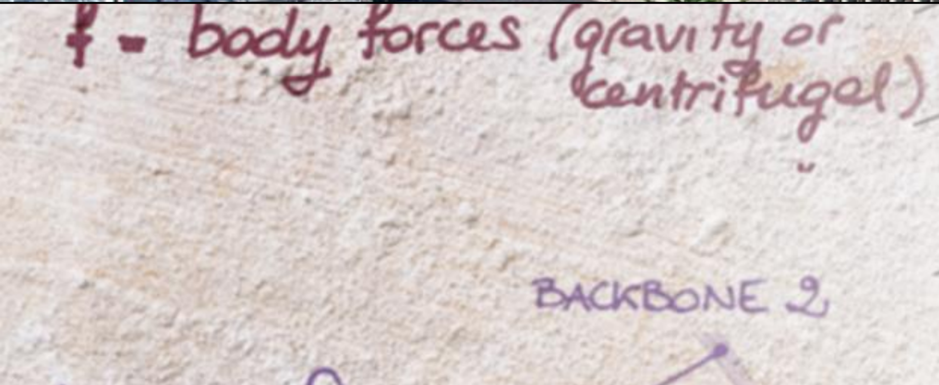
TORSTEN HOEFLER

# MPI Remote Memory Access Programming (MPI3-RMA) and Advanced MPI Programming

presented at RWTH Aachen, Jan. 2019
based on tutorials in collaboration with Bill Gropp, Rajeev Thakur, and Pavan Balaji




sighpc

acm

## The PASC19 Conference

The Platform for Advanced Scientific Computing (PASC) Conference, co-sponsored by the Association for Computing Machinery (ACM) and the Swiss National Supercomputing Centre (CSCS), will be held from June 12 to 14, 2019 at ETH Zurich, located in Zurich, Switzerland

# MPI-1

- **MPI is a message-passing library interface standard.**
  - Specification, not implementation
  - Library, not a language
  - All explicit parallelism, no magic

- **MPI-1 supports the classical message-passing programming model: basic point-to-point communication, collectives, datatypes, etc**

- **MPI-1 was defined (1994) by a broadly based group of parallel computer vendors, computer scientists, and applications developers.**
  - 2-year intensive process

- **Implementations appeared quickly and now MPI is taken for granted as vendor-supported software on any parallel machine.**

- **Free, portable implementations exist for clusters and other environments (MPICH, Open MPI)**

# Timeline of the MPI Standard

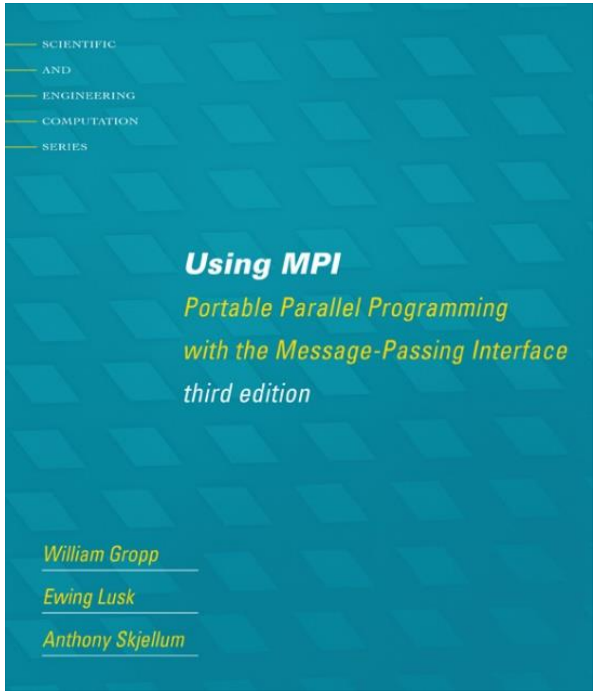
- **MPI-1 (1994), presented at SC'93**
  - Basic point-to-point communication, collectives, datatypes, etc
- **MPI-2 (1997)**
  - Added parallel I/O, Remote Memory Access (one-sided operations), dynamic processes, thread support, C++ bindings, …
- **---- Stable for 10 years ----**
- **MPI-2.1 (2008)**
  - Minor clarifications and bug fixes to MPI-2
- **MPI-2.2 (2009)**
  - Small updates and additions to MPI 2.1
- **MPI-3.0 (2012)**
  - Major new features and additions to MPI
- **MPI-3.1 (2015)**
  - Minor updates and fixes to MPI 3.0

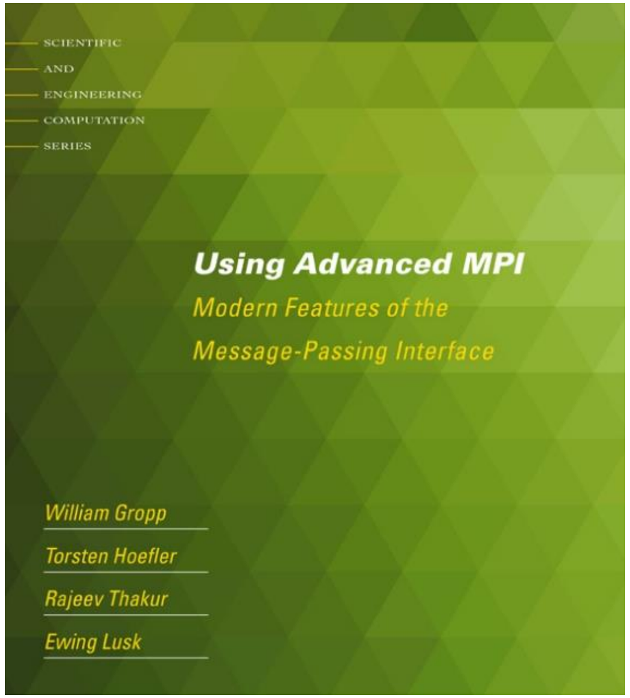
# Overview of New Features in MPI-3

- **Major new features**
  - Nonblocking collectives
  - Neighborhood collectives
  - Improved one-sided communication interface
  - Tools interface
  - Fortran 2008 bindings
- **Other new features**
  - Matching Probe and Recv for thread-safe probe and receive
  - Noncollective communicator creation function
  - "const" correct C bindings
  - Comm_split_type function
  - Nonblocking Comm_dup
  - Type_create_hindexed_block function
- **C++ bindings removed**
- **Previously deprecated functions removed**
- **MPI 3.1 added nonblocking collective I/O functions**

# Tutorial Books on MPI

- **For basic MPI**
  - ***Using MPI, 3rd edition, 2014***, by William Gropp, Ewing Lusk, and Anthony Skjellum
  - https://mitpress.mit.edu/books/using-MPI-third-edition
- **For advanced MPI, including MPI-3**
  - ***Using Advanced MPI, 2014*****,** by William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk
  - https://mitpress.mit.edu/books/using-advanced-MPI

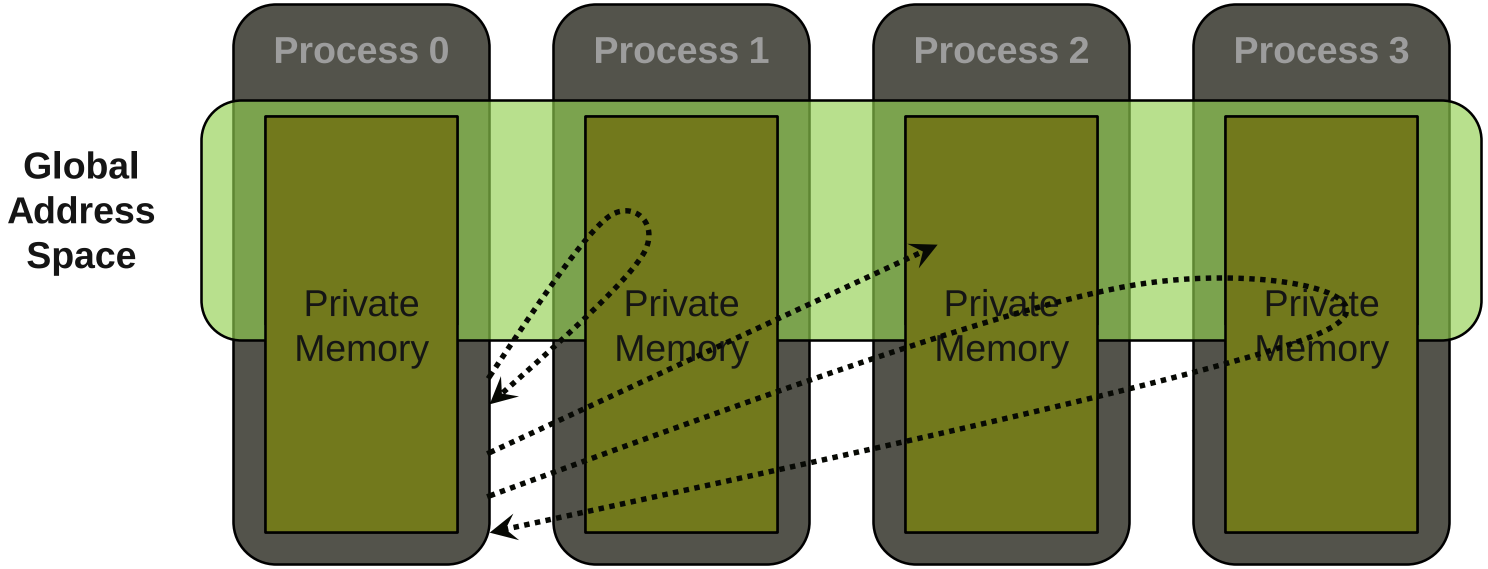


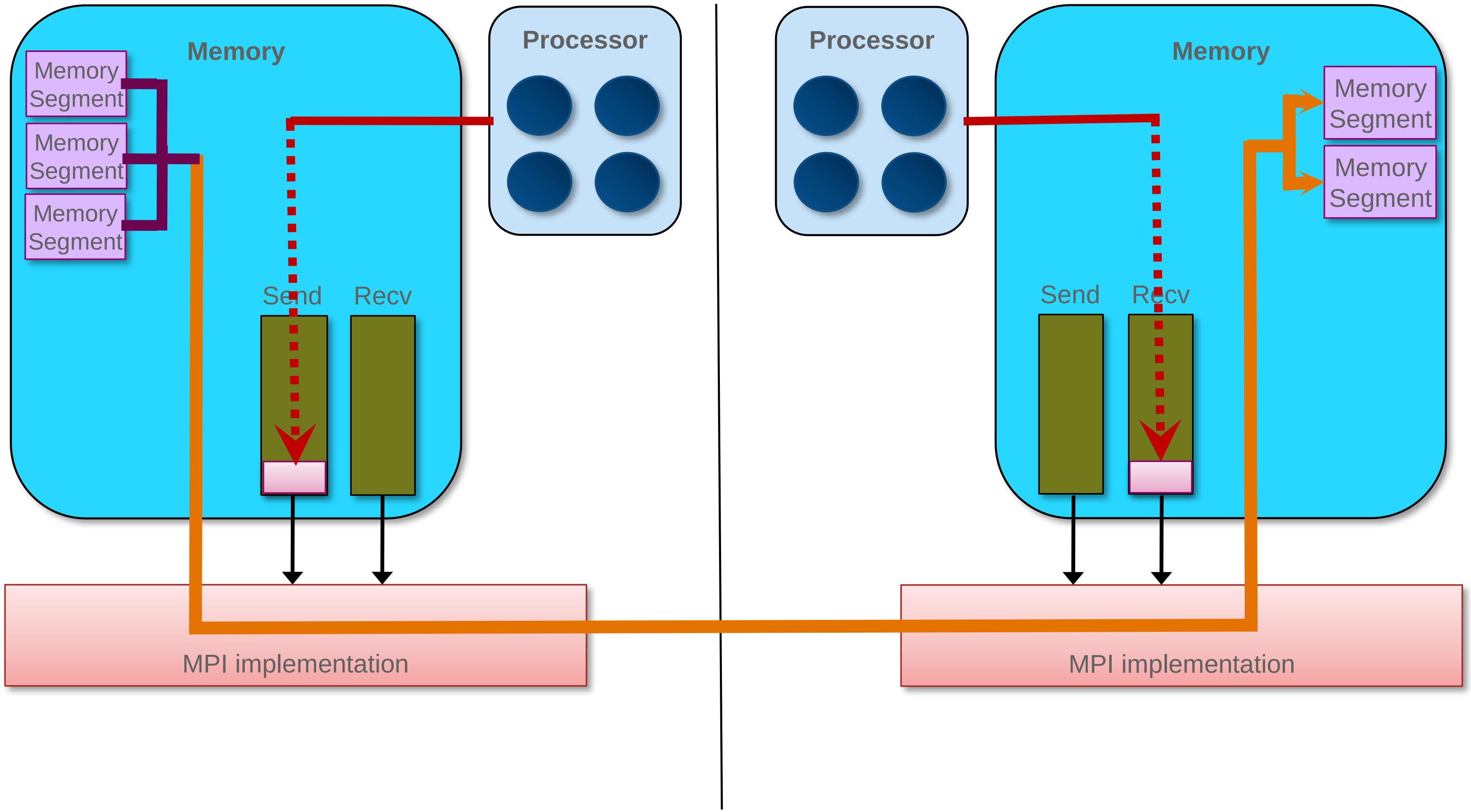
# Advanced Topics: One-sided Communication

