MPI for Scalable Computing (continued from yesterday)

Bill Gropp, University of Illinois at Urbana-Champaign

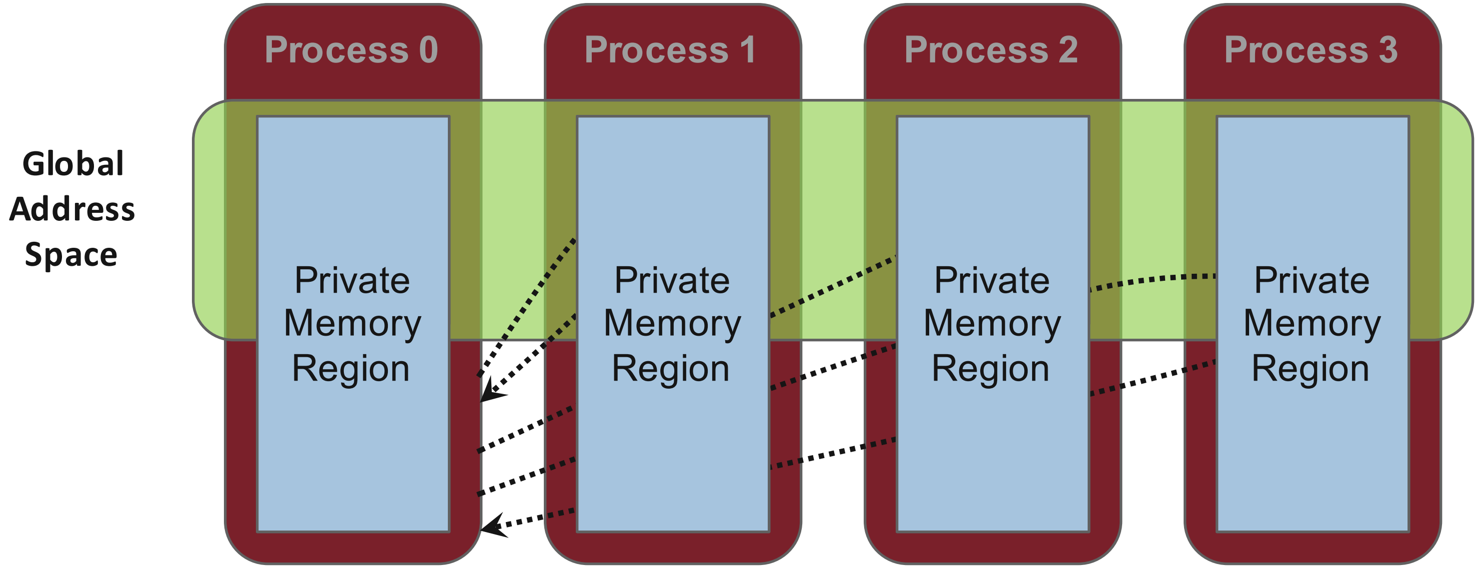
Rusty Lusk, Argonne National Laboratory

Rajeev Thakur, Argonne National Laboratory

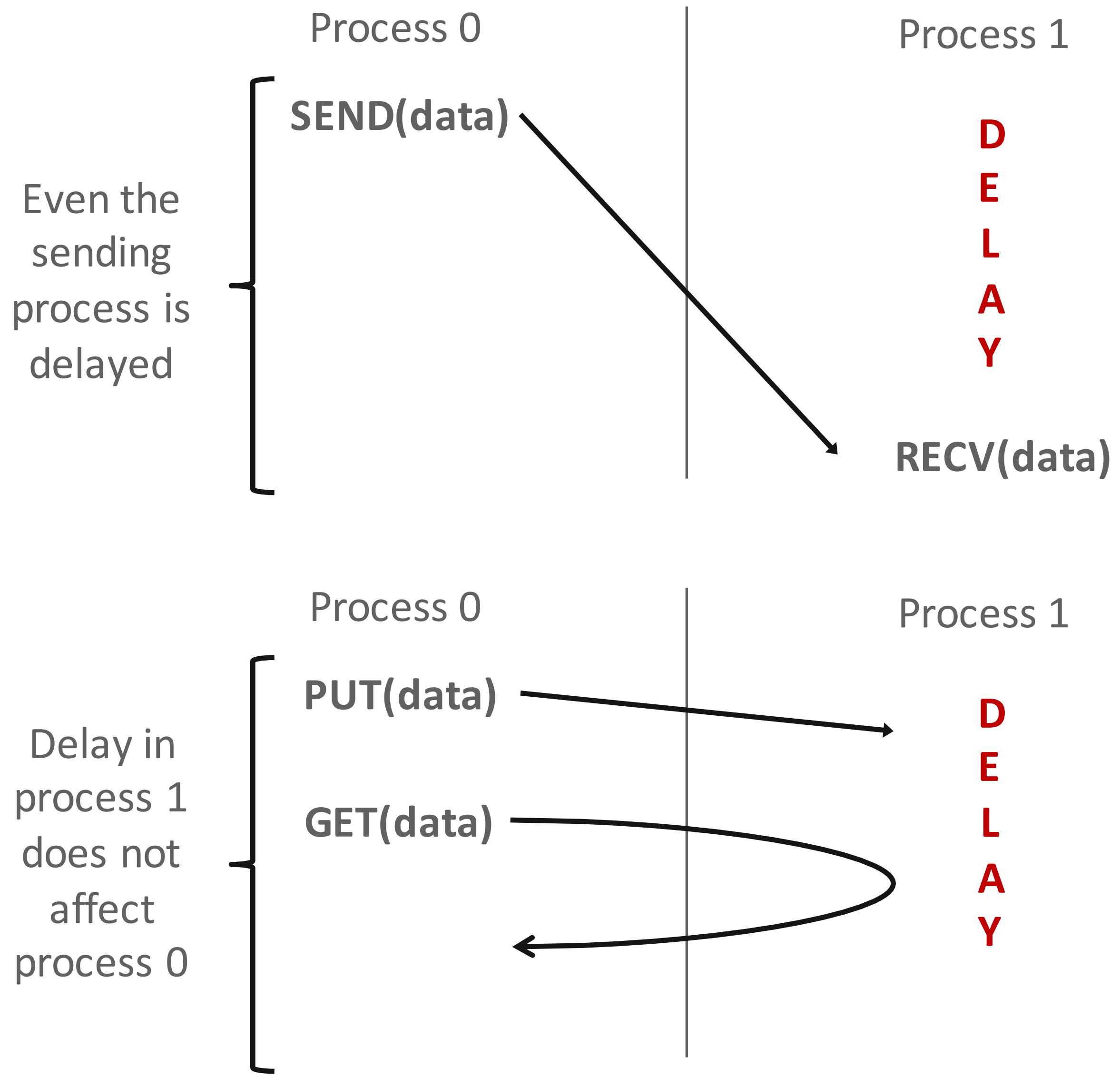
One-Sided Communication

One-Sided Communication

- The basic idea of one-sided communication models is to decouple data movement with process synchronization
  - Should be able to move data without requiring that the remote process synchronize
  - Each process exposes a part of its memory to other processes
  - Other processes can directly read from or write to this memory



Comparing One-sided and Two-sided Programming



Advantages of RMA Operations

- Can do multiple data transfers with a single synchronization operation
  - like BSP model
- Bypass tag matching
  - effectively precomputed as part of remote offset
- Some irregular communication patterns can be more economically expressed
- Can be significantly faster than send/receive on systems with hardware support for remote memory access, such as shared memory systems

Irregular Communication Patterns with RMA

- If communication pattern is not known *a priori*, the send-recv model requires an extra step to determine how many sends-recvs to issue
- RMA, however, can handle it easily because only the origin or target process needs to issue the put or get call
- This makes dynamic communication easier to code in RMA

