Lecture 34: One-sided Communication in MPI

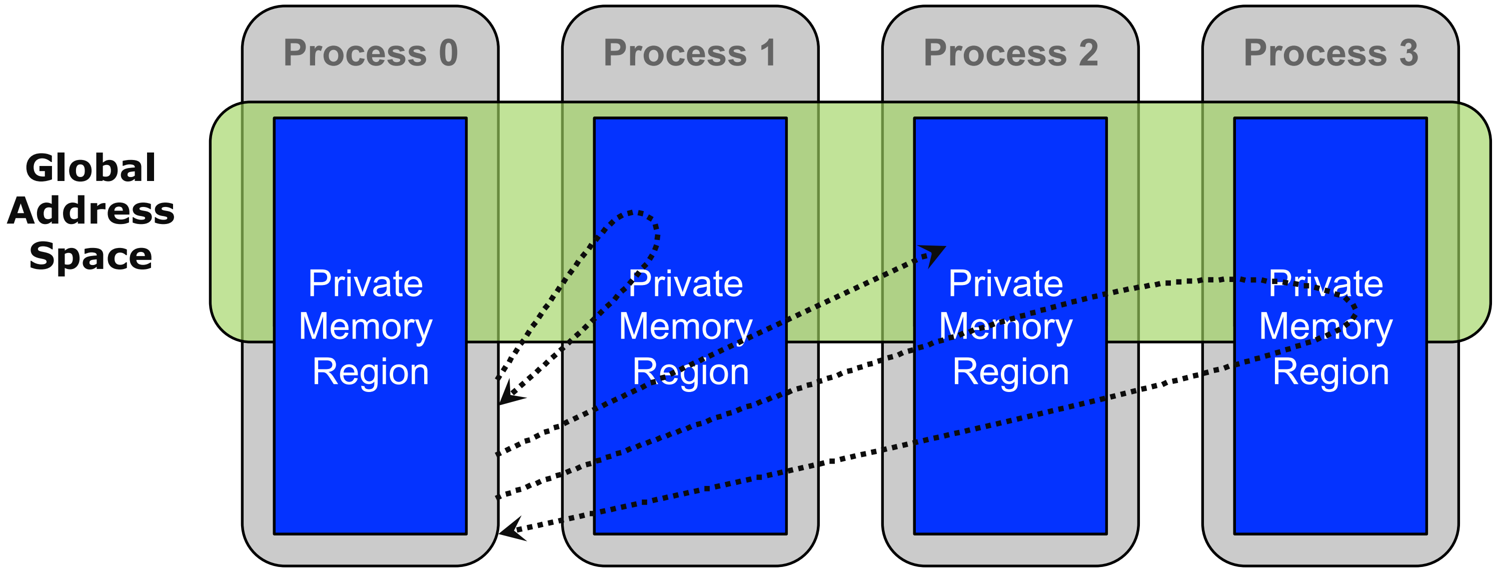
William Gropp
www.cs.illinois.edu/~wgropp

