

# MPI for Scalable Computing

William Gropp<sup>1</sup> Rajeev Thakur<sup>2</sup> Pavan Balaji<sup>2</sup>

<sup>1</sup>University of Illinois

<sup>2</sup>Argonne National Laboratory

# The MPI Part of ATPESC

- We assume everyone already has some MPI experience
- We will focus more on understanding MPI concepts than on coding details
- Emphasis will be on issues affecting scalability and performance
- There will be code walkthroughs and hands-on exercises

# Outline

- Morning
  - Introduction to MPI
  - Performance issues in MPI programs
  - Sources of scalability problems
  - Avoiding communication delays
    - understanding synchronization
  - Minimizing data motion
    - using MPI datatypes
  - Topics in collective communication
  - Hands-on exercises
- Afternoon (may start earlier)
  - One-sided communication (or remote memory access)
    - Hands-on exercises
  - Hybrid programming
  - Process topologies and neighborhood collectives
- After dinner
  - Hands-on exercises

# What is MPI?

- MPI is a message-passing library interface standard.
  - Specification, not implementation
  - Library, not a language
  - Classical message-passing programming model
- MPI-1 was defined (1994) by a broadly-based group of parallel computer vendors, computer scientists, and applications developers.
  - 2-year intensive process
- Implementations appeared quickly and now MPI is taken for granted as vendor-supported software on any parallel machine.
- Free, portable implementations exist for clusters and other environments (MPICH, Open MPI)



# Timeline of the MPI Standard

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

# Status of MPI-3.1 Implementations

	MPICH	MVAPICH	Open MPI	Cray	Tianhe	Intel		IBM			HPE	Fujitsu	IMS	MPC	NEC	Sunway	RIKEN	AMPI
						IMPI	MPICH-OFI	BG/Q (legacy) <sup>1</sup>	PE (legacy) <sup>2</sup>	Spectrum								
NBC																		
Nbr. Coll.													X					
RMA													(*)					Q2 '18
Shr. mem																		Q1 '18
MPI_T													*					Q2 '18
Comm-create group													*					
F08 Bindings									X			X	X					Q2 '18
New Dtypes																		
Large Counts																		
MProbe																		Q1 '18
NBC I/O					X			X	X			X	X	*		X		Q3 '18

Release dates are estimates; subject to change at any time

"X" indicates no publicly announced plan to support that feature

Platform-specific restrictions might apply to the supported features

<sup>1</sup> Open Source but unsupported

<sup>2</sup> No MPI\_T variables exposed

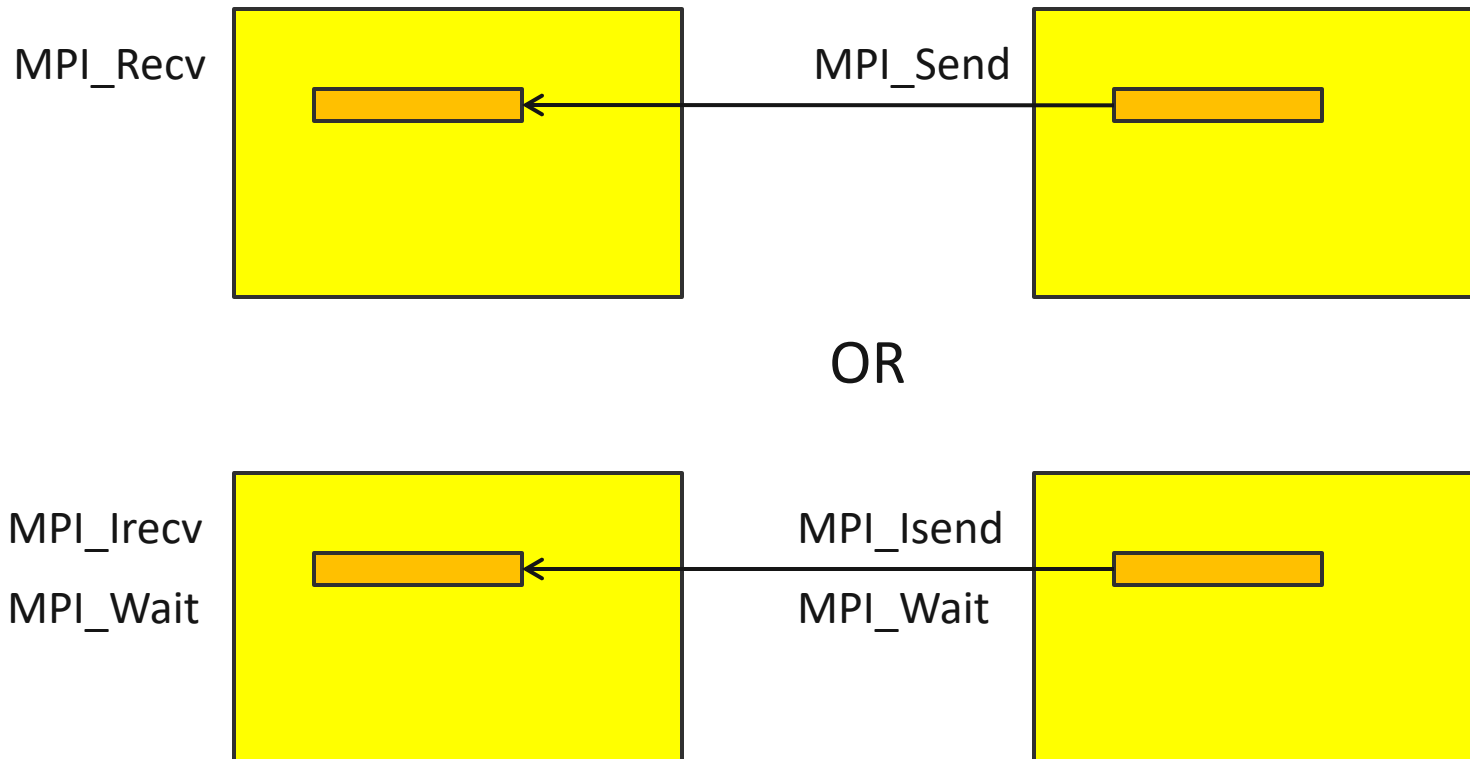
\* Under development

(\*) Partly done

# Important considerations while using MPI

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

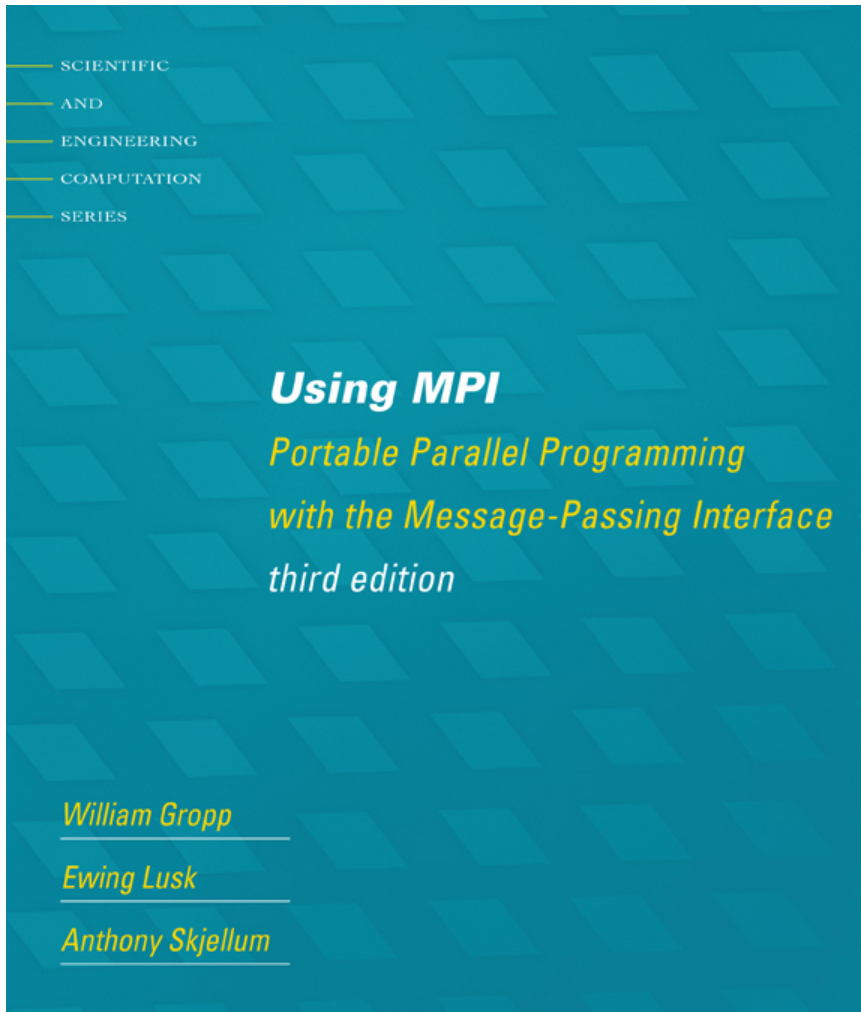
# Basic MPI Communication



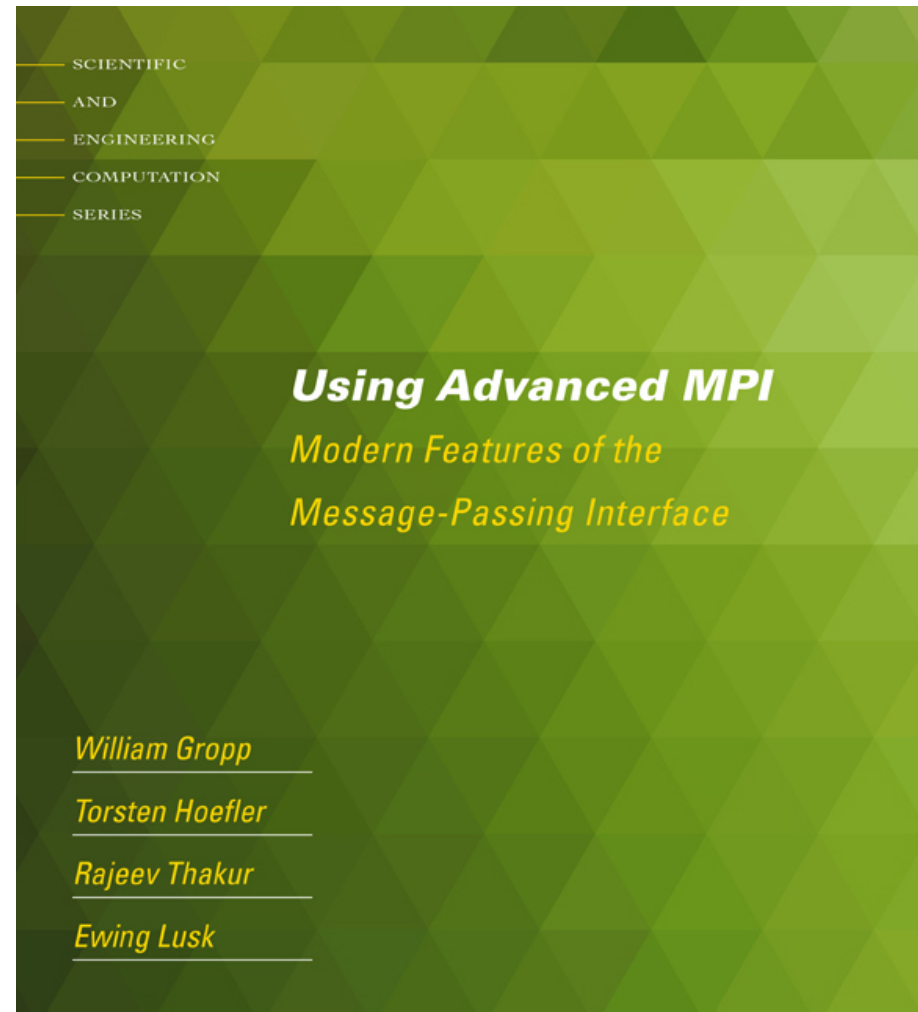
# Web Pointers

- MPI Standard : <http://www.mpi-forum.org/docs/docs.html>
- MPI Forum : <http://www.mpi-forum.org/>
- MPI implementations:
  - MPICH : <http://www.mpich.org>
  - MVAPICH : <http://mvapich.cse.ohio-state.edu/>
  - Intel MPI: <http://software.intel.com/en-us/intel-mpi-library/>
  - Microsoft MPI: <https://msdn.microsoft.com/en-us/library/bb524831%28v=vs.85%29.aspx>
  - Open MPI : <http://www.open-mpi.org/>
  - IBM MPI, Cray MPI, HP MPI, TH MPI, ...
- Several MPI tutorials can be found on the web