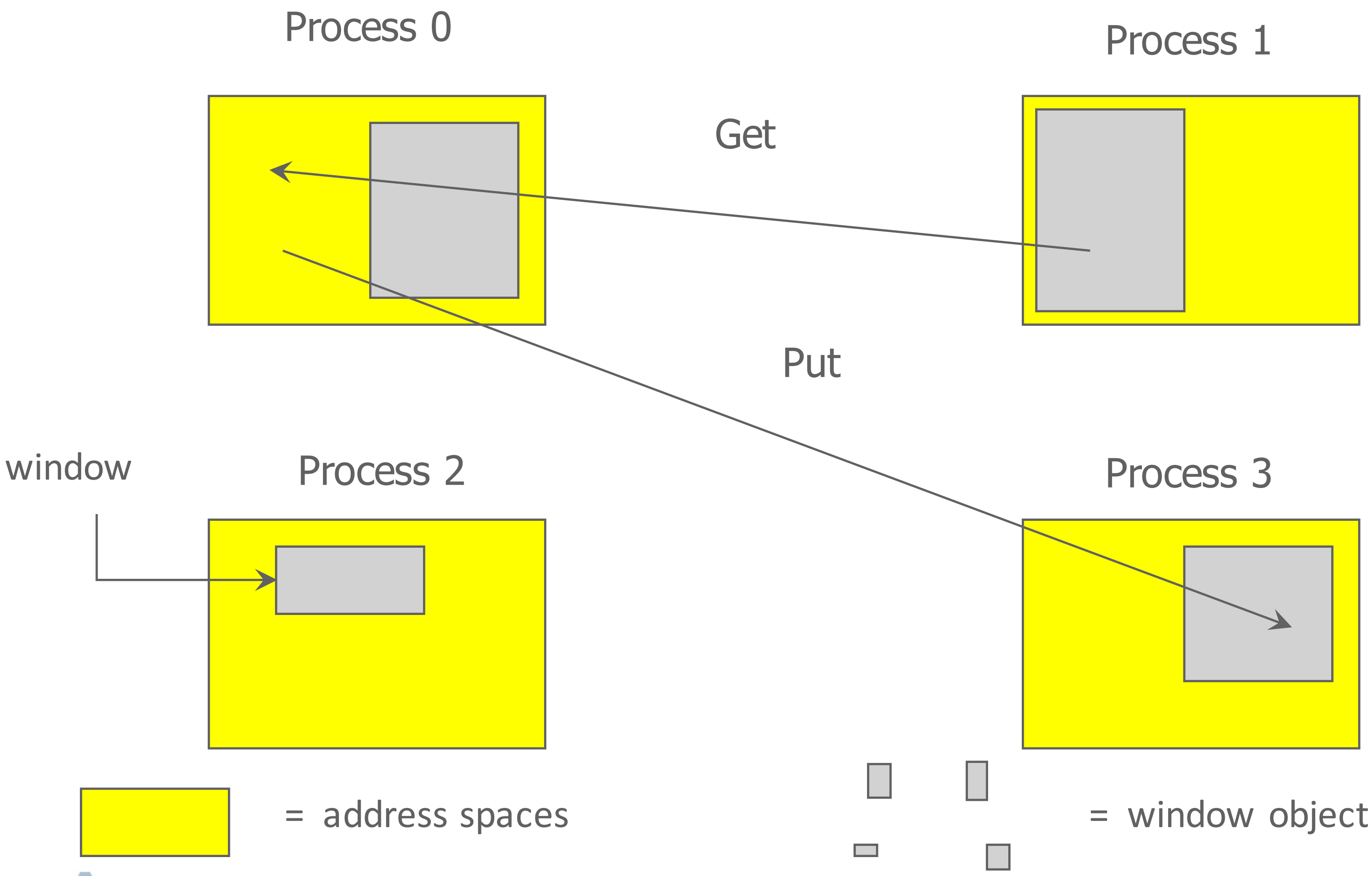
What we need to know in MPI RMA

- How to create remote accessible memory?
- Reading, Writing and Updating remote memory
- Data Synchronization
- Memory Model

