www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

MPI: One-Sided Communication

Marc Jordà, Antonio J. Peña

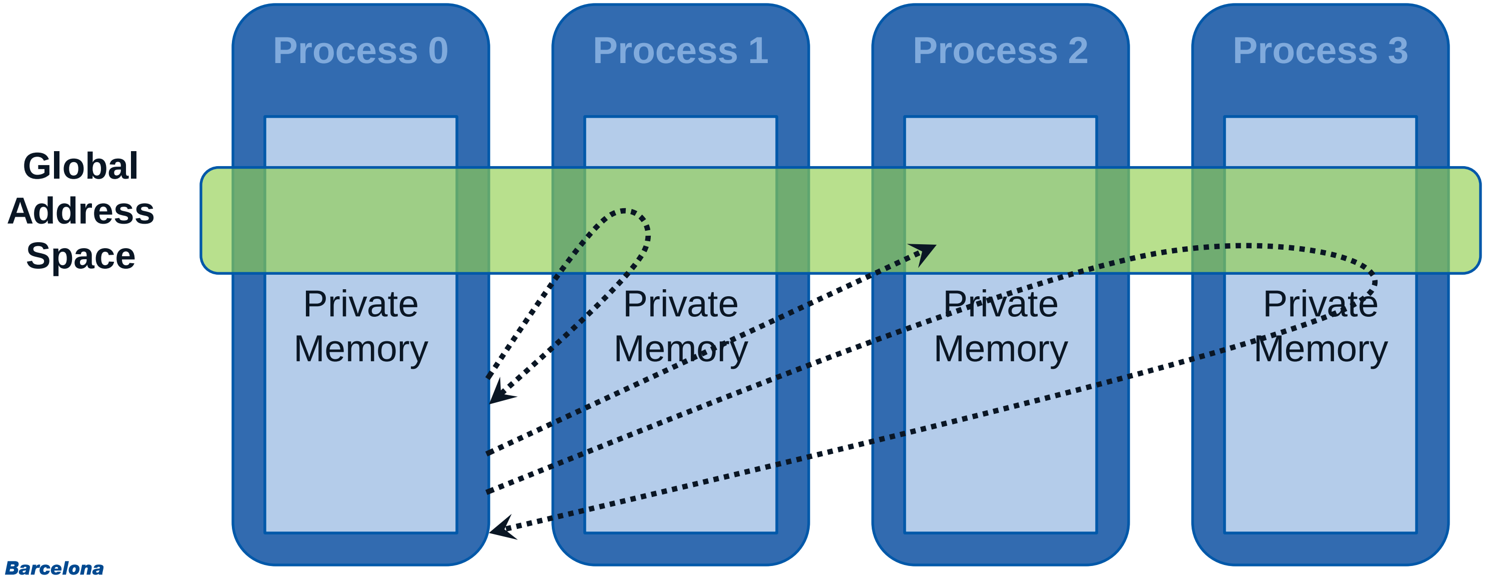
Montevideo, 21-25 October 2019

What will be covered in this tutorial

- What is MPI?
- How to write a simple program in MPI
- Running your application with MPICH
- **More advanced topics:**
  - Non-blocking communication, collective communication, datatypes
  - **One-sided communication**
  - Hybrid programming with shared memory and accelerators
  - Non-blocking collectives, topologies, and neighborhood collectives

