Programming with MPI

Point-to-Point Transfers

Nick Maclaren

nmm1@cam.ac.uk

May 2008
Digression

Most books and courses teach **point-to-point** first
And then follow up by teaching **collectives**

This course hasn’t – why not?

- **Point-to-point** is hard to use **correctly**
I usually make a complete mess of it, first time
See Hoare’s *Communicating Sequential Processes*
Hoare designed **BSP** based on his experience!

After all, who programs in **assembler** nowadays?
**Point-to-point** is the **assembler-level** interface
Using Point-to-Point

- Above all, KISS – Keep It Simple and Stupid
- Design proper primitives, don’t just code

Simplest to use one of two design models for that:
  - Your own collective – see later
  - Two processes, doing nothing else

- It’s easiest if your primitives don’t overlap
  Can separate by barriers and debug separately
  Almost essential for tuning – see later
Think of point-to-point as a sort of Email. Like that, messages come in envelopes.

MPI’s envelopes contain the following:

- The source process
- The destination process
- The communicator
- An identifying tag

One of the first two is the calling process. The others are specified in the arguments.
Receive Status (1)

A receive action returns a status
This contains the following:
- The source process
- The identifying tag
- Other, hidden, information

Already know the communicator and destination
A function to extract the message size
Receive Status (2)

In C, the status is a typedef structure `MPI_Status`

In Fortran, it is an integer array
`INTEGER, DIMENSION(MPI_STATUS_SIZE)`

- You declare these yourself, as normal
  Including in static memory or on the stack

- They are not like communicators
  You don’t call MPI to allocate and free them
Receive Status (3)

For now, you can largely ignore the status. You don’t need to look at it for very simple use.

- In **MPI 1**, had to provide the *argument*. This is the form that I shall use in examples.

- **MPI 2** allowed you to not provide it. I don’t recommend doing that, in general.
The Simplest Use

Assume *communicator* is MPI_COMM_WORLD

The *tag* is needed only for advanced use
Quite useful for added *checking*, though

So it’s only the *destination* and *source*
The *latter* is set automatically for *send*!
And the *former* is for *receive*!

The functions are MPI_Send and MPI_Recv
Fortran Example (1)

REAL(KIND=KIND(0.0D0)) :: buffer ( 100 )
INTEGER :: myrank , error
INTEGER , PARAMETER :: from = 2 , to = 3 , &
tag = 123

CALL MPI_Rank ( myrank , error )
IF ( myrank == from ) THEN
   CALL MPI_Send ( buffer , 100 , &
                   MPI_DOUBLE_PRECISION , to , tag , &
                   MPI_COMM_WORLD , error )
END IF
REAL(KIND=KIND(0.0D0)) :: buffer ( 100 )
INTEGER :: myrank, error, status ( MPI_STATUS_SIZE )
INTEGER, PARAMETER :: from = 2, to = 3, &
tag = 123

CALL MPI_Rank ( myrank, error )
IF ( myrank == to ) THEN
   CALL MPI_Recv ( buffer, 100, &
      MPI_DOUBLE_PRECISION, from, tag, &
      MPI_COMM_WORLD, status, error )
END IF
C Example (1)

double buffer [ 100 ];
int myrank, from = 2, to = 3, tag = 123, error;

error = MPI_Rank ( & myrank );
if (myrank == from )
  error = MPI_Send (buffer, 100, MPI_DOUBLE, 
                 to, tag, MPI_COMM_WORLD );
C Example (2)

double buffer [ 100 ] ;
MPI_Status status ;
int myrank, from = 2, to = 3, tag = 123, error ;

error = MPI_Rank ( & myrank ) ;
if ( myrank == to )
    error = MPI_Recv ( buffer, 100, MPI_DOUBLE ,
from, tag, MPI_COMM_WORLD, & status ) ;
Beyond That

Trivial as that is, it’s enough to cause trouble. There are some examples on how that can happen.

And it’s not enough for all real programs. MPI provides lots of knobs, bells and whistles.

