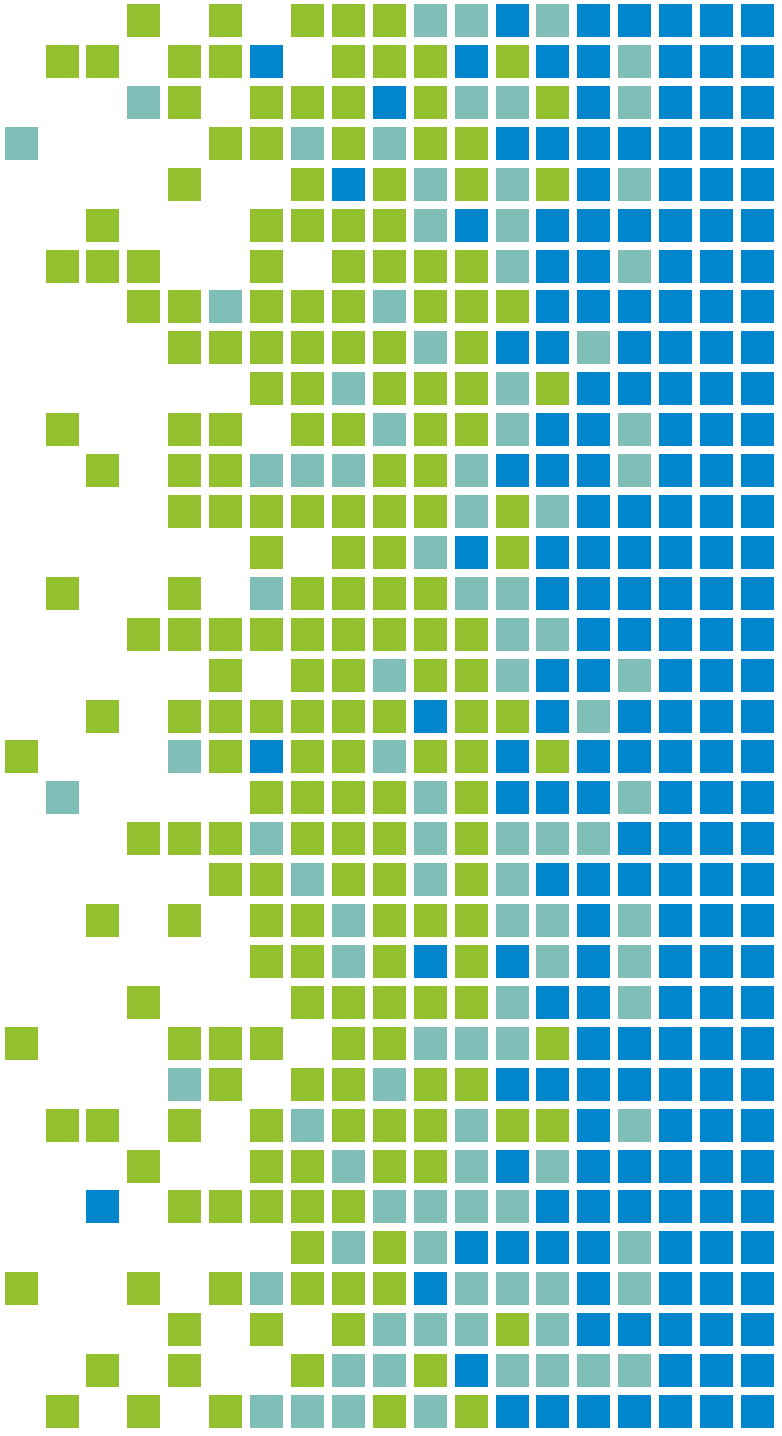
ICHEC
Irish Centre for High-End Computing

PRACE Course: Intermediate MPI

9-11 November 2022

Persistent + Packed Communications



Persistent Communications

They are a type of non-blocking communication.

We will cover just point-to-point operations.

Can be used when they are multiple messages of the same type being sent and received.

Can be optimal over using non-blocking comms that we have already seen when called many times.

For instance if a message is being sent each time within a inner loop.

Potentially overheads are reduced because handles are not created and destroyed each time.

They can be used with standard, buffered, synchronous or ready point-to-point routines.

Both sides do not need to be persistent.

There is a four stage process to using persistent communications.

Four Stage Process

Creation of handles/channel,

this is only done once no matter how many messages are sent/received,
no messages are actually sent/received.

