Exchanging Data with RMA

Revisiting Mesh Communication

- Recall how we designed the parallel implementation
  - Determine source and destination data
- Do not need full generality of send/receive
  - Each process can completely define what data needs to be moved to itself, relative to each processes local mesh
    - *Each process can "get" data from its neighbors*
  - Alternately, each can define what data is needed by the neighbor processes
    - *Each process can "put" data to its neighbors*

Remote Memory Access

- Separates data transfer from indication of completion (synchronization)
- In message-passing, they are combined

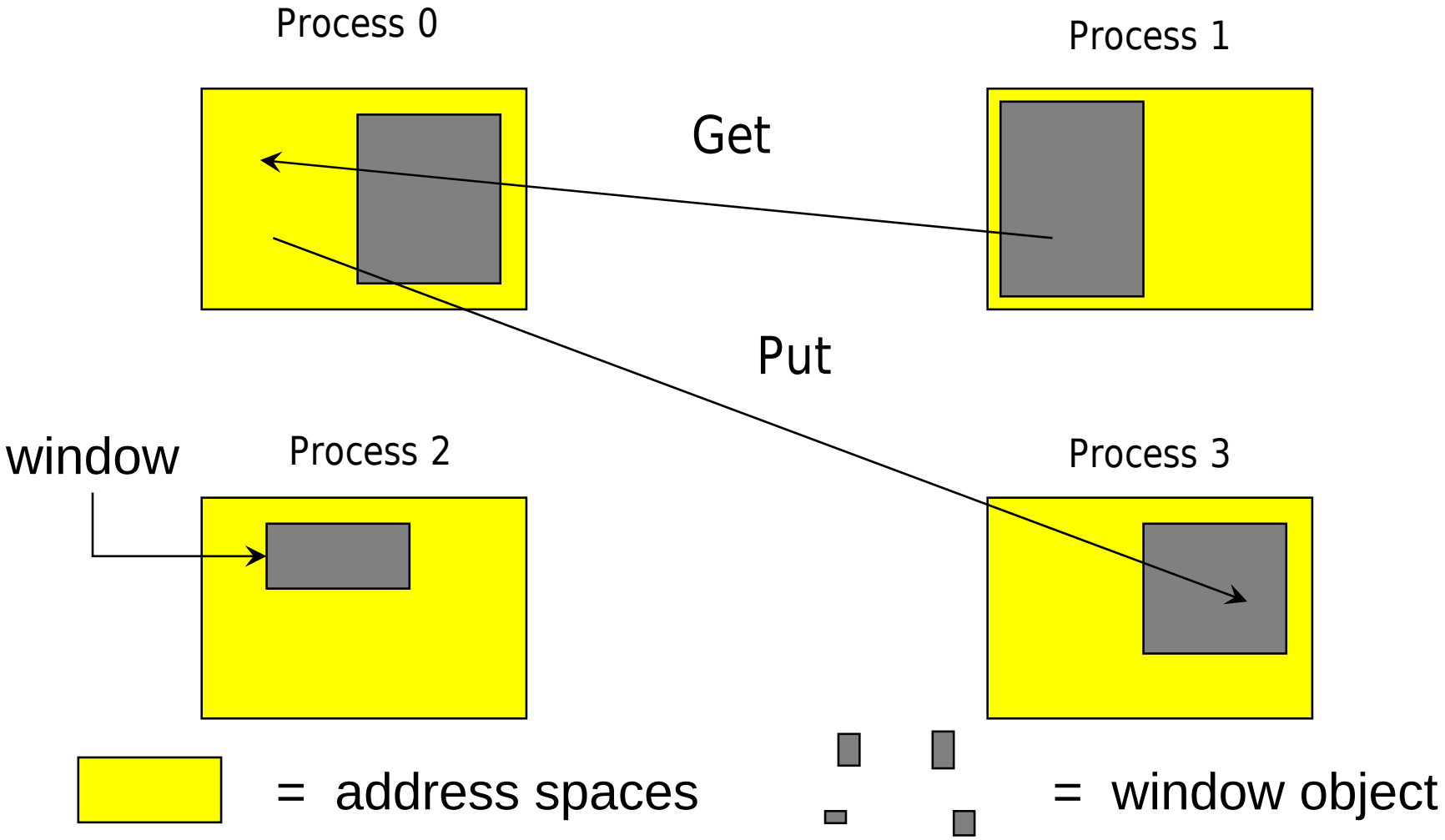
Proc 0	Proc 1
store	
send	receive
	load

Proc 0		Proc 1
fence		fence
put		
fence		fence
		load
	or	
store		
fence		fence
		get