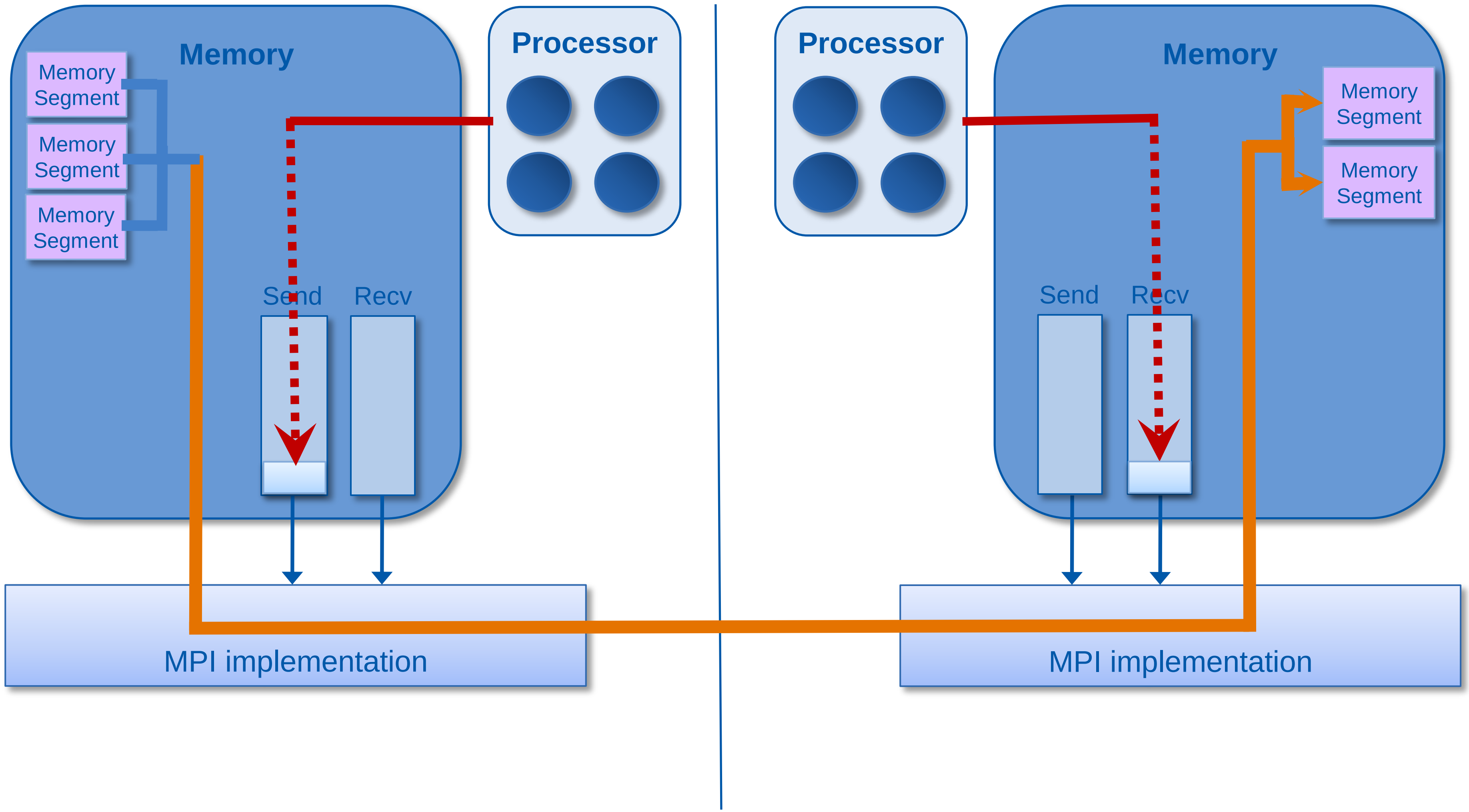
One-sided Communication

- 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

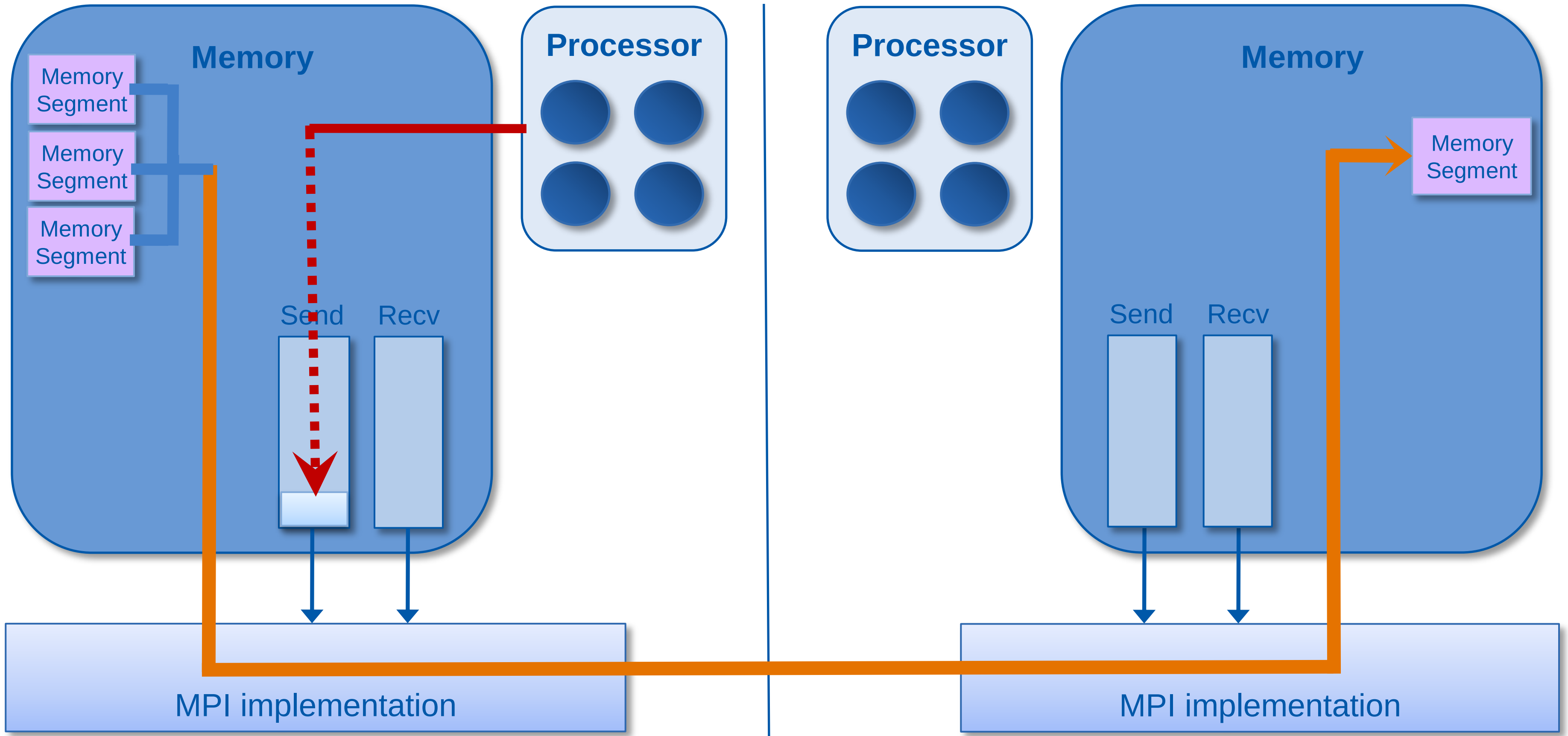


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Two-sided Communication Example

