Advanced Parallel Programming with MPI-1, MPI-2, and MPI-3

Pavan Balaji

Computer Scientist

Argonne National Laboratory

Email: balaji@mcs.anl.gov

Web: http://www.mcs.anl.gov/~balaji

Torsten Hoefler

Assistant Professor

ETH Zurich

Email: htor@inf.ethz.ch

Web: http://www.unixer.de/

What is MPI?

- MPI: Message Passing Interface
  - The MPI Forum organized in 1992 with broad participation by:
    - Vendors: IBM, Intel, TMC, SGI, Convex, Meiko
    - Portability library writers: PVM, p4
    - Users: application scientists and library writers
    - MPI-1 finished in 18 months
  - Incorporates the best ideas in a “standard” way
    - Each function takes fixed arguments
    - Each function has fixed semantics
      - Standardizes what the MPI implementation provides and what the application can and cannot expect
      - Each system can implement it differently as long as the semantics match
- MPI is not…
  - a language or compiler specification
  - a specific implementation or product

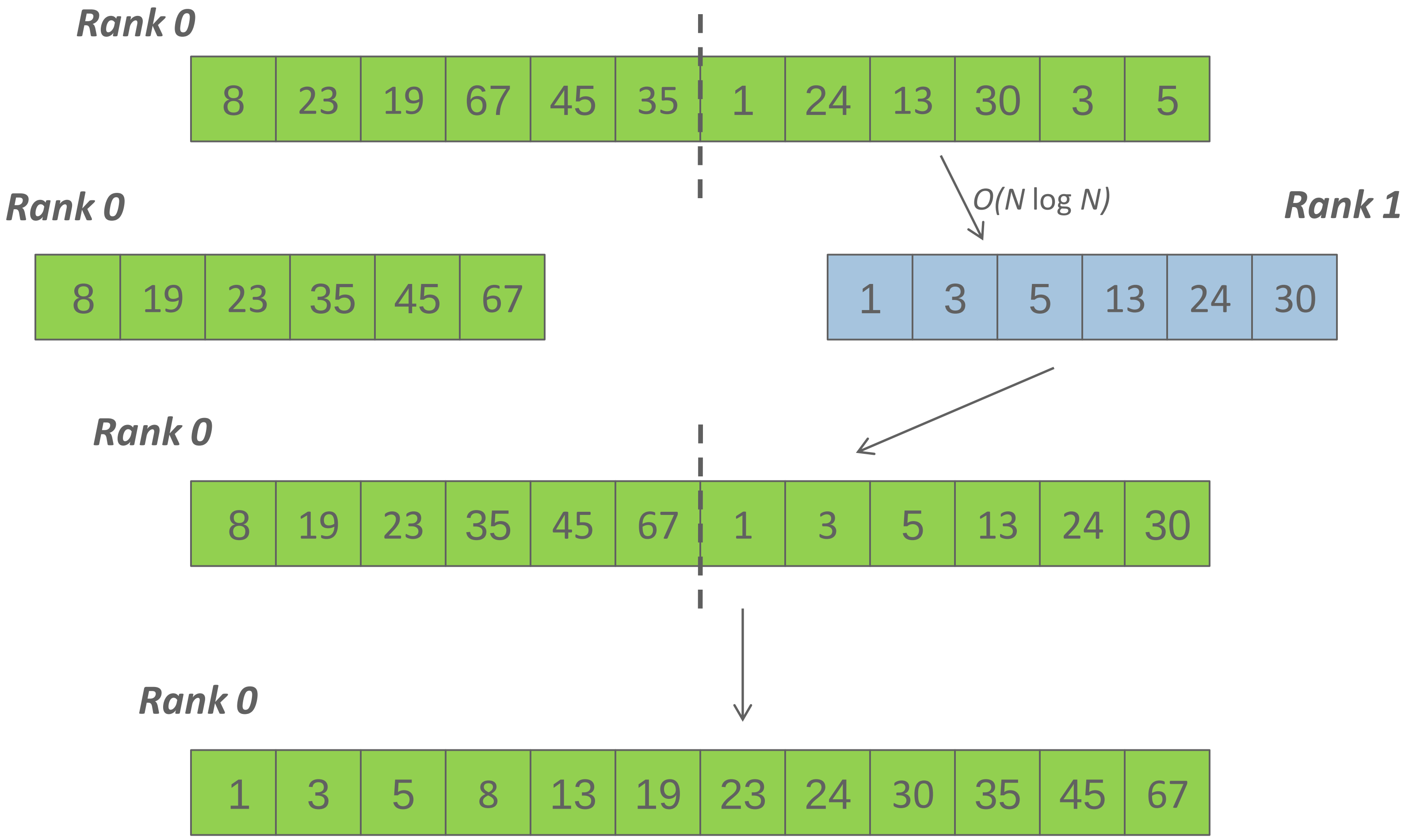
Following MPI Standards

- MPI-2 was released in 2000
  - Several additional features including MPI + threads, MPI-I/O, remote memory access functionality and many others
- MPI-2.1 (2008) and MPI-2.2 (2009) were recently released with some corrections to the standard and small features
- MPI-3 (2012) added several new features to MPI
- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- Other information on Web:
  - at http://www.mcs.anl.gov/mpi
  - pointers to lots of material including tutorials, a FAQ, other MPI pages

Important considerations while using MPI

- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs

Parallel Sort using MPI Send/Recv



Parallel Sort using MPI Send/Recv (contd.)

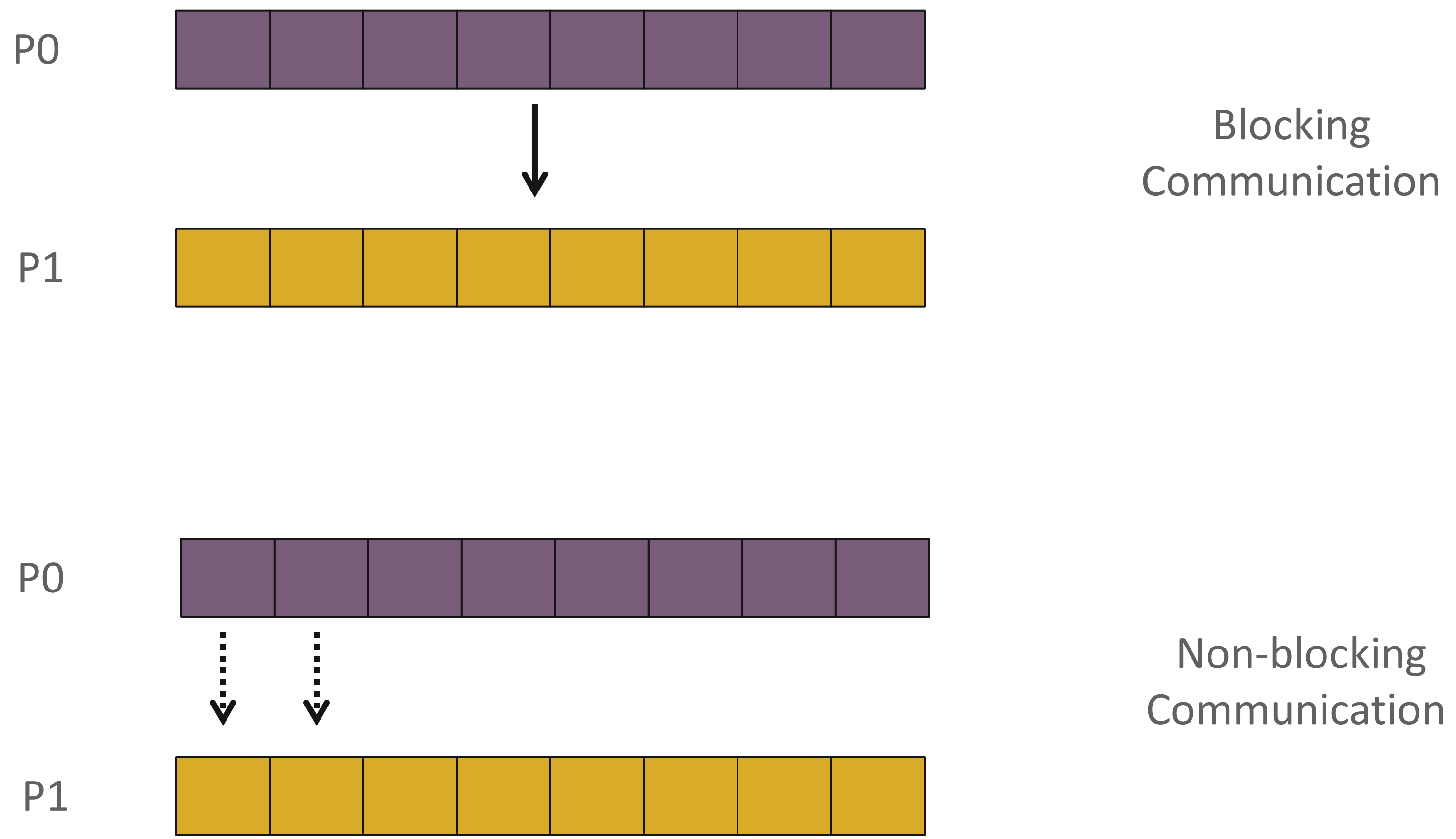
```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char ** argv)
{
    int rank;
    int a[1000], b[500];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        MPI_Send(&a[500], 500, MPI_INT, 1, 0, MPI_COMM_WORLD);
        sort(a, 500);
        MPI_Recv(b, 500, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);

        /* Serial: Merge array b and sorted part of array a */
    }
    else if (rank == 1) {
        MPI_Recv(b, 500, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        sort(b, 500);
        MPI_Send(b, 500, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize(); return 0;
}
```

A Non-Blocking communication example



A Non-Blocking communication example

```
int main(int argc, char ** argv)
{
    [...snip...]
    if (rank == 0) {
        for (i=0; i< 100; i++) {
            /* Compute each data element and send it out */
            data[i] = compute(i);
            MPI_ISend(&data[i], 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
                      &request[i]);
        }
        MPI_Waitall(100, request, MPI_STATUSES_IGNORE)
    }
    else {
        for (i = 0; i < 100; i++)
            MPI_Recv(&data[i], 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                     MPI_STATUS_IGNORE);
    }
    [...snip...]
}
```

MPI Collective Routines

- Many Routines: **MPI_ALLGATHER, MPI_ALLGATHERV, MPI_ALLREDUCE, MPI_ALLTOALL, MPI_ALLTOALLV, MPI_BCAST, MPI_GATHER, MPI_GATHERV, MPI_REDUCE, MPI_REDUCESCATTER, MPI_SCAN, MPI_SCATTER, MPI_SCATTERV**
- "**All**" versions deliver results to all participating processes
- "**V**" versions (stands for vector) allow the hunks to have different sizes
- **MPI_ALLREDUCE**, **MPI_REDUCE**, **MPI_REDUCESCATTER**, and **MPI_SCAN** take both built-in and user-defined combiner functions

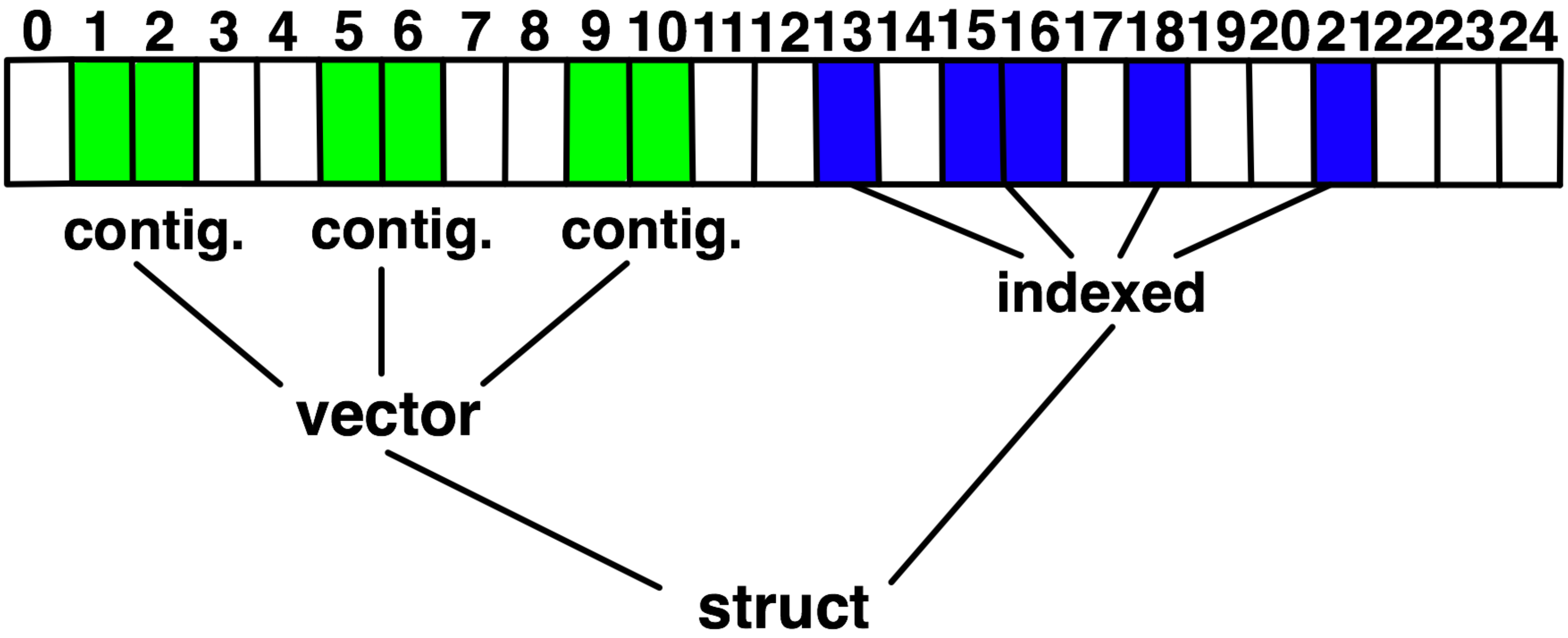
MPI Built-in Collective Computation Operations

