Message-Passing and MPI Programming

More on Point-to-Point

N.M. Maclaren
Computing Service

nmm1@cam.ac.uk
ext. 34761

July 2010

6.1 Introduction

These facilities are the most complicated so far, but you may well want to use them. There is a Latin aphorism "*festina lente*", which translates as "*make haste slowly*", and means do not trip over your own feet trying to rush ahead. Make sure that you understand the earlier lectures before starting to use this one.

We shall start with a couple of potentially useful minor features, which are not difficult. We shall then cover non-blocking transfers, which are easy to use, but not always easy to understand. However, they are correspondingly important and useful.

6.2 Sending to Oneself

A process can send a message to itself. That is not generally a good idea, though some of the practical examples and specimen answers do, mainly to demonstrate particular points. If you use blocking calls carelessly, and a process sends to itself, it will deadlock; the only safe and easy case is send-receive. For all other point-to-point transfers, you **must** use buffering **and** call the send before the receive – that is guaranteed to work. Otherwise, a process should send to itself **only** with non-blocking calls (covered in a moment), but care is still needed, even with those.

Consider when writing your own collectives. You can treat the local process separately, or you can use the whole communicator symmetrically. If you do the latter, processes will send to themselves. The obvious code will work in MPI, but only subject to the above constraints. You should do whatever makes your code cleanest.

6.3 Null Processes

You can specify a null source or destination. Sends and receives return immediately (or some approximation to that), successfully, and receives do not update the transfer buffer. This may enable you to simplify code at boundaries, and you should use the facility only if it clarifies your code.

Warning: be very careful if you use this with non-blocking transfers, because MPI is not entirely clear what happens in that case. It probably creates a normal request that is flagged as having completed, but does not say so explicitly.

To do this, use `MPI_PROC_NULL` as a process number (`MPI::PROC_NULL` in C++). The status will contain `MPI_PROC_NULL` (or possibly `MPI_ANY_SOURCE`) as the source and `MPI_ANY_TAG` as the tag; `MPI_Get_count` on it returns zero. The ambiguity in the source value is not important, because reasonable programs will not look at it, anyway.

6.4 Non-Blocking Transfers

These are also called asynchronous communication, and books and Web pages may use either term, or other variants. Old mainframe programmers know these are the best way to implement I/O, and books often describe these as more efficient. Unfortunately, all modern systems are synchronous at the hardware and network transfer levels, so most of the potential efficiency is not realised. This lecture will describe only how to use them, and not the implementation issues and consequent effects on tuning. A lot of it is describing what **not** to do, because experience with asynchronism is rare nowadays.

MPI non-blocking transfers are a very good example of classic asynchronous communication, so learning them is more general than for just using MPI. The basic logic is:

- The main call starts an asynchronous transfer, and returns a handle, called a request.
- Later, you wait on the request until finished; only then has the transfer completed.

In MPI, the wait frees the request and sets the status, and you rarely need to free the request yourself. You can also test whether a request is ready.

- The actual buffer update is anywhere in between those two calls (i.e. asynchronous to the process) and, indeed, bytes may change in a random order or more than once.

The buffer must not even be inspected during the window, except that pure send buffers are read-only objects and may be read. The buffer obviously must not move, so no reallocation or other manipulation is permitted. You need to take care in Fortran (covered later) and garbage collected languages, and C++ copy or move constructors (which includes when using the STL), and may need to play fancy games to stop that. Once the transfer has completed, you can use the buffer as normal.

The window of restriction is between initiating the send or receive and completing the wait for the request to complete. To summarise:

- For a non-blocking send, you must not update, reallocate or free the buffer in the window; reading the buffer does not cause problems.
- For a non-blocking receive, you must not access, reallocate or free the buffer in the window.

Chaos awaits if you break those rules, though it will often not show up in simple tests, so, regrettably, there are no practical exercises to demonstrate it. Non-blocking transfers are the **only** cause of race conditions causing wrong answers (rather than simple deadlock) in the subset of MPI that this course covers.

