中国科学院大学
University of Chinese Academy of Sciences

Introduction to MPI

Part3. *Advanced Topics*

人工智能技术学院
缪青海
miaoqh@ucas.ac.cn

Content

- **Topology Mapping**
- Remote Memory Access
- Others:
  - Nonblocking Collective Communication
  - Hybrid Programming with Threads, GPUs
  - MPI I/O

MPI Virtual Topologies

- In terms of MPI, a virtual topology describes a mapping/ordering of MPI processes into a geometric "shape".

- Virtual topologies are built upon MPI communicators and groups.
  - It is an attribute of processes only in the group.

MPI Virtual Topologies

- MPI topologies are virtual:
  - The term “Virtual Topology” gives this main idea: **machine independent**
  - no relation between the physical structure of the parallel machine and the process topology.
- Must be "programmed" by the application developer.

Why use Virtual Topologies

- **Convenience**:
  - Virtual topologies may be useful for applications with specific communication patterns - patterns that match an MPI topology structure.
  - For example, a Cartesian topology might prove convenient for an application that requires 4-way nearest neighbor communications for grid based data.

Why use Virtual Topologies

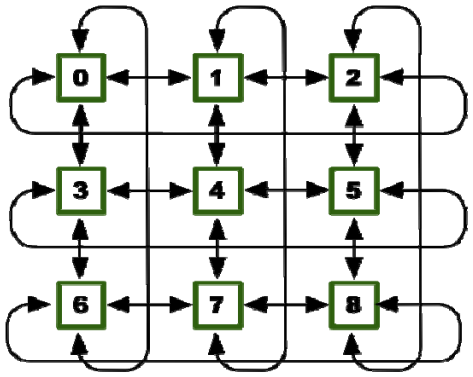
- **Communication Efficiency** :
  - Some hardware architectures may impose penalties for communications between successively distant "nodes".
  - A particular implementation may optimize process mapping based upon the physical characteristics of a given parallel machine.
  - The mapping of processes into an MPI virtual topology is dependent upon the MPI implementation, and may be totally ignored

MPI Topology History

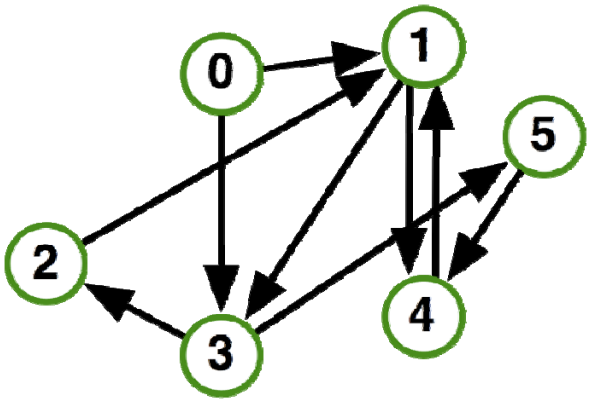
- Convenience functions (in MPI-1)
  - Useful especially for Cartesian topologies
    - Query neighbors in n-dimensional space
  - Graph topology: each rank specifies full graph ☹
- Scalable Graph topology (MPI-2.2)
  - Graph topology: each rank specifies its neighbors **or** an arbitrary subset of the graph
- Neighborhood collectives (MPI-3.0)
  - Adding communication functions defined on graph topologies (neighborhood of distance one)

Types of Virtual Topologies

- There are two types of MPI topologies:



Cartesian Topology



Graph Topology

Cartesian virtual topology

```
MPI_Cart_create(MPI_Comm comm_old, int ndims, const int *dims,
                const int *periods, int reorder, MPI_Comm *comm_cart)
```

- Specify ndims-dimensional topology
  - Optionally periodic in each dimension (Torus)
- Some processes may return MPI_COMM_NULL
  - Product sum of dims must be <= P
- Reorder argument allows for topology mapping
  - Each calling process may have a new rank in the created communicator
  - Data has to be remapped manually

MPI_Cart_create Example

```
int dims[3] = {5,5,5};
int periods[3] = {1,1,1};
MPI_Comm topocomm;
MPI_Cart_create(comm, 3, dims, periods, 0, &topocomm);
```

- Creates logical 3-d Torus of size 5x5x5
- But we're starting MPI processes with a one-dimensional argument (-p X)
  - User has to determine size of each dimension
  - Often as "square" as possible, MPI can help!

MPI_Dims_create

```
MPI_Dims_create(int nnodes, int ndims, int *dims)
```

- Create dims array for Cart_create with nnodes and ndims
  - Dimensions are as close as possible (well, in theory)
- Non-zero entries in dims will not be changed
  - nnodes must be multiple of all non-zeroes

MPI_Dims_create Example

```
int p;
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Dims_create(p, 3, dims);

int periods[3] = {1,1,1};
MPI_Comm topocomm;
MPI_Cart_create(comm, 3, dims, periods, 0, &topocomm);
```

- Makes life a little bit easier
  - Some problems may be better with a non-square layout though

Cartesian Query Functions

- Library support and convenience!
  - MPI_Cartdim_get()
    - Gets dimensions of a Cartesian communicator
  - MPI_Cart_get()
    - Gets size of dimensions
  - MPI_Cart_rank()
    - Translate coordinates to rank
  - MPI_Cart_coords()
    - Translate rank to coordinates

Cartesian Communication Helpers

MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)

- Shift in one dimension
  - Dimensions are numbered from 0 to ndims-1
  - Displacement indicates neighbor distance (-1, 1, …)
  - May return MPI_PROC_NULL
- Very convenient, all you need for nearest neighbor communication
  - No “over the edge” though

Cartesian Example

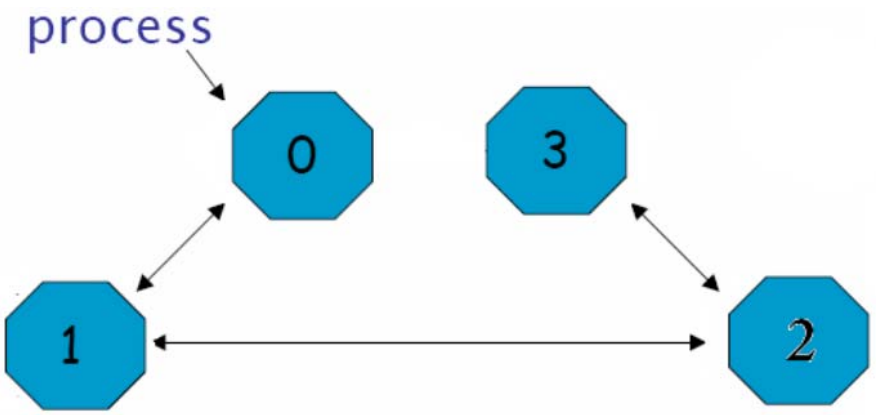
- A simplified mapping of processes into a Cartesian virtual topology (Grid)

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

code/mpi_topology_cartesian.c

Graph Topology

- More generally, the process organizing is described by a graph.
- Elements of Graph Topology:
  - Communication link
  - Nodes in the graph
  - Neighbours of per node
  - Type of mapping

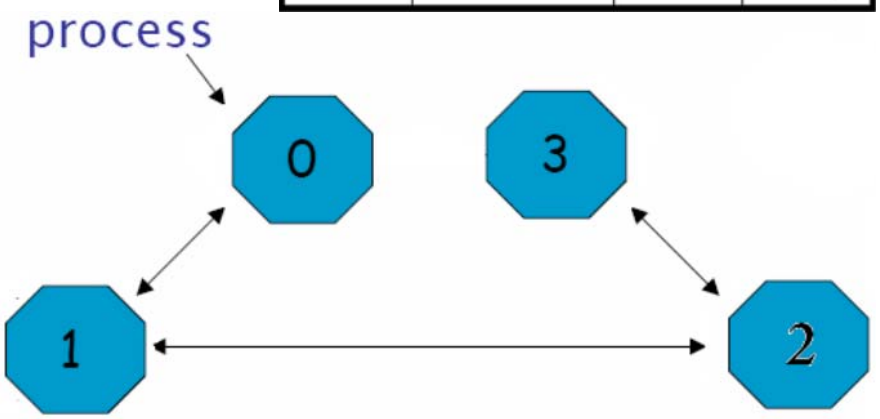


Graph Topology

- Elements of Graph Topology:

Node	Nneighbors	index	edges
0	1	1	1
1	2	3	0,2
2	2	5	1,3
3	1	6	2

- Nodes:
  - Processors
- Lines:
  - Communicators between nodes
- Arrows:
  - Show origins and destinations of links
- Index:
  - array of integers describing node degrees



Distributed Graph

- **MPI_Graph_create is discouraged**
  - Not scalable
  - Not deprecated yet but hopefully soon
- **New distributed interface:**
  - Scalable, allows distributed graph specification
    - Either local neighbors **or** any edge in the graph
  - Specify edge weights
    - Meaning undefined but optimization opportunity for vendors!
  - Info arguments
    - Communicate assertions of semantics to the MPI library
    - E.g., semantics of edge weights

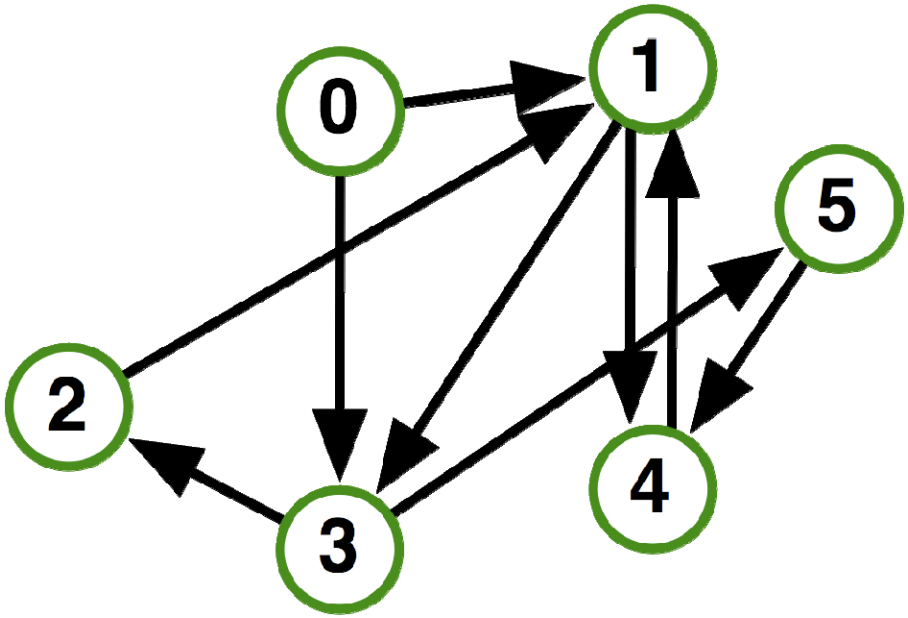
MPI_Dist_graph_create_adjacent

```
MPI_Dist_graph_create_adjacent(MPI_Comm comm_old,
        int indegree, const int sources[], const int sourceweights[],
        int outdegree, const int destinations[],
        const int destweights[], MPI_Info info, int reorder,
        MPI_Comm *comm_dist_graph)
```

- indegree, sources, ~weights – source proc. Spec.
- outdegree, destinations, ~weights – dest. proc. spec.
- info, reorder, comm_dist_graph – as usual
- directed graph
- Each edge is specified twice, once as out-edge (at the source) and once as in-edge (at the dest)

MPI_Dist_graph_create_adjacent

- Process 0:
  - Indegree: 0
  - Outdegree: 2
  - Dests: {3,1}
- Process 1:
  - Indegree: 3
  - Outdegree: 2
  - Sources: {4,0,2}
  - Dests: {3,4}
- …