`MPI_MAX`	Maximum
`MPI_MIN`	Minimum
`MPI_PROD`	Product
`MPI_SUM`	Sum
`MPI_LAND`	Logical and
`MPI_LOR`	Logical or
`MPI_LXOR`	Logical exclusive or
`MPI_BAND`	Bitwise and
`MPI_BOR`	Bitwise or
`MPI_BXOR`	Bitwise exclusive or
`MPI_MAXLOC`	Maximum and location
`MPI_MINLOC`	Minimum and location

Introduction to Datatypes in MPI

- Datatypes allow to (de)serialize **arbitrary** data layouts into a message stream
  - Networks provide serial channels
  - Same for block devices and I/O
- Several constructors allow arbitrary layouts
  - Recursive specification possible
  - *Declarative* specification of data-layout
    - “what” and not “how”, leaves optimization to implementation (*many unexplored* possibilities!)
  - Choosing the right constructors is not always simple

Derived Datatype Example

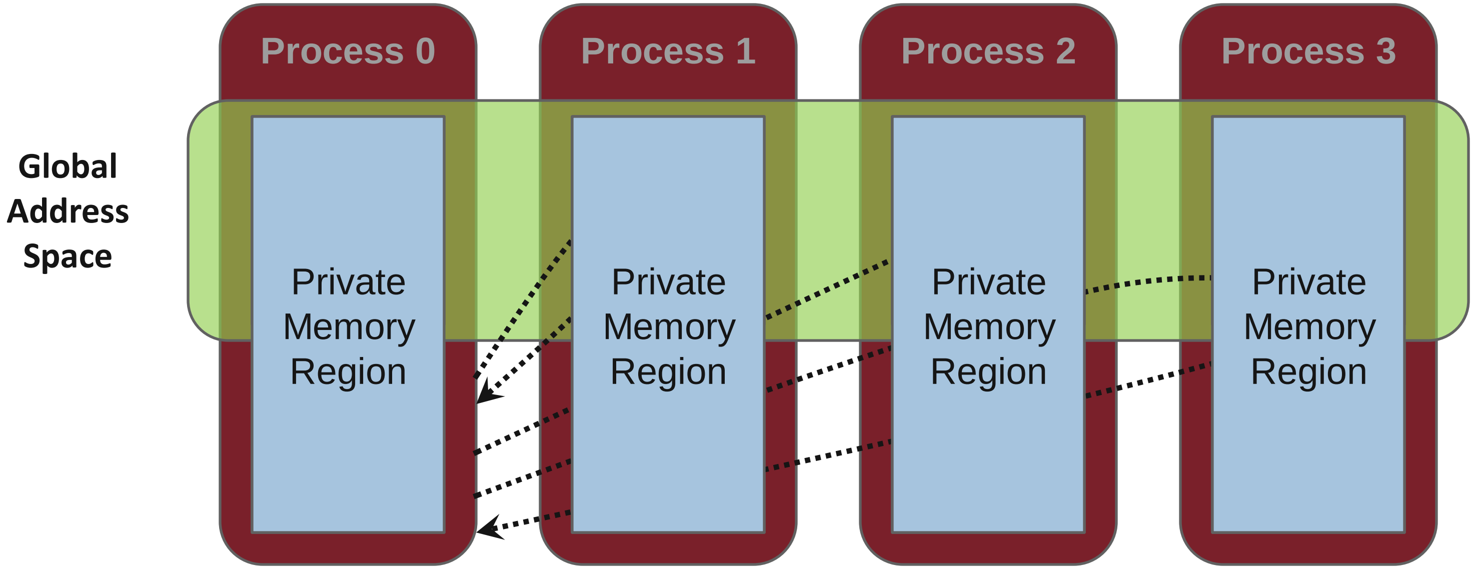


- Explain Lower Bound, Size, Extent

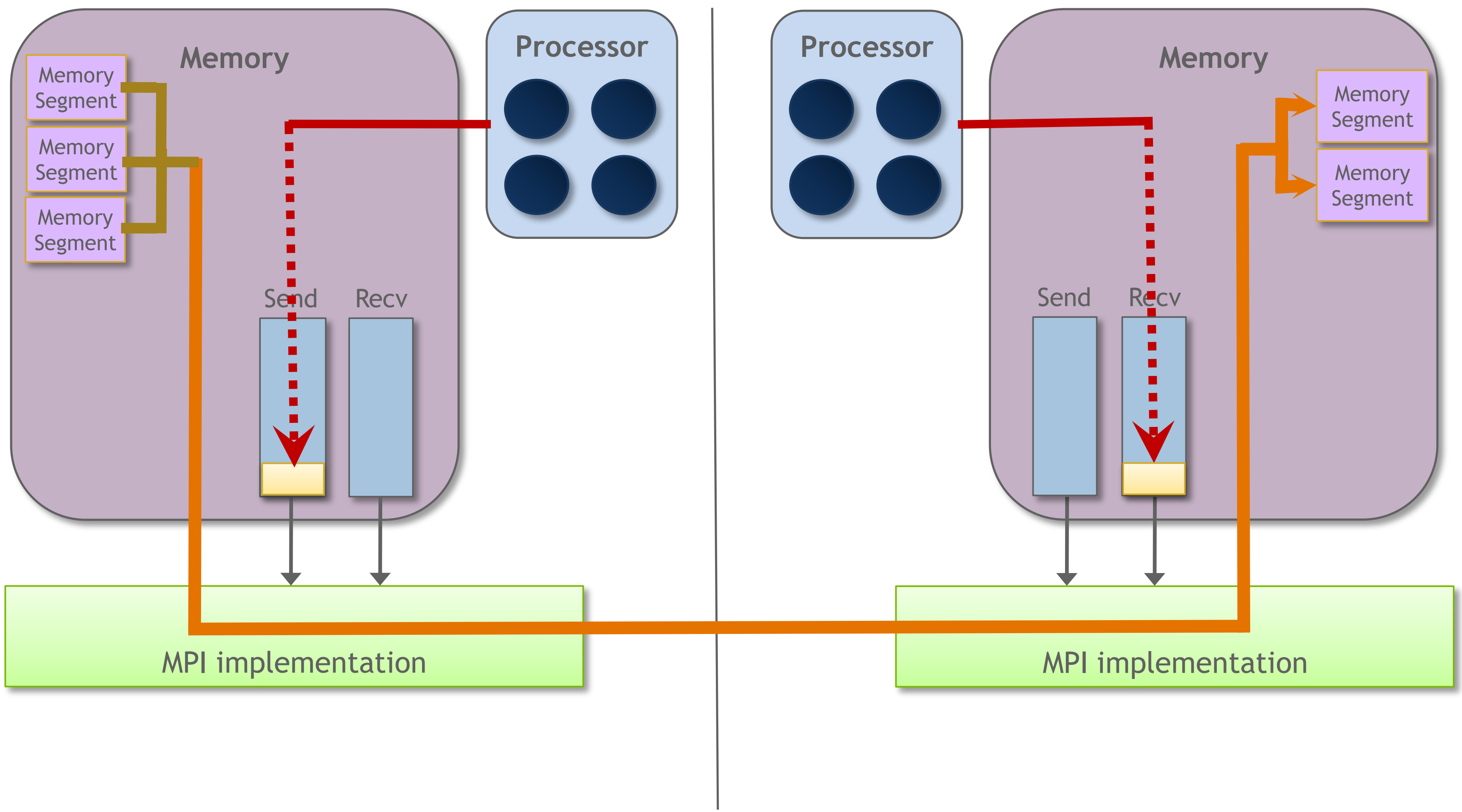
Advanced Topics: One-sided Communication

One-sided Communication

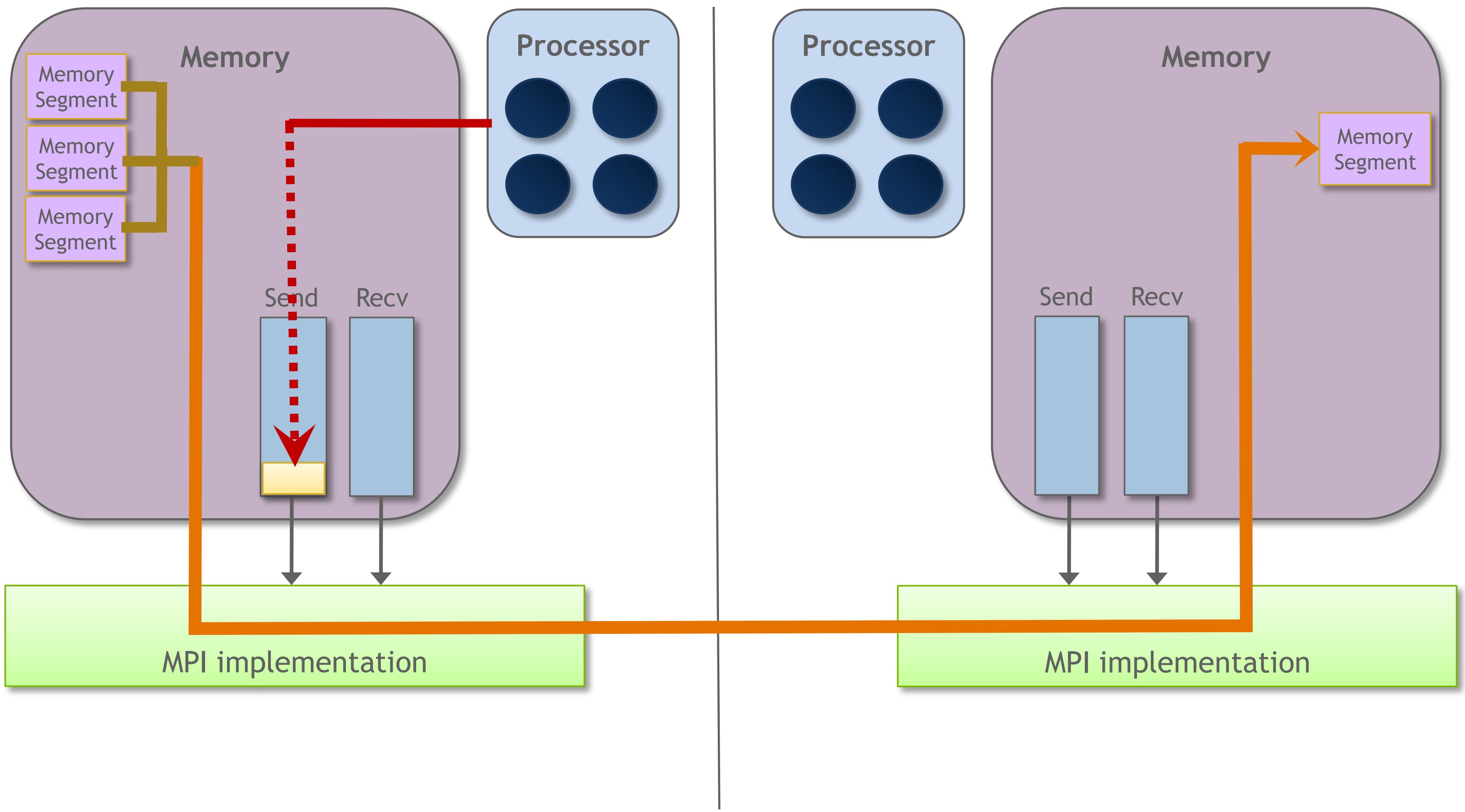
- The basic idea of one-sided communication models is to decouple data movement with process synchronization
  - Should be able move data without requiring that the remote process synchronize
  - Each process exposes a part of its memory to other processes
  - Other processes can directly read from or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