# Tutorial Books on MPI (November 2014)



**Basic MPI**

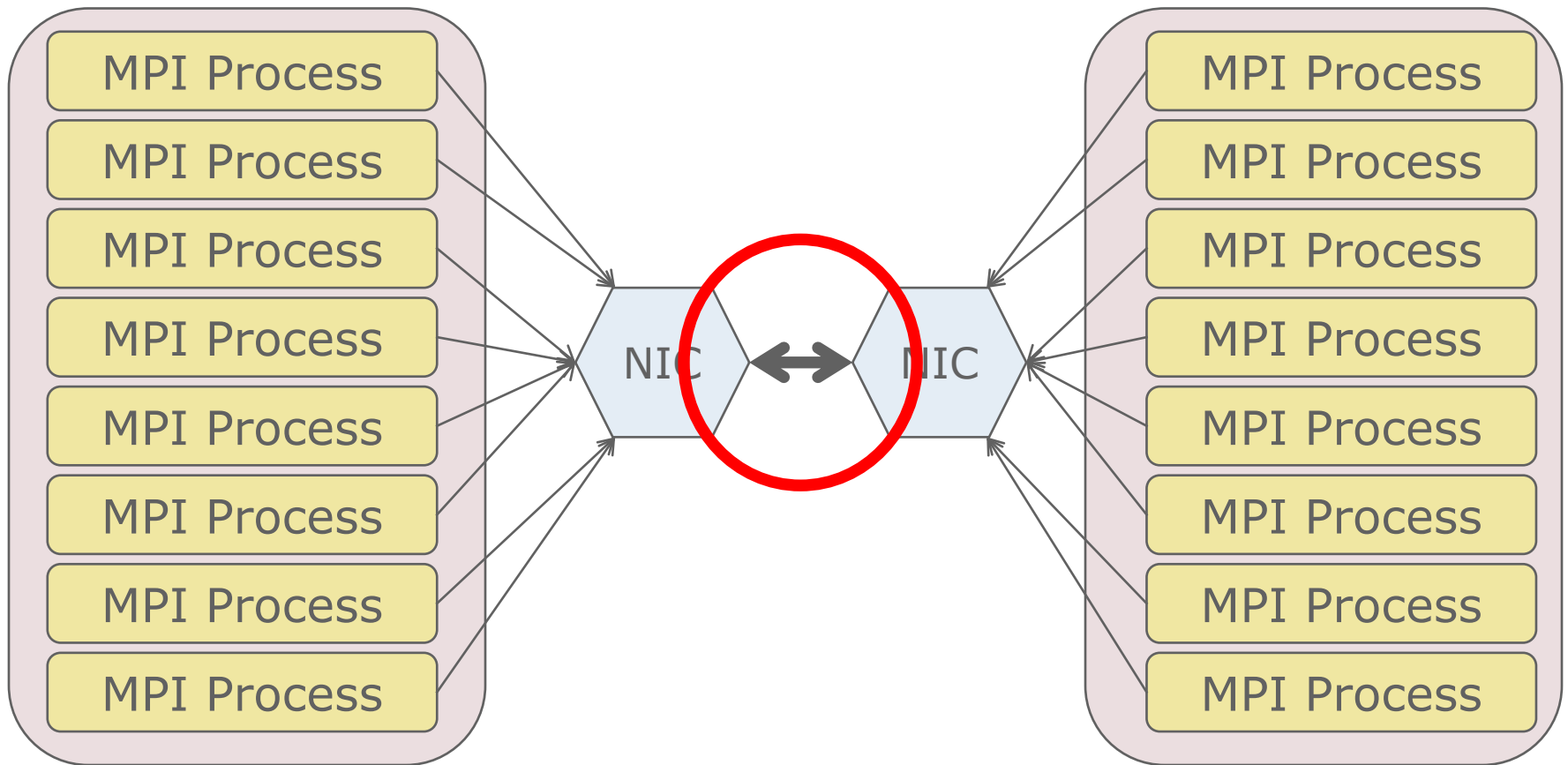


**Advanced MPI, including MPI-2 and MPI-3**

# Understanding MPI Performance on Modern Processors

- MPI was developed when a single processor required multiple chips and most processors and nodes had a single core.
- Building effective, scalable applications requires having a model of how the system executes, how it performs, and what operations it can perform
  - This is (roughly) the *execution model* for the system, along with a *performance model*
- For decades, a simple model worked for designing and understanding MPI programs
  - Programs communicate either with point-to-point communication (send/recv), with a performance model of  $T = s + r n$ , where  $s$  is latency (startup) and  $r$  is inverse bandwidth (rate), or collective communication
- But today, processors are multi-core and many nodes are multi-chip.
  - How does that change how we think about performance and MPI?

# SMP Nodes: One Model

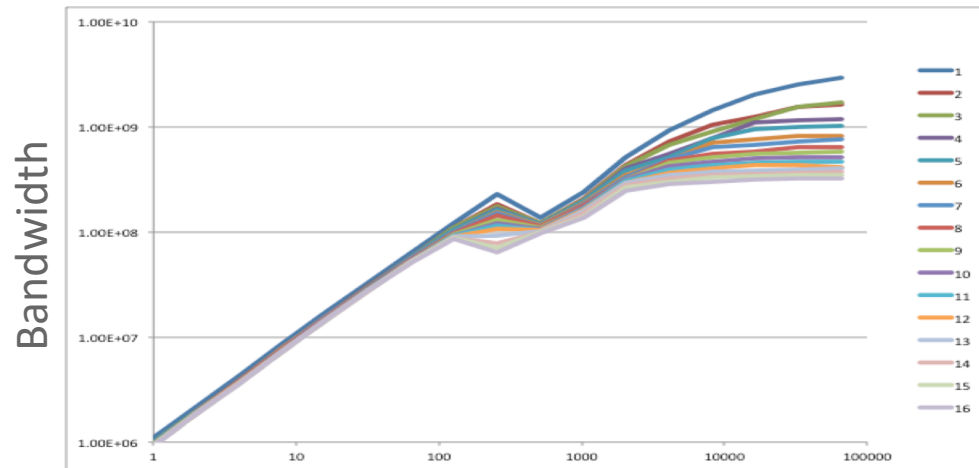
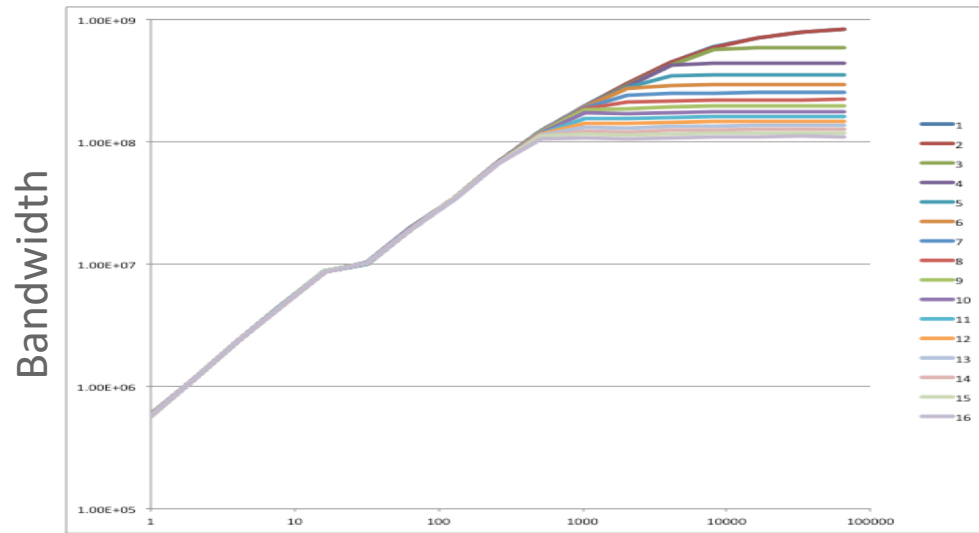




# Classic Performance Model

- $s + r n$ 
  - Sometimes called the “postal model”
- Model combines overhead and network latency ( $s$ ) and a single communication rate  $1/r$  for  $n$  bytes of data
- Good fit to machines when it was introduced
- But does it match modern SMP-based machines?
  - Let's look at the the communication rate per process with processes communicating between two nodes

# Rates Per MPI Process



- Ping-pong between 2 nodes using 1-16 cores on each node
- Top is BG/Q, bottom Cray XE6
- “Classic” model predicts a single curve – rates independent of the number of communicating processes

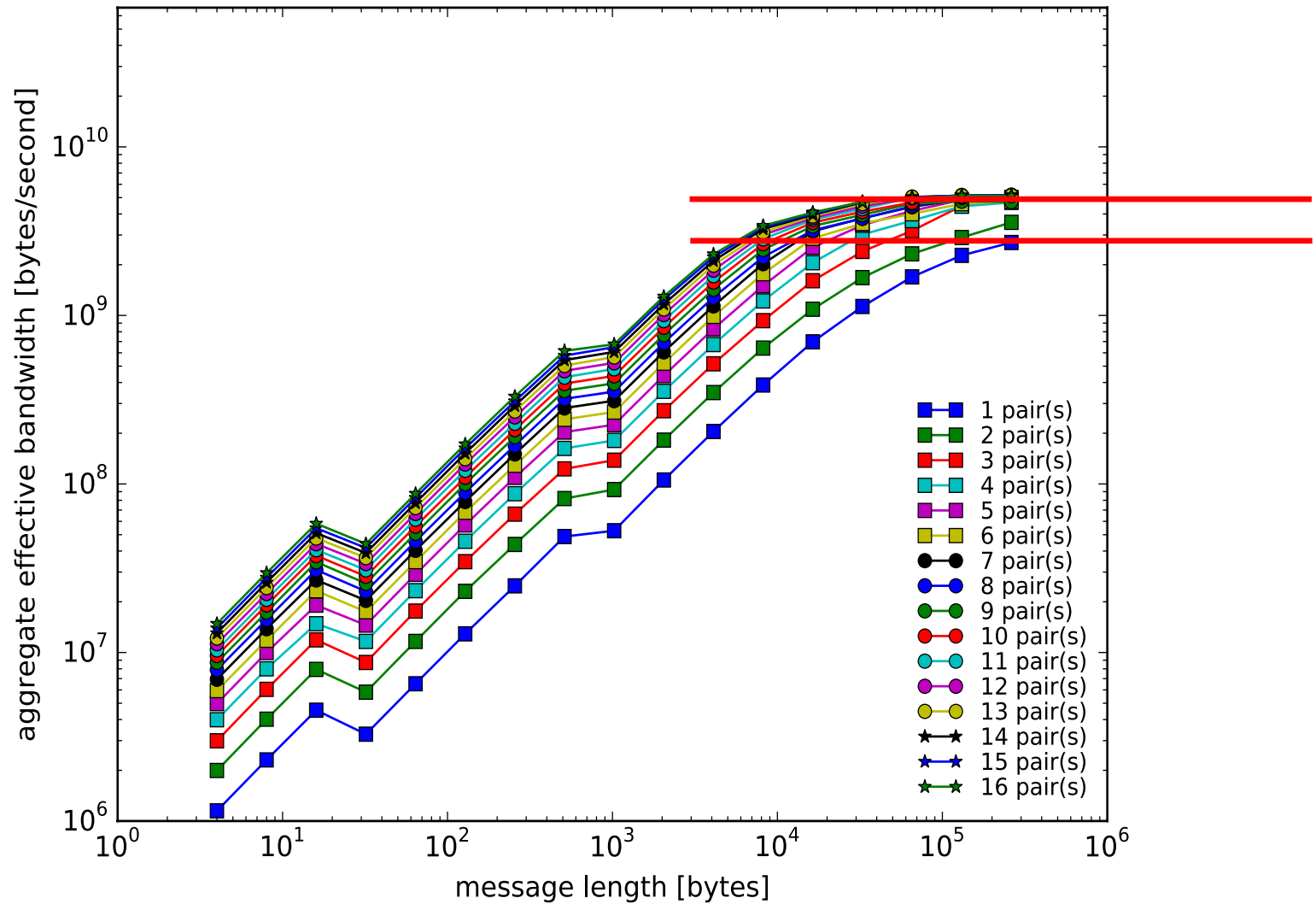
## A Slightly Better Model

- For  $k$  processes sending messages, the sustained rate is
  - $\min(R_{\text{NIC-NIC}}, k R_{\text{CORE-NIC}})$
- Thus
  - $T = s + k n / \min(R_{\text{NIC-NIC}}, k R_{\text{CORE-NIC}})$
- Note if  $R_{\text{NIC-NIC}}$  is very large (very fast network), this reduces to
  - $T = s + k n / (k R_{\text{CORE-NIC}}) = s + n / R_{\text{CORE-NIC}}$

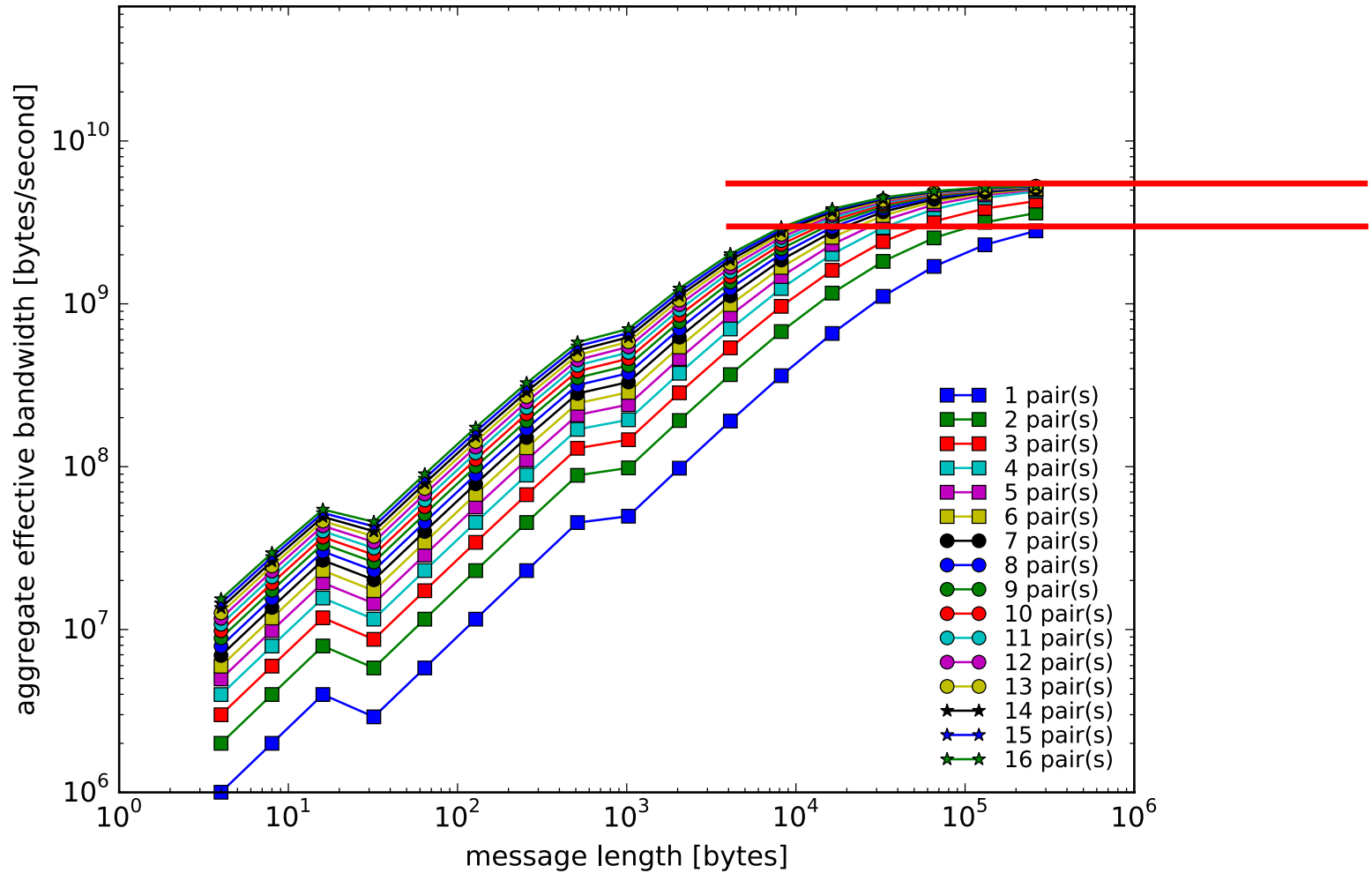
# How Well Does this Model Work?