Sending/receiving messages .

Checking the operations have completed. Remember they are non-blocking.

Destroying or freeing the handles,

this is done only once.

Create Handle

The same as the other point-to-point modes, there are four send modes and one receive mode:

- Standard send

```
MPI_Send_init(*buf,length,type,dest,tag,comm,*request
)
```

- Synchronous Send

```
MPI_Ssend_init(*buf,length,type,dest,tag,comm,*reques
t)
```

- Buffered Send

```
MPI_Bsend_init(*buf,length,type,dest,tag,comm,*reques
t)
```

- Ready Send

```
MPI_Rsend_init(*buf,length,type,dest,tag,comm,*reques
t)
```

- Receive

```
MPI_Recv_init(*buf,length,type,source,tag,comm,*reque
st)
```

You can see they are the same as the non-blocking equivalents.

Send/Receive Messages

The same routine is used to actually send and receive messages.

The request handle is all that is needed.

Single message

```
MPI_Start(*request)
```

All the persistent communications created for a rank

```
MPI_Startall(num_of_requests,request_array)
```

Then MPI_Wait or MPI_Test are needed to ensure that the messages have actually been sent or arrived.

This is the same mechanism as non-blocking communication.

Destroy Handle

To destroy the handle for a persistent communication use the routine below.

All

```
MPI_Request_free(*request)
```

After this you cannot use the persistent communication and must setup a new handle.

Example

Below is an example of a message sent from rank 0->1

```
int i,msg
MPI_Request recvobj;
MPI_Status status;

MPI_Recv_init(&msg,1,MPI_INT,0,0,MPI_COMM_WORLD,&recvobj);

for (i=0; i<1000; i++)
{
   if (rank==0) MPI_Send(&msg,1,MPI_INT,1,0,MPI_COMM_WORLD);
   if (rank==1)
   {
      MPI_Start(&recvobj);
      MPI_Wait(&recvobj,&status);
   }
}

MPI_Request_free(&recvobj);
```

Packing

Sending small messages is inefficient when the latency is high.

Persistent communications reduces some overhead but cannot mitigate entirely high latency.

Smaller messages can be packed together into a larger one.

There are specific MPI routines to pack messages MPI_Pack and to upack them MPI_Unpack.

The packed message can be sent and received like any other message.

The message however cannot be larger than 4GB.

It is similar to sending structs but is more general.

MPI_Pack

MPI_Pack is called multiple times packing the `buffer` into `outbuf`.

The `buffer` contains `n_in` variables of the same `type`.

Syntax.

```
MPI_Pack(*buffer,n_in,type,*outbuf,
outsize,*position,Comm)
```

The size of the `outbuf`, in bytes, is defined by an integer `outsize`, hence the limit on the size.

Then `position` controls where buffer is placed in `outbuf`.

Conveniently the value is updated each time it is called.

The programmer is responsible for making sure the buffers are large enough.

MPI_Unpack

Similarly when unpacking the message MPI_Unpack is called multiple times.

`buffer` contains the packed message of size `buf_siz` (in bytes).

`outbuf` contains a piece of the unpacked message with `n_out` elements of the same `type`.

Syntax.

```
MPI_Unpack(*buffer,buf_siz,*position,*outbuf,
n_out, type,Comm)
```

Again `position` controls the buffer address, starts at 0 and is updated each time a part of the message is unpacked.

The programmer is responsible for making sure the buffers are large enough.

MPI_Pack_size

How can the size of the buffer for the packed message be calculated?

MPI_Pack_size is useful in determining the size of MPI data types (in bytes).

For the basic data types (MPI_INT, MPI_FLOAT) they the same size as the corresponding lanuguage types (int, float).

As we have seen this may not be the case for derived data types.

It is good practice to always use this routine when packing messages.

It returns the size of a `n_in` variables of type `type`.

Syntax.

```
MPI_Pack_size(n_in,type,Comm,*size)
```

Example

Example of sending a packed message in a ring.

Move to Kay.

Practical 8

Given the code stub add the code to perform communication in a ring using persistent communications.

Note that we are sending the same set of messages around the ring multiple times.

This is set by the input argument.

If this is too easy,

- create two groups
- Have both perform a message in a ring separately
- Create an intercommunicator and send the resulting message of each group to the other
- At the end all ranks in each group should have the combined result of both final messages