Lecture 35: More on One Sided Communication

William Gropp

www.cs.illinois.edu/~wgropp
Synchronization in MPI RMA

• Active target requires cooperation by all processes in the group of the window object
  ♦ MPI_Win_fence, MPI_Win_{post/start/complete/wait/test}
  ♦ Good for many but not all RMA applications

• What if each process may need to independently access data?
  ♦ Use Passive target synchronization
Lock/Unlock: Passive Target Synchronization

- Passive mode: One-sided, *asynchronous* communication
  - Target does **not** participate in communication operation
- Shared memory-like model
Passive Target Synchronization

- **Lock/Unlock**: Begin/end passive mode epoch
  - Target process does not make a corresponding MPI call
  - Can initiate multiple passive target epochs to different processes
  - Concurrent epochs to same process not allowed (affects threads)

- **Lock type**
  - **SHARED**: Other processes using shared can access concurrently
  - **EXCLUSIVE**: No other processes can access concurrently

- **Flush**: Remotely complete RMA operations to the target process
  - After completion, data can be read by target process or a different process

- **Flush_local**: Locally complete RMA operations to the target process
Lock is not Lock

- The name “Lock” is unfortunate
  - Lock is really “begin epoch”
  - Unlock is really “end epoch”
- An MPI “Lock” does not establish a critical section or mutual exclusion
  - With “MPI_LOCK_EXCLUSIVE” the RMA operations have exclusive access to the data they access/update during the time that they access the remote window
  - This is very different than a “lock” in the sense of a thread lock
Understanding the MPI RMA Completion Model

• Very relaxed
  ♦ To give the implementer the greatest flexibility
  ♦ Describing this relaxed model precisely is difficult
    • Only Implementer needs to obey the rules
  ♦ But it doesn’t matter; simple rules work for most programmers

• When does the data actually move?
## Data Moves Early

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Win_lock</td>
<td>(lock granted)</td>
</tr>
<tr>
<td>(win_lock returns)</td>
<td></td>
</tr>
<tr>
<td>MPI_Put</td>
<td>(window updated)</td>
</tr>
<tr>
<td>MPI_Put</td>
<td>(window updated)</td>
</tr>
<tr>
<td>MPI_Get</td>
<td>(window accessed)</td>
</tr>
<tr>
<td>MPI_Win_unlock</td>
<td>(lock released)</td>
</tr>
<tr>
<td>(unlock returns)</td>
<td></td>
</tr>
</tbody>
</table>
Data Moves Late

Process 0

MPI_Win_lock (save information)

MPI_Put (save information)

MPI_Put (save information)

MPI_Get (save information)

MPI_Win_unlock (acquire lock, process requests, release lock)

(unlock returns)

Process 1
Understanding Why Late May Be Good

• Use a simple performance model:
  ♦ Assume data size is small
  ♦ Each communication on network takes time \( L \)

• Early approach:
  ♦ 8 separate messages, so 8L

• Late approach:
  ♦ 2 messages (including data), so 2L

• Late approach is 4 x faster than the early approach for small amounts of data
Understanding Why Early May Be Good

- Use a simple performance model:
  - Assume data size is large
  - Each data communication on network takes time $L + \text{rn}$, each control message takes time $L$
  - Assume communication can be overlapped with computation or other communication, but that latency ($L$) cannot be overlapped
Understanding Why Early May Be Good

- Early approach:
  - $5L + 3(L+rn)$; all but the $8L$ can be overlapped with computation

- Late approach:
  - $2$ messages, so $2L + 3rn$. Nothing may be overlapped

- Assuming full overlap, Early is $8L$ and Late is $2L+3rn$, so Late can be arbitrarily slower than Early; equal when $n = 2L/r$
Advanced Passive Target Synchronization

- **Lock_all**: Shared lock, passive target epoch to all other processes
  - Expected usage is long-lived: lock_all, put/get, flush, ..., unlock_all
- **Flush_all** – remotely complete RMA operations to all processes
- **Flush_local_all** – locally complete RMA operations to all processes

```
MPI_Win_lock_all(int assert, MPI_Win win)
MPI_Win_unlock_all(MPI_Win win)
MPI_Win_flush_all/flush_local_all(MPI_Win win)
```
Implementing GA-like Computation by RMA Lock/Unlock

local buffer on P0

local buffer on P1
Code Example

- ga_mpi_ddt_rma.c
- Only synchronization from origin processes, no synchronization from target processes
- Code thanks to Xin Zhao, posted on Moodle
Which synchronization mode should I use, when?