- You should use only what you need.
  Don’t use something because it looks cool.

- You need to know what can be done.
  When you need something extra, look it up.
Blocking (1)

Receive will block until a matching send
If one is never posted, it will hang indefinitely

Send may block until a matching receive
Or it may copy the message and return
and MPI will transfer it in due course

Unspecified and up to the implementation
May vary between messages, or phase of the moon
• Correct MPI programs will work either way

• You can control that yourself — see later
Blocking (2)

Processes A and B want to swap data

Both send the existing value, and then receive? It will sometimes work and sometimes hang

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
</tr>
</thead>
<tbody>
<tr>
<td>send to B</td>
<td>send to A</td>
</tr>
<tr>
<td>Both may wait until transfers received</td>
<td></td>
</tr>
<tr>
<td>receive from B</td>
<td>receive from A</td>
</tr>
</tbody>
</table>
Blocking (3)

In that case, it’s trivial to avoid

- If \( A < B \), \( A \) sends first and \( \text{receives} \) second
  And \( B \) receives first and \( \text{sends} \) second

And conversely if \( A > B \)

Complicated transfer graphs are easy to get wrong
MPI provides several ways to avoid the problem
Use whichever is simplest for your purposes
Transfer Modes (1)

MPI_Ssend is synchronous (will block)
returns when the message has been received

MPI_Bsend is buffered (won’t block)
so the swap example above will never hang

• Exactly the same usage as for MPI_Send
MPI_Send simply calls one or the other

Generally, don’t use either of them
Both have important, but advanced, uses
Transfer Modes (2)

A *synchronous send* avoids a separate handshake
Completing the call acknowledges *receipt*

- Use it if it avoids an explicit acknowledgement

Buffering is more tricky, surprisingly enough
*Sends are erroneous* if the buffer becomes full

- And the *default buffer size is zero!*
  But *exceeding it is undefined behaviour!*
  Using buffering is covered later
Composite Send and Receive (1)

- There is a composite send and receive. Will do the in the right order to avoid deadlock. Can also match ordinary send and receive.

- It also has a form that updates in place. Sends buffer and then receives into it. That may involve extra copying, of course.

Use these if they are what you want to do. They aren’t likely to be any more efficient.
Composite Send and Receive (2)

Fortran example:

```fortran
REAL(KIND=KIND(0.0D0)) :: &
  putbuf ( 100 ), getbuf ( 100 )
INTEGER :: error, status ( MPI_STATUS_SIZE )
INTEGER, PARAMETER :: from = 2, to = 3, &
  fromtag = 123, totag = 456

CALL MPI_Sendrecv ( putbuf, 100, &
  MPI_DOUBLE_PRECISION, to, totag, &
  getbuf, 100, MPI_DOUBLE_PRECISION, &
  from, fromtag, &
  MPI_COMM_WORLD, status, error )
```
Composite Send and Receive (3)

Fortran in place example:

```fortran
REAL(KIND=KIND(0.0D0)) :: buffer ( 100 )
INTEGER :: error , status ( MPI_STATUS_SIZE )
INTEGER , PARAMETER :: from = 2 , to = 3 , &
   fromtag = 123 , totag = 456

CALL MPI_Sendrecv_replace ( &
   buffer , 100 , MPI_DOUBLE_PRECISION , &
   to , totag , from , fromtag , &
   MPI_COMM_WORLD , status , error )
```

Programming with MPI – p. 21/??
C Example

double putbuf [ 100 ] , getbuf [ 100 ] , buffer [ 100 ] ;
MPI_Status status ;
int from = 2 , to = 3 , fromtag = 123 , totag = 456 ,
error ;

error = MPI_Sendrecv (
    putbuf , 100 , MPI_DOUBLE , to , totag ,
    getbuf , 100 , MPI_DOUBLE , from , fromtag ,
    MPI_COMM_WORLD , & status ) ;

error = MPI_Sendrecv_replace ( 
    buffer , 100 , MPI_DOUBLE , to , totag ,
    from , fromtag , MPI_COMM_WORLD , & status ) ;
Unknown Message Size (1)

