

Remote Memory Access

Getting started with RMA

Reusing this material



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

http://creativecommons.org/licenses/by-nc-sa/4.0/deed.en_US

This means you are free to copy and redistribute the material and adapt and build on the material under the following terms: You must give appropriate credit, provide a link to the license and indicate if changes were made. If you adapt or build on the material you must distribute your work under the same license as the original.

Note that this presentation contains images owned by others. Please seek their permission before reusing these images.

Outline

- MPI RMA Basic Concepts
 - Why RMA?
 - Terminology
 - Program flow
- Getting started with RMA
 - Management of windows
 - Fence synchronization
 - Moving data around
- Practical
 - Modifying P2P code to use RMA

MPI RMA Concepts

Why RMA

- One-sided communication functions are an interface to MPI RMA
 - I think “one sided” is a confusing term because, as we will see, whilst the communication calls themselves are one sided often the synchronisation is issued on both sides
- Is a natural fit for some codes
- Can provide a performance/scalability increase for your codes
 - Programmability reasons
 - Hardware (interconnect) reasons
 - But is not a silver bullet!

Terminology

- Origin is the process initiating the request (performs the call)
 - Irrespective of whether data is being retrieved or written
- Target is the process whose memory is accessed
 - By the origin, either remotely reading or writing to this
- All remote access performed on windows of memory
- All access calls are non-blocking and issued inside an epoch
 - The epoch is what forces synchronisation of these calls

RMA program flow

- Collectively initialise a window
 - a) Start an RMA epoch (synchronisation)
 - b) Issue communication calls
 - c) Stop an RMA epoch (synchronisation)



Repeat as many times as you want

- Collectively free the window

Getting started with RMA

Window management, fences and data movement

Window creation

- A collective call, issued by all processes in the communicator

```
int MPI_Win_create(void *base, MPI_Aint size, int disp_unit,  
                  MPI_Info info, MPI_Comm comm, MPI_Win *win)
```

- Each process may specify completely different locations, sizes, displacement units and info arguments.
- You can specify no memory with a zero size and NULL base
- The same region of memory may appear in multiple windows that have been defined for a process. But concurrent communications to overlapping windows are disallowed.
- Performance may be improved by ensuring that the windows align with boundaries such as word or cache-line boundaries.

Other window management

- Retrieving window attributes

```
int MPI_Win_get_attr(MPI_Win win, int win_keyval,  
                    void *attribute_val, int *flag)
```

- `win_keyval` is one of `MPI_WIN_BASE`, `MPI_WIN_SIZE`, `MPI_WIN_DISP_UNIT`, `MPI_WIN_CREATE_FLAVOR`, `MPI_WIN_MODEL`
- `Attribute_val` if the attribute is available and in this case (`flag` is true), otherwise `flag` will be false

- Freeing a window

```
int MPI_Win_free(MPI_Win *win)
```

- All RMA calls must have been completed (i.e. the epoch stopped)

Fences

- Synchronisation calls are required to start and stop an epoch
 - Fences are the simplest way of doing this where global communication phases alternate with global communication
- Most closely follows a barrier synchronisation
 - A (collective) fence is called at the start and stop of an epoch

```
int MPI_Win_fence(int assert, MPI_Win win)
```

```
MPI_Win_fence(0, window);
```

Communication calls go here

```
MPI_win_fence(0, window);
```

*RMA can not be started until
this first fence*

*All issued communication
calls block here*

Fence attributes

- Attributes allow you to tell the MPI library more information for performance (but MPI implementations are allowed to ignore it!)
 - **MPI_MODE_NOSTORE** local window is not updated by local writes of any form since last synchronisation. *Can be different on processes*
 - **MPI_MODE_NOPUT** local window will not be updated by put/accumulate RMA operations until AFTER the next synchronisation call. *Can be different on processes*
 - **MPI_MODE_NOPRECEDE** fence does not complete any sequence of locally issues RMA calls. *Attribute must be given by all processes*
 - **MPI_MODE_NOSUCCEED** fence does not start any sequence of locally issued RMA calls. *Attribute must be given by all processes*
- Attributes can be or'd together, i.e.
 - `MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), window)` or `ior(MPI_MODE_NOPUT, MPI_MODE_NOPRECEDE)`

RMA Communication calls

- Three general calls, all non-blocking:

- Get data from target's memory

```
int MPI_Get(void *origin_addr, int origin_count,  
            MPI_Datatype origin_datatype, int target_rank,  
            MPI_Aint target_disp, int target_count,  
            MPI_Datatype target_datatype, MPI_Win win)
```

- Put data into target's memory

```
int MPI_Put(const void *origin_addr, int origin_count,  
            MPI_Datatype origin_datatype, int target_rank,  
            MPI_Aint target_disp, int target_count,  
            MPI_Datatype target_datatype, MPI_Win win)
```

- Accumulate data in target's memory with some other data

```
int MPI_Accumulate(void *origin_addr, int origin_count,  
                  MPI_Datatype origin_datatype, int target_rank,  
                  MPI_Aint target_disp, int target_count,  
                  MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
```

RMA communication comments

- Similarly to non-blocking P2P one must wait for synchronisation (i.e. end of the epoch) until accessing retrieved data (*get*) or overwriting written data (*put/accumulate*)
- `target_disp` is multiplied by window displacement unit, `origin_count` and `target_count` are in units of data type
- Undefined operations:
 - Local stores/reads with a remote PUT in an epoch
 - Several origin processes performing concurrent PUT to the same target location
 - Single origin process performing multiple PUTs to the same target location in a single epoch
- Accumulate supports the `MPI_Reduce` operations, but NOT user defined operations. Also supports `MPI_REPLACE` which is effectively the same as a `put`.

