Message-Passing and MPI Programming

Point-to-Point Transfers

N.M. Maclaren

nmm1@cam.ac.uk

July 2010

4.1 Introduction

Most books and courses teach point-to-point first, and then follow up by teaching collectives. This course has not done that, and you may well ask why not. The reason is that point-to-point is hard to use correctly; I usually make a complete mess of it, first time, and have to spend some time debugging. C.A.R. Hoare's "`Communicating Sequential Processes`" is one of the seminal works on point-to-point message passing. In the light of his experience teaching point-to-point parallelism, Hoare designed the very much simpler BSP! It is reasonable to regard point-to-point as the assembler-level interface of message passing, and who programs in assembler nowadays? The following rules are essential:

- Above all, *KISS – Keep It Simple and Stupid*

According to Wikipedia, that acronym was first coined by Kelly Johnson, lead engineer at the Lockheed Skunk Works, and it means that the simplest design is the best – i.e. fastest and easiest to get working and most reliable in use. That is an engineering principle that has proven to be reliable since prehistory! Hoare has also coined a couple of aphorisms along the same lines. Also:

- Design proper primitives, do not just code

Work out what your code needs to do, and design your own composite communication primitive, with a clean specification. You can then debug that on its own, and use it as if it were a built-in facility.

It is simplest to use one of two design models for your primitives:

Your own collective
Two processes, doing nothing else

- It is easiest if your primitives do not overlap

If you can separate your primitives by barriers, you can debug their use separately, and not worry about one primitive interfering with another. It is almost essential for tuning, as is mentioned later.

4.2 Envelopes and Status

You can think of point-to-point as a sort of Email – just like that (and traditional paper mail), messages come in envelopes with addressing information on them. MPI's envelopes contain the following:

- The source process
- The communicator

- An identifying tag

One of the first two is the calling process, because it is either the sender or the receiver, and the others are specified in the arguments. Note that they do **not** contain the destination address (or, rather, it is not in the part of the envelope shown to the MPI programmer).

A receive action returns a status, and that contains the following:

- The source process
- The identifying tag
- Other, hidden, information

The receive call already knows the communicator and destination, if you think about it. There is a function to extract the message size from the hidden information.

In C, the status is a structure with a typedef name MPI_Status. In Fortran, it is a default integer array of size MPI_STATUS_SIZE. You declare these yourself, as normal, including in static memory or on the stack. They are **not** like communicators, and you do **not** call MPI to allocate and free them.

For now, you can largely ignore the status, because you do not need to look at it for very simple use. In MPI 1, you had to provide the argument, and this is the form that I shall use in examples; MPI 2 allowed you to not provide it, but I do not recommend doing that, in general.

4.3 The Simplest Use

Let us assume that the communicator is MPI_COMM_WORLD. The tag is needed only for advanced use, though it is quite useful for added checking. So we need specify only the destination and source; but the latter is set automatically for send, and the former is for receive! The functions are MPI_Send and MPI_Recv.

Fortran:

```
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:

```
        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 ) ;