The send and receive sizes need not match

- It is an error if the receive is smaller

Only the send count values are updated
E.g. sending 30 items and receiving 100 items
will leave the last 70 items unchanged

- But there is a better way to do this
Allows receiving truly unknown size messages
This is where you start to use the status
Unknown Message Size (2)

• Can accept the message with MPI_Probe
  Calling it probe is a bit of a misnomer
  It accepts the message and updates the status
  But it doesn’t transfer the data anywhere

• You discover the size with MPI_Get_count
  Then you can allocate a suitable buffer
  MPI_Get_count needs the datatype
  Allows for conversion, not covered here

• Lastly, you receive the message as normal
Fortran Example

REAL(KIND=KIND(0.0D0)) , &
  ALLOCATABLE :: buffer ( : )
INTEGER :: error , count , &
  status ( MPI_STATUS_SIZE )
INTEGER , PARAMETER :: from = 2 , tag = 123

CALL MPI_Probe ( from , tag , &
  MPI_COMM_WORLD , status , error )
CALL MPI_Get_count ( status , &
  MPI.DOUBLE_PRECISION , count , error )
ALLOCATE ( buffer ( count ) )
CALL MPI_Recv ( buffer , count , &
  MPI.DOUBLE_PRECISION , . . .

Programming with MPI – p. 25/??
**C Example**

double * buffer;
int from = 2, tag = 123, error, count;
MPI_Status status;

error = MPI_Probe ( from, tag,
            MPI_COMM_WORLD, & status );
error = MPI_Get_count ( & status,
            MPI_DOUBLE, & count );
buffer = malloc ( sizeof ( double ) * count );
if ( buffer == NULL ) . . . ;
error = MPI_Recv ( buffer, count, MPI_DOUBLE,
            from, tag, MPI_COMM_WORLD, & status );
Checking for Messages (1)

• Real `probe` function is called `MPI_Iprobe`
  It returns immediately even if no matching message

An extra `Boolean` argument saying if there is one

• If there is one, it behaves just like `MPI_Probe`
• If there isn’t one, the `status` is not updated

It’s so similar, shall show only the actual differences
Checking for Messages (2)

**Fortran example:**

```fortran
LOGICAL :: flag

CALL MPI_Iprobe ( from, tag, &
MPI_COMM_WORLD, flag, status, error )
```

**C example:**

```c
int flag;

error = MPI_Iprobe ( from, tag, 
MPI_COMM_WORLD, & flag, & status );
```
Wild Cards (1)

• You can accept messages from any process
  Just use **MPI_ANY_SOURCE** for from

The actual **source** is stored in the **status**
  using the name **MPI_SOURCE**

**Fortran** example:   `status(MPI_SOURCE)`
**C** example:        `status . MPI_SOURCE`

• Be warned     –   your footgun is now loaded
Wild Cards (2)

- You can accept messages with any tag. Just use `MPI_ANY_TAG` for tag.
- Use the name `MPI_TAG` like `MPI_SOURCE`.

I advise using the tag for cross-checking:
- It could be a message sequence number.
- Or identify the object being transferred.
- Or whatever else would help debugging.

- On receipt, check it is what you expect.
  If it isn’t, you can write your own diagnostics.
  Including as much program state as you want.
Fortran Example

INTEGER :: error, count, from, tag, &
 status ( MPI_STATUS_SIZE )

CALL MPI_Probe ( MPI_ANY_SOURCE, &
 MPI_ANY_TAG, MPI_COMM_WORLD, &
 status, error )
CALL MPI_Get_count ( status, &
 MPIDOUBLE_PRECISION, count, error )
from = status ( MPI_SOURCE )
tag = status ( MPI_TAG )
C Example

```c
int error, from, tag, count;
MPI_Status status;

error = MPI_Probe ( MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, & status );
error = MPI_Get_count ( & status, MPI_DOUBLE, & count );
from = status . MPI_SOURCE;
tag = status . MPI_TAG;
```
Message Ordering (1)