# One-sided Communication

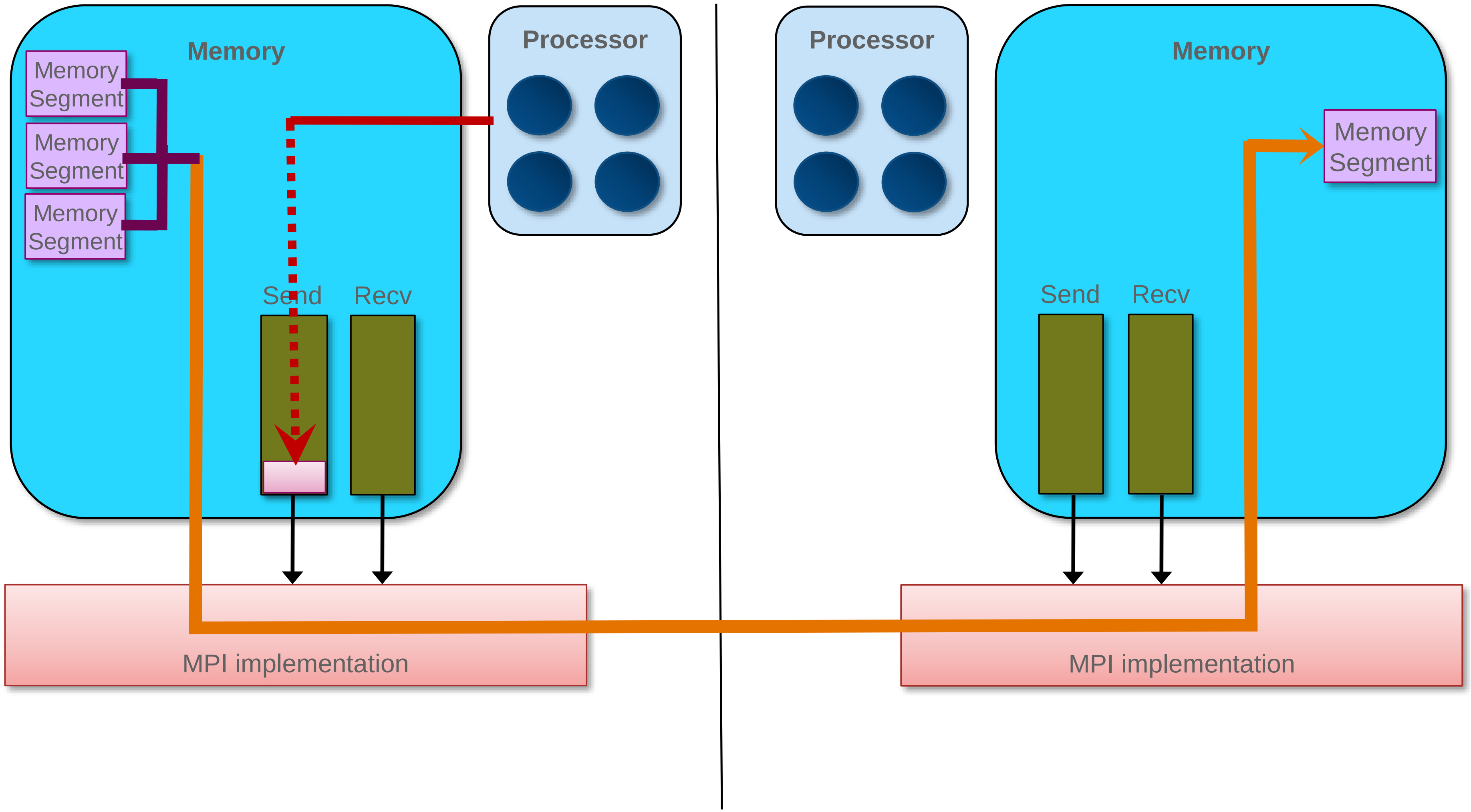
- **The basic idea of one-sided communication models is to decouple data movement with process synchronization**
  - Should be able to move data without requiring that the remote process synchronize
  - Each process exposes a part of its memory to other processes
  - Other processes can directly read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



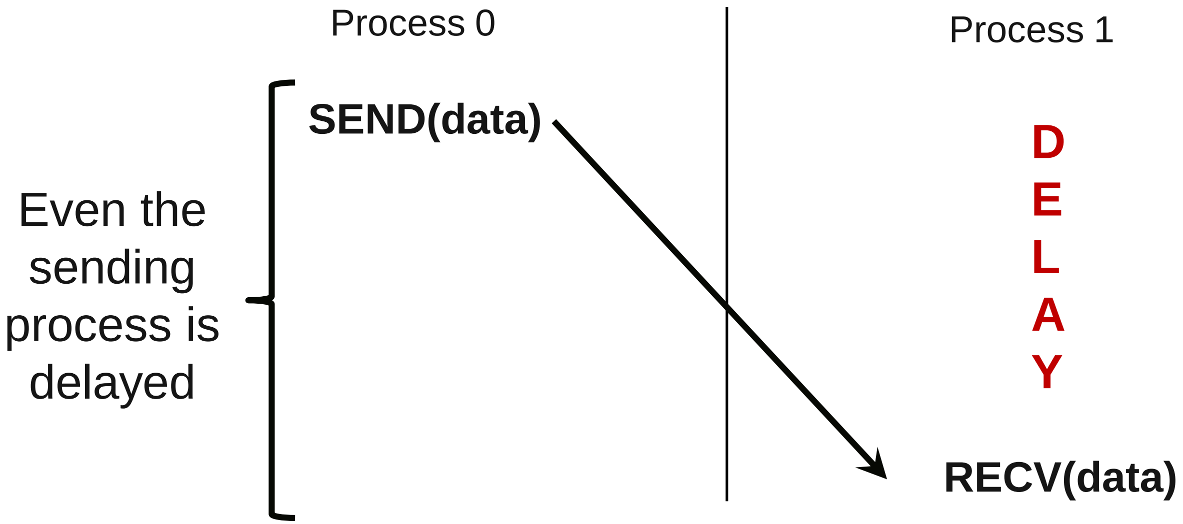
# Two-sided Communication Example

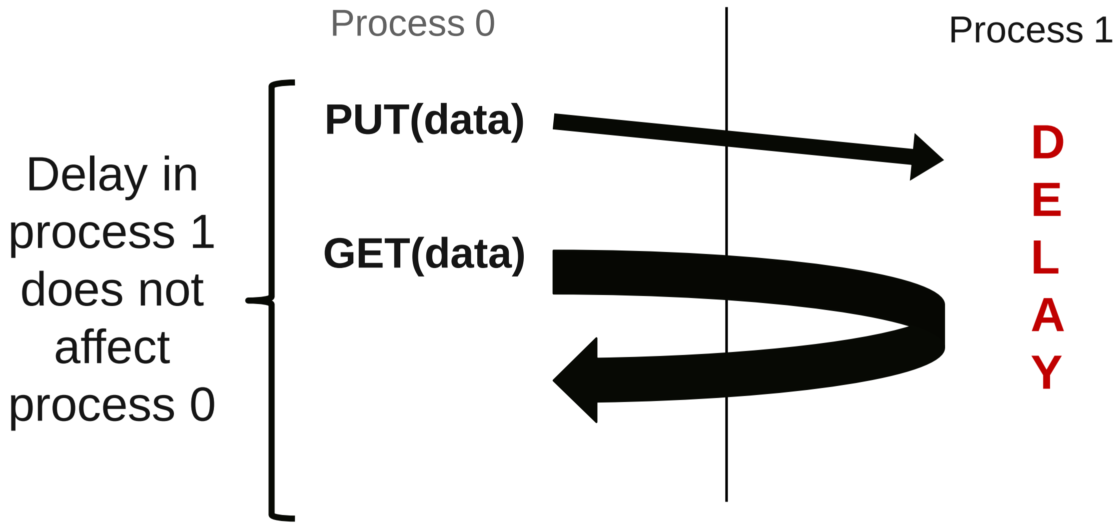


# One-sided Communication Example



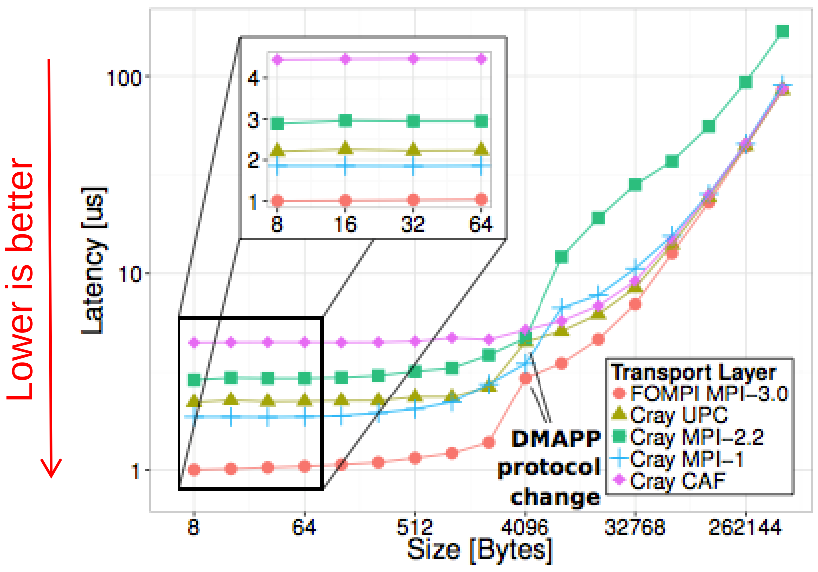
# Comparing One-sided and Two-sided Programming