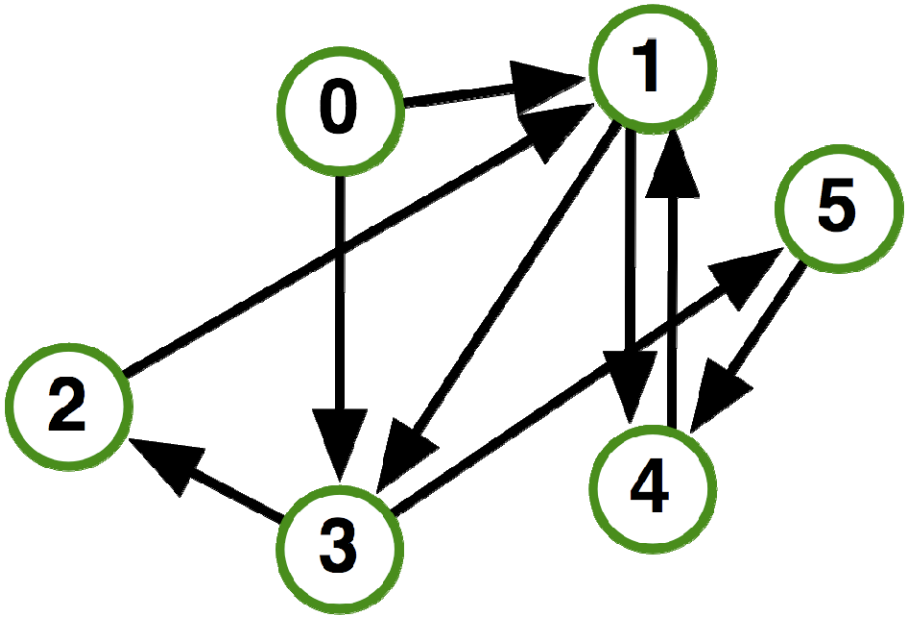
MPI_Dist_graph_create

```
MPI_Dist_graph_create(MPI_Comm comm_old, int n,
        const int sources[], const int degrees[],
        const int destinations[], const int weights[], MPI_Info info,
        int reorder, MPI_Comm *comm_dist_graph)
```

- n – number of source nodes
- sources – n source nodes
- degrees – number of edges for each source
- destinations, weights – dest. processor specification
- info, reorder – as usual
- More flexible and convenient
  - Requires global communication
  - Slightly more expensive than adjacent specification

MPI_Dist_graph_create

- Process 0:
  - N: 2
  - Sources: {0,1}
  - Degrees: {2,1}*
  - Dests: {3,1,4}
- Process 1:
  - N: 2
  - Sources: {2,3}
  - Degrees: {1,1}
  - Dests: {1,2}
- …



* Note that in this example, process 0 specifies only one of the two outgoing edges of process 1; the second outgoing edge needs to be specified by another process

Distributed Graph Neighbor Queries

```
MPI_Dist_graph_neighbors_count(MPI_Comm comm,
    int *indegree,int *outdegree, int *weighted)
```

- Query the number of neighbors of **calling process**
- Returns indegree and outdegree!
- Also info if weighted

Distributed Graph Neighbor Queries

```
MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree,
        int sources[], int sourceweights[], int maxoutdegree,
        int destinations[],int destweights[])
```

- Query the neighbor list of **calling process**
- Optionally return weights

Further Graph Queries

```
MPI_Topo_test(MPI_Comm comm, int *status)
```

- Status is either:
  - MPI_GRAPH (ugs)
  - MPI_CART
  - MPI_DIST_GRAPH
  - MPI_UNDEFINED (no topology)
- Enables to write libraries on top of MPI topologies!

Neighborhood Collectives

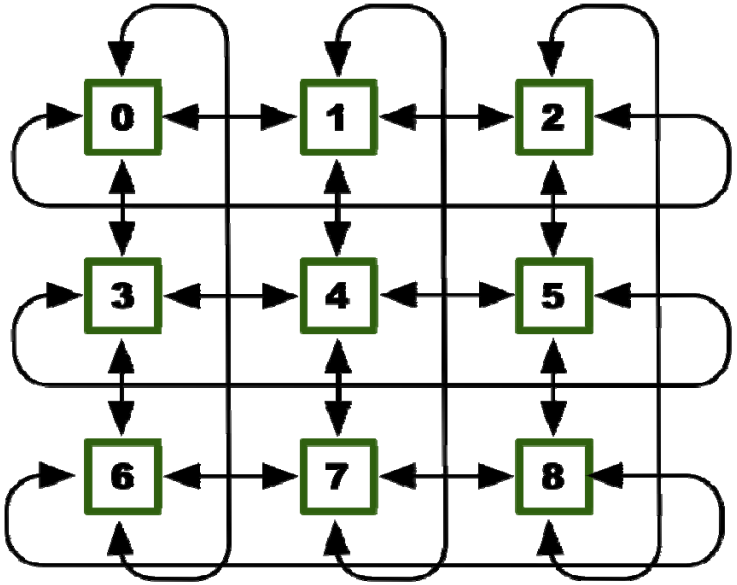
- Topologies implement no communication!
  - Just helper functions
- Collective communications only cover some patterns
  - E.g., no stencil pattern
- Several requests for "build your own collective" functionality in MPI
  - Neighborhood collectives are a simplified version
  - Cf. Datatypes for communication patterns!

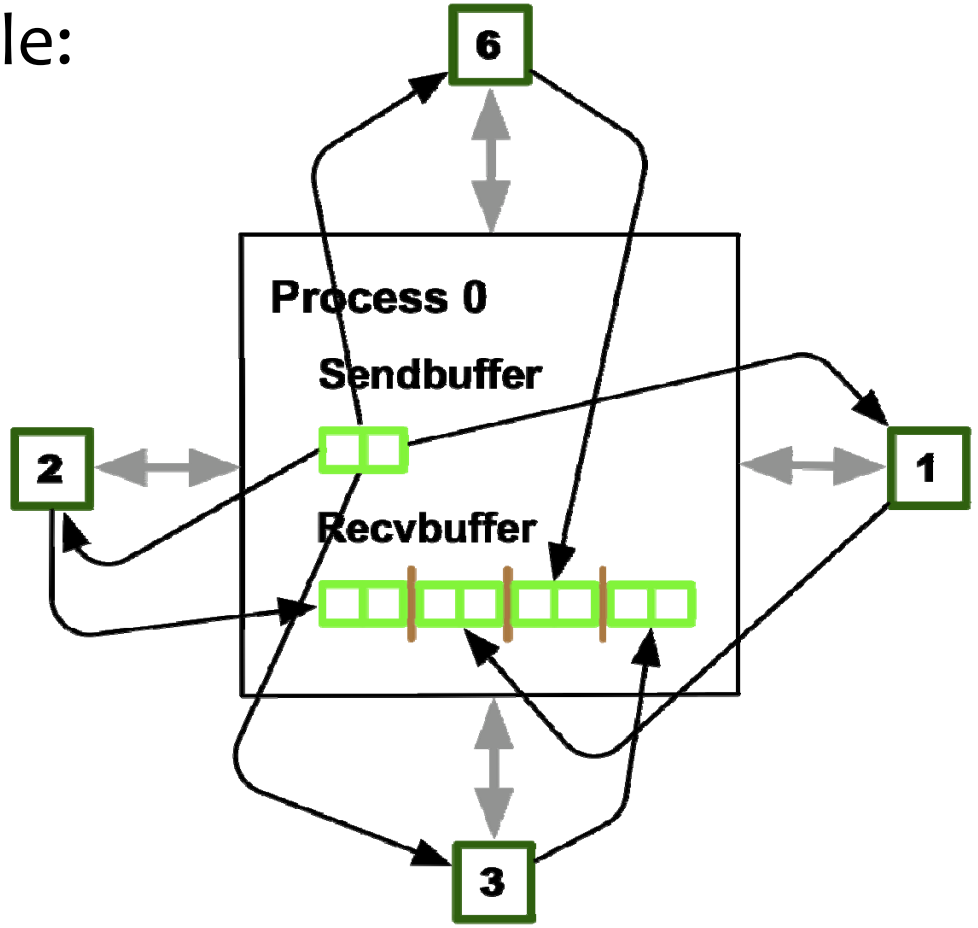
Cartesian Neighborhood Collectives

- Communicate with direct neighbors in Cartesian topology
  - Corresponds to cart_shift with disp=1
  - Collective (all processes in comm must call it, including processes without neighbors)
  - Buffers are laid out as neighbor sequence:
    - Defined by order of dimensions, first negative, then positive
    - 2*ndims sources and destinations
    - Processes at borders (MPI_PROC_NULL) leave holes in buffers (will not be updated or communicated)!

Cartesian Neighborhood Collectives

- Buffer ordering example:





Graph Neighborhood Collectives

- Collective Communication along arbitrary neighborhoods
  - Order is determined by order of neighbors as returned by (dist_)graph_neighbors.
  - Distributed graph is directed, may have different numbers of send/recv neighbors
  - Can express dense collective operations ☺
  - Any persistent communication pattern!

MPI_Neighbor_allgather

```
MPI_Neighbor_allgather(const void* sendbuf, int sendcount,
        MPI_Datatype sendtype, void* recvbuf, int recvcount,
        MPI_Datatype recvtype, MPI_Comm comm)
```

- Sends the same message to all neighbors
- Receives indegree distinct messages
- Similar to MPI_Gather
  - The all prefix expresses that each process is a “root” of his neighborhood
- Vector version for full flexibility

MPI_Neighbor_alltoall

```
MPI_Neighbor_alltoall(const void* sendbuf, int sendcount,
        MPI_Datatype sendtype, void* recvbuf, int recvcount,
        MPI_Datatype recvtype, MPI_Comm comm)
```

- Sends outdegree distinct messages
- Received indegree distinct messages
- Similar to MPI_Alltoall
  - Neighborhood specifies full communication relationship
- Vector and w versions for full flexibility

Nonblocking Neighborhood Collectives

```
MPI_Ineighbor_allgather(…, MPI_Request *req);
MPI_Ineighbor_alltoall(…, MPI_Request *req);
```

- Very similar to nonblocking collectives
- Collective invocation
- Matching in-order (no tags)
  - No wild tricks with neighborhoods! In order matching per communicator!

Code Example

- Code/mpi_topology_graph.c

- Reference:
  - The Scalable Process Topology Interface of MPI 2.2

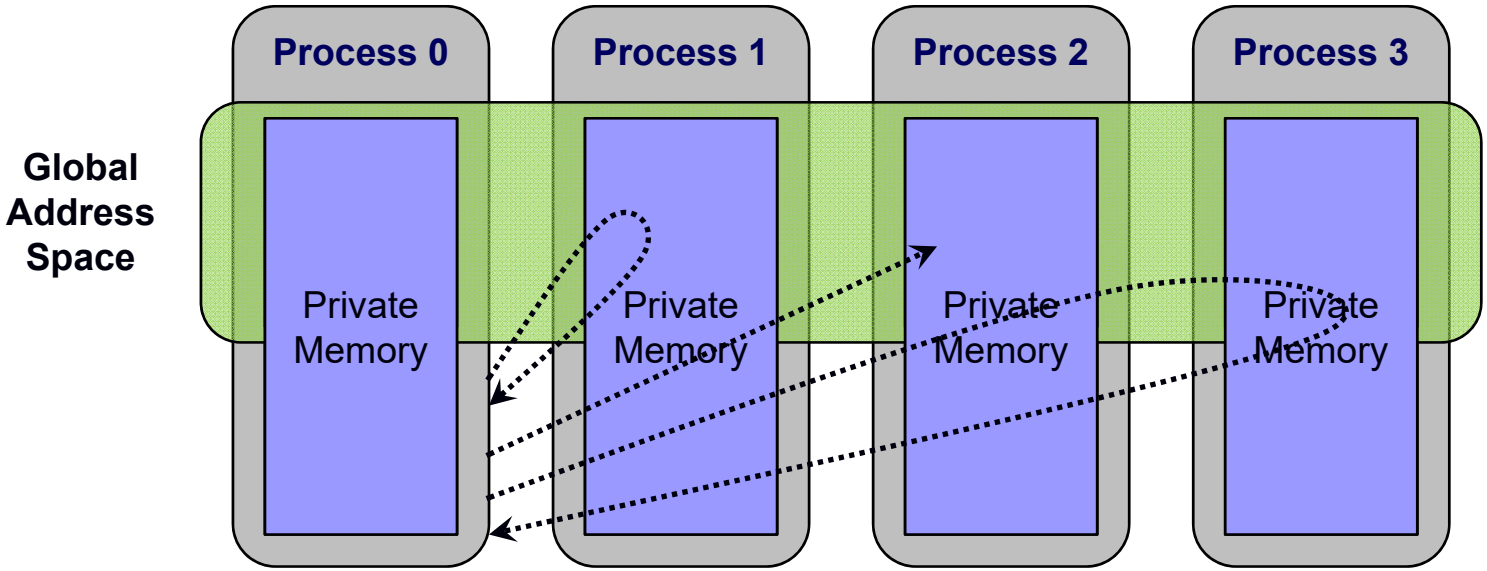
Content

- Topology Mapping
- **Remote Memory Access**
- Others:
  - Nonblocking Collective Communication
  - Hybrid Programming with Threads, GPUs
  - MPI I/O

