Programming with MPI

More on Point-to-Point

Nick Maclaren

nmm1@cam.ac.uk

May 2008

Increasing Complexity

You may well want to use these features
“Festina lente” (“make haste slowly”)

Start with a variety of useful minor features

Then onto non–blocking transfers
Easy to use, not always easy to understand
But it’s correspondingly important and useful

Sending to Oneself (1)

A process can send a message to itself

- I generally don't recommend doing that

I do it in some of the specimen answers, though

Using blocking calls carelessly will deadlock

Send–receive is the only safe and easy case

For other point–to–point, you MUST use buffering

- Otherwise use only with non–blocking calls

But you need to be careful even doing that

Sending to Oneself (2)

Consider writing one's own collectives

- You can treat the local process separately
- Or use the whole communicator symmetrically

For the latter, processes send to themselves
The obvious non–blocking code will work in MPI
Only if both the send and receive are non–blocking

- Do whichever makes your code cleaner

Null Processes (1)

You can specify a null source or destination

sends return immediately (?), successfully

receives return immediately (?), successfully
without updating the transfer buffer

This may enable you to simplify code at boundaries

- Use the facility only if it clarifies your code

- Be careful with non–blocking transfers

MPI isn't clear what happens in that case

Null Processes (2)

Use MPI_PROC_NULL as a process number

The status value contains
source = MPI_PROC_NULL, tag = MPI_ANY_TAG
MPI_Get_count on it returns zero

⇒ Actually, MPI isn't entirely consistent
source may also be MPI_ANY_SOURCE
Reasonable programs won't look at it, anyway

Non-Blocking Transfers

Also called asynchronous transfers

Ex–mainframe programmers know these are best
Books often describe these as more efficient

Unfortunately, all modern systems are synchronous
This lecture will describe only how to use them

- A lot of it is describing what not to do

Experience with asynchronism is rare nowadays

How They Work (1)

- The main call starts an asynchronous transfer
It returns a handle, called a request

- Later, you wait on the request until finished
Only then has the transfer completed

- The wait frees the request and sets the status
You rarely need to free the request yourself

- You can also test whether a request is ready

How They Work (2)

- The actual buffer update is anywhere in between

Indeed, bytes may change in a random order

- It must not even be inspected during that

  Pure send buffers may be read

- The buffer obviously must not move!

Take care in a garbage collected language
And with C++/STL and copy/move constructors
You may need to play fancy games to stop that

Use as normal once the transfer has completed

Race Conditions (1)

The window is between the send or receive
and waiting for the request to complete

For a non–blocking send:

- You must not update the buffer in the window

Reading the buffer doesn't cause problems

For a non–blocking receive:

- You must not access the buffer in the window

Obviously, you must not allocate or free it

Race Conditions (2)

- Chaos awaits if you break those rules

Though it will often not show up in simple tests

Non–blocking transfers are the only cause
in the subset of MPI that this course covers ...

- Fortran users watch out for array copying

That counts as a form of reallocation

Some guidelines later, or see Fortran course
If they don't help, need to ask an expert

Using Non-Blocking (1)

Generally, start them as soon as possible
Wait for completion when you need the buffer

More advanced use is waiting on several requests
And dealing with them in the order they are ready

Most advanced use is checking for first to complete
And carrying on with something else if not

Using Non-Blocking (2)

- You can use them together with blocking forms

All reasonable combinations work as expected
E.g. non–blocking send and blocking receive

Use them only if you can start them ahead of time

- If you can't start them well in advance
  then use the simpler blocking forms

Also advanced uses for avoiding deadlock

- Generally, leave that sort of thing to experts

Using Non-Blocking (3)

All send variants have non–blocking forms

Includes MPI_Issend and MPI_Ibsend
They have potential, but obscure, uses

Easy to use – knowing when and why is hard
This course will not mention them further

Will cover only MPI_Isend and MPI_Irecv
Few programmers will want any of the other forms

Wait vs Test (1)

Blocking waits have names like MPI_Wait

Non−blocking waits have names like MPI_Test

There is only one difference between the two forms
which matters if the transfer is not ready

- Waits hang until the transfer has finished
- Tests return successfully and immediately

Tests have an extra Boolean flag variable
indicating whether the transfer has finished

Completion (1)

An active request is one that has started
but has not yet been completed

Requests are completed by two–step procedure:

- Become ready (i.e. finish transferring)
- A wait or test call returns their status

A request is released automatically
as part of the completion process
You almost never have to take any special action

Completion (2)

Wait and test work on send requests

- The status is largely meaningless (unset)

The extra lectures describe what it does mean

- The status does not include the arguments

That decision was taken on efficiency grounds

Completion (3)

Wait and test update the request
Upon release, it is set to MPI_REQUEST_NULL

- You rarely need to know or check that

But it is useful for some advanced uses
You can check if a request has been completed

Initialise requests to MPI_REQUEST_NULL

Usage

Very similar to blocking send and receive
Almost all arguments are used identically

- It really is just splitting the calls in two

A request is another opaque handle

MPI_Isend and MPI_Irecv return a request
and the latter does not return a status

MPI_Wait takes a request and returns a status

Fortran Example (1)

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

CALL MPI_Isend ( buffer , 100 ,       &
      MPI_DOUBLE_PRECISION ,  to , tag ,       &
      MPI_COMM_WORLD , request , error )

CALL MPI_Wait ( request , status , error )
```

Fortran Example (2)

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

CALL MPI_Irecv ( buffer , 100 ,       &
      MPI_DOUBLE_PRECISION ,  from , tag ,       &
      MPI_COMM_WORLD , request , error )

CALL MPI_Wait ( request , status , error )
```