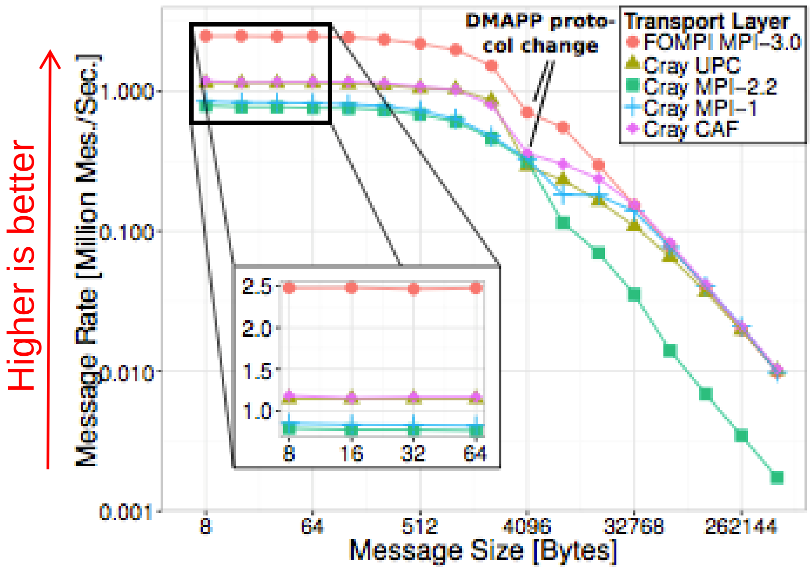



# MPI RMA can be efficiently implemented

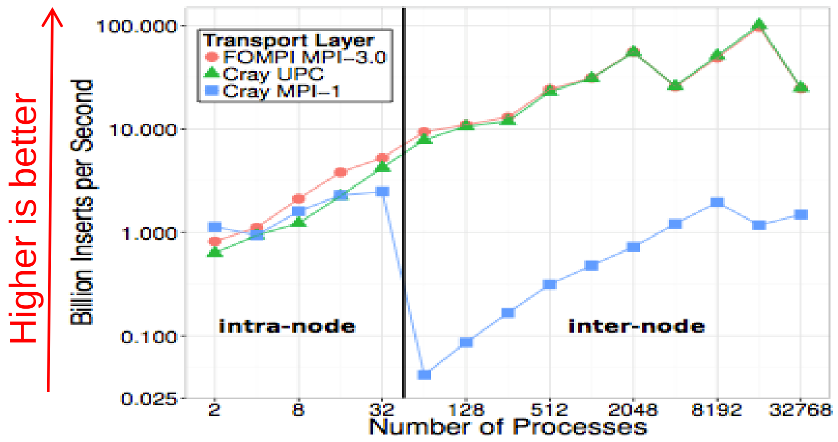
- "Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided" by Robert Gerstenberger, Maciej Besta, Torsten Hoefler (SC13 Best Paper Award)
- They implemented complete MPI-3 RMA for Cray Gemini (XK5, XE6) and Aries (XC30) systems on top of lowest-level Cray APIs
- Achieved better latency, bandwidth, message rate, and application performance than Cray's MPI RMA, UPC, and Coarray Fortran




(a) Latency inter-node Put

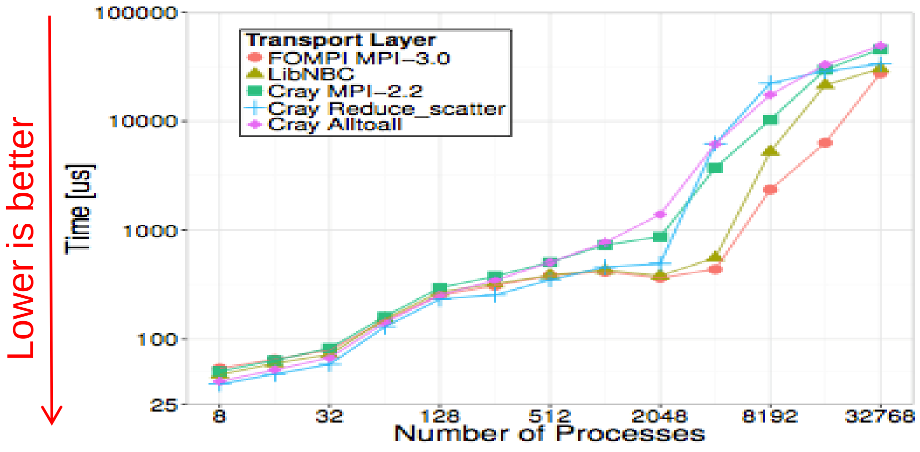



(b) Message Rate inter-node



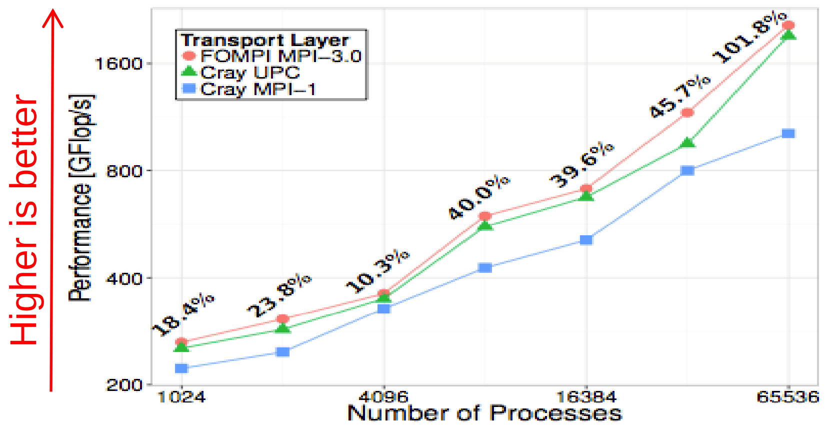

(a) Inserts per second for inserting 16k elements per process including synchronization.

Distributed Hash Table




(b) Time to perform one dynamic sparse data exchange (DSDE) with 6 random neighbors

Dynamic Sparse Data Exchange




(c) 3D FFT Performance. The annotations represent the improvement of FoMPI over MPI-1.

3D FFT

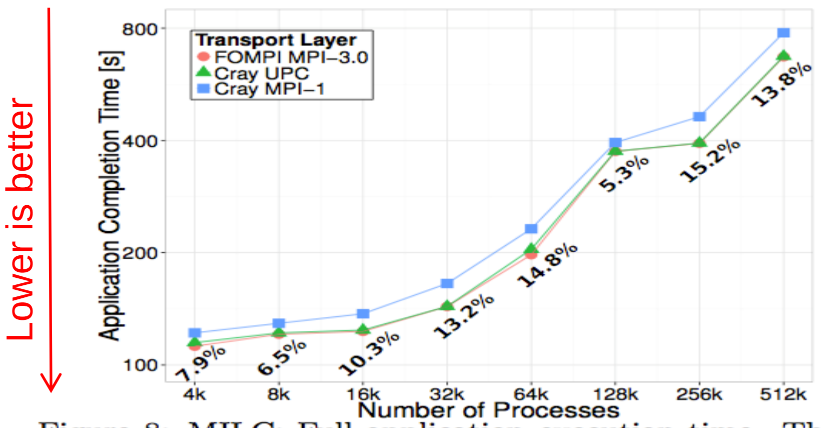



Figure 8: MILC: Full application execution time. The annotations represent the improvement of FoMPI and UPC over MPI-1.

MILC

# MPI RMA is Carefully and Precisely Specified

- **To work on both cache-coherent and non-cache-coherent systems**
  - Even though there aren't many non-cache-coherent systems, it is designed with the future in mind

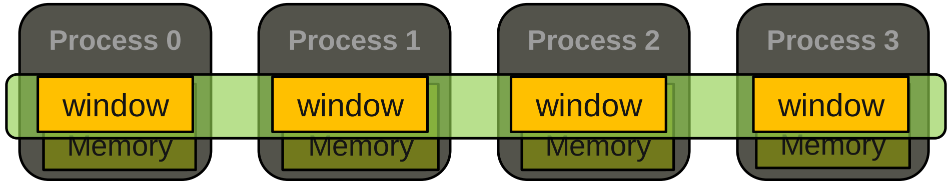
- **There even exists a *formal model* for MPI-3 RMA that can be used by tools and compilers for optimization, verification, etc.**
  - See "Remote Memory Access Programming in MPI-3" by Hoefler, Dinan, Thakur, Barrett, Balaji, Gropp, Underwood. ACM TOPC, July 2015.
  - http://htor.inf.ethz.ch/publications/index.php?pub=201

# 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 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"
- **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_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 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_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 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

```
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_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;
}
```

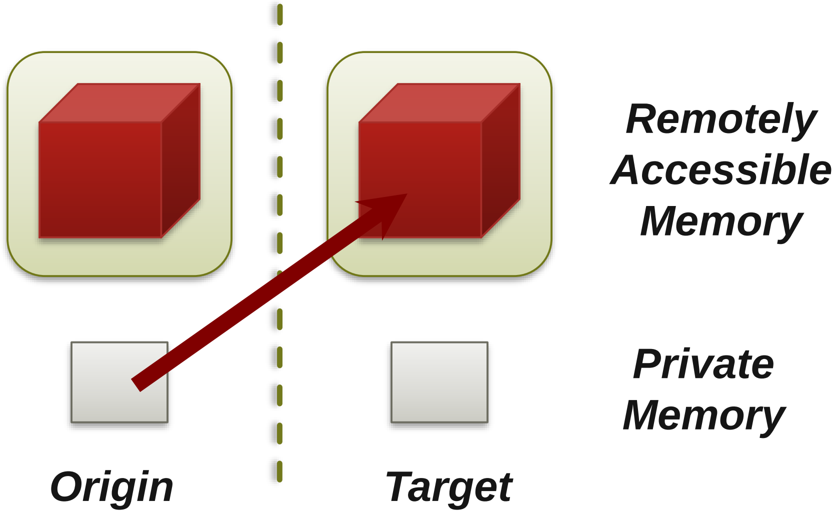
# Data movement

- 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(const 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)
```

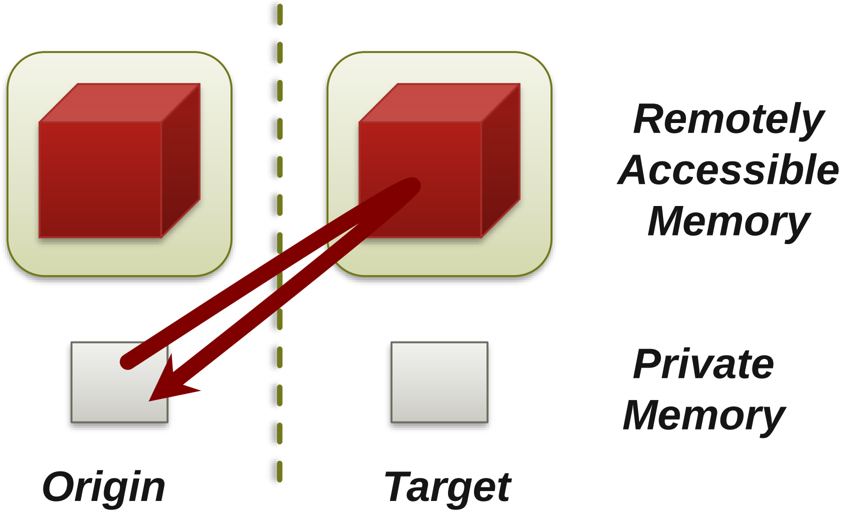
- **Move data from origin, to target**
- **Separate data description triples for origin and target**



# 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)
```

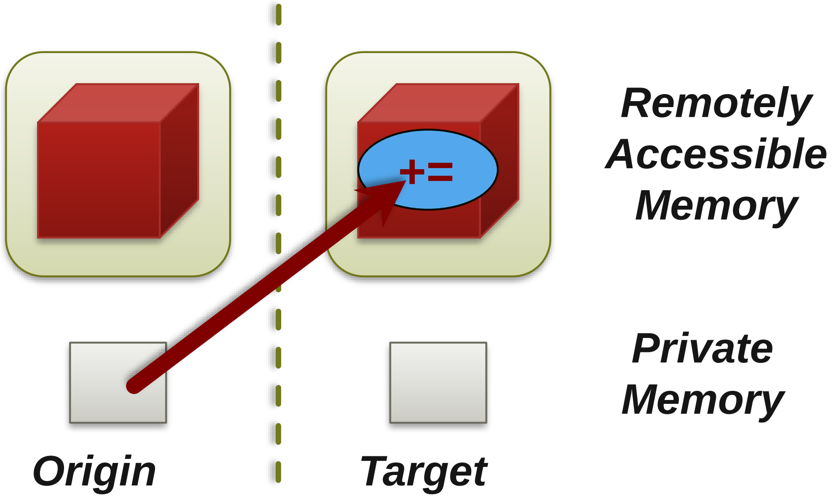
- **Move data to origin, from target**
- **Separate data description triples for origin and target**