Fortran warning: you need to watch out for array copying, especially when using modern Fortran (e.g. assumed-shape arrays). That counts as a form of reallocation. For more details on this, see the lecture *Miscellaneous Guidelines* and the course *Introduction to Modern Fortran*, especially *Advanced Use Of Procedures.*

6.5 Using Non-Blocking Transfers

Generally, you should start them as soon as possible, and wait for completion only when you need the buffer. A more advanced use is waiting on several requests, and and dealing with them in the order they are ready. The most advanced use covered here is checking for the first to complete, and carrying on with something else if none have completed yet.

You can use non-blocking transfers together with the blocking forms – e.g. a non-blocking send can match a blocking receive – and all reasonable combinations work as expected. You should use them only if you can start them ahead of time – if you cannot start them well in advance, it is likely to be more efficient to use the simpler blocking forms.

There are also advanced uses for avoiding deadlock, though you should generally leave that sort of thing to experts. However, this course does describe the techniques, as there are circumstances when you may need to use them, such as for I/O (see later).

All of the send variants have non-blocking forms, including `MPI_Issend` and `MPI_Ibsend`, which have potential, but obscure, uses. They are easy to use, but knowing when and why is hard, and this course will not mention them further. It will cover only `MPI_Isend` and `MPI_Irecv`; few programmers will want any of the other forms.

6.6 Completion

Blocking waits have names like `MPI_Wait` and non-blocking have names like `MPI_Test`. These is only one difference between the two forms, which matters only if the transfer is not ready: waits hang until the transfer has finished, but tests return successfully and immediately. Obviously, tests have an extra Boolean flag variable indicating whether the transfer has finished (except in C++, as described later).

An active request is one that has started but has not yet been completed. Requests are completed by two-step procedure: they become ready (i.e. finish transferring), and then a wait or test call returns their status. A request is released automatically as part of the completion process, and you almost never have to take any special action.

Wait and test also work on send requests, but the status is largely meaningless (i.e. unset) – the few meanings it has are covered in the extra lectures. It does **not** include the arguments (e.g. destination and tag); that decision was taken on efficiency grounds.

Wait and test (when the transfer has finished, that is) 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, because you can check if a request has been completed. To do this reliably, you should also initialise requests to `MPI_REQUEST_NULL`; that is good practice.

6.7 Usage

Non-blocking send and receive are very similar to the blocking forms, and almost all arguments are used identically. It is just splitting the calls in two, using the request as an opaque handle to transfer data between the two calls; you do nothing with it. `MPI_Isend` and `MPI_Irecv` return a request, and the latter does **not** return a status. `MPI_Wait` and `MPI_Test` take a request and return a status.

Fortran Send:

```
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 Receive:

```
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 Send:

```
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 Receive:

```
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 ) ;
```

C++ has a slightly different syntax, and the request is the result of `MPI::Isend` and `MPI_Irecv`. `Wait` is a member function of the request class. You can omit the status argument for `Wait`, but I recommend doing that only for requests created by `MPI::Isend`, and you can provide it even for them.

C++ Send:

```
double buffer [ 100 ] ;
MPI::Request request ;
MPI::Status status ;
int from = 2 , to = 3 , tag = 123 ;

request = MPI::COMM_WORLD . Isend ( buffer , 100 ,
    MPI::DOUBLE , to , tag ) ;

request . Wait ( ) ;
```

C++ Receive:

```
double buffer [ 100 ] ;
MPI::Request request ;
MPI::Status status ;
int from = 2 , to = 3 , tag = 123 ;

request = MPI::COMM_WORLD . Irecv ( buffer , 100 ,
    MPI::DOUBLE , from , tag ) ;

request . Wait ( status ) ;
```

6.8 Wait versus Test

Remember `MPI_Iprobe` versus `MPI_Probe`? The difference is also the difference between `MPI_Test` and `MPI_Wait`. The former has an extra Boolean argument saying if the request is ready, which is returned as the function result in C++.

- If ready, it sets the flag to True, and sets the status (i.e. it behaves just like `MPI_Wait`).
- If not, the request is not completed, it sets the flag to False, and the status becomes undefined.

Here are just the actual differences that would be needed for the above examples – they include the declaration of the flag, and the test call that replaces the wait one.

Fortran:

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

C:

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

C++:

```
int flag ;
flag = request . Test ( status ) ;
```

6.9 Multiple Completion

You can test or wait for an array of requests until one, all or some complete. The last form is omitted here, because it is more complicated and generally not advised. The functions are very difficult to teach because there are so many special cases, but they are not 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 the facilities that the course has covered so far.

Multiple completions are simply shorthand for coding a loop, though with some important optimisations. They behave exactly like the individual request forms, and the only complexity is in explaining the details.

In Fortran, unlike C and C++, a status is already an array. An array of statuses in Fortran is a two-dimensional array, with the second dimension indexing which status (remember that Fortran first dimensions vary fastest). For example:

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

6.10 Waiting and Testing for All

These are easy to use, given our assumptions, and are almost a shorthand for a loop that waits or tests each request in turn. They take arrays of requests and statuses and check for or complete all the requests. The functions are called `MPI_Testall` and `MPI_Waitall`.

When `MPI_Waitall` and when `MPI_Testall`'s flag is True, all of the statuses are set, appropriately. The statuses all become undefined when `MPI_Testall`'s flag is False; they may well end up being full of random rubbish.

Fortran:

```
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:

```
        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 ) ;
```

C++:

```
        int i , flag ;
        MPI::Request requests [ 100 ] ;
        MPI::Status statuses [ 100 ] ;

        for ( i = 1 ; i < 100 ; ++ i )
            MPI::COMM_WORLD . Irecv ( . . . , requests ( i ) ) ;

        requests[0] . Waitall ( 100 , requests , statuses ) ;
        flag = requests[0] . Testall ( 100 , requests , statuses ) ;
```

Note that even these are class methods in C++, and so you have to attach them to a request – any request will do, even one that is not part of the array and not active, but the usage above is about the cleanest way of doing it. That usage is an example of why I dislike object orientation when it is made compulsory.

6.11 Waiting and Testing for Any

These are not much harder to use, given our assumptions. They take arrays of requests and statuses, check for or complete one of the requests and return its index and status. If more than one request is ready, an arbitrary one is selected. The functions are called `MPI_Testany` and `MPI_Waitany`. The status is undefined when `MPI_Testany`'s flag is False.

A common way of using these is to loop round until there is nothing to do; MPI's specification simplifies this. If there are no active requests in the array, it will return an index of `MPI_UNDEFINED` and an empty status (for now, treat it as undefined). The flag of `MPI_Testany` is True in that case.

Fortran:

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

C:

```c
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 */
        error = MPI_Waitany ( 100 , requests ,
            & index , & status ) ;
    }
    if ( index == MPI_UNDEFINED ) break ;
    /* Now handle requests [ index ] */
}
```

The C++ interfaces are rather irregular: `MPI::Waitany` returns the index, but `MPI::Testany` returns the flag and the index is returned as for C and Fortran.

C++:

```
int i , requests [ 100 ] , index , flag ;
MPI::Request requests [ 100 ] ;
MPI::Status status ;

while ( 1 ) {
    flag = requests[0] . Testany ( 100 , requests ,
        index , status ) ;
    if ( ! flag ) {
        // Do  something while waiting
        index = requests[0] . Waitany ( 100 ,
            requests , status ) ;
    }
    if ( index == MPI::UNDEFINED ) break ;
    // Now handle requests [ index ]
}
```

6.12 Reminders

Remember the assumptions described earlier?

- The array has length one or more.
- `MPI_ERRORS_ARE_FATAL` is set.
- You use only the facilities that the course has covered so far.

An extra lecture covers when those are **not** so, but you are recommended not to open that can of worms. MPI's specification is clean and well-designed, but the problem is inherently complicated, and so the specification is, too.

And, lastly, calling non-blocking functions is very easy; do not be fooled into thinking that using them is. You now have a loaded, semi-automatic footgun ...

The difficulties arise with race conditions and all that they imply; adding diagnostics often makes them vanish, and they can be diabolically difficult to track down. Remember the aphorism "*festina lente*" – do not rush into asynchronous programming, and start by using it **very** simply, and package the uses into higher-level primitives.