- Tested on a wide range of systems:
  - Cray XE6 with Gemini network
  - IBM BG/Q
  - Cluster with InfiniBand
  - Cluster with another network
- Results in
  - Modeling MPI Communication Performance on SMP Nodes: Is it Time to Retire the Ping Pong Test
    - W Gropp, L Olson, P Samfass
    - Proceedings of EuroMPI 16
    - <https://doi.org/10.1145/2966884.2966919>
- Cray XE6 results follow

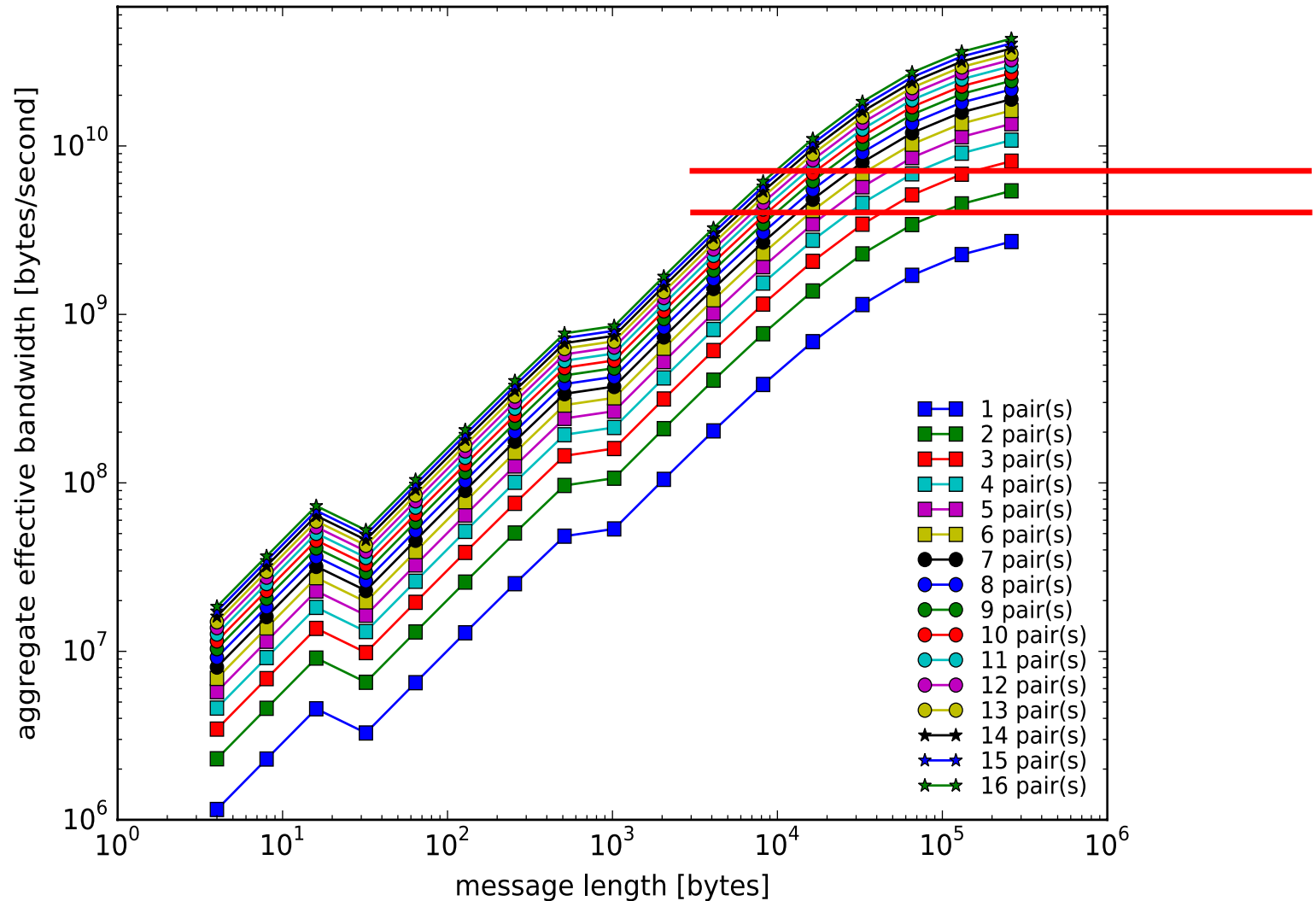
# Cray: Measured Data



# Cray: 3 parameter (new) model

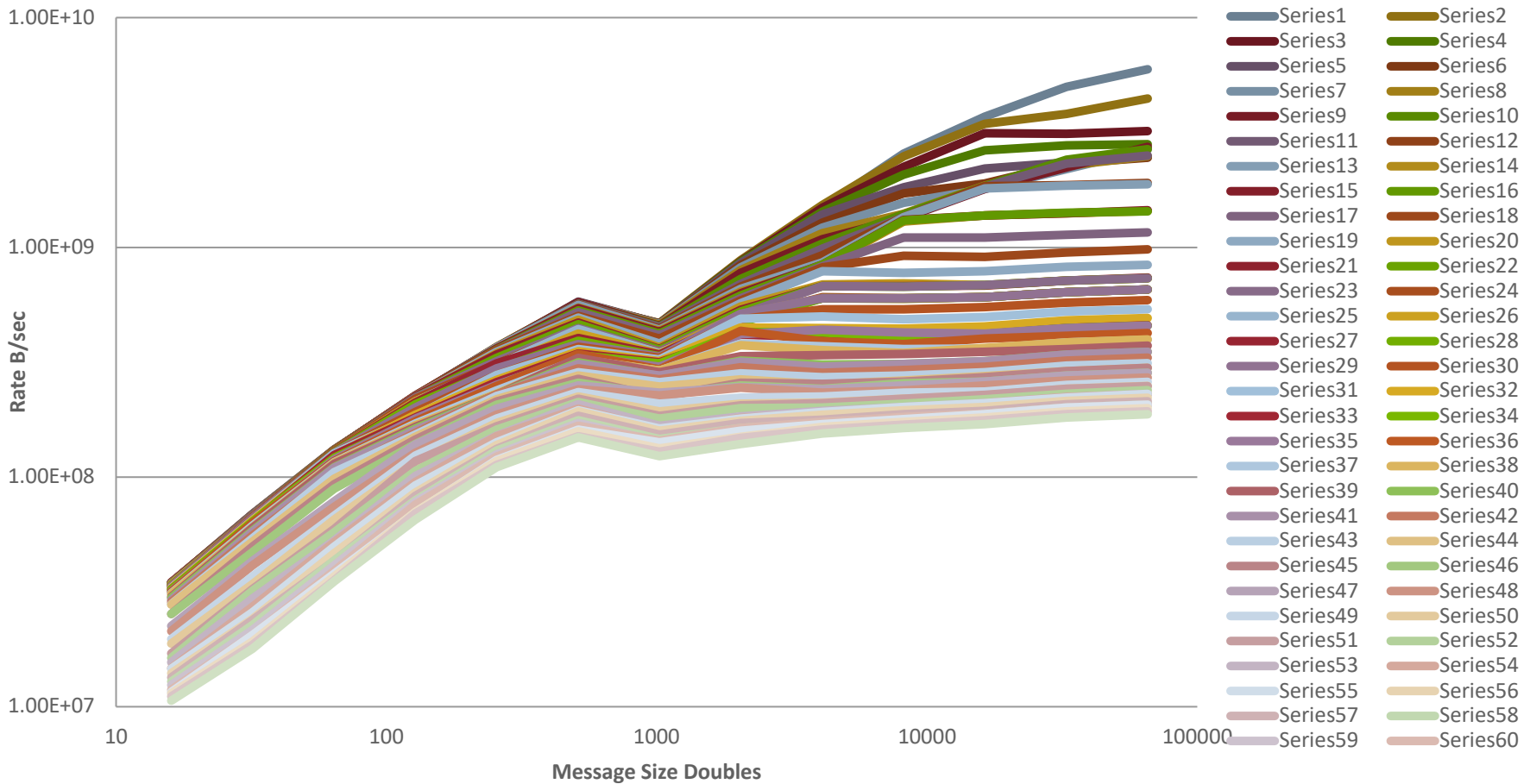


# Cray: 2 parameter ( $t=s+rn$ ) model



# Mpingpong results for Theta

## Intranode Pingpong Performance





# Lesson

- For  $k$  processes sending messages concurrently from the same node, the correct (more precisely, a much better) time model is
  - $T = s + k n / \min(R_{\text{NIC-NIC}}, k R_{\text{CORE-NIC}})$
- Many efficient programs can be designed entirely in terms of exchanging data between processes
  - But when communication rate is high, the bottlenecks in the system, particularly to/from a node, can become significant
  - In these cases, finding ways to reduce communication volume between nodes or increase the amount of time for the communication (thus lowering the rate) may be needed to achieve good performance.

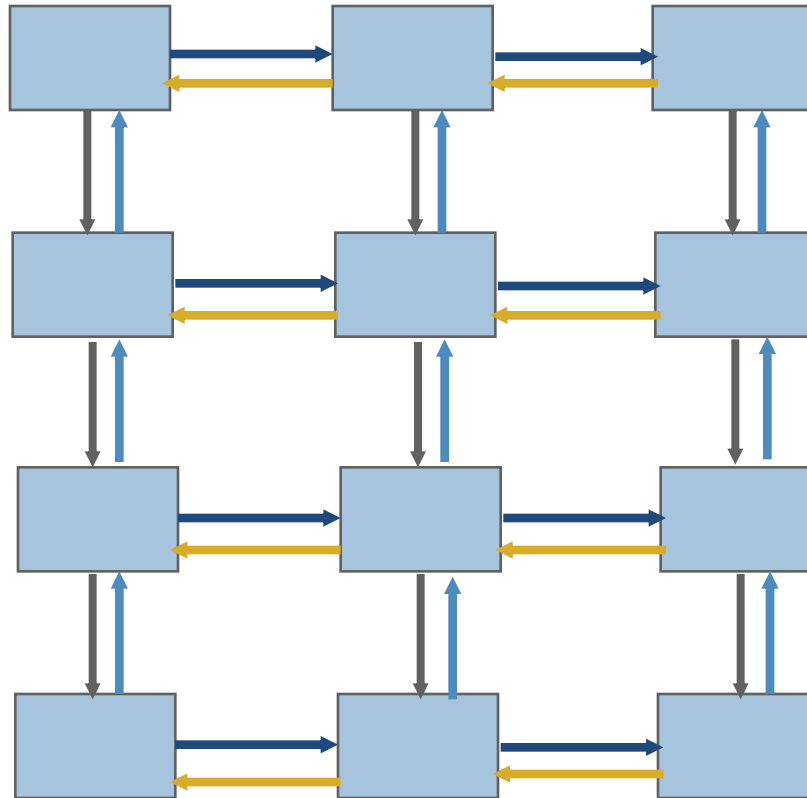
# Costs of Unintended Synchronization

# Unexpected Hot Spots

- Even simple operations can give surprising performance behavior.
- Examples arise even in common grid exchange patterns
- Message passing illustrates problems present even in shared memory
  - Blocking operations may cause unavoidable stalls

# Mesh Exchange

- Exchange data on a mesh



## Sample Code

- Do i=1,n\_neighbors  
    Call MPI\_Send(edge(1,i), len, MPI\_REAL,&  
                  nbr(i), tag,comm, ierr)  
Enddo  
Do i=1,n\_neighbors  
    Call MPI\_Recv(edge(1,i), len, MPI\_REAL,&  
                  nbr(i), tag, comm, status, ierr)  
Enddo

# Deadlocks!

- All of the sends may block, waiting for a matching receive (will for large enough messages)
- The variation of  
if (has down nbr) then  
    Call MPI\_Send( ... down ... )  
endif  
if (has up nbr) then  
    Call MPI\_Recv( ... up ... )  
endif  
...  
sequentializes (all except the bottom process blocks)

# Sequentialization

Start Send	Start Send	Start Send	Start Send	Start Send	Start Send Send	Send Recv	Recv
				Send	Recv		
			Send	Recv			
		Send	Recv				
	Send						
Send	Recv						

## Fix 1: Use Irecv

- Do i=1,n\_neighbors  
    Call MPI\_Irecv(inedge(1,i), len, MPI\_REAL, nbr(i), tag,&  
                    comm, requests(i), ierr)  
Enddo  
Do i=1,n\_neighbors  
    Call MPI\_Send(edge(1,i), len, MPI\_REAL, nbr(i), tag,&  
                    comm, ierr)  
Enddo  
Call MPI\_Waitall(n\_neighbors, requests, statuses, ierr)
- Does not perform well in practice. Why?

