Tutorial on MPI: The Message-Passing Interface

William Gropp

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
gropp@mcs.anl.gov
Buffering issues

Where does data go when you send it? One possibility is:

A: [Diagram showing data flow from Process 1 to Process 2 through Local Buffer and The Network]

B: [Diagram showing data flow from Process 2 to Process 1 through Local Buffer]
This is not very efficient. There are three copies in addition to the exchange of data between processes. We prefer

But this requires that either that MPI_Send not return until the data has been delivered or that we allow a send operation to return before completing the transfer. In this case, we need to test for completion later.
Blocking and Non-Blocking communication

- So far we have used blocking communication:
  - `MPI_Send` does not complete until buffer is empty (available for reuse).
  - `MPI_Recv` does not complete until buffer is full (available for use).

- Simple, but can be “unsafe”:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(1)</td>
<td>Send(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Recv(0)</td>
</tr>
</tbody>
</table>

Completion depends in general on size of message and amount of system buffering.

*Send works for small enough messages but fails when messages get too large. Too large ranges from zero bytes to 100’s of Megabytes.*
Some Solutions to the “Unsafe” Problem

- Order the operations more carefully:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send(1)</td>
<td>Recv(0)</td>
</tr>
<tr>
<td>Recv(1)</td>
<td>Send(0)</td>
</tr>
</tbody>
</table>

- Supply receive buffer at same time as send, with `MPI_Sendrecv`:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sendrecv(1)</td>
<td>Sendrecv(0)</td>
</tr>
</tbody>
</table>

- Use non-blocking operations:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isend(1)</td>
<td>Isend(0)</td>
</tr>
<tr>
<td>Irecv(1)</td>
<td>Irecv(0)</td>
</tr>
<tr>
<td>Waitall</td>
<td>Waitall</td>
</tr>
</tbody>
</table>

- Use `MPI_Bsend`
Non-blocking operations return (immediately) “request handles” that can be waited on and queried:

- MPI_Isend(start, count, datatype, dest, tag, comm, request)
- MPI_Irecv(start, count, datatype, dest, tag, comm, request)
- MPI_Wait(request, status)

One can also test without waiting: MPI_Test(request, flag, status)
Multiple completions

It is often desirable to wait on multiple requests. An example is a master/slave program, where the master waits for one or more slaves to send it a message.

- `MPI_Waitall(count, array_of_requests, array_of_statuses)`

- `MPI_Waitany(count, array_of_requests, index, status)`

- `MPI_Waitsome(incount, array_of_requests, outcount, array_of_indices, array_of_statuses)`

There are corresponding versions of test for each of these.

> The `MPI_Waitsome` and `MPI_Waitsome` may be used to implement master/slave algorithms that provide fair access to the master by the slaves.
Fairness

What happens with this program:

```c
#include "mpi.h"
#include <stdio.h>
int main(argc, argv)
  int argc;
  char **argv;
{
  int rank, size, i, buf[1];
  MPI_Status status;

  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  if (rank == 0) {
    for (i=0; i<100*size-1; i++) {
      MPI_Recv( buf, 1, MPI_INT, MPI_ANY_SOURCE,
                MPI_ANY_TAG, MPI_COMM_WORLD, &status );
      printf( "Msg from %d with tag %d\n",
              status.MPI_SOURCE, status.MPI_TAG );
    }
  }
  else {
    for (i=0; i<100; i++)
      MPI_Send( buf, 1, MPI_INT, 0, i, MPI_COMM_WORLD );
  }
  MPI_Finalize();
  return 0;
}
```
Fairness in message-passing

An parallel algorithm is *fair* if no process is effectively ignored. In the preceding program, processes with low rank (like process zero) may be the only one whose messages are received.