Example

Based on an example at
cvw.cac.cornell.edu/MPIoneSided/fence

```
MPI_Win win;
```

```
if (rank == 0) {  
    MPI_Win_create(buf, sizeof(int)*20, 1, MPI_INFO_NULL, comm, &win);  
} else {  
    MPI_Win_create(NULL, 0, 1, MPI_INFO_NULL, comm, &win);  
}
```

Rank 0 creates a window of 20 integers, displacement unit = 1

Other ranks create a window but attach no local memory

```
MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
```

```
if (rank != 0) {  
    MPI_Get(mybuf, 20, MPI_INT, 0, 0, 20, MPI_INT, win);  
}
```

```
MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
```

```
MPI_Win_free(&win)
```

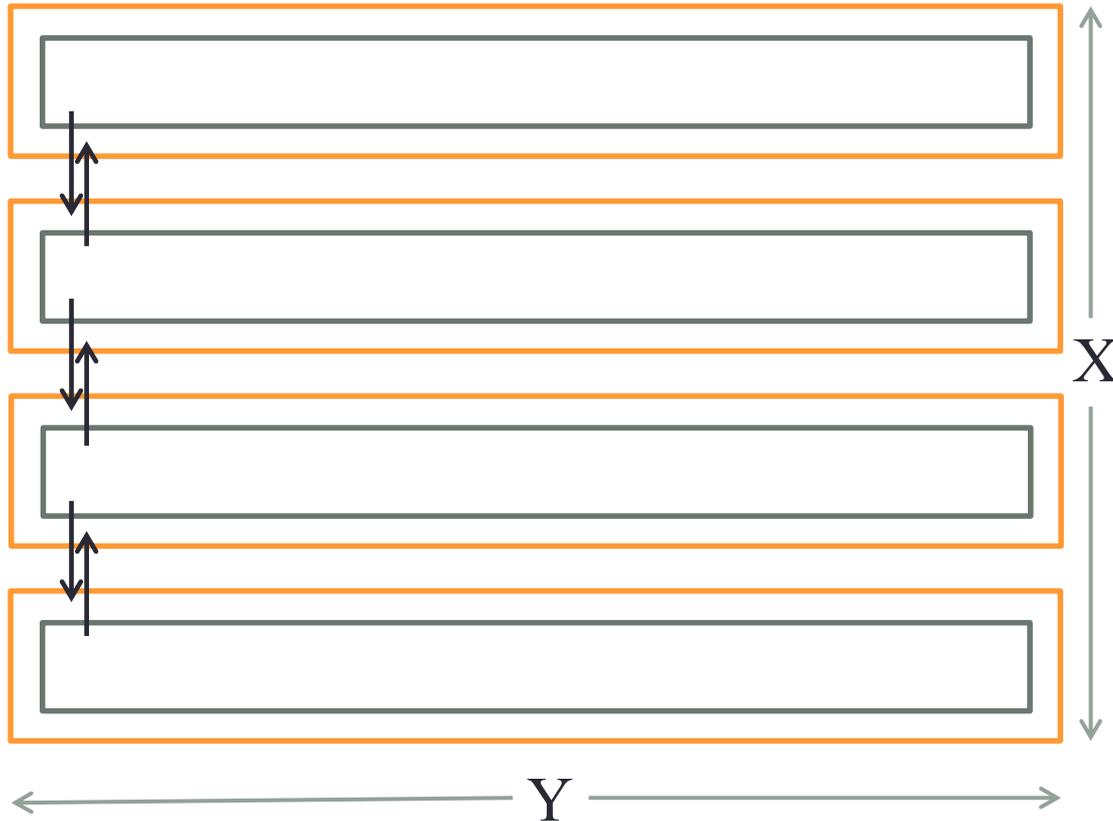
Fence, no preceding RMA calls

Non-zero ranks get the 20 integers from rank 0

Fence, complete all communications and no RMA calls in next epoch

Practical

2D Jacobi solving Laplace's equation



- Decomposed in X dimension only.
- All halo swapping communications are currently non-blocking P2P
- Replace these with RMA
- C and Fortran versions provided

Practical

- MPI API online reference:
 - <http://www.mpich.org/static/docs/v3.2/www3/>
- Instructions at
 - http://www.archer.ac.uk/training/course-material/2016/09/160929_AdvMPI_EPCC/mpi_rma.pdf
- Zip file at
 - http://www.archer.ac.uk/training/course-material/2016/09/160929_AdvMPI_EPCC/jacobi.zip
 - Makefile and submission script included using qsub on ARCHER

```
qsub -q course1 subjacobi.pbs
```