# Atomic Data Aggregation: Accumulate

```
MPI_Accumulate(const 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)
```

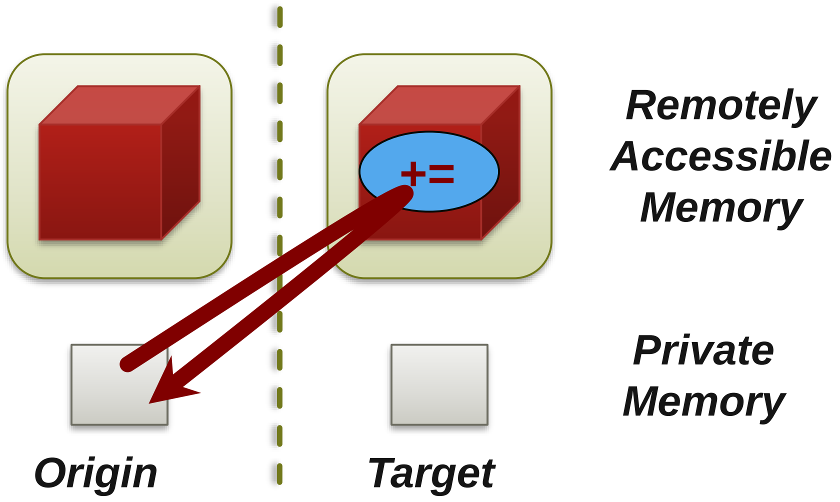
- **Atomic update operation, similar to a put**
  - Reduces origin and target data into target buffer using op argument as combiner
  - Op = `MPI_SUM`, `MPI_PROD`, `MPI_OR`, `MPI_REPLACE`, `MPI_NO_OP`, …
  - Predefined ops only, no user-defined operations
- **Different data layouts between target/origin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



# Atomic Data Aggregation: Get Accumulate

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

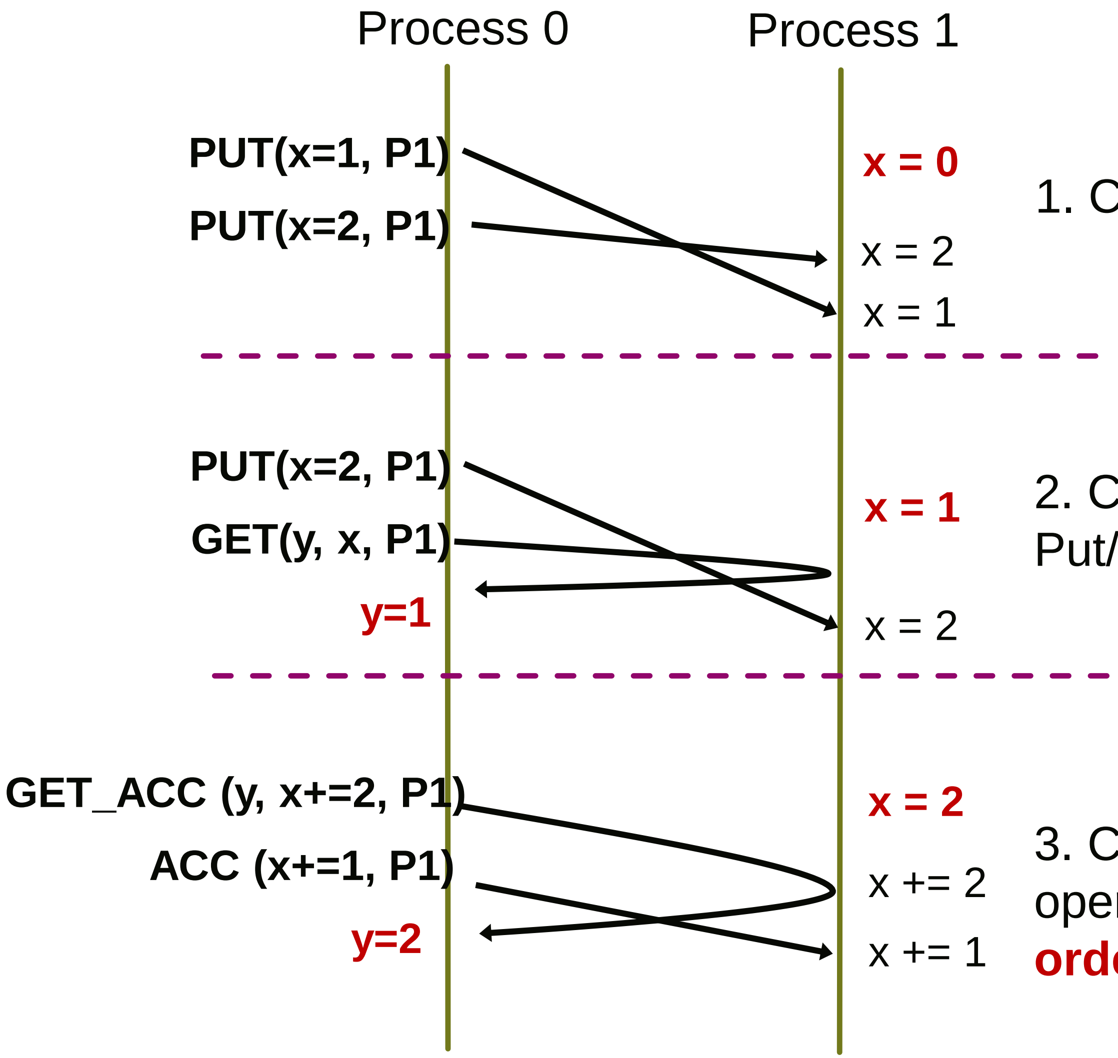
```
MPI_Compare_and_swap(const void *origin_addr,
        const void *compare_addr, void *result_addr,
        MPI_Datatype dtype, int target_rank,
        MPI_Aint target_disp, MPI_Win win)
```

- **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 (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

# 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 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)

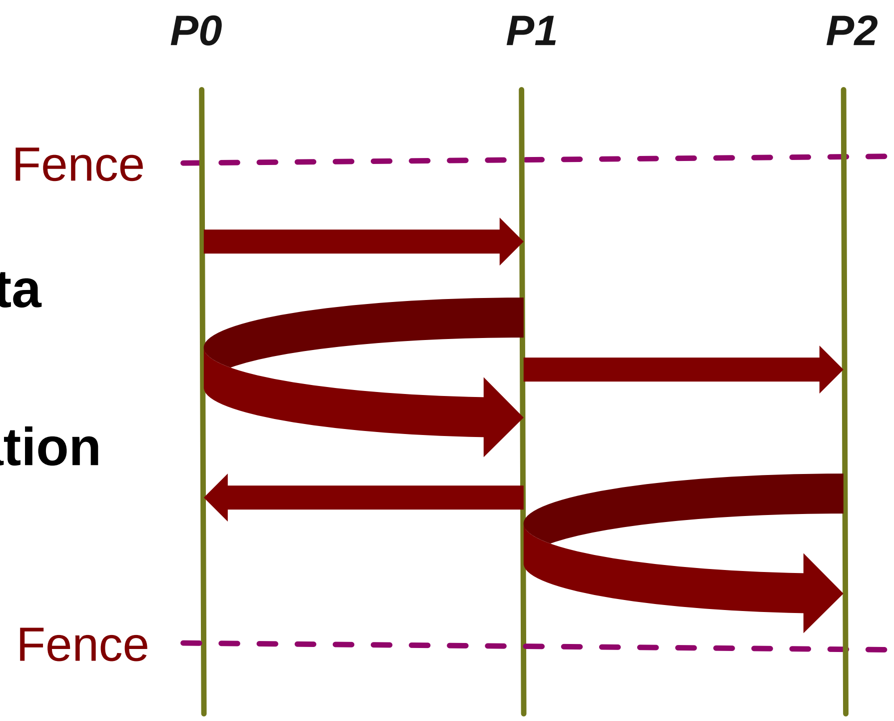
- **Data accesses occur within "epochs"**
  - *Access epochs*: contain a set of operations issued by an origin process
  - *Exposure epochs*: enable remote processes to update a target's window
  - Epochs define ordering and completion semantics
  - Synchronization models provide mechanisms for establishing epochs