MPI makes no guarantees about fairness. However, MPI makes it possible to write efficient, fair programs.
Providing Fairness

One alternative is

```c
#define large 128
MPI_Request requests[large];
MPI_Status statuses[large];
int indices[large];
int buf[large];
for (i=1; i<size; i++)
    MPI_Irecv( buf+i, 1, MPI_INT, i,
               MPI_ANY_TAG, MPI_COMM_WORLD, &requests[i-1] );
while(not done) {
    MPI_Waitsome( size-1, requests, &ndone, indices, statuses );
    for (i=0; i<ndone; i++) {
        j = indices[i];
        printf( "Msg from %d with tag %d\n",
                statuses[i].MPI_SOURCE,
                statuses[i].MPI_TAG );
        MPI_Irecv( buf+j, 1, MPI_INT, j,
                   MPI_ANY_TAG, MPI_COMM_WORLD, &requests[j] );
    }
}
```
Providing Fairness (Fortran)

One alternative is

```fortran
parameter( large = 128 )
integer requests(large);
integer statuses(MPI_STATUS_SIZE,large);
integer indices(large);
integer buf(large);
logical done
do 10 i = 1, size-1
10 call MPI_Irecv( buf(i), 1, MPI_INTEGER, i,
  *      MPI_ANY_TAG, MPI_COMM_WORLD, requests(i), ierr )
20 if (.not. done) then
   call MPI_Waitsome( size-1, requests, ndone,
      indices, statuses, ierr )
   do 30 i=1, ndone
      j = indices(i)
      print *, 'Msg from ', statuses(MPI_SOURCE,i), ' with tag',
      *      statuses(MPI_TAG,i)
      call MPI_Irecv( buf(j), 1, MPI_INTEGER, j,
        MPI_ANY_TAG, MPI_COMM_WORLD, requests(j), ierr )
      done = ...
30  continue
    goto 20
endif
```
Exercise - Fairness

Objective: Use nonblocking communications
Complete the program fragment on
‘providing fairness’. Make sure that you
leave no uncompleted requests. How would
you test your program?
More on nonblocking communication

In applications where the time to send data between processes is large, it is often helpful to cause communication and computation to overlap. This can easily be done with MPI’s non-blocking routines.

For example, in a 2-D finite difference mesh, moving data needed for the boundaries can be done at the same time as computation on the interior.

```c
MPI_Irecv( ... each ghost edge ... );
MPI_Isend( ... data for each ghost edge ... );
... compute on interior
while (still some uncompleted requests) {
    MPI_Waitany( ... requests ... )
    if (request is a receive)
        ... compute on that edge ...
}
```

Note that we call MPI_Waitany several times. This exploits the fact that after a request is satisfied, it is set to MPI_REQUEST_NULL, and that this is a valid request object to the wait and test routines.
Communication Modes

MPI provides multiple modes for sending messages:

- **Synchronous mode** (**MPI_Send**): the send does not complete until a matching receive has begun. (Unsafe programs become incorrect and usually deadlock within an **MPI_Send**.)

- **Buffered mode** (**MPI_Bsend**): the user supplies the buffer to system for its use. (User supplies enough memory to make unsafe program safe).

- **Ready mode** (**MPI_Rsend**): user guarantees that matching receive has been posted.
  - allows access to fast protocols
  - undefined behavior if the matching receive is not posted

Non-blocking versions:
**MPI_Isend, MPI_Irsend, MPI_Ibsend**

Note that an **MPI_Recv** may receive messages sent with any send mode.
Buffered Send

MPI provides a send routine that may be used when MPI_Isend is awkward to use (e.g., lots of small messages).

MPI_Bsend makes use of a user-provided buffer to save any messages that can not be immediately sent.

```c
int bufsize;
char *buf = malloc(bufsize);
MPI_Buffer_attach( buf, bufsize );
...
MPI_Bsend( ... same as MPI_Send ... );
...
MPI_Buffer_detach( &buf, &bufsize );
```

The MPI_Buffer_detach call does not complete until all messages are sent.

The performance of MPI_Bsend depends on the implementation of MPI and may also depend on the size of the message. For example, making a message one byte longer may cause a significant drop in performance.
Reusing the same buffer

Consider a loop

```c
MPI_Buffer_attach( buf, bufsize );
while (!done) {
    ...
    MPI_Bsend( ... );
}
```

where the `buf` is large enough to hold the message in the `MPI_Bsend`. This code may fail because the

```c
{  
    void *buf; int bufsize;
    MPI_Buffer_detach( &buf, &bufsize );
    MPI_Buffer_attach( buf, bufsize );
}
```