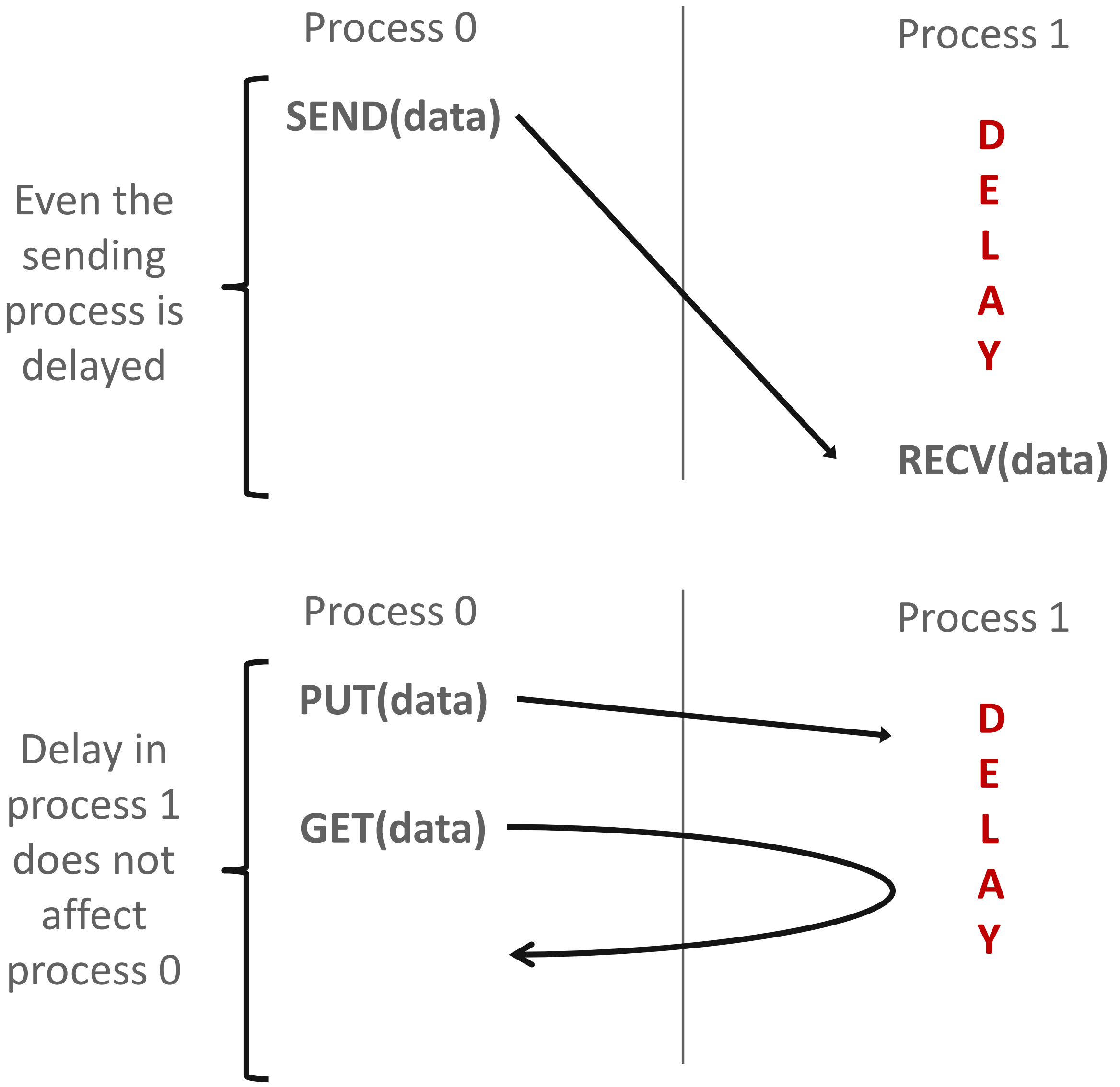
Two-sided Communication Example



One-sided Communication Example



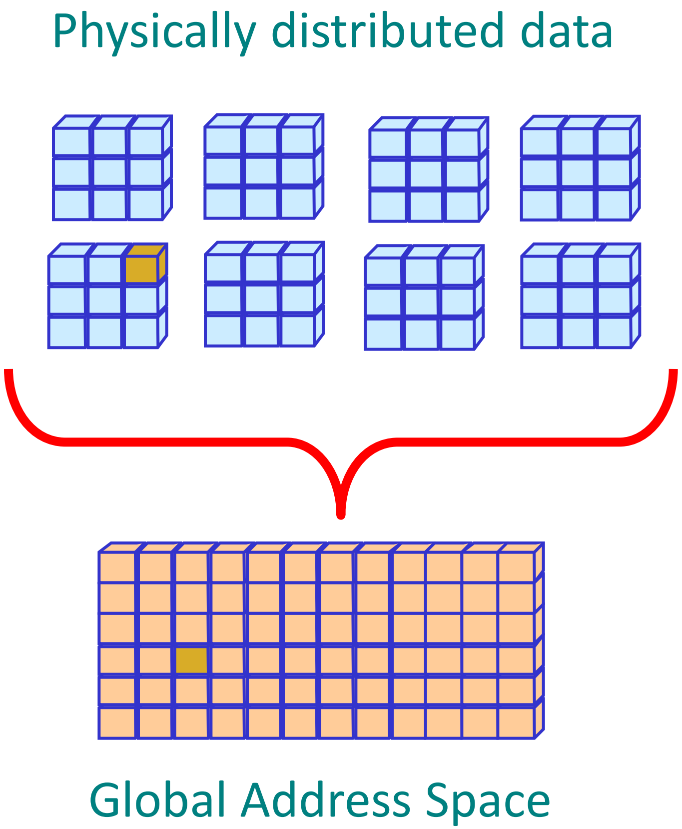
Comparing One-sided and Two-sided Programming



Possible Applications of One-sided Communication

- One-sided communication (or sometimes referred to as global address space communication) is very useful for many applications that require asynchronous access to remote memory
  - E.g., a nuclear physics application called as Greene's Function Monte Carlo requires to store nearly 50 GB of memory per task for its calculations
  - No single node can provide that much memory
  - With one-sided communication, each task can store this data in global space, and access it as needed
  - Note: Remember that the memory is still "far away" (accesses require data movement over the network); so large data transfers are better for performance

Globally Accessible Large Arrays



- Presents a shared view of physically distributed dense array objects over the nodes of a cluster
- Accesses are using one-sided communication model using Put/Get and Accumulate (or update) semantics
- Used in wide variety of applications
  - Computational Chemistry (e.g., NWChem, molcas, molpro)
  - Bioinformatics (e.g., ScalaBLAST)
  - Ground Water Modeling (e.g., STOMP)

Window Creation: Static Model

```
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)

Window Creation: Dynamic Model

```
int MPI_Win_create_dynamic(…, 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
- Application can dynamically attach memory to this window
- Application can access data on this window only after a memory region has been attached

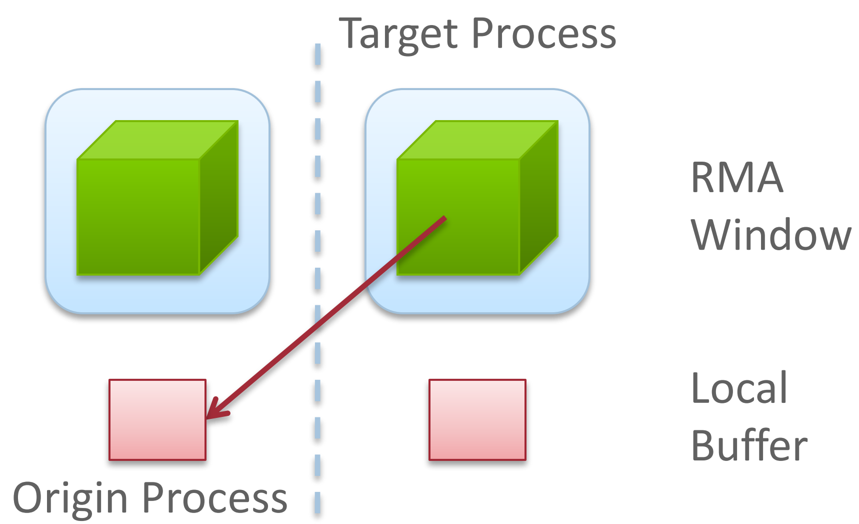
Data movement

- MPI_Get, MPI_Put, MPI_Accumulate, MPI_Get_accumulate, etc., move data between public copy of target window and origin local buffer
- **Nonblocking**, subsequent synchronization may block
- Origin buffer address
- Target buffer displacement
  - Displacement in units of the window's "disp_unit"
- Distinct from load/store from/to private copy

Data movement: *Get*

```
MPI_Get(
    origin_addr, origin_count, origin_datatype,
    target_rank,
    target_disp, target_count, target_datatype,
    win)
```

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



Data movement: *Put*

MPI_Put(

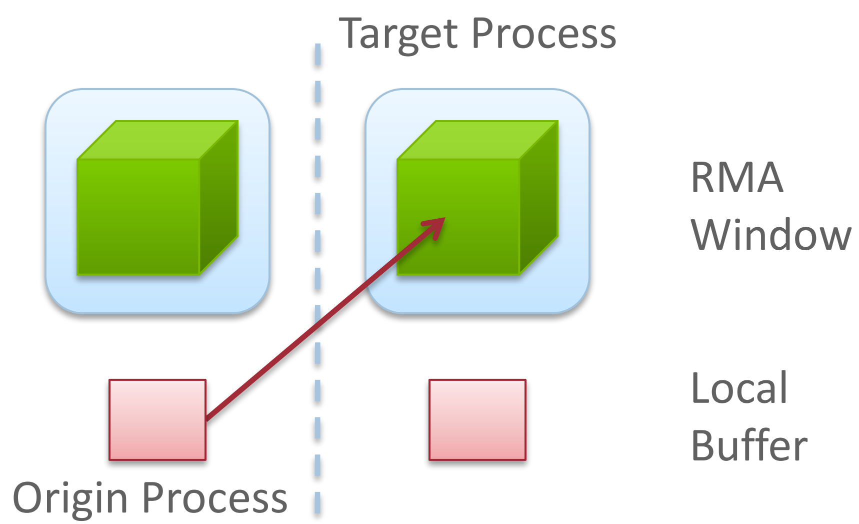
origin_addr, origin_count, origin_datatype,

target_rank,

target_disp, target_count, target_datatype,

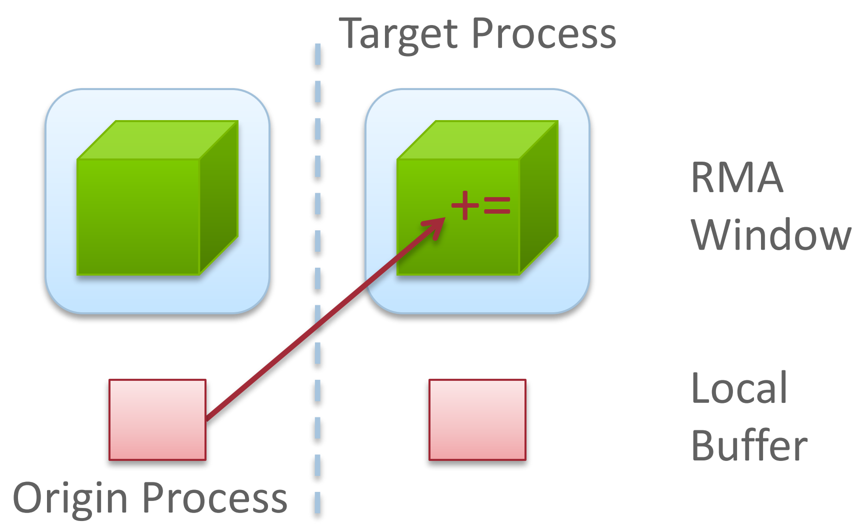
win)

- Move data from origin, to target
- Same arguments as MPI_Get



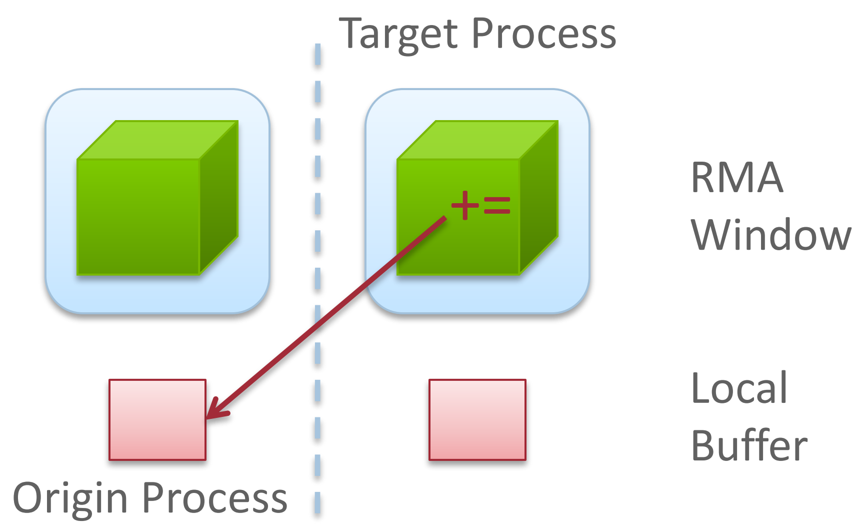
Data aggregation: *Accumulate*

- Like MPI_Put, but applies an MPI_Op instead
  - Predefined ops only, no user-defined!
- Result ends up at target buffer
- Different data layouts between target/origin OK, basic type elements must match
- Put-like behavior with MPI_REPLACE (implements *f(a,b)=b*)
  - Atomic PUT



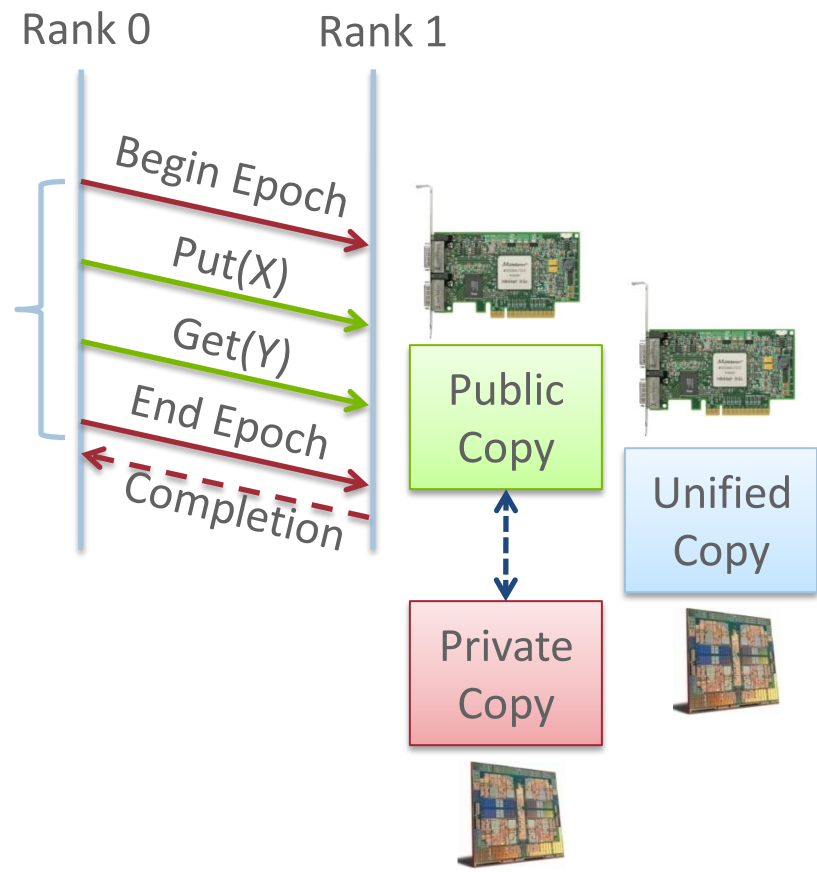
Data aggregation: *Get Accumulate*

- Like MPI_Get, but applies an MPI_Op instead
  - Predefined ops only, no user-defined!
- Result at target buffer; original data comes to the source
- Different data layouts between target/origin OK, basic type elements must match
- Get-like behavior with MPI_NO_OP
  - Atomic GET