- RMA communication has low overheads versus send/recv
  - Two-sided: Matching, queuing, buffering, unexpected receives, etc...
  - One-sided: No matching, no buffering, always ready to receive
  - Utilize RDMA provided by high-speed interconnects (e.g. InfiniBand)

- Active mode: bulk synchronization
  - E.g. ghost cell exchange

- Passive mode: asynchronous data movement
  - Useful when dataset is large, requiring memory of multiple nodes
  - Also, when data access and synchronization pattern is dynamic
  - Common use case: distributed, shared arrays

- Passive target locking mode
  - Lock/unlock – Useful when exclusive epochs are needed
  - Lock_all/unlock_all – Useful when only shared epochs are needed
MPI RMA Memory Model

- MPI-3 provides two memory models: separate and unified
- MPI-2: Separate Model
  - Logical public and private copies
  - MPI provides software coherence between window copies
  - Extremely portable, to systems that don’t provide hardware coherence
- MPI-3: New Unified Model
  - Single copy of the window
  - System must provide coherence
  - Superset of separate semantics
    - E.g. allows concurrent local/remote access
  - Provides access to full performance potential of hardware
MPI RMA Memory Model (separate windows)

- Very portable, compatible with non-coherent memory systems
- Limits concurrent accesses to enable software coherence
MPI RMA Memory Model (unified windows)

- Allows concurrent local/remote accesses
- Concurrent, conflicting operations are allowed (not invalid)
  - Outcome is not defined by MPI (defined by the hardware)
- Can enable better performance by reducing synchronization
## MPI RMA Operation Compatibility (Separate)

<table>
<thead>
<tr>
<th></th>
<th>Load</th>
<th>Store</th>
<th>Get</th>
<th>Put</th>
<th>Acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>OVL+NOVL</td>
<td>OVL+NOVL</td>
<td>OVL+NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
</tr>
<tr>
<td>Store</td>
<td>OVL+NOVL</td>
<td>OVL+NOVL</td>
<td>NOVL</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Get</td>
<td>OVL+NOVL</td>
<td>NOVL</td>
<td>OVL+NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
</tr>
<tr>
<td>Put</td>
<td>NOVL</td>
<td>X</td>
<td>NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
</tr>
<tr>
<td>Acc</td>
<td>NOVL</td>
<td>X</td>
<td>NOVL</td>
<td>NOVL</td>
<td>OVL+NOVL</td>
</tr>
</tbody>
</table>

This matrix shows the compatibility of MPI-RMA operations when two or more processes access a window at the same target concurrently.

**OVL** – Overlapping operations permitted

**NOVL** – Nonoverlapping operations permitted

**X** – Combining these operations is OK, but data might be garbage
### MPI RMA Operation Compatibility (Unified)

<table>
<thead>
<tr>
<th></th>
<th>Load</th>
<th>Store</th>
<th>Get</th>
<th>Put</th>
<th>Acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>OVL+NOVL</td>
<td>OVL+NOVL</td>
<td>OVL+NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
</tr>
<tr>
<td>Store</td>
<td>OVL+NOVL</td>
<td>OVL+NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
</tr>
<tr>
<td>Get</td>
<td>OVL+NOVL</td>
<td>NOVL</td>
<td>OVL+NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
</tr>
<tr>
<td>Put</td>
<td>NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
</tr>
<tr>
<td>Acc</td>
<td>NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
<td>NOVL</td>
<td>OVL+NOVL</td>
</tr>
</tbody>
</table>

This matrix shows the compatibility of MPI-RMA operations when two or more processes access a window at the same target concurrently.

**OVL** – Overlapping operations permitted  
**NOVL** – Nonoverlapping operations permitted
Summary of MPI RMA

• MPI provides a powerful one-sided communication model

• General and precisely specified model
  ♦ Complexity of the precision is sometimes confused with complexity for the user
    • There are simple models for the user that address most common use cases

• Implementations improving but many still poor, so test performance before using

• One more feature – MPI and shared memory – in the next lecture