C Example (1)

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

error = MPI_Isend ( buffer , 100 , MPI_DOUBLE ,
        to , tag , MPI_COMM_WORLD , & request ) ;

error = MPI_Wait ( & request , & status ) ;
```

C Example (2)

```c
double buffer [ 100 ] ;
MPI_Request request ;
MPI_Status status ;
int from = 2 , to = 3 , tag = 123 , error ;

error = MPI_Irecv ( buffer , 100 , MPI_DOUBLE ,
        from , tag , MPI_COMM_WORLD , & request ) ;

error = MPI_Wait ( & request , & status ) ;
```

Non-blocking Waits (1)

Remember MPI_Iprobe versus MPI_Probe?
That is also MPI_Test versus MPI_Wait

An extra Boolean argument saying if ready

- If ready, it behaves just like MPI_Wait
- If not, the request is not completed

And the status becomes undefined

Here are just the actual differences

Non-blocking Waits (2)

Fortran example:

```
LOGICAL :: flag
CALL MPI_Test ( request , flag , status , error )
```

C example:

```
int flag ;
error = MPI_Test ( & request , & flag , & status ) ;
```

Multiple Completion (1)

You can test or wait for an array of requests
Until one, all or some (not advised) complete

The functions are very difficult to teach
because there are so many special cases

- But they aren't hard to use, if you KISS

For now, we make the following assumptions

  - The array has length one or more
  - MPI_ERRORS_ARE_FATAL is set
  - You use only what I have covered so far

Multiple Completion (2)

They are simply shorthand for coding a loop
Though with some important optimisations

Behave exactly like the individual request forms

- The complexity is in explaining the details

Note that a Fortran status array is:

```
INTEGER , DIMENSION ( MPI_STATUS_SIZE , * )
```

Fortran first dimensions vary fastest

Waiting/Testing For All

These are easy to use, given our assumptions
They take arrays of requests and statuses

MPI_Testall and MPI_Waitall
These check for or complete all the requests

MPI_Waitall and when MPI_Testall's flag is True:
All of the statuses are set, appropriately
They are undefined when MPI_Testall's flag is False

Fortran Wait/Test For All

```
INTEGER :: i , error , requests ( 100 ) ,        &
      statuses ( MPI_STATUS_SIZE , 100 )
LOGICAL :: flag

DO i = 1 , 100
      CALL MPI_Irecv ( . . . ,        &
            MPI_COMM_WORLD , requests ( i ) , error )
END DO

CALL MPI_Waitall ( 100 , requests , statuses , error )
CALL MPI_Testall ( 100 , requests , flag ,        &
      statuses , error )
```

C Wait/Test For All

```
int i , error , flag ;
MPI_Request requests [ 100 ] ;
MPI_Status statuses [ 100 ] ;

for ( i = 1 ; i < 100 ; ++ i )
      error = MPI_Irecv ( . . . ,
            MPI_COMM_WORLD , & requests [ i ] ) ;

error = MPI_Waitall ( 100 , requests , statuses ) ;
error = MPI_Testall ( 100 , requests , & flag ,
      statuses ) ;
```

Waiting/Testing For Any

MPI_Testany and MPI_Waitany

These check for or complete only one request
and return its index and status

You often loop round until there is nothing to do

Stop when the index is MPI_UNDEFINED

The flag of MPI_Testany is True in that case

Fortran Wait for Any

```
INTEGER :: i , error , requests ( 100 ) ,        &
    index , status ( MPI_STATUS_SIZE )

DO
    CALL MPI_Waitany ( 100 , requests , index ,        &
        status , error )
    IF ( index == MPI_UNDEFINED ) EXIT
    ! Now handle transfer number index
END DO
```

Fortran Test for Any

```
INTEGER :: i , error , requests ( 100 ) ,        &
      index , status ( MPI_STATUS_SIZE )
LOGICAL :: flag

DO
      CALL MPI_Testany ( 100 , requests , index , flag ,        &
            status , error )
      IF ( .NOT. flag) THEN
            ! Do something while waiting
      ELIF ( index == MPI_UNDEFINED )
            EXIT
      ELSE
            ! Now handle transfer number index
      END IF
END DO
```

C Wait for Any

```
int i , error , index ;
MPI_Request requests [ 100 ] ;
MPI_Status status ;

while ( 1 ) {
        error = MPI_Waitany ( 100 , requests ,
                & index , & status ) ;
        if ( index == MPI_UNDEFINED ) break ;
        /* Now handle transfer number index */
}
```

C Test for Any

```
int i , error , index , flag ;
MPI_Request requests [ 100 ] ;
MPI_Status status ;

while ( 1 ) {
        error = MPI_Testany ( 100 , requests , & index ,
                & flag , & status ) ;
        if ( ! flag ) {
                /* Do  something while waiting */
        } else if ( index == MPI_UNDEFINED )
                break ;
        } else {
                /* Now handle transfer number index */
        }
}
```

Assumptions Revisited

Remember the assumptions I described?

- The array has length one or more
- MPI_ERRORS_ARE_FATAL is set
- You use only what I have covered so far

Another lecture covers when those are not so

- My advice is not to open that can of worms

Warning

Calling non–blocking functions is very easy

- Don’t be fooled into thinking that using them is

You now have a loaded, semi–automatic footgun . . .

The difficulties arise with race conditions etc.
Adding diagnostics often makes them vanish

Remember the aphorism: “Festina lente”
Don’t rush into asynchronous programming
Start with using it very simply
And package the uses into higher–level primitives