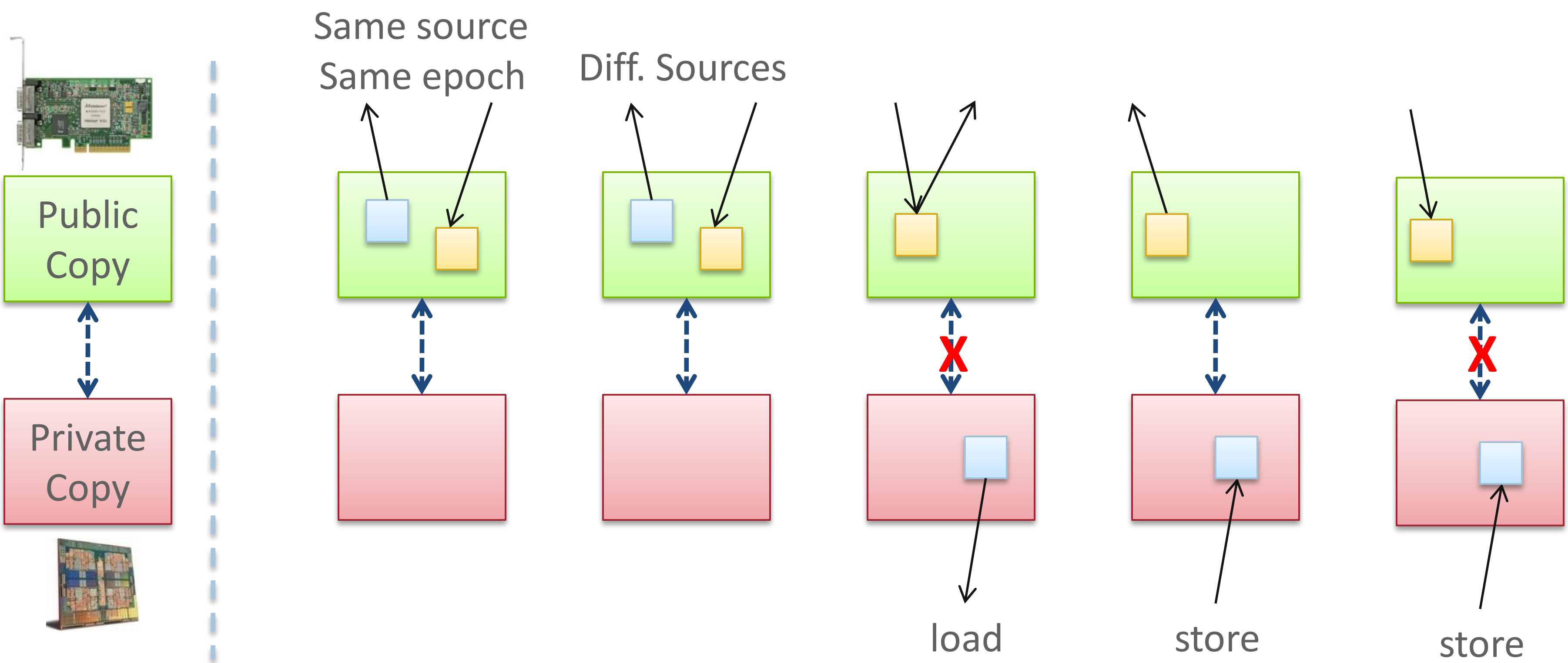


MPI RMA Memory Model

- Window: Expose memory for RMA
  - Logical public and private copies
  - Portable data consistency model
- Accesses must occur within an epoch
- Active and Passive synchronization modes
  - Active: target participates
  - Passive: target does not participate

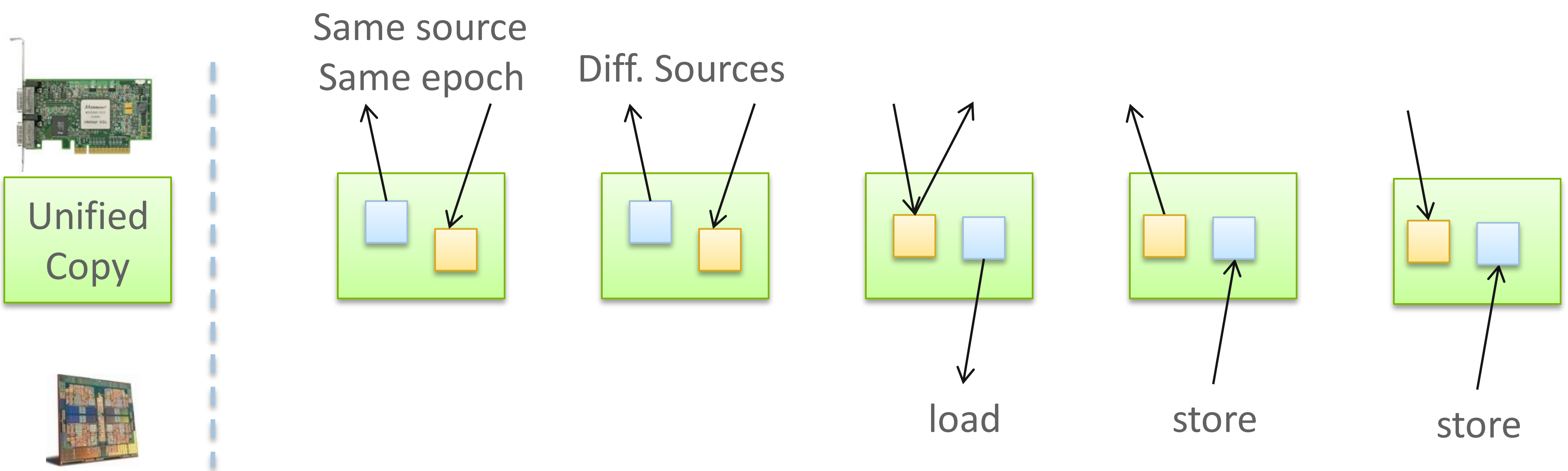


MPI RMA Memory Model (separate windows)



- Compatible with non-coherent memory systems

MPI RMA Memory Model (unified windows)



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

Ordering of Operations in MPI RMA

- For Put/Get operations, ordering does not matter
  - If you do two PUTs to the same location, the resultant can be garbage
- Two accumulate operations to the same location are valid
  - If you want “atomic PUTs”, you can do accumulates with MPI_REPLACE
- In MPI-2, there was no ordering of operations
- In MPI-3, all accumulate operations are ordered by default
  - User can tell the MPI implementation that (s)he does not require ordering as optimization hints
  - You can ask for “read-after-write” ordering, “write-after-write” ordering, or “read-after-read” ordering

Additional Atomic Operations

- Compare-and-swap
  - Compare the target value with an input value; if they are the same, replace the target with some other value
  - Useful for linked list creations – if next pointer is NULL, do something
- Get Accumulate
  - Fetch the value at the target location before applying the accumulate operation
  - "Fetch-and-Op" style operation
- Fetch-and-Op
  - Special case of Get accumulate for predefined datatypes – faster for the hardware to implement

Other MPI-3 RMA features

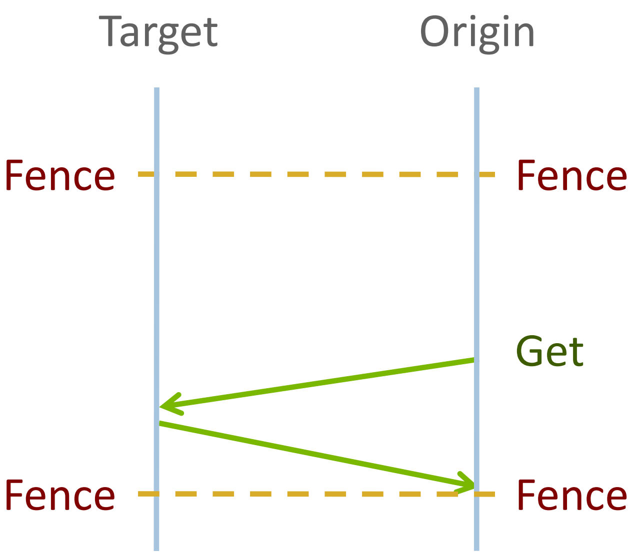
- Request based RMA operations
  - Can wait for single requests
  - Issue a large number of operations and wait for some of them to finish so you can reuse buffers
- Flush
  - Can wait for RMA operations to complete without closing an epoch
  - Lock; put; put; flush; get; get; put; Unlock
- Sync
  - Synchronize public and private memory

RMA Synchronization Models

- Three models
  - Fence (active target)
  - Post-start-complete-wait (active target)
  - Lock/Unlock (passive target)

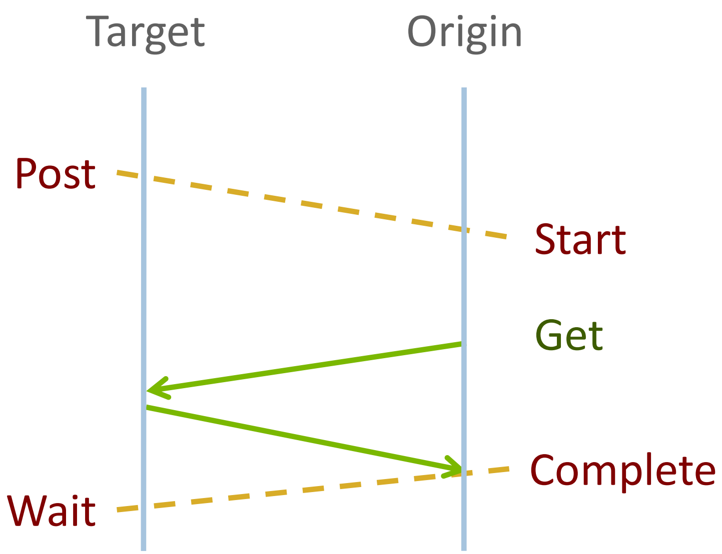
Fence Synchronization

- MPI_Win_fence(assert, win)
- Collective, assume it synchronizes like a barrier
- Starts *and* ends access & exposure epochs (usually)

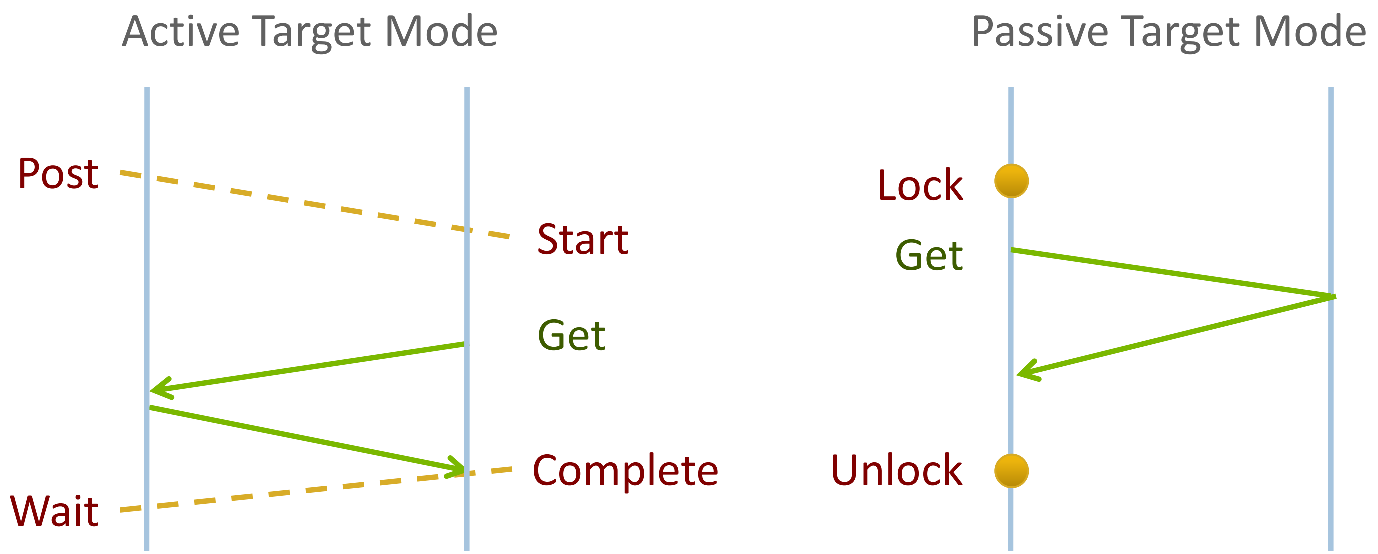


PSCW Synchronization

- 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_compete
- All may block, to enforce P-S/C-W ordering
  - Processes can be both origins and targets



Lock/Unlock Synchronization



- Passive mode: One-sided, *asynchronous* communication
  - Target does **not** participate in communication operation
- Erroneous to combine active and passive modes

Passive Target Synchronization

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

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

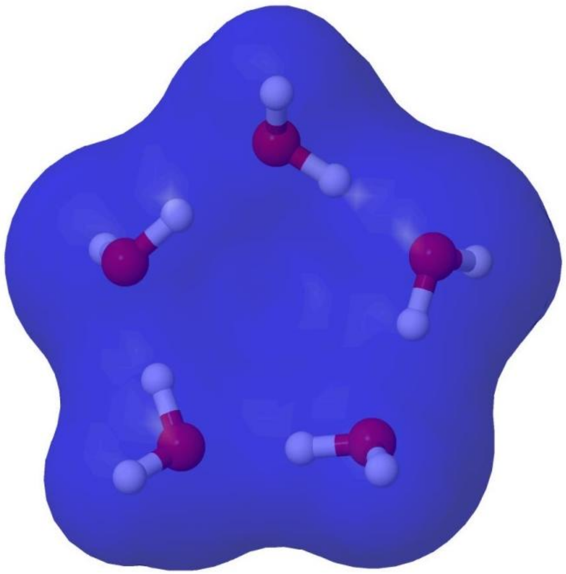
- Begin/end passive mode epoch
  - Doesn't function like a mutex, name can be confusing
  - Communication operations within epoch are all nonblocking
- Lock type
  - SHARED: Other processes using shared can access concurrently
  - EXCLUSIVE: No other processes can access concurrently

When should I use passive mode?