    *E.g., starting, ending, and synchronizing epochs*

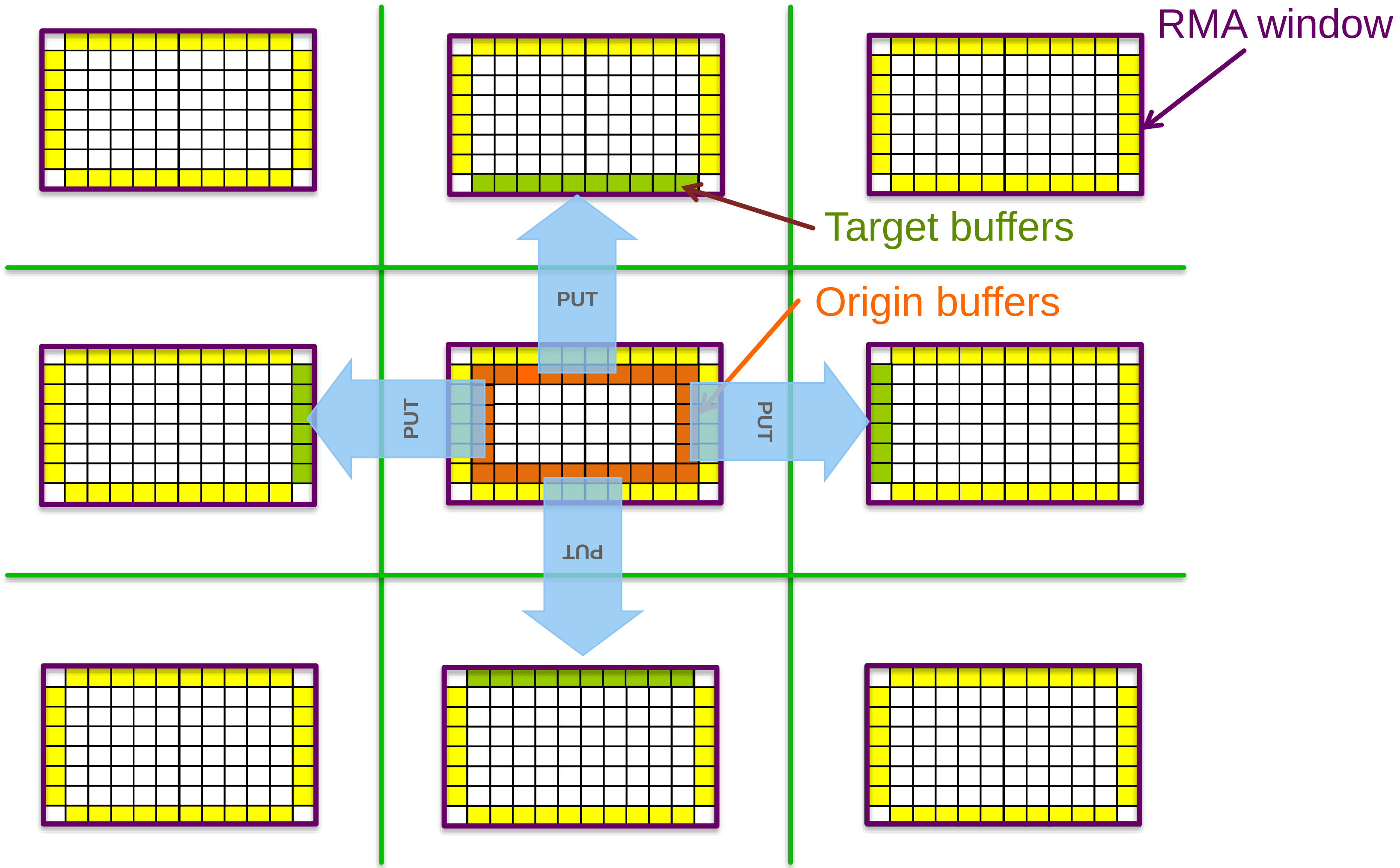
# Fence: Active Target Synchronization

```
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



# Example: Stencil with RMA Fence (1/2)



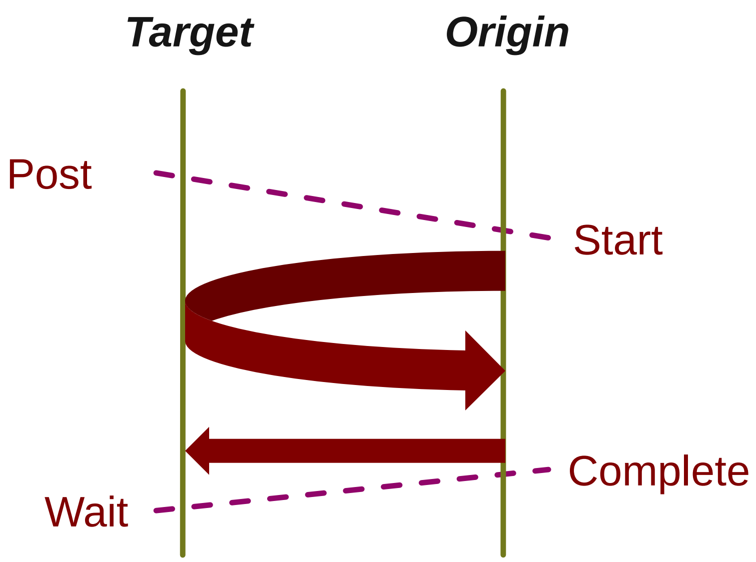
# Example: Stencil with RMA Fence (2/2)

- *stencil_mpi_ddt_rma.c*
- Use MPI_PUTs to move data, explicit receives are not needed
- Data location specified by MPI datatypes
- Manual packing of data no longer required

# PSCW: Generalized Active Target Synchronization

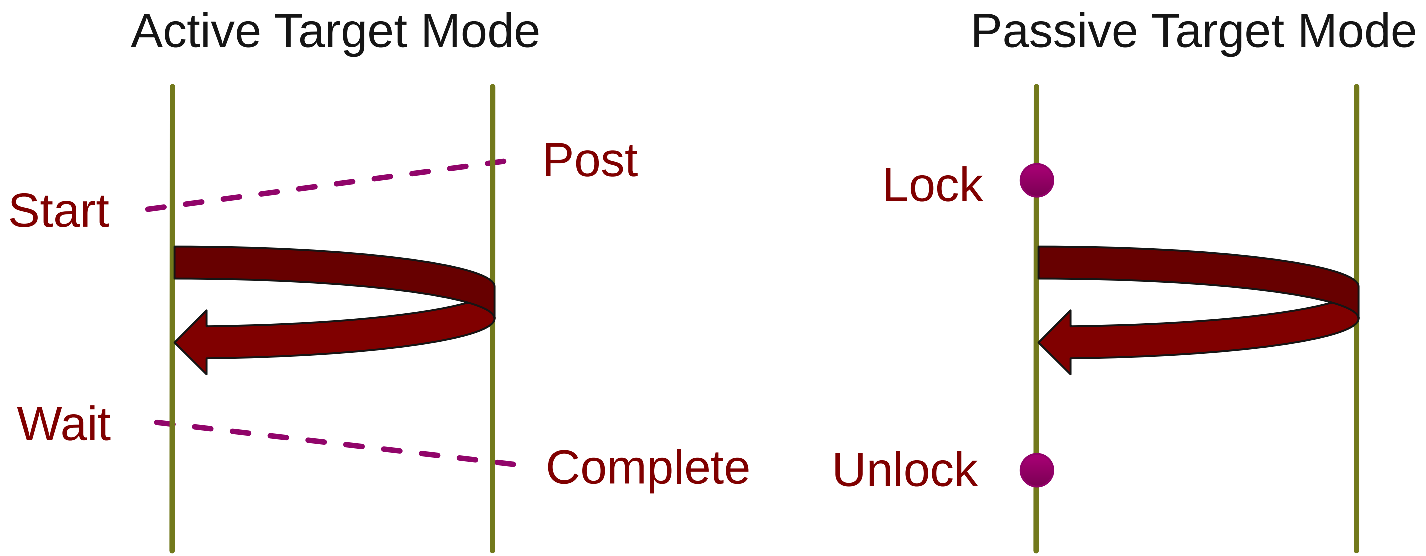
```
MPI_Win_post/start(MPI_Group grp, int assert, MPI_Win win)
MPI_Win_complete/wait(MPI_Win win)
```

- **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)
```

```
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**

# Newer 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)
```

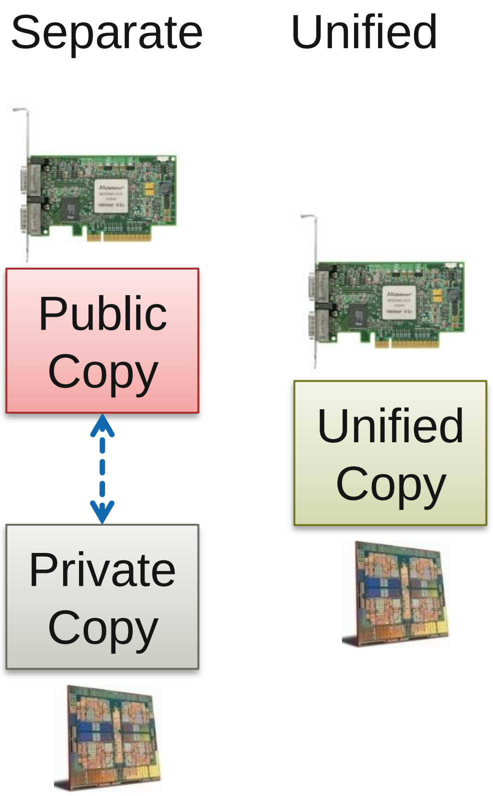
- **`Lock_all`: Shared lock, passive target epoch to all other processes**
  - Expected usage is long-lived: `lock_all`, `put/get`, `flush`, …, `unlock_all`
- **`Flush_all` – remotely complete RMA operations to all processes**
- **`Flush_local_all` – locally complete RMA operations to all processes**

# MPI RMA Memory Model

- **MPI-3 provides two memory models: separate and unified**

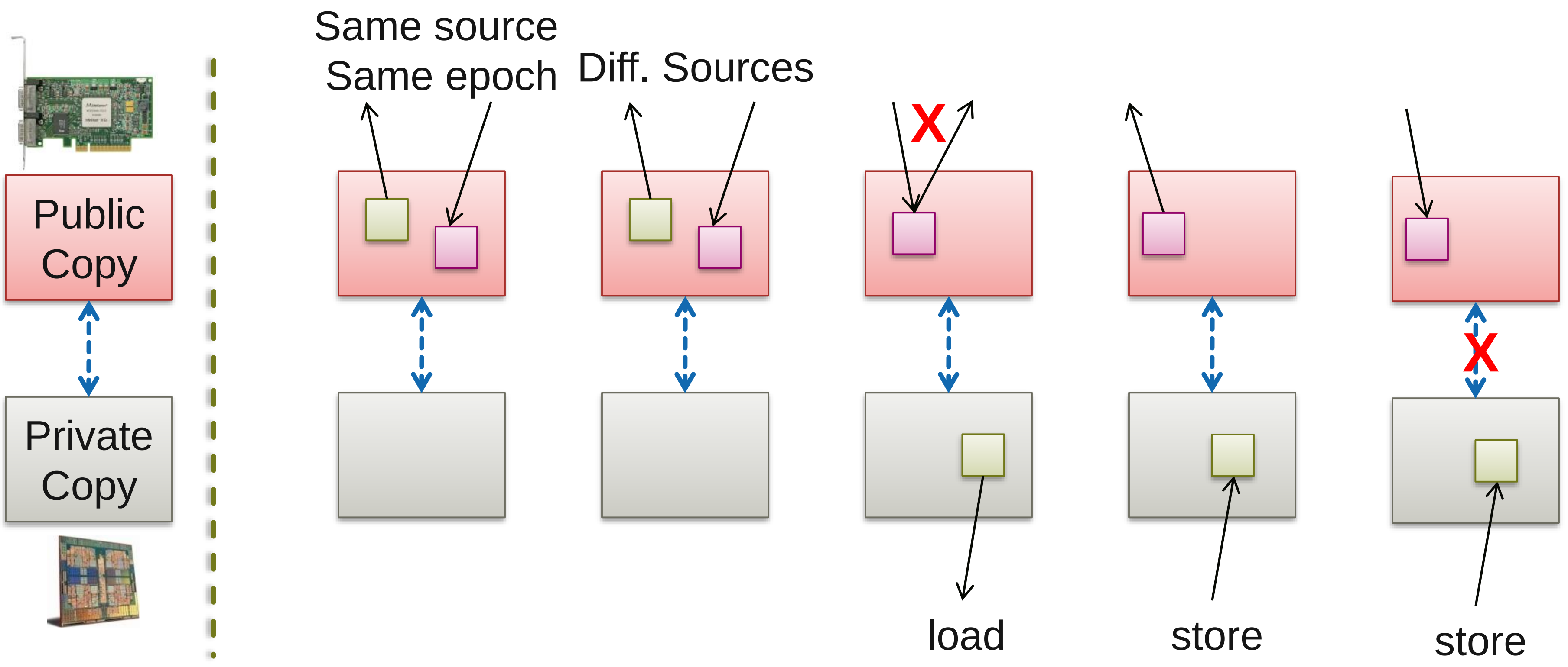
- **MPI-2: Separate Model**
  - Logical public and private copies
  - MPI provides software coherence between window copies
  - Extremely portable, to systems that don't provide hardware coherence

- **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



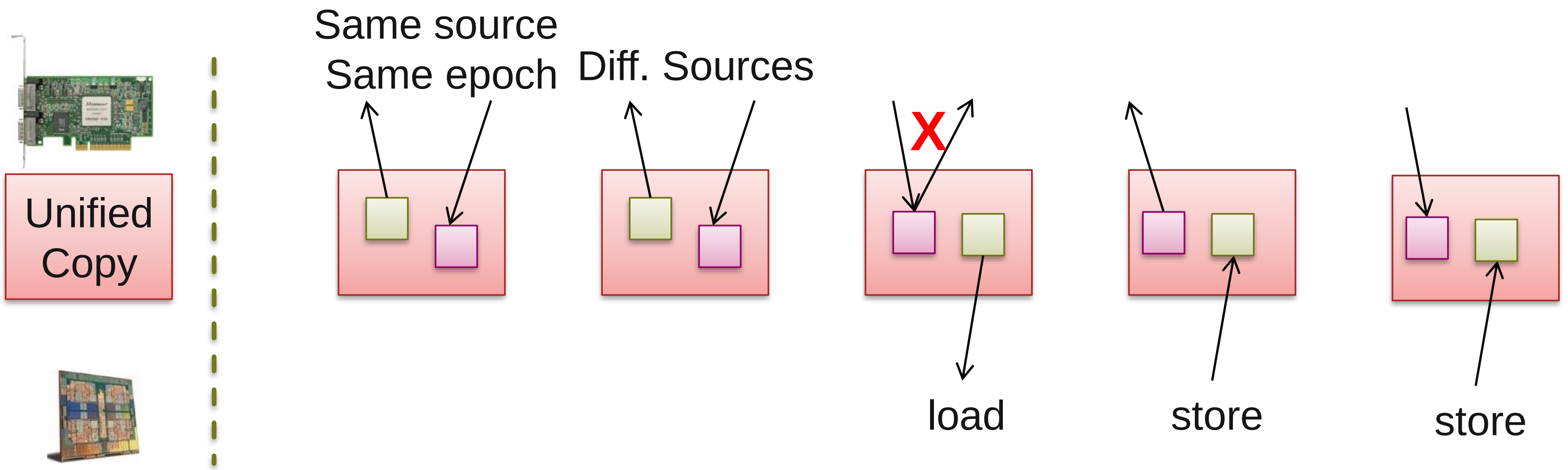
# MPI RMA Memory Model (separate windows)