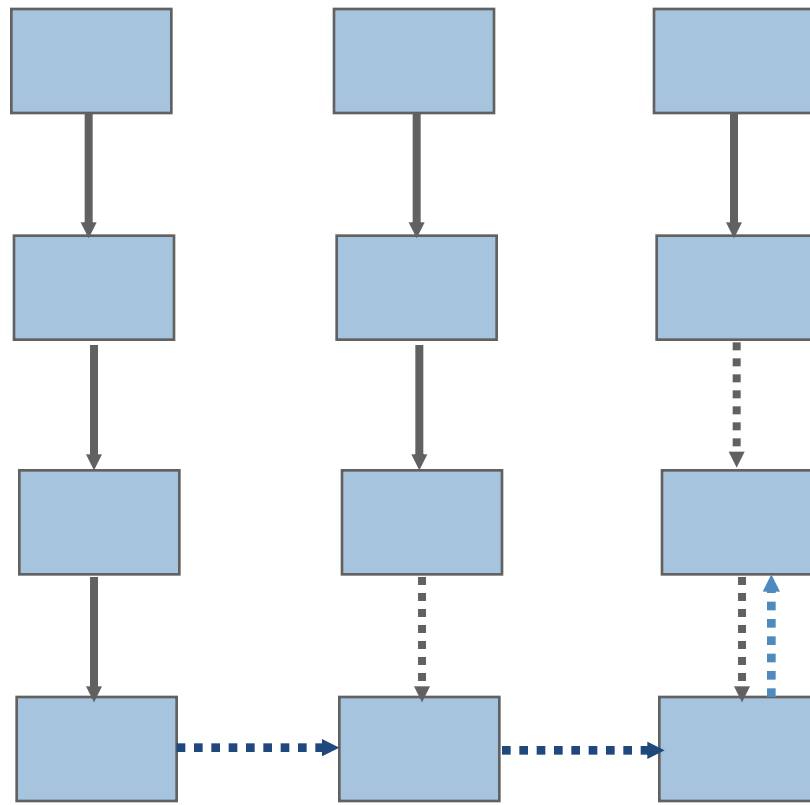


# Understanding the Behavior: Timing Model

- Sends interleave
- Sends block (data larger than buffering will allow)
- Sends control timing
- Receives do not interfere with Sends
- Exchange can be done in 4 steps (down, right, up, left)