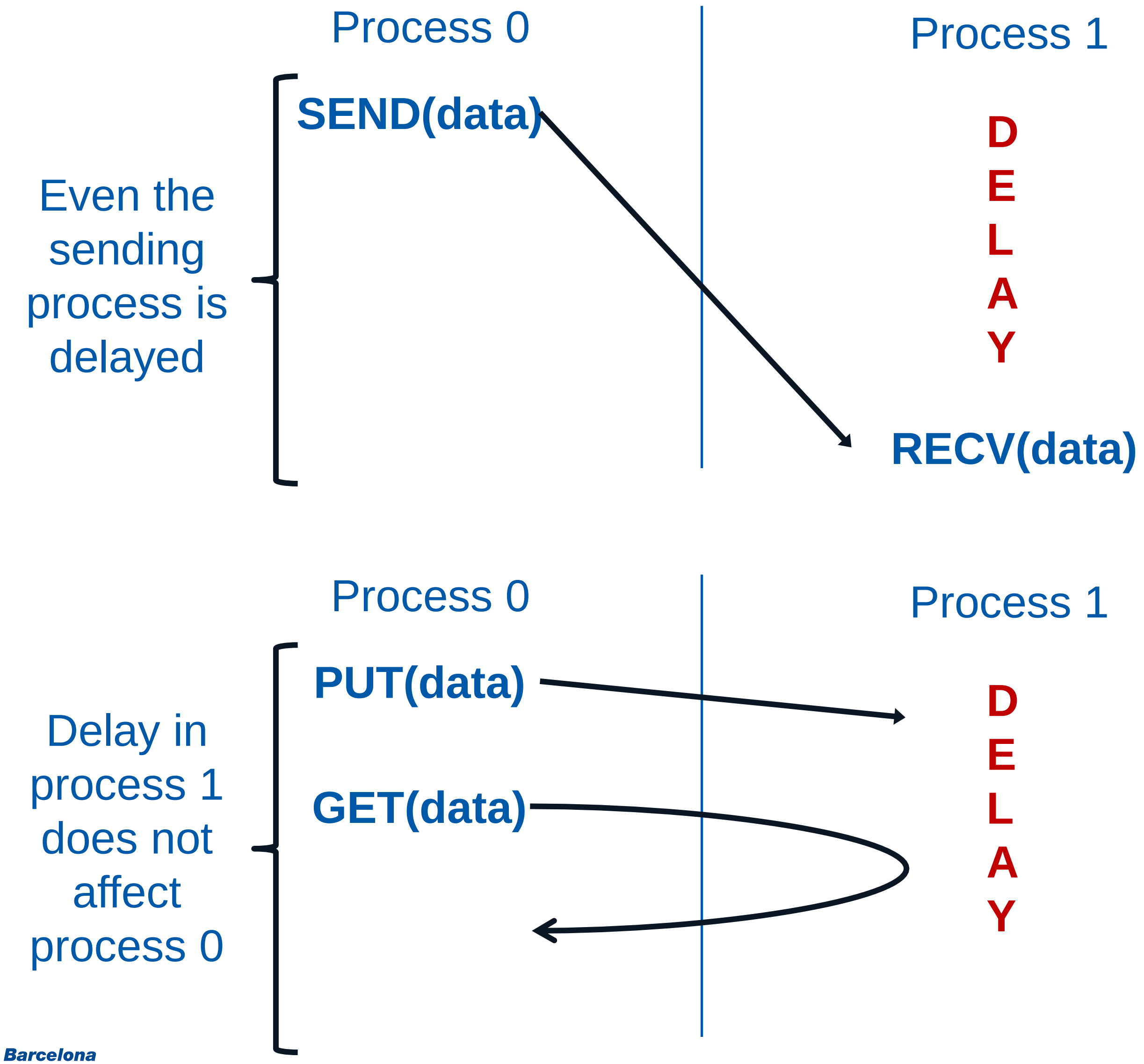


One-sided Communication Example



- May take advantage of interconnect RDMA, shared memory (intranode)
- Otherwise can be emulated

Comparing One-sided and Two-sided Programming

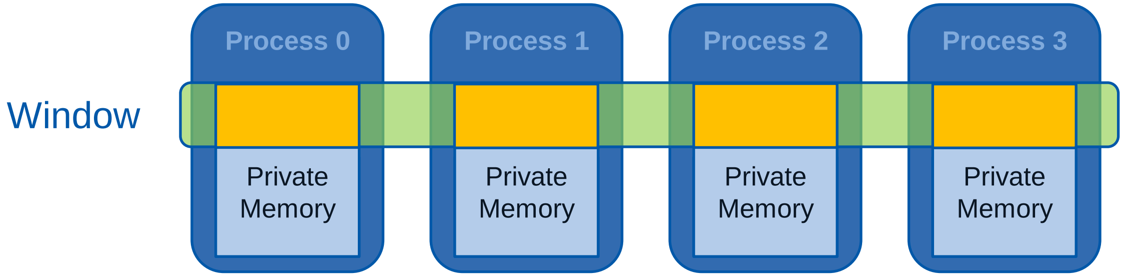


What we need to know in MPI RMA

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

Creating Public Memory

- Memory used by proceses 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
- 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

- MPI_Win_allocate
  - You want to create a buffer and directly make it remotely accessible
- MPI_Win_create
  - You already have an allocated buffer that you would like to make remotely accessible
- MPI_Win_create_dynamic
  - You don’t have a buffer yet, but will have one in the future
  - You can add/remove buffers with *MPI_Win_attach/detach*
- MPI_Win_allocate_shared
  - You want multiple processes on the same node to share a buffer with remote load/store access

MPI_WIN_ALLOCATE

```
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 <u>in bytes</u> (nonnegative integer)
  - disp_unit - local unit size for displacements, <u>in bytes</u> (positive integer)
  - info - flags passed to the MPI runtime (may enable optimization)
  - 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

```
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 <u>in bytes</u> (nonnegative integer)
  - disp_unit - local unit size for displacements, <u>in bytes</u> (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_CREATE_DYNAMIC

```
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;}
```