        double buffer [ 100 ] ;
        MPI_Status status ;
        int myrank , from = 2 , to = 3 , tag = 123 , error ;
        error = MPI_Rank ( & myrank ) ;
        if ( myrank == from )
            error = MPI_Recv ( buffer , 100 , MPI_DOUBLE ,
                from , tag , MPI_COMM_WORLD , & status ) ;
```

Trivial as the above is, it is enough to cause trouble. There are some examples on how that can happen, but simple examples do not really show the potential consequences. And it is not enough for all real programs; MPI provides lots of knobs, bells and whistles, so that almost every rasonable requirement is catered for. However, you should use only what you need, and you should not use something just because it looks cool.

In general, you need only to know what can be done; when you need something extra, then you can look it up.

4.4 Blocking

A receive call will block until a matching send; if one is never posted, it will hang indefinitely. Send may block until the message is received, or it may copy the message and return immediately; in that case, MPI will send it on in due course. This is unspecified and entirely up to the implementation; it may vary between messages, or with the phase of the moon. Correct MPI programs will work whichever approach the implementation takes. However, you can control it yourself, as described later.

Let us say that processes A and B want to swap data, so both send the existing value, and then both receive it. That will sometimes work and sometimes hang.

Process A	Process B
send to B	send to A
Both *may* now wait until transfers received but need not do so	
receive from B	receive from A

In that case, it is trivial to avoid, by a simple test on the process numbers. If A is less than B, A sends first and receives second, and B receives first and sends second; and conversely if A is greater than B. However, is is easy to get wrong when extending this to more complicated transfer graphs. MPI provides several ways to avoid the problem, so use whichever is simplest for your purposes.

You can specify the transfer mode explicitly. Generally, it is better not to do that, though the examples ask you to, in order to expose problems with using MPI incorrectly that otherwise might remain hidden. In real programs, both have important, but advanced, uses.

- `MPI_Ssend` is synchronous (i.e. it **will** block) and returns only when the message has started to be received. The swap example above will hang, predictably.
- `MPI_Bsend` is buffered (i.e. it copies the message and will **not** block). The swap example above will never hang.

These have exactly the same usage as `MPI_Send`, which is actually implemented by calling one of these. Because the implementation can choose which is likely to be more efficient, using any information that it has (including such things as the network load), using plain `MPI_Send` is recommended.

A synchronous send avoids a separate handshake, because completing the call acknowledges receipt. This can be simpler (and hence more efficient) in some programs, so a good rule is to use it if it avoids an explicit acknowledgement, and not otherwise.

Buffering is more tricky, surprisingly enough, because sends are erroneous if the buffer becomes full, and the default buffer size is zero. But, because exceeding it is undefined behaviour, implementations are permitted (but not required) to use a non-zero default buffer size! If you find that confusing, do not worry. Using buffering is covered later.

4.5 Composite Send and Receive

There is a composite send and receive, which will do them in the right order to avoid deadlock. It can also match ordinary send and receive, so one process can use it and another use separate ones (in either order). It also has a form that updates the buffers in place, by logically sending the buffer and then receiving into it. That may involve extra copying, of course. Use these if they are what you want to do; they are not likely to be any more efficient.

Fortran:

```
REAL(KIND=KIND(0.0D0)) ::    &
    putbuf ( 100 ) , getbuf ( 100 ) , buffer ( 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 )

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

C:

```
    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 ) ;
```

4.6 Unknown Message Sizes

The send and receive sizes need not match, but it is an error if the receive is smaller (i.e. there is not enough space for the message). If the send 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 in the receive buffer. However, there is a better way to do this, which allows receiving truly unknown size messages, and this is where you start to use the status.

You can accept the message with `MPI_Probe` – calling it that is a bit of a misnomer, and the name `MPI_Accept` would make more sense, because it accepts the message and updates the status, but it does not transfer the data anywhere (i.e. it leaves it in the 'mailroom').

You can discover the size with `MPI_Get_count`, and then you can allocate a suitable buffer. Note that `MPI_Get_count` needs the datatype, which; that allows for conversion, which is not covered in this course.

Lastly, you receive the message as normal.

Fortran:

```
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 ,    &
    from , tag , MPI_COMM_WORLD , status )
```

C:

```
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 ) ;
```

4.7 Checking for Messages

The real probe function is called MPI_Iprobe, and it returns immediately even if there is no matching message. It has an extra Boolean argument saying if there is a matching message. If there is a message, it behaves just like MPI_Probe; if there is not, the status is not updated. It is so similar, I shall show only the actual differences.

Fortran:

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

C:

```
int flag ;
error = MPI_Iprobe ( from , tag ,
        MPI_COMM_WORLD , & flag , & status ) ;
```

After these, flag is True and status is set if there is a message, and flag is False and status is unchanged if not.

4.8 Wild Cards etc.

You can accept messages from any process – just use MPI_ANY_SOURCE for the source argument (from in the above examples). The actual source (i.e. identifier of sending process) is stored in the status using the name MPI_SOURCE. I.e. in C, it is in status.MPI_SOURCE; in Fortran, it is in status(MPI_SOURCE).

- Be warned – your footgun is now loaded.

Similarly, you can accept messages with any tag – just use MPI_ANY_TAG for the tag argument. You extract the tag from the status in the same way as the source, but using the name MPI_TAG instead of MPI_SOURCE.

One of the best uses of tags is for for cross-checking, to ensure that the message you received is the one you expected and not some extraneous one that you have just picked up. As an aside, the lack of such checking on events is a common cause of incorrect behaviour in GUIs (windowing systems).