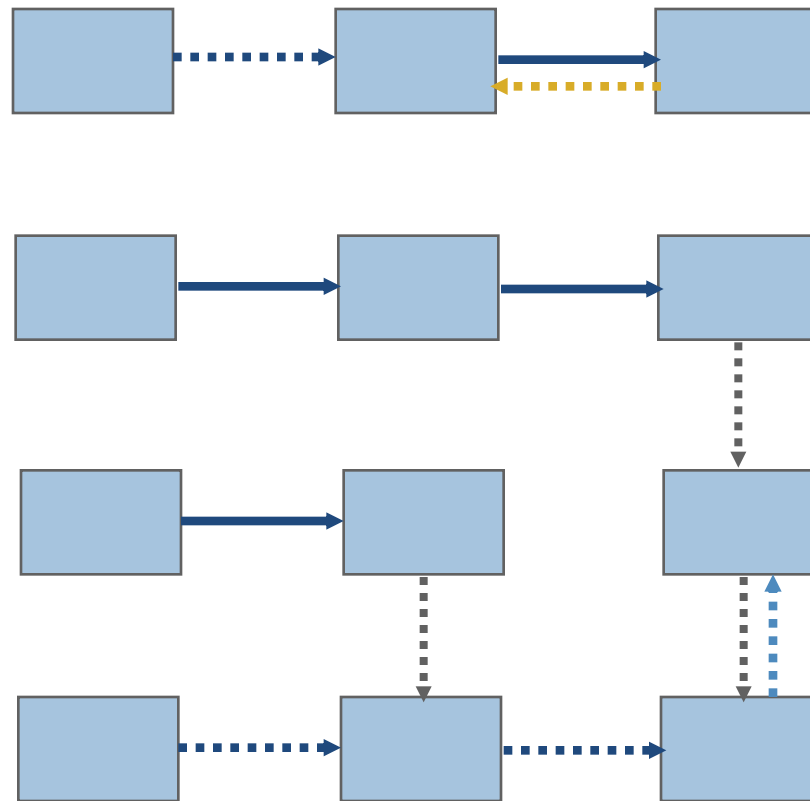
# Mesh Exchange - Step 1

- Exchange data on a mesh



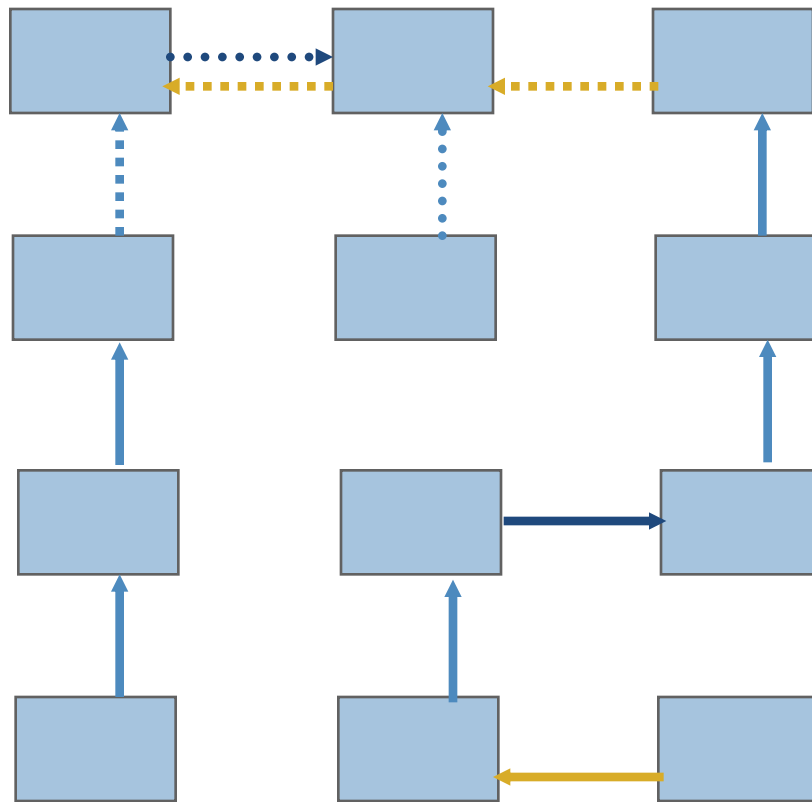
## Mesh Exchange - Step 2

- Exchange data on a mesh



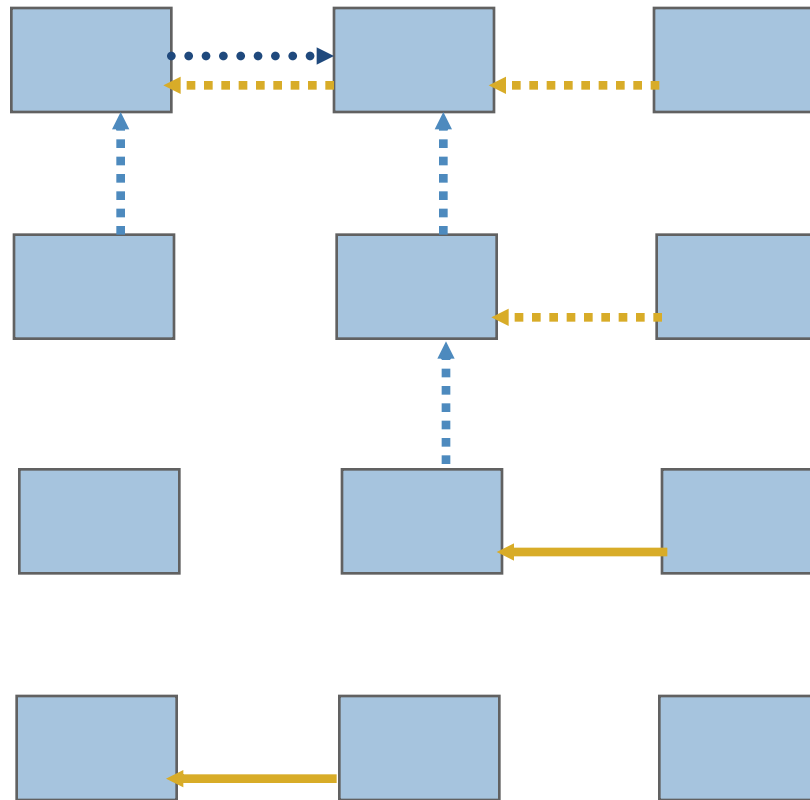
## Mesh Exchange - Step 3

- Exchange data on a mesh



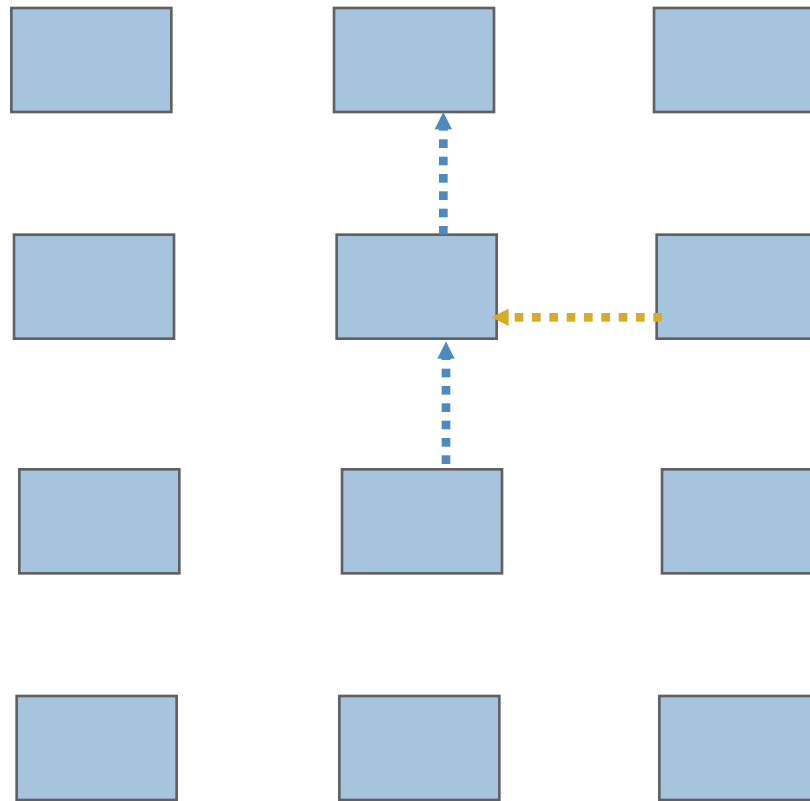
# Mesh Exchange - Step 4

- Exchange data on a mesh



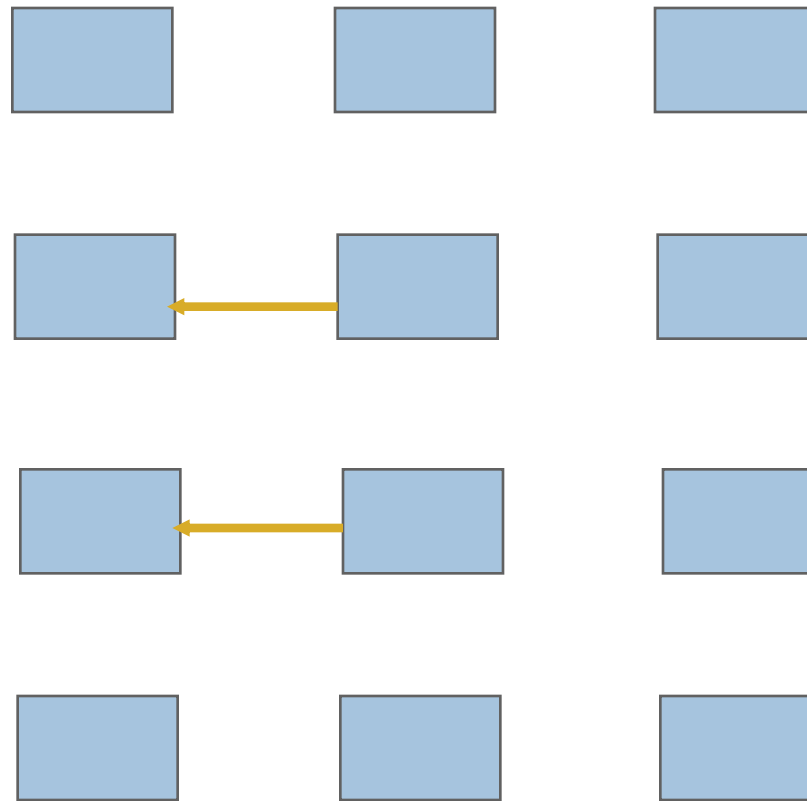
## Mesh Exchange - Step 5

- Exchange data on a mesh

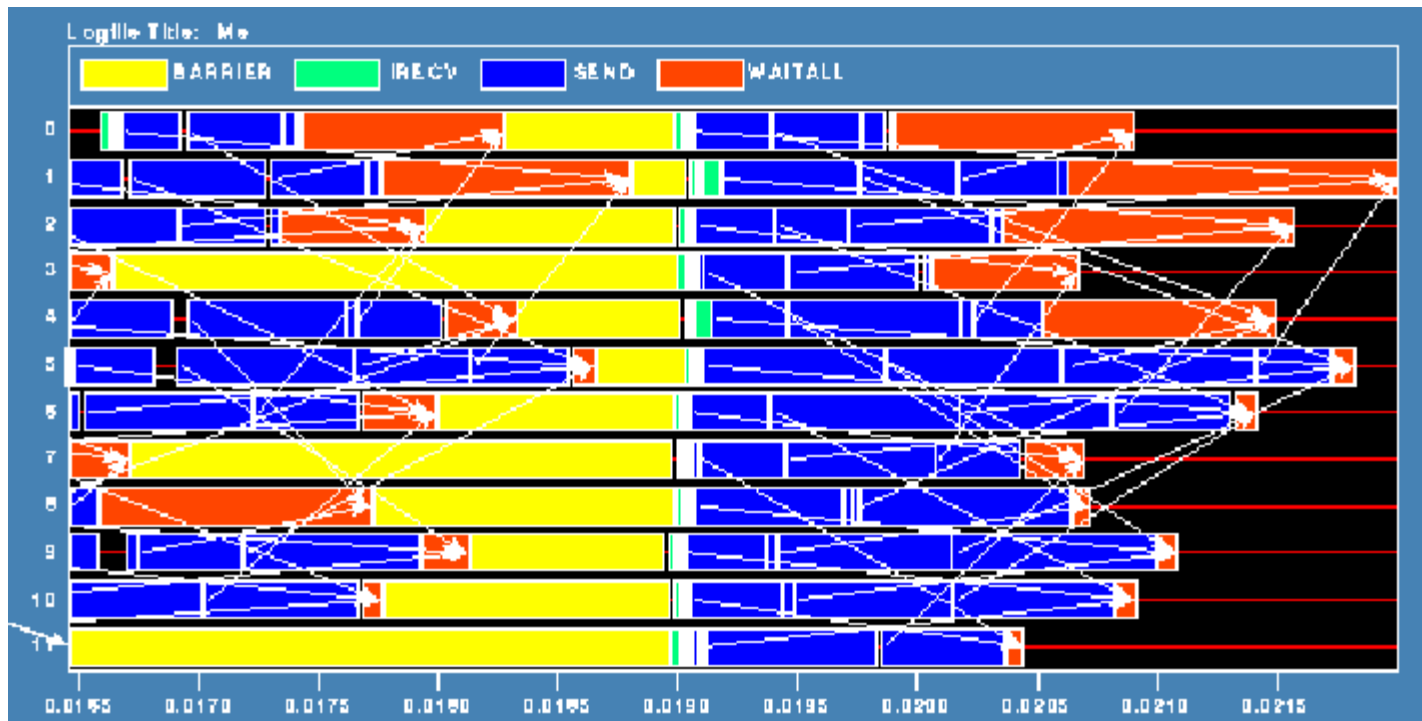


# Mesh Exchange - Step 6

- Exchange data on a mesh



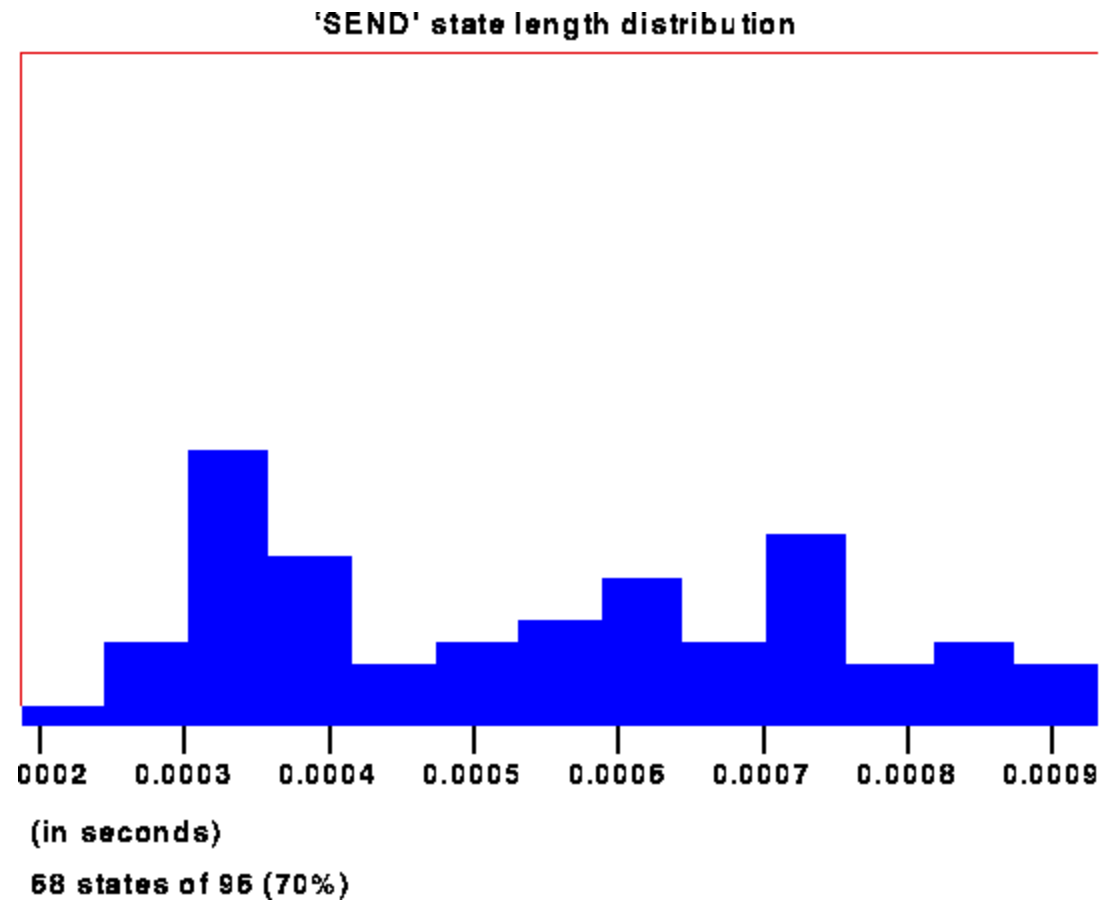
# Timeline from IBM SP



- Note that process 1 finishes last, as predicted



# Distribution of Sends



# Why Six Steps?

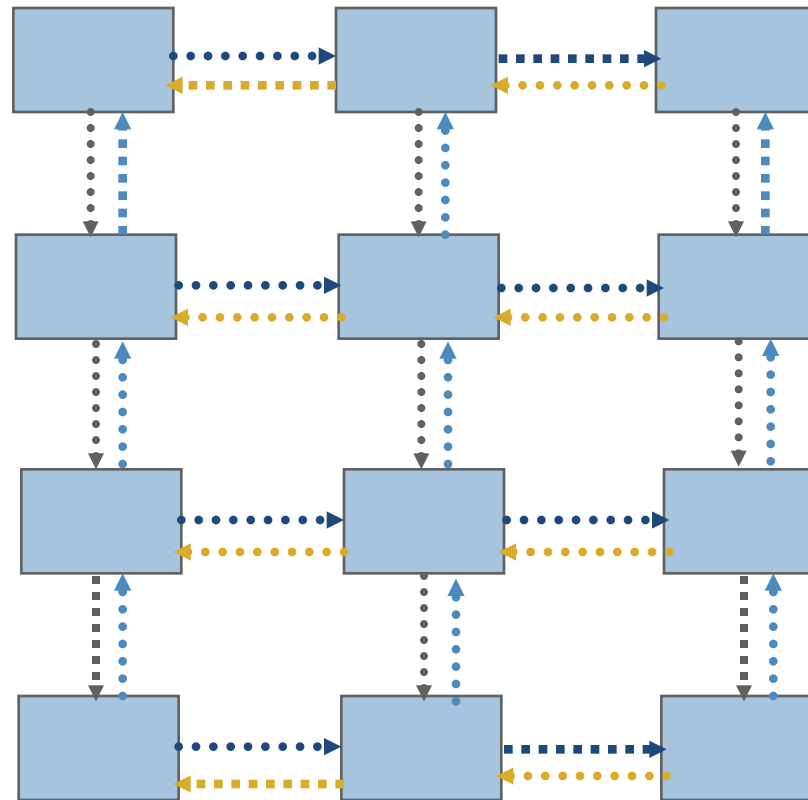
- Ordering of Sends introduces delays when there is contention at the receiver
- Takes roughly twice as long as it should
- Bandwidth is being wasted
- Same thing would happen if using memcpy and shared memory

## Fix 2: Use Isend and Irecv

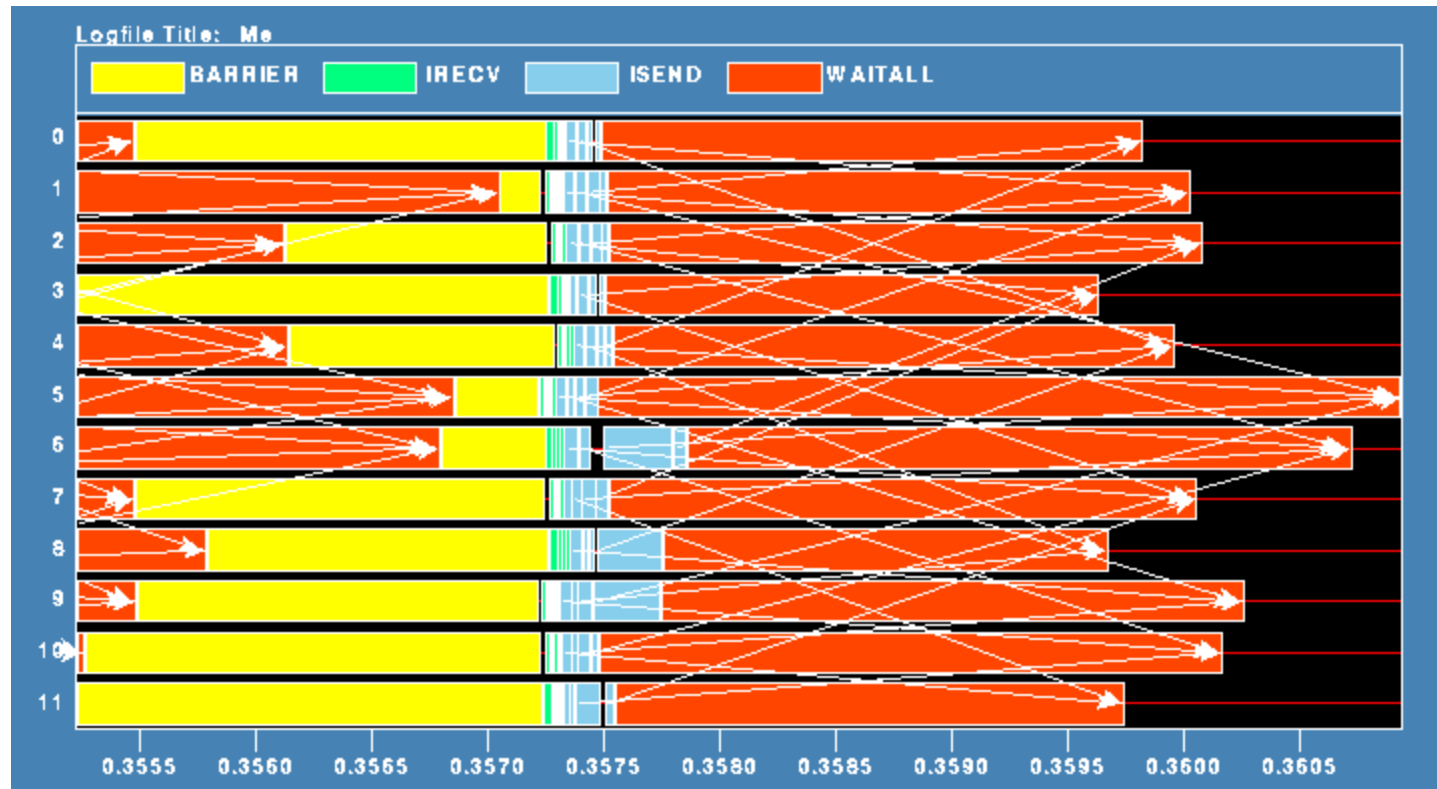
- Do i=1,n\_neighbors  
    Call MPI\_Irecv(inedge(1,i),len,MPI\_REAL,nbr(i),tag,&  
                  comm, requests(i),ierr)  
Enddo  
Do i=1,n\_neighbors  
    Call MPI\_Isend(edge(1,i), len, MPI\_REAL, nbr(i), tag,&  
                  comm, requests(n\_neighbors+i), ierr)  
Enddo  
Call MPI\_Waitall(2\*n\_neighbors, requests, statuses, ierr)

# Mesh Exchange - Steps 1-4

- Four interleaved steps



# Timeline from IBM SP



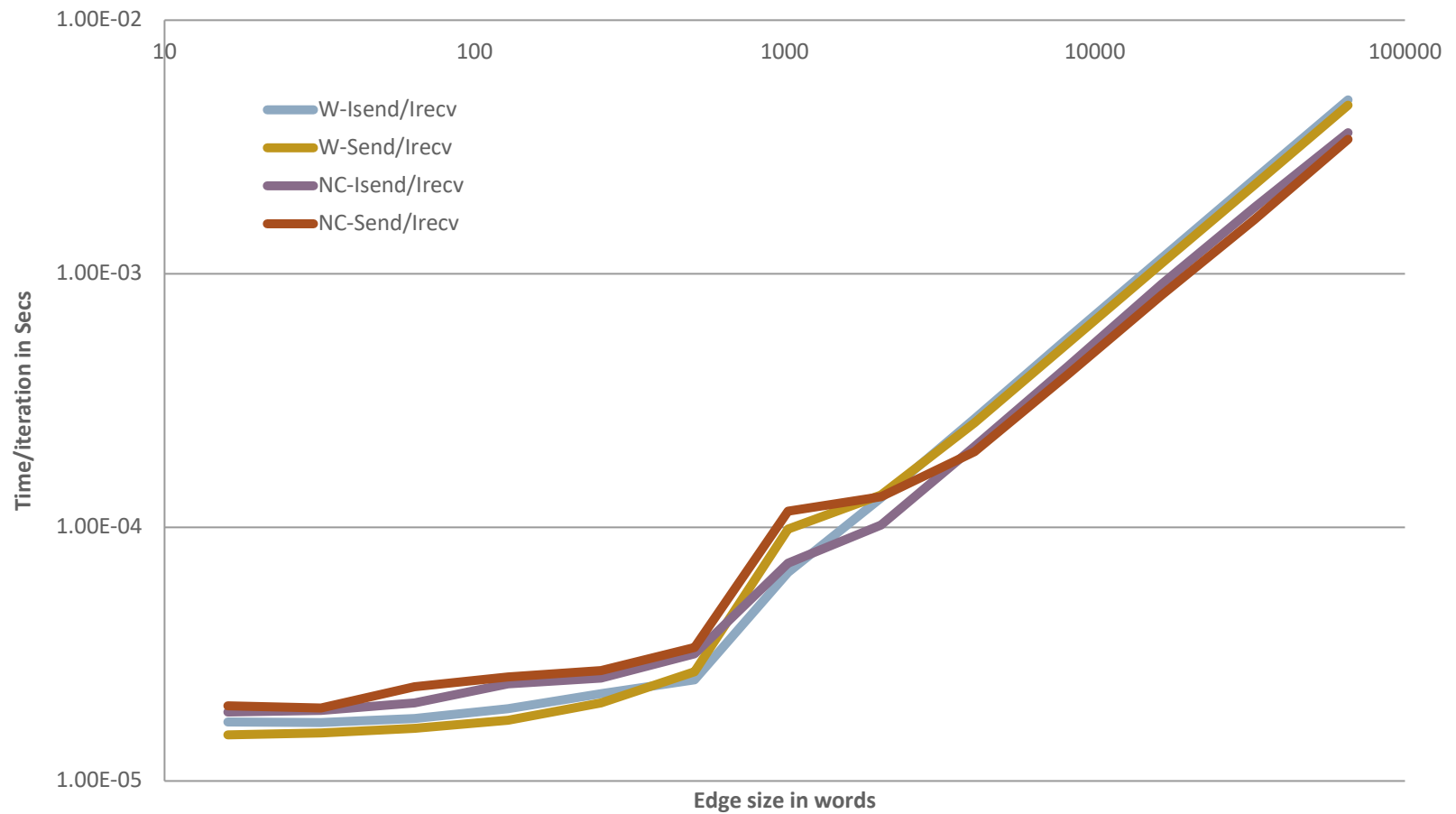
Note processes 5 and 6 are the only interior processors; these perform more communication than the other processors

# Lesson: Defer Synchronization

- Send-receive accomplishes two things:
  - Data transfer
  - Synchronization
- In many cases, there is more synchronization than required
- Consider the use of nonblocking operations and `MPI_Waitall` to defer synchronization
  - Effectiveness depends on how data is moved by the MPI implementation
  - E.g., If large messages are moved by blocking RMA operations “under the covers,” the implementation can’t adapt to contention at the target processes, and you may see no benefit.
  - This is more likely with larger messages