Thanks to

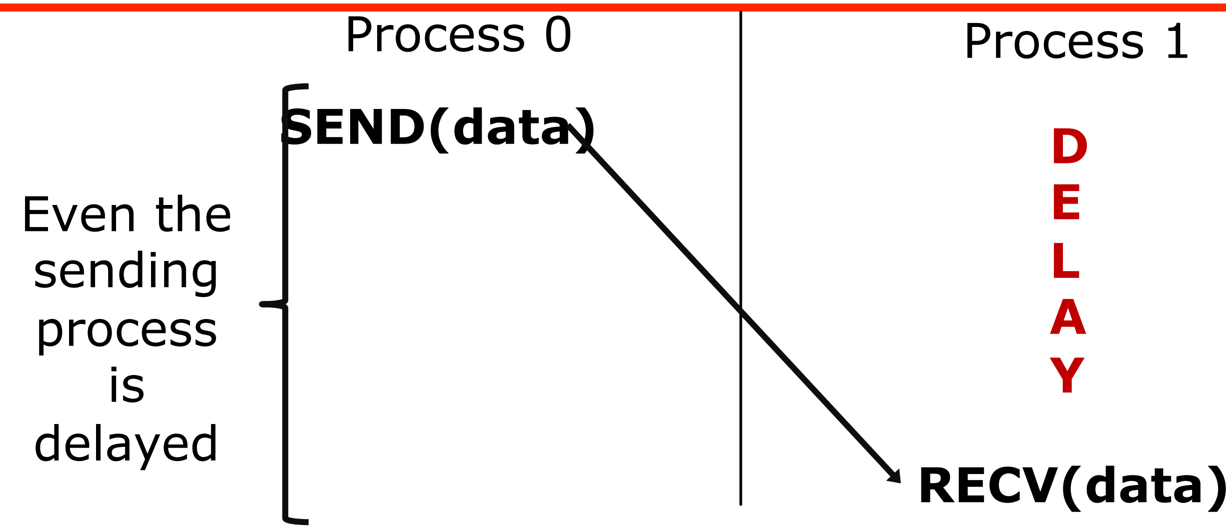
- This material based on the SC14 Tutorial presented by
  - Pavan Balaji
  - William Gropp
  - Torsten Hoefler
  - Rajeev Thakur

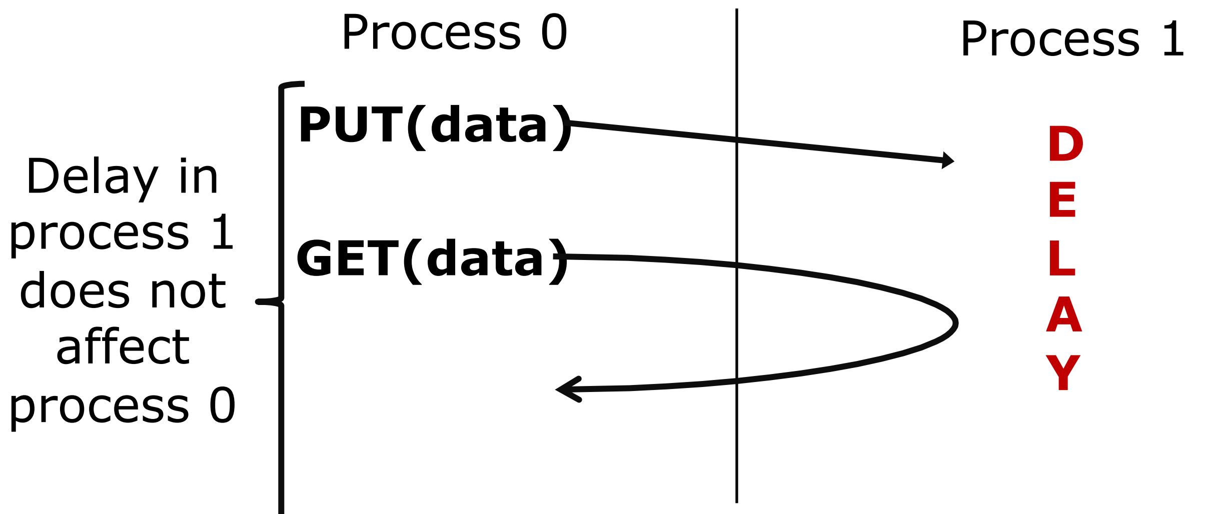
One-Sided Communication

- 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*, but the data locations are known, the send-receive model requires an extra step to determine how many sends-receives 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