- **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

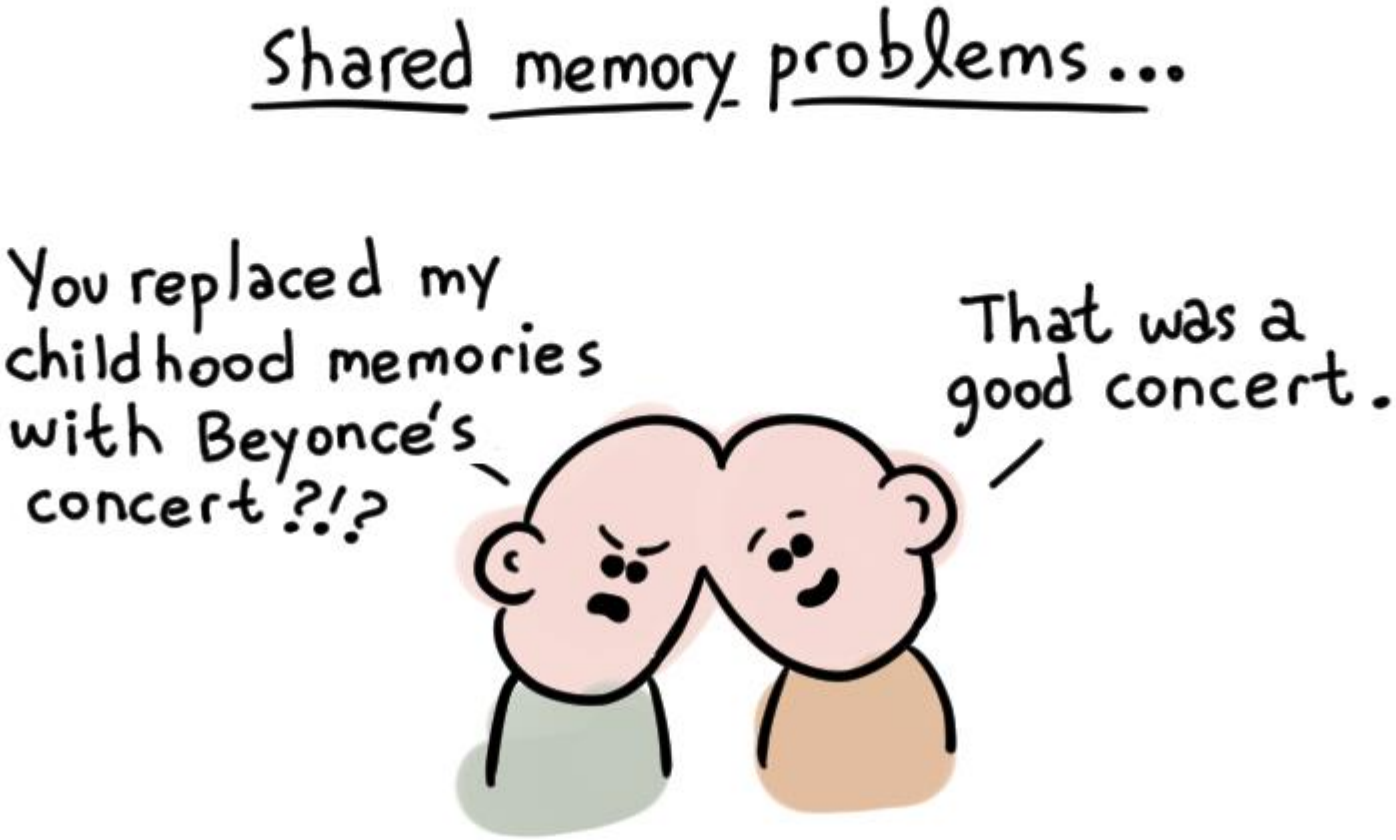
# MPI RMA Operation Compatibility (Unified)

| | 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

# MPI + Shared-Memory



# Hybrid Programming with Shared Memory

- **MPI-3 allows different processes to allocate shared memory through MPI**
  - MPI_Win_allocate_shared

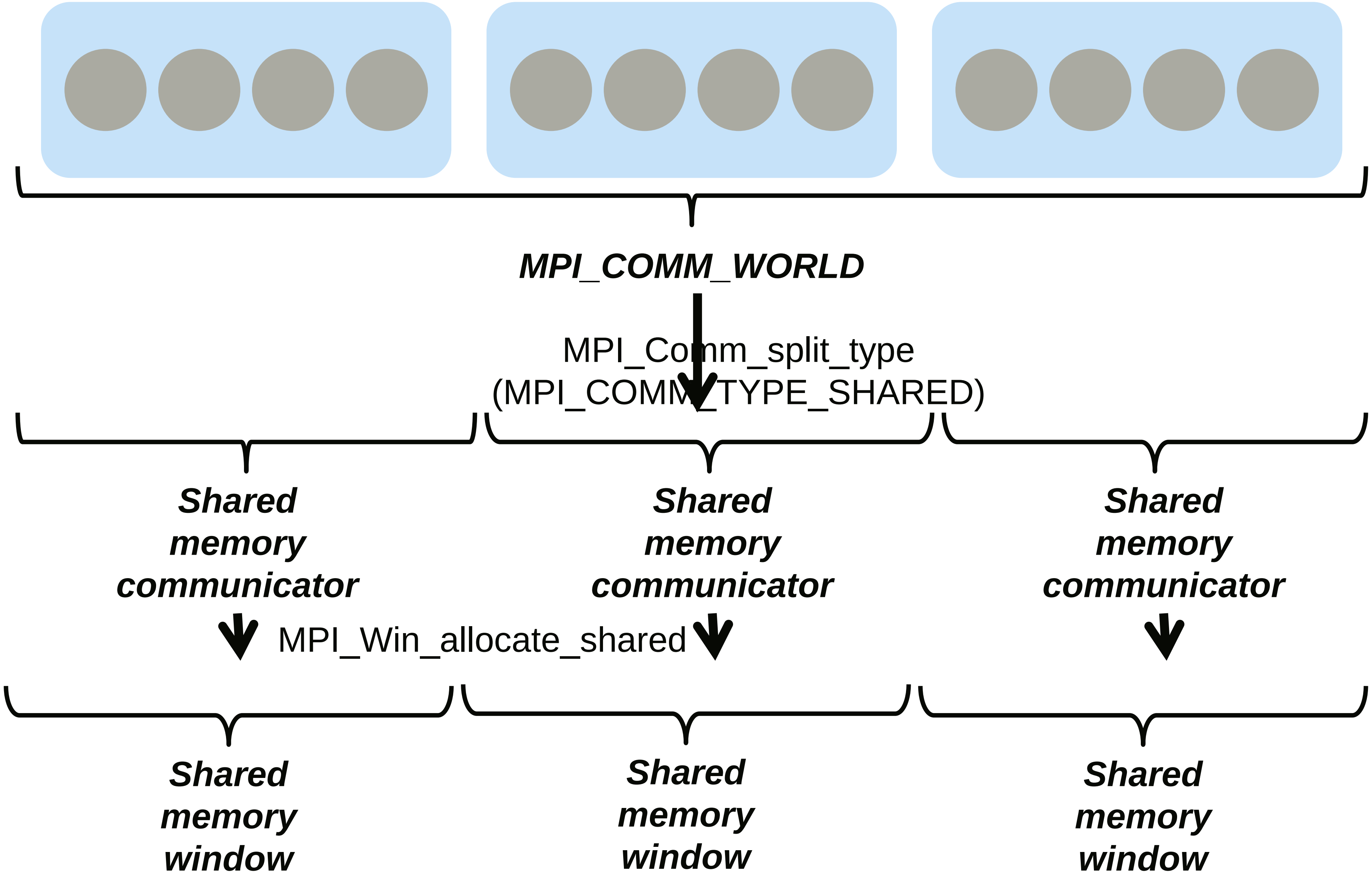
- **Uses many of the concepts of one-sided communication**

- **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**
  - Controlled sharing!

# Creating Shared Memory Regions in MPI

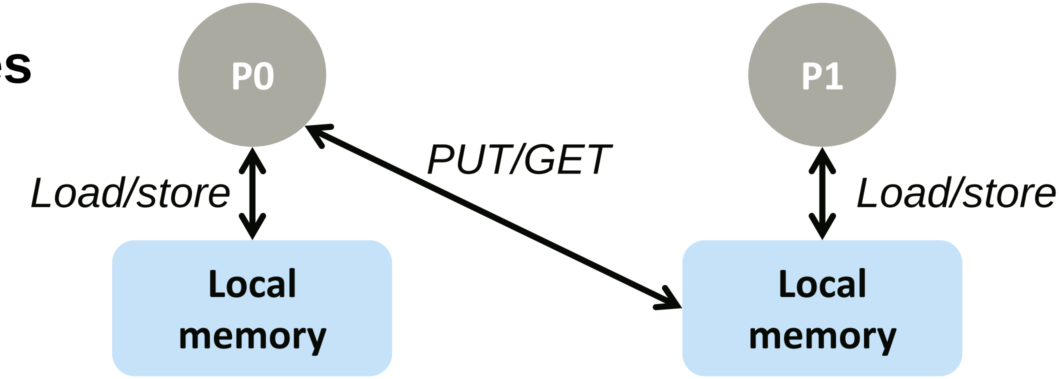


# Regular RMA windows vs. Shared memory windows

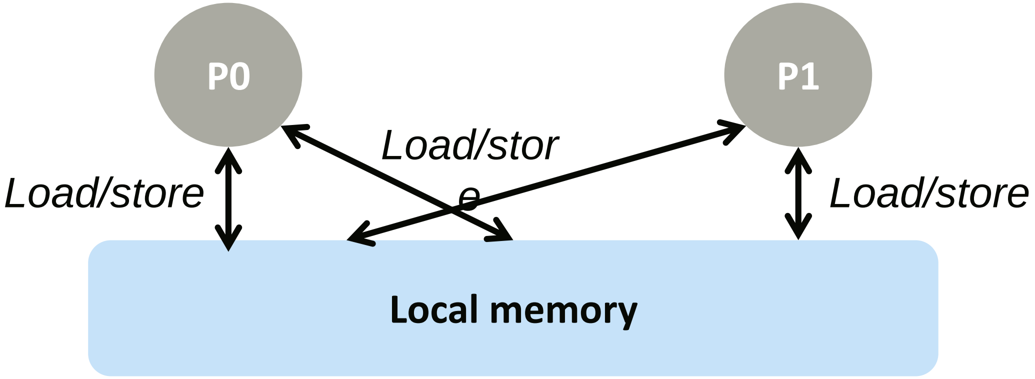
- **Shared memory windows allow application processes to directly perform load/store accesses on all of the window memory**
  - E.g., x[100] = 10

- **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




***Traditional RMA windows***




***Shared memory windows***

# MPI_COMM_SPLIT_TYPE

```
MPI_Comm_split_type(MPI_Comm comm, int split_type,
                    int key, MPI_Info info, MPI_Comm *newcomm)
```

- **Create a communicator where processes "share a property"**
  - Properties are defined by the "split_type"

- **Arguments:**
  - comm - input communicator (handle)
  - Split_type - property of the partitioning (integer)
  - Key - Rank assignment ordering (nonnegative integer)
  - info - info argument (handle)
  - newcomm- output communicator (handle)

# MPI_WIN_ALLOCATE_SHARED

```
MPI_Win_allocate_shared(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**
  - Data exposed in a window can be accessed with RMA ops or load/store

- **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)

# 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;
}
```

