Advanced Parallel Programming with MPI

Pavan Balaji

Argonne National Laboratory

balaji@mcs.anl.gov

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

General principles in this tutorial

- Everything is practically oriented
- We will use lots of real example code to illustrate concepts
- At the end, you should be able to use what you have learned and write real code, run real programs
- Feel free to interrupt and ask questions
- If my pace is too fast or two slow, let me know

About Myself

- Computer Scientist in the Mathematics and Computer Science Division at Argonne National Laboratory
- Research interests in parallel programming, message passing, global address space and task space models
- Co-PI of the MPICH implementation of MPI
- Participate in the MPI Forum that defines the MPI standard
  - Co-author of the MPI-2.1, MPI-2.2 and the upcoming MPI-3.0 standards
  - Chair the hybrid programming working group for MPI-3
  - Committee member for the remote memory access (global address space runtime system) working group for MPI-3

What we will cover in this tutorial

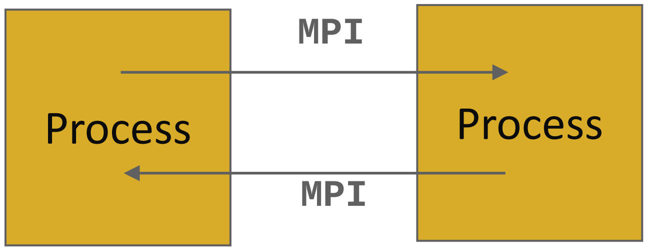
- **MPI: History and Philosophy**
- Basic definitions and concepts in MPI
- Advanced Topics
  - One-sided Communication
  - Hybrid programming (MPI+OpenMP/pthreads/CUDA/OpenCL)
  - Virtual Topology
- Conclusions and Final Q/A

Sample Parallel Programming Models

- Shared Memory Programming
  - Processes share memory address space (threads model)
  - Application ensures no data corruption (Lock/Unlock)
- Transparent Parallelization
  - Compiler works magic on sequential programs
- Directive-based Parallelization
  - Compiler needs help (e.g., OpenMP)
- Message Passing
  - Explicit communication between processes (like sending and receiving emails)

The Message-Passing Model

- A *process* is (traditionally) a program counter and address space.
- Processes may have multiple *threads* (program counters and associated stacks) sharing a single address space. MPI is for communication among processes, which have separate address spaces.
- Inter-process communication consists of
  - synchronization
  - movement of data from one process's address space to another's.



Standardizing Message-Passing Models with MPI

- Early vendor systems (Intel's NX, IBM's EUI, TMC's CMMD) were not portable (or very capable)
- Early portable systems (PVM, p4, TCGMSG, Chameleon) were mainly research efforts
  - Did not address the full spectrum of message-passing issues
  - Lacked vendor support
  - Were not implemented at the most efficient level
- The MPI Forum was a collection of vendors, portability writers and users that wanted to standardize all these efforts

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

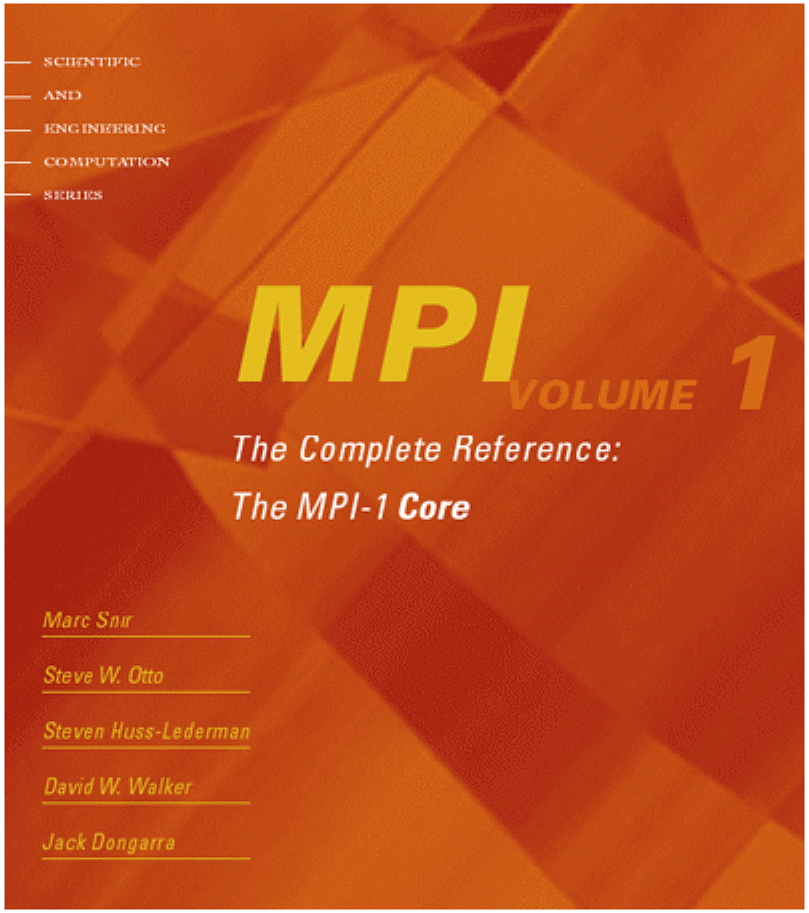
What is in MPI-1

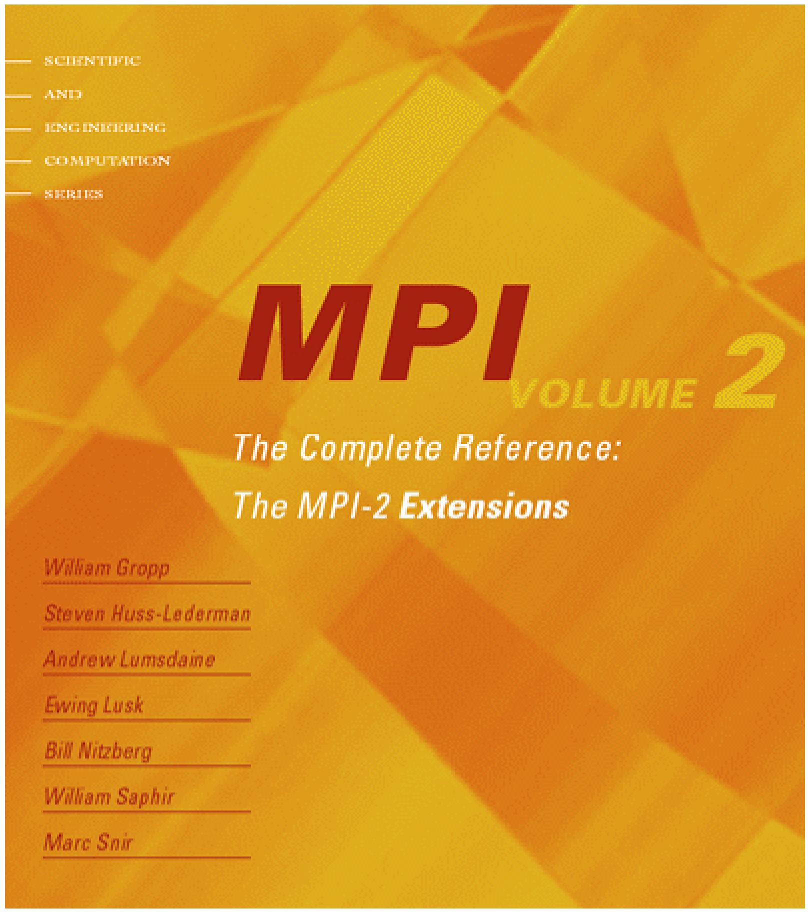
- Basic functions for communication (100+ functions)
- Blocking sends, receives
- Nonblocking sends and receives
- Variants of above
- Rich set of collective communication functions
  - Broadcast, scatter, gather, etc
  - Very important for performance; widely used
- Datatypes to describe data layout
- Process topologies
- C, C++ and Fortran bindings
- Error codes and classes

Following MPI Standards

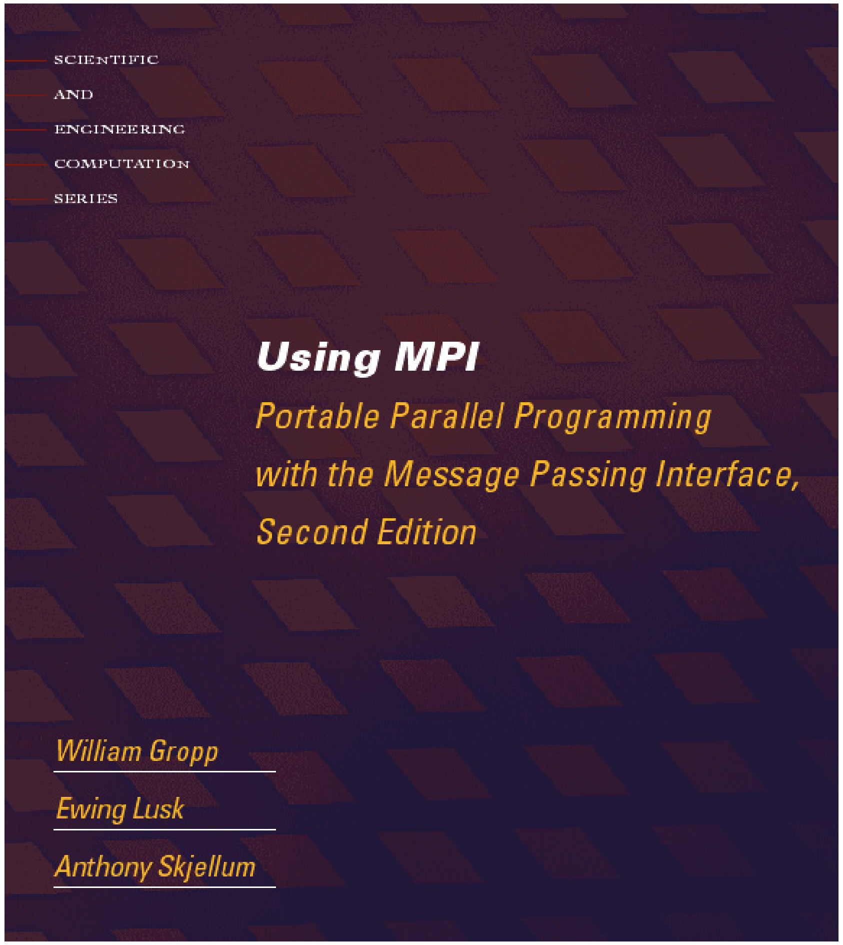
- MPI-2 was released in 2000
  - Several additional features including MPI + threads, MPI-I/O, remote memory access functionality and many others
- MPI-2.1 (2008) and MPI-2.2 (2009) were recently released with some corrections to the standard and small features
- MPI-3.0 is being released this September
- 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 stuff, including other talks and tutorials, a FAQ, other MPI pages

The MPI Standard (1 & 2)





Tutorial Material on MPI-1 and MPI-2

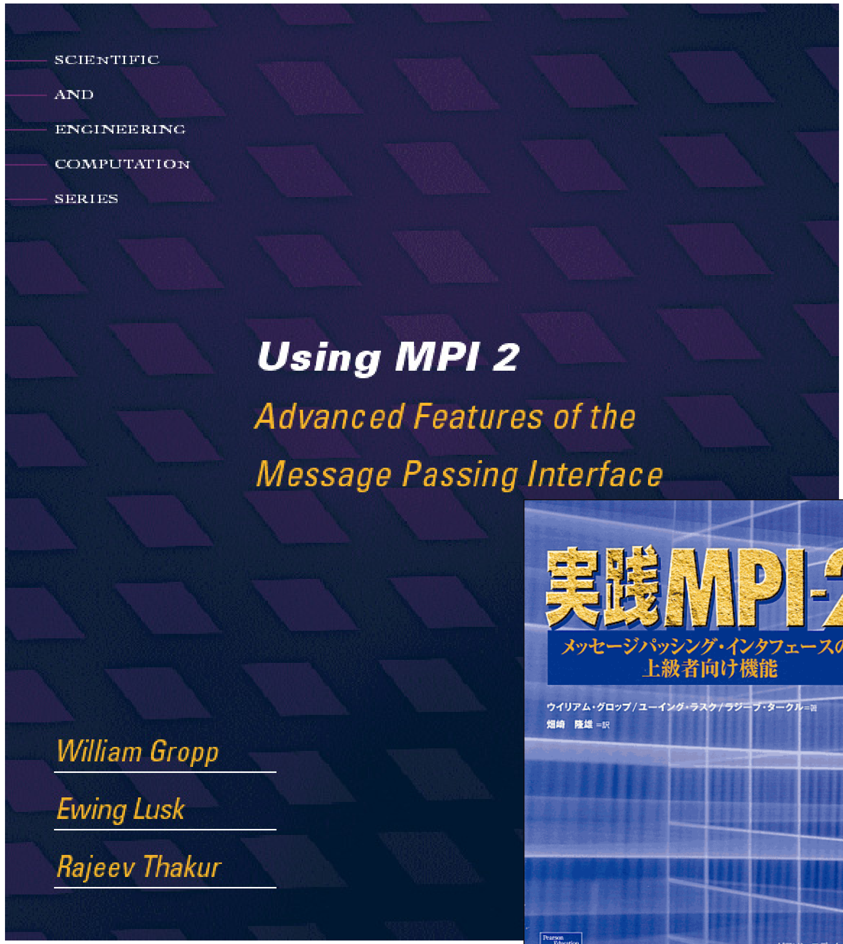


SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES

Using MPI

Portable Parallel Programming with the Message Passing Interface, Second Edition

William Gropp
Ewing Lusk
Anthony Skjellum

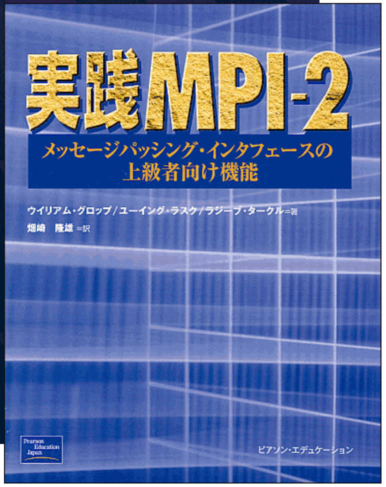


SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES

Using MPI 2

Advanced Features of the Message Passing Interface

William Gropp
Ewing Lusk
Rajeev Thakur



実践MPI-2

メッセージパッシング・インタフェースの
上級者向け機能

ウイリアム・グロップ/ユーイング・ラスク/ラジーブ・タークル=著
畑崎 隆雄 =訳

Pearson Education Japan

ピアソン・エデュケーション

http://www.mcs.anl.gov/mpi/usingmpi
http://www.mcs.anl.gov/mpi/usingmpi2

Reasons for Using MPI

- **Standardization** - MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries
- **Portability** - There is no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard
- **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance
- **Functionality** – Rich set of features
- **Availability** - A variety of implementations are available, both vendor and public domain
  - MPICH is a popular open-source and free implementation of MPI
  - Vendors and other collaborators take MPICH and add support for their systems
    - Intel MPI, IBM Blue Gene MPI, Cray MPI, Microsoft MPI, MVAPICH, MPICH-MX

Important considerations while using MPI

- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs
  - Don't expect magic to happen: If you ask MPI to move data from process 1 to process 2, MPI will do that for you
- High-performance and portability
  - Some users prefer rich feature set, while others prefer a small set doing exactly what they want
  - MPI has always chosen to provide a rich set of portable features. If you want a small subset providing only the things you want, you should write a high-level library on top of MPI
    - Almost all domains do that – E.g., PETSc, Trillinos, FFTW, ADLB, …

Basic Definitions and Concepts

Compiling and Running MPI applications (more details later)

- MPI is a library
  - Applications can be written in C, C++ or Fortran and appropriate calls to MPI can be added where required
- Compilation:
  - Regular applications:
    - `gcc test.c -o test`
  - MPI applications
    - `mpicc test.c -o test`
- Execution:
  - Regular applications
    - `./test`
  - MPI applications (running with 16 processes)
    - `mpiexec –np 16 ./test`

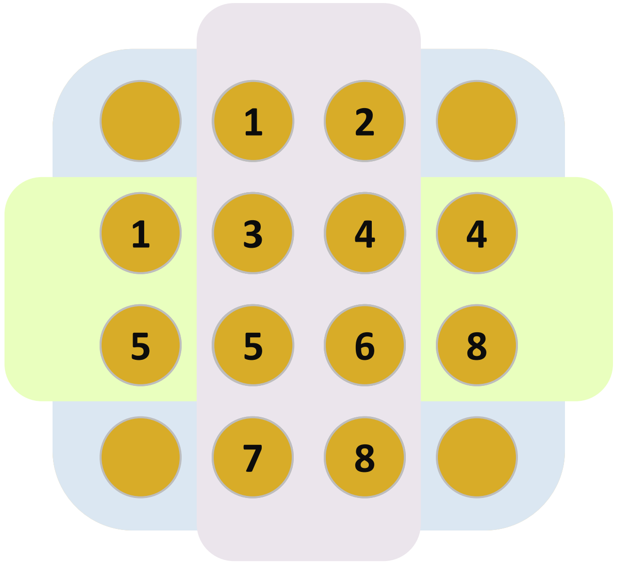
Process Identification

- MPI processes can be collected into groups
  - Each group can have multiple contexts (identifiers)
  - *Group + identifier == communicator*
  - When an MPI application starts, the group of all processes is initially given a predefined communicator called **`MPI_COMM_WORLD`**
  - More communicators can be created out of **`MPI_COMM_WORLD`**
- A process is identified by a unique number within each communicator, called *rank*
  - For two different communicators, the same process can have two different ranks: so the meaning of a “rank” is only defined when you specify the communicator

Communicators

```
mpiexec  -np  16  ./test
```

Communicators do not need to contain all processes in the system



When you start an MPI program, there is one predefined communicator **MPI_COMM_WORLD**

Every process in a communicator has an ID called as "rank"

Can make copies of this communicator (same group of processes, but different "aliases")

The same process might have different ranks in different communicators

Communicators can be created "by hand" or using tools provided by MPI

Simple MPI Program Identifying Processes

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char ** argv)
{
    int rank, size;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d\n", rank, size);

