Single Sided MPI

Reusing this material

archer

epcc | THE UNIVERSITY OF EDINBURGH

Why RMA

- One-sided communication functions are an interface to MPI RMA

- 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 a RMA epoch (synchronisation)
  b) Issue communication calls
  c) Stop a RMA epoch (synchronisation)

- Collectively free the window

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.

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_NOSTORE | MPI_MODE_NO_SUCCEED), window)
```

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 communications

- Similarly to non-blocking P2P one must wait for synchronisation (i.e. end of epoch) until accessing retrieved data (*get*) or overwriting written data (*put/accumulate*)
- `target_disp` is in bytes (multipled by window displacement unit), `origin_count` and `target_count` are in elements 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);
}
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)
```

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

Other ranks create a window but attach no local memory

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

Synchronisation modes

- Active target
  - Both processes are explicitly involved in the data movement. Only one process issues the data transfer call but all processes issue the synchronisation.
- Passive target
  - Only the origin process is involved in the data movement, there are no calls made on the target process. For instance two origin processes might communicate by accessing the same location in a target window, and the target process (which does not participate) might be distinct from the origin processes.
- *Fence is an example of active target as each process issues the fence calls*

Epoch types

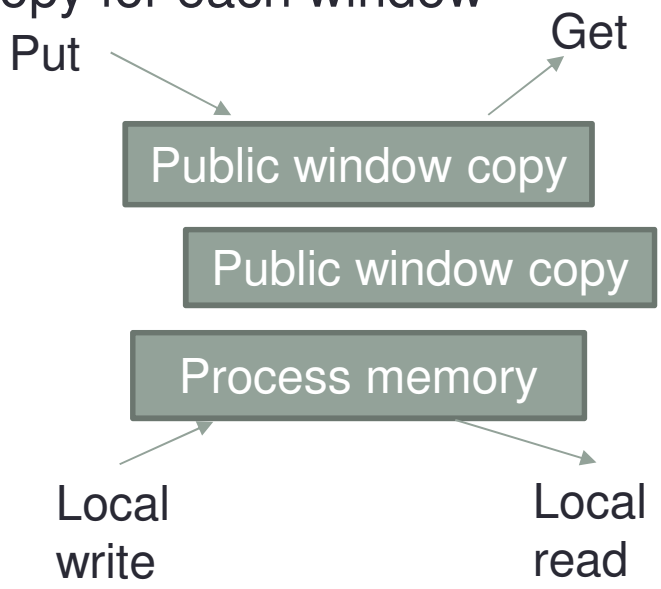
- Access epoch
  - RMA communication calls (get, put etc) can only be issued inside an access epoch. This is started with an RMA synchronisation call on the origin and completes with the next synchronisation call.
  - i.e. it is used to access the remote memory of another process.
- Exposure epoch
  - Used in active target communication, this is required to expose memory on the target so it can be accessed by other processes' RMA operations.
- *Fences abstract the programmer from this as they will complete/start both access and exposure epochs automatically as required*

RMA Memory model

- Public and private window copies
  - Public memory region is addressable by other processes (i.e. exposed main memory)
  - Private memory (i.e. transparent caches or communication buffers) which is only locally visible but elements from public memory might be stored.
- Coherent if updates to main memory are automatically reflected in private copy consistently
- Non-coherent if updates need to be explicitly synchronised

RMA Memory model

- MPI therefore has two models
  - Unified if public and private copies are identical – used if possible, realistic on cache coherent machines. *(This was added in MPI v3)*
  - Separate if they are not, here there is only one copy of a variable in process memory but also a distinct public copy for each window that contains it. The old model
- In the separate model a suitable synchronisation call (i.e. end of an epoch) must be issued to make these consistent. In the unified model some synchronisation calls might be omitted for performance reasons
- The window attribute tells you which model it follows



Request handles

- There is also an `R` variant of all communication calls which associate a request handle with the operation
  - i.e. `Rget, Rput` and `Raccumulate`
  - Only valid in passive target synchronisation
- This request handle is then used as you would any other request handle (you can call `MPI_test, MPI_wait` etc on it.)
- For a `put/accumulate` guarantees that the origin's buffer can be overwritten (but not that data has arrives)
- For a `get` guarantees that data is available in the origin's buffer

Dynamic windows

- Traditional windows allocation required that memory is attached at window creation
  - But what if we don't know the amount of memory needed at that point?
- Dynamic windows allow memory can be exposed without additional synchronisation

```
int MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm,
MPI_Win *win)
```

- Memory is then attached with

```
int MPI_Win_attach(MPI_Win win, void *base, MPI_Aint size)
```

- Detached with

```
int MPI_Win_detach(MPI_Win win, const void *base)
```

Dynamic windows

- Memory being attached can not overlap with any other memory that is already attached
- The target displacement argument of the origin's communication call is the address at the target
- You can use `MPI_Get_Address` on the target to retrieve this
- Must ensure that the memory has been attached on the target process before the origin issues any RMA calls referencing it

Practical

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

MPI API online reference http://www.mpich.org/static/docs/v3.2/www3/