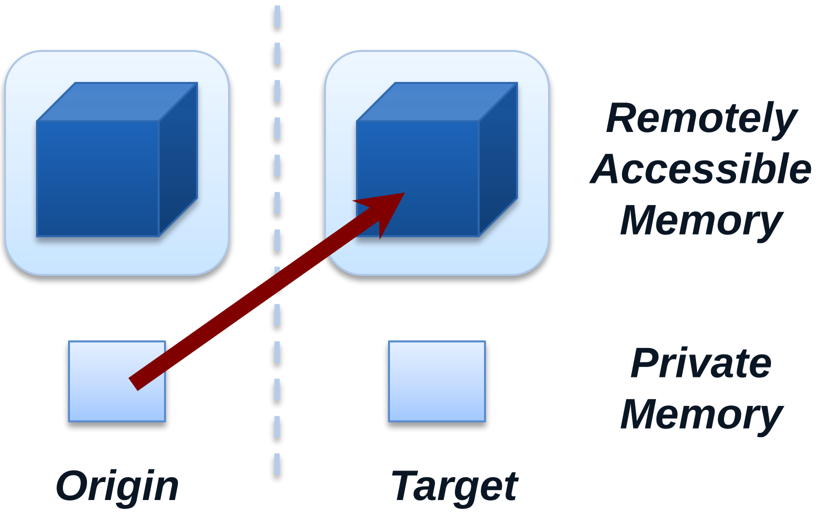
Data movement

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

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)
```

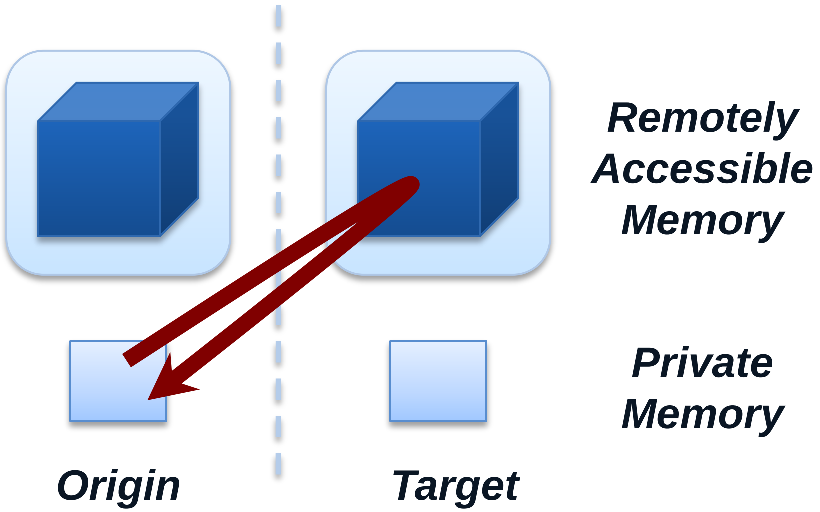
- Move data <u>from</u> origin, <u>to</u> target
- Separate data description triples for **origin** and **target**



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
- Separate data description triples for **origin** and **target**

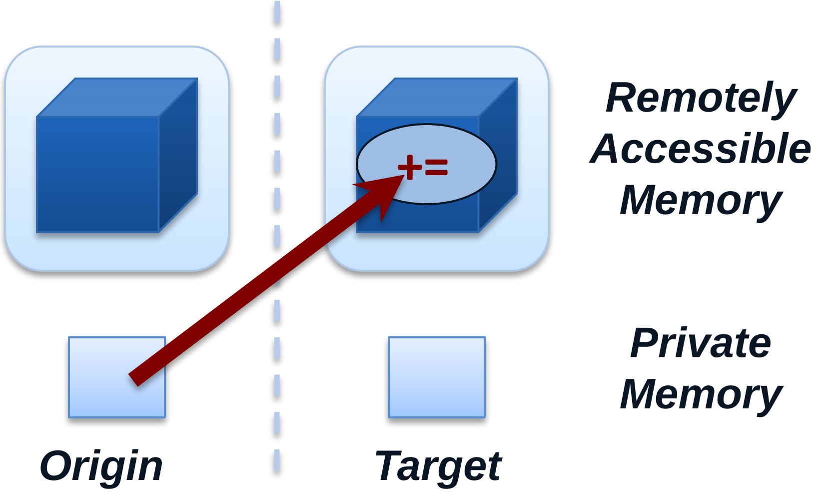


Barcelona Supercomputing Center
Centro Nacional de Supercomputación

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)
```

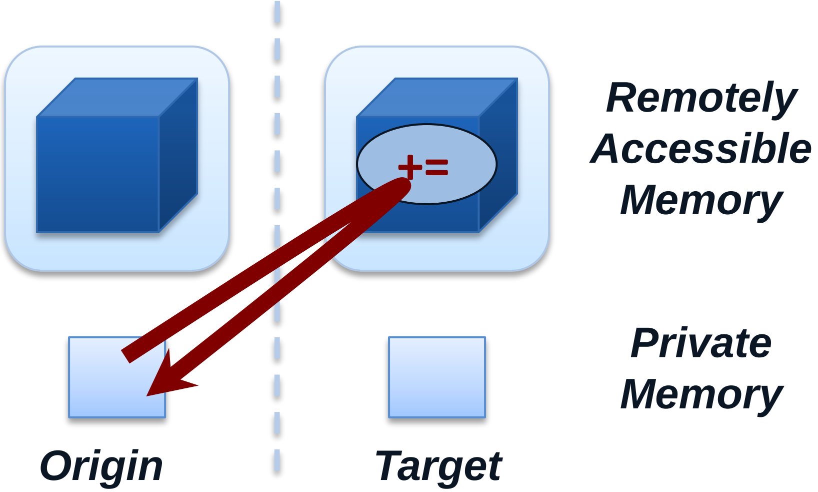
- Atomic update operation, similar to a put
  - Reduces origin and target data into target buffer using *op* argument as combiner
  - Op = MPI_SUM, MPI_PROD, MPI_OR, MPI_REPLACE, MPI_NO_OP, …
  - Predefined ops only, no user-defined operations
- Different data layouts between target/origin OK
  - Basic 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_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 = compare value