    MPI_Finalize();
    return 0;
}
```

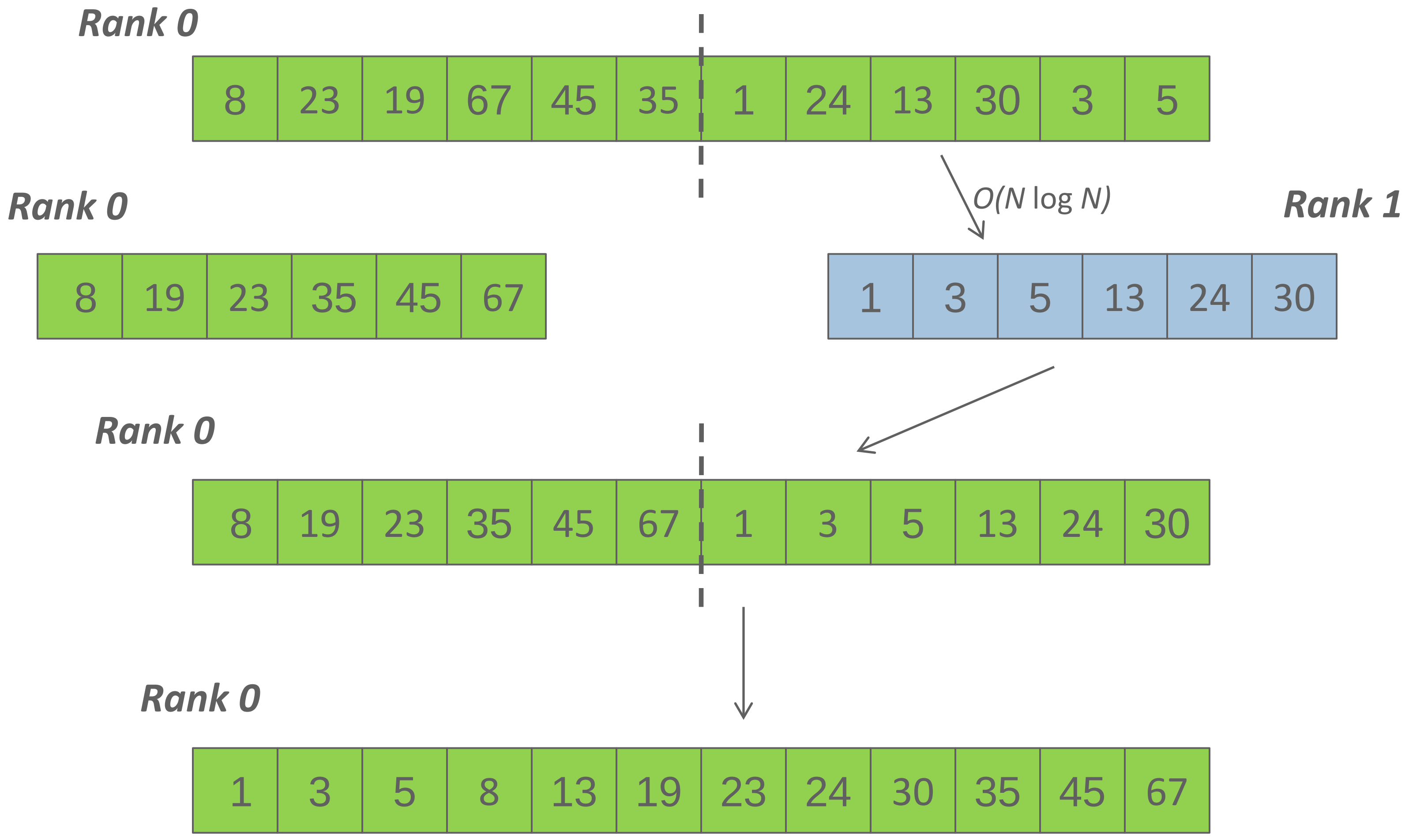
MPI Tags

- Messages are sent with an accompanying user-defined integer *tag*, to assist the receiving process in identifying the message
- For example, if an application is expecting two types of messages from a peer, tags can help distinguish these two types
- Messages can be screened at the receiving end by specifying a specific tag

Simple Communication in MPI

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char ** argv)
{
    int rank, data[100];

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0)
        MPI_Send(data, 100, MPI_INT, 1, 0, MPI_COMM_WORLD);
    else if (rank == 1)
        MPI_Recv(data, 100, MPI_INT, 0, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);

    MPI_Finalize();
    return 0;
}
```

Parallel Sort using MPI Send/Recv



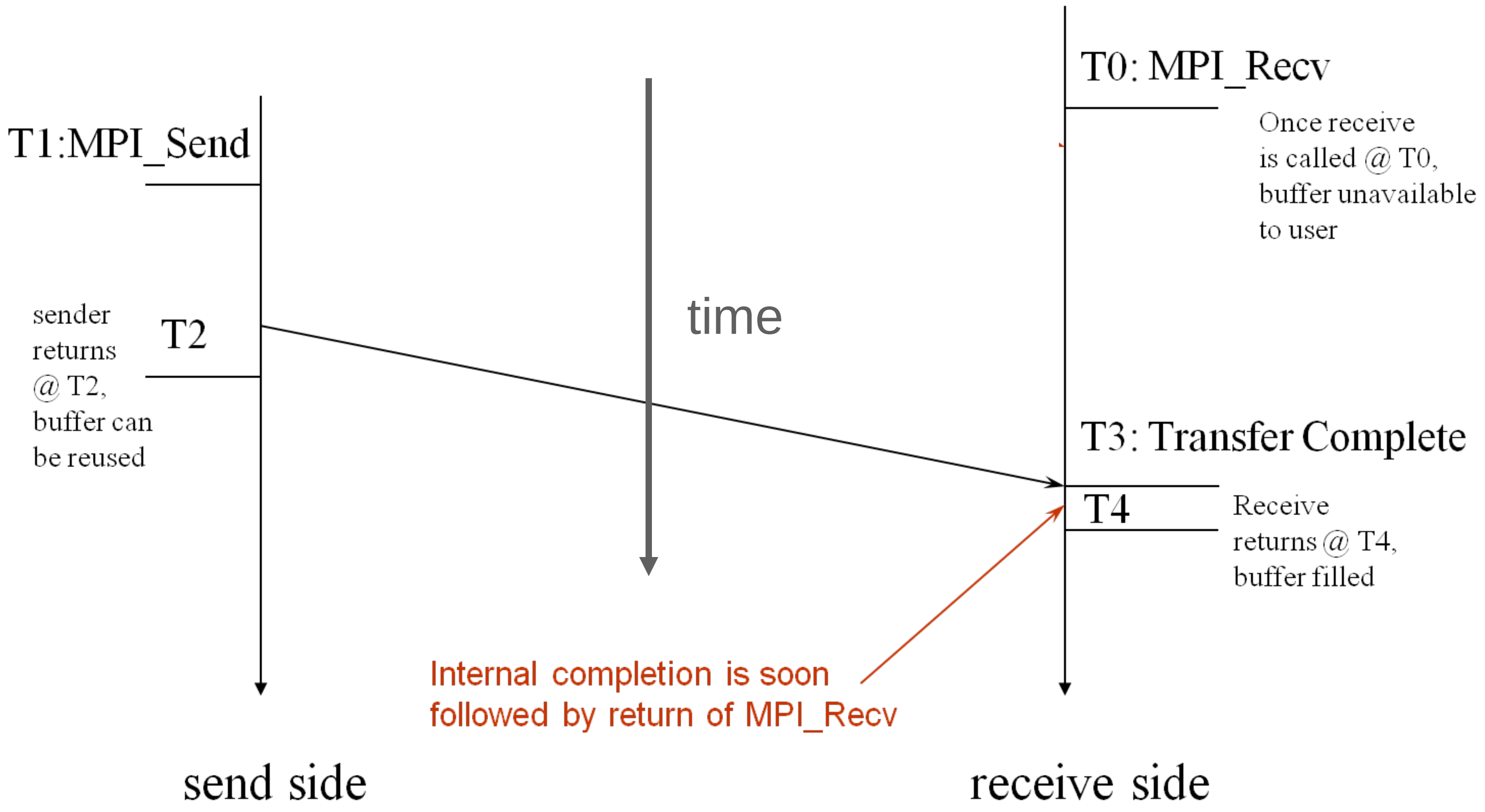
Parallel Sort using MPI Send/Recv (contd.)

```
#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;
}
```

MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - `MPI_INIT – initialize the MPI library (must be the first routine called)`
  - `MPI_COMM_SIZE - get the size of a communicator`
  - `MPI_COMM_RANK – get the rank of the calling process in the communicator`
  - `MPI_SEND – send a message to another process`
  - `MPI_RECV – send a message to another process`
  - `MPI_FINALIZE – clean up all MPI state (must be the last MPI function called by a process)`
- For performance, however, you need to use other MPI features

Blocking Send-Receive Diagram


T0: MPI_Recv
Once receive is called @ T0, buffer unavailable to user
T1:MPI_Send
sender returns @ T2, buffer can be reused
T2
time
T3: Transfer Complete
T4
Receive returns @ T4, buffer filled
Internal completion is soon followed by return of MPI_Recv
send side
receive side

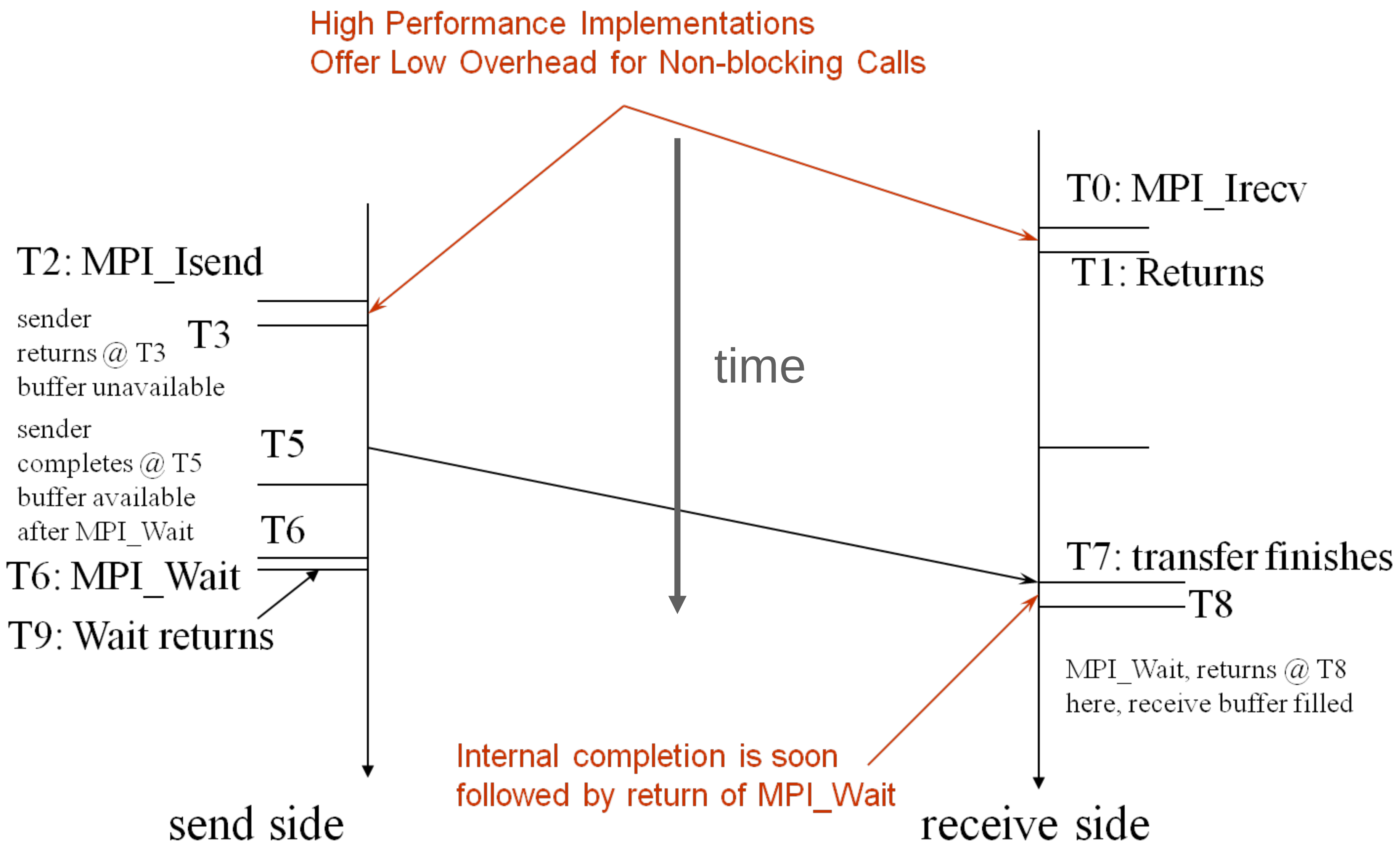
Non-Blocking Communication

- Non-blocking (asynchronous) operations return (immediately) “request handles” that can be waited on and queried
  - `MPI_ISEND(start, count, datatype, dest, tag, comm, request)`
  - `MPI_IRECV(start, count, datatype, src, tag, comm, request)`
  - `MPI_WAIT(request, status)`
- Non-blocking operations allow overlapping computation and communication
- One can also test without waiting using **MPI_TEST**
  - **MPI_TEST(request, flag, status)**
- Anywhere you use **MPI_SEND** or **MPI_RECV**, you can use the pair of **MPI_ISEND/MPI_WAIT** or **MPI_IRECV/MPI_WAIT**
- Combinations of blocking and non-blocking sends/receives can be used to synchronize execution instead of barriers

Multiple Completions

- It is sometimes desirable to wait on multiple requests:
  - `MPI_Waitall(count, array_of_requests, array_of_statuses)`
  - `MPI_Waitany(count, array_of_requests, &index, &status)`
  - `MPI_Waitsome(count, array_of_requests, array_of_indices, array_of_statuses)`
- There are corresponding versions of **`test`** for each of these

Non-Blocking Send-Receive Diagram


High Performance Implementations
Offer Low Overhead for Non-blocking Calls
T0: MPI_Irecv
T1: Returns
T2: MPI_Isend
sender returns @ T3 buffer unavailable
T3
time
sender completes @ T5 buffer available after MPI_Wait
T5
T6
T6: MPI_Wait
T9: Wait returns
T7: transfer finishes
T8
MPI_Wait, returns @ T8 here, receive buffer filled
Internal completion is soon followed by return of MPI_Wait
send side
receive side

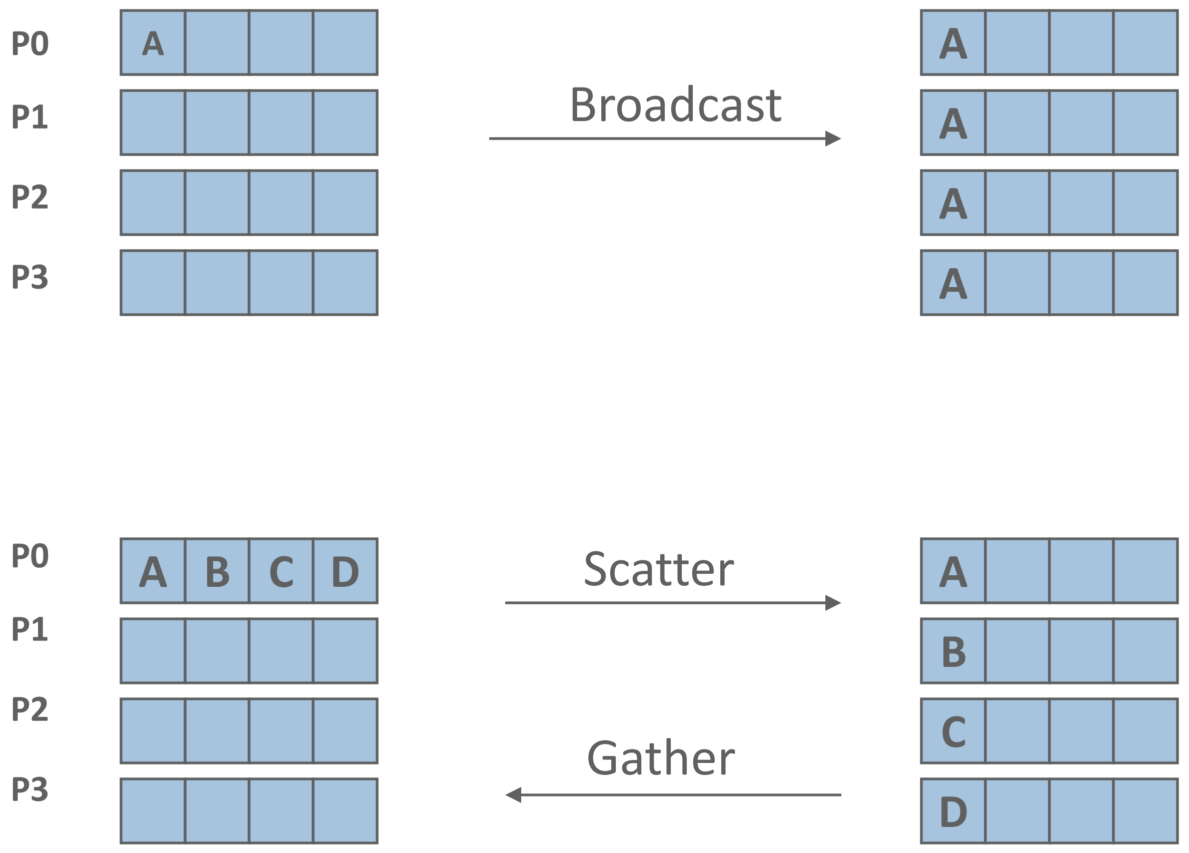
MPI Collective Communication

- Communication and computation is coordinated among a group of processes in a communicator
- Tags are not used; different communicators deliver similar functionality
- No non-blocking collective operations in MPI-1 and MPI-2
  - They are added in MPI-3
- Three classes of operations: synchronization, data movement, collective computation

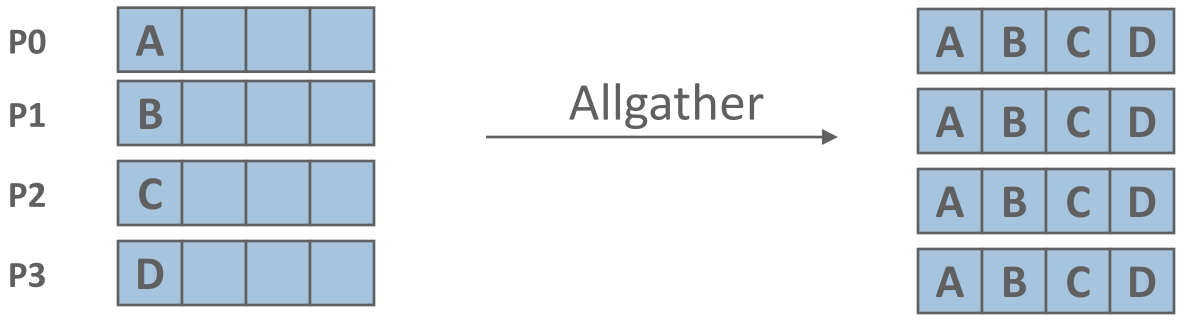
Synchronization

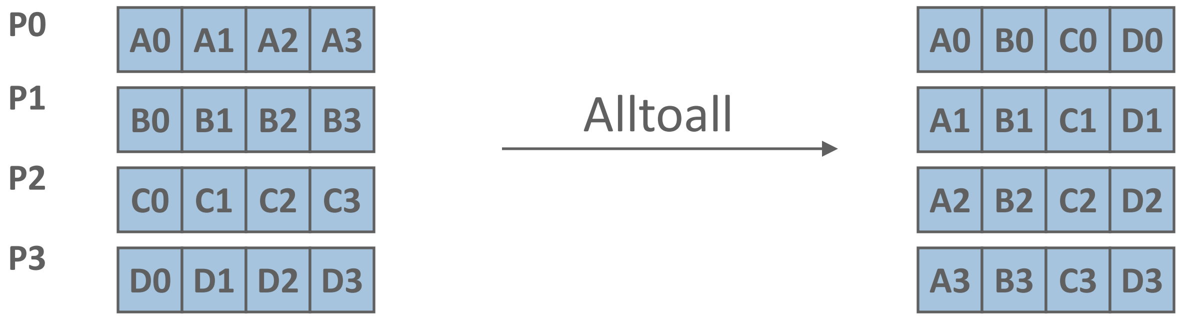
- `MPI_BARRIER(comm)`
  - Blocks until all processes in the group of the communicator **`comm`** call it
  - A process cannot get out of the barrier until all other processes have reached barrier