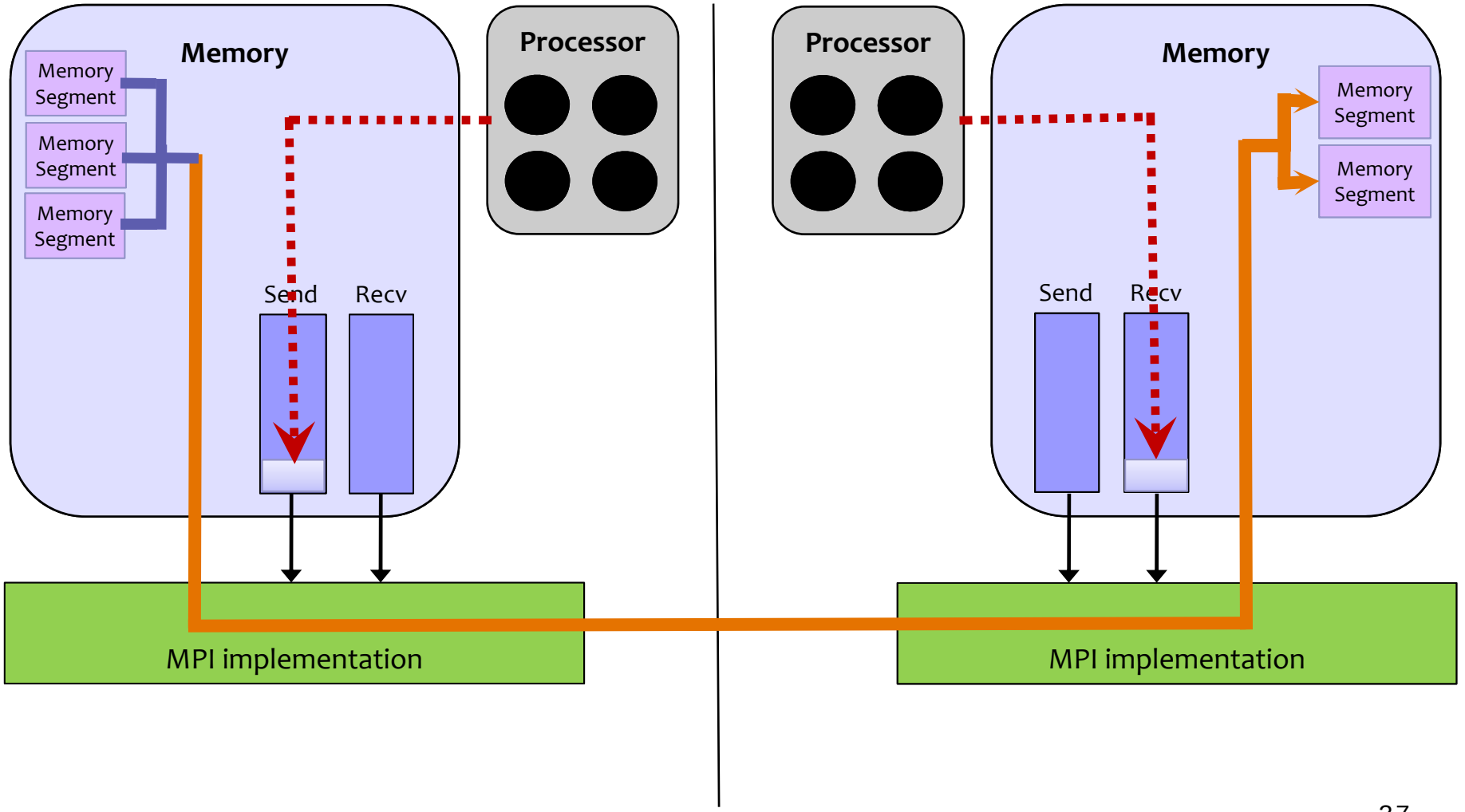
One-sided Communication

- The basic idea of one-sided communication models is to **decouple** data movement with process synchronization
  - Should be able 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

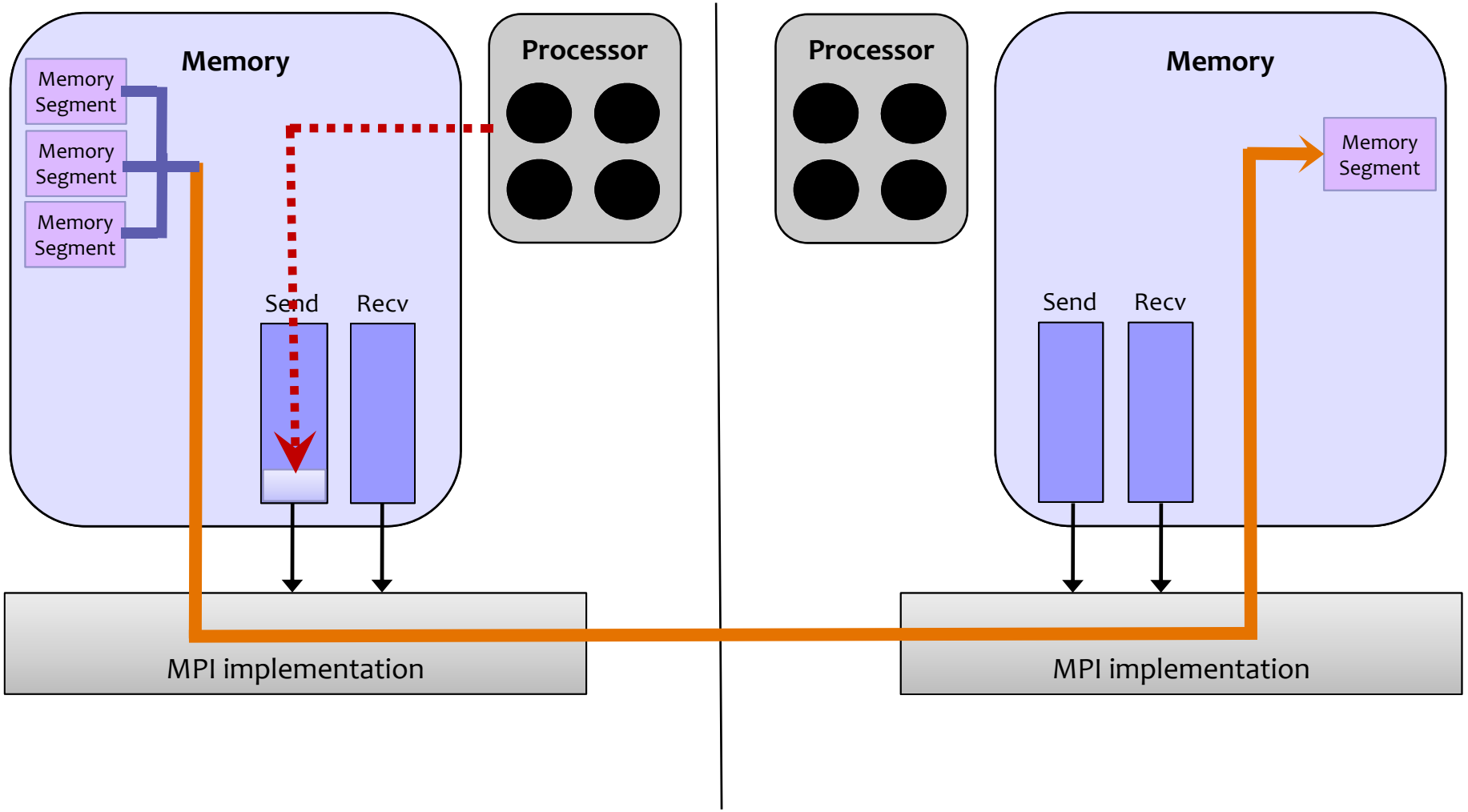
One-sided Communication



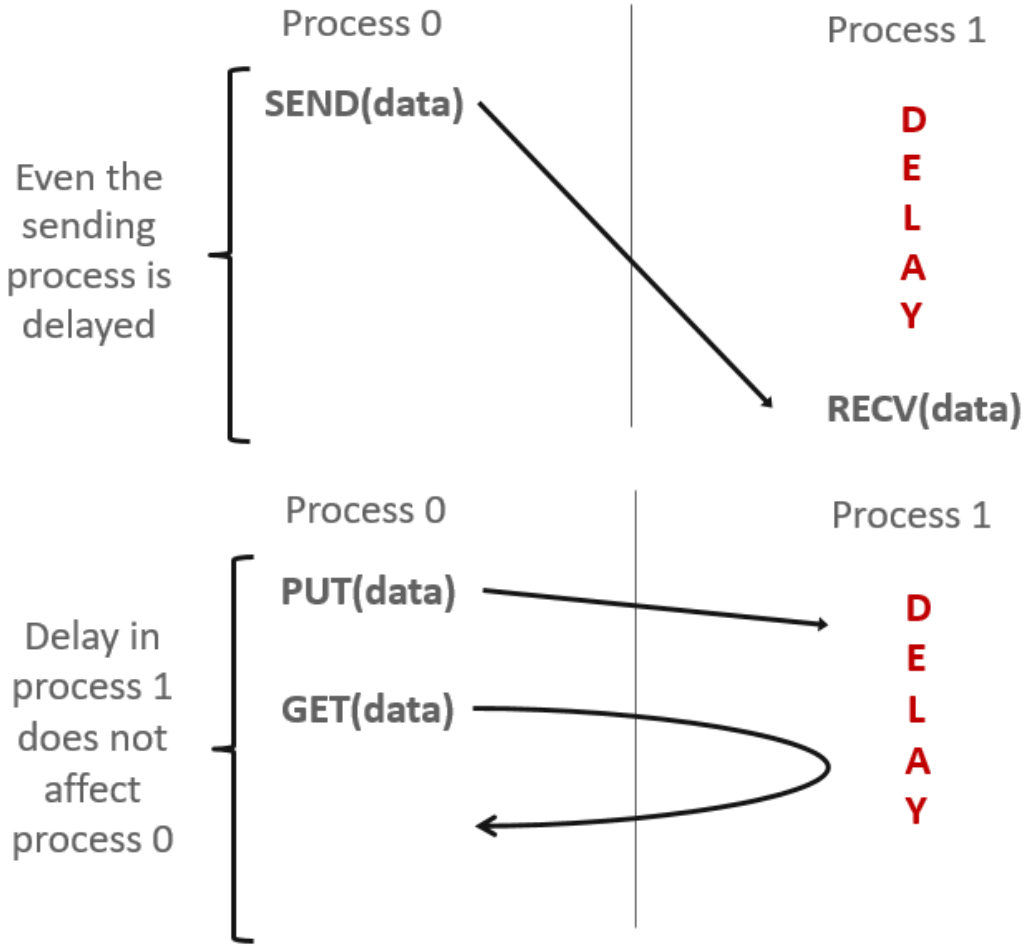
Two-sided Communication Example



One-sided Communication Example



Comparing One-sided and Two-sided



Keys in MPI RMA

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

Creating Public Memory

- Any memory used by a process is, by default, only locally accessible
  - X = malloc(100);
- Once the memory is allocated, 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"

Creating Public Memory

- 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.