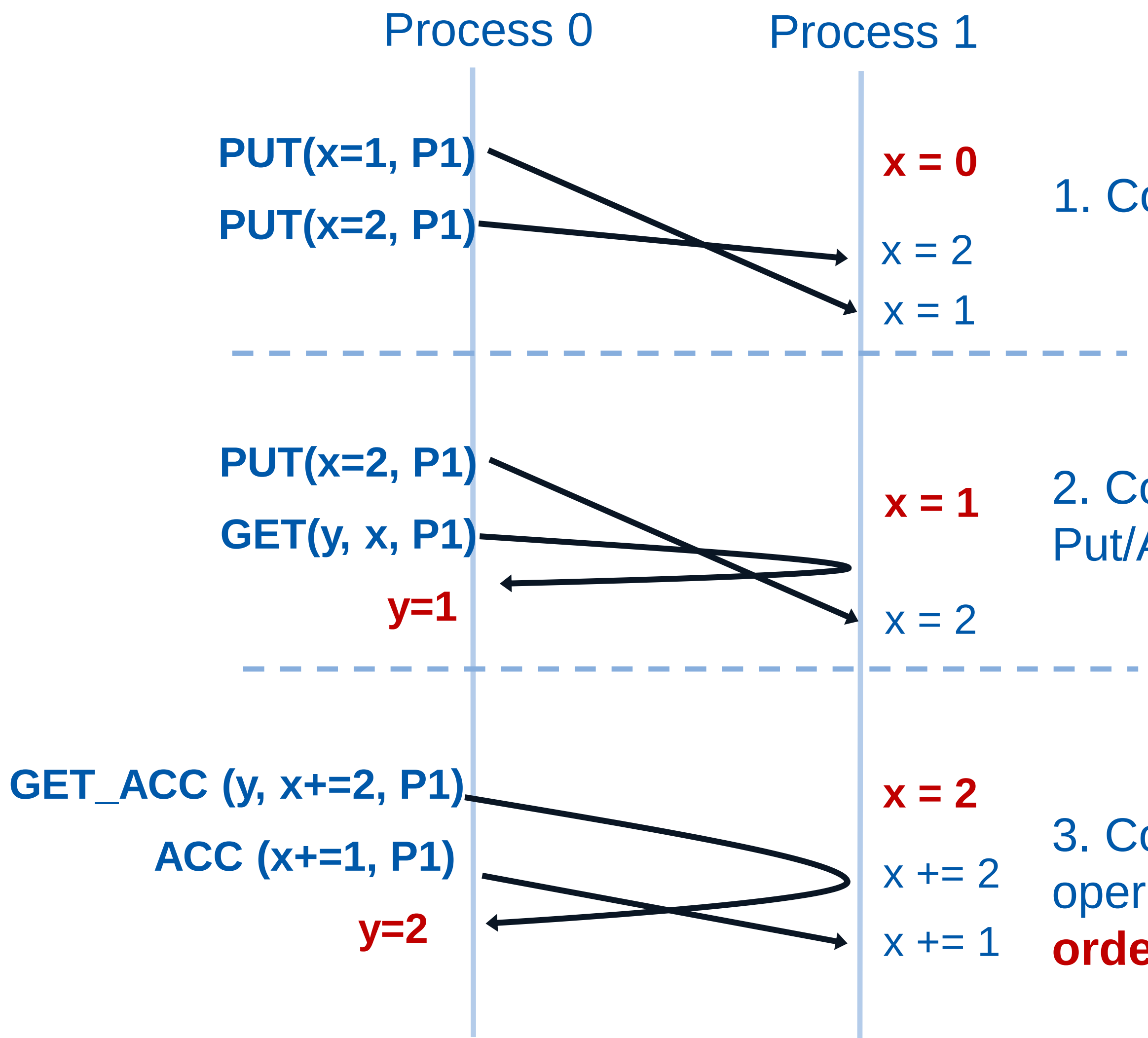
Ordering of Operations in MPI RMA

- No guaranteed ordering for Put/Get operations
- Result of concurrent Puts to the same location **undefined**
- Result of Get concurrent Put/Accumulate to same location **undefined**
  - Can be garbage in both cases

- Result of concurrent accumulate operations to the same location are **defined according to the order in which occurred**
  - Atomic put: Accumulate with op = MPI_REPLACE
  - Atomic get: Get_accumulate with op = MPI_NO_OP

- Accumulate operations from a process are ordered by default
  - User can tell the MPI implementation that ordering in not required as an optimization hint
  - You can ask for only the needed orderings: RAW (read-after-write), WAR, RAR, or WAW

Examples with operation ordering



1. Concurrent Puts: **undefined**

2. Concurrent Get and Put/Accumulates: **undefined**

3. Concurrent Accumulate operations to the same location: **ordering is guaranteed**

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 available for process Y to read?
  - RMA synchronization models define these semantics

- **Three synchronization models** provided by MPI:
  - Fence (active target, target process is involved in synchronization)
  - Post-start-complete-wait (generalized active target)
  - Lock/Unlock (passive target, target process not involved)

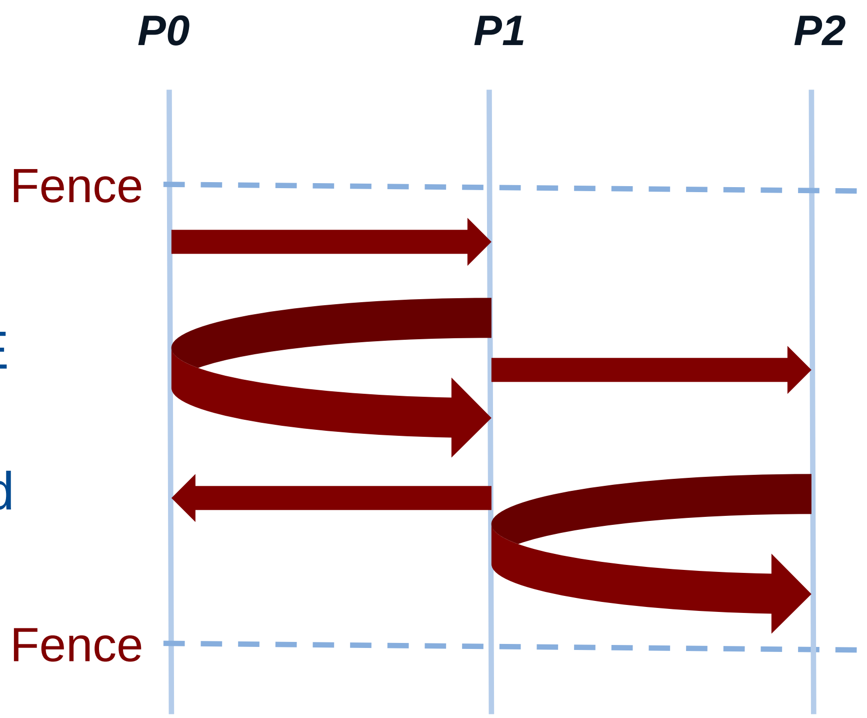
- Data accesses (Get, Put, Accum.) occur within ***epochs***
  - ***Access epochs***: a process can use get/put/accum on remote data
  - ***Exposure epochs***: a process exposes its mem segment in *win* to other processes
  - Epochs define ordering and completion semantics
  - Synchronization models provide mechanisms define the epochs
    - E.g., starting, ending, and synchronizing epochs