Collective Data Movement

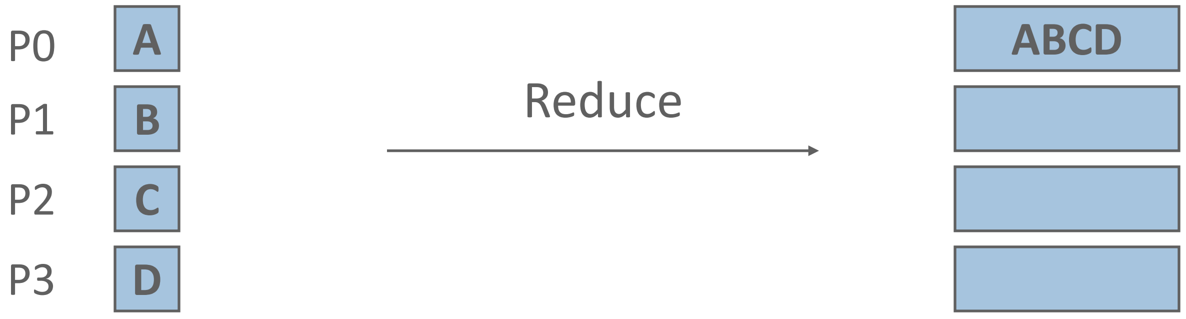


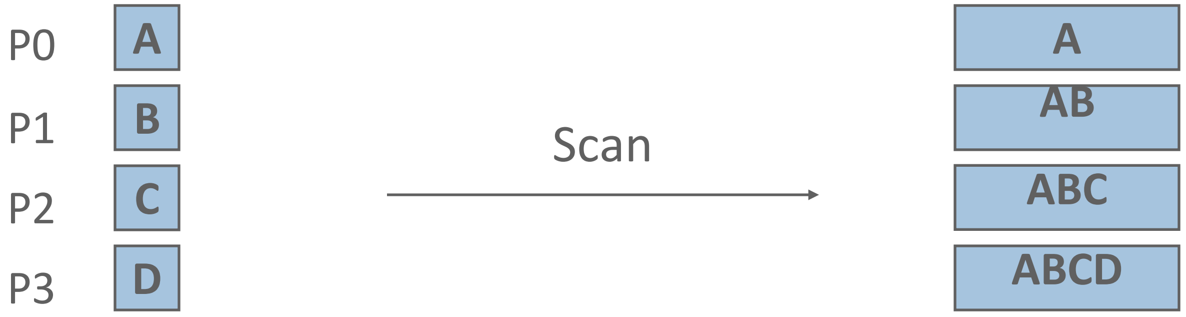
More Collective Data Movement





Collective Computation

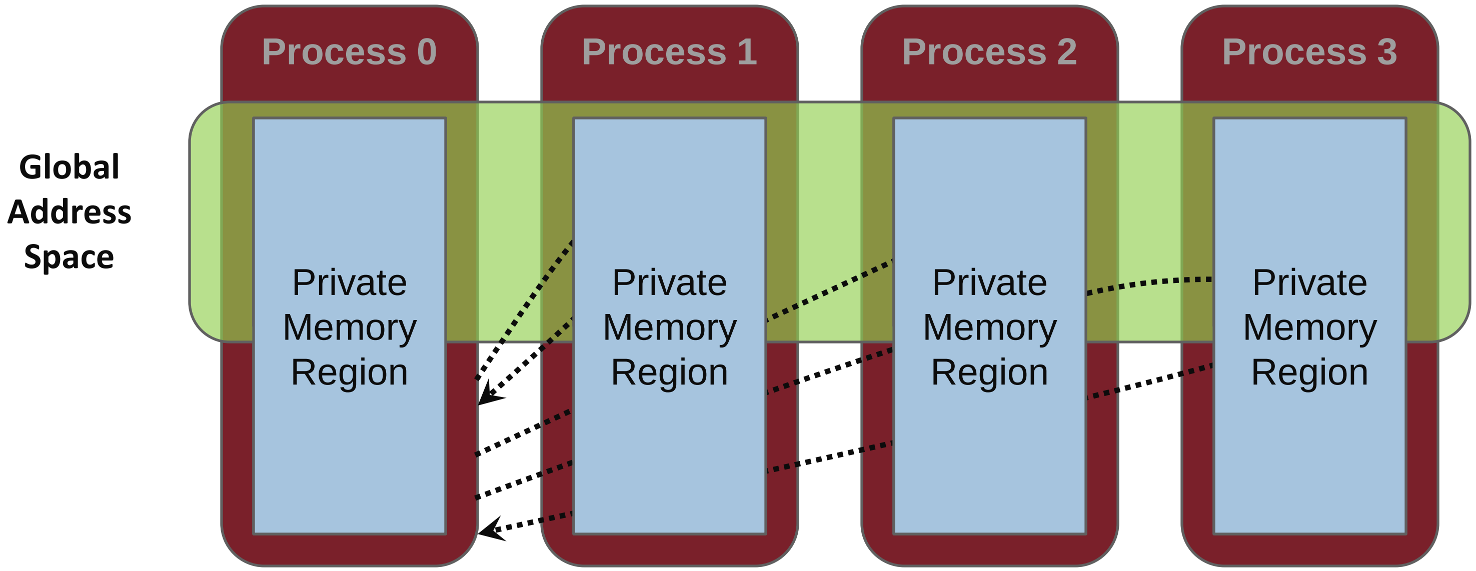




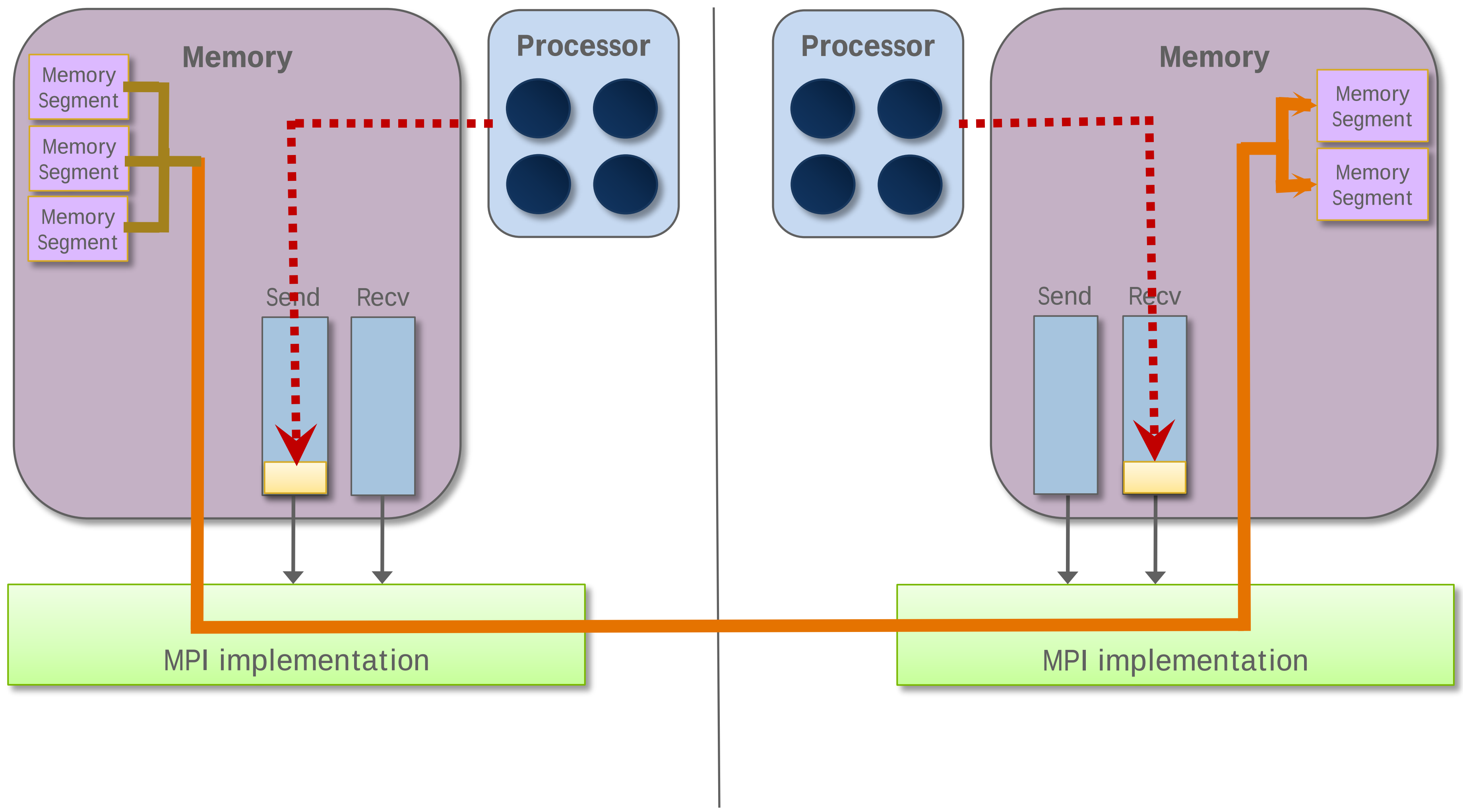
Advanced Topics: One-sided Communication

One-sided Communication

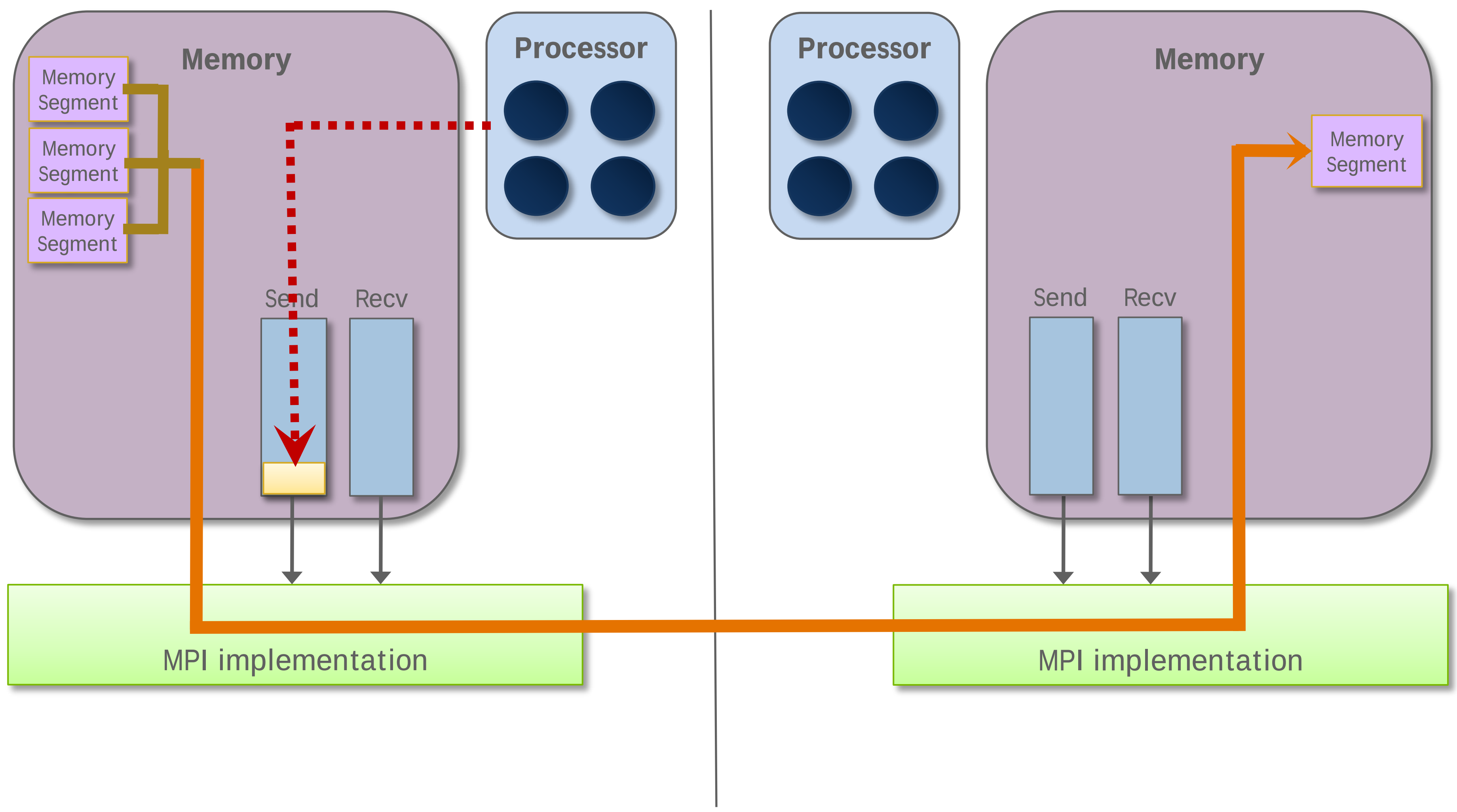
- The basic idea of one-sided communication models is to decouple data movement with process synchronization
  - Should be able move data without requiring that the remote process synchronize
  - Each process exposes a part of its memory to other processes
  - Other processes can directly read from or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