- RMA performance advantages from low protocol overheads
  - Two-sided: Matching, queueing, buffering, unexpected receives, etc…
  - Direct support from high-speed interconnects (e.g. InfiniBand)
- Passive mode: *asynchronous* one-sided communication
  - Data characteristics:
    - Big data analysis requiring memory aggregation
    - Asynchronous data exchange
    - Data-dependent access pattern
  - Computation characteristics:
    - Adaptive methods (e.g. AMR, MADNESS)
    - Asynchronous dynamic load balancing
- Common structure: shared arrays

Use Case: Distributed Shared Arrays

- Quantum Monte Carlo: Ensemble data
  - Represents initial quantum state
  - Spline representation, cubic basis functions
  - Large(100+ GB), read-only table of coeff.
  - Accesses are random
- Coupled cluster simulations
  - Evolving quantum state of the system
  - Very large, tables of coefficients
  - Table_t read-only, Table_{t+1} accumulate-only
  - Accesses are non-local/overlapping
- Global Arrays PGAS programming model
  - Can be supported with passive mode RMA [Dinan et al., IPDPS'12]



Advanced Topics: Hybrid Programming with Threads and Shared Memory

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

- Almost all chips are multicore these days
- Today's clusters often comprise multiple CPUs per node sharing memory, and the nodes themselves are connected by a network
- Common options for programming such 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"

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)
- MPI defines an alternative to MPI_Init
  - MPI_Init_thread(requested, provided)
    - *Application indicates what level it needs; MPI implementation returns the level it supports*

MPI+OpenMP

- MPI_THREAD_SINGLE
  - There is no OpenMP multithreading in the program.
- MPI_THREAD_FUNNELED
  - All of the MPI calls are made by the master thread. i.e. all MPI calls are
    - *Outside OpenMP parallel regions, or*
    - *Inside OpenMP master regions, or*
    - *Guarded by call to MPI_Is_thread_main MPI call.*
      - (same thread that called MPI_Init_thread)
- MPI_THREAD_SERIALIZED

```
#pragma omp parallel
…
#pragma omp critical
{
  …MPI calls allowed here…
}
```

- MPI_THREAD_MULTIPLE
  - Any thread may make an MPI call at any time

Specification of MPI_THREAD_MULTIPLE

- When multiple threads make MPI calls concurrently, the outcome will be as if the calls executed sequentially in some (any) order
- Blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions
- 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
- 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

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)*

An Incorrect Program

	Process 0	Process 1
Thread 1	MPI_Bcast(comm)	MPI_Bcast(comm)
Thread 2	MPI_Barrier(comm)	MPI_Barrier(comm)

- Here the user must use some kind of synchronization to ensure that either thread 1 or thread 2 gets scheduled first on both processes
- Otherwise a broadcast may get matched with a barrier on the same communicator, which is not allowed in MPI

A Correct Example

	Process 0	Process 1
Thread 1	MPI_Recv(src=1)	MPI_Recv(src=0)
Thread 2	MPI_Send(dst=1)	MPI_Send(dst=0)

- An implementation must ensure that the above example never deadlocks for any ordering of thread execution
- That means the implementation cannot simply acquire a thread lock and block within an MPI function. It must release the lock to allow other threads to make progress.

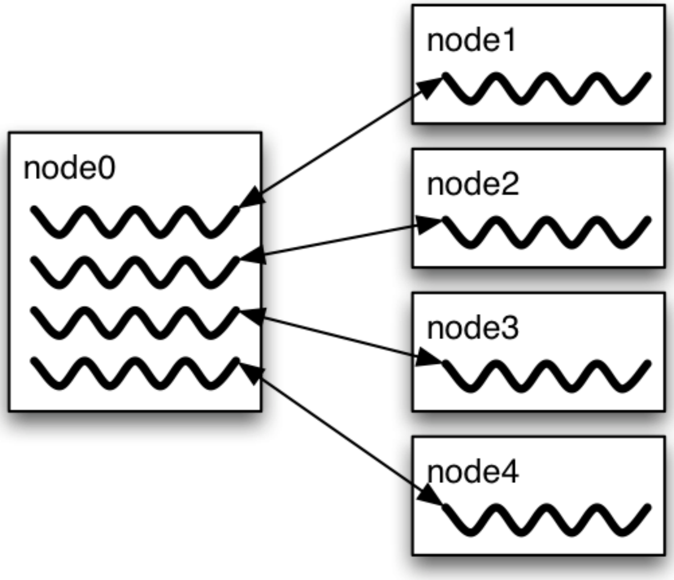
The Current Situation

- 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
- 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!

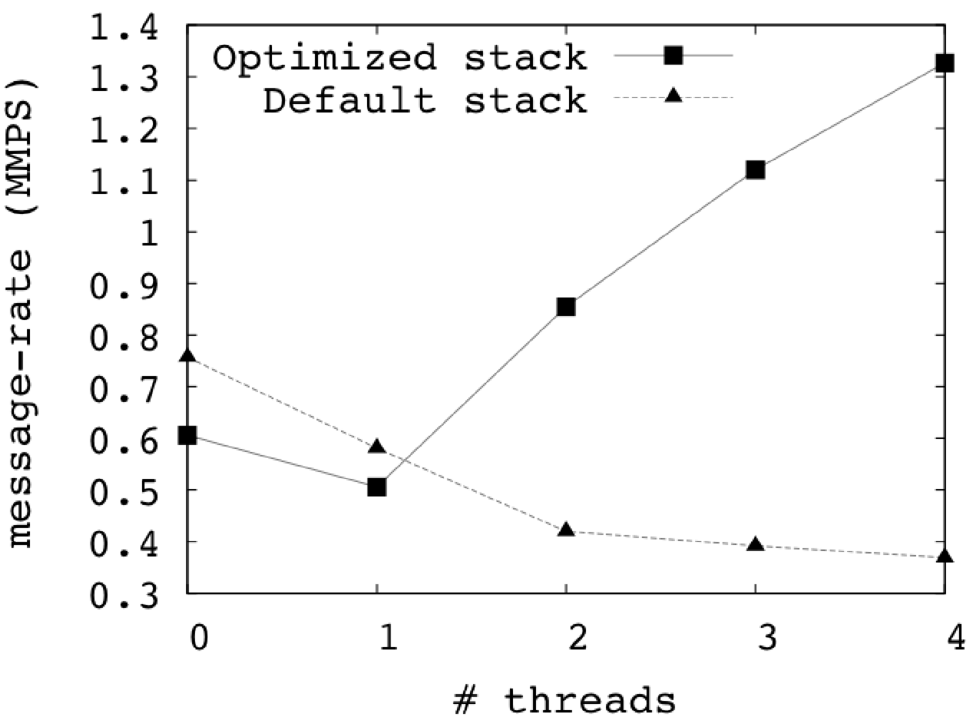
Performance with MPI_THREAD_MULTIPLE

- 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
  - Details in our *Parallel Computing* (journal) paper (2009)

Message Rate Results on BG/P



Message Rate Benchmark



Why is it hard to optimize MPI_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
- In MPI-3.1 (next version of the standard), we plan to add additional features to allow the user to provide hints (e.g., requests posted to this communicator are not shared with other threads)

Thread Programming is Hard

- *"The Problem with Threads,"* IEEE Computer
  - Prof. Ed Lee, UC Berkeley
  - http://ptolemy.eecs.berkeley.edu/publications/papers/06/problemwithThreads/
- *"Why Threads are a Bad Idea (for most purposes)"*
  - John Ousterhout
  - http://home.pacbell.net/ouster/threads.pdf
- *"Night of the Living Threads"*
  http://weblogs.mozillazine.org/roc/archives/2005/12/night_of_the_living_threads.html
- Too hard to know whether code is correct
- Too hard to debug
  - I would rather debug an MPI program than a threads program

Ptolemy and Threads

- Ptolemy is a framework for modeling, simulation, and design of concurrent, real-time, embedded systems
- Developed at UC Berkeley (PI: Ed Lee)
- It is a rigorously tested, widely used piece of software
- Ptolemy II was first released in 2000
- Yet, on April 26, 2004, four years after it was first released, the code deadlocked!
- The bug was lurking for 4 years of widespread use and testing!
- A faster machine or something that changed the timing caught the bug

An Example I encountered recently

- We received a bug report about a very simple multithreaded MPI program that hangs
- Run with 2 processes
- Each process has 2 threads
- Both threads communicate with threads on the other process as shown in the next slide
- I spent several hours trying to debug MPICH2 before discovering that the bug is actually in the user's program ☹

2 Proceses, 2 Threads, Each Thread Executes this Code

```
for (j = 0; j < 2; j++) {
    if (rank == 1) {
        for (i = 0; i < 3; i++)
            MPI_Send(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
        for (i = 0; i < 3; i++)
            MPI_Recv(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &stat);
    }
    else {  /* rank == 0 */
        for (i = 0; i < 3; i++)
            MPI_Recv(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD, &stat);
        for (i = 0; i < 3; i++)
            MPI_Send(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
    }
}
```

What Happened

	Rank 0	Rank 1
Thread 1	3 recvs 3 sends - - - - - - - - 3 recvs ← 3 sends	3 sends 3 recvs ← - - - - - - - - 3 sends 3 recvs
Thread 2	3 recvs 3 sends - - - - - - - - 3 recvs ← 3 sends	3 sends 3 recvs - - - - - - - - 3 sends 3 recvs ←

- All 4 threads stuck in receives because the sends from one iteration got matched with receives from the next iteration
- Solution: Use iteration number as tag in the messages

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
- Much simpler to program than threads

Advanced Topics: Nonblocking Collectives

Nonblocking Collective Communication

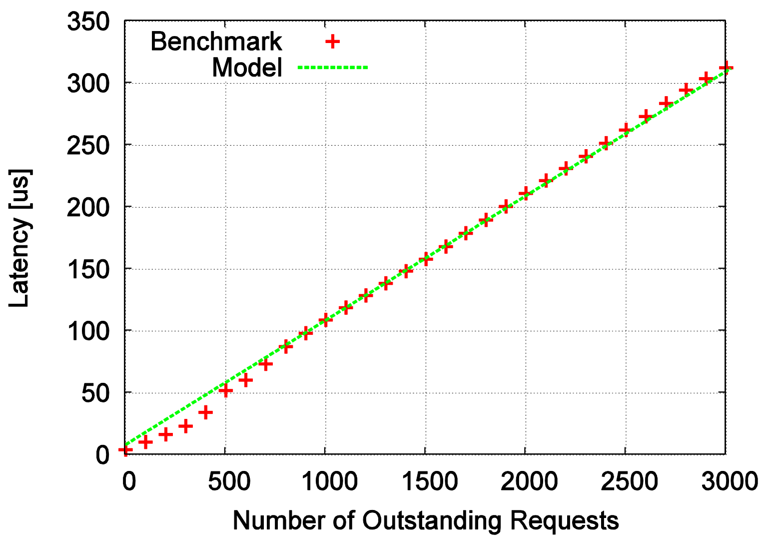
- Nonblocking communication
  - Deadlock avoidance
  - Overlapping communication/computation
- Collective communication
  - Collection of pre-defined optimized routines
- Nonblocking collective communication
  - Combines both advantages
  - System noise/imbalance resiliency
  - Semantic advantages
  - Examples

Nonblocking Communication

- Semantics are simple:
  - Function returns no matter what
  - No progress guarantee!
- E.g., MPI_Isend(<send-args>, MPI_Request *req);
- Nonblocking tests:
  - Test, Testany, Testall, Testsome
- Blocking wait:
  - Wait, Waitany, Waitall, Waitsome

Nonblocking Communication

- Blocking vs. nonblocking communication
  - Mostly equivalent, nonblocking has constant request management overhead
  - Nonblocking may have other non-trivial overheads
- Request queue length
  - Linear impact on performance
  - E.g., BG/P: 100ns/req
    - Tune unexpected Q length!