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
    - You may want to dynamically add/remove buffers to/from the window
  - **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 (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 (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;
}
```

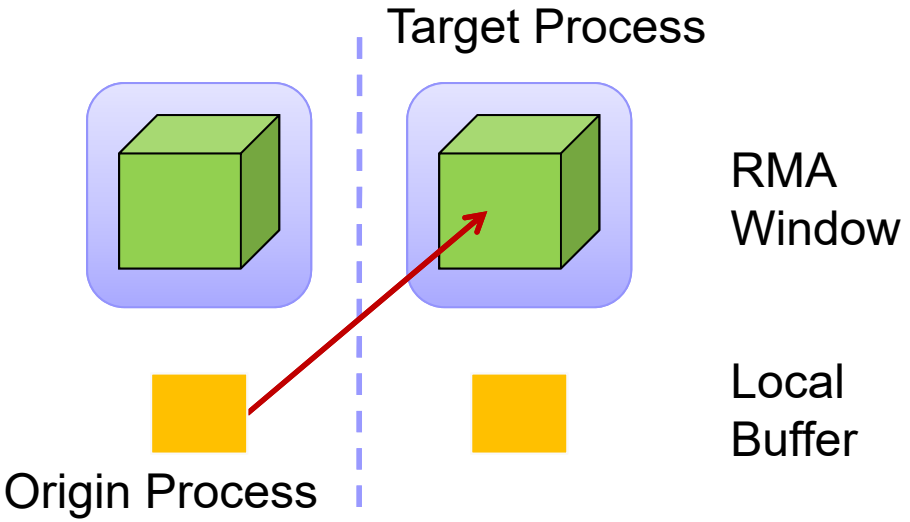
Data movement

- MPI provides ability to read, write and atomically modify data in remotely accessible memory regions
  - MPI_PUT
  - MPI_GET
  - 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_datatype, int target_rank,
        MPI_Aint target_disp, int target_count,
        MPI_Datatype target_datatype, MPI_Win win)
```

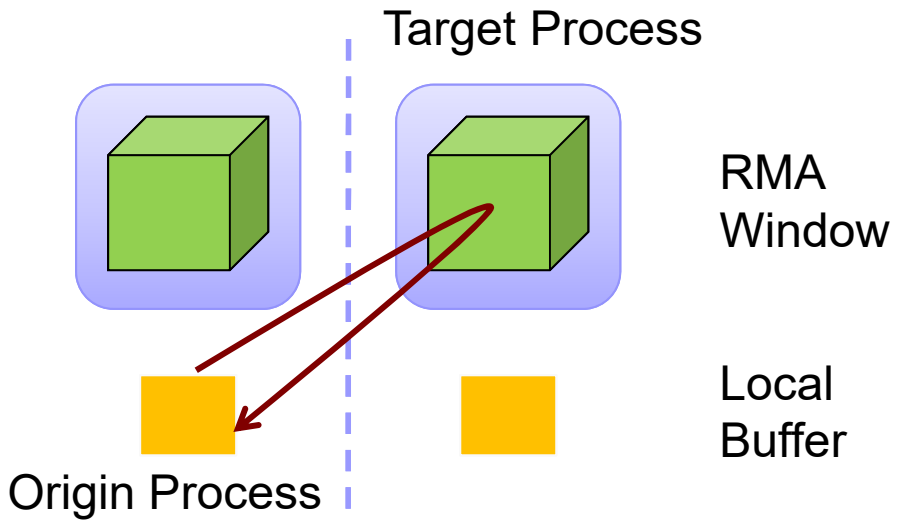
- Move data from origin, to target
- Separate data description triples for origin and target



Data movement: *Get*

```
MPI_Get(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)
```

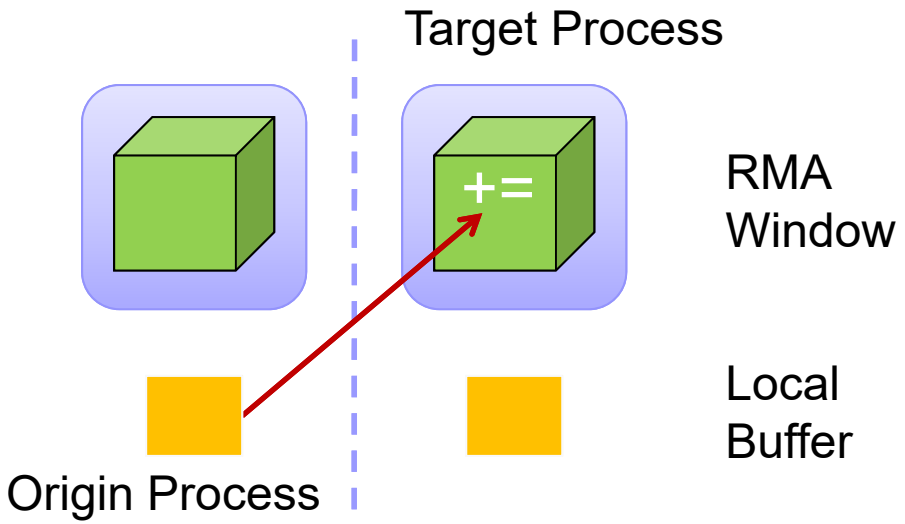
- Move data to origin, from target



Atomic Data Aggregation: *Accumulate*

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

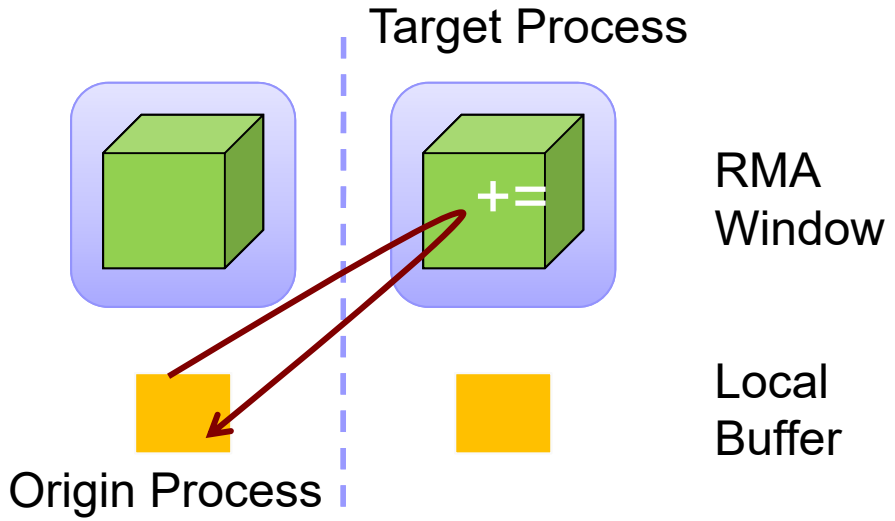
- Atomic update operation, similar to a put
  - Reduces origin and target data into target buffer using op argument as combiner
  - Predefined ops only, no user-defined operations
- Different data layouts between target/origin OK
  - Basic type elements must match
- Op = MPI_REPLACE
  - Implements *f(a,b)=b*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



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)
```

- 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
- Atomic get with MPI_NO_OP
- Atomic swap with MPI_REPLACE



Atomic Data Aggregation: *CAS and FOP*

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

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

- CAS: Atomic swap if target value is equal to compare value
- 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

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 (s)he does not require ordering as optimization hint
  - You can ask for only the needed orderings: RAW (read-after-write), WAR, RAR, or WAW

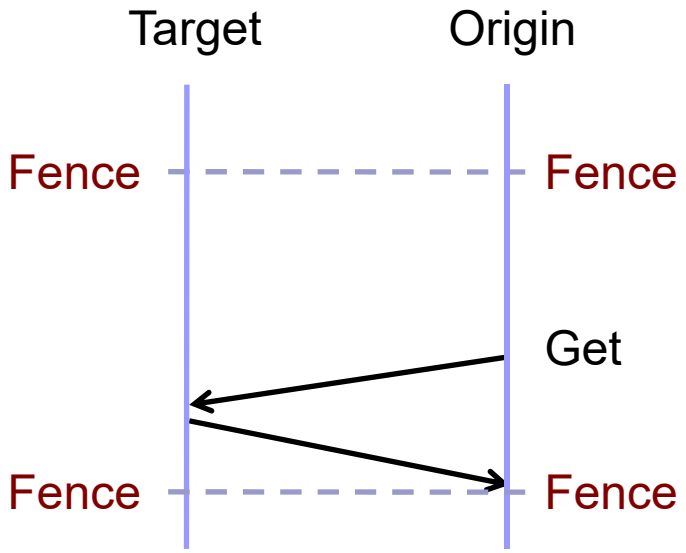
RMA Synchronization Models

- 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 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

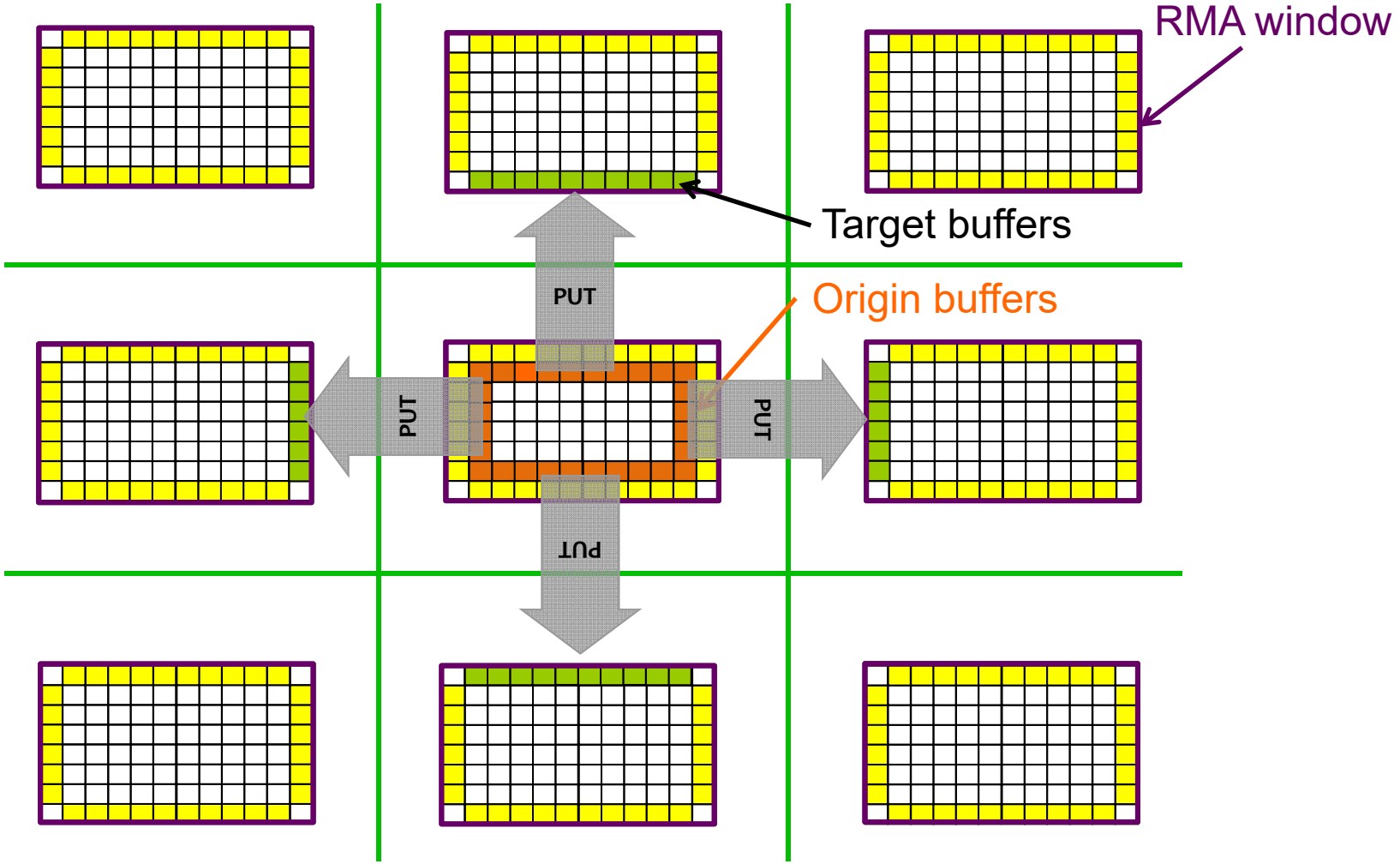
Fence: Active Target Synchronization