Creating Public Memory

- Any memory created by a process is, by default, only locally accessible
  - X = malloc(100);
- Once the memory is created, the user has to make an explicit MPI call to declare a memory region as remotely accessible
  - MPI terminology for remotely accessible memory is a "window"
  - A group of processes collectively create a "window"
- Once a memory region is declared as remotely accessible, all processes in the window can read/write data to this memory without explicitly synchronizing with the target process

Remote Memory Access Windows and Window Objects



Basic RMA Functions for Communication

- **`MPI_Win_create`** exposes local memory to RMA operation by other processes in a communicator
  - Collective operation
  - Creates window object
- **`MPI_Win_free`** deallocates window object

- **`MPI_Put`** moves data from local memory to remote memory
- **`MPI_Get`** retrieves data from remote memory into local memory
- **`MPI_Accumulate`** updates remote memory using local values
- Data movement operations are non-blocking
- **Subsequent synchronization on window object needed to ensure operation is complete**

Window creation models

- Four models exist
  - MPI_WIN_CREATE
    - You already have an allocated buffer that you would like to make remotely accessible
  - MPI_WIN_ALLOCATE
    - You want to create a buffer and directly make it remotely accessible
  - MPI_WIN_CREATE_DYNAMIC
    - You don’t have a buffer yet, but will have one in the future
  - MPI_WIN_ALLOCATE_SHARED
    - You want multiple processes on the same node share a buffer
    - We will not cover this model today

MPI_WIN_CREATE

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

- Expose a region of memory in an RMA window
  - Only data exposed in a window can be accessed with RMA ops.
- Arguments:
  - base - pointer to local data to expose
  - size - size of local data in bytes (nonnegative integer)
  - disp_unit - local unit size for displacements, in bytes (positive integer)
  - info - info argument (handle)
  - comm - communicator (handle)

Example with MPI_WIN_CREATE

```
int main(int argc, char ** argv)
{
    int *a;    MPI_Win win;

    MPI_Init(&argc, &argv);

    /* create private memory */
    a = (void *) malloc(1000 * sizeof(int));
    /* use private memory like you normally would */
    a[0] = 1;  a[1] = 2;

    /* collectively declare memory as remotely accessible */
    MPI_Win_create(a, 1000*sizeof(int), sizeof(int), MPI_INFO_NULL,
                      MPI_COMM_WORLD, &win);

    /* Array 'a' is now accessibly by all processes in
     * MPI_COMM_WORLD */

    MPI_Win_free(&win);

    MPI_Finalize(); return 0;
}
```

MPI_WIN_ALLOCATE

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

- Create a remotely accessible memory region in an RMA window
  - Only data exposed in a window can be accessed with RMA ops.
- Arguments:
  - size - size of local data in bytes (nonnegative integer)
  - disp_unit - local unit size for displacements, in bytes (positive integer)
  - info - info argument (handle)
  - comm - communicator (handle)
  - baseptr - pointer to exposed local data

Example with MPI_WIN_ALLOCATE

```
int main(int argc, char ** argv)
{
    int *a;    MPI_Win win;

    MPI_Init(&argc, &argv);

    /* collectively create remotely accessible memory in the
   window */
    MPI_Win_allocate(1000*sizeof(int), sizeof(int),
   MPI_INFO_NULL,
                         MPI_COMM_WORLD, &a, &win);

    /* Array 'a' is now accessibly by all processes in
     * MPI_COMM_WORLD */

    MPI_Win_free(&win);

   MPI_Finalize(); return 0;
}
```

MPI_WIN_CREATE_DYNAMIC

```
int MPI_Win_create_dynamic(…, MPI_Comm comm, MPI_Win *win)
```

- Create an RMA window, to which data can later be attached
  - Only data exposed in a window can be accessed with RMA ops
- Application can dynamically attach memory to this window
- Application can access data on this window only after a memory region has been attached