# Hotspot results for Theta

## 2-d Mesh Exchange Comparison



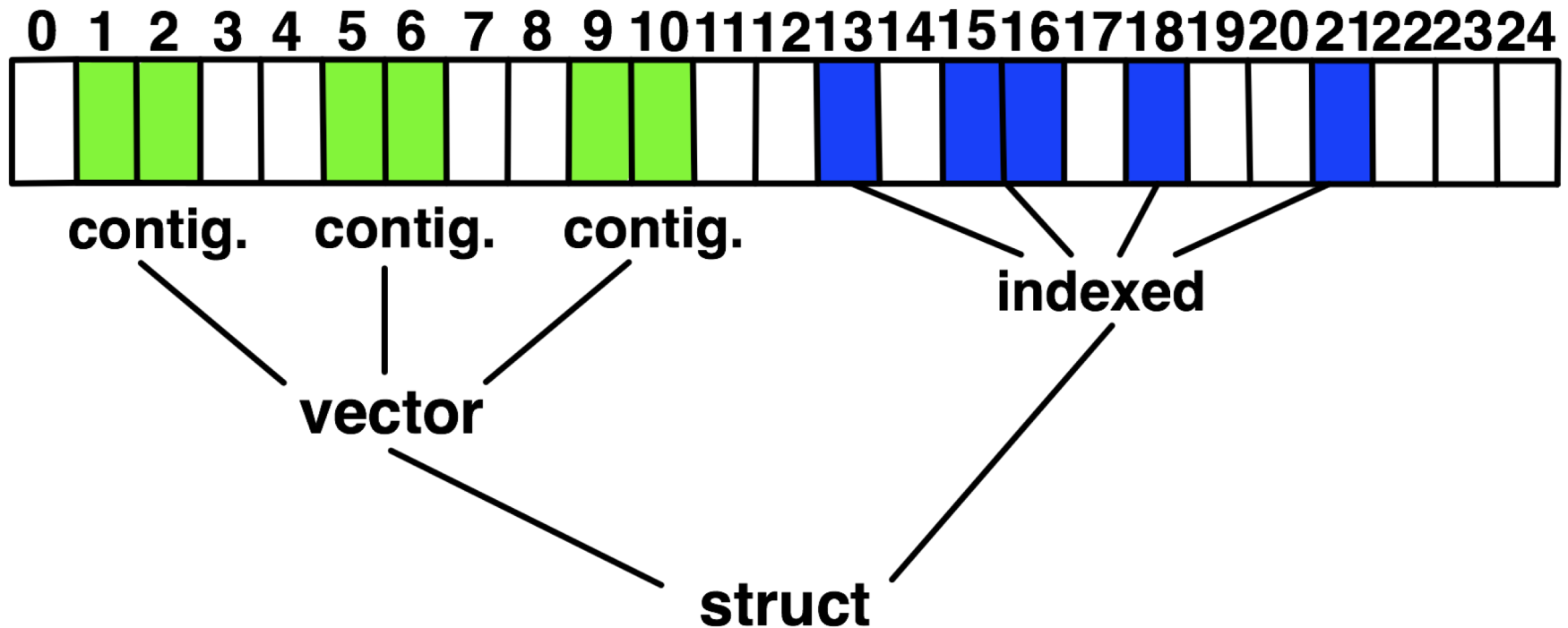
# Datatypes



# Introduction to Datatypes in MPI

- Datatypes allow users to 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



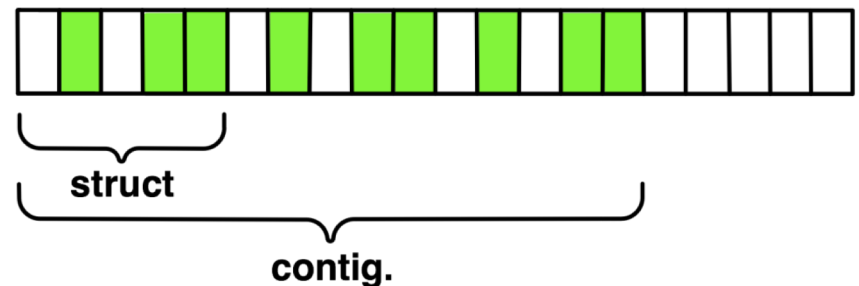
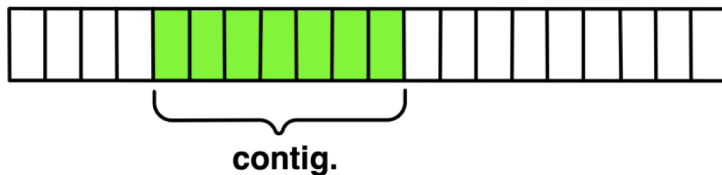
# MPI's Intrinsic Datatypes

- Why intrinsic types?
  - Heterogeneity, nice to send a Boolean from C to Fortran
  - Conversion rules are complex, not discussed here
  - Length matches to language types
    - No sizeof(int) mess
- Users should generally use intrinsic types as basic types for communication and type construction!
  - MPI\_BYTE should only be used for data that are raw bytes
- MPI-2.2 added some missing C types
  - E.g., unsigned long long

# MPI\_Type\_contiguous

```
MPI_Type_contiguous(int count, MPI_Datatype  
oldtype, MPI_Datatype *newtype)
```

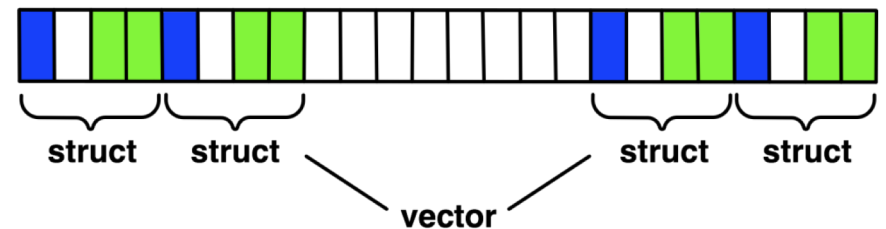
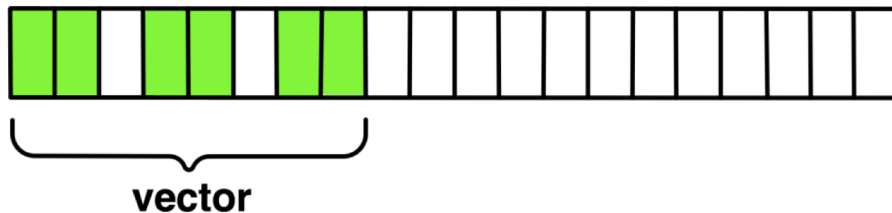
- Contiguous array of oldtype
- Should not be used as last type (can be replaced by count)



# MPI\_Type\_vector

```
MPI_Type_vector(int count, int blocklength, int stride,  
MPI_Datatype oldtype, MPI_Datatype *newtype)
```

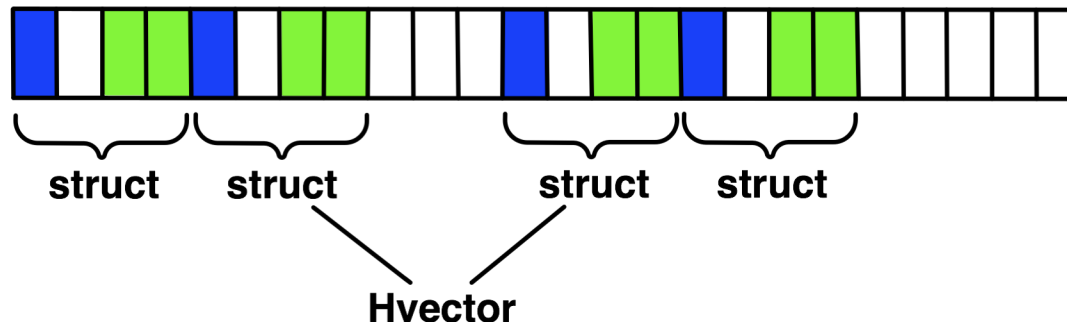
- Specify strided blocks of data of oldtype
- Very useful for Cartesian arrays



# MPI\_Type\_create\_hvector

```
MPI_Type_create_hvector(int count, int blocklength, MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- Create non-unit strided vectors
- Useful for composition, e.g., vector of structs



# MPI\_Type\_create\_indexed\_block

```
MPI_Type_create_indexed_block(int count, int blocklength,  
int *array_of_displacements, MPI_Datatype oldtype,  
MPI_Datatype *newtype)
```

- Like Create\_indexed but blocklength is the same



- blen=2
- displs={0,5,9,13,18}

# MPI\_Type\_indexed

```
MPI_Type_indexed(int count, int *array_of_blocklengths,  
int *array_of_displacements, MPI_Datatype oldtype,  
MPI_Datatype *newtype)
```

- Pulling irregular subsets of data from a single array (cf. vector collectives)
  - Dynamic codes with index lists, expensive though!



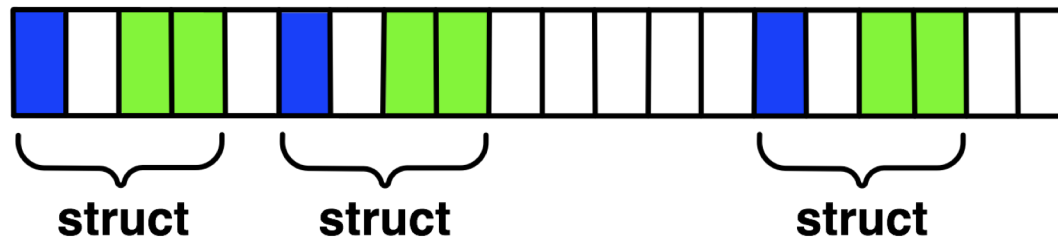
- `blen={1,1,2,1,2,1}`
- `displs={0,3,5,9,13,17}`



# MPI\_Type\_create\_hindexed

```
MPI_Type_create_hindexed(int count, int *arr_of_blocklengths,  
MPI_Aint *arr_of_displacements, MPI_Datatype oldtype,  
MPI_Datatype *newtype)
```

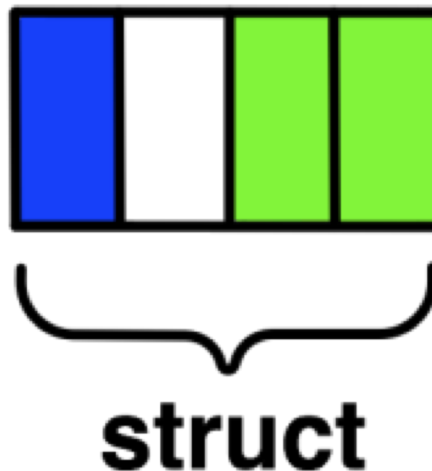
- Indexed with non-unit displacements, e.g., pulling types out of different arrays



# MPI\_Type\_create\_struct

```
MPI_Type_create_struct(int count, int array_of_blocklengths[],  
MPI_Aint array_of_displacements[], MPI_Datatype  
array_of_types[], MPI_Datatype *newtype)
```

- Most general constructor, allows different types and arbitrary arrays (also most costly)



## MPI\_Type\_create\_subarray

```
MPI_Type_create_subarray(int ndims, int array_of_sizes[],  
int array_of_subsizes[], int array_of_starts[], int order,  
MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- Specify subarray of n-dimensional array (sizes) by start (starts) and size (subsize)

(0,0)	(1,0)	(2,0)	(3,0)
(0,1)	(1,1)	(2,1)	(3,1)
(0,2)	(1,2)	(2,2)	(3,2)
(0,3)	(1,3)	(2,3)	(3,3)

# MPI\_Type\_create\_darray

```
MPI_Type_create_darray(int size, int rank, int ndims,  
int array_of_gsizes[], int array_of_distrib[], int  
array_of_dargs[], int array_of_psize[], int order,  
MPI_Datatype oldtype, MPI_Datatype *newtype)
```

- Create distributed array, supports block, cyclic and no distribution for each dimension
  - Very useful for I/O

(0,0)	(1,0)	(2,0)	(3,0)
(0,1)	(1,1)	(2,1)	(3,1)
(0,2)	(1,2)	(2,2)	(3,2)
(0,3)	(1,3)	(2,3)	(3,3)

# MPI\_BOTTOM and MPI\_Get\_address

- MPI\_BOTTOM is the absolute zero address
  - Portability (e.g., may be non-zero in globally shared memory)
- MPI\_Get\_address
  - Returns address relative to MPI\_BOTTOM
  - Portability (do not use “&” operator in C!)
- Very important when
  - Building struct datatypes
  - Data spans multiple arrays

# Commit, Free, and Dup

- Types must be committed before use
  - Only the ones that are used explicitly in a call!
  - `MPI_Type_commit` may perform time-consuming optimizations (but few implementations currently exploit this feature)
- `MPI_Type_free`
  - Free MPI resources of datatypes
  - Does not affect types built from it
- `MPI_Type_dup`
  - Duplicates a type
  - Library abstraction (composability)

# Other Datatype Functions

- Pack/Unpack
  - Mainly for compatibility to legacy libraries
  - Provides a way to incrementally pack and unpack data
    - Use only when this is required for the code – MPI Datatypes will perform better than Pack/Unpack at moving the same data
- Get\_envelope/contents
  - Only for expert library developers
  - Libraries like MPITypes<sup>1</sup> make this easier
- MPI\_Type\_create\_resized
  - Change extent and size (useful but must be used with great care)

*<http://www.mcs.anl.gov/mpitypes/>*

# Datatype Selection Order

- Simple and effective performance model:
  - More parameters == slower
- **contig < vector < index\_block < index < struct**
- Some (most) MPIs are inconsistent WRT this performance rule
  - But this rule is portable



# Datatype Performance in Practice

- Datatypes *can* provide performance benefits, particularly for certain regular patterns
  - However, many implementations do not optimize datatype operations
  - If performance is critical, you will need to test
    - Even manual packing/unpacking can be slow if not properly optimized by the compiler – make sure to check optimization reports or if the compiler doesn't provide good reports, inspect the assembly code
- For parallel I/O, datatypes *do* provide large performance benefits in many cases

# Collectives and Nonblocking Collectives

# Introduction to Collective Operations in MPI

- Collective operations are called by all processes in a communicator.
- **MPI\_BCAST** distributes data from one process (the root) to all others in a communicator.
- **MPI\_REDUCE** combines data from all processes in the communicator and returns it to one process.
- In many numerical algorithms, **SEND/RECV** can be replaced by **BCAST/REDUCE**, improving both simplicity and efficiency.