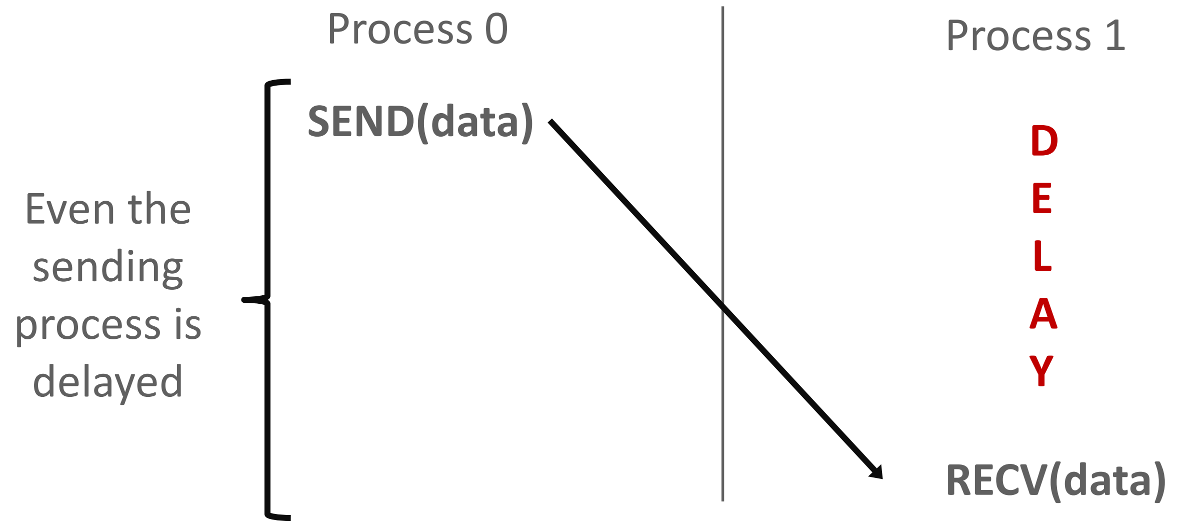
Two-sided Communication Example

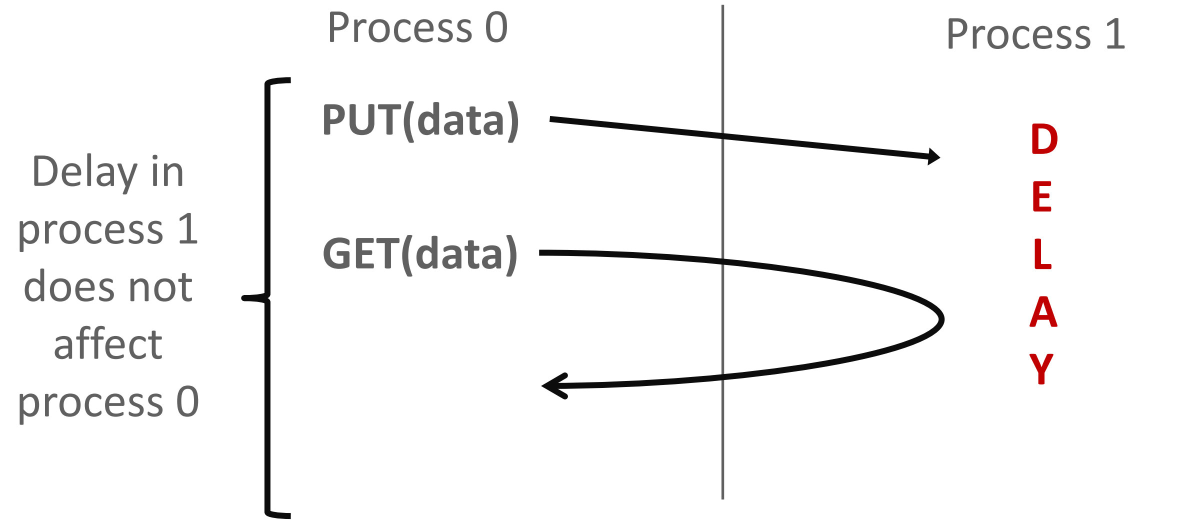


One-sided Communication Example



Comparing One-sided and Two-sided Programming

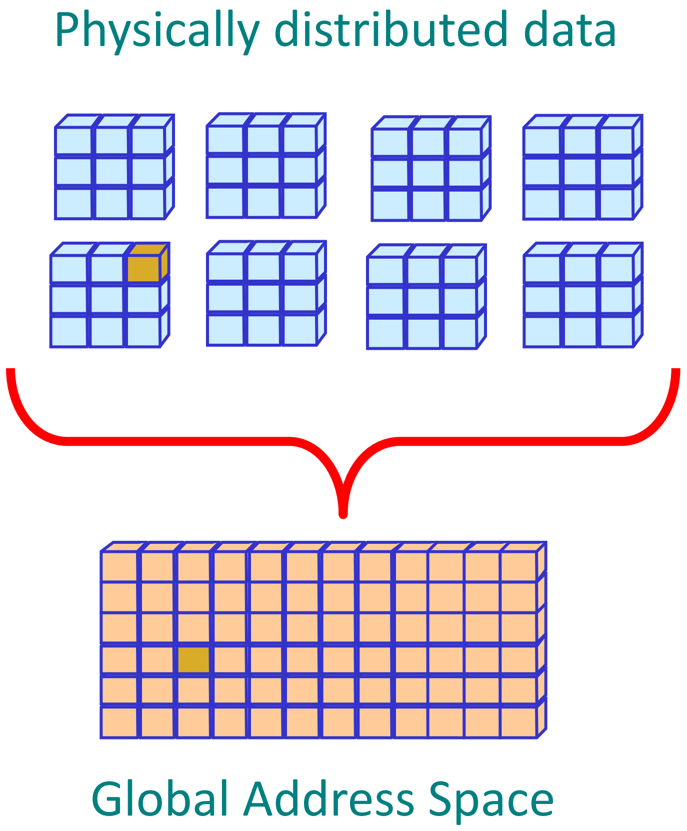




Possible Applications of One-sided Communication

- 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

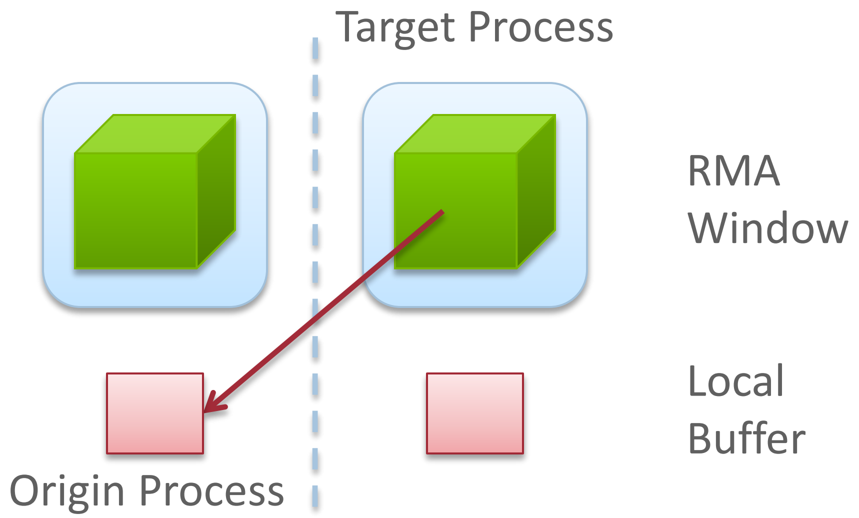
Data movement

- 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)
```

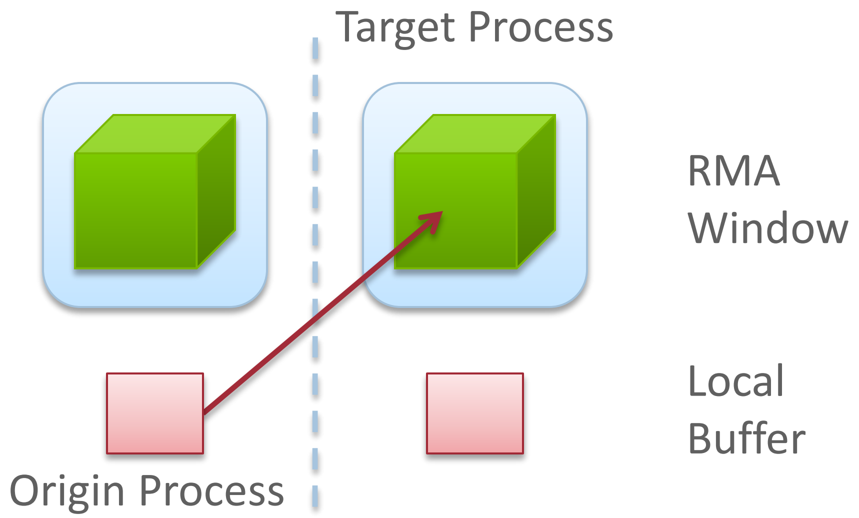
- Move data to origin, from target
- Separate data description triples for origin and target



Data movement: Put

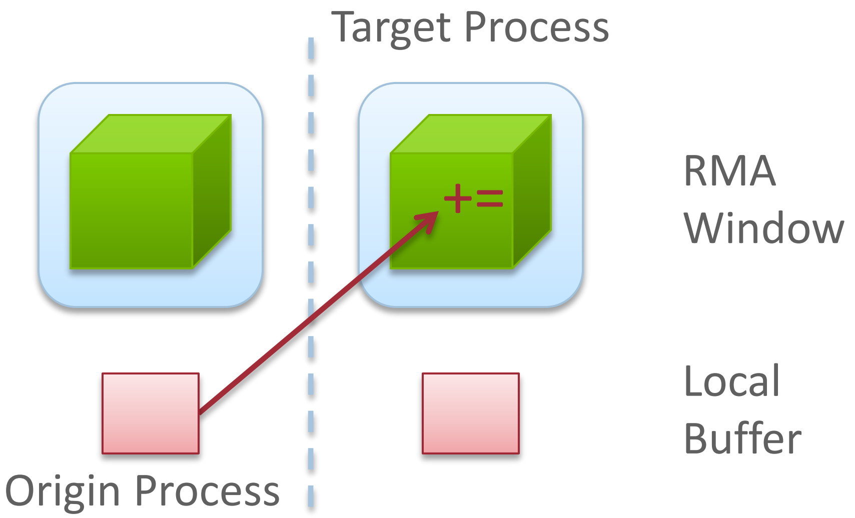
```
MPI_Put(
    origin_addr, origin_count, origin_datatype,
    target_rank,
    target_disp, target_count, target_datatype,
    win)
```

- Move data <u>from</u> origin, <u>to</u> target
- Same arguments as MPI_Get



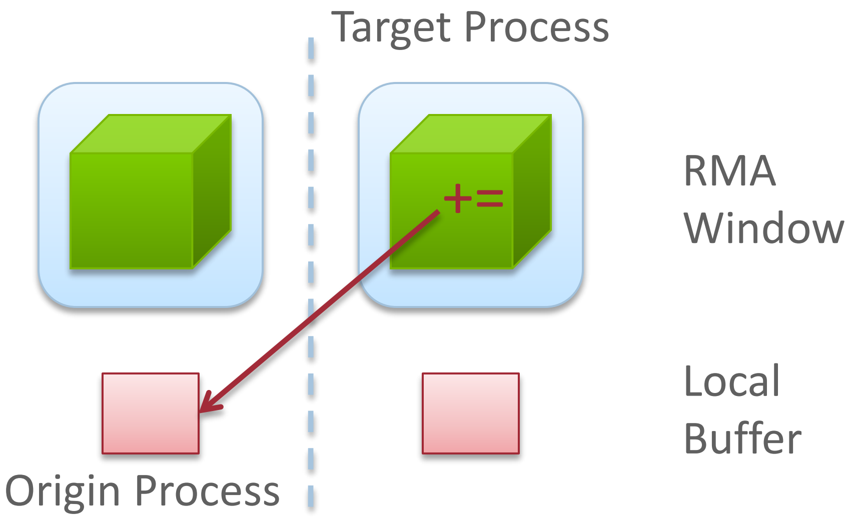
Data aggregation: *Accumulate*

- 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



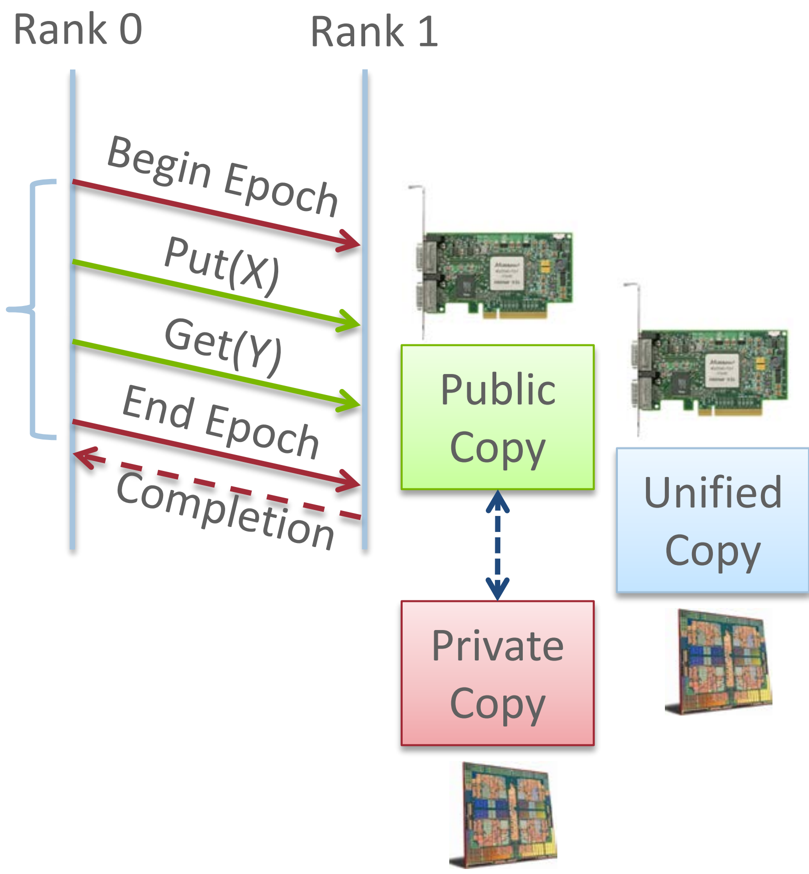
Data aggregation: *Get Accumulate*

- 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

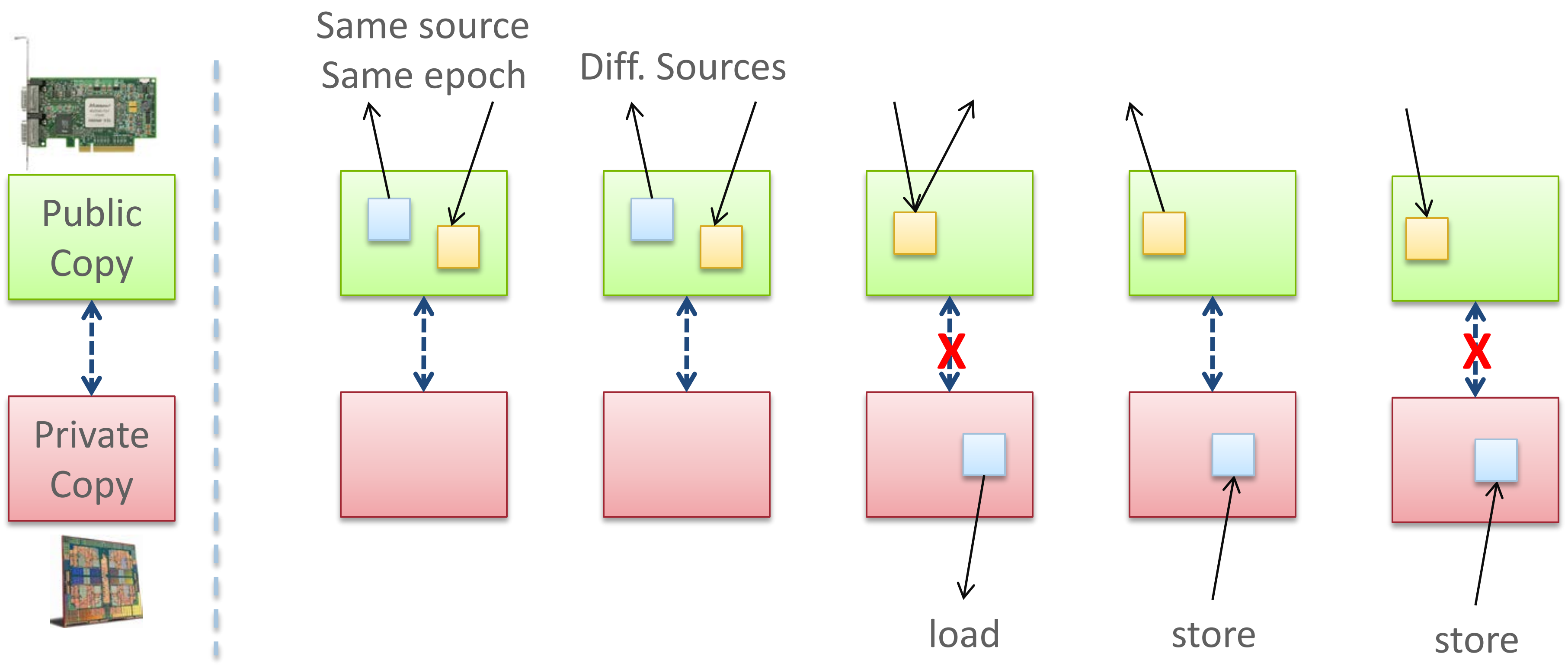


MPI RMA Memory Model

- 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

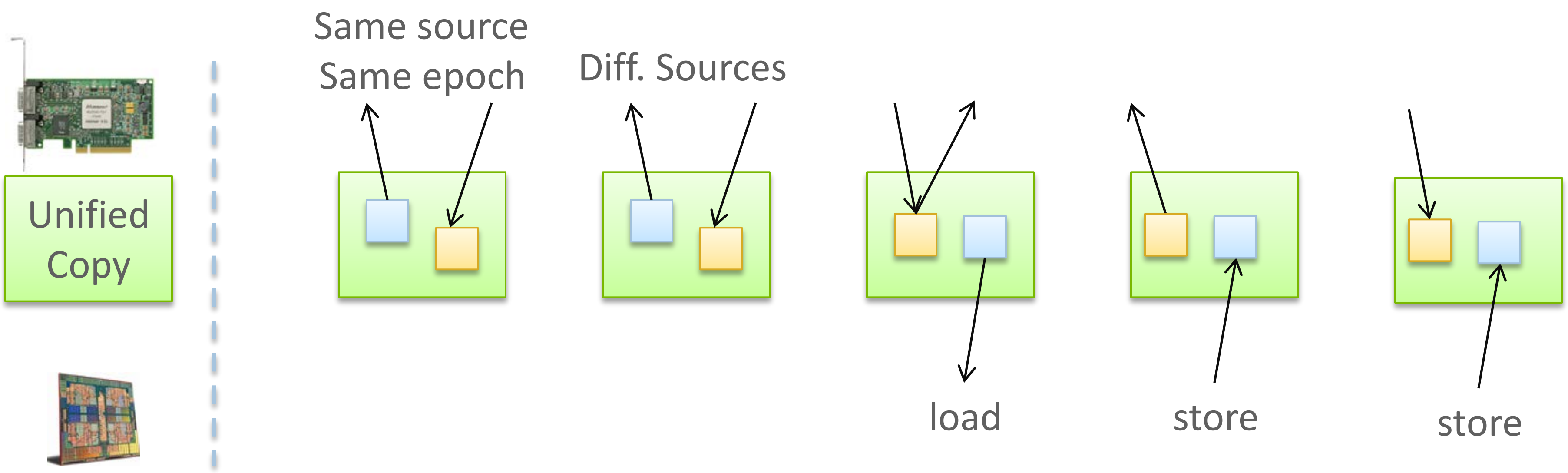


MPI RMA Memory Model (separate windows)



- 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

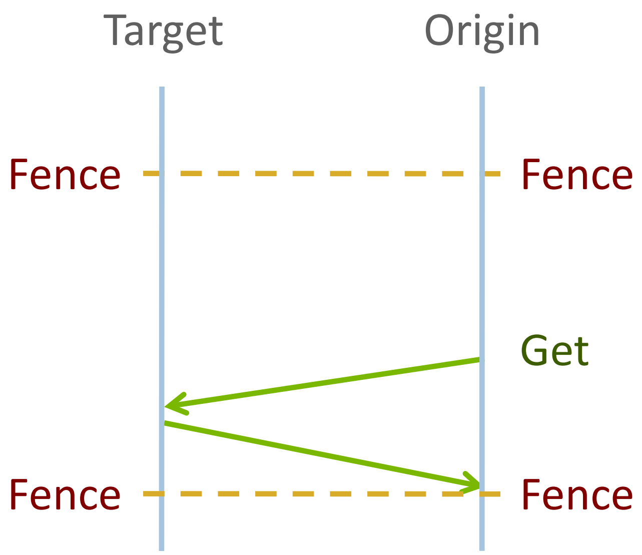
- Request based RMA operations
  - Can wait for single requests
  - Issue a large number of operations and wait for some of them to finish so you can reuse buffers
- Flush
  - Can wait for RMA operations to complete without closing an epoch
  - Lock; put; put; flush; get; get; put; Unlock
- Sync
  -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

RMA Synchronization Models

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

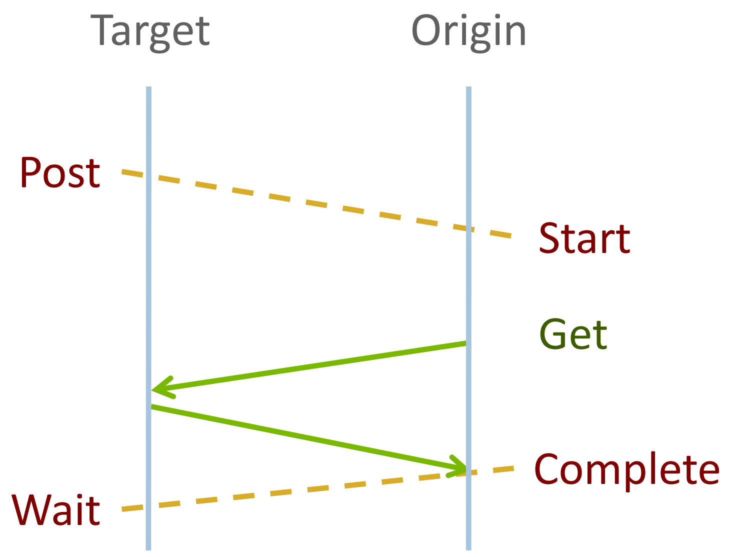
Fence Synchronization

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

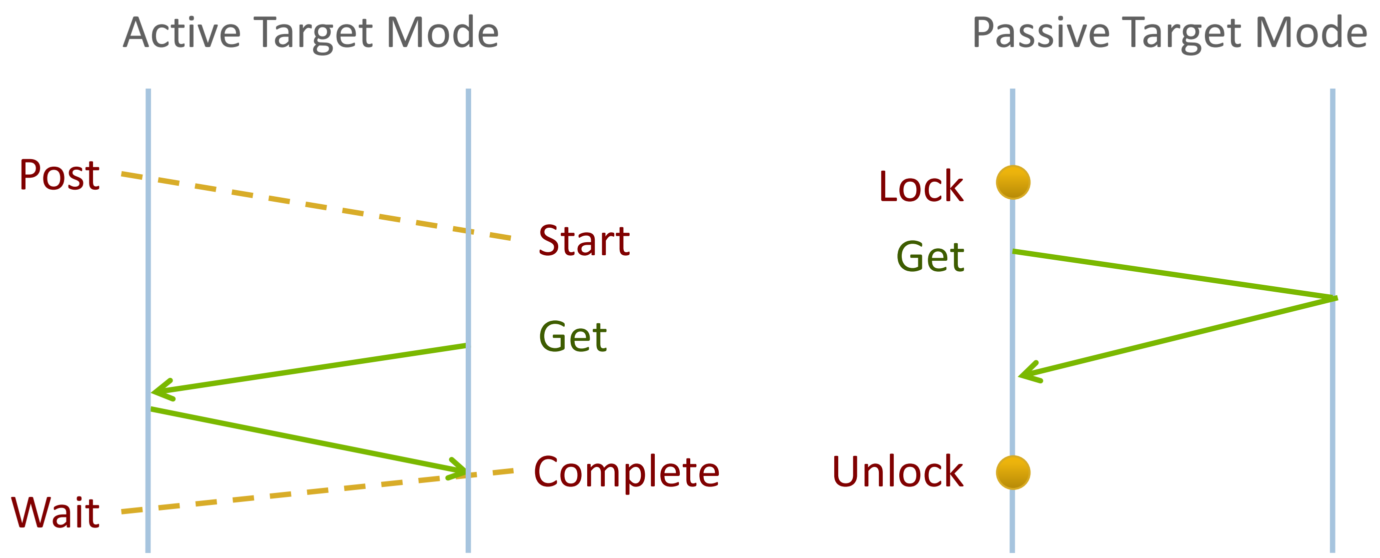


PSCW Synchronization

- 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)
```

