ICHEC
Irish Centre for High-End Computing

PRACE Course: Intermediate MPI

9-11 November 2022

One-sided Communications

Introduction

Normally when we perform point-to-point communications then both sender and receiver need to be involved.

This is still true for collective communications.

With blocking communications this can leave either the sender or receiver waiting for the operation to complete.

This can be minimized with non-blocking communications but comes at a price with more complexity.

However there is one big drawback in that the receiver needs to know how many messages it will receive.

One-sided communications (OS comms) are where only one rank performs the operation.

Memory Windows

We are not going to cover everything in OS comms just the basics.

To setup OS comms we need to create a designated handle.

This needs to be associated with actual buffer.

The buffer can be associated/reserved before or after the handle is created.

This buffer or memory window is now visible to the other ranks.

After this then any rank (origin) in the communicator can read and write from any other (target) without the target's knowledge.

Windows do not have to be the same size of every rank.

Finally the handle is destroyed after OS comms have finished.

Create Handle I

If the buffer has been reserved beforehand then we can use the function below.

```
    MPI_Win_create(*buf,size,disp_unit,info,comm,*win)
```

The handle (`win`) is passed out and a special type `MPI_Win`.

The space for `buf` must be reserved beforehand with a size>=`size`.

`size` is in bytes but with type `MPI_Aint`.

There is no data type given but `disp_unit` can be >1 to ease pointer arithmetic.

The `info` argument passes optimization hints at runtime.

For this course we will use `MPI_INFO_NULL`.

Create Handle II

If the buffer has not been reserved beforehand then we can use the function below.

```
MPI_Win_create_dynamic(info,comm,*win)
```

The arguments are the same as before except of course the buffer is missing.

The buffer can be dynamically allocated afterwards but before OS comms can start the buffer must be associated or attached to the Window.

For this use `MPI_Win_attach`.

Create Handle III

The final method reserves the space for the buffer and associates it with the window handle.

```
MPI_Win_allocate(size,disp_unit,info,comm,*buf,*win
)
```

The arguments are the same as before.

What is returned is the pointer to the buffer and the window handle.

Destroy Handle

To destroy the Window handle use

```
    MPI_Win_free(*win)
```

This will also free the buffer when using `MPI_Win_allocate`.

If the buffer has been attached manually it should be detached before freeing the window.

```
    MPI_Win_detach(win,*buf)
```

Get and Put

Once we have setup the conditions we can start to send (put) and receive (get) data.

We will return later to how we synchronize all the ranks involved.

The get function then takes data from a Window and stores it locally.

```
MPI_Get(*obuf,ocount,otype,target,tdisp,tcount,ttyp
e,win)
```

Notice that there is no communicator but a window.

Similar to the other functions a datatype is needed.

We must also state where in the Window the target data is stored.

This is done via the displacement (`tdisp`) from the start of the Window address.

Get and Put

MPI_Put then takes local data and puts it into the Window of the target rank.

You can see that the function arguments are the same as Get.

MPI_Put:

```
MPI_Put(*obuf,ocount,otype,target,tdisp,tcount,ttyp
e,win)
```

Both put and get allow us to have different origin and target datatypes.

This would allow a generic buffer to be created where different datatypes could be stored.

Accumuluate

`MPI_Accumulate` is the OS comms equivalent of MPI_Reduce.

This function takes the data from the origin and puts it in the target Window.

Normally the target buffer would be overwritten but this function will perform an operation using the old value and the one being sent.

MPI_Accumulate:

```
MPI_Accumulate(*obuf,ocount,otype,target,tdisp,tcou
nt,ttype,op,win)
```

For this function the datatypes must be the same for origin and target.

Synchronisation

The target is not informed about the number nor status of the data transfers.

Thus explicit synchronization is needed before Window can be used or modified.

There are several synchronization functions, we will examine just one.

MPI_Win_fence:

```
MPI_Win_fence(assert,win)
```

This acts like a barrier in that all ranks in the OS comms must reach the fence before continuing.

Guarantees that all the local/remote operations are complete.

Caching is also complete.

Win_fence

The Win_fence synchronization brackets the set of OS comms operations that are independent and is called an epoch.

The `assert` argument can be used to optimize the communication.

Zero or "no assumptions" can always be used for `assert`.

MPI_MODE_NOSTORE: local window was not updated by local saves or local gets since last fence.

MPI_MODE_NOPUT: local window will not be updated by put or accumulate after the fence.

MPI_MODE_NOPRECEDE: fence does not complete local RMA ops. All ranks must set.

MPI_MODE_NOSUCCEED: fence does not start local RMA calls. All ranks must set.

Examples

There are two examples both use the message in a ring.

One example is that all the right neighbours gets the message from the Window of the left.

Second one all the left neighbours puts the message into the Window of the right.

Move to Kay.

Practical 9

Use the stub code and change the message in a ring to use MPI_Accumulate.

It is not strictly sending a message around the ring.

However goal is the same, all ranks should have the sum of the ranks.

Also change the way in which the window is being created.

If that was too easy then complete the previous two-rings problem.