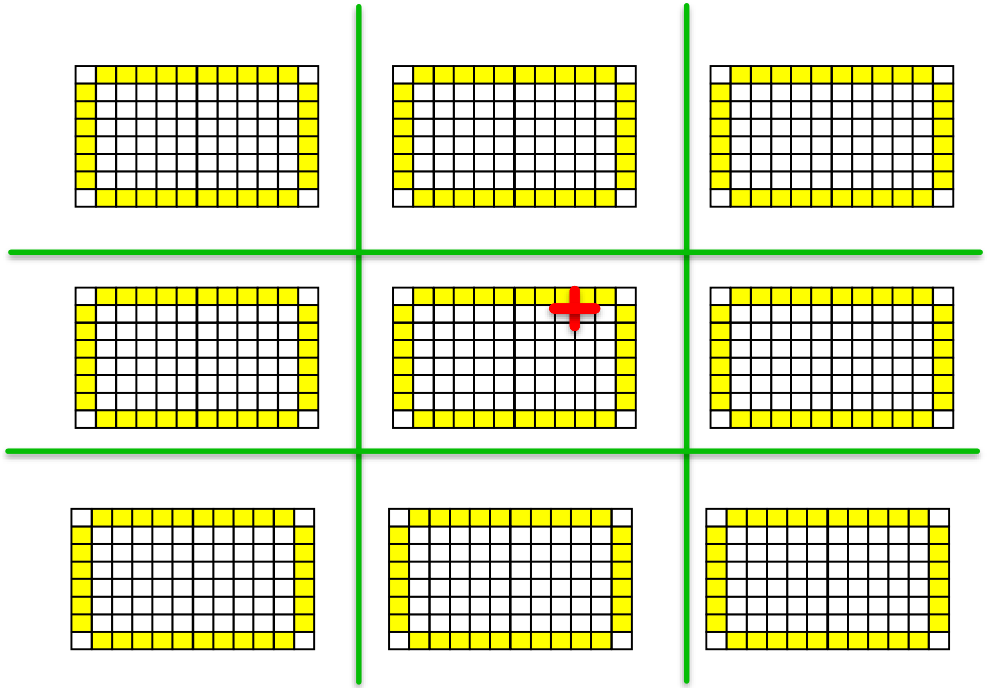
# Memory allocation and placement

- **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 (e.g., which physical memory it will be pinned to)**
  - 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”

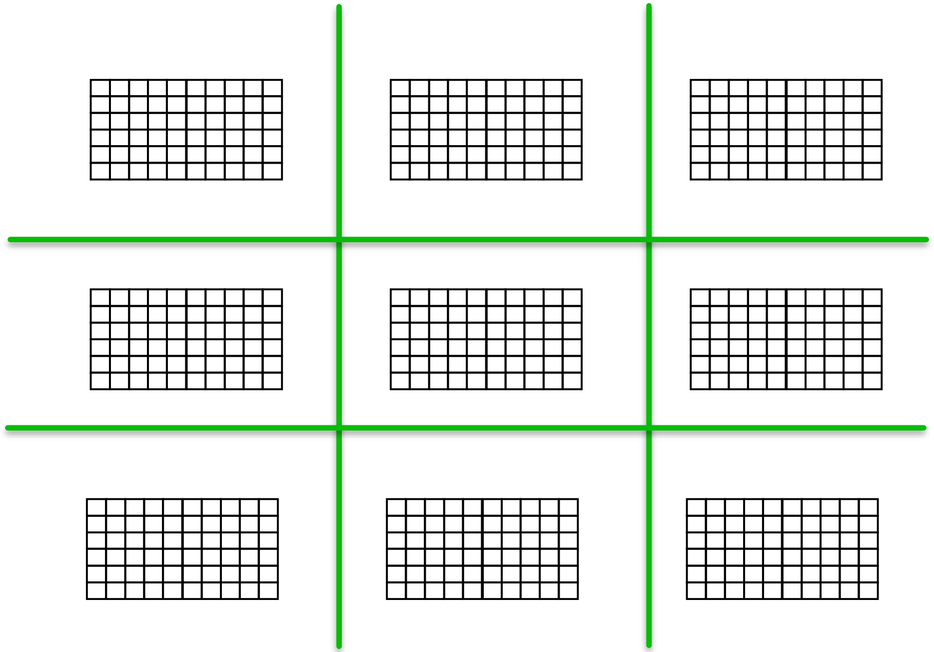
- **The total allocated shared memory on a communicator is contiguous by default**
  - Users can pass an info hint called “noncontig” that will allow the MPI implementation to align memory allocations from each process to appropriate boundaries to assist with placement

# Example Computation: Stencil



***Message passing model requires ghost-cells to be explicitly communicated to neighbor processes***

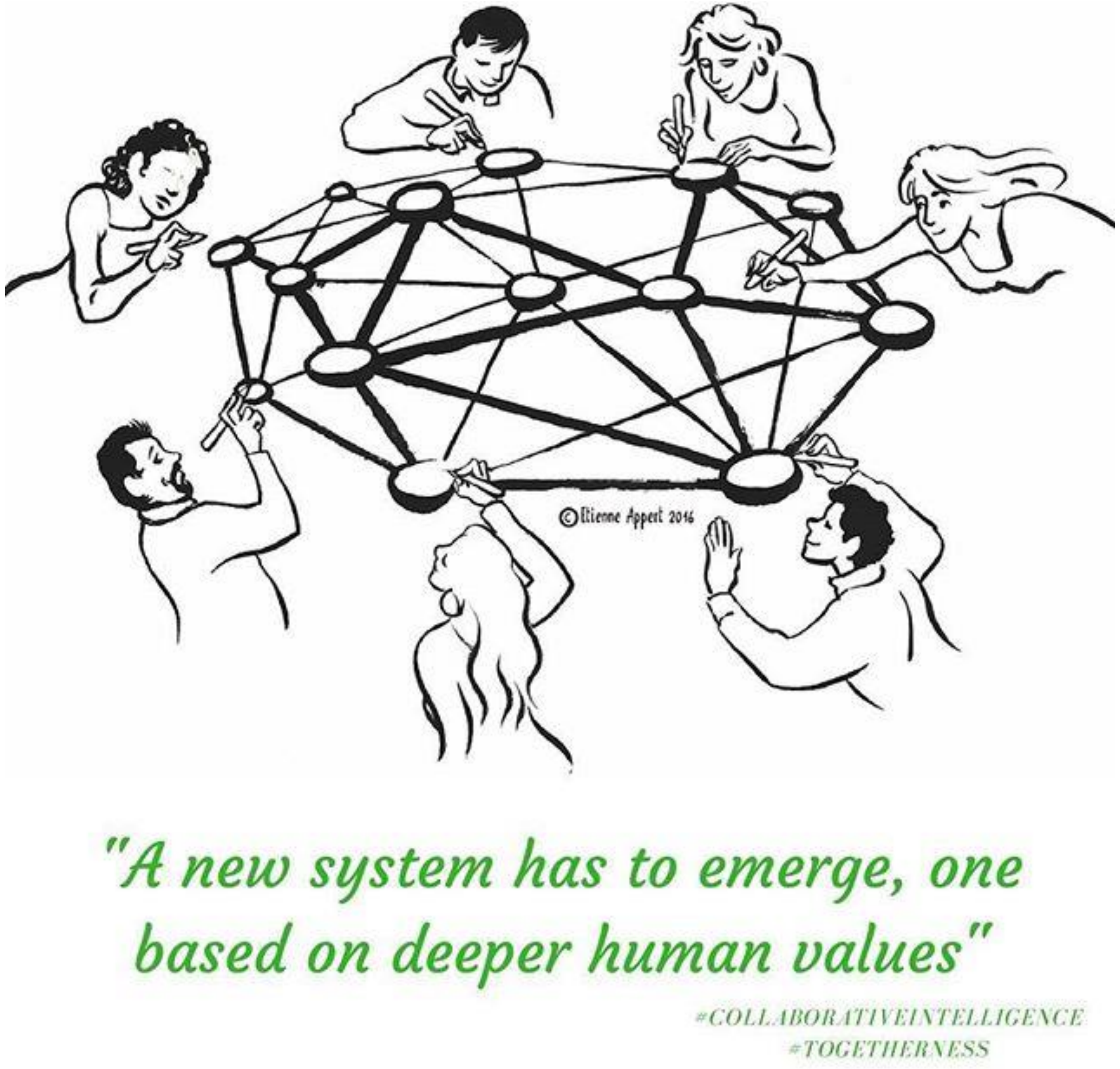
***In the shared-memory model, there is no communication. Neighbors directly access your data.***



# Walkthrough of 2D Stencil Code with Shared Memory Windows

- ***stencil_mpi_shmem.c***

# Advanced Topics: Nonblocking Collectives primer only



# 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 updated during operation
  - MPI_Cancel not supported
  - No matching with blocking collectives

*Hoefler et al.: Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI*

# 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*

*Hoefler et al.: Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI*

# A Non-Blocking Barrier?

- **What can that be good for? Well, quite a bit!**

- **Semantics:**
  - MPI_Ibarrier() – calling process entered the barrier, **no** synchronization happens
  - Synchronization **may** happen asynchronously
  - MPI_Test/Wait() – synchronization happens **if** necessary

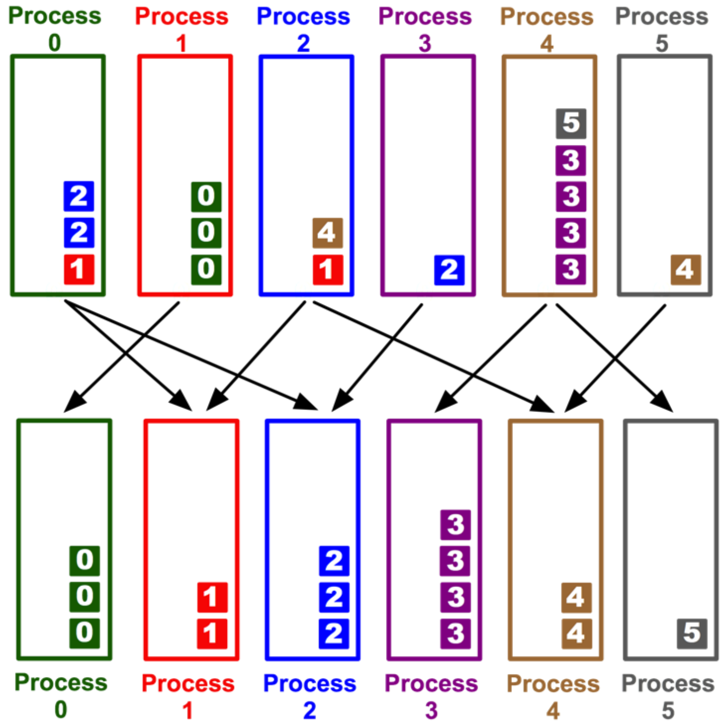
- **Uses:**
  - Overlap barrier latency (small benefit)
  - Use the split semantics! Processes **notify** non-collectively but **synchronize** collectively!