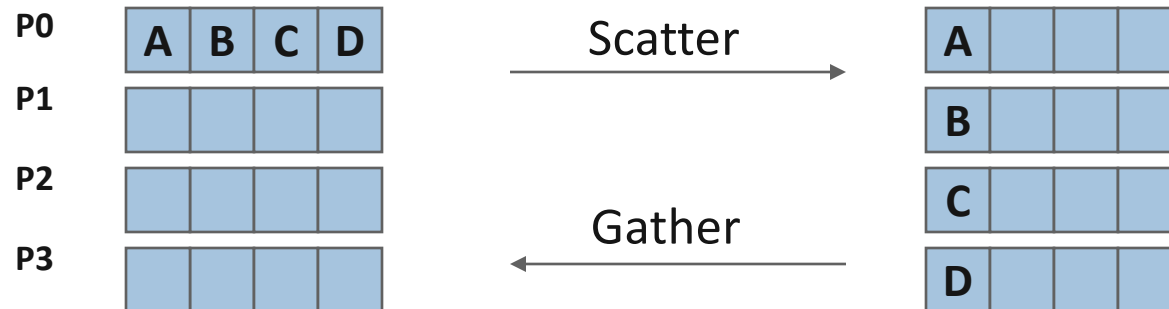
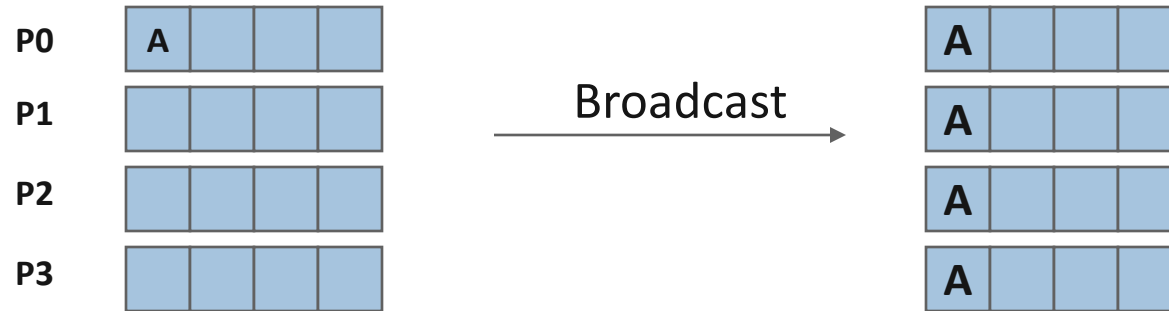
# MPI Collective Communication

- Communication and computation is coordinated among a group of processes in a communicator
- Tags are not used; different communicators deliver similar functionality
- Non-blocking collective operations in MPI-3
- Three classes of operations: synchronization, data movement, collective computation

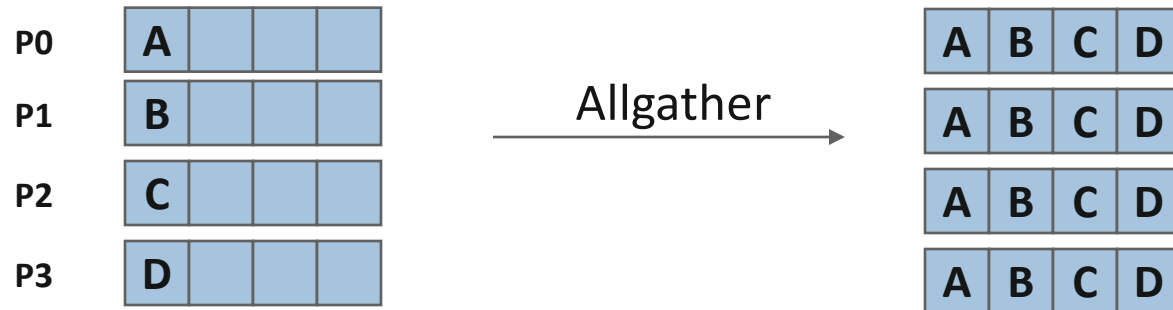
# Synchronization

- **MPI\_BARRIER(comm)**
  - Blocks until all processes in the group of communicator **comm** call it
  - A process cannot get out of the barrier until all other processes have reached barrier
- Note that a barrier is rarely, if ever, necessary in an MPI program
- Adding barriers “just to be sure” is a bad practice and causes unnecessary synchronization. **Remove unnecessary barriers from your code.**
- One legitimate use of a barrier is before the first call to MPI\_Wtime to start a timing measurement. This causes each process to start at *approximately* the same time.
- Avoid using barriers other than for this.

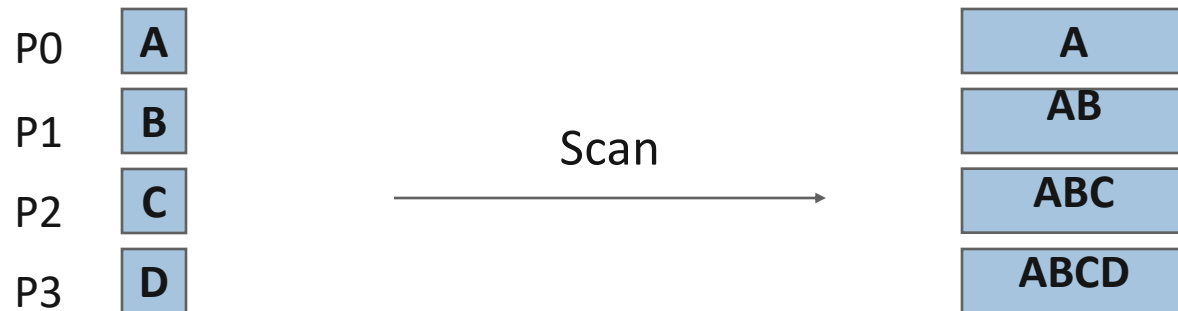
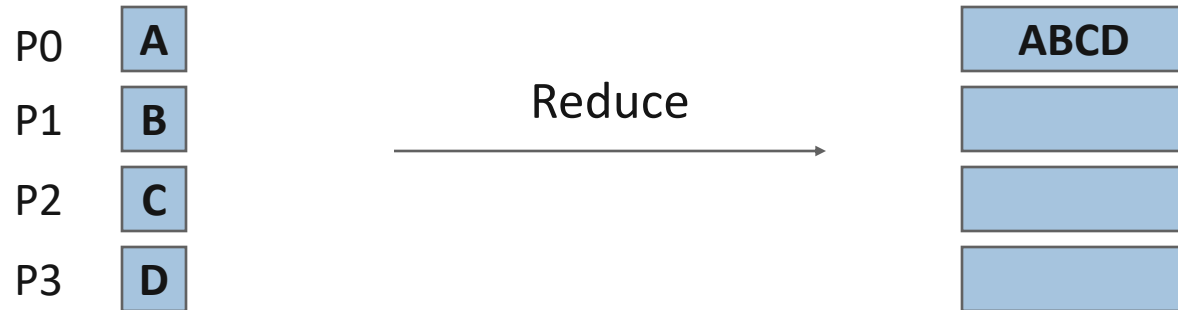
# Collective Data Movement



# More Collective Data Movement



# Collective Computation





# MPI Collective Routines

- Many Routines, including: `MPI_ALLGATHER`, `MPI_ALLGATHERV`, `MPI_ALLREDUCE`, `MPI_ALLTOALL`, `MPI_ALLTOALLV`, `MPI_BCAST`, `MPI_EXSCAN`, `MPI_GATHER`, `MPI_GATHERV`, `MPI_REDUCE`, `MPI_REDUCE_SCATTER`, `MPI_SCAN`, `MPI_SCATTER`, `MPI_SCATTERV`
- “**A**ll” versions deliver results to all participating processes
- “**V**” versions (stands for vector) allow the chunks to have different sizes
- “**W**” versions for ALLTOALL allow the chunks to have different sizes in bytes, rather than units of datatypes
- `MPI_ALLREDUCE`, `MPI_REDUCE`, `MPI_REDUCE_SCATTER`, `MPI_REDUCE_SCATTER_BLOCK`, `MPI_EXSCAN`, and `MPI_SCAN` take both built-in and user-defined combiner functions

# MPI Built-in Collective Computation Operations

■ <code>MPI_MAX</code>	Maximum
■ <code>MPI_MIN</code>	Minimum
■ <code>MPI_PROD</code>	Product
■ <code>MPI_SUM</code>	Sum
■ <code>MPI_LAND</code>	Logical and
■ <code>MPI_LOR</code>	Logical or
■ <code>MPI_LXOR</code>	Logical exclusive or
■ <code>MPI_BAND</code>	Bitwise and
■ <code>MPI_BOR</code>	Bitwise or
■ <code>MPI_BXOR</code>	Bitwise exclusive or
■ <code>MPI_MAXLOC</code>	Maximum and location
■ <code>MPI_MINLOC</code>	Minimum and location
■ <code>MPI_REPLACE,</code> <code>MPI_NO_OP</code>	Replace and no operation (RMA)

# Defining your own Collective Operations

- Create your own collective computations with:

```
MPI_OP_CREATE(user_fn, commutes, &op);
```

```
MPI_OP_FREE(&op);
```

```
user_fn(invec, inoutvec, len, datatype);
```

- The user function should perform:

```
inoutvec[i] = invec[i] op inoutvec[i];
```

```
for i from 0 to len-1
```

- The user function can be non-commutative, but must be associative

# 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

# 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 Collective Communication

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

# Nonblocking Collective Communication

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



# 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!

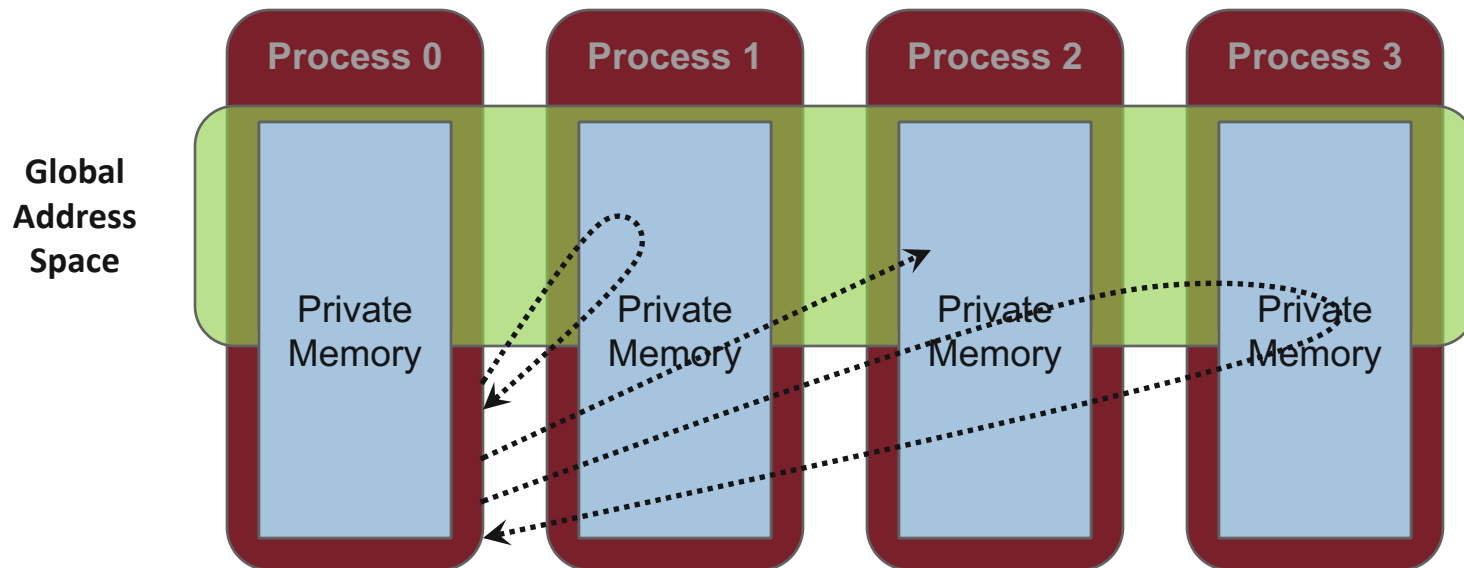
# Nonblocking And Collective Summary

- Nonblocking communication
  - Overlap and relax synchronization
- Collective communication
  - Specialized pre-optimized routines
  - Performance portability
  - Hopefully transparent performance
- They can be composed
  - E.g., software pipelining

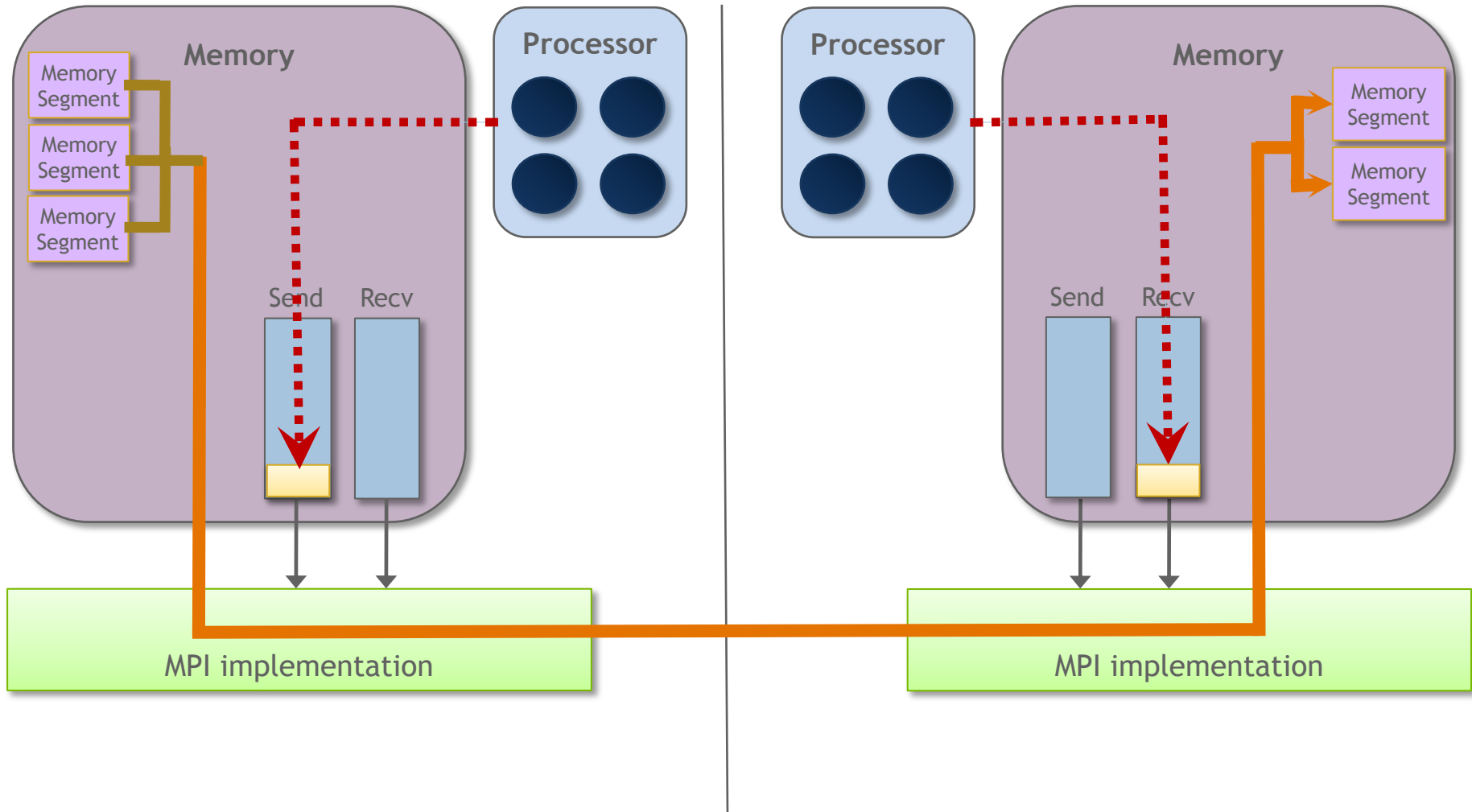
# Advanced Topics: One-sided Communication