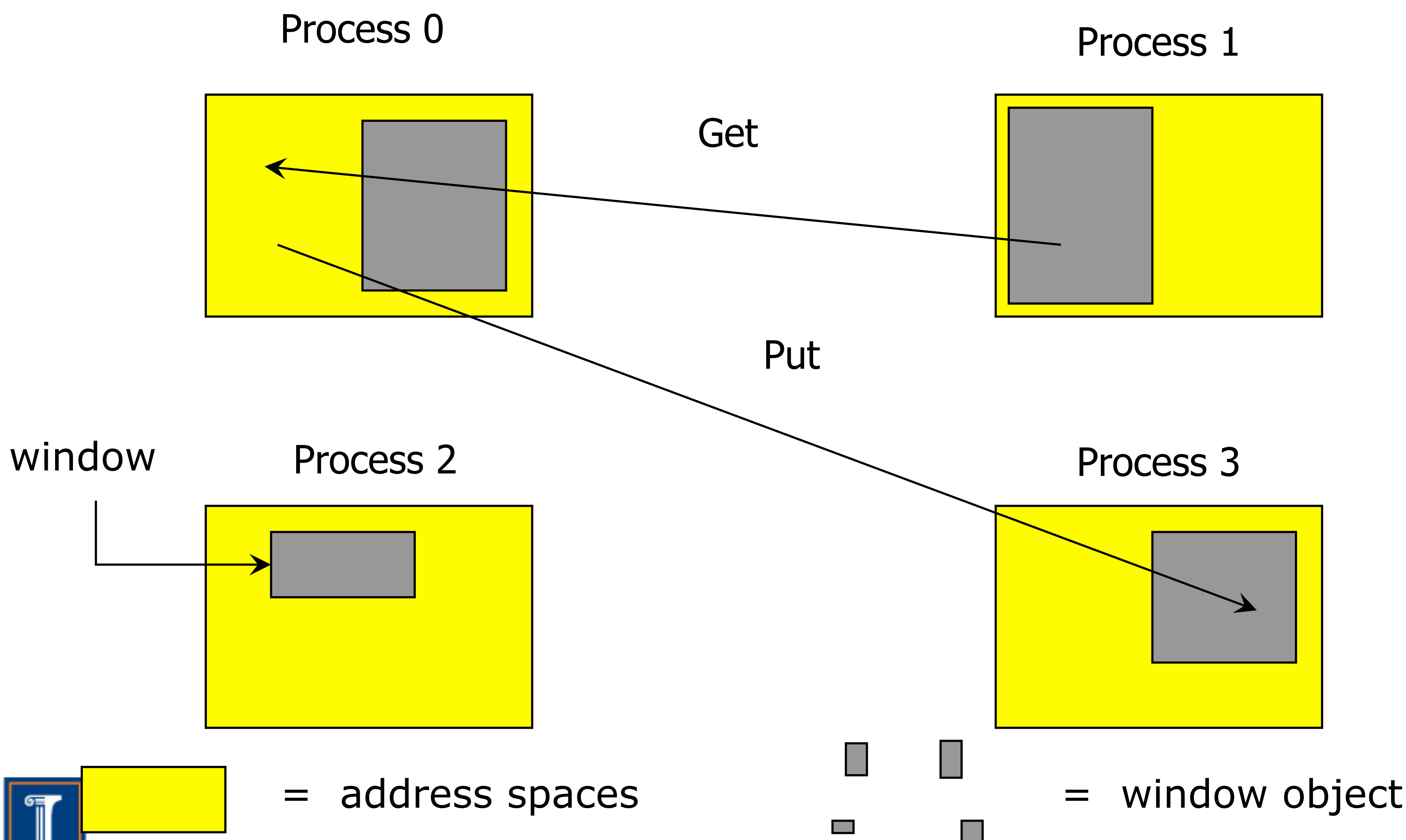
What we need to know in MPI RMA

- 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 object"
- Once a memory region is declared as remotely accessible, all processes in the window object 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

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)
  - win – window object (handle)

Example with MPI_WIN_CREATE

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

    MPI_Init(&argc, &argv);

    /* create private memory */
    MPI_Alloc_mem(1000*sizeof(int), MPI_INFO_NULL, &a);
    /* 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_Free_mem(a);
    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
  - win – window object (handle)

Example with MPI_WIN_ALLOCATE

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

    MPI_Init(&argc, &argv);

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

    /* Array ‘a’ is now accessible from all processes in
     * MPI_COMM_WORLD */

    MPI_Win_free(&win);