- It could be a message sequence number
- Or identify the object being transferred
- Anything else that would help with debugging

On receipt, the program checks that it is expected and, if not, produces an appropriate diagnostic (ie.g. "How did **this** packet get **here**?"). And, of course, the diagnostics can include as much program state as you want, unlike the default ones produced by MPI.

Fortran:

```
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 , MPI_DOUBLE_PRECISION ,    &
    count , error )
from = status ( MPI_SOURCE )
tag = status ( MPI_TAG )
```

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 ;
```

MPI specifies that each process has *FIFO* receipt (i.e. it maintains a queue of messages in the order they arrived). Incoming messages never overtake each other; every probe and receive matches in queue order and returns first message that satisfies **all** of the constraints.

So probe and receive get same message if:

- There has been no intervening receive
- Same communicator, source and tag

Note that using wild cards for the source in both the probe and receive counts as using the same source, and similarly for the tag. There are other safe usages, too, but that one is both easy to describe and easy to use correctly. You can also use wild cards in the probe, extract the source and tag from the status, and then use *those values* in the receive.

Also, if process **A** send multiple messages to process **B**, those messages will arrive in the order they were sent. No ordering is defined between the arrival of messages if either the sender or the receiver differ, and messages can get delayed considerably on some systems.

The main purpose of tags is not for checking. MPI provided them to allow independent communication paths, many books and Web pages will describe that use, and some will even encourage it. **Do not do it**. It is the equivalent of cocking your footgun, because using tags like that is very hard even for experts. As with all rules, exceptional problems may require you to break them, and this course gives an example later where it can be useful for I/O.

4.9 Buffered Sends

These are trivial to use, but need extra mechanism. This is one of the very few areas where the MPI standard is a bit of a mess, and so it is important to follow the rules, however unreasonable they seem.

As mentioned previously, the default buffer is effectively implementation dependent, and does not even have to be documented – IBM chose to use 8 bytes for *poe*! So you **have** to allocate a buffer first – it is just a block of memory, and any type will do. That is really the only extra complexity, and you can usually just make the buffer very big.

You attach a single buffer to a process – **not** a communicator – which is different from other, similar MPI options. When you have finished doing buffered transfers, you should detach it, and it may be used for buffering by MPI in between. Generally, attaching it immediately after calling `MPI_Init` and detaching it immediately before calling `MPI_Finalize` is good practice.

Detach returns the values previously stored, but it is unclear whether you need to set the values before the call, so it is safer to do so. It costs almost nothing.

You must not access it **in any way**, including using it for workspace when there are not outstanding buffered sends, or looking at its contents, because its use and contents are completely undefined. Be careful in garbage-collected languages, because you need to make sure that the buffer will not move around. Even in Fortran, C and C++, make sure that it does not go out of scope or Fortran copy-in/copy-out of arguments.

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 )
```

Returning the previously stored value makes sense for the old size, but it is completely baffling as to what it might mean for `buffer`! That makes no sense in terms of Fortran's rules for argument passing.

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, to return the values.

Using buffered sends is rarely advisable, because they usually hide problems rather than fixing them, and can be quite a lot less efficient. However, if you have a completely baffling deadlock, try changing all sends to buffered; if that helps, you have a race condition, so track it down and fix it properly. The other main use is for I/O and similar out-of-band communication, which is described later.

You can calculate how much space you need for buffering, but doing that is overkill for almost all programs, and this course does not describe how. You should generally just do a back of an envelope calculation, and allocate a buffer that is significantly larger than you need – nowadays, wasting a few megabytes is not a problem. The relevant facilities for space calculation are:

Constant:	`MPI_BSEND_OVERHEAD`	
Function:	`MPI_Pack_size`	
Function:	`MPI_Sizeof`	(Fortran only)

4.10 Epilogue

There is more on point-to-point later, mainly on non-blocking (i.e. asynchronous) transfers. But we have covered most of what MPI provides for blocking transfers, and the exercises will try out quite a lot of this. The main one is to code a rotation collective; each process sends to its successor, and the last one sends back to the beginning. That is a practical example of one way that you are recommended to use point-to-point transfers.

The practical exercises often state that you should use buffered or synchronous sends, and the reason is to expose or hide cases of deadlock, so that certain errors show up predictably and other ones that would merely be confusing do not show up. To remind you, this is advised **only** when testing, and you should normally use ordinary sends, on the grounds of efficiency.