# A Semantics Example: DSDE

- **Dynamic Sparse Data Exchange**
  - Dynamic: comm. pattern varies across iterations
  - Sparse: number of neighbors is limited ($O(\log P)$)
  - Data exchange: only senders know neighbors
- **Main Problem: metadata**
  - Determine who wants to send how much data to me (I must post receive and reserve memory)

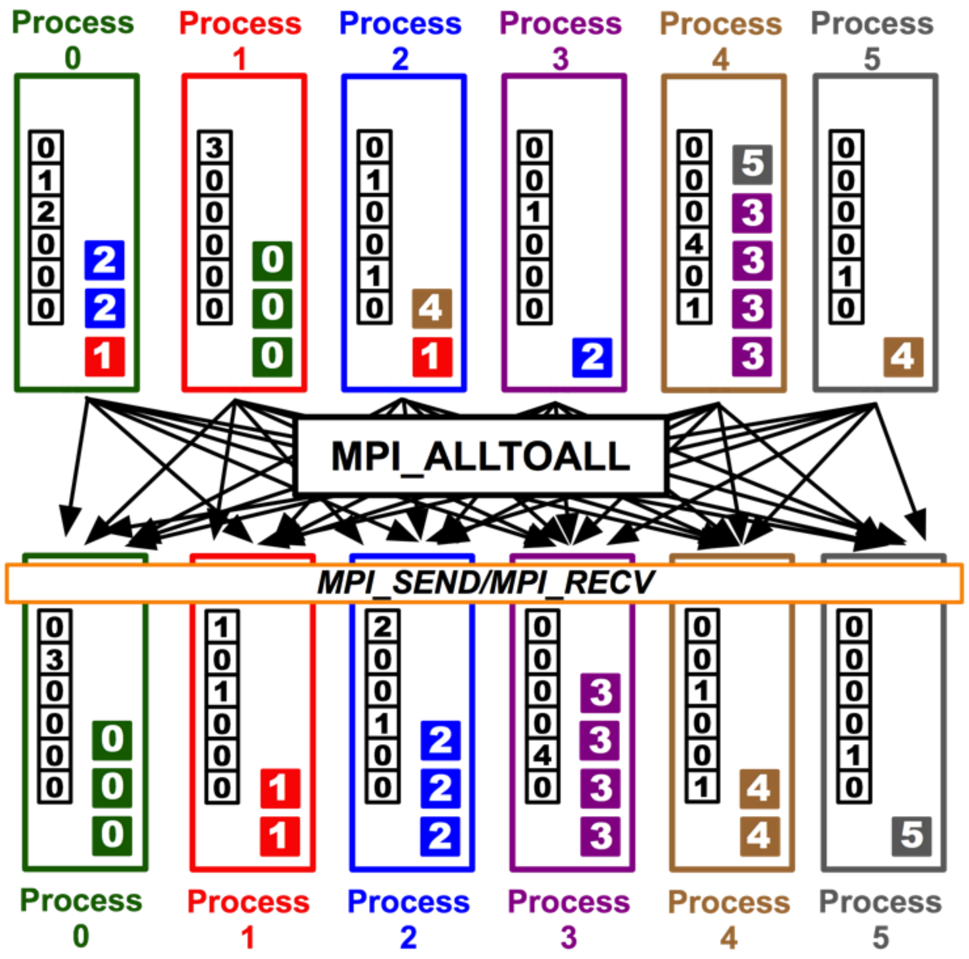
**-- OR --**

  - Use MPI semantics:
    *Unknown sender (MPI_ANY_SOURCE)*
    *Unknown message size (MPI_PROBE)*
    *Reduces problem to counting the number of neighbors*
    *Allow faster implementation!*



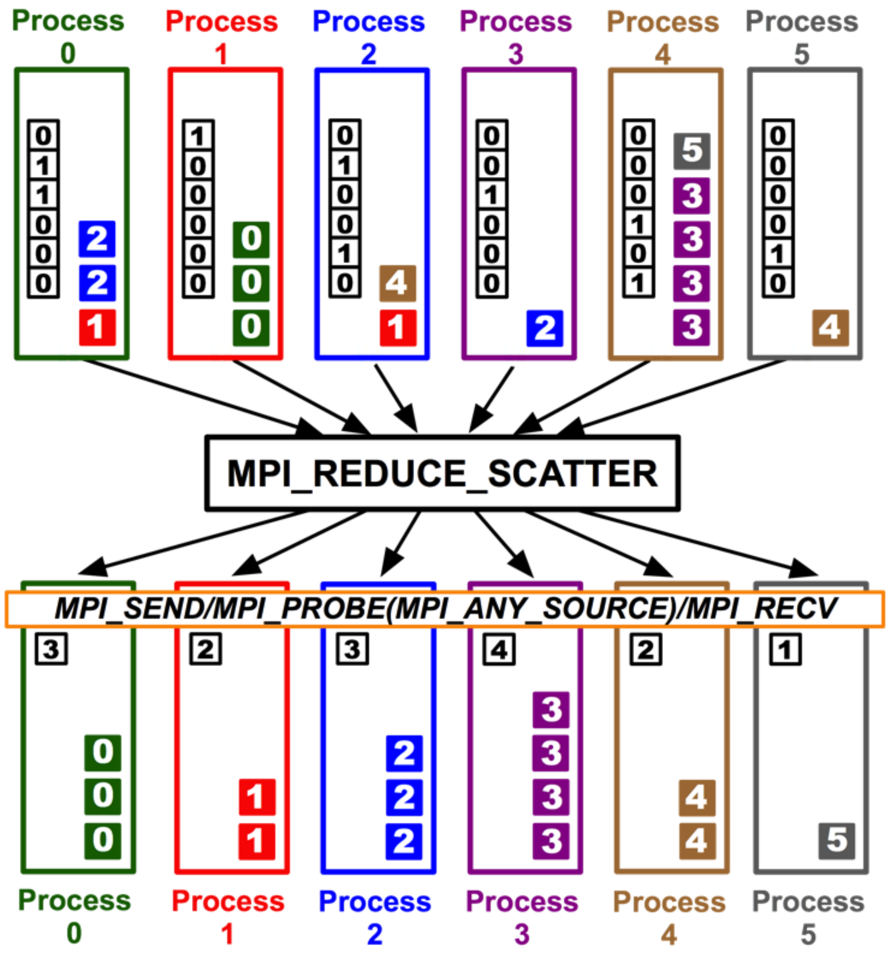
# Using Alltoall (PEX)

- **Based on Personalized Exchange ($\Theta(P)$)**
  - Processes exchange metadata (sizes) about neighborhoods with all-to-all
  - Processes post receives afterwards
  - Most intuitive but least performance and scalability!



# Reduce_scatter (PCX)

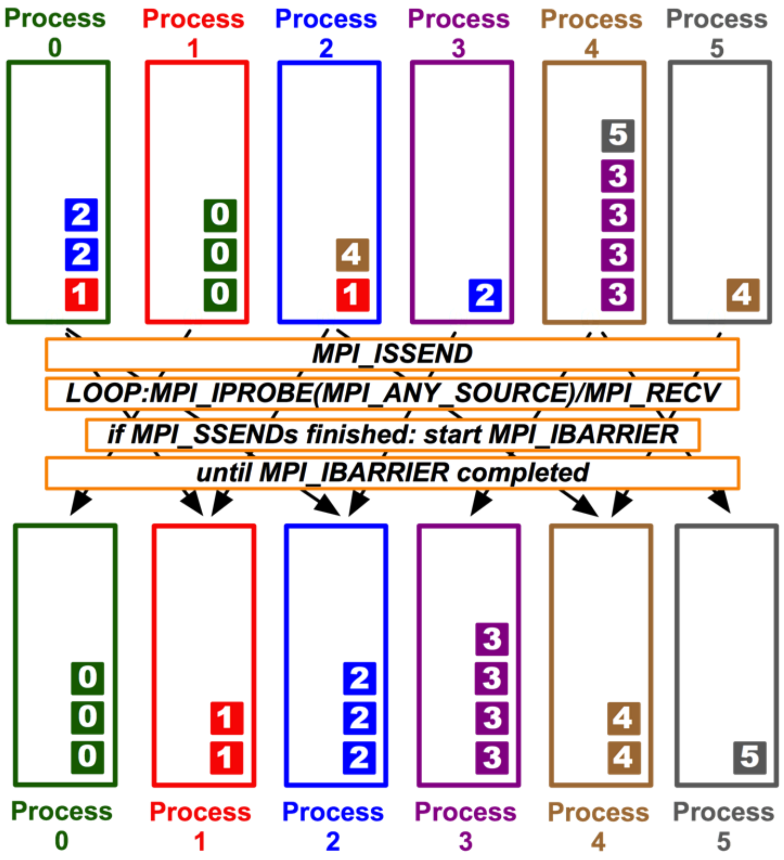
- **Bases on Personalized Census ($\Theta(P)$)**
  - Processes exchange metadata (counts) about neighborhoods with reduce_scatter
  - Receivers checks with wildcard MPI_IPROBE and receives messages
  - Better than PEX but non-deterministic!



# MPI_Ibarrier (NBX)

- **Complexity - census (barrier): $(\Theta(\log P))$**
  - Combines metadata with actual transmission
  - Point-to-point synchronization
  - Continue receiving until barrier completes
  - Processes start coll. synch. (barrier) when p2p phase ended

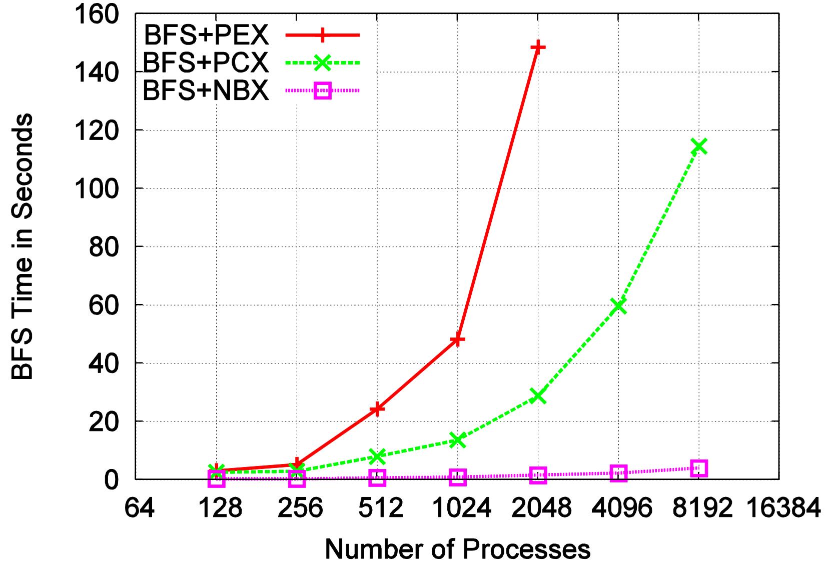
    *barrier = distributed marker!*
  - Better than Alltoall, reduce-scatter!



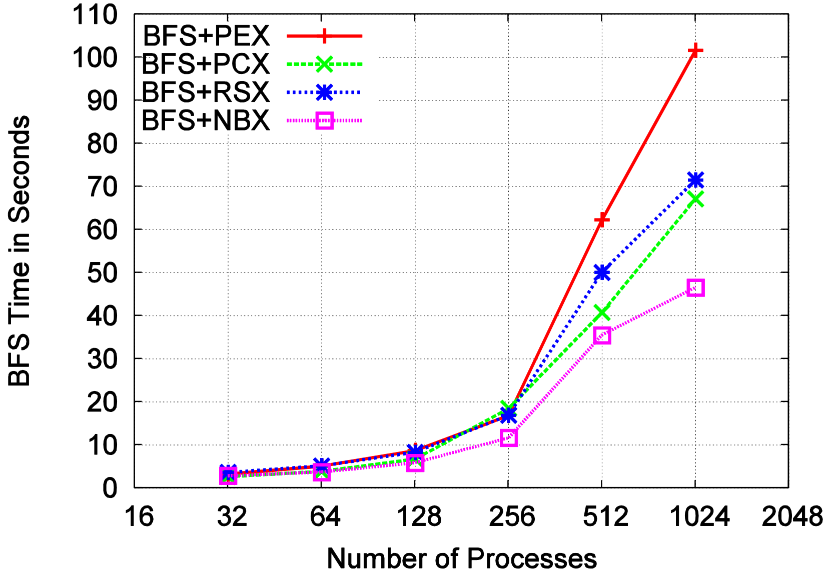
# Parallel Breadth First Search

- **On a clustered Erdős-Rényi graph, weak scaling**
  - 6.75 million edges per node (filled 1 GiB)

BlueGene/P – with HW barrier!




Myrinet 2000 with LibNBC




- **HW barrier support is significant at large scale!**

*T. Hoefler et al.: Scalable Communication Protocols for Dynamic Sparse Data Exchange*

# Nonblocking Collectives Summary

- **Nonblocking communication does two things:**
  - Overlap and relax synchronization
- **Collective communication does one thing**
  - Specialized pre-optimized routines
  - Performance portability
  - Hopefully transparent performance
- **They can be composed**
  - E.g., software pipelining