# One-sided Communication

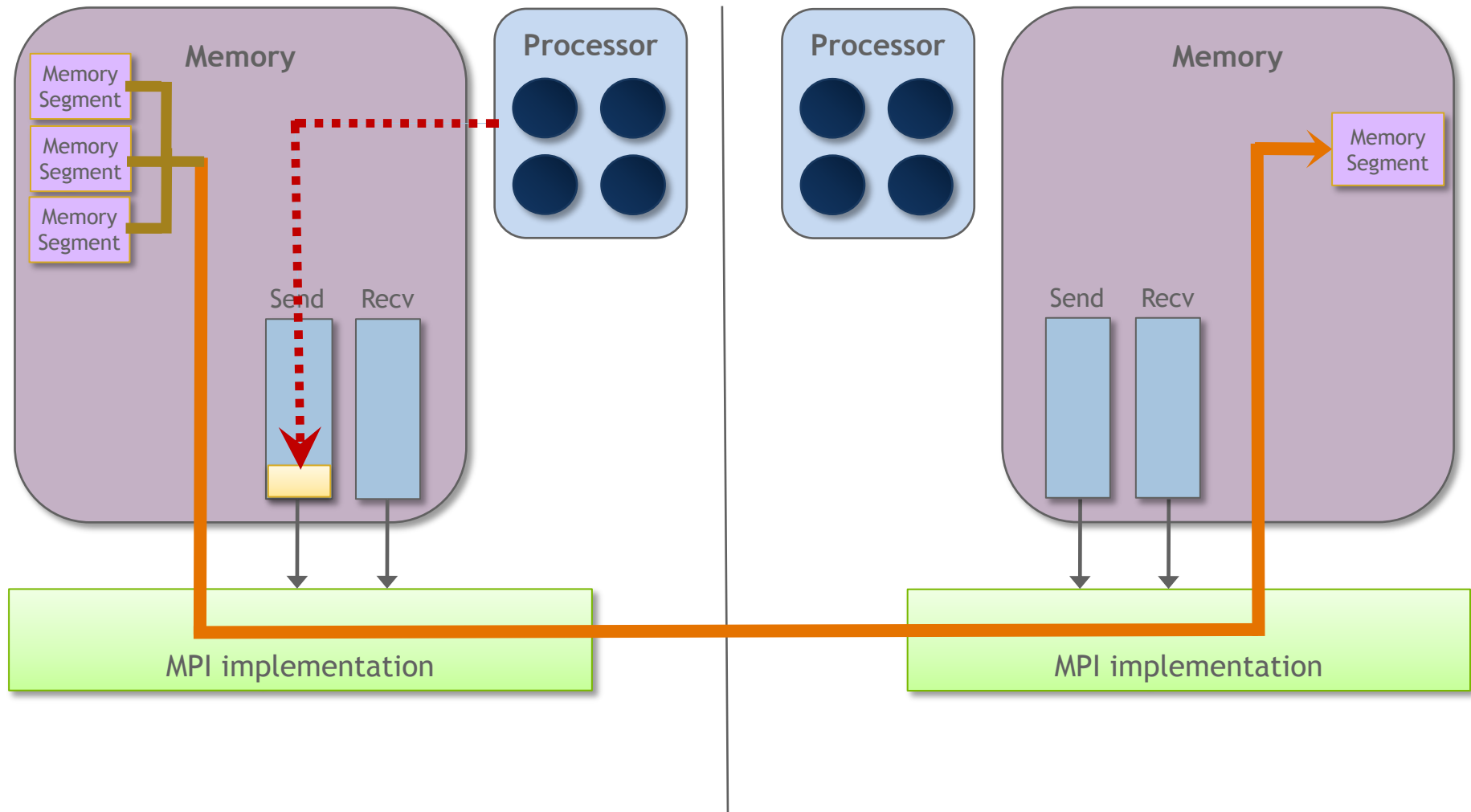
- The basic idea of one-sided communication models is to decouple data movement with process synchronization
  - Should be able to move data without requiring that the remote process synchronize
  - Each process exposes a part of its memory to other processes
  - Other processes can directly read from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



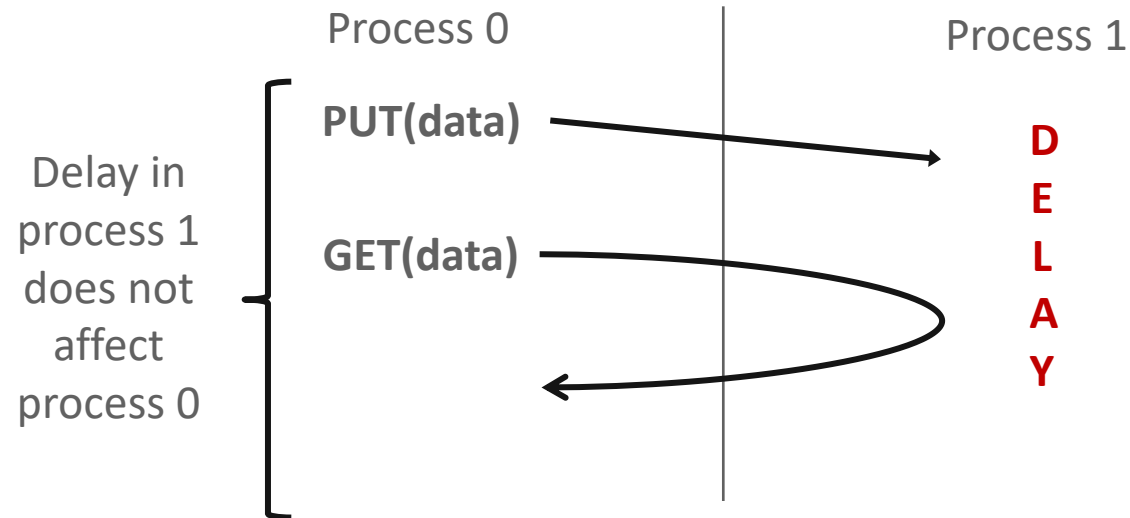
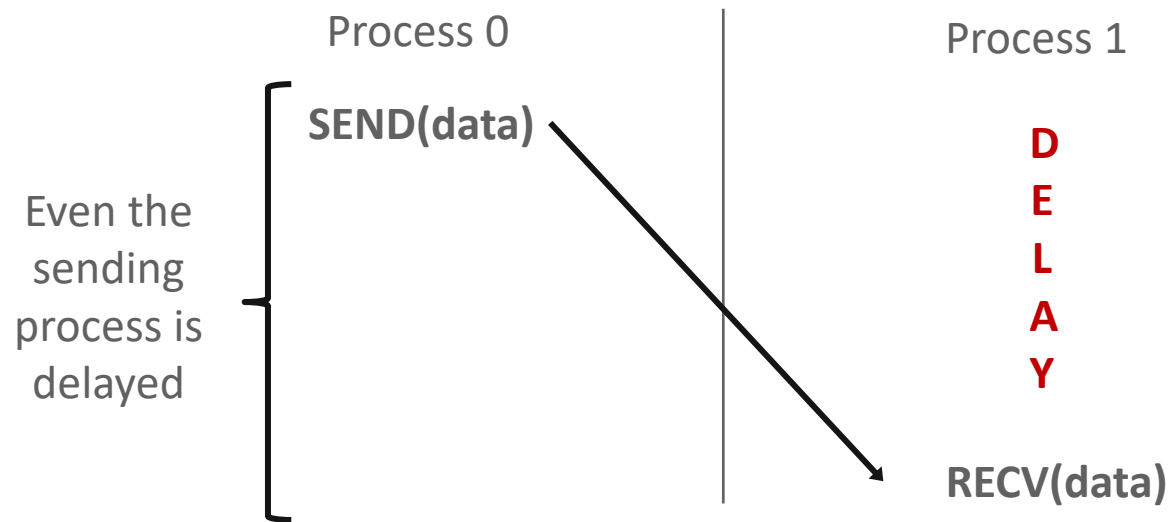
# Two-sided Communication Example



# One-sided Communication Example



# Comparing One-sided and Two-sided Programming



# What we need to know in MPI RMA

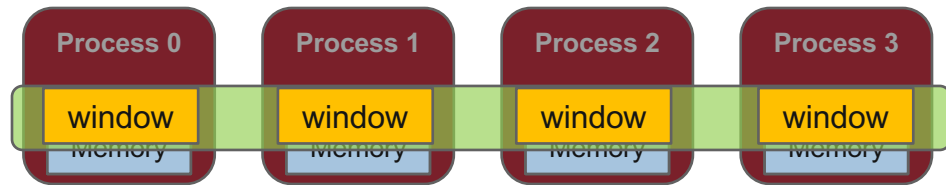
- How to create remote accessible memory?
- Reading, Writing and Updating remote memory
- Data Synchronization
- Memory Model



# Creating Public Memory

- Any memory used by a process is, by default, only locally accessible

- `X = malloc(100);`



- Once the memory is allocated, the user has to make an explicit MPI call to declare a memory region as remotely accessible
  - MPI terminology for remotely accessible memory is a “**window**”
  - A group of processes collectively create a “window”
- Once a memory region is declared as remotely accessible, all processes in the window can read/write data to this memory without explicitly synchronizing with the target process

# Window creation models

- Four models exist
  - MPI\_WIN\_ALLOCATE
    - You want to create a buffer and directly make it remotely accessible
  - MPI\_WIN\_CREATE
    - You already have an allocated buffer that you would like to make remotely accessible
  - MPI\_WIN\_CREATE\_DYNAMIC
    - You don't have a buffer yet, but will have one in the future
    - You may want to dynamically add/remove buffers to/from the window
  - MPI\_WIN\_ALLOCATE\_SHARED
    - You want multiple processes on the same node to share a buffer

# MPI\_WIN\_ALLOCATE

```
MPI_Win_allocate(MPI_Aint size, int disp_unit,  
                 MPI_Info info, MPI_Comm comm, void *baseptr,  
                 MPI_Win *win)
```

- Create a remotely accessible memory region in an RMA window
  - Only data exposed in a window can be accessed with RMA ops.
- Arguments:
  - size - size of local data in bytes (nonnegative integer)
  - disp\_unit - local unit size for displacements, in bytes (positive integer)
  - info - info argument (handle)
  - comm - communicator (handle)
  - baseptr - pointer to exposed local data
  - win - window (handle)

# Example with MPI\_WIN\_ALLOCATE

```
int main(int argc, char ** argv)
{
    int *a;      MPI_Win win;

    MPI_Init(&argc, &argv);

    /* collectively create remote accessible memory in a window */
    MPI_Win_allocate(1000*sizeof(int), sizeof(int), MPI_INFO_NULL,
                     MPI_COMM_WORLD, &a, &win);

    /* Array 'a' is now accessible from all processes in
       * MPI_COMM_WORLD */

    MPI_Win_free(&win);

    MPI_Finalize(); return 0;
}
```

# MPI\_WIN\_CREATE

```
MPI_Win_create(void *base, MPI_Aint size,  
               int disp_unit, MPI_Info info,  
               MPI_Comm comm, MPI_Win *win)
```

- Expose a region of memory in an RMA window
  - Only data exposed in a window can be accessed with RMA ops.
- Arguments:
  - base            - pointer to local data to expose
  - size            - size of local data in bytes (nonnegative integer)
  - disp\_unit      - local unit size for displacements, in bytes (positive integer)
  - info            - info argument (handle)
  - comm           - communicator (handle)
  - win            - window (handle)

# Example with MPI\_WIN\_CREATE

```
int main(int argc, char ** argv)
{
    int *a;      MPI_Win win;

    MPI_Init(&argc, &argv);

    /* create private memory */
    MPI_Alloc_mem(1000*sizeof(int), MPI_INFO_NULL, &a);
    /* use private memory like you normally would */
    a[0] = 1;  a[1] = 2;

    /* collectively declare memory as remotely accessible */
    MPI_Win_create(a, 1000*sizeof(int), sizeof(int),
                   MPI_INFO_NULL, MPI_COMM_WORLD, &win);

    /* Array 'a' is now accessibly by all processes in
       * MPI_COMM_WORLD */

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

# MPI\_WIN\_CREATE\_DYNAMIC

```
MPI_Win_create_dynamic(MPI_Info info, MPI_Comm comm,  
                        MPI_Win *win)
```

- Create an RMA window, to which data can later be attached
  - Only data exposed in a window can be accessed with RMA ops
- Initially “empty”
  - Application can dynamically attach/detach memory to this window by calling MPI\_Win\_attach/detach
  - Application can access data on this window only after a memory region has been attached
- Window origin is MPI\_BOTTOM
  - Displacements are segment addresses relative to MPI\_BOTTOM
  - Must tell others the displacement after calling attach

# Example with MPI\_WIN\_CREATE\_DYNAMIC

```
int main(int argc, char ** argv)
{
    int *a;      MPI_Win win;

    MPI_Init(&argc, &argv);
    MPI_Win_create_dynamic(MPI_INFO_NULL, MPI_COMM_WORLD, &win);

    /* create private memory */
    a = (int *) malloc(1000 * sizeof(int));
    /* use private memory like you normally would */
    a[0] = 1;  a[1] = 2;

    /* locally declare memory as remotely accessible */
    MPI_Win_attach(win, a, 1000*sizeof(int));

    /* Array 'a' is now accessible from all processes */

    /* undeclare remotely accessible memory */
    MPI_Win_detach(win, a);  free(a);
    MPI_Win_free(&win);

    MPI_Finalize(); return 0;
}
```