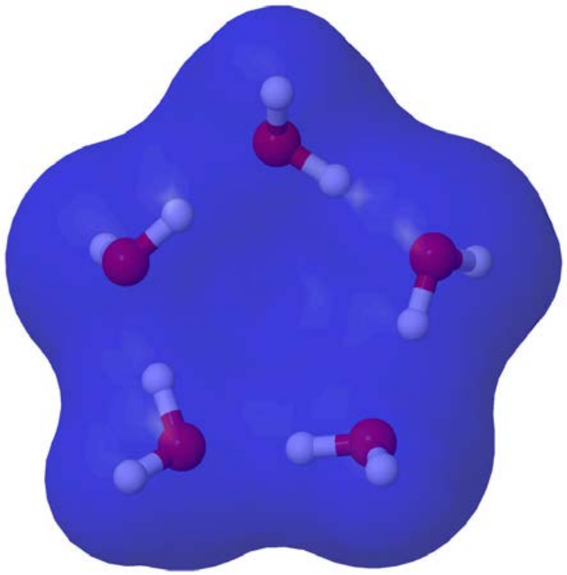
- Begin/end passive mode epoch
  - Doesn’t function like a mutex, name can be confusing
  - Communication operations within epoch are all nonblocking
- Lock type
  - SHARED: Other processes using shared can access concurrently
  - EXCLUSIVE: No other processes can access concurrently

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

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.

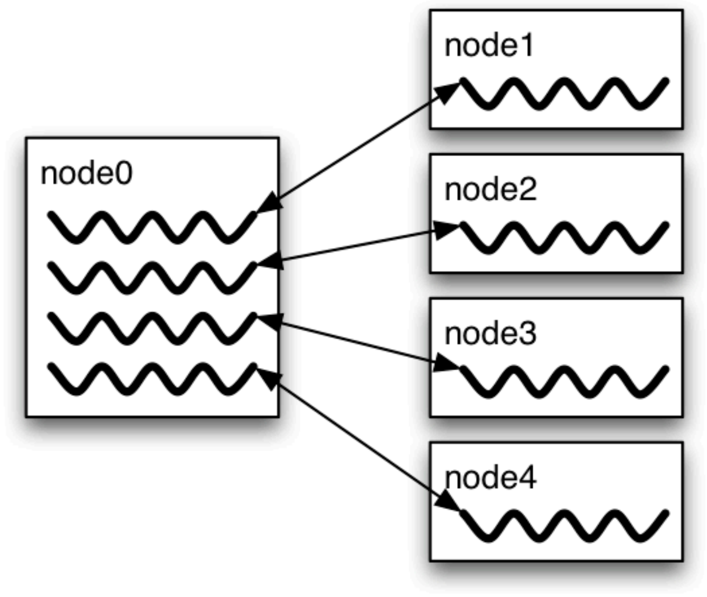
The Current Situation

- 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!

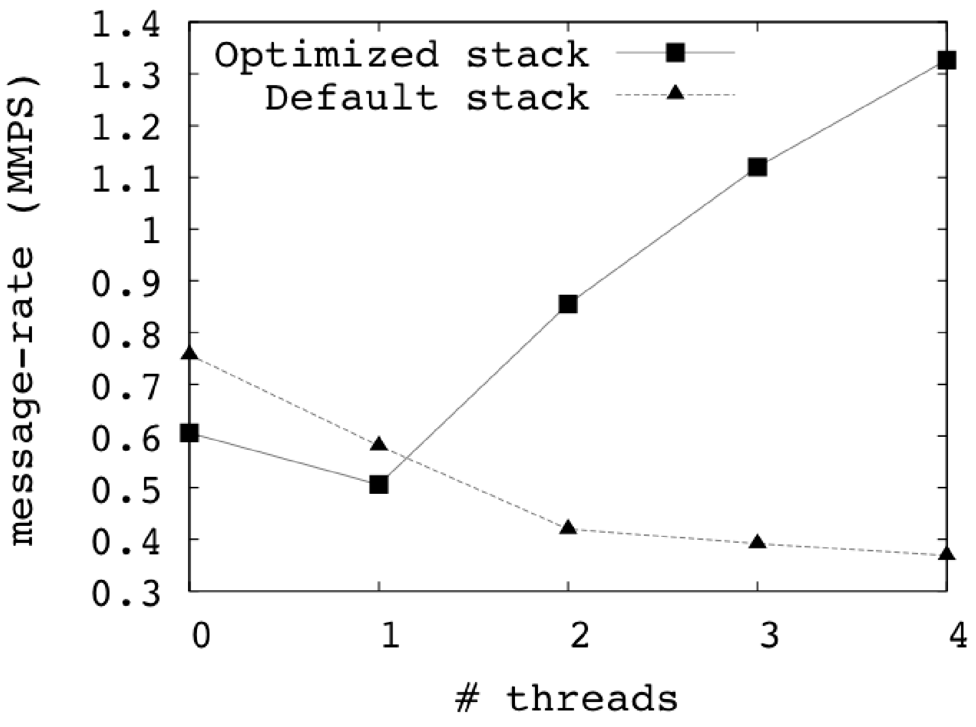
Performance with MPI_THREAD_MULTIPLE

- 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

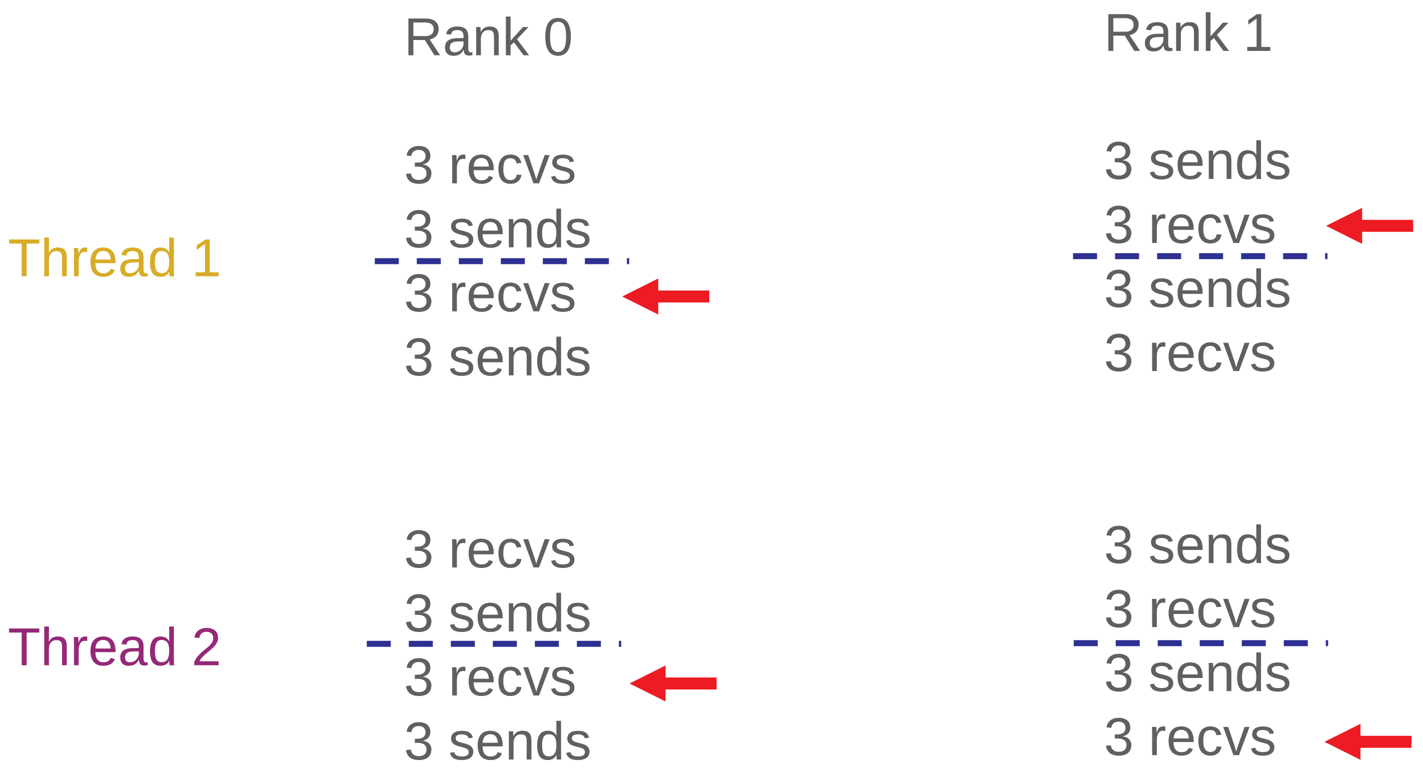
An Example I encountered recently

- 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



- 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

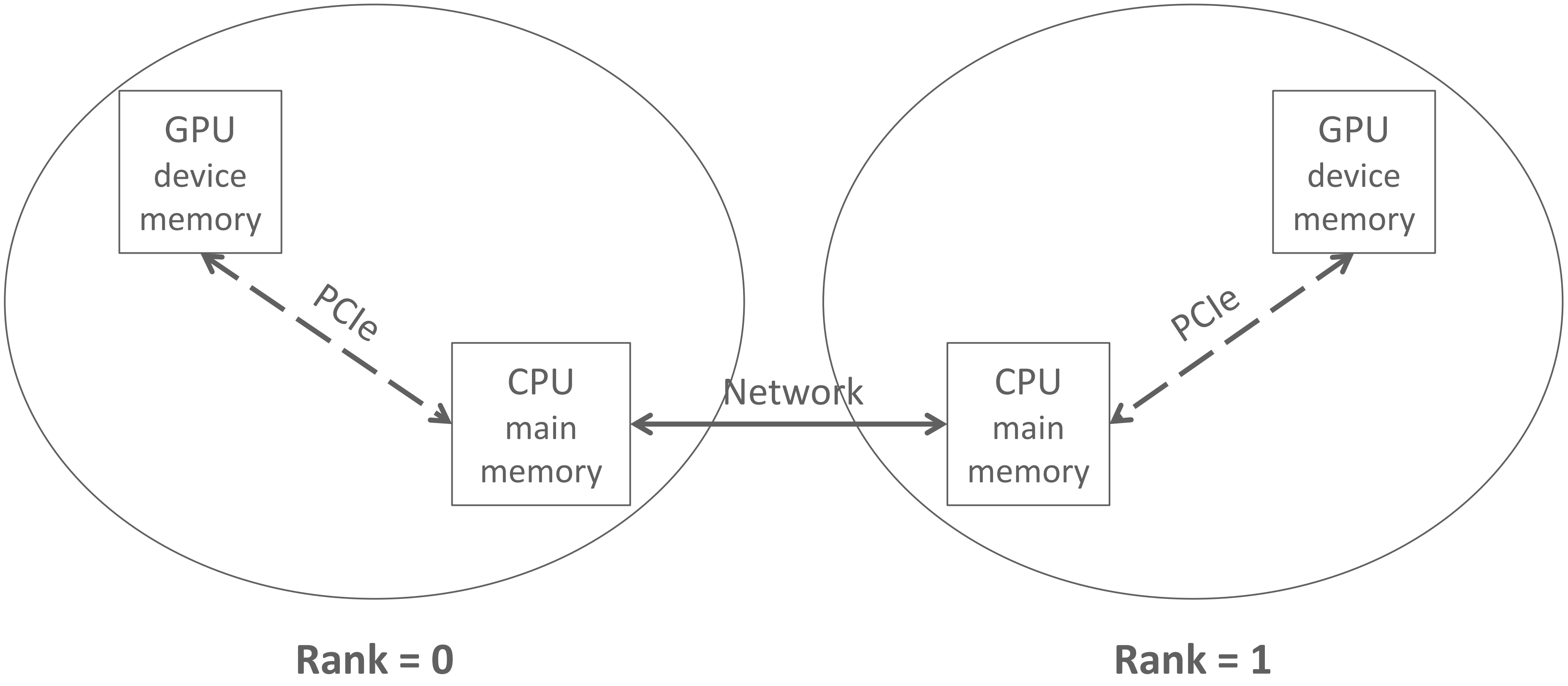
Hybrid Programming with Shared Memory

- 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

Hybrid Programming with GPU models

- Simple GPU interoperability works out of the box
- Many MPI processes
- Each MPI process can launch CUDA/OpenCL/... kernels to compute on data
- Move data back to the process memory
- Use MPI to move data between processes

Interoperability with GPUs: Current Data Model



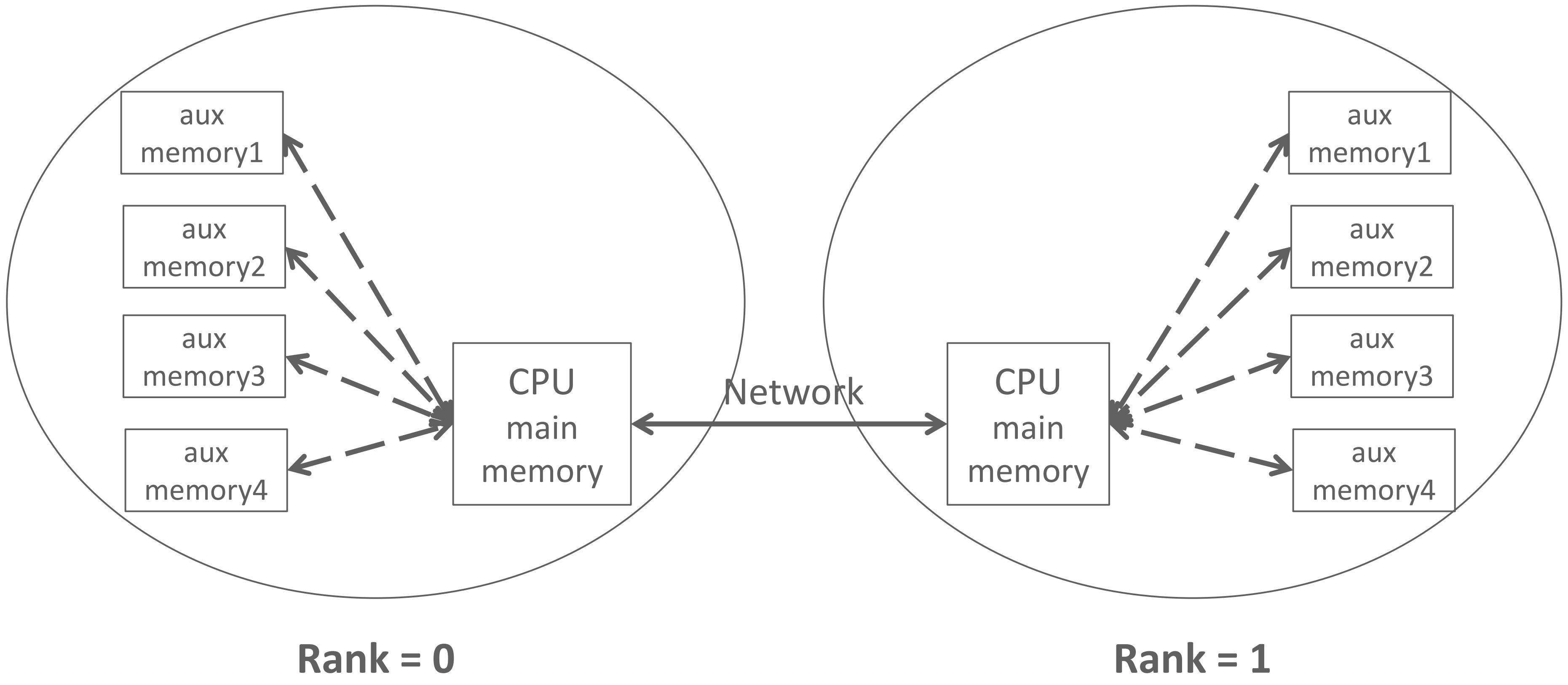
```
if(rank == 0)
{
  cudaMemcpy(s_buf, s_dev_buf, D2H);
  MPI_Send(s_buf, .. ..);
}
```

```
if(rank == 1)
{
  MPI_Recv(r_buf, .. ..);
  cudaMemcpy(r_dev_buf, r_buf, H2D);
}
```

Tighter Interoperability: MPI-ACC (research project)

- Productivity Goal (API)
  - Implement the rich data transfer interface of MPI for CUDA/OpenCL/..
- Performance Goal
  - Pipeline the data movement between GPU memory, host memory and remote node using architecture specific enhancements
    - NVIDIA: GPU Direct
    - Multi-stream copies between GPU and memory (multiple command queues can benefit from parallelism in the DMA engine)
  - Future architectures:
    - Zero-copy data movement if accelerators have direct network access
    - Eliminate "GPU-to-host" data transfers if the heterogeneous processors share memory spaces
- All of the above should happen *automatically* within the MPI implementation, i.e. applications should not redo their data movement for each architecture