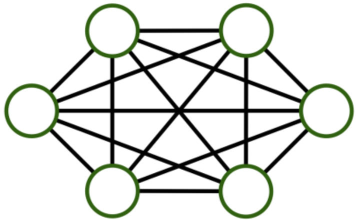
Collective Communication

- Three types:
  - Synchronization (Barrier)
  - Data Movement (Scatter, Gather, Alltoall, Allgather)
  - Reductions (Reduce, Allreduce, (Ex)Scan, Red_scat)
- Common semantics:
  - no tags (communicators can serve as such)
  - Blocking semantics (return when complete)
  - Not necessarily synchronizing (only barrier and all*)
- Overview of functions and performance models

Collective Communication

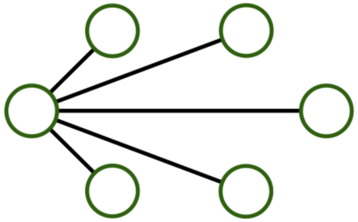
- Barrier –
  - Often $\alpha+\beta \log_2 P$

$$\Theta(\log(P))$$



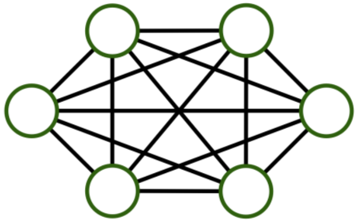
- Scatter, Gather –
  - Often $\alpha P+\beta Ps$

$$\Omega(\log(P) + Ps)$$



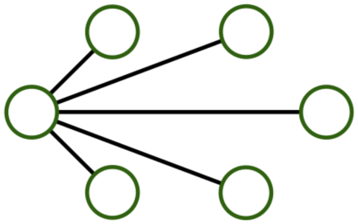
- Alltoall, Allgather -
  - Often $\alpha P+\beta Ps$

$$\Omega(\log(P) + Ps)$$

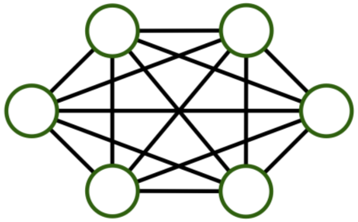


Collective Communication

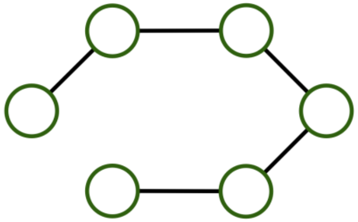
- Reduce – $\Omega(\log(P) + s)$
  - Often $\alpha \log_2 P + \beta m + \gamma m$



- Allreduce – $\Omega(\log(P) + s)$
  - Often $\alpha \log_2 P + \beta m + \gamma m$



- (Ex)scan – $\Omega(\log(P) + s)$
  - Often $\alpha P + \beta m + \gamma m$



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

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

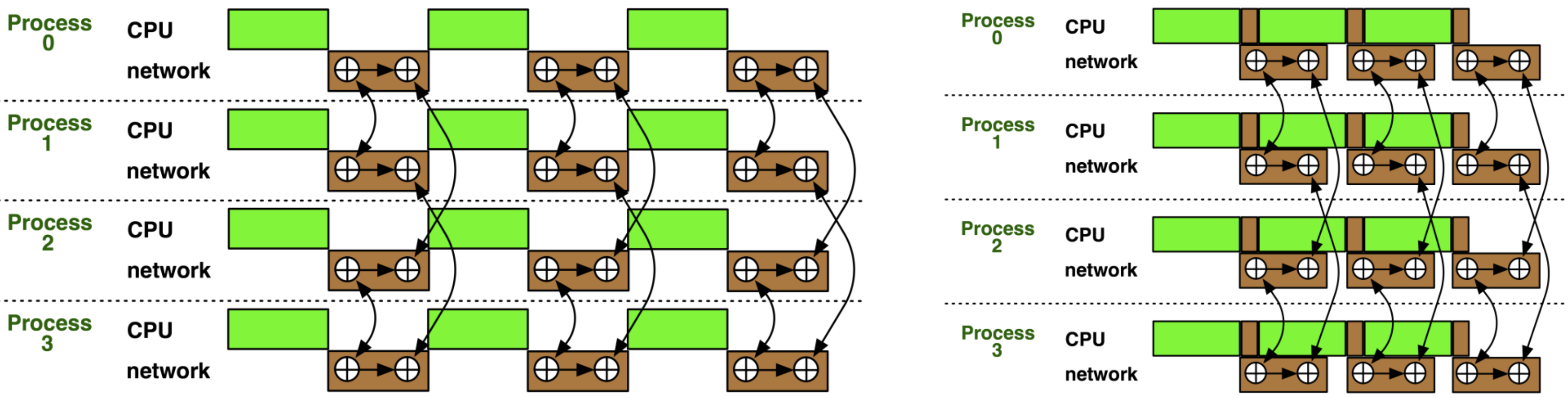
Nonblocking Collective Communication

- Semantic advantages:
  - Enable asynchronous progression (and manual)
    - Software pipelinling
  - 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

Nonblocking Collectives Overlap

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



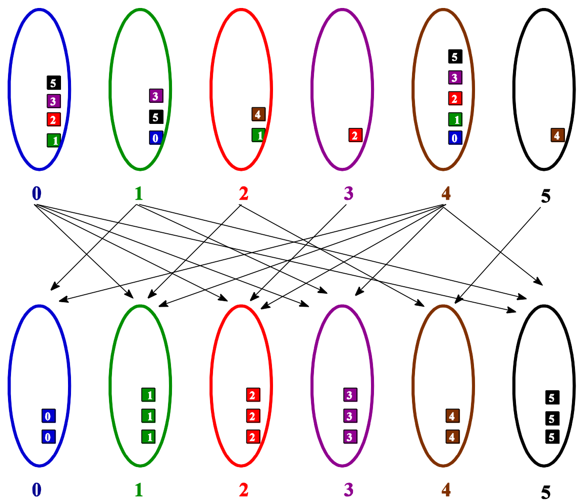
Hoefler: Leveraging Non-blocking Collective Communication in High-performance Applications

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 ($\mathcal{O}(\log P)$)
  - Data exchange: only senders know neighbors

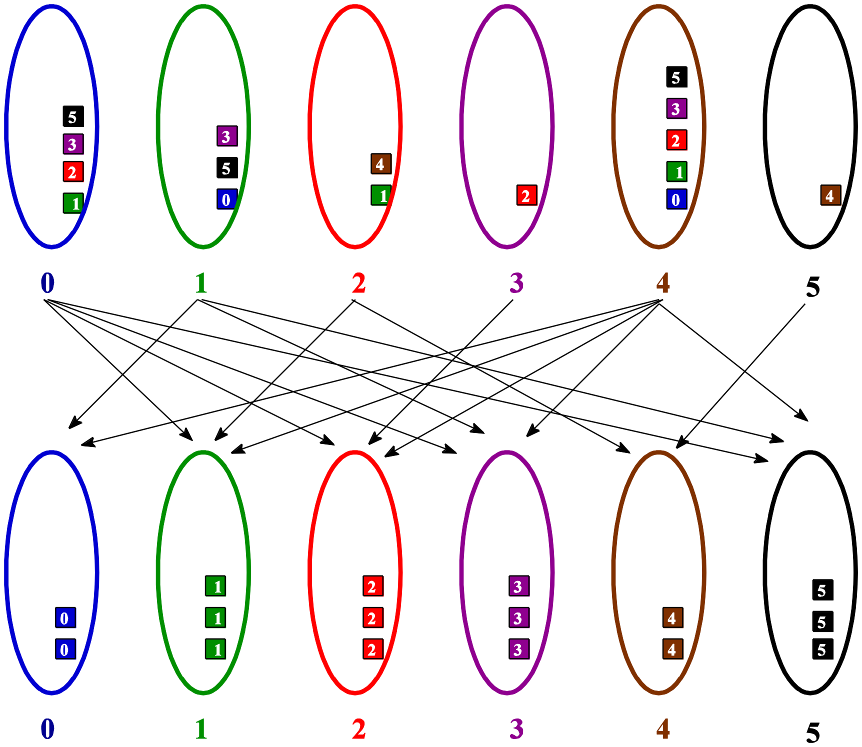


Dynamic Sparse Data Exchange (DSDE)

- Main Problem: metadata
  - Determine who wants to send how much data to me (I must post receive and reserve memory)

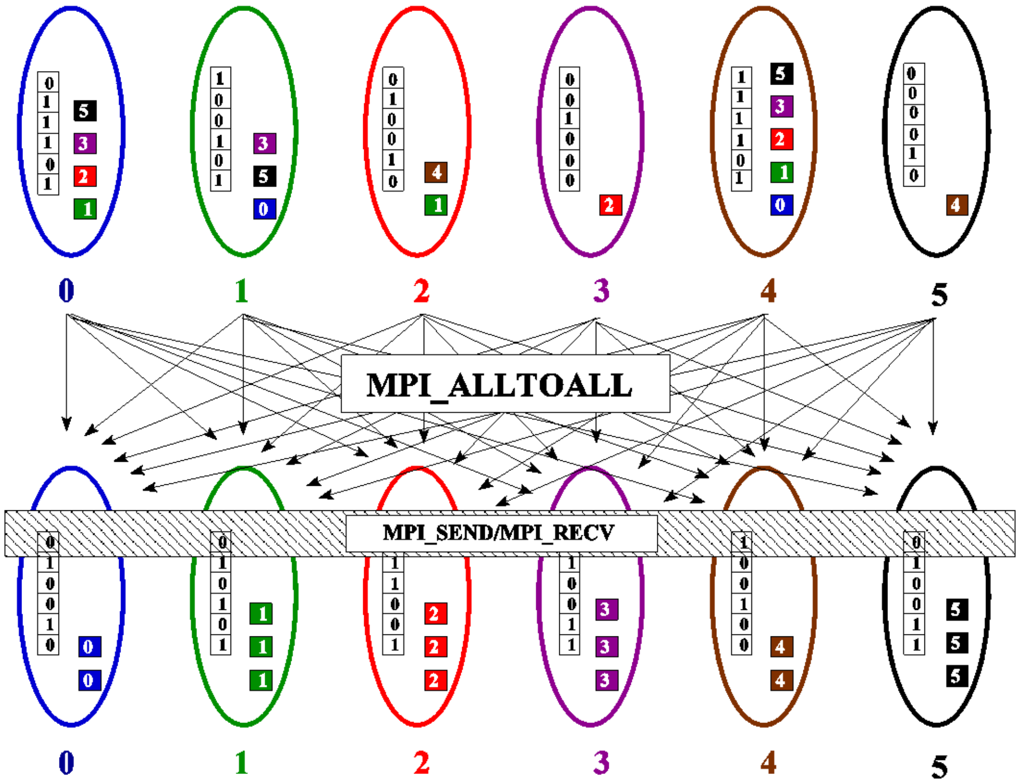
  OR:

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



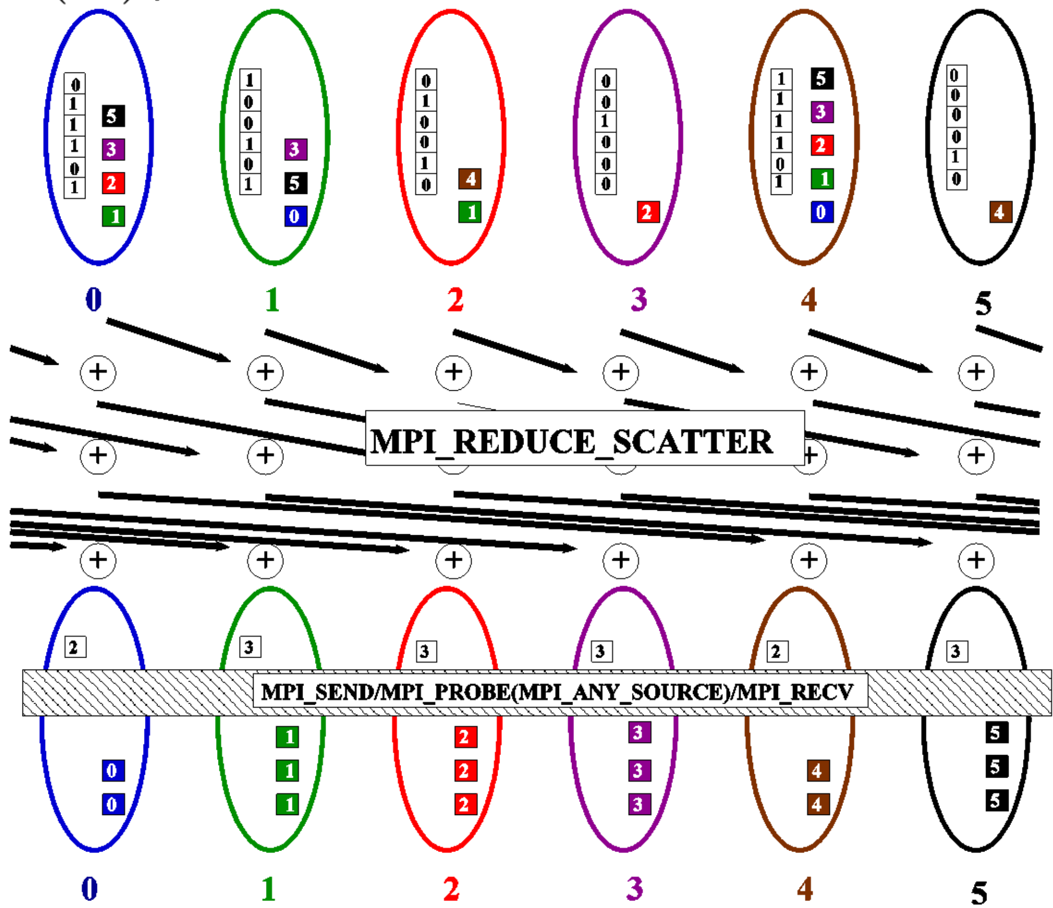
Using Alltoall (PEX)

- Bases 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!