Example with MPI_WIN_CREATE_DYNAMIC

```
int main(int argc, char ** argv)
{
    int *a;    MPI_Win win;

    MPI_Init(&argc, &argv);
    MPI_Win_create_dynamic(MPI_INFO_NULL, MPI_COMM_WORLD, &win);

    /* create private memory */
    a = (void *) malloc(1000 * sizeof(int));
    /* use private memory like you normally would */
    a[0] = 1;  a[1] = 2;

    /* locally declare memory as remotely accessible */
    MPI_Win_attach(win, a, 1000*sizeof(int));

    /*Array 'a' is now accessibly by all processes in MPI_COMM_WORLD*/

    /* undeclare public memory */
    MPI_Win_detach(win, a);
    MPI_Win_free(&win);

   MPI_Finalize(); return 0;
}
```

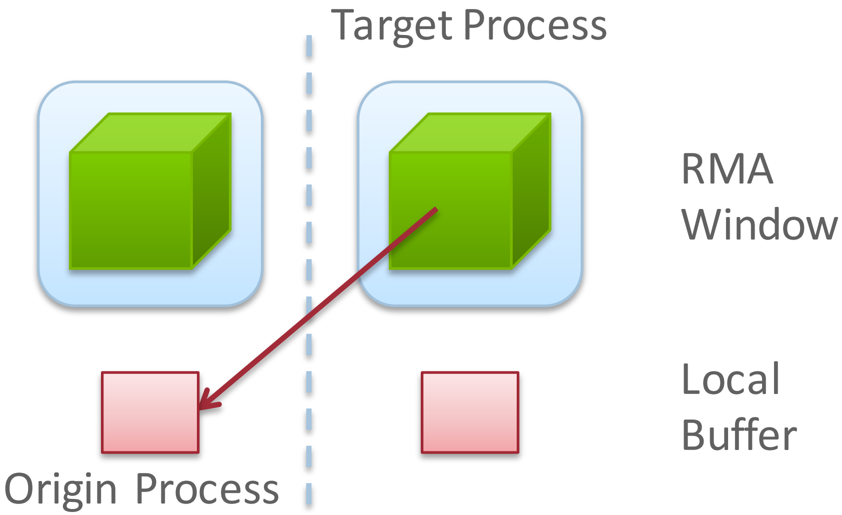
Data movement

- MPI provides ability to read, write and atomically modify data in remotely accessible memory regions
  - MPI_GET
  - MPI_PUT
  - MPI_ACCUMULATE
  - MPI_GET_ACCUMULATE
  - MPI_COMPARE_AND_SWAP
  - MPI_FETCH_AND_OP

Data movement: *Get*

```
MPI_Get(origin_addr, origin_count, origin_datatype,
        target_rank, target_disp, target_count,
target_datatype,
        win)
```

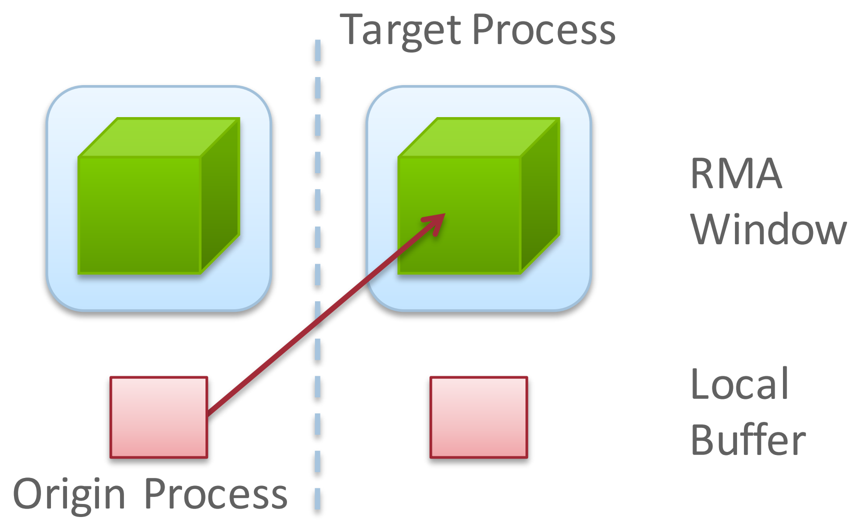
- Move data to origin, from target
- Separate data description triples for origin and target