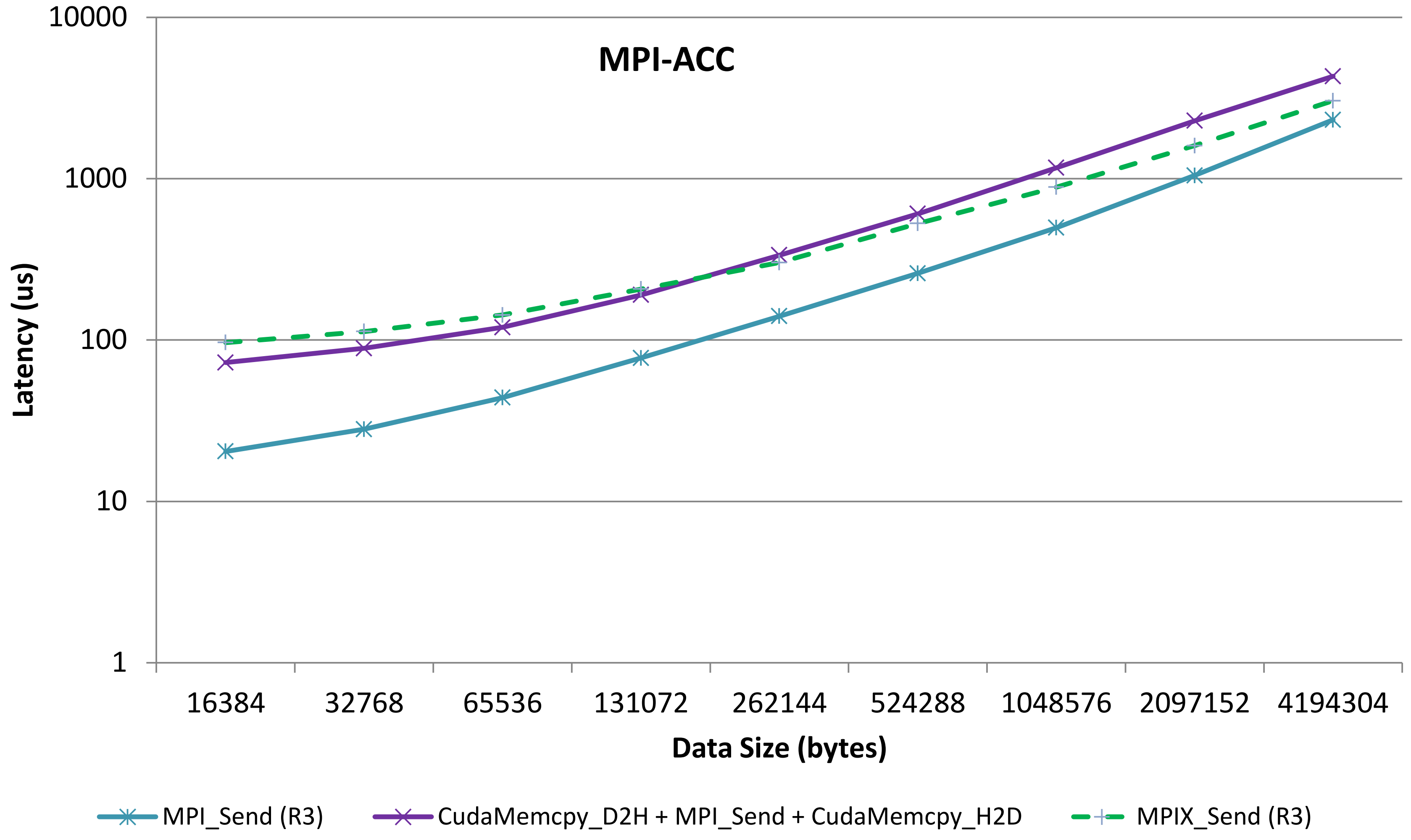
Interoperability with GPUs: New Data Model



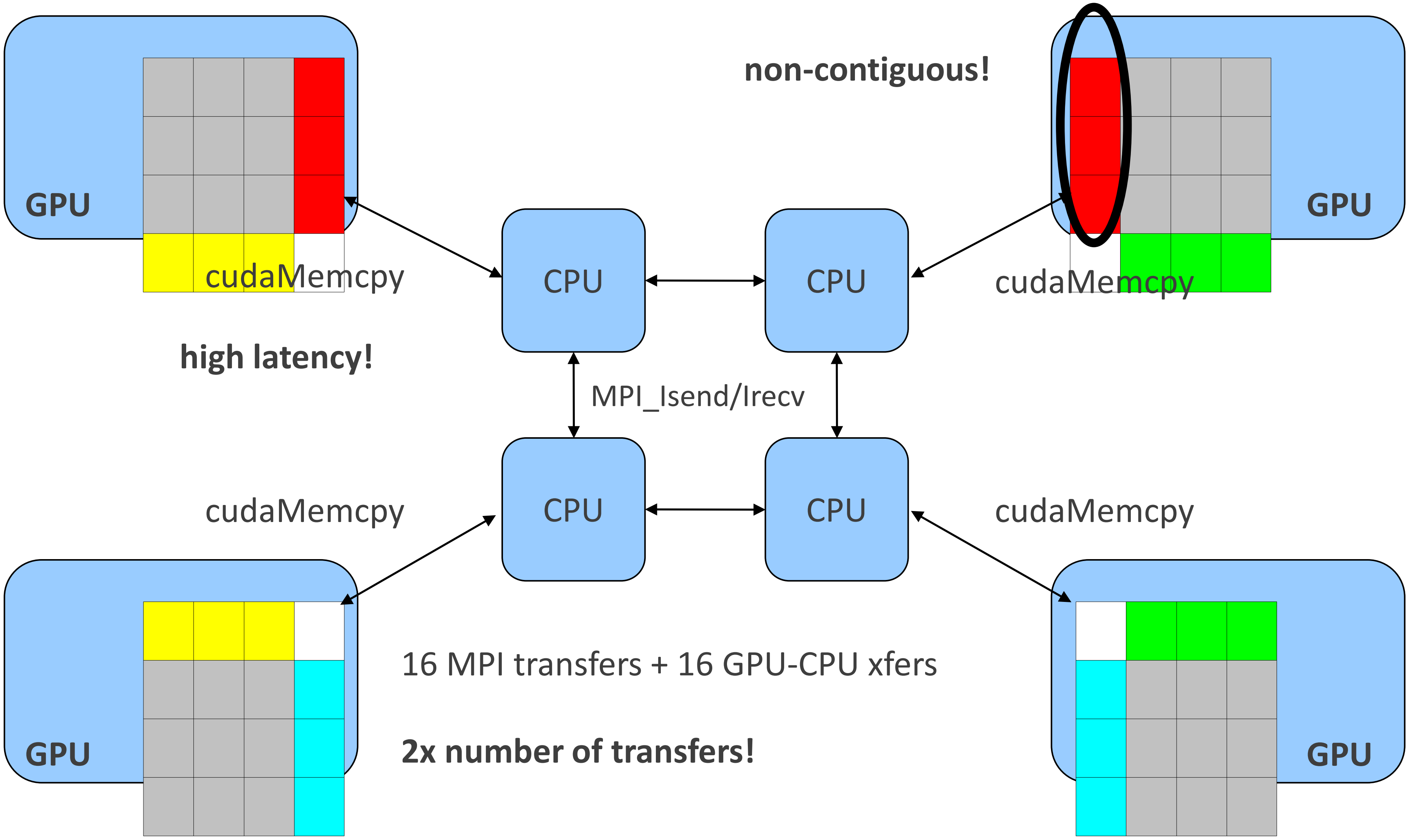
```
if(rank == 0)
{
  MPI_Send(s_aux_buf, .. ..);
}
```

```
if(rank == 1)
{
  MPI_Recv(r_aux_buf, .. ..);
}
```

Experimental Results (CUDA – RNDV mode)



MPI + GPU Example - Stencil Computation

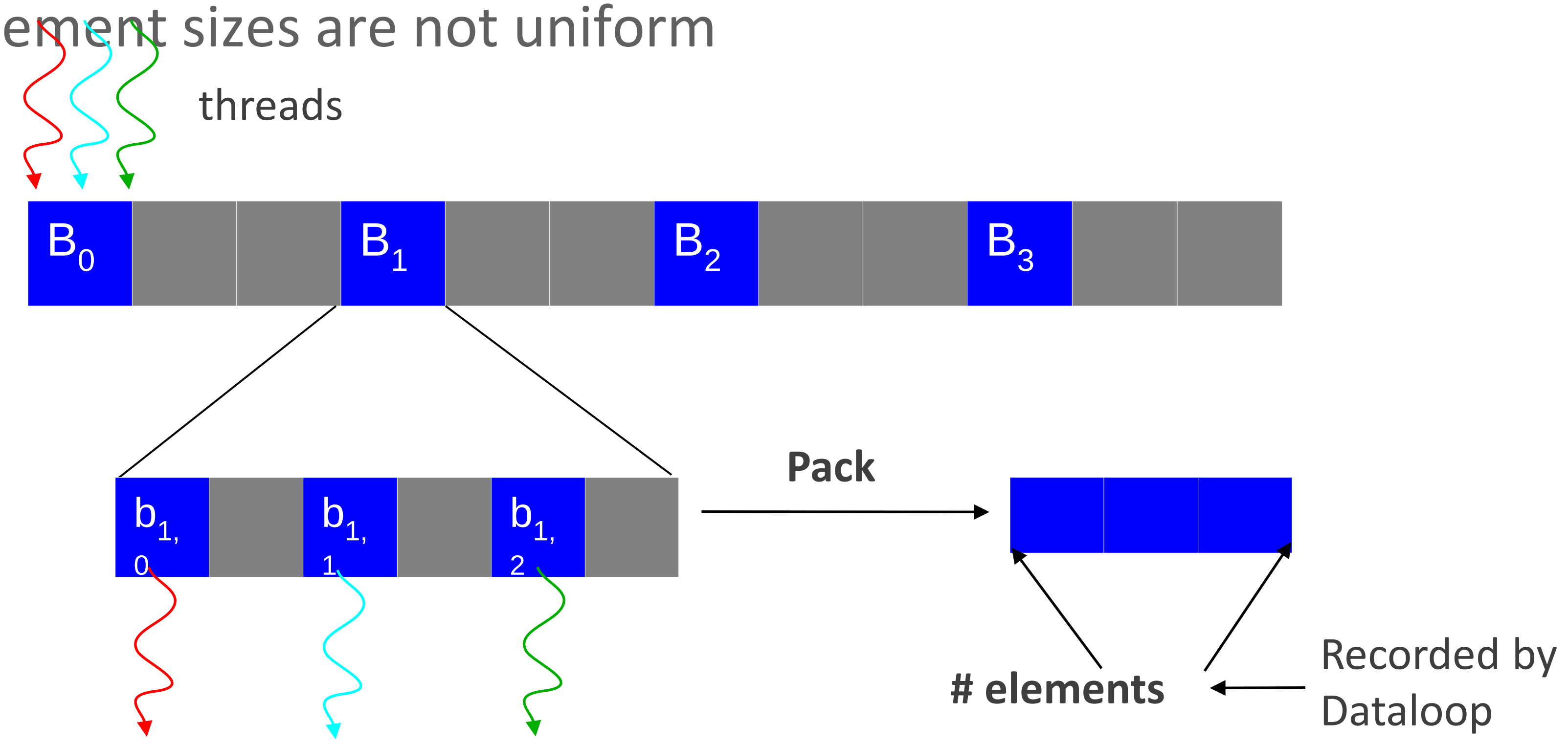


16 MPI transfers + 16 GPU-CPU xfers

2x number of transfers!

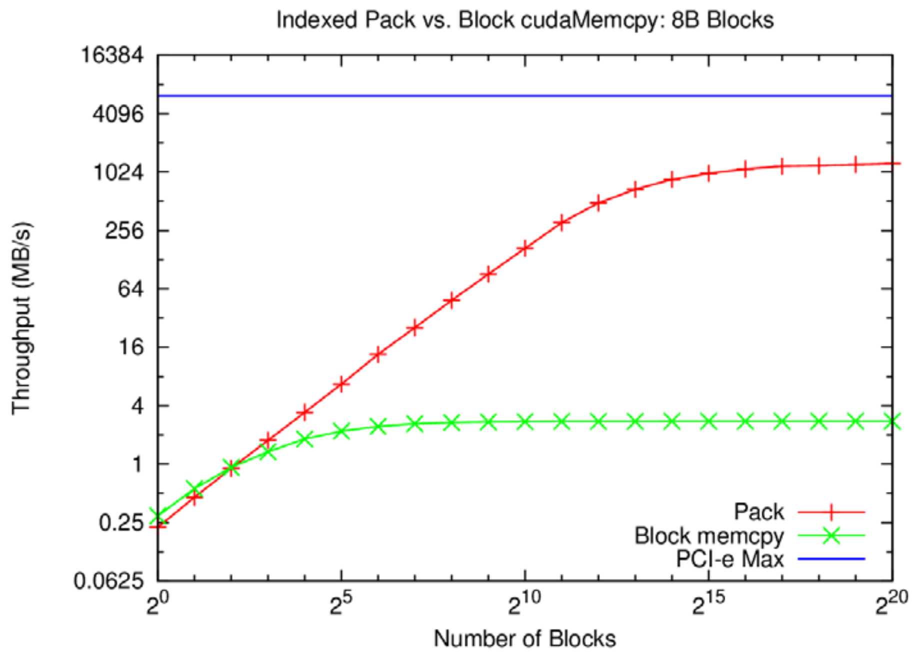
GPU optimizations for Data Packing

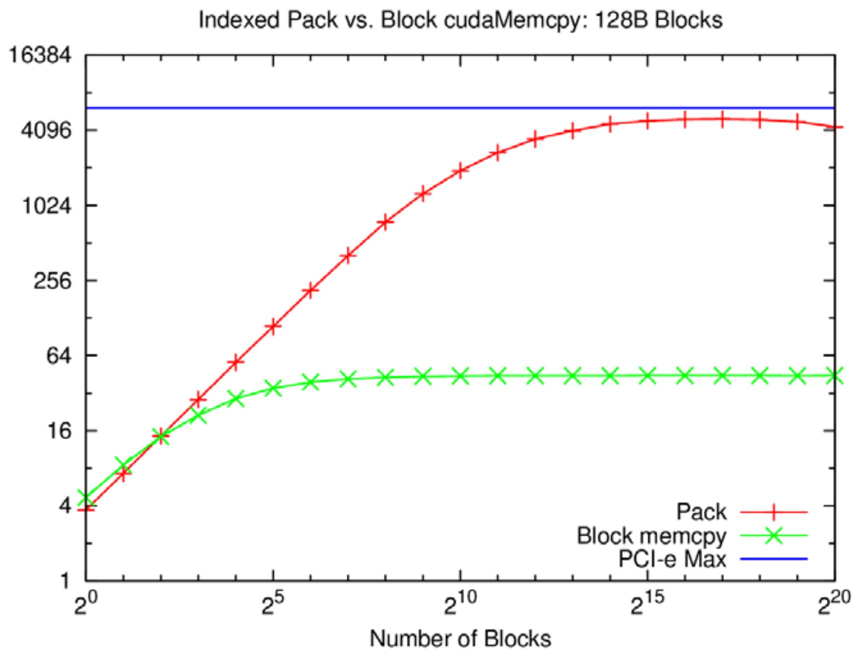
- Element-wise traversal by different threads
- Embarrassingly parallel problem, except for structs, where element sizes are not uniform

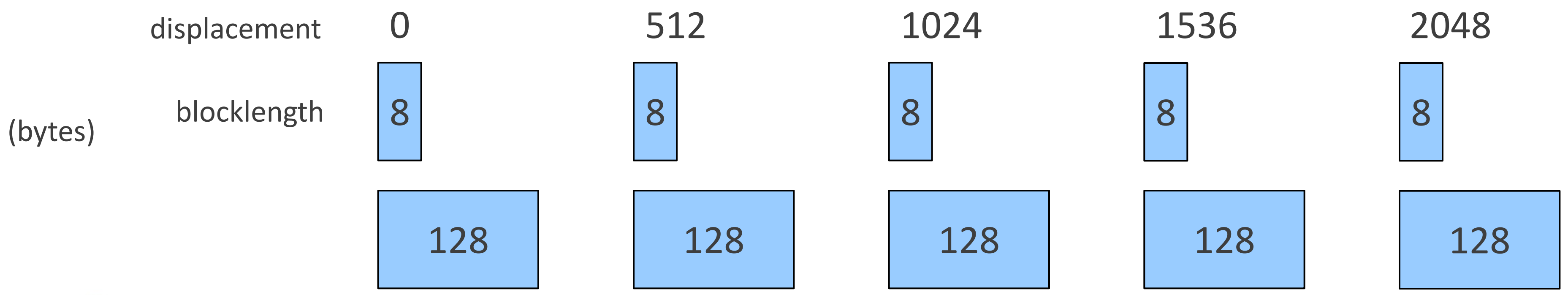


traverse by **element #**, read/write using **extent/size**

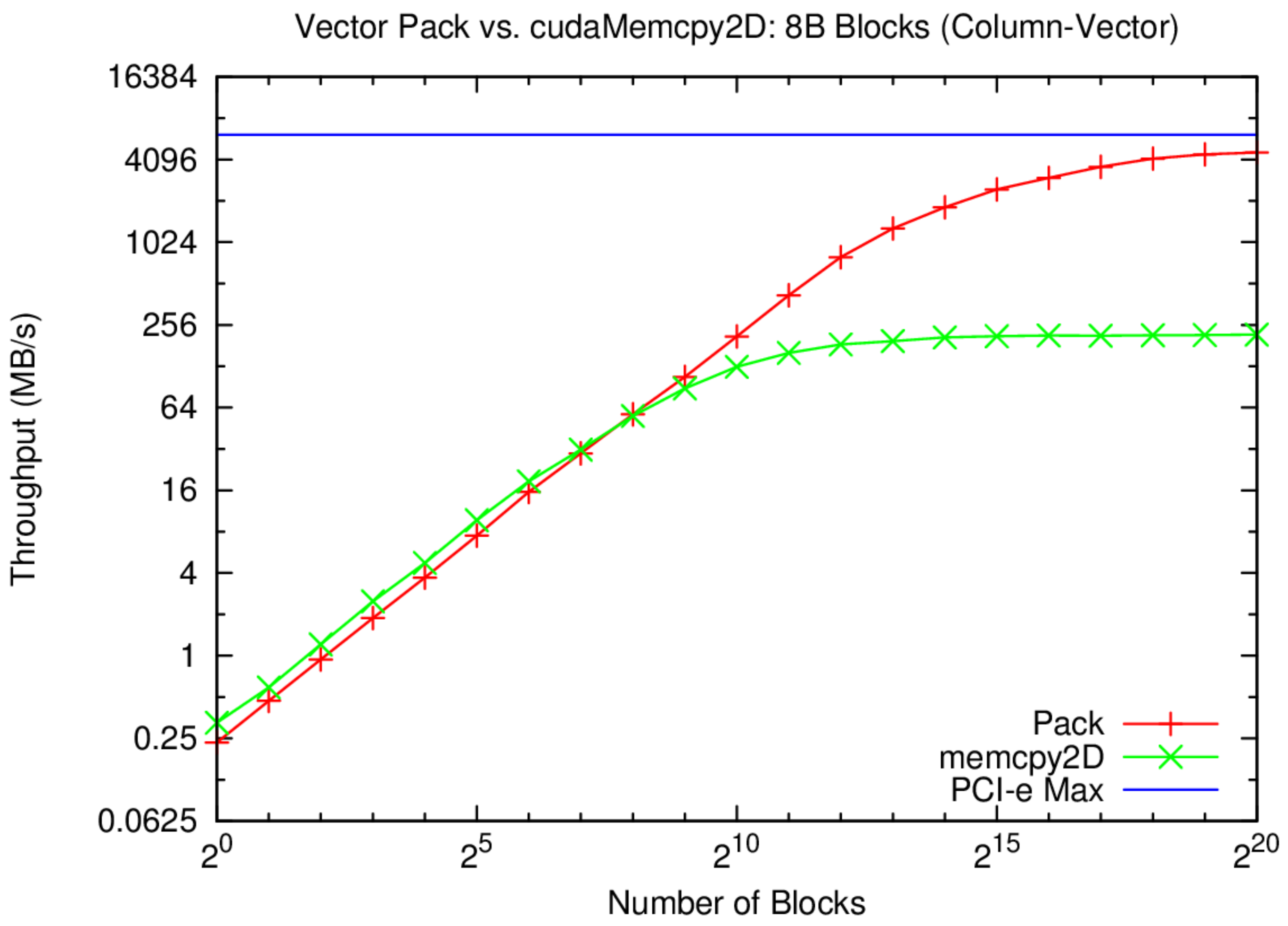
Packing Throughput (Indexed)







Packing Throughput (Column-Vector)



Advanced Topics: Virtual Topology

MPI Virtual Topology

- MPI topology functions:
  - Define the communication topology of the application
    - Logical process arrangement or virtual topology
  - Possibly reorder the processes to efficiently map over the system architecture (physical topology) for more performance
- Virtual topology models:
  - Cartesian topology: multi-dimensional Cartesian arrangement
  - Graph topology: non-specific graph arrangement
- Graph topology representation
  - Non-distributed: easier to manage, less scalable
  - Distributed: new to the standard, more scalable

MPI Graph and Cartesian Topology Functions

- MPI defines a set of virtual topology definition functions for graph and Cartesian structures.
- MPI_Graph_create and MPI_Cart_create non-distributed functions:
  - Are collective calls that accept a virtual topology
  - Return a new MPI communicator enclosing the desired topology
  - The input topology is in a non-distributed form
  - All nodes have a full view of the entire structure
    - Pass the whole information to the function
  - If the user opts for reordering, the function may reorder the ranks for an efficient process-to-core mapping.