Each **process** has a **FIFO** receipt (queue)
**Incoming** messages never overtake each other

Every **probe** and **receive** match in queue order
First message that satisfies **all** of the constraints

**Probe** and **receive** get same message if
  - There has been no intervening **receive**
  - Same **communicator**, **source** and **tag**

Other safe usages, too, but that one is easy
Message Ordering (2)

If you probe using wild cards, you can also extract the source and tag from status and then use those values in the receive.

If process A does multiple sends to process B, those messages arrive in the same order.

- No ordering if sender or receiver differ

And messages can be delayed considerably.
Tag Warning

The main purpose of **tags** is not for **checking**
It’s to allow **independent** communication paths

Many books and Web pages will describe that use
Some will even encourage it

**Don’t do it**

It’s the equivalent of cocking your footgun
Using **tags** like that is very hard for **experts**

- I will contradict myself later, under I/O
Buffered Sends (1)

These are trivial to use, but need extra mechanism

- Default buffer size is *implementation dependent* and doesn’t even have to be documented! *IBM* chose to use 8 bytes for *poe*

- So you *have* to allocate a buffer first. It’s just a block of memory – any type will do

That’s really the only extra complexity. And you can usually just make it very big
Buffered Sends (2)

You attach a single buffer to a process not a communicator – why not?

When you have finished doing transfers, detach it
• It is used for scratch space by MPI in between
Best to set immediately after MPI_Init
And detach immediately before MPI_Finalize

The MPI standard is (unusually) not very clear
Does the detach read its arguments or not?
I recommend setting them before the call anyway
Buffered Sends (3)

When a buffer is in use by MPI

- Do NOT fiddle with it in ANY way! Its use and contents are completely undefined

- Watch out in garbage-collected languages. Make sure that the buffer will not move around

- Even in Fortran and C Make sure that it does not go out of scope Or falls foul of Fortran copy-in/copy-out
Allocating a Buffer (1)

**Fortran example:**

```fortran
INTEGER, PARAMETER :: buffsize = 10000
CHARACTER :: buffer ( buffsize )
INTEGER :: oldsize, error

CALL MPI_Buffer_attach ( buffer, buffsize, error )
oldsize = buffsize
CALL MPI_Buffer_detach ( buffer, oldsize, error )
```

*Detach* returns the values previously stored
I have no idea what this means for *buffer*!
Allocating a Buffer (2)

C example:

```c
#define buffsize 10000
void * buffer = malloc ( buffsize ), * oldbuff;
int oldsize, error;

error = MPI_Buffer_attach ( buffer, buffsize );

oldbuff = buffer;
oldsize = buffsize;
error = MPI_Buffer_detach ( & oldbuff, & oldsize );
```

Note the indirections (&) in detach
Detach stores the values previously stored
Use of Buffered Sends (1)

Using them is generally not advisable
They usually hide problems rather than fix them
And they can be quite a lot less efficient

If you have a completely baffling failure
try changing all sends to buffered

- If that helps, you have a race condition
You then must track it down and fix it properly

The other main use is for I/O (see later)
Use of Buffered Sends (2)

You can calculate how much space you need

Constant **MPI_BSEND_OVERHEAD**

Function **MPI_Pack_size**

Function **MPI_Sizeof**  
[ Only in Fortran ]

Using those is overkill for almost all programs
This course doesn’t describe their use
Epilogue

There is more on point-to-point later
Mainly non-blocking (asynchronous) transfers

But we have covered most of blocking transfers
Exercises will try out quite a lot of this

Main one is to code a rotation collective
Each process sends to its successor
And the last one sends back to the beginning
Practicals

Practicals often use **buffered** or **synchronous** sends
Reason is to **expose** or **hide** cases of **deadlock**

- This is advised **only** when testing
You should normally use **ordinary** sends