```
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



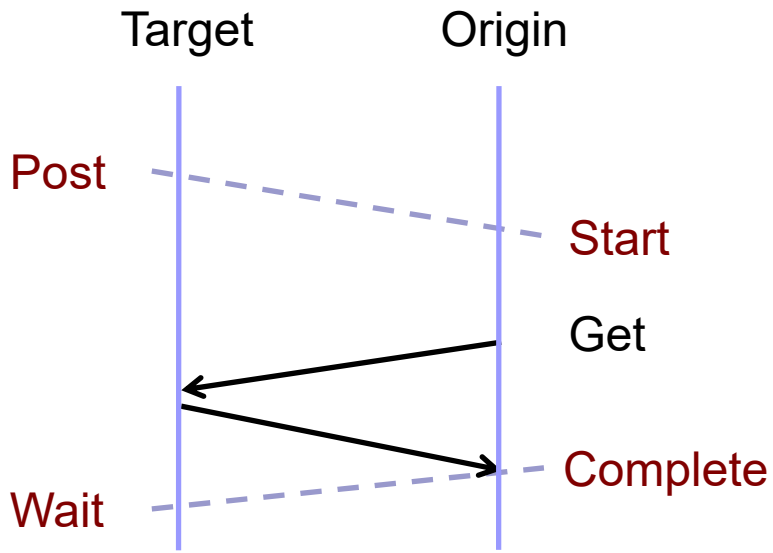
Stencil Computation by RMA Fence



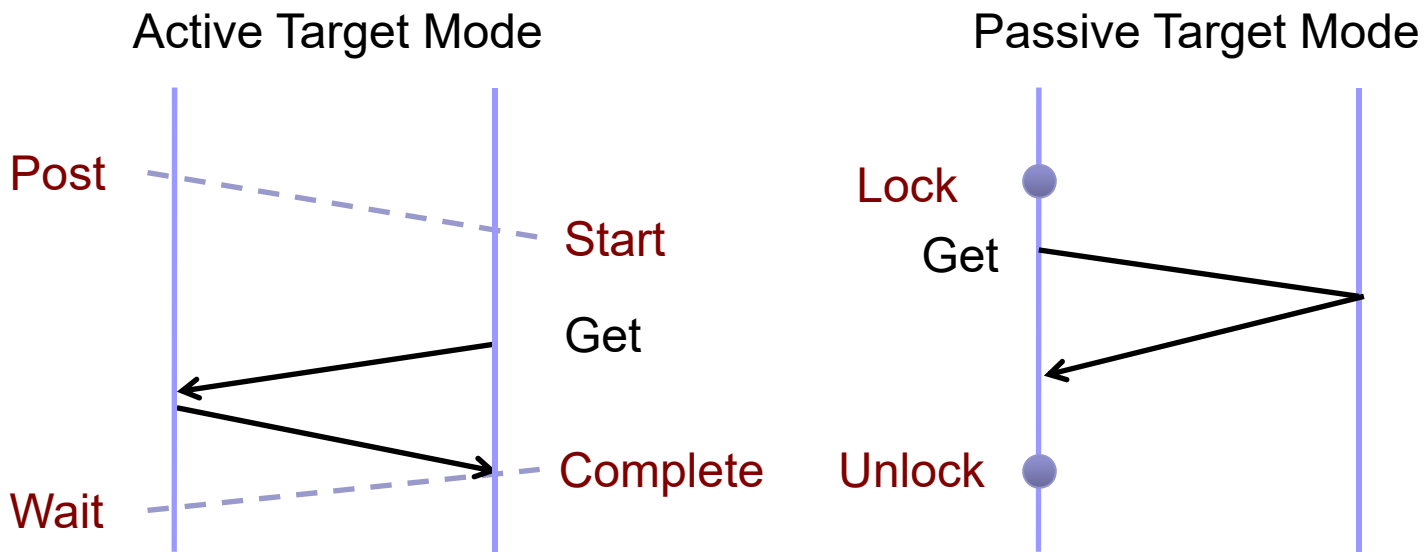
PSCW: Generalized Active Target Synchronization

```
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



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

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

MPI_Win_unlock(int rank, MPI_Win win)
```

- 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

Advanced Passive Target Synchronization

```
MPI_Win_lock_all(int assert, MPI_Win win)
MPI_Win_unlock_all(MPI_Win win)

MPI_Win_flush/flush_local(int rank, MPI_Win win)
MPI_Win_flush_all/flush_local_all(MPI_Win win)
```

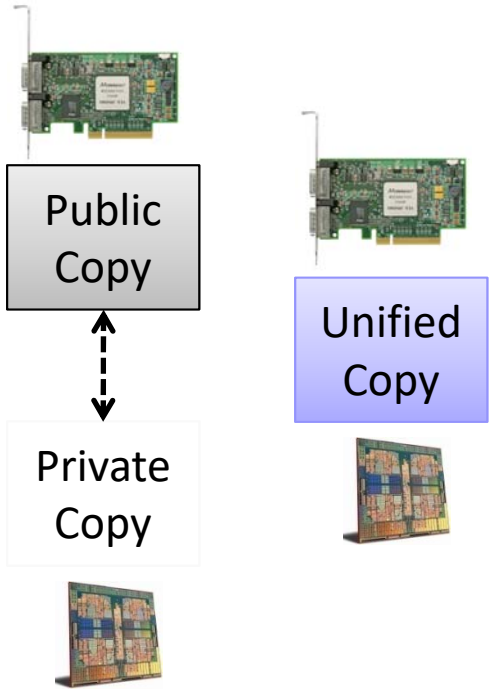
- Lock_all: Shared lock, passive target epoch to all other processes
  - Expected usage is long-lived: lock_all, put/get, flush, …, unlock_all
- Flush: Remotely complete RMA operations to the target process
  - Flush_all – remotely complete RMA operations to all processes
  - After completion, data can be read by target process or a different process
- Flush_local: Locally complete RMA operations to the target process
  - Flush_local_all – locally complete RMA operations to all processes

Which synchronization mode should I use, when?

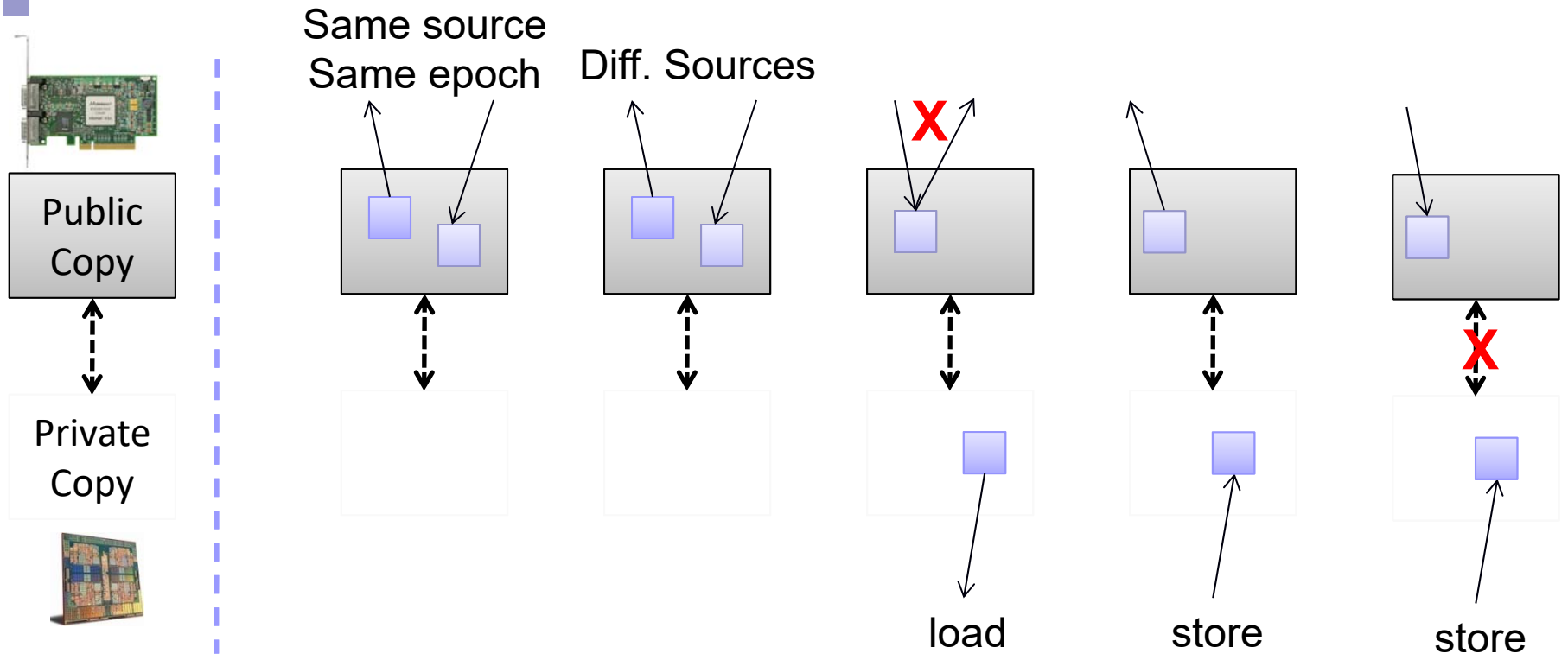
- RMA communication has low overheads versus send/recv
  - Two-sided: Matching, queuing, buffering, unexpected receives, etc…
  - One-sided: No matching, no buffering, always ready to receive
  - Utilize RDMA provided by high-speed interconnects (e.g. InfiniBand)
- Active mode: bulk synchronization
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

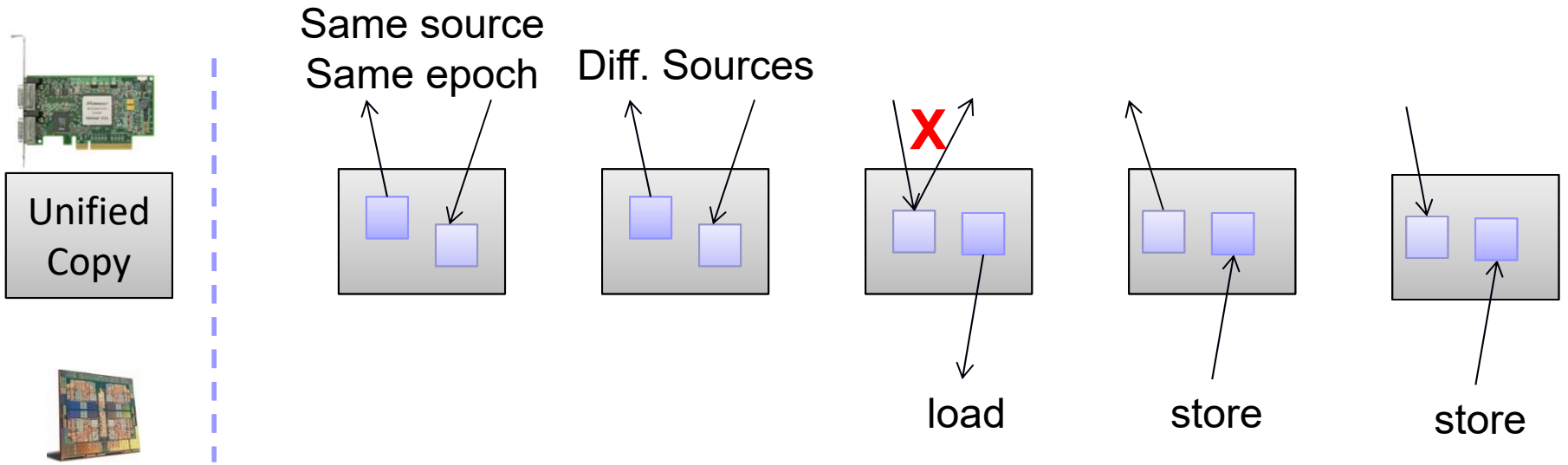


MPI RMA Memory Model (separate windows)



- 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)

	Load	Store	Get	Put	Acc
Load	OVL+NOVL	OVL+NOVL	OVL+NOVL	NOVL	NOVL
Store	OVL+NOVL	OVL+NOVL	NOVL	X	X
Get	OVL+NOVL	NOVL	OVL+NOVL	NOVL	NOVL
Put	NOVL	X	NOVL	NOVL	NOVL
Acc	NOVL	X	NOVL	NOVL	OVL+NOVL

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)

	Load	Store	Get	Put	Acc
Load	OVL+NOVL	OVL+NOVL	OVL+NOVL	NOVL	NOVL
Store	OVL+NOVL	OVL+NOVL	NOVL	NOVL	NOVL
Get	OVL+NOVL	NOVL	OVL+NOVL	NOVL	NOVL
Put	NOVL	NOVL	NOVL	NOVL	NOVL
Acc	NOVL	NOVL	NOVL	NOVL	OVL+NOVL