Fence: Active Target Synchronization

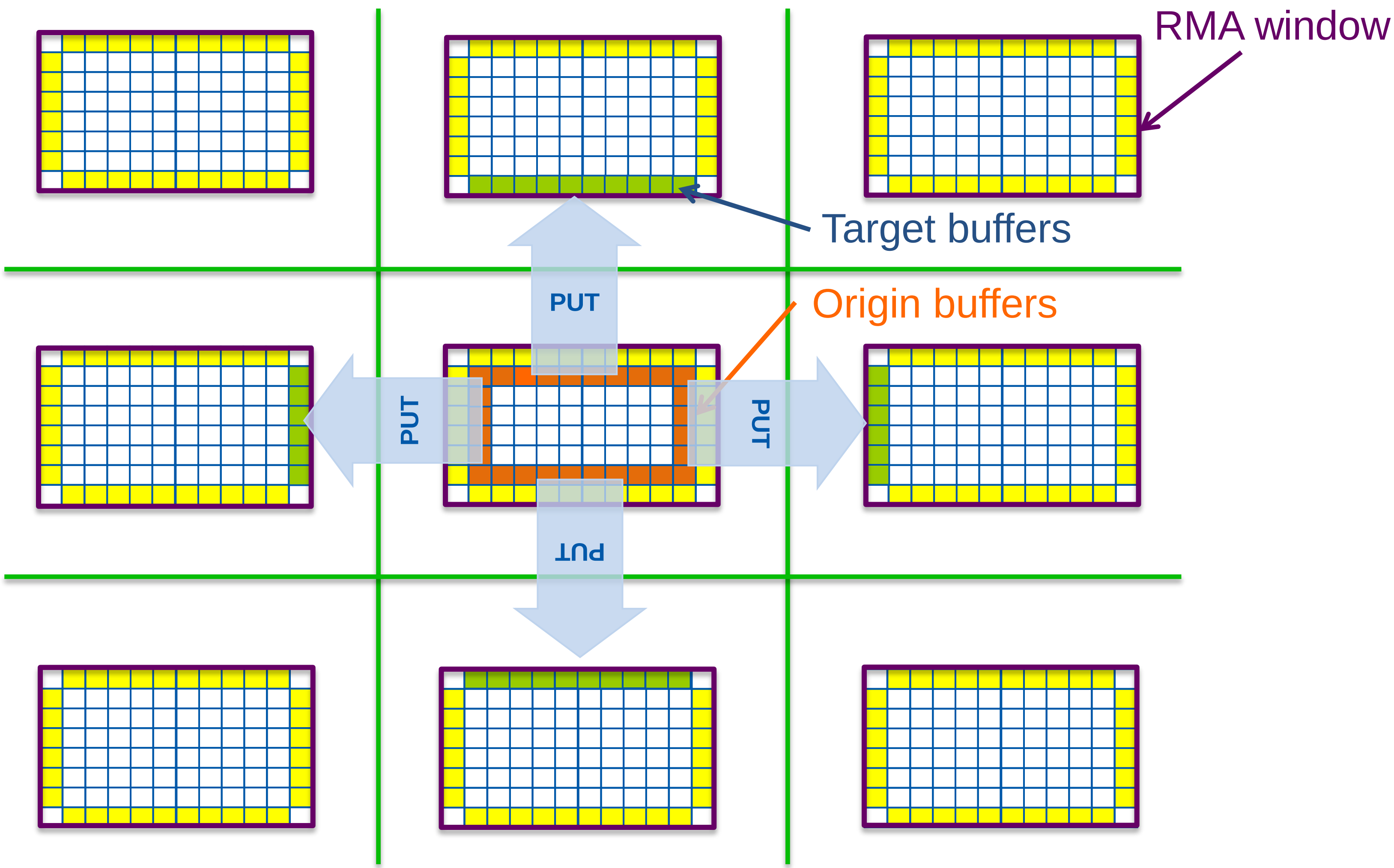
```
MPI_Win_fence(int assert, MPI_Win win)
```

- Collective synchronization model
- Starts *and* ends access and exposure epochs on all processes in the window

1. All processes in group of *win* do an MPI_WIN_FENCE to **open** an epoch
2. Everyone can issue PUT/GET operations to read/write data
3. Everyone does an MPI_WIN_FENCE to **close** the epoch
4. All operations complete at the second fence synchronization



Implementing Stencil Computation with RMA Fence



Code Example

- *stencil_mpi_ddt_rma.c*
- MPI_Put used to move data; explicit receives not needed
- Data location specified by MPI datatypes
- Manual packing of data no longer required

