

Fortran:

```
MPI_Op_create (funct, commute, op, ierror)

        external func
        logical commute
        integer op, ierror
```

Exercise 4: Collective Communication

1. The code in `Exercises/serial_pi.f/f90/c` implements a simple mid-point rule to evaluate

$$\pi = 4 \int_0^1 \frac{1}{1+x^2} dx$$

   Use `MPI_Bcast()` and `MPI_Reduce()` to parallelize the code.

   [Solution in `pi.f/f90/c`]

2. One way to organize output from multiple processes is to nominate one of the processes to receive output from the other processes and do the writes (in rank order) only from that process. Use `MPI_Gather()` to implement this idea to write out the "*hello world from proc ...*" strings.

   [Solution in `IO_gather.f/f90/c`]

Miscellaneous

MPI Datatypes

- Basic types
- Derived types
  - vectors
  - structs
  - others

Derived Datatypes - Type Maps

Creating a derived datatype requires describing the layout of the data in memory, i.e. the displacement to each component type.

basic datatype 0	displacement of datatype 0
basic datatype 1	displacement of datatype 1
...	...
basic datatype n-1	displacement of datatype n-1

MPI IO

MPI IO allows data across multiple processes to be saved to disk in a convenient fashon.

- Cooperative IO operations that can produce files usable outside MPI
- Somewhat C oriented interface - have to emulate Fortran sequential IO

Example:

```
MPI_File_open(MPI_COMM_WORLD, filename, MPI_MODE_RDWR,
              info, fh);
MPI_File_write_ordered(fh, a, n, MPI_INT, mpistatus);
MPI_File_close(fh);
```

MPI IO (cont'd)

- MPI IO is not necessarily high performance
- Multiple ways to access the file
  - individual file pointer
  - shared file pointer

Virtual Topologies

- Convenient process naming
- Naming scheme to fit the communication pattern
- Simplifies writing of code
- Can allow MPI to optimise communications

How to use a Virtual Topology

- Creating a topology produces a new communicator
- MPI provides "mapping functions"
- Mapping functions compute processor ranks, based on the topology naming scheme.

MPI 2

- One-sided communications:
  - direct access to another processes memory
  - `MPI_Put, MPI_Get, ...`
  - requires setting up address mappings between processes ("windows")
  - gets in to shared memory issues like locking
- Dynamic process
  - create and destroy processes dynamically
  - `MPI_Spawn`
- C++ and Fortran90 bindings

Tools for Developing MPI Programs

Debugging - Totalview

- TotalView debugger control all processes (and threads) from a single session.
- Setup your environment by doing

```
> module load totalview
```

- Compile code with -g compiler option.
- Start totalview debugger with

```
totalview mpirun -a -n 2 mpiprog
```

- Comes up with mpirun source code! Hit "g" and answer "yes" to the prompt
- See the APAC-NF Totalview software page (http://nf.apac.edu.au/facilities/software/) and links from there for more details and links to User Guides.

Totalview Exercise

```
cd Solutions/F
ifort -g par_laplace.f -o lap –lmpi
```

Part 1

```
qsub -I -q express -wd -lncpus=4,vmem=400mb -v DISPLAY
```

Wait for prompt then enter

```
totalview mpirun -a -n 4 ./lap
```

Type "g" then select "Yes" to get beyond the mpirun source

Scroll source window and click on line 38 block

Type "G" to run all processes to run to line 38

Click on rank variable to see value.

Use "P+" and "P-" button to see other processes and compare rank.

Use options under Process and Group menu to step one and all processes.

Part 2

Use Restart under the Group menu to restart the debug session.

Set breakpoint at line 146 Right click on breakpoint in bottom right window

Select Properties

Select Evaluate and enter

```
if (iter == 100) then
  $stop
endif
```

Click OK

Type "G" for group go.

Check that iter = 100

Double click on the variable name "u"

Select Tools the Visualize

Use middle button to rotate

Use the "P+" button on the main window to find the rank 3 process (prun.3)

Repeat above

Exit totalview AND exit qsub

Part 3

Edit line 124 of par_laplace.f to replace "tag" by "tag+1"

Recompile and restart totalview.

Type "G" to start all processes

Select "Halt" when things are hung

Select "Tools" then "Message Queue" to see messages

Select "Tools" then "Message Queue Graph" to see messages

Double click on message to see message details.

Profiling - Intel Trace Collector/Analyzer

- The Intel Trace Collector and Analyzer gives a myriad performance profiling information including a "timeline" view of the MPI activity of each process as well as summary performance information.
- Setup your environment by doing

```
module load intel-itc module load intel-ita
```

- Link code with:
`-L$VT_ROOT -lVT -lmpi -ldwarf -lelf -lvtunwind -lnsl -lm -lpthread` for Fortran and C code.
- Run code as normal. For an executable `a.out`, a number of files `a.out.stf.*` will be created.
- Then run
`traceanalyzer a.out.stf`
- See the software web page for further details.

Profiling - Jumpshot

- The Jumpshot profiler gives a "timeline" view of the MPI activity of each process as well as summary performance information.
- Setup your environment by doing

```
module load jumpshot
```

- Link code with:
`-L$JUMPSHOT_LIB -lmpe_f2cmpi -llmpe -lmpe -lmpi` for Fortran code and
`-L$JUMPSHOT_LIB -llmpe -lmpe -lmpi -lm` for C code.
- Run code as normal. A file Unknown.clog will be created in the same directory as the executable
- Then do
`jumpshot Unknown.clog`
- The first step will be to convert the clog file to slog2 format.
- See the software web page for further details.

Communication Modes

Synchronous mode:

Synchronous send requires a matching receive before it can return.

Buffered mode:

Buffered send requires the user to provide buffer space with MPI_Buffer_attach(type buf(*), integer size) (there is an analogous detach command). The MPI system may provide some buffer space by default but not standard. Returns when message is copied in to buffer.

Standard mode:

Standard send can be either synchronous or buffered using a system provided buffer and will possibly be a hybrid depending on the message size. Cannot rely on either behaviour being used. Programs that assume the receiver has received the message when the send returns or rely on having sufficient system buffering space to complete are "unsafe". A lot of programs do this though.

Ready mode:

Ready sends only used when you absolutely know the receiver is ready (like when it makes a request).

MPI vs OpenMP

MPI

Pros:

- Very portable
- Requires no special compiler
- Requires no special hardware but can make use of high performance hardware
- Very flexible -- can handle just about any model of parallelism
- No shared data! (You dont have to worry about processes "treading on each other's data" by mistake.)
- Can download free libraries for your Linux PC!
- Forces you to do things the "right way" in terms of decomposing your problem.

Cons:

- All-or-nothing parallelism (difficult to incrementally parallelise existing serial codes)
- No shared data! Requires distributed data structures
- Could be thought of assembler for parallel computing -- you generally have to write more code
- Partitioning operations on distributed arrays can be messy.

OpenMP

Pros:

- Incremental parallelism -- can parallelize existing serial codes one bit at a time
- Quite simple set of directives
- Shared data!
- Partitioning operations on arrays is very simple.

Cons:

- Requires proprietary compilers
- Requires shared memory multiprocessors
- Shared data!
- Having to think about what data is shared and what data is private
- Cannot handle models like master/slave work allocation (yet)
- Generally not as scalable (more synchronization points)
- Not well-suited for non-trivial data structures like linked lists, trees etc