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

Content

- Topology Mapping
- Remote Memory Access
- Others:
  - Nonblocking Collective Communication
  - Hybrid Programming with Threads, GPUs
  - MPI I/O

Nonblocking Collective Communication

- Nonblocking (send/recv) communication
  - Deadlock avoidance
  - Overlapping communication/computation
- Collective communication
  - Collection of pre-defined optimized routines
- → Nonblocking collective communication
  - **Combines both techniques** (more than the sum of the parts ☺)
  - System noise/imbalance resiliency
  - Semantic advantages

Nonblocking Collective Communication

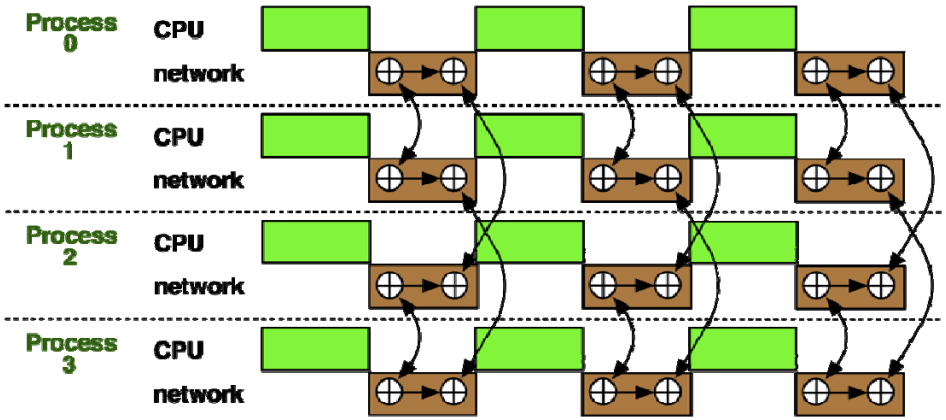
- Nonblocking variants of all collectives
  - MPI_Ibcast(<bcast args>, MPI_Request *req);
- Semantics
  - Function returns no matter what
  - No guaranteed progress (quality of implementation)
  - Usual completion calls (wait, test) + mixing
  - Out-of order completion
- Restrictions
  - No tags, in-order matching
  - Send and vector buffers may not be touched during operation
  - MPI_Cancel not supported
  - No matching with blocking collectives

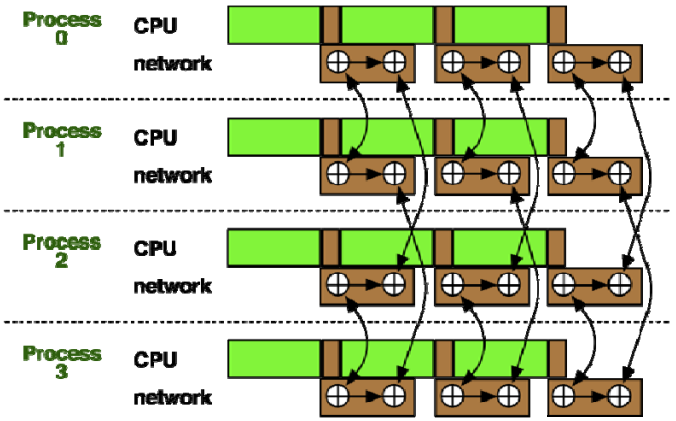
Nonblocking Collective Communication

- Semantic advantages
  - Enable asynchronous progression (and manual)
    - Software pipelining
  - Decouple data transfer and synchronization
    - Noise resiliency!
  - Allow overlapping communicators
    - See also neighborhood collectives
  - Multiple outstanding operations at any time
    - Enables pipelining window

Nonblocking Collectives Overlap

- Software pipelining
  - More complex parameters
  - Progression issues
  - Not scale-invariant





Content

- Topology Mapping
- Remote Memory Access
- Others:
  - Nonblocking Collective Communication
  - Hybrid Programming with Threads, GPUs
  - MPI I/O

MPI and Threads

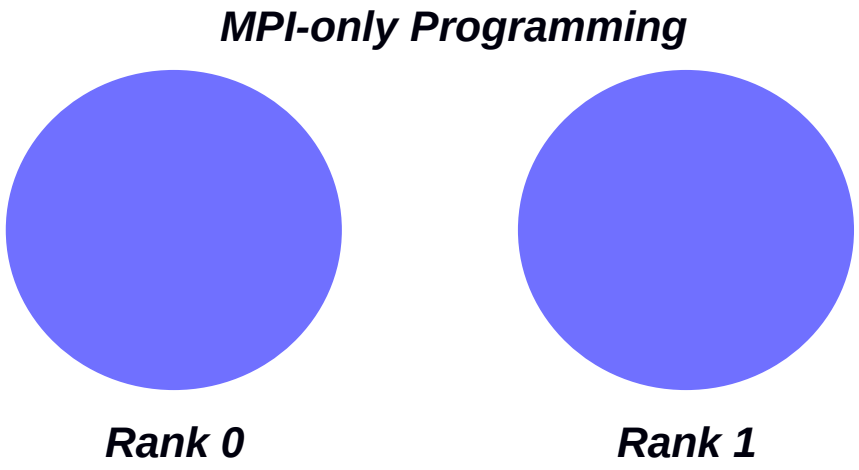
- MPI describes parallelism between *processes* (with separate address spaces)
- *Thread* parallelism provides a shared-memory model within a process

- **OpenMP and Pthreads** are common models
  - OpenMP provides convenient features for loop-level parallelism. Threads are created and managed by the compiler, based on user directives.
  - Pthreads provide more complex and dynamic approaches. Threads are created and managed explicitly by the user.

Programming for Multicore

- Common options for programming **multicore clusters**
  - All MPI
    - MPI between processes both within a node and across nodes
    - MPI internally uses shared memory to communicate within a node
  - MPI + OpenMP
    - Use OpenMP within a node and MPI across nodes
  - MPI + Pthreads
    - Use Pthreads within a node and MPI across nodes
- The latter two approaches are known as “hybrid programming”

Hybrid Programming: MPI+Threads



- In MPI-only programming, each MPI process has a single program counter
- In MPI+threads hybrid programming, there can be multiple threads executing simultaneously
  - All threads share all MPI objects (communicators, requests)
  - The MPI implementation might need to take precautions to make sure the state of the MPI stack is consistent

MPI's Four Levels of Thread Safety

- MPI defines four levels of thread safety -- these are commitments the application makes to the MPI
  - **MPI_THREAD_SINGLE**: only one thread exists in the application
  - **MPI_THREAD_FUNNELED**: multithreaded, but only the main thread makes MPI calls (the one that called MPI_Init_thread)
  - **MPI_THREAD_SERIALIZED**: multithreaded, but only one thread *at a time* makes MPI calls
  - **MPI_THREAD_MULTIPLE**: multithreaded and any thread can make MPI calls at any time (with some restrictions to avoid races – see next slide)
- Thread levels are in increasing order
  - If an application works in FUNNELED mode, it can work in SERIALIZED
- MPI defines an alternative to MPI_Init
  - MPI_Init_thread(requested, provided)
    - *Application gives level it needs; MPI implementation gives level it supports*

MPI_THREAD_SINGLE

- There are no threads in the system
  - E.g., there are no OpenMP parallel regions

```
int main(int argc, char ** argv)
{
    int buf[100];

    MPI_Init(&argc, &argv);

    for (i = 0; i < 100; i++)
        compute(buf[i]);

    /* Do MPI stuff */

    MPI_Finalize();

    return 0;
}
```

MPI_THREAD_FUNNELED

- All MPI calls are made by the master thread
  - Outside the OpenMP parallel regions, In OpenMP master regions

```
int main(int argc, char ** argv)
{
    int buf[100], provided;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
    if (provided < MPI_THREAD_FUNNELED)
        MPI_Abort(MPI_COMM_WORLD, 1);

#pragma omp parallel for
    for (i = 0; i < 100; i++)
        compute(buf[i]);

    /* Do MPI stuff */
    MPI_Finalize();
    return 0;
}
```

MPI_THREAD_SERIALIZED

- Only one thread can make MPI calls at a time
  - Protected by OpenMP critical regions

```
int main(int argc, char ** argv)
{
    int buf[100], provided;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &provided);
    if (provided < MPI_THREAD_SERIALIZED)
        MPI_Abort(MPI_COMM_WORLD, 1);

#pragma omp parallel for
    for (i = 0; i < 100; i++) {
        compute(buf[i]);
#pragma omp critical
        /* Do MPI stuff */
    }

    MPI_Finalize();
    return 0;
}
```

MPI_THREAD_MULTIPLE

- Any thread can make MPI calls any time (restrictions apply)

```
int main(int argc, char ** argv)
{
    int buf[100], provided;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
    if (provided < MPI_THREAD_MULTIPLE)
        MPI_Abort(MPI_COMM_WORLD, 1);

#pragma omp parallel for
    for (i = 0; i < 100; i++) {
        compute(buf[i]);
        /* Do MPI stuff */
    }

    MPI_Finalize();

    return 0;
}
```

Threads and MPI

- An implementation **is not required** to support levels higher than MPI_THREAD_SINGLE; that is:
  - An implementation is not required to be **thread safe**
  - A **fully thread-safe** implementation will support MPI_THREAD_MULTIPLE
- A program that calls MPI_Init (instead of MPI_Init_thread) should assume that only MPI_THREAD_SINGLE is supported
- *A threaded MPI program that does not call MPI_Init_thread is an incorrect program (common user error we see)*

Specification of MPI_THREAD_MULTIPLE

- ***Ordering:*** When multiple threads make MPI calls concurrently, the outcome will be as if the calls executed sequentially in some (any) order
  - Ordering is maintained within each thread
  - User must ensure that collective operations on the same communicator, window, or file handle are correctly ordered among threads
    - E.g., cannot call a broadcast on one thread and a reduce on another thread on the same communicator
  - It is the user's responsibility to prevent races when threads in the same application post conflicting MPI calls
    - E.g., accessing an info object from one thread and freeing it from another thread

Specification of MPI_THREAD_MULTIPLE

- ***Blocking:*** Blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions

The Current Situation

- All MPI implementations support MPI_THREAD_SINGLE (duh).
- They probably support MPI_THREAD_FUNNELED even if they don't admit it.
  - Does require thread-safe malloc
  - Probably OK in OpenMP programs

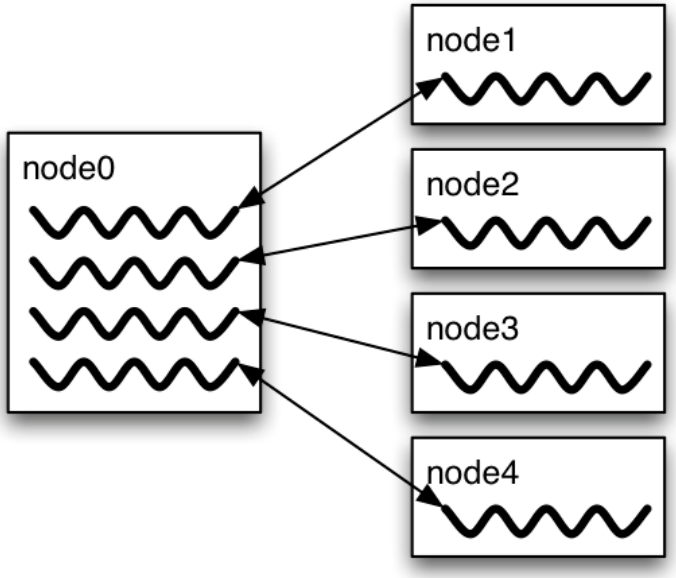
The Current Situation

- Many (but not all) implementations support THREAD_MULTIPLE
  - Hard to implement efficiently though (lock granularity issue)
- "Easy" OpenMP programs (loops parallelized with OpenMP, communication in between loops) only need FUNNELED
  - So don't need "thread-safe" MPI for many hybrid programs
  - But watch out for Amdahl's Law!