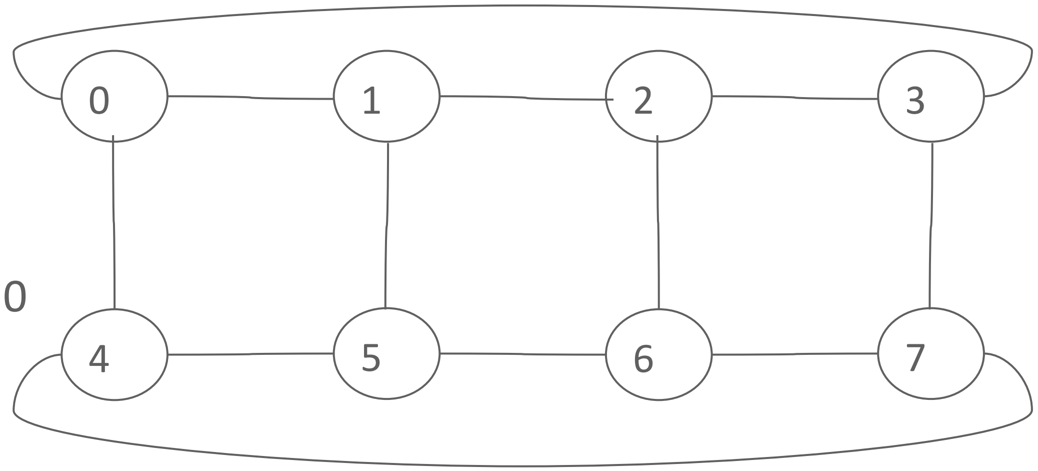
MPI Graph and Cartesian Topology Functions (II)

- **MPI_Cart_create(***comm_old*, *ndims*, *dims*, *periods*, *reorder*, *comm_cart* **)**

– *comm_old*	[in] input communicator without topology (handle)
– *ndims*	[in] number of dimensions of Cartesian grid (integer)
– *dims*	[in] integer array of size ndims specifying the number of processes in each dimension
– periods	[in] logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension
– reorder	[in] ranking may be reordered (true) or not (false) (logical)
– comm_graph	[out] communicator with Cartesian topology (handle)

Dimension	#Processes
1	4
2	2

ndims = 2
dims = 4, 2
periods = 1, 0



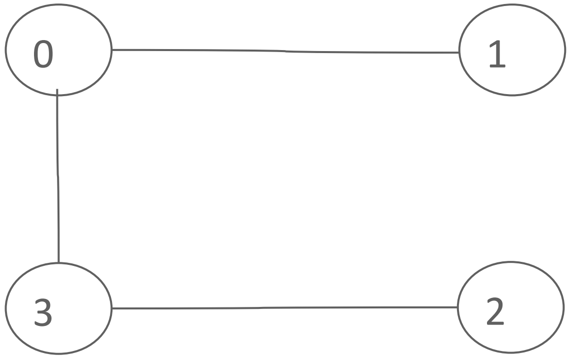
4x2 2D-Torus

MPI Graph and Cartesian Topology Functions (III)

- **MPI_Graph_create(***comm_old*, *nnodes*, *index*, *edges*, *reorder*, *comm_graph* **)**
  - *comm_old* [in] input communicator without topology (handle)
  - *nnodes* [in] number of nodes in graph (integer)
  - *index* [in] array of integers describing node degrees
  - edges [in] array of integers describing graph edges
  - reorder [in] ranking may be reordered (true) or not (false) (logical)
  - comm_graph [out] communicator with graph topology added (handle)

Process	Neighbors
0	1, 3
1	0
2	3
3	0, 2

nnodes = 4
index = 2, 3, 4, 6
edges = 1, 3, 0, 3, 0, 2



Design of MPI Topology Functions (I)

- Both Cartesian and graph interfaces are treated as graph at the underlying layers
  - Cartesian topology is internally copied to a graph topology
- Virtual topology graph:
  - Vertices: MPI *processes*
  - Edges: existence, or significance, of communication between any two processes
  - Significance of communication : normalized total communication volume between any pair of processes, used as edge weights
  - Edge replication is used to represent graph edge weight
    - Recap: MPI non-distributed interface does not support weighted edges

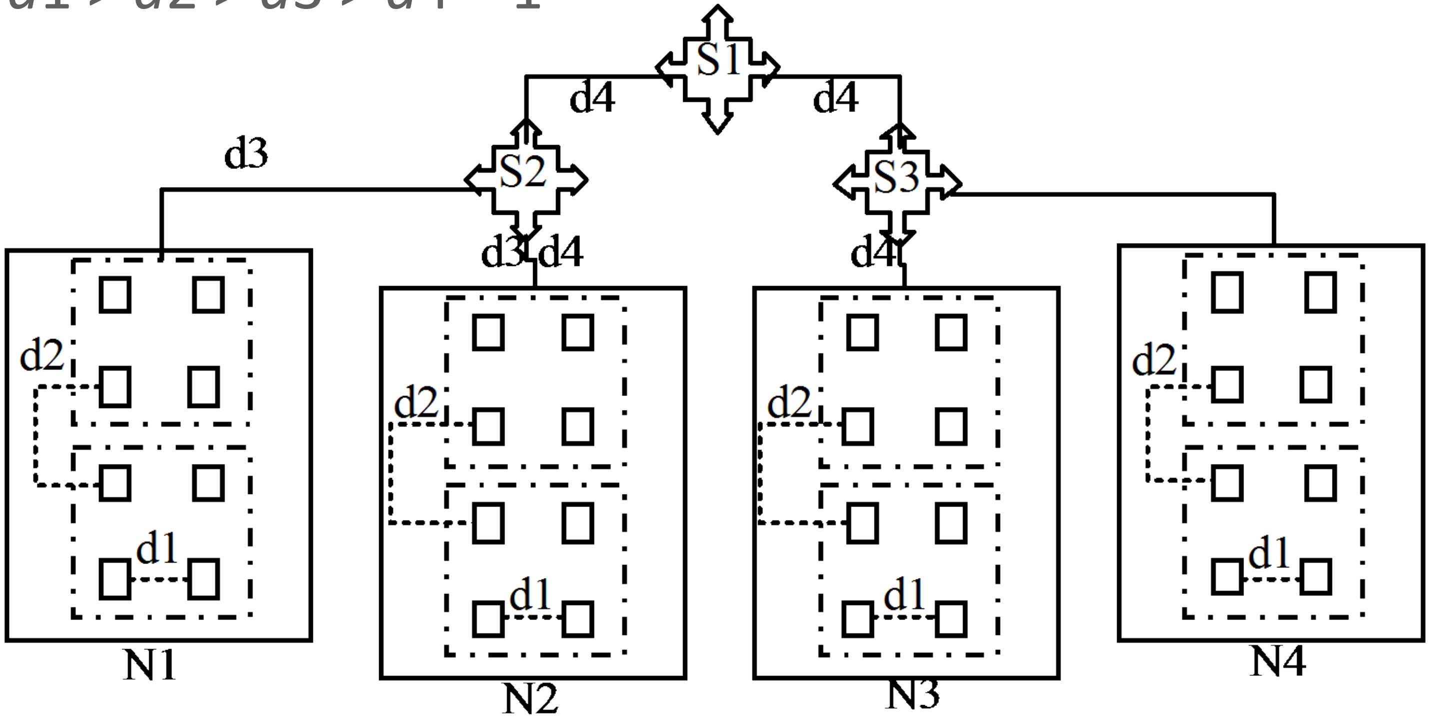
Design of MPI Topology Functions (II)

- Physical topology graph:
  - Integrated node and network architecture
  - Vertices: architectural components such as:
    - *Network nodes*
    - *Cores*
    - *Caches*
  - Edges: communication *links* between the components
  - Edge weights: communication performance between components
    - Processor cores: closer cores have higher edge weight
    - Network nodes: closer nodes have higher edge weight
    - Farthest on-node cores get higher weight than closest network nodes

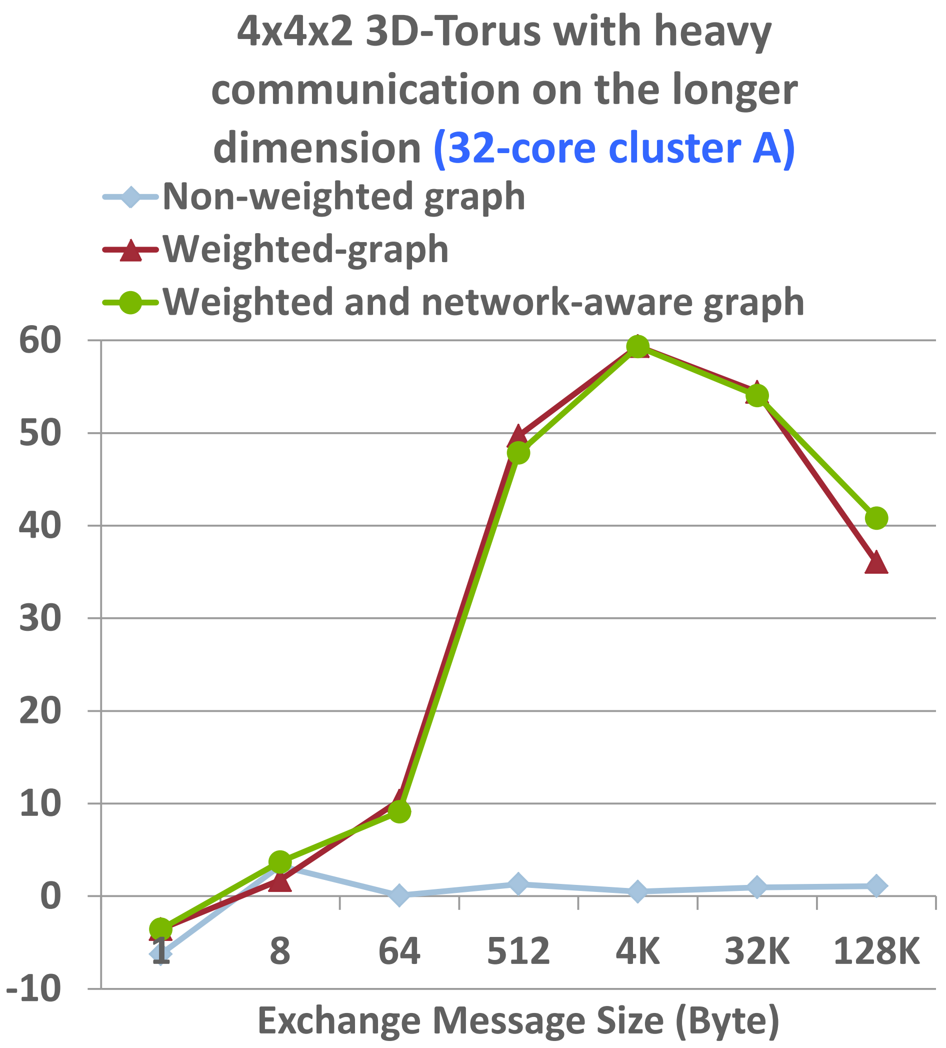
Physical Topology Distance Example

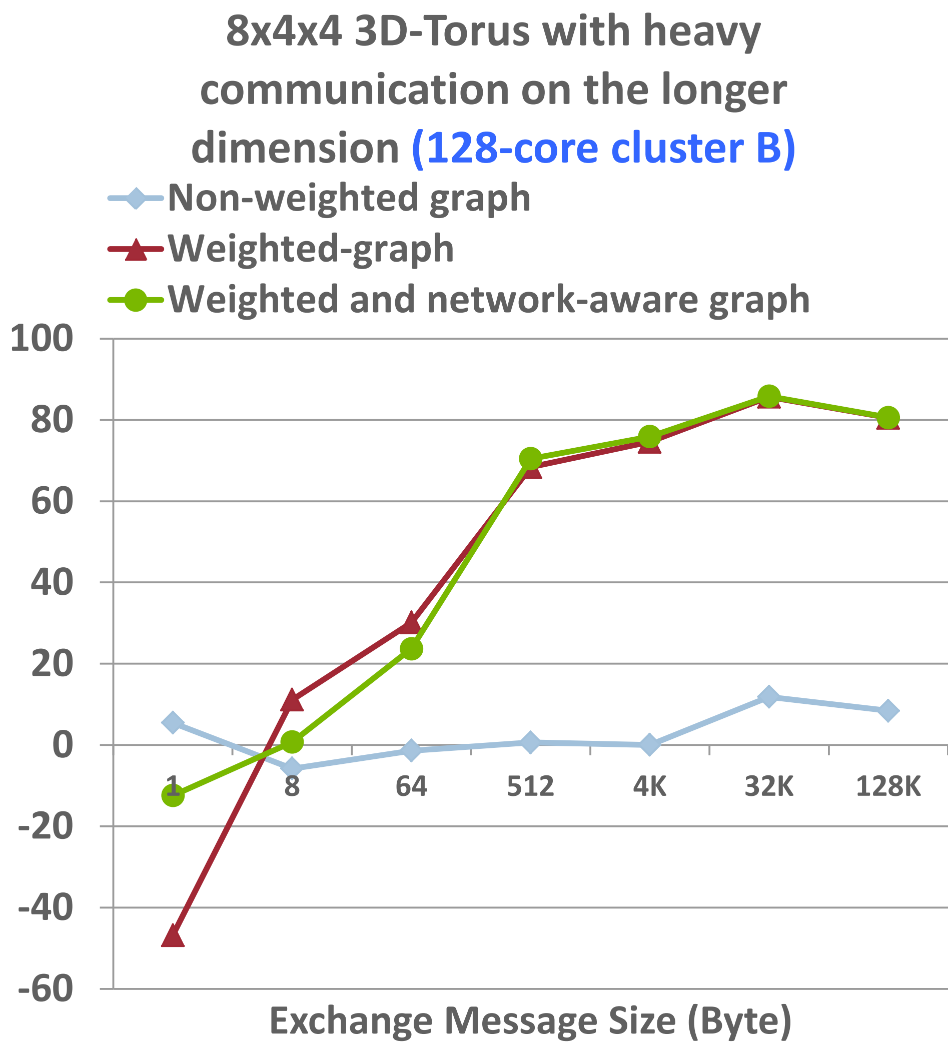
- *d*1 will have the highest load value in the graph.
- The path between *N*2 and *N*3 (*d*4) will have the lowest load value, indicating the lowest performance path.