Data movement: *Put*

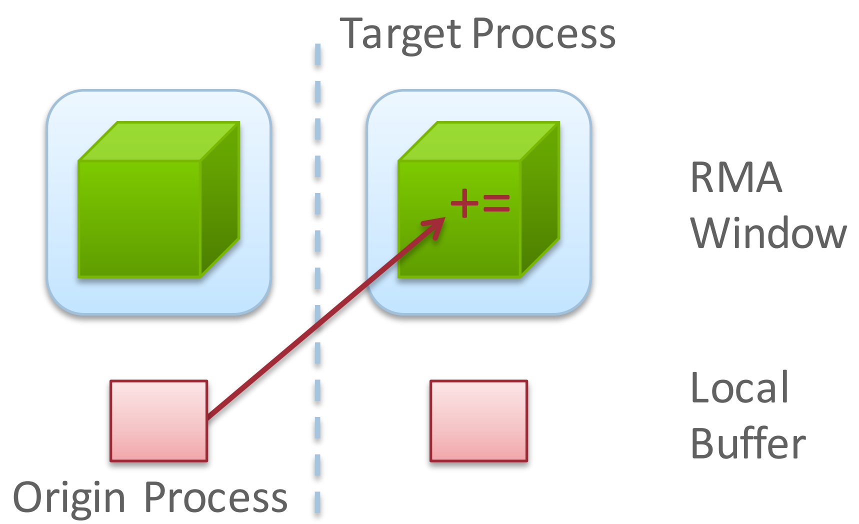
```
MPI_Put(origin_addr, origin_count, origin_datatype,
        target_rank, target_disp, target_count,
target_datatype,
        win)
```

- Move data <u>from</u> origin, <u>to</u> target
- Same arguments as MPI_Get



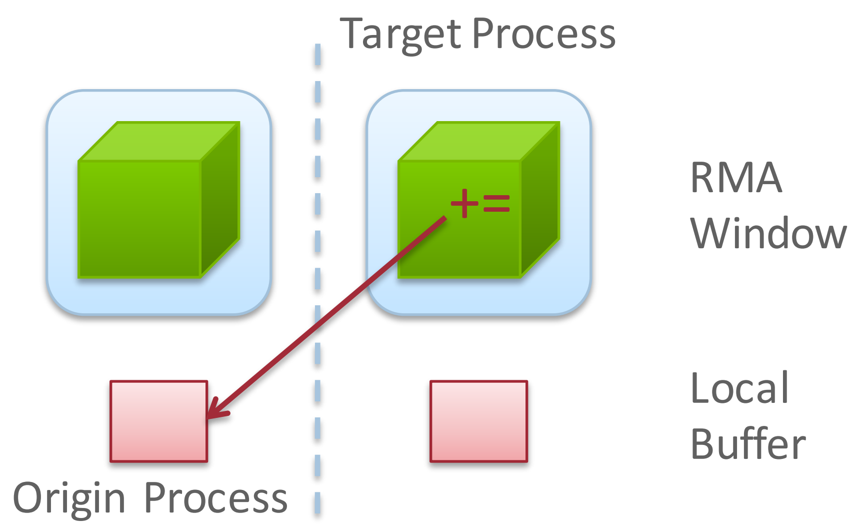
Data aggregation: *Accumulate*

- Like MPI_Put, but applies an MPI_Op instead
  - Predefined ops only, no user-defined!
- Result ends up at target buffer
- Different data layouts between target/origin OK, basic type elements must match
- Put-like behavior with MPI_REPLACE (implements *f(a,b)=b*)
  - Per element atomic PUT



Data aggregation: *Get Accumulate*

- Like MPI_Get, but applies an MPI_Op instead
  - Predefined ops only, no user-defined!
- Result at target buffer; original data comes to the source
- Different data layouts between target/origin OK, basic type elements must match
- Get-like behavior with MPI_NO_OP
  - Per element atomic GET