Remote Memory Access in MPI-2 (also called One-Sided Operations)

- Goals of MPI-2 RMA Design
  - Balancing efficiency and portability across a wide class of architectures
    - *shared-memory multiprocessors*
    - *NUMA architectures*
    - *distributed-memory MPP's, clusters*
    - *Workstation networks*
  - Retaining "look and feel" of MPI-1
  - Dealing with subtle memory behavior issues: cache coherence, sequential consistency

Remote Memory Access Windows and Window Objects

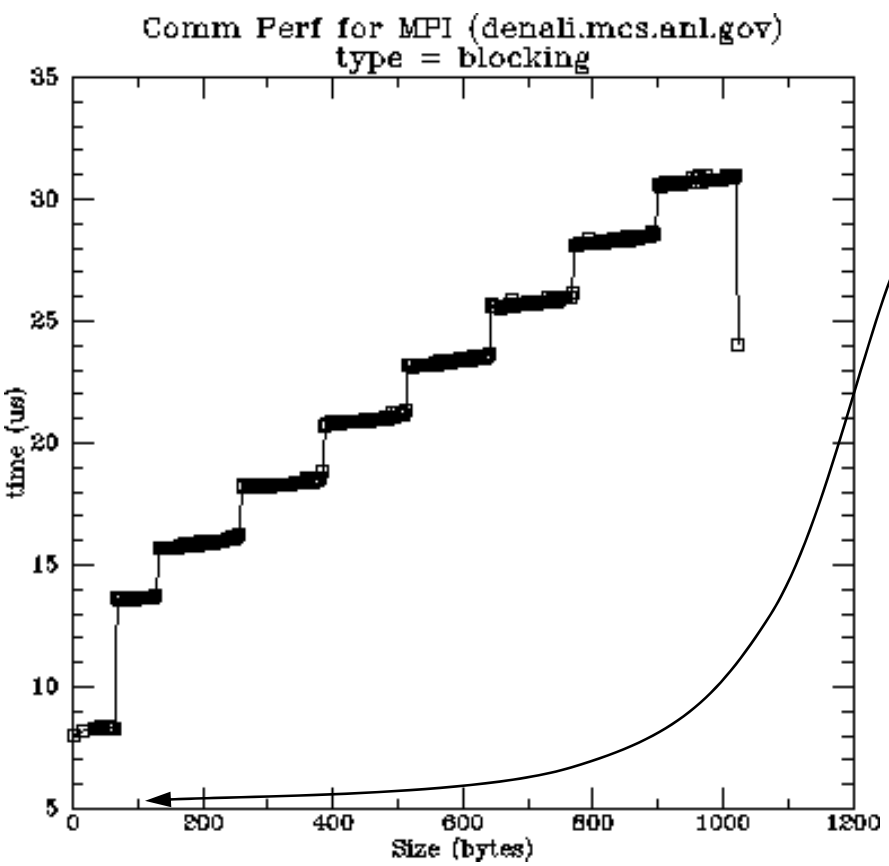


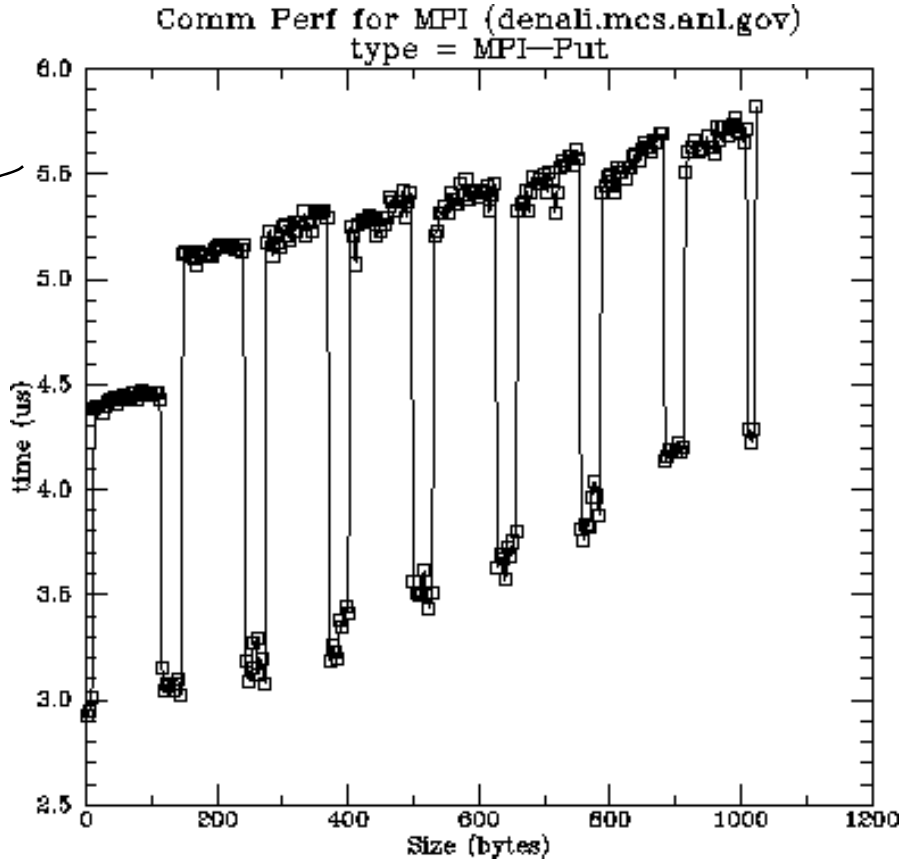
Basic RMA Functions for Communication

- **`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_Ge`**t 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**

Performance of RMA





Caveats: On SGI, MPI_Put uses specially allocated memory

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

RMA Window Objects

```
MPI_Win_create(base, size, disp_unit, info,
               comm, win)
```

- Exposes memory given by **`(base, size)`** to RMA operations by other processes in **`comm`**
- **`win`** is window object used in RMA operations
- **`disp_unit`** scales displacements:
  - 1 (no scaling) or **`sizeof(type)`**, where window is an array of elements of type **`type`**
  - Allows use of array indices
  - Allows heterogeneity

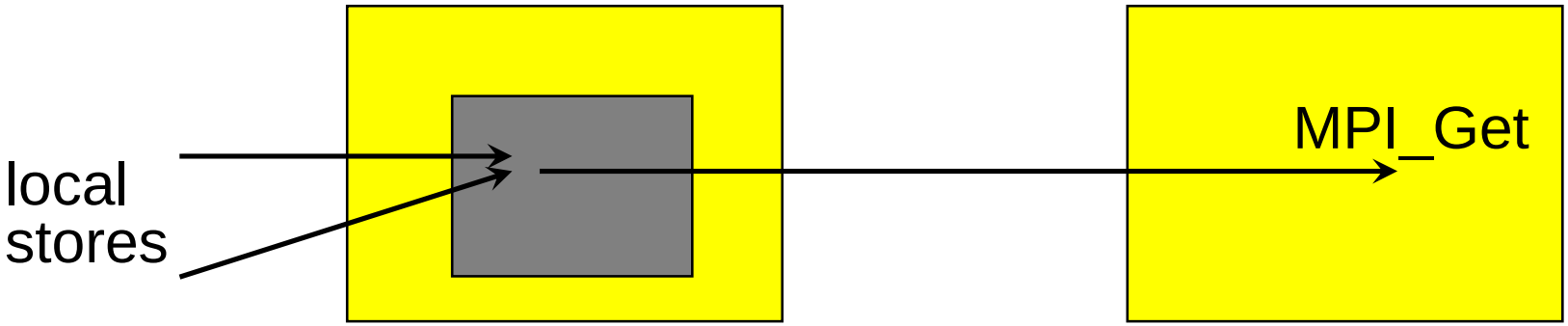
RMA Communication Calls

- **`MPI_Put`** - stores into remote memory
- **`MPI_Get`** - reads from remote memory
- **`MPI_Accumulate`** - updates remote memory
- All are non-blocking: data transfer is described, maybe even initiated, but may continue after call returns
- Subsequent synchronization on window object is needed to ensure operations are complete

Put, Get, and Accumulate

- **`MPI_Put(origin_addr, origin_count,`**
  `origin_datatype,`
  `target_rank, target_offset,`
  `target_count, target_datatype,`
  `window)`
- **`MPI_Get( ... )`**
- **`MPI_Accumulate( ..., op, ... )`**
- **`op`** is as in **`MPI_Reduce`**, but no user-defined operations are allowed

The Synchronization Issue



- Issue: Which value is retrieved?
  - Some form of synchronization is required between local load/stores and remote get/put/accumulates
- MPI provides multiple forms

Synchronization with Fence

Simplest methods for synchronizing on window objects:

- **`MPI_Win_fence`** - like barrier, supports BSP model

Process 0	Process 1
MPI_Win_fence(win)	MPI_Win_fence(win)
MPI_Put MPI_Put	
MPI_Win_fence(win)	MPI_Win_fence(win)

Mesh Exchange Using MPI RMA

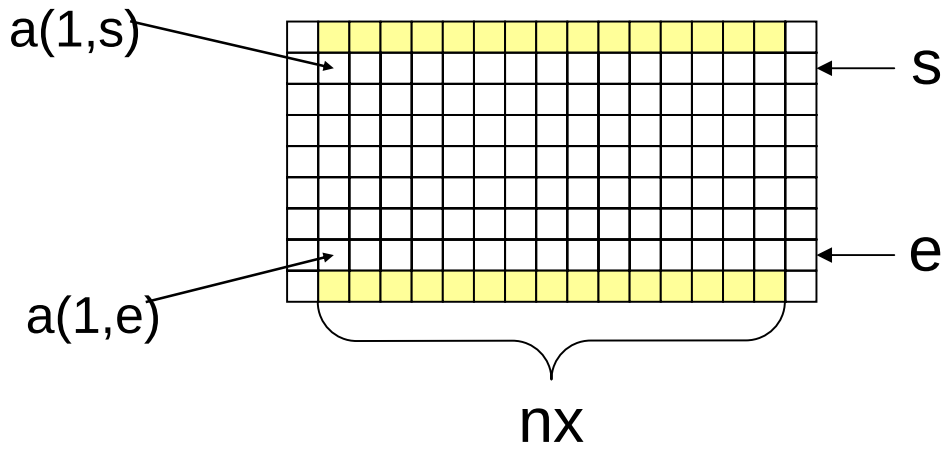
- Define the windows
  - Why – safety, options for performance (later)
- Define the data to move
- Mark the points where RMA can start and where it must complete (e.g., fence/put/put/fence)

Outline of 1D RMA Exchange

- Create Window object
- Computing target offsets
- Exchange operation

Computing the Offsets

- Offset to top ghost row
  - 1
- Offset to bottom ghost row
  - 1 + (# cells in a row)*(# of rows – 1)
  - = 1 + (nx + 2)*(e – s + 2)



Fence Life Exchange Code Walkthrough

- Points to observe
  - MPI_Win_fence is used to separate RMA accesses from non-RMA accesses
    - *Both starts and ends data movement phase*
  - Any memory may be used
    - *No special malloc or restrictions on arrays*
  - Uses same exchange interface as the point-to-point version

See mlife-fence.c pp. 1-3 for code example.

Comments on Window Creation

- MPI-2 provides MPI_SIZEOF for Fortran users
  - Not universally implemented
  - Use MPI_Type_size for portability
- Using a displacement size corresponding to a basic type allows use of put/get/accumulate on heterogeneous systems
  - Even when the sizes of basic types differ
- Displacement size also allows easier computation of offsets in terms of array index instead of byte offset

More on Fence

- MPI_Win_fence is collective over the group of the window object
- MPI_Win_fence is used to *separate*, not just complete, RMA and local memory operations
  - That is why there are *two* fence calls
- Why?
  - MPI RMA is designed to be portable to a wide variety of machines, including those without cache coherent hardware (including some of the fastest machines made)
  - See performance tuning for more info

Scalable Synchronization with Post/Start/Complete/Wait

- Fence synchronization is not scalable because it is collective over the group in the window object
- MPI provides a second synchronization mode: *Scalable Synchronization*
  - Uses four routines instead of the single MPI_Win_fence:
    - *2 routines to mark the begin and end of calls to RMA routines*
      - MPI_Win_start, MPI_Win_complete
    - *2 routines to mark the begin and end of access to the memory window*
      - MPI_Win_post, MPI_Win_wait
- P/S/C/W allows synchronization to be performed only among communicating processes

Synchronization with P/S/C/W

- Origin process calls MPI_Win_start and MPI_Win_complete
- Target process calls MPI_Win_post and MPI_Win_wait

Process 0	Process 1
MPI_Win_start(target_grp)	MPI_Win_post(origin_grp)
MPI_Put MPI_Put	
MPI_Win_complete(target_grp)	MPI_Win_wait(origin_grp)

P/S/C/W Life Exchange Code Walkthrough

- Points to Observe
  - Use of MPI group routines to describe neighboring processes
  - No change to MPI_Put calls
    - *You can start with MPI_Win_fence, then switch to P/S/C/W calls if necessary to improve performance*

See mlife-pscw.c pp. 1-4 for code example.