High Performance Computing: Tools and Applications

Edmond Chow
School of Computational Science and Engineering
Georgia Institute of Technology

Lecture 19

MPI remote memory access (RMA)

- *Put*/*Get* directly from/to the memory of another process
- In contrast to “message passing,” processes do not need to coordinate the communication. This is most valuable when processes don’t know what data is needed on another process, or what data will be sent from another process.
- Similarities and contrasts with shared memory programming

Abstract view - addresses on the remote process

- To *put* or *get* data on a remote process, we need to know about addresses on the remote process.
- Instead of having all memory accessible to RMA operations, each process defines a *window* of its memory that can be accessed.
  - This is much cheaper than making all memory accessible to RMA
  - This helps solve the problem of remote addresses
- Addresses of memory on remote processes begin at 0, corresponding to the beginning of the window on the remote process
- Above, each remote location is a process and its address. There is different functionality in MPI called *shared memory MPI* that defines a *global* shared address space used by all the processes (rather than have to specify which process and its address, an address is enough).

Abstract view - RMA operations

- Put (write)
- Get (read) - often more expensive than put
- Accumulate
- All operations are *nonblocking*

Abstract view - completion

- How can a process know that a put onto its process has been completed (ready for reading)?
- Needing this is not intuitive. There is no equivalent in shared memory.
- A “barrier” can be used after all puts.

Abstract view - completion

- How can a process know that a get has been completed?
  - blocking get?
  - nonblocking get with wait?
- “get start” and “get end” functions could be used to signal that the gets are completed

Three steps

1. Define windows
2. Move the data
3. Find out when the data is available

Comparison to shared memory access

- don’t need to define windows
- read/write; writes to same location in a window leads to undefined behavior
- don’t need completion functions

Memory windows

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

- Called collectively by all processes in `comm`
- `base` is the local address of beginning of window
- `size` is size of window in bytes
- `disp_unit` is the unit size for displacements (in bytes)
- `info` used to provide performance tuning options; `MPI_INFO_NULL` can be used
- free the window: `MPI_Win_free(MPI_Win *win);`

Some terminology

- origin = the process performing the get or put (not the origin of the data)
- target = the other process

Moving data

```
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);

int MPI_Get(void *origin_addr, ...others same...);
```

- address on target is `target_disp` units relative to beginning of target's window
- for put, source data (origin) can be anywhere in process memory
- for get, destination data (origin) can be anywhere in process memory
- What happens if different processes put to the same location in the window of a process? Result is undefined.
- Note: result of `get` may be stored in the origin's window, so gets cannot be overlapped or else result is undefined

Accumulate

- Accumulate a value onto a target, e.g., `a[1] = a[1] + b` on a target

```
int MPI_Accumulate(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_Op op, MPI_Win win);
```

- related to *put*; signature same as `MPI_Put` except for `MPI_Op`
- targets from different origins can overlap (unlike puts), but order of accumulates is not defined
- cannot overlap puts and accumulates on a target

Completing RMA data transfers

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

- Called collectively by all processes that earlier collectively created `win`
- Blocks until all RMA operations complete (that were initiated since the last fence)
- Example: Jacobi-Schwarz. Interleaved stages of computation and communication.
- What happens if a process writes to its local window? Result is undefined. (Use fence to separate puts/accumulates with local writes.)
- Can a write to a local window be visible to another process immediately via get?
- `assert` can be used for performance tuning; can use value 0

Passive target synchronization

- Completion of `MPI_Get` on origin should not involve the target
- Abstract example: code on the origin:

```
MPI_Win_lock(target)
MPI_Get(target)
MPI_Get(target)
MPI_Win_unlock(target)
// results of gets are now available
```

- Note: `lock` and `unlock` are *not* called collectively like in `fence`
- Example with puts:

```
MPI_Win_lock(target)
MPI_Put(target)
MPI_Put(target)
MPI_Win_unlock(target)
// put is done; could overwrite local buffer
```

- How efficient is this compared to Irecv/Send?

Passive target synchronization

Instead of “lock” and unlock“, think:”begin access" and “end access”

```
int MPI_Win_lock(int lock_type, int rank,
  int assert, MPI_Win win);

int MPI_Win_unlock(int rank, MPI_Win win);
```

- `unlock` will block until RMA operation completes
- `lock_type`
  - `MPI_LOCK_SHARED`
    - many processes can access the target window
    - not a lock; just a way to indicate completion
    - like before, access cannot overlap (except for `accumulate`)
  - `MPI_LOCK_EXCLUSIVE`
    - exclusive or “atomic” access to target window
- There are also `MPI_Win_flush` and `MPI_Win_flush_local` to flush RMA operations if you don’t want to unlock the window yet (useful when you immediately need the result of an RMA op)

Non-collective active target synchronization

- Also known as “scalable synchronization”
- Like `MPI_Win_fence`, but only between processes that communicate
- Better than `fence` for Jacobi-Schwarz, because don’t need a barrier between all processes
- Exposure epoch: period of time that local window can be a target of RMA ops
- Access epoch: period of time that process can access remote windows
- Abstract example: code on one process:

```
MPI_Win_post(neighbor_group, win);  // begin exposure epoch
MPI_Win_start(neighbor_group, win); // begin access epoch
...
MPI_Win_complete(win);              // end access epoch
MPI_Win_wait(win);                  // end exposure epoch
```

Summary of completion techniques

- collective fence (active target synchronization)
- passive target synchronization (true one-sided operation)
- scalable synchronization (non-collective active target synchronization)