Ordering of Operations in MPI RMA

- For Put/Get operations, ordering does not matter
  - If you do two concurrent PUTs to the same location, the result can be garbage
- Two accumulate operations to the same location are valid
  - If you want "atomic PUTs", you can do accumulates with MPI_REPLACE
- All accumulate operations are ordered by default
  - User can tell the MPI implementation that (s)he does not require ordering as optimization hints
  - You can ask for "read-after-write" ordering, "write-after-write" ordering, or "read-after-read" ordering

Additional Atomic Operations

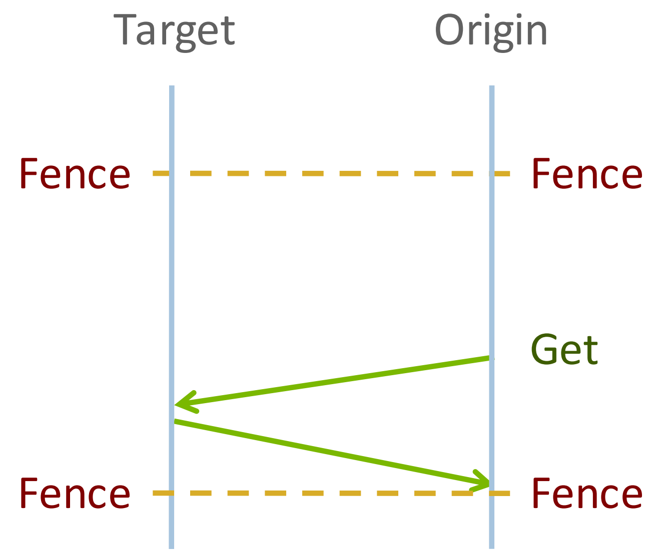
- Compare-and-swap
  - Compare the target value with an input value; if they are the same, replace the target with some other value
  - Useful for linked list creations – if next pointer is NULL, do something
- Fetch-and-Op
  - Special case of Get accumulate for predefined datatypes – (probably) faster for the hardware to implement

RMA Synchronization Models

- RMA data visibility
  - When is a process allowed to read/write from remotely accessible memory?
  - How do I know when data written by process X is available for process Y to read?
  - RMA synchronization models provide these capabilities
- MPI RMA model allows data to be accessed only within an "epoch"
  - Three types of epochs possible:
    - Fence (active target)
    - Post-start-complete-wait (active target)
    - Lock/Unlock (passive target)
- Data visibility is managed using RMA synchronization primitives
  - MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL
  - Epochs also perform synchronization

Fence Synchronization

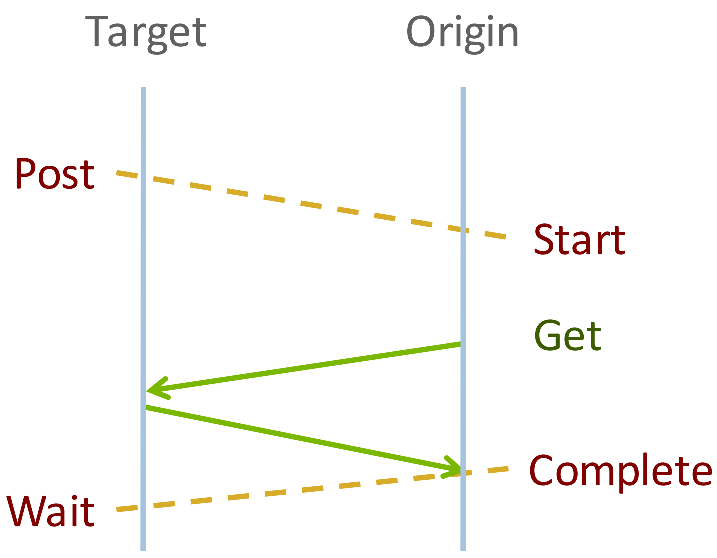
- `MPI_Win_fence(assert, win)`
- Collective synchronization model -- assume it synchronizes like a barrier
- Starts *and* ends access & exposure epochs (usually)



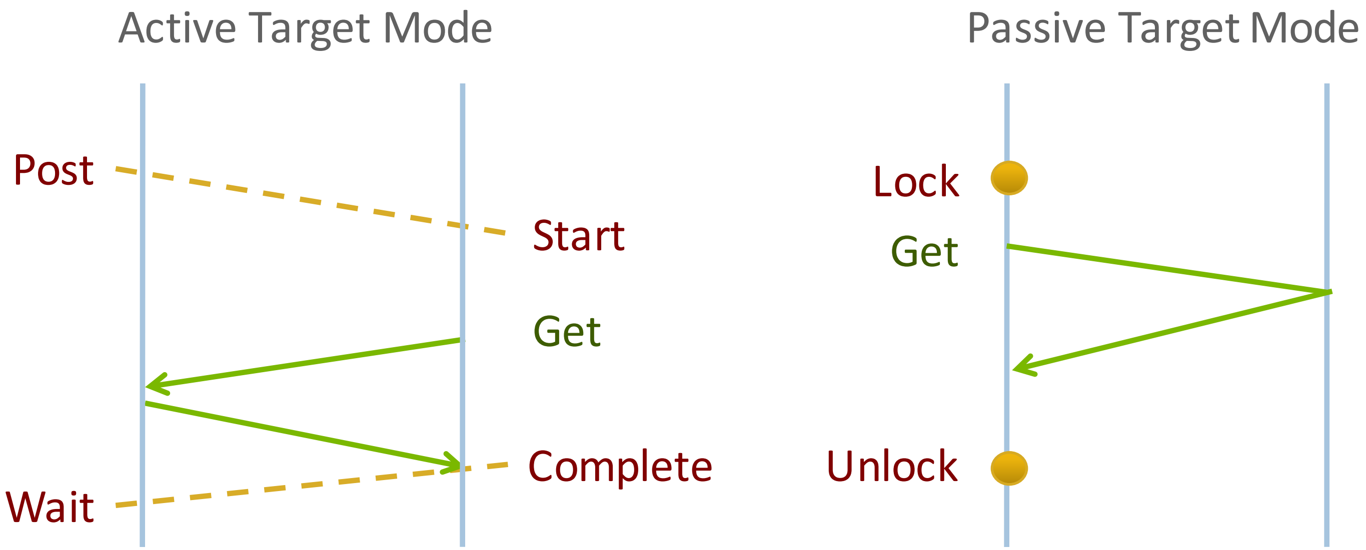
- Everyone does an MPI_WIN_FENCE to open an epoch
- Everyone issues PUT/GET operations to read/write data
- Everyone does an MPI_WIN_FENCE to close the epoch

PSCW Synchronization

- Target: Exposure epoch
  - Opened with MPI_Win_post
  - Closed by MPI_Win_wait
- Origin: Access epoch
  - Opened by MPI_Win_start
  - Closed by MPI_Win_compete
- All may block, to enforce P-S/C-W ordering
  - Processes can be both origins and targets
- Like FENCE, but the target may allow a smaller group of processes to access its data



Lock/Unlock Synchronization



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

Passive Target Synchronization

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

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

- Begin/end passive mode epoch
  - Doesn’t function like a mutex, name can be confusing
  - Communication operations within epoch are all nonblocking
- Lock type
  - SHARED: Other processes using shared can access concurrently
  - EXCLUSIVE: No other processes can access concurrently

When should I use passive mode?

- RMA performance advantages from low protocol overheads
  - Two-sided: Matching, queuing, buffering, unexpected receives, etc…
  - Direct support from high-speed interconnects (e.g. InfiniBand)
- Passive mode: *asynchronous* one-sided communication
  - Data characteristics:
    - Big data analysis requiring memory aggregation
    - Asynchronous data exchange
    - Data-dependent access pattern
  - Computation characteristics:
    - Adaptive methods (e.g. AMR, MADNESS)
    - Asynchronous dynamic load balancing
- Common structure: shared arrays