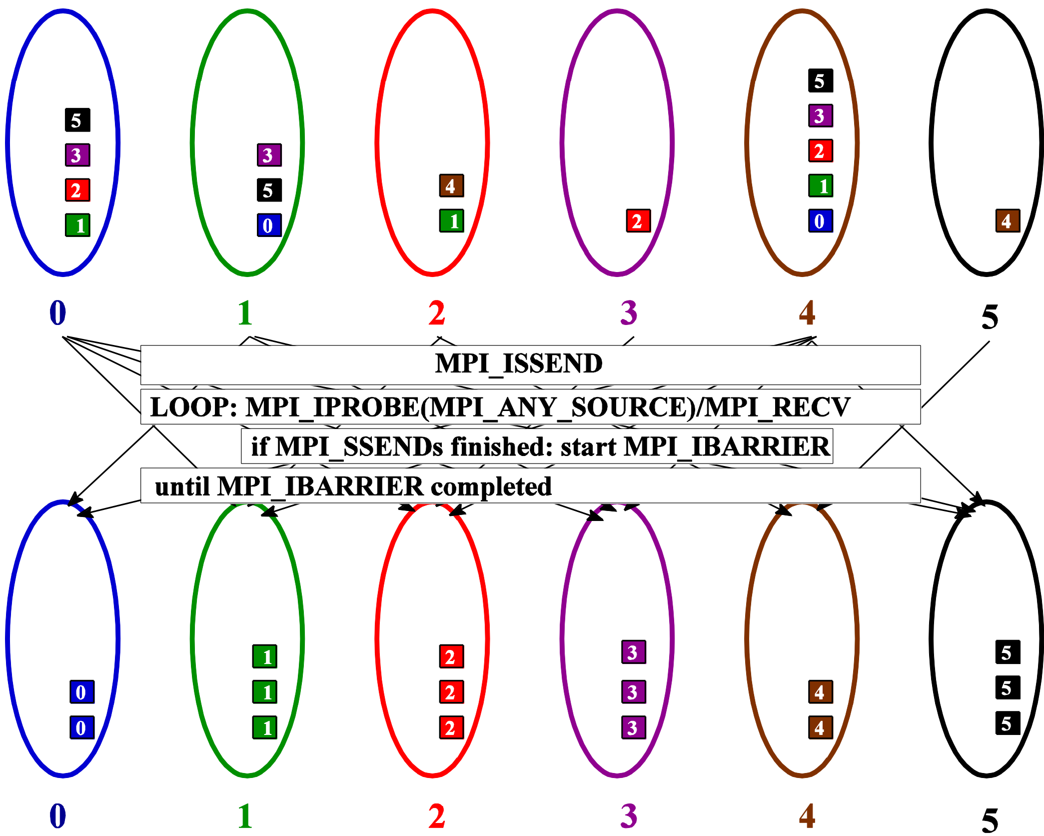
Reduce_scatter (PCX)

- 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)

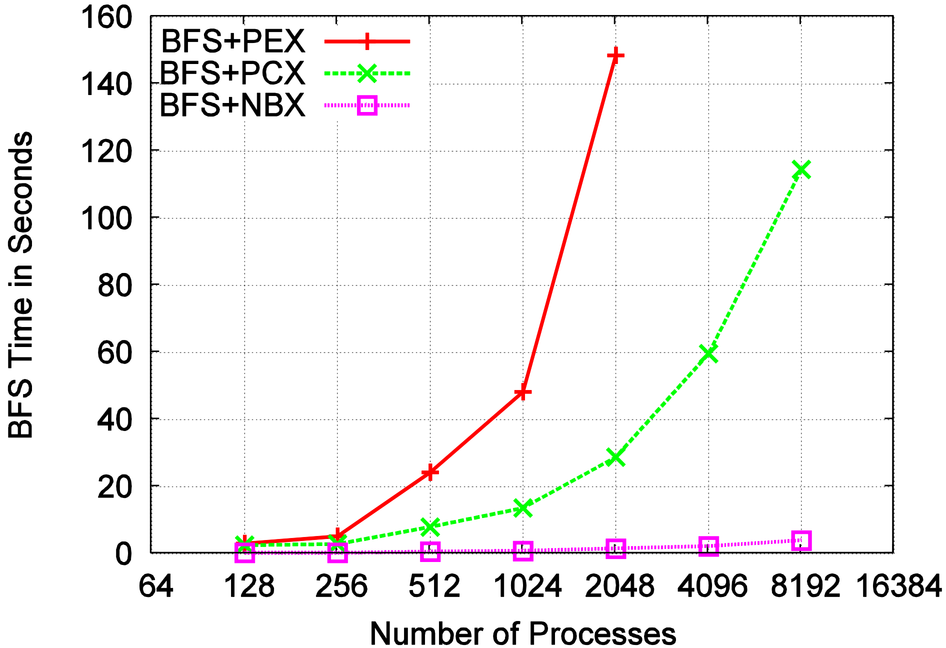
- Complexity - census (barrier): ($\Theta(\log(P))$)
  - Combines metadata with actual transmission
  - Point-to-point synchronization
  - Continue receiving until barrier completes
  - Processes start coll. synch. (barrier) when p2p phase ended
    - barrier = distributed marker!
  - Better than PEX, PCX, RSX!



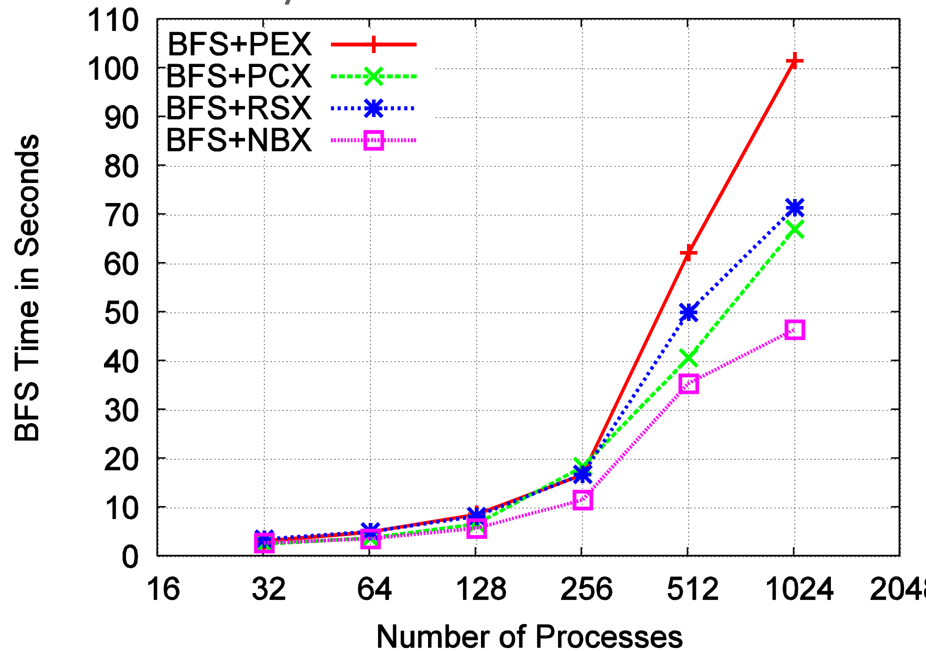
Parallel Breadth First Search

- 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!

A Complex Example: FFT

```
for(int x=0; x<n/p; ++x) 1d_fft(/* x-th stencil */);

// pack data for alltoall
MPI_Alltoall(&in, n/p*n/p, cplx_t, &out, n/p*n/p, cplx_t, comm);
// unpack data from alltoall and transpose

for(int y=0; y<n/p; ++y) 1d_fft(/* y-th stencil */);

// pack data for alltoall
MPI_Alltoall(&in, n/p*n/p, cplx_t, &out, n/p*n/p, cplx_t, comm);
// unpack data from alltoall and transpose
```

FFT Software Pipelining

```
NBC_Request req[nb];
for(int b=0; b<nb; ++b) { // loop over blocks
  for(int x=b*n/p/nb; x<(b+1)n/p/nb; ++x) 1d_fft(/* x-th stencil*/);

  // pack b-th block of data for alltoall
  NBC_Ialltoall(&in, n/p*n/p/bs, cplx_t, &out, n/p*n/p, cplx_t, comm, &req[b]);
}
NBC_Waitall(nb, req, MPI_STATUSES_IGNORE);

// modified unpack data from alltoall and transpose
for(int y=0; y<n/p; ++y) 1d_fft(/* y-th stencil */);
// pack data for alltoall
MPI_Alltoall(&in, n/p*n/p, cplx_t, &out, n/p*n/p, cplx_t, comm);
// unpack data from alltoall and transpose
```

A Complex Example: FFT

- Main parameter: nb vs. n → blocksize
- Strike balance between k-1st alltoall and kth FFT stencil block
- Costs per iteration:
  - Alltoall (bandwidth) costs: $T_{a2a} \approx n^2/p/nb * \beta$
  - FFT costs: $T_{fft} \approx n/p/nb * T_{1DFFT}(n)$
- Adjust blocksize parameters to actual machine
  - Either with model or simple sweep

Nonblocking And Collective Summary

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

Advanced Topics: Network Locality and Topology Mapping

Topology Mapping and Neighborhood Collectives

- Topology mapping basics
  - Allocation mapping vs. rank reordering
  - Ad-hoc solutions vs. portability
- MPI topologies
  - Cartesian
  - Distributed graph
- Collectives on topologies – neighborhood colls
  - Use-cases

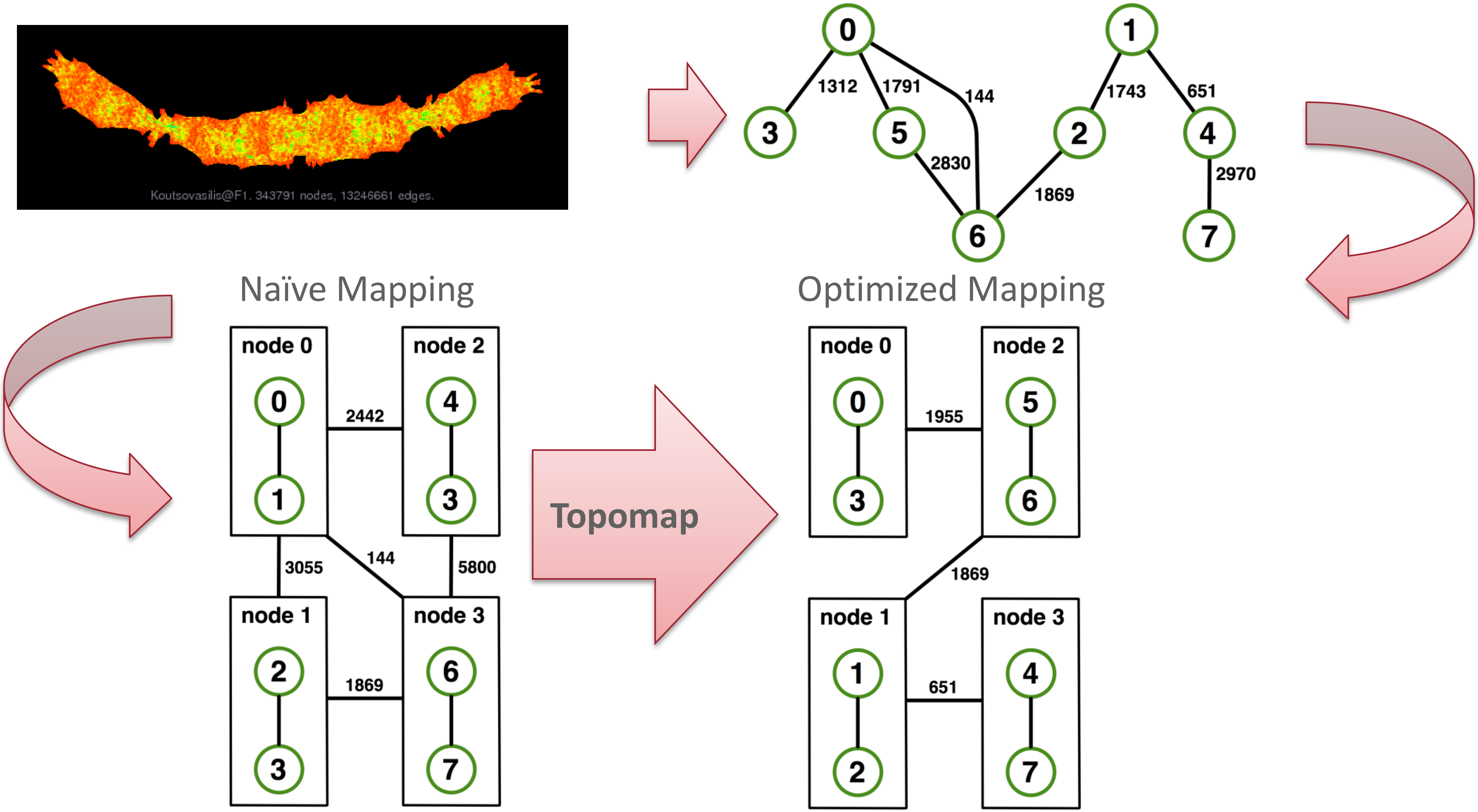
Topology Mapping Basics

- First type: Allocation mapping
  - Up-front specification of communication pattern
  - Batch system picks good set of nodes for given topology
- Properties:
  - Not widely supported by current batch systems
  - Either predefined allocation (BG/P), random allocation, or “global bandwidth maximation”
  - Also problematic to specify communication pattern upfront, not always possible (or static)

Topology Mapping Basics

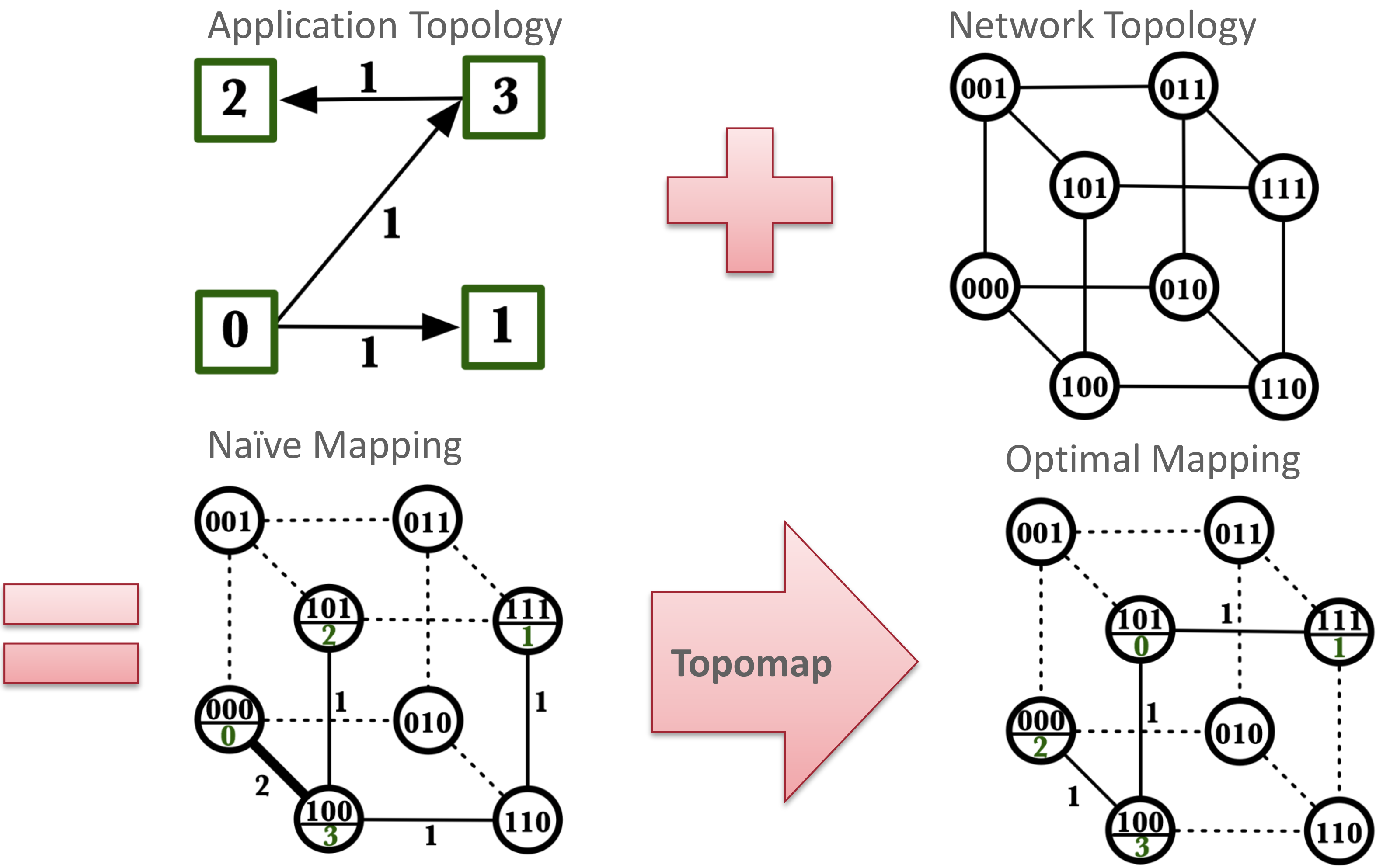
- Rank reordering
  - Change numbering in a given allocation to reduce congestion or dilation
  - Sometimes automatic (early IBM SP machines)
- Properties
  - Always possible, but effect may be limited (e.g., in a bad allocation)
  - Portable way: MPI process topologies
    - Network topology is not exposed
  - Manual data shuffling after remapping step