➔ $d1 > d2 > d3 > d4 = 1$



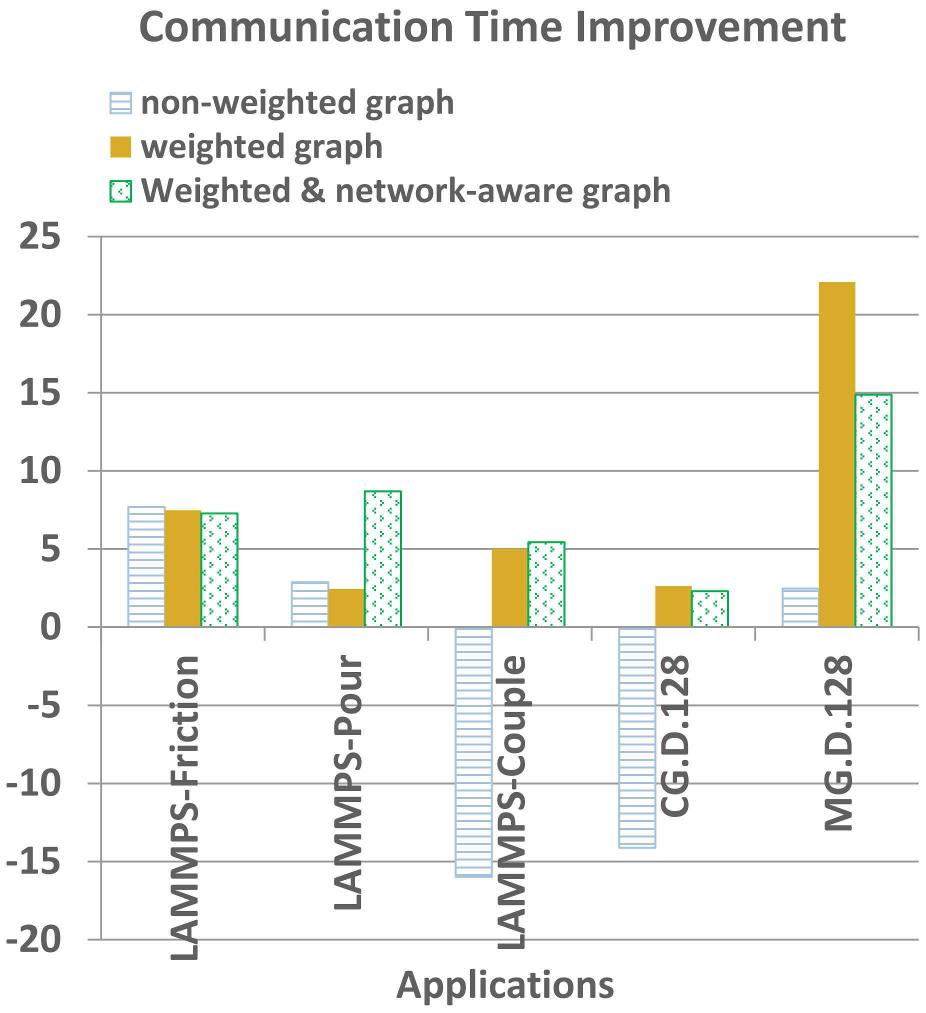
Exchange Micro-benchmark: Topology-aware Mapping Improvement over Block Mapping (%)

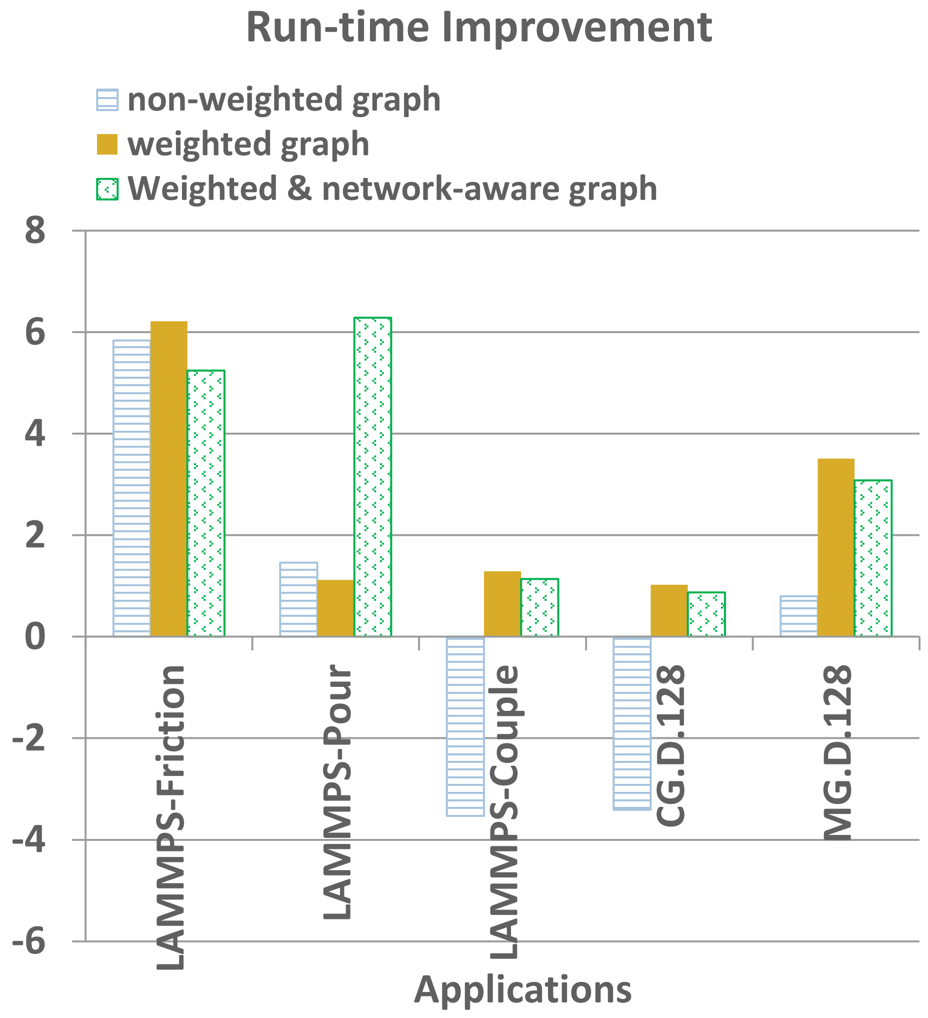




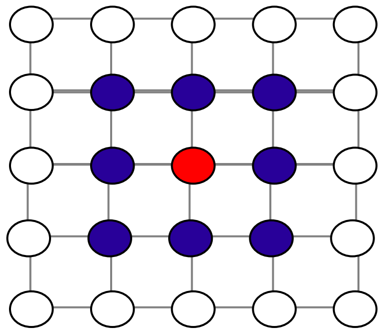
Applications: Topology-aware Mapping Improvement over Block Mapping (%)

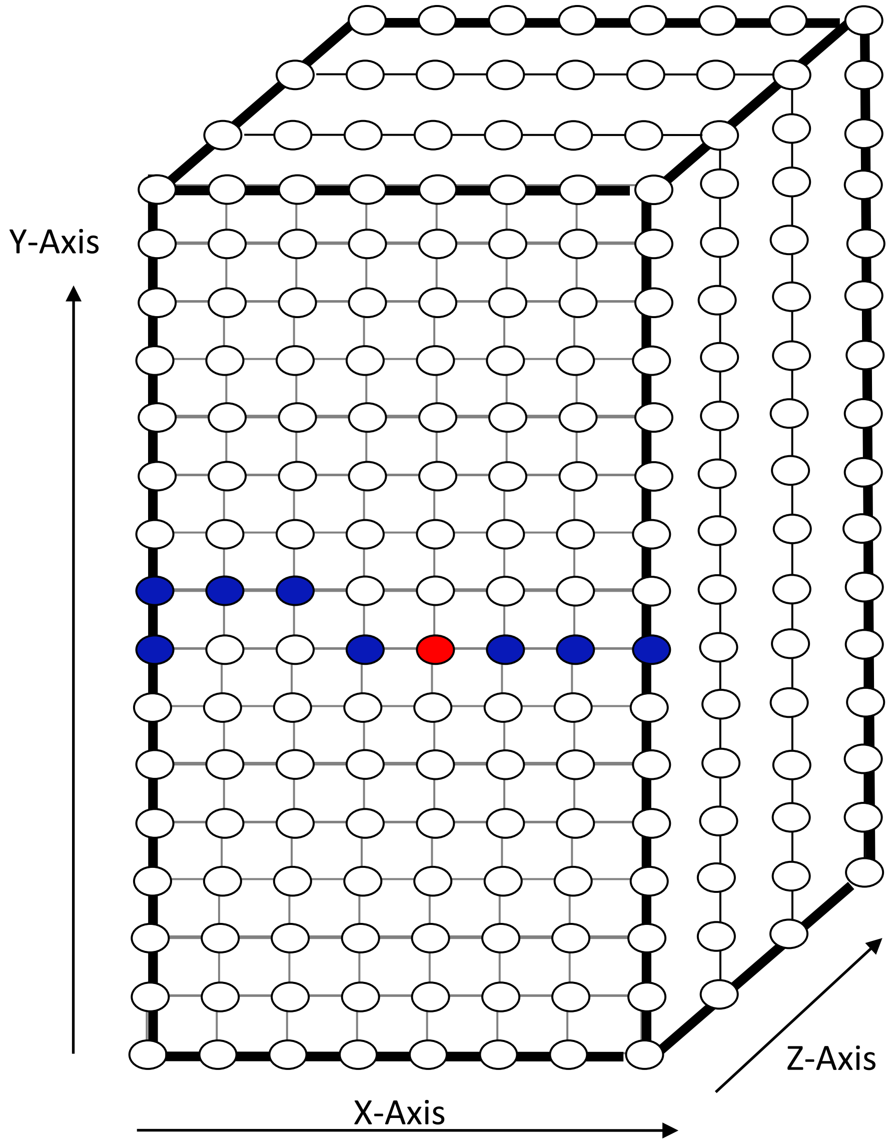
128-core cluster B



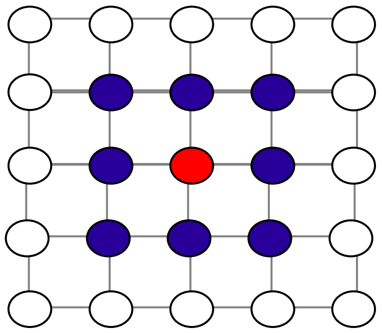


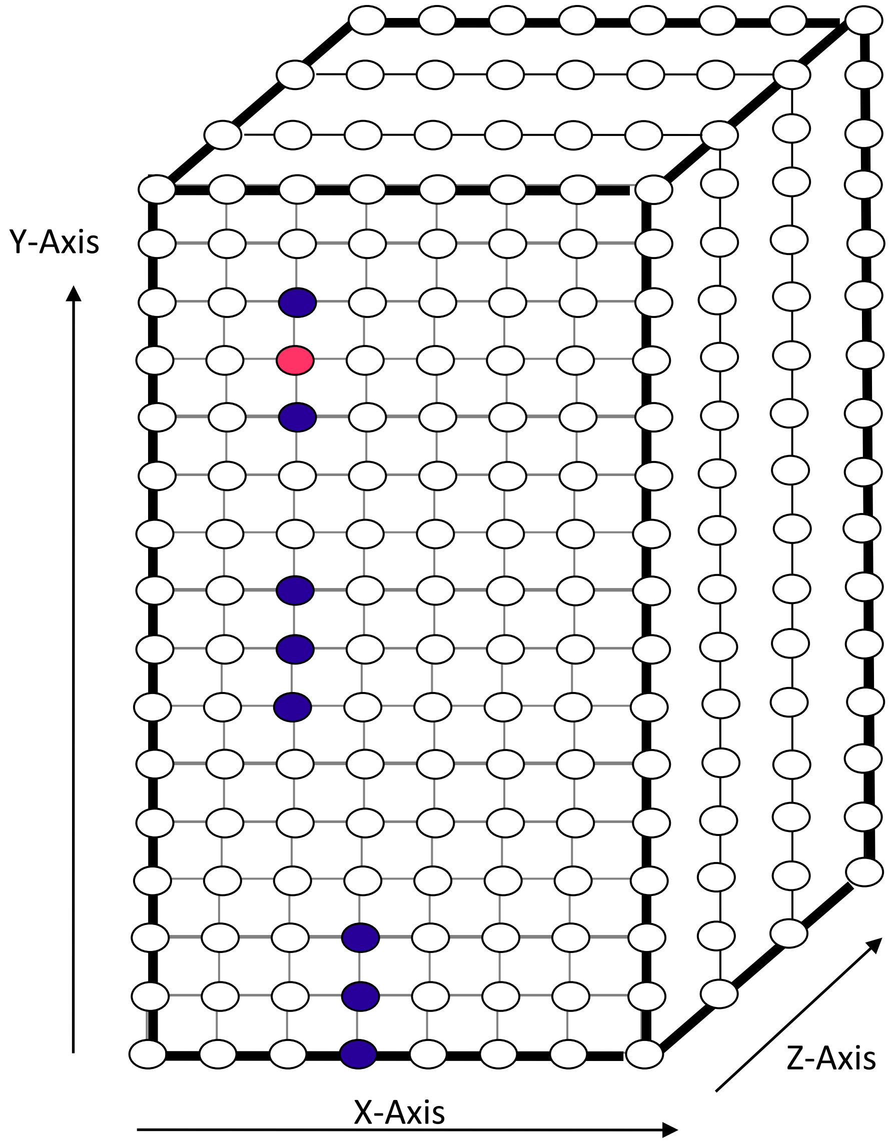
2D Nearest Neighbor: Process Mapping (XYZ)



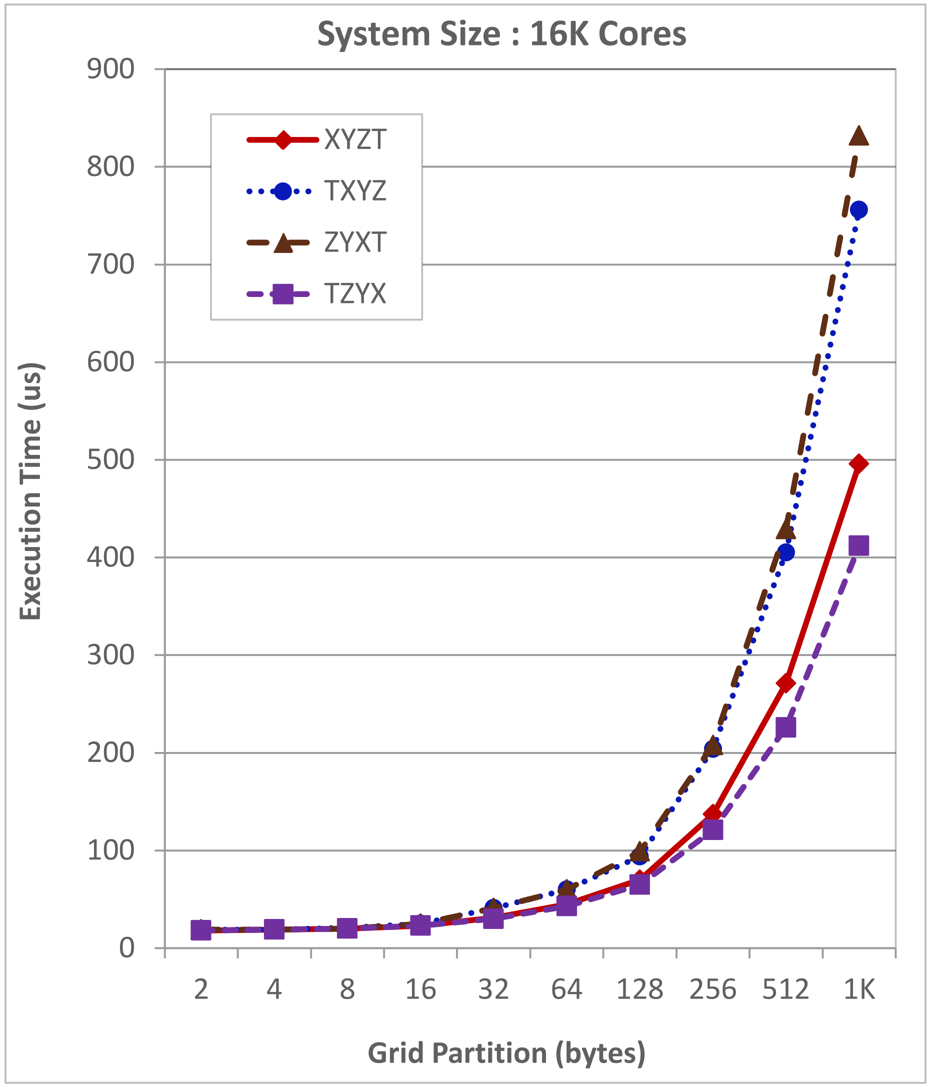


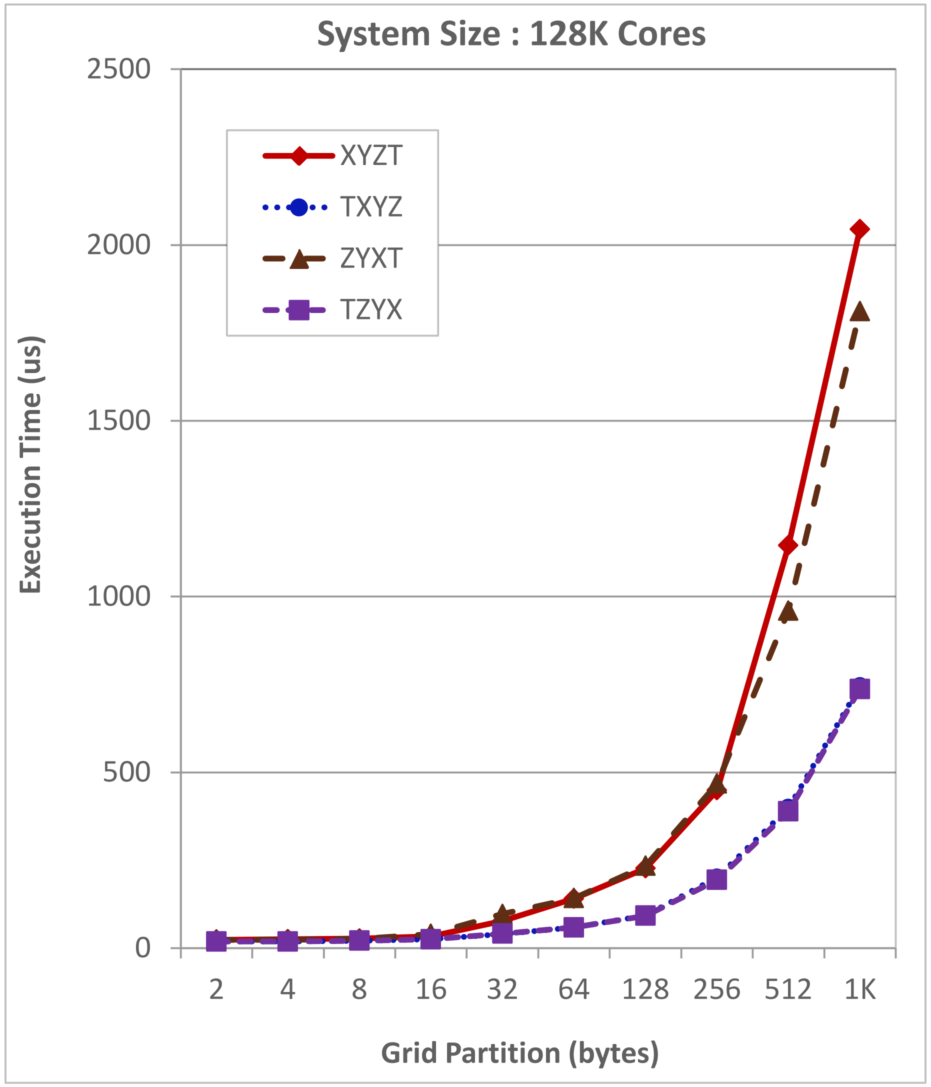
2D Nearest Neighbor: Process Mapping (YXZ)





Nearest Neighbor Performance





Physical Topology Information Retrieval

- Virtual topology functionality relies on the user providing MPI with the application communication pattern
- What about work-stealing applications? Communication is pretty random
- MPI-3 introduced a new function called MPI_Comm_split_type
  - Idea is to split a communicator based on some physical hardware information
  - E.g., you can split a communicator to contain processes that can create a shared memory region
  - Implementations can extend it to allow any form of creation – same node, same NUMA socket, same cache domain, same switch, same rack

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
- MPICH2 : http://www.mcs.anl.gov/research/projects/mpich2/
- MPICH mailing list: mpich-discuss@mcs.anl.gov
- MPI Forum : http://www.mpi-forum.org/

- Other MPI implementations:
  - IBM MPI (MPICH for BG)
  - Cray MPI (MPICH for Cray)
  - MVAPICH2 (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