PSCW: Generalized Active Target Synchronization

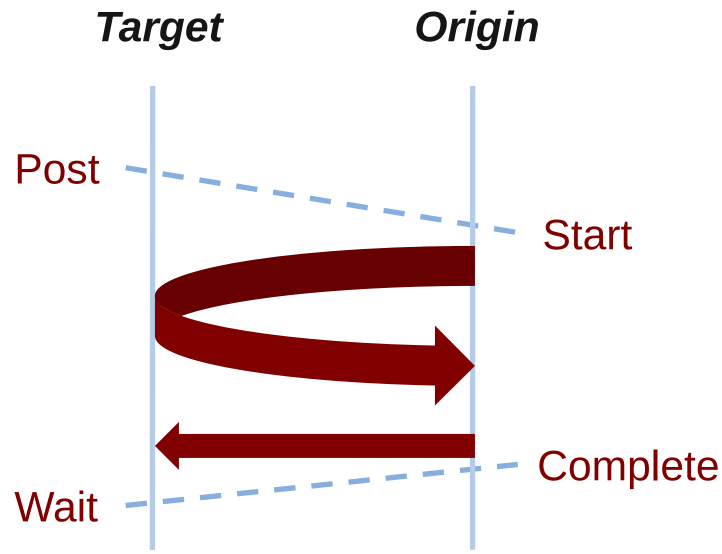
```
MPI_Win_start/post(MPI_Group grp, int assert, MPI_Win win)
MPI_Win_complete/wait(MPI_Win win)
```

- Like FENCE, but origin and target specify who they communicate with

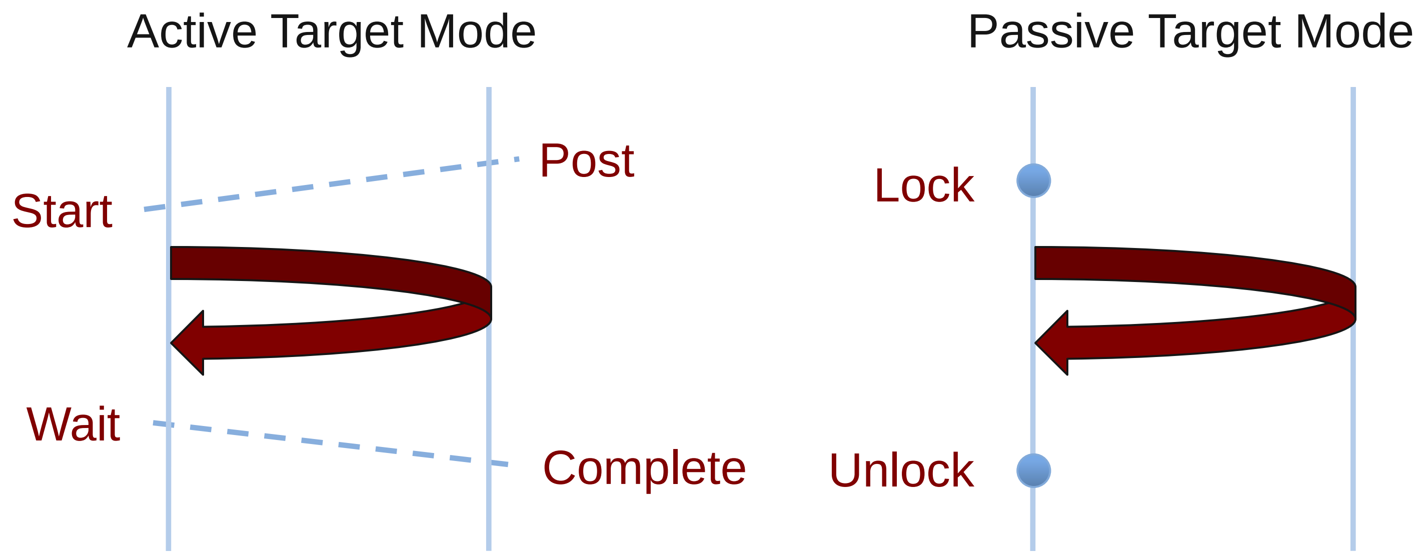
- Target: **Exposure epoch**
  - Opened with MPI_Win_post
  - Closed by MPI_Win_wait

- 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

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Passive Target Synchronization

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

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

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

- 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

Advanced Passive Target Synchronization

```
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)
```

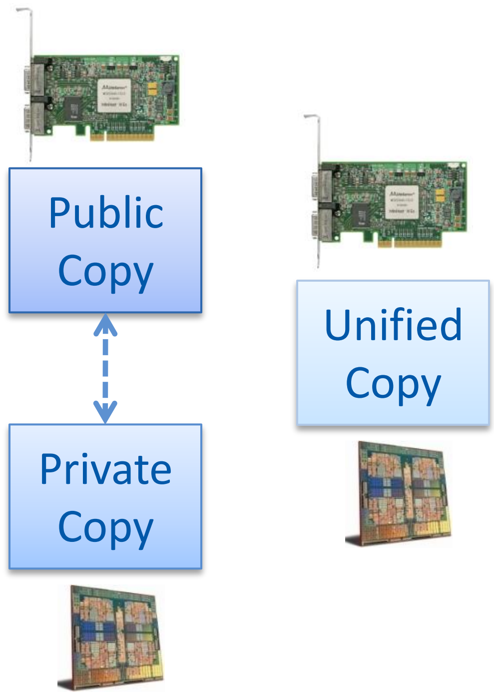
- Lock_all: **Shared** lock, passive target epoch to all other procs.
  - Expected usage is long-lived: lock_all, put/get, flush, …, unlock_all
- Flush_all – remotely complete RMA operations to all procs.
- Flush_local_all – locally complete RMA operations to all procs.

Which synchronization mode should I use, when?