On-Node Reordering



Gottschling and Hoefler: Productive Parallel Linear Algebra Programming with Unstructured Topology Adaption

Off-Node (Network) Reordering



MPI Topology Intro

- Convenience functions (in MPI-1)
  - Create a graph and query it, nothing else
  - 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)

MPI_Cart_create

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);
```

- 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

MPI_Graph_create

- Don't use!!!!!

```
MPI_Graph_create(MPI_Comm comm_old, int nnodes, const int *index, const int *edges, int reorder, MPI_Comm *comm_graph)
```

- nnodes is the total number of nodes
- index i stores the total number of neighbors for the first i nodes (sum)
  - Acts as offset into edges array
- edges stores the edge list for all processes
  - Edge list for process j starts at index[j] in edges
  - Process j has index[j+1]-index[j] edges

MPI_Graph_create

- Don't use!!!!!

```
MPI_Graph_create(MPI_Comm comm_old, int nnodes, const
int *index, const int *edges, int reorder, MPI_Comm
*comm_graph)
```

- index i stores the total number of neighbors for the first i nodes (sum)
  - Acts as offset into edges array
- edges stores the edge list for all processes
  - Edge list for process j starts at index[j] in edges
  - Process j has index[j+1]-index[j] edges

Distributed graph constructor

- 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

Hoefler et al.: The Scalable Process Topology Interface of MPI 2.2

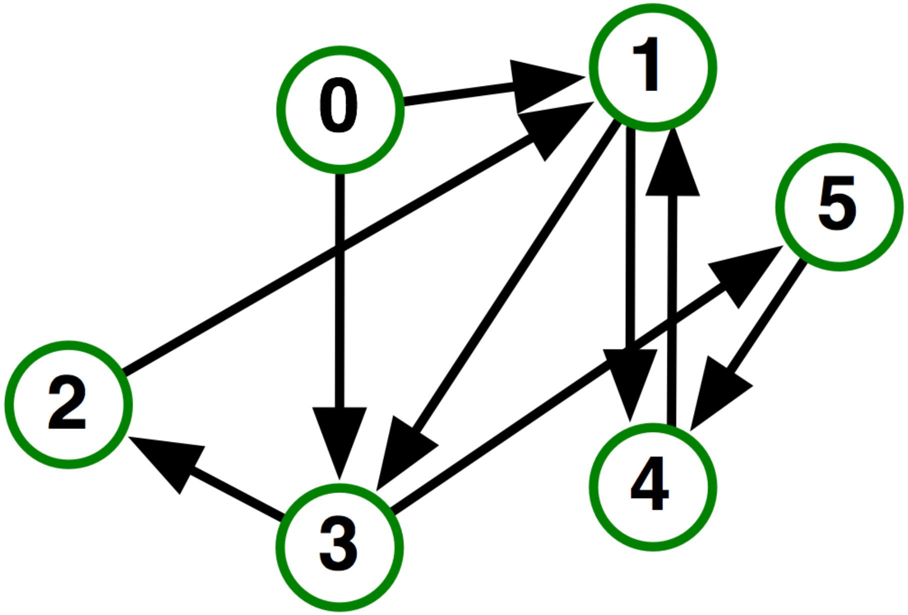
MPI_Dist_graph_create_adjacent

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



Hoefler et al.: The Scalable Process Topology Interface of MPI 2.2

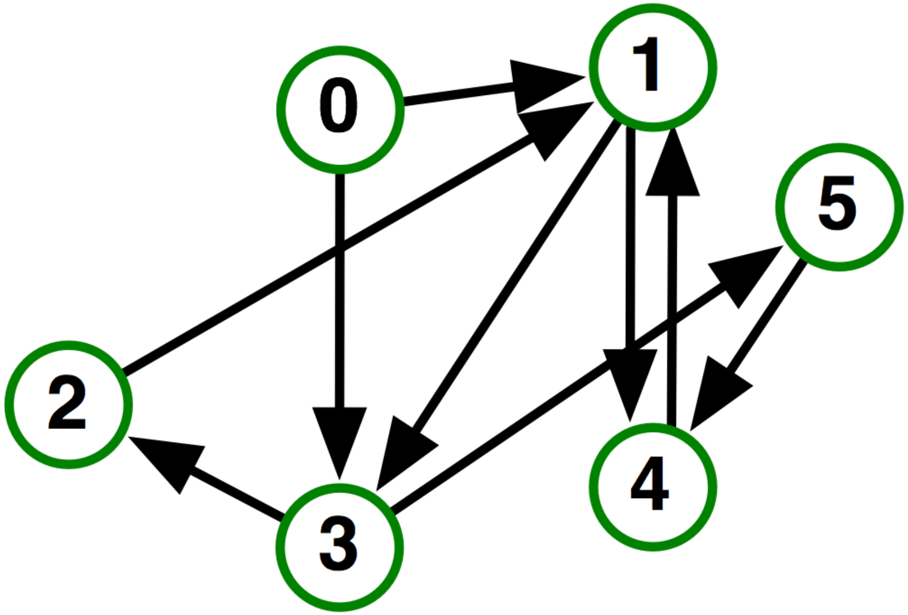
MPI_Dist_graph_create

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}
- ...



Distributed Graph Neighbor Queries

- MPI_Dist_graph_neighbors_count()

```
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
- MPI_Dist_graph_neighbors()
  - Query the neighbor list of **calling process**
  - Optionally return weights

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

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

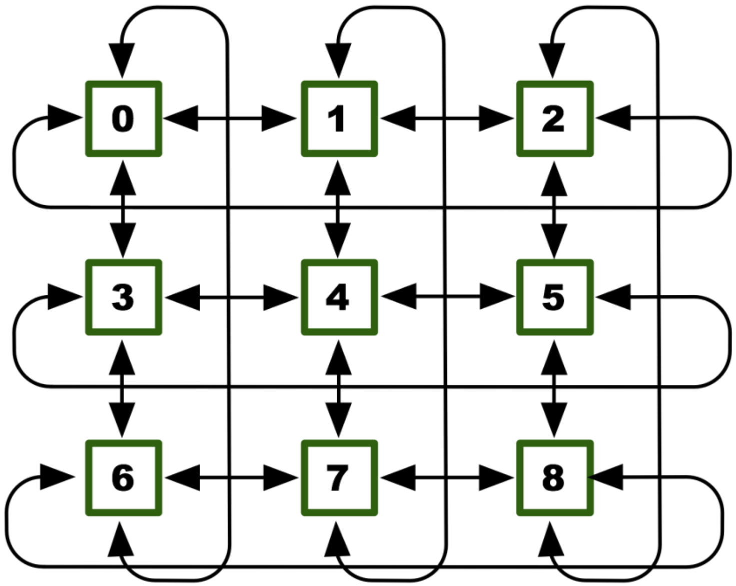
- Topologies implement no communication!
  - Just helper functions
- Collective communications only cover some patterns
  - E.g., no stencil pattern
- Several requests for “build your own collective” functionality in MPI
  - Neighborhood collectives are a simplified version
  - Cf. Datatypes for communication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

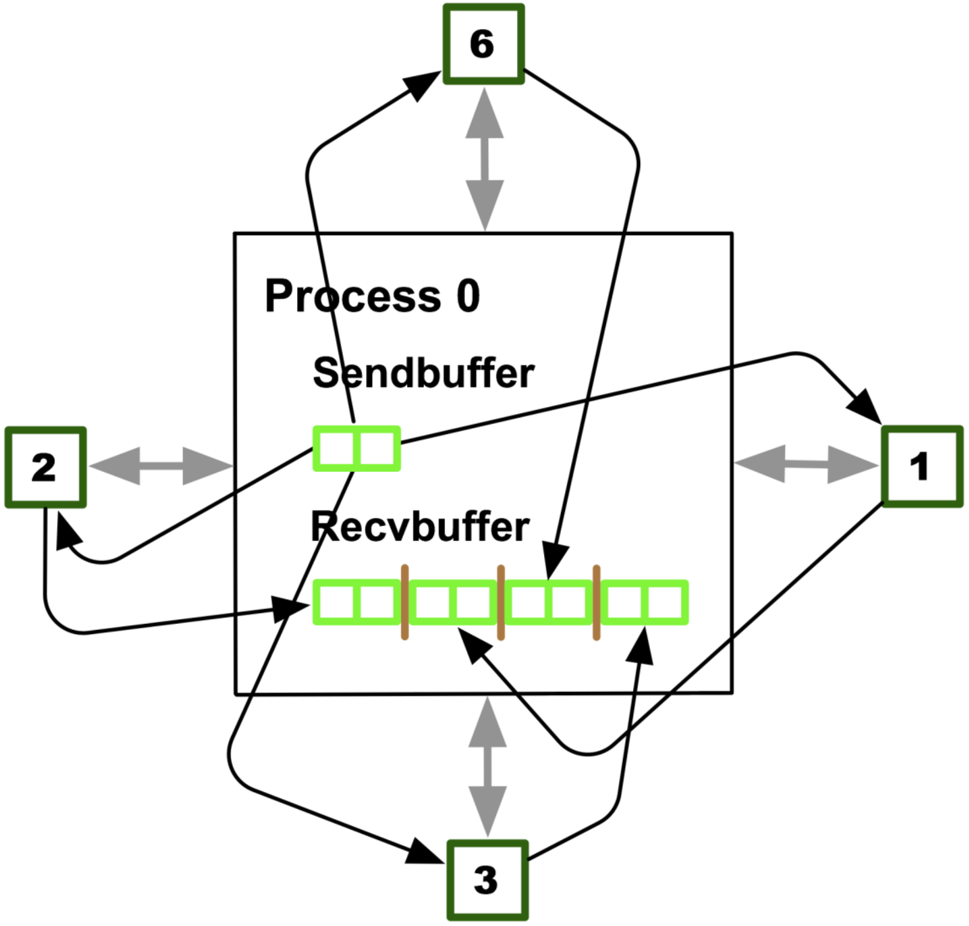
Cartesian Neighborhood Collectives

- 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 and w versions 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!

Why is Neighborhood Reduce Missing?

```
MPI_Ineighbor_allreducev(…);
```

- Was originally proposed (see original paper)
- High optimization opportunities
  - Interesting tradeoffs!
  - Research topic
- Not standardized due to missing use-cases
  - My team is working on an implementation
  - Offering the obvious interface

Topology Summary

- Topology functions allow to specify application communication patterns/topology
  - Convenience functions (e.g., Cartesian)
  - Storing neighborhood relations (Graph)
- Enables topology mapping (reorder=1)
  - Not widely implemented yet
  - May requires manual data re-distribution (according to new rank order)
- MPI does not expose information about the network topology (would be very complex)

Neighborhood Collectives Summary

- Neighborhood collectives add communication functions to process topologies
  - Collective optimization potential!
- Allgather
  - One item to all neighbors
- Alltoall
  - Personalized item to each neighbor
- High optimization potential (similar to collective operations)
  - Interface encourages use of topology mapping!

Section Summary

- Process topologies enable:
  - High-abstraction to specify communication pattern
  - Has to be relatively static (temporal locality)
    - Creation is expensive (collective)
  - Offers basic communication functions
- Library can optimize:
  - Communication schedule for neighborhood colls
  - Topology mapping

Concluding Remarks

- Parallelism is critical today, given that that is the only way to achieve performance improvement with the modern hardware
- MPI is an industry standard model for parallel programming
  - A large number of implementations of MPI exist (both commercial and public domain)
  - Virtually every system in the world supports MPI
- Gives user explicit control on data management
- Widely used by many many scientific applications with great success
- Your application can be next!

Web Pointers

- MPI standard : http://www.mpi-forum.org/docs/docs.html
- MPICH : http://www.mpich.org
- MPICH mailing list: discuss@mpich.org
- MPI Forum : http://www.mpi-forum.org/
- Other MPI implementations:
  - MVAPICH (MPICH on InfiniBand) : http://mvapich.cse.ohio-state.edu/
  - Intel MPI (MPICH derivative): http://software.intel.com/en-us/intel-mpi-library/
  - Microsoft MPI (MPICH derivative)
  - Open MPI : http://www.open-mpi.org/
- Several MPI tutorials can be found on the web