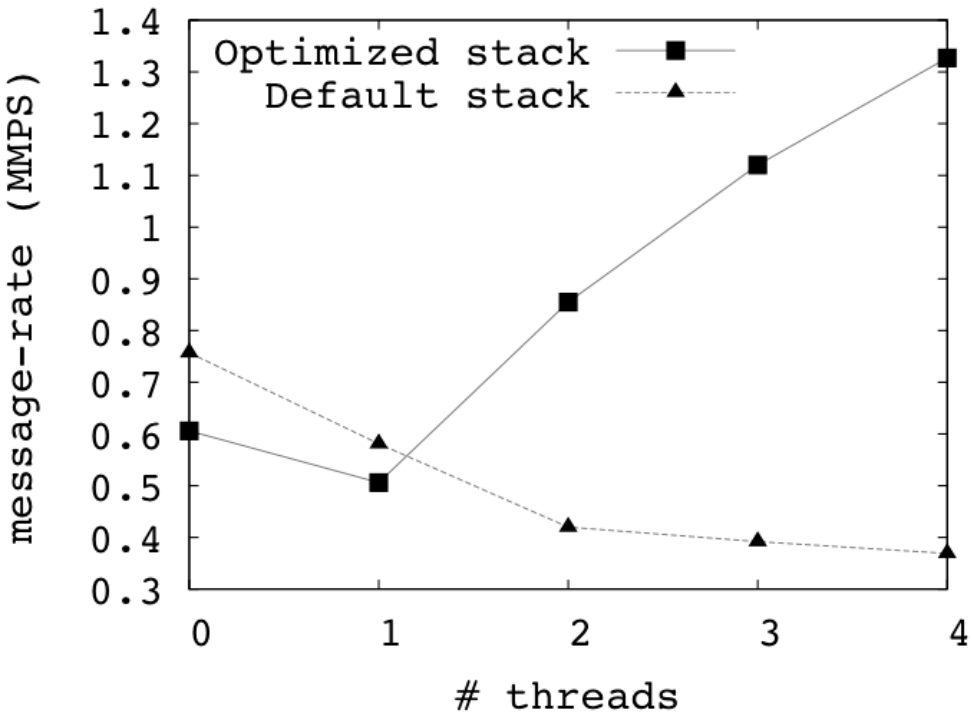
Performance with MPI_THREAD_MULTIPLE

- Thread safety does not come for free
- The implementation must protect certain data structures or parts of code with mutexes or critical sections
- To measure the performance impact, we ran tests to measure communication performance when using multiple threads versus multiple processes
  - For results, see Thakur/Gropp paper: "Test Suite for Evaluating Performance of Multithreaded MPI Communication," *Parallel Computing*, 2009

Message Rate Results on BG/P



Message Rate Benchmark



"Enabling Concurrent Multithreaded MPI Communication on Multicore Petascale Systems"
EuroMPI 2010

Why hard to optimize PI_THREAD_MULTIPLE

- MPI internally maintains several resources
- Because of MPI semantics, it is required that all threads have access to some of the data structures
  - E.g., thread 1 can post an Irecv, and thread 2 can wait for its completion – thus the request queue has to be shared between both threads
  - Since multiple threads are accessing this shared queue, it needs to be locked – adds a lot of overhead

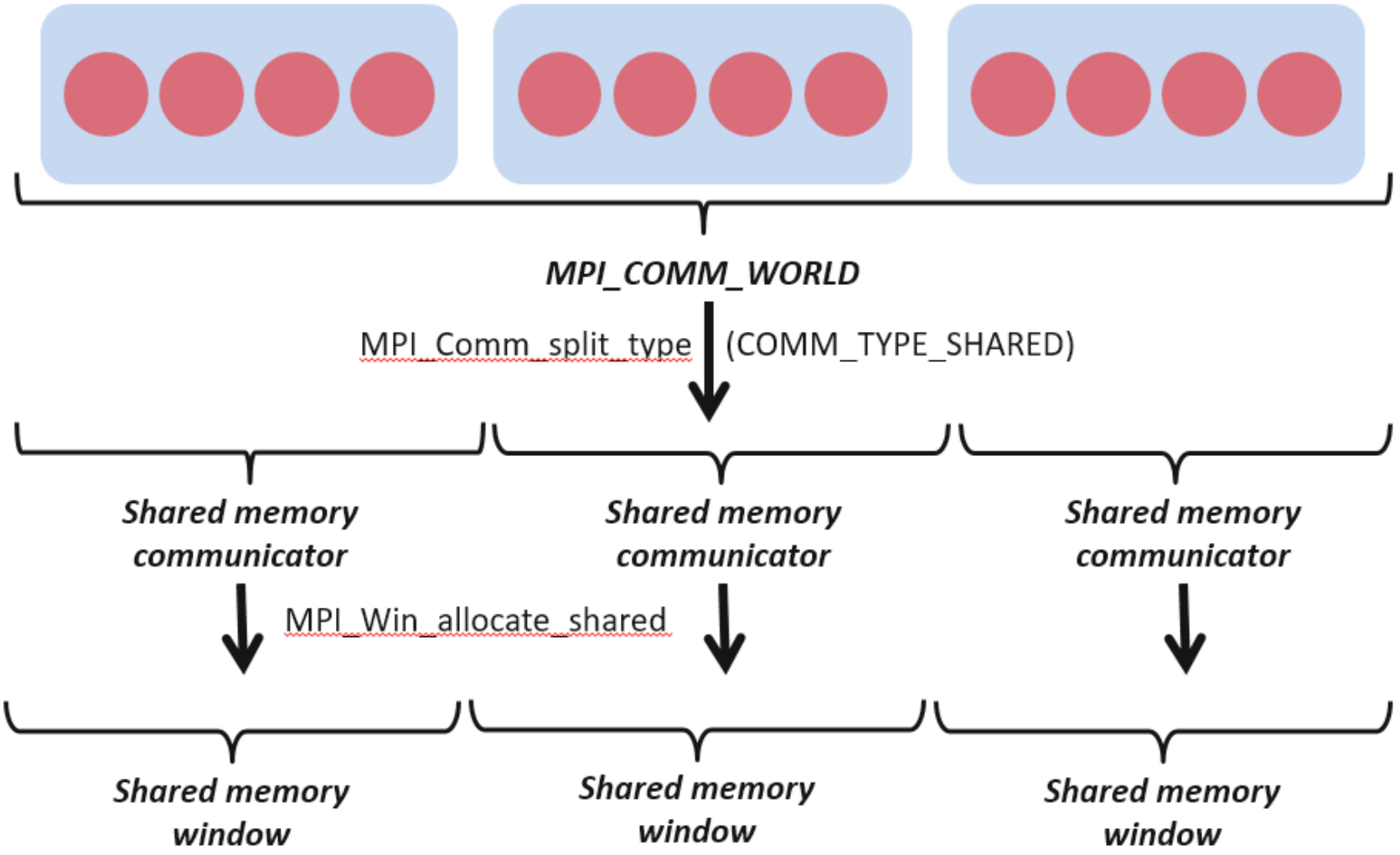
Hybrid Programming: Correctness Requirements

- Hybrid programming with MPI+threads does not do much to reduce the complexity of thread programming
  - Your application still has to be a correct multi-threaded application
  - On top of that, you also need to make sure you are correctly following MPI semantics
- Many commercial debuggers offer support for debugging hybrid MPI+threads applications (mostly for MPI+Pthreads and MPI+OpenMP)

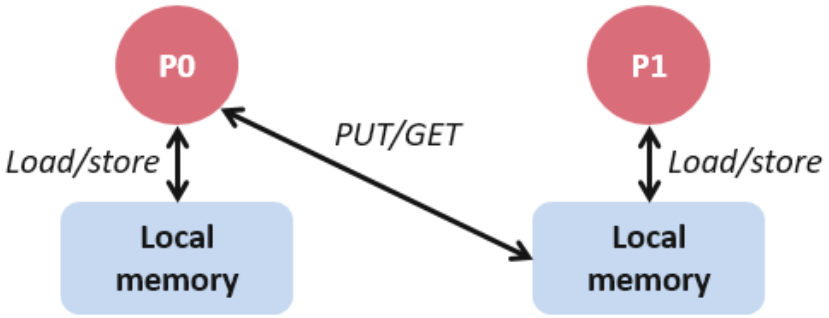
Hybrid Programming with Shared Memory

- MPI-3 allows different processes to allocate shared memory through MPI
  - MPI_Win_allocate_shared
- Uses many of the concepts of one-sided communication
- Applications can do hybrid programming using MPI or load/store accesses on the shared memory window
- Other MPI functions can be used to synchronize access to shared memory regions
- Can be simpler to program than threads

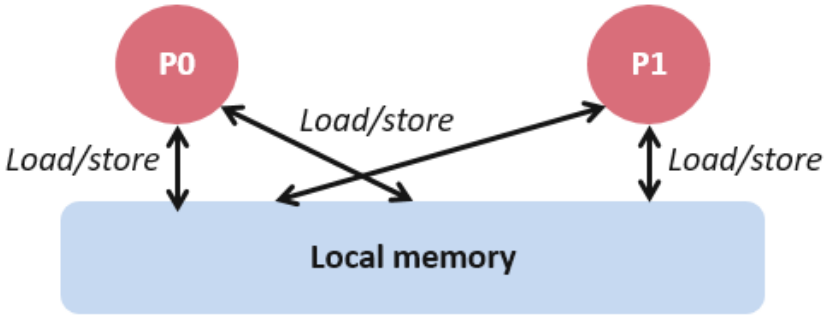
Creating Shared Memory Regions in MPI



Regular RMA windows vs. Shared memory windows



Traditional RMA windows



Shared memory windows

- Shared memory windows allow application processes to directly perform load/store accesses on all of the window memory
  - E.g., x[100] = 10
- All of the existing RMA functions can also be used on such memory for more advanced semantics such as atomic operations
- Can be very useful when processes want to use threads only to get access to all of the memory on the node
  - You can create a shared memory window and put your shared data

Memory allocation and placement

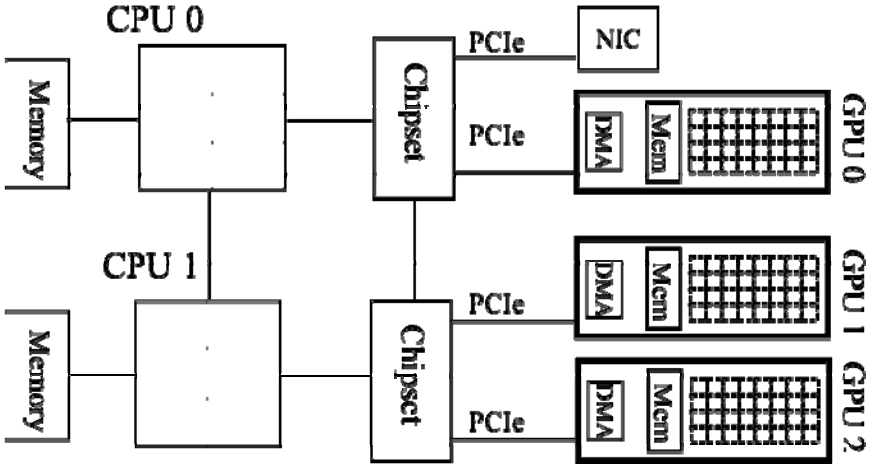
- Shared memory allocation does not need to be uniform across processes
  - Processes can allocate a different amount of memory (even zero)
- The MPI standard does not specify where the memory would be placed
  - Implementations can choose their own strategies, though it is expected that an implementation will try to place shared memory allocated by a process "close to it"
- The total allocated shared memory on a communicator is contiguous by default
  - Users can pass an info hint called "noncontig" that will allow the MPI implementation to align memory allocations from each process to appropriate boundaries to assist with placement

Shared Arrays with Shared memory windows

```
int main(int argc, char ** argv)
{
    int buf[100];

    MPI_Init(&argc, &argv);
    MPI_Comm_split_type(..., MPI_COMM_TYPE_SHARED, .., &comm);
    MPI_Win_allocate_shared(comm, ..., &win);

    MPI_Win_lockall(win);

    /* copy data to local part of shared memory */
    MPI_Win_sync(win);

    /* use shared memory */

    MPI_Win_unlock_all(win);

    MPI_Win_free(&win);
    MPI_Finalize();
    return 0;
}
```