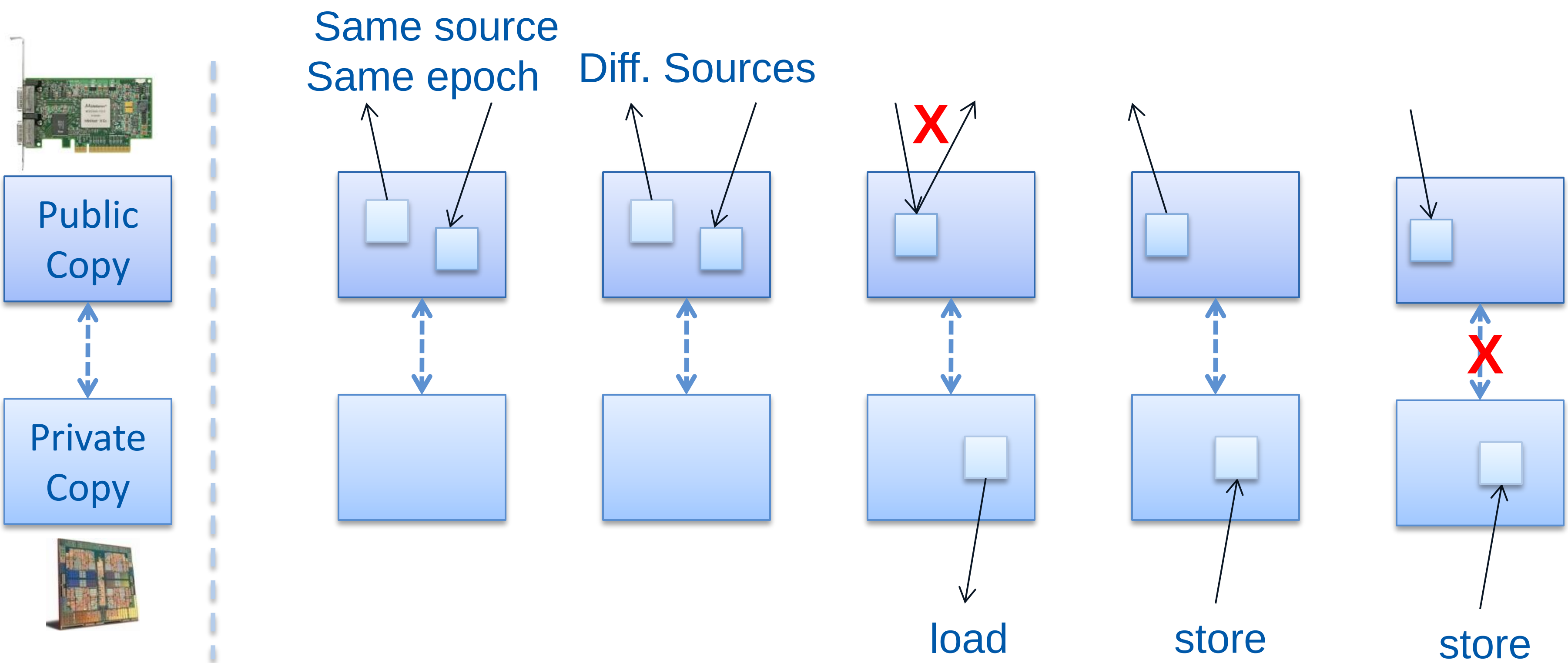
- RMA communication has low overheads versus send/recv
  - Two-sided: Matching, queuing, buffering, unexpected receives, etc.
  - One-sided: No matching, no buffering, always ready to receive
  - Utilize RDMA provided by high-speed interconnects (e.g. InfiniBand)
- Active mode: bulk synchronization
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

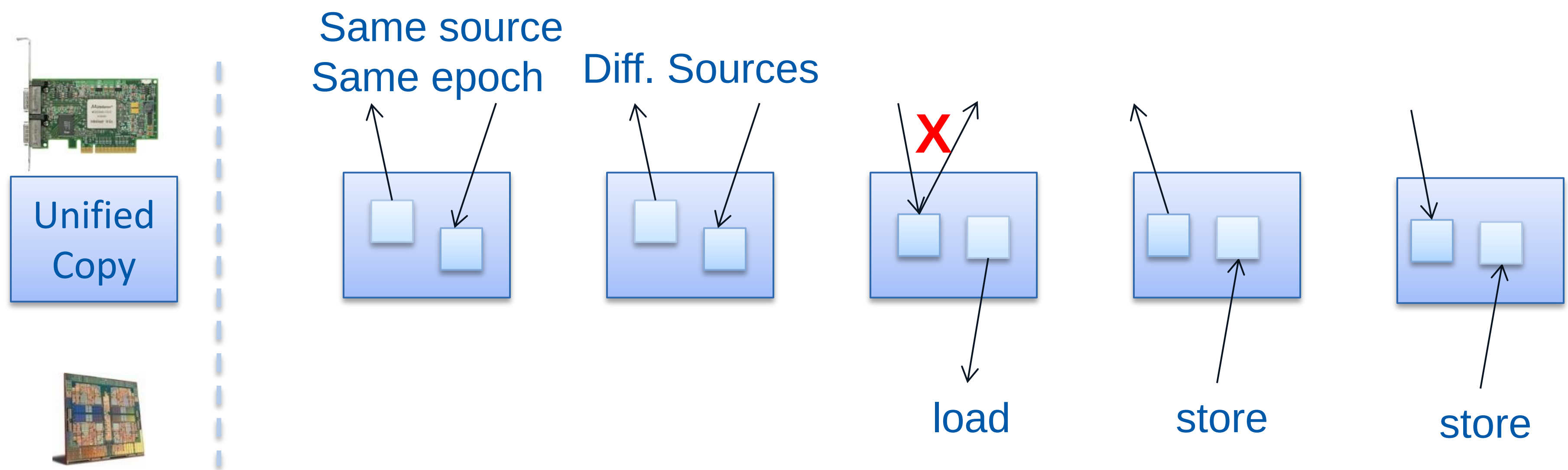


MPI RMA Memory Model (separate windows)



- 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

www.bsc.es

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Thank you!

For further information please contact

marc.jorda@bsc.es, antonio.pena@bsc.es