    MPI_Finalize(); return 0;
}
```

MPI_WIN_CREATE_DYNAMIC

```
int MPI_Win_create_dynamic(MPI_Info info, 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
- Initially "empty"
  - Application can dynamically attach/detach memory to this window by calling MPI_Win_attach/detach
  - Application can access data on this window only after a memory region has been attached
- Window origin is MPI_BOTTOM
  - Displacements are segment addresses relative to MPI_BOTTOM
  - Must tell others the displacement after calling attach

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 = (int *) 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 accessible from all processes */

    /* undeclare remotely accessible memory */
    MPI_Win_detach(win, a);  free(a);
    MPI_Win_free(&win);

    MPI_Finalize(); return 0;
}
```

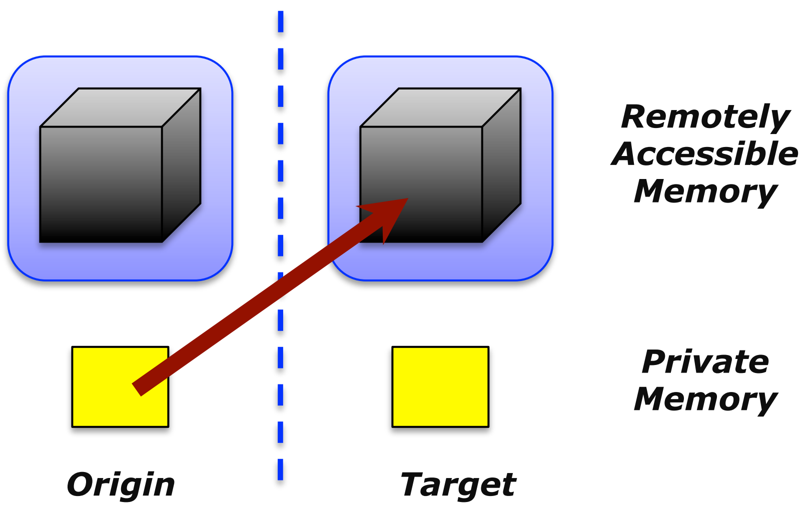
Data movement

- 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: *Put*

```
MPI_Put(void *origin_addr, int origin_count,
    MPI_Datatype origin_dtype, int target_rank,
    MPI_Aint target_disp, int target_count,
    MPI_Datatype target_dtype, MPI_Win win)
```

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

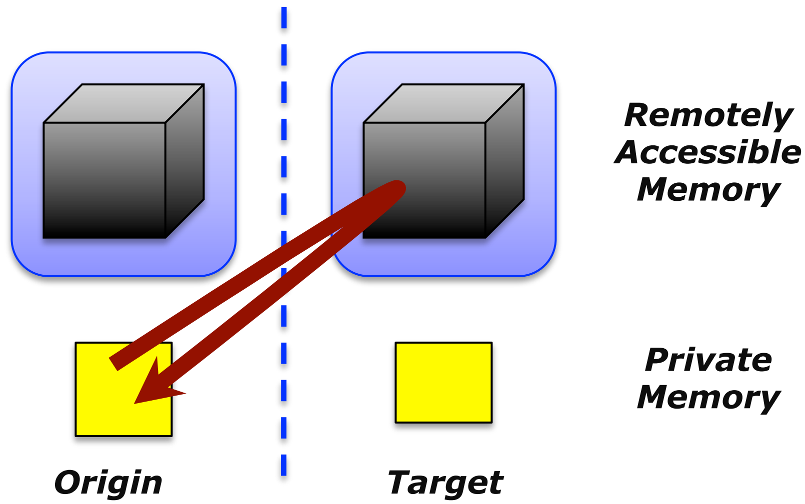


PARALLEL@ILLINOIS

Data movement: *Get*

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

- Move data to origin, from target

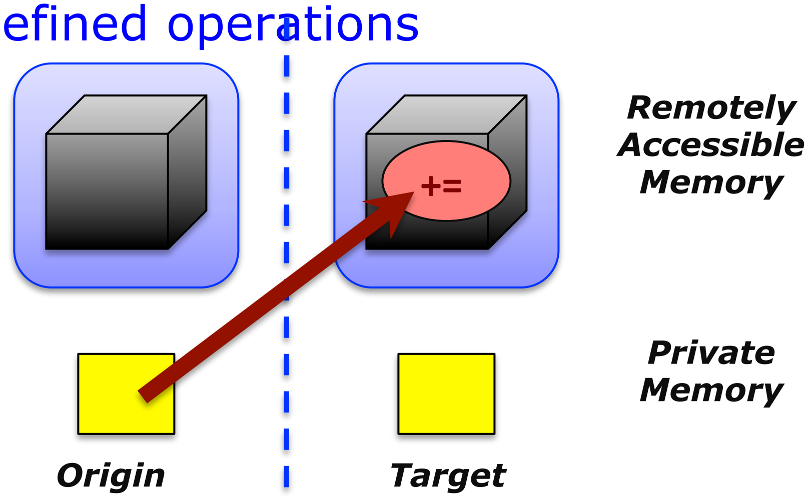


PARALLEL@ILLINOIS

Atomic Data Aggregation: *Accumulate*

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

- Element-wise atomic update operation, similar to a put
  - ♦ Reduces origin and target data into target buffer using op argument as combiner
  - ♦ Predefined ops only, no user-defined operations
- Different data layouts between target/origin OK
  - ♦ Basic type elements must match
- Op = MPI_REPLACE
  - ♦ Implements *f(a,b)=b*
  - ♦ Element-wise atomic PUT

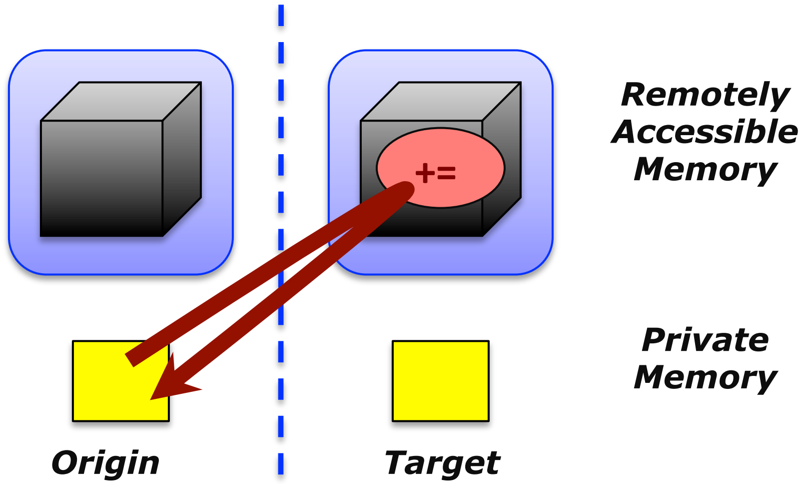


PARALLEL@ILLINOIS

Atomic Data Aggregation: Get Accumulate

```
MPI_Get_accumulate(void *origin_addr, int origin_count,
    MPI_Datatype origin_dtype, void *result_addr,
    int result_count, MPI_Datatype result_dtype,
    int target_rank, MPI_Aint target_disp,
    int target_count, MPI_Datatype target_dype,
    MPI_Op op, MPI_Win win)
```

- Element-wise atomic read-modify-write
  - Op = MPI_SUM, MPI_PROD, MPI_OR, MPI_REPLACE, MPI_NO_OP, …
  - Predefined ops only
- Result stored in target buffer
- Original data stored in result buf
- Different data layouts between target/origin OK
  - Basic type elements must match
- Element-wise atomic get with MPI_NO_OP
- Element-wise atomic swap with MPI_REPLACE



Atomic Data Aggregation: *CAS and FOP*

```
MPI_Fetch_and_op(void *origin_addr, void *result_addr,
    MPI_Datatype dtype, int target_rank,
    MPI_Aint target_disp, MPI_Op op, MPI_Win win)
```

```
MPI_Compare_and_swap(void *origin_addr, void *compare_addr,
    void *result_addr, MPI_Datatype dtype, int target_rank,
    MPI_Aint target_disp, MPI_Win win)
```

- FOP: Simpler version of MPI_Get_accumulate
  - All buffers share a single predefined datatype
  - No count argument (it's always 1)
  - Simpler interface allows hardware optimization
- CAS: Atomic swap if target value is equal to compare value

Ordering of Operations in MPI RMA

- No guaranteed ordering for Put/Get operations
- Result of concurrent Puts to the same location undefined
- Result of Get concurrent Put/Accumulate undefined
  - Can be garbage in both cases
- Result of concurrent accumulate operations to the same location are defined according to the order in which the occurred
  - Atomic put: Accumulate with op = MPI_REPLACE
  - Atomic get: Get_accumulate with op = MPI_NO_OP
- Accumulate operations from a given process are ordered by default
  - User can tell the MPI implementation that ordering is not required as optimization hint
  - You can ask for only the needed orderings, e.g., RAW (read-after-write), WAR, RAR, or WAW

RMA Synchronization Models

- RMA data access model
  - When is a process allowed to read/write remotely accessible memory?
  - When is data written by process X is available for process Y to read?
  - RMA synchronization models define these semantics
- Three synchronization models provided by MPI:
  - Fence (active target)
  - Post-start-complete-wait (generalized active target)
  - Lock/Unlock (passive target)
- Data accesses occur within "epochs"
  - *Access epochs*: contain a set of operations issued by an origin process
  - *Exposure epochs*: enable remote processes to access and/or update a target's window
  - Epochs define ordering and completion semantics
  - Synchronization models provide mechanisms for establishing epochs
    - E.g., starting, ending, and synchronizing epochs

Active and Passive targets

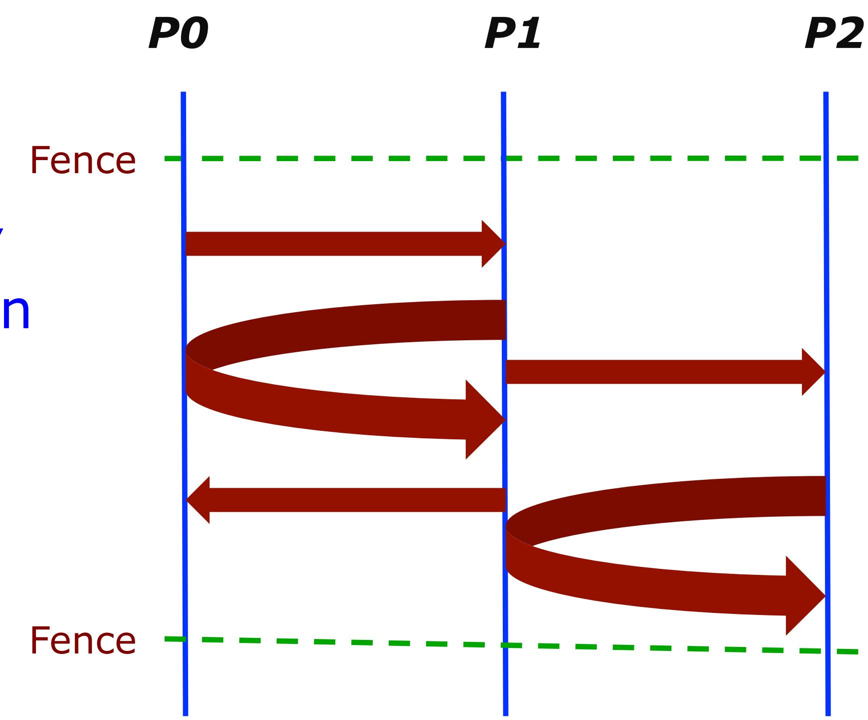
Active target: - data moved from one proc to other. Both participate.
- Similar to message passing, but the target node only participates in synchronization.
- target window is accessed only within exposure epoch.

Passive target: - Only the origin process participates.
- target process does not participate explictly.
- no concept of exposure epoch.

Fence: Active Target Synchronization

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

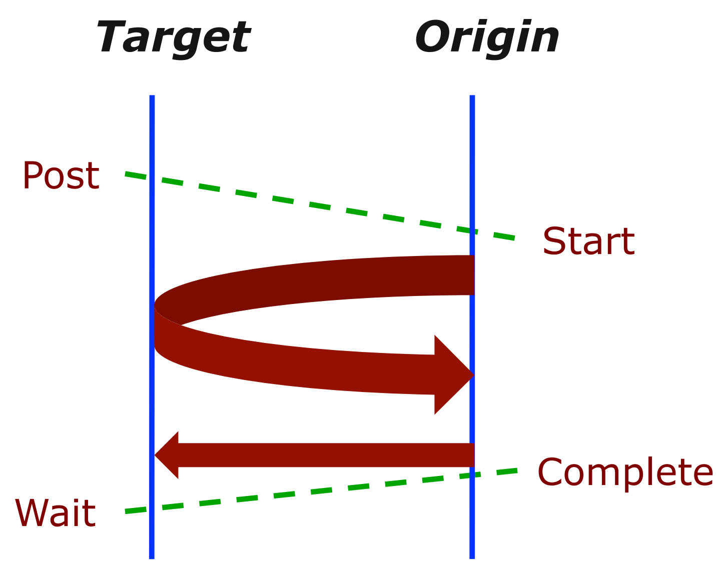
- Collective synchronization model
- Starts *and* ends access and exposure epochs on all processes in the window
- All processes in group of "win" do an MPI_WIN_FENCE to open an epoch
- Everyone can issue PUT/GET operations to read/write data
- Everyone does an MPI_WIN_FENCE to close the epoch
- All operations complete at the second fence synchronization



PSCW: Generalized Active Target Synchronization

```
MPI_Win_post/start(MPI_Group grp, int assert, MPI_Win win)
MPI_Win_complete/wait(MPI_Win win)
```

- Like FENCE, but origin and target specify who they communicate with
- 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_complete
- All synchronization operations may block, to enforce P-S/C-W ordering
  - Processes can be both origins and targets



PARALLEL@ILLINOIS

Post/Start — Complete/Wait

The synchronization between post and start ensures
- the put call of the origin process does not start until the target process exposes the window (with the post call);

The target process will expose the window
- only after preceding local accesses to the window have completed.

The synchronization between complete and wait ensures that
- the put call of the origin process completes before the window is unexposed (with the wait call).

The target process will execute following local accesses to the target window only after the wait returned.

Using Active Target Synchronization

- Active target RMA works well for many BSP-style program
  - Halo exchange
  - Dense linear algebra
- How might you write the dense matrix-vector multiply using
  - MPI_Get: Instead of Allgather
  - MPI_Put: Instead of send/receive
- Do you think using Get instead of Allgather is a good choice at scale? Why or why not? How would use use a performance model to argue your choice?

Passive synchronization

o- Using MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)

- lock_type: MPI_LOCK_EXCLUSIVE or MPI_LOCK_SHARED
- rank: of the target (??)
- assert - keep it to 0.

- Starts an RMA access epoch.

o- MPI_Win_unlock (int rank, MPI_Win win)

- completes an RMA access epoch started by a call to MPI_Win_lock.

```
while(!converged(A)){
 update(A);
 MPI_Win_fence(0, win);
 for(i=0; i < toneighbors; i++)
   MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
                    todisp[i], 1, totype[i], win);
 MPI_Win_fence(0, win);
 }
```

```
while(!converged(A)){
 update(A);
 MPI_Win_post(fromgroup, 0, win);
 MPI_Win_start(togroup, 0, win); // may wait for post
 for(i=0; i < toneighbors; i++)
   MPI_Put(&frombuf[i], 1, fromtype[i], toneighbor[i],
                   todisp[i], 1, totype[i], win);
 MPI_Win_complete(win);
 MPI_Win_wait(win);  // blocks for complete.
 }
```

Etc.

o- Semantics of RMA communication.

- Public view and private view.

o- Do not access local locations during update.

Resources

http://mpi-forum.org/docs/mpi-2.0