Accelerators in Parallel Computing

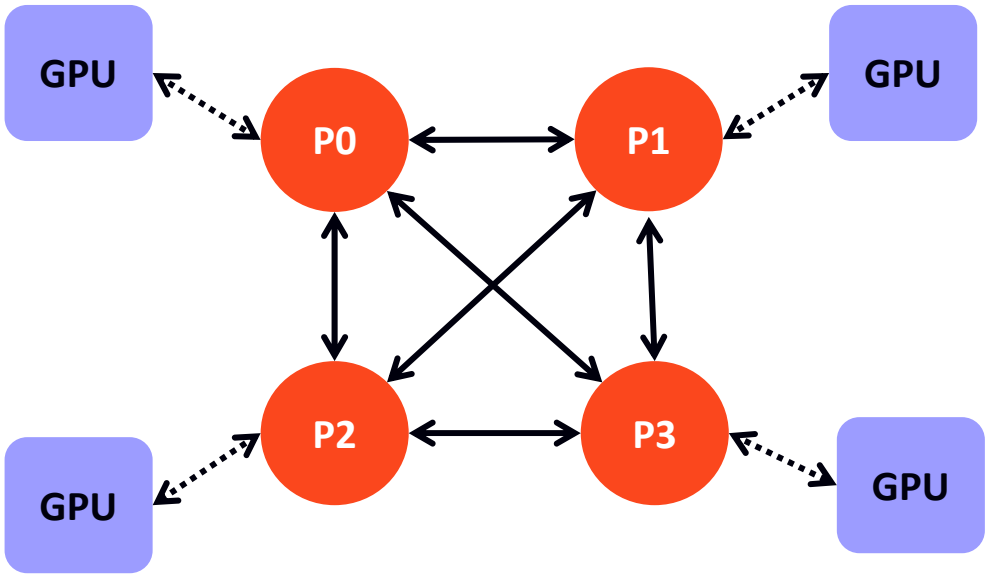
- General purpose, highly parallel processors
  - High FLOPs/Watt and FLOPs/$
  - Unit of execution *Kernel*
  - Separate memory subsystem
  - Prog. Models: CUDA, OpenCL, …
- Clusters with accelerators are becoming common
- New programmability and performance challenges for programming models and runtime systems



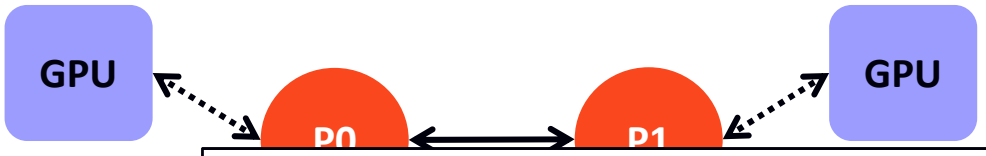
Hybrid Programming with Accelerators

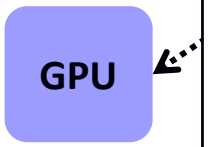
- Many users are looking to use accelerators within their MPI applications
- The MPI standard **does not provide** any special semantics to interact with accelerators
  - Current MPI threading semantics are considered sufficient by most users.
  - There are some research efforts for making accelerator memory directly accessibly by MPI, but those are not a part of the MPI standard.

Current Model for MPI+Accelerator Applications



Current Model for MPI+Accelerator Applications






```
double *dev_buf, *host_buf;
cudaMalloc(&dev_buf, size);
cudaMallocHost(&host_buf, size);

if (my_rank == sender) { /* sender */
  computation_on_GPU(dev_buf);
  cudaMemcpy(host_buf, dev_buf, size, ...);
  MPI_Send(host_buf, size, ...);
} else {                    /* receiver */
  MPI_Recv(host_buf, size, ...);
  cudaMemcpy(dev_buf, host_buf, size, ...);
  computation_on_GPU(dev_buf);
}
```

Alternate MPI+Accelerator models being studied

- Some MPI implementations (MPICH, Open MPI, MVAPICH) are investigating **how the MPI implementation can directly send/receive data from accelerators**
  - Unified virtual address (UVA) space techniques where all memory (including accelerator memory) is represented with a "void *"
  - Communicator and datatype attribute models where users can inform the MPI implementation of where the data resides

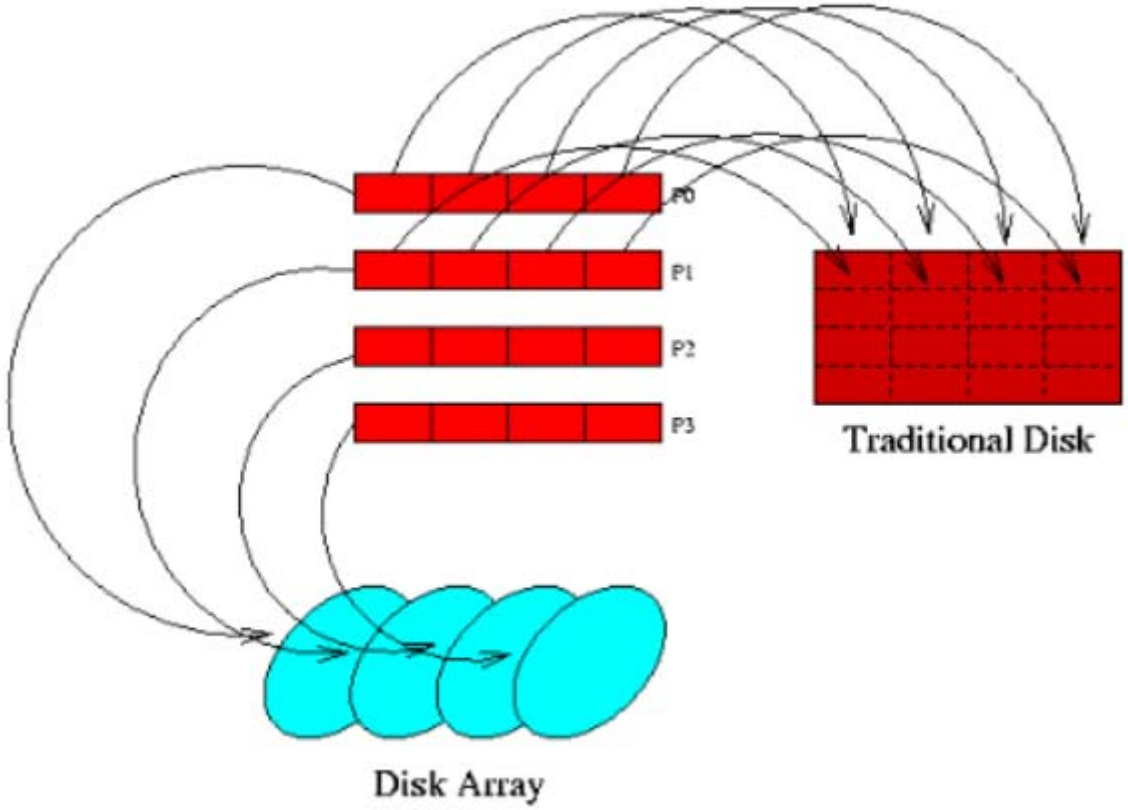
- Clear performance advantages demonstrated in research papers, but these features are not yet a part of the MPI standard (as of MPI-3)
  - Could be incorporated in a future version of the standard

Content

- Topology Mapping
- Remote Memory Access
- Others:
  - Nonblocking Collective Communication
  - Hybrid Programming with Threads, GPUs
  - MPI I/O

What is parallel I/O

- Multiple processes accessing a single file



What is parallel I/O

- Multiple processes accessing a single file
- Often, both data and file access is non-contiguous
  - Ghost cells cause non-contiguous data access
  - Block or cyclic distributions cause non-contiguous file access
- Want to access data and files with as few I/O calls as possible

Why use parallel I/O

- Many users do not have time to learn the complexities of I/O optimization
- Use of parallel I/O can simplify coding
  - Single read/write operation vs. multiple read/write operations
- Parallel I/O potentially offers significant performance improvement over traditional approaches

Why use parallel I/O

- Traditional approaches
  - Each process writes to a separate file
    - Often requires an additional post-processing step
    - Without post-processing, restarts must use same number of processor
  - Result sent to a master processor, which collects results and writes out to disk
  - Each processor calculates position in file and writes individually

Why use parallel I/O

- MPI I/O approaches
  - MPI-I/O is a set of extensions to the original MPI standard
  - This is an interface specification: It does NOT give implementation specifics
  - It provides routines for file manipulation and data access
  - Calls to MPI-I/O routines are portable across a large number of architectures

Terms and Definitions

- **Displacement** - Number of bytes from the beginning of a file
- **etype** - unit of data access within a file
- **filetype** - datatype used to express access patterns of a file
- **file view** - definition of access patterns of a file
  - Defines what parts of a file are visible to a process

Terms and Definitions

- **Offset** - Position in the file, relative to the current view, expressed in terms of number of etypes
- **file pointers** - offsets into the file maintained by MPI
  - Individual file pointer - local to the process that opened the file
  - Shared file pointer - shared (and manipulated) by the group of processes that opened the file

FILE MANIPULATION

```
MPI_FILE_OPEN( MPI_Comm comm, char  *filename, int mode,
               MPI_Info info, MPI_File  *fh,  ierr )
```

- Opens the file identified by filename on each processor in communicator Comm
- Collective over this group of processors
- Each processor must use same value for mode and reference the same file
- info is used to give hints about access patterns

DERIVED DATATYPES & VIEWS

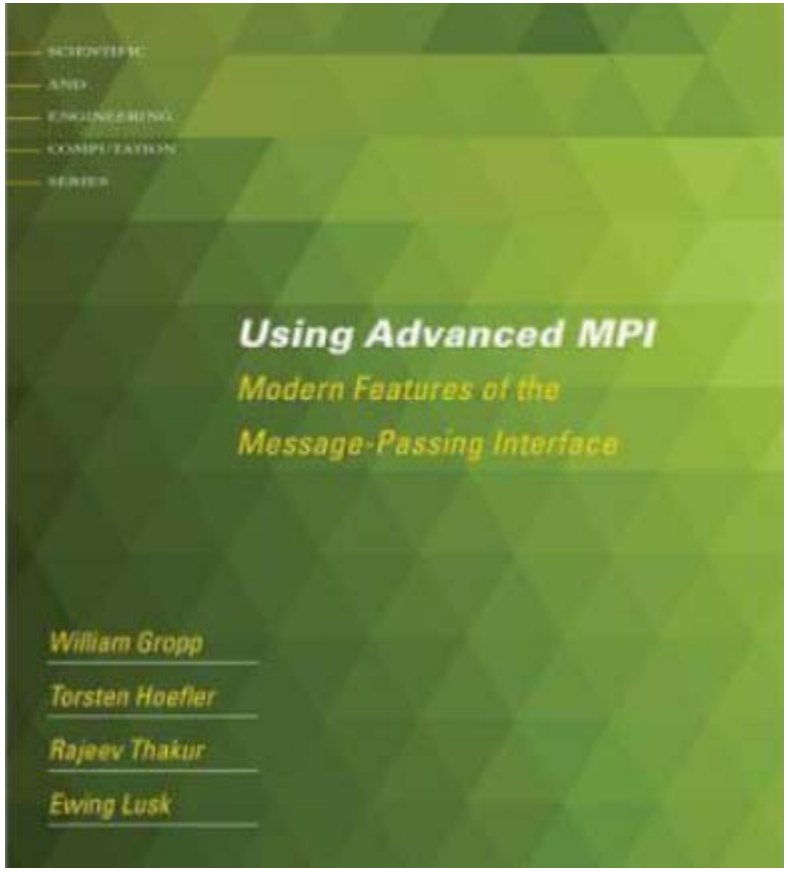
- Derived datatypes are not part of MPI-I/O
- They are used extensively in conjunction with MPI-I/O
- A filetype is really a datatype expressing the access pattern of a file
- Filetypes are used to set file views

CONCLUSIONS

- MPI-I/O potentially offers significant improvement in I/O performance
- This improvement can be attained with minimal effort on part of the user
  - Simpler programming with fewer calls to I/O routines
  - Easier program maintenance due to simple API

Reference Book

- *For more advanced features*
- *Using Advanced MPI*
  - *RMA*
  - *Parallel I/O*
  - *Dynamic process management*
  - *…*



http://www.mcs.anl.gov/research/projects/mpi/usingmpi/

references

- Parallel Programming with MPI, Argonne National Laboratory, http://www.anl.gov/events/parallel-programming-mpi.
- A Comprehensive MPI Tutorial Resource, https://github.com/wesleykendall/mpitutorial.
- 迟学斌等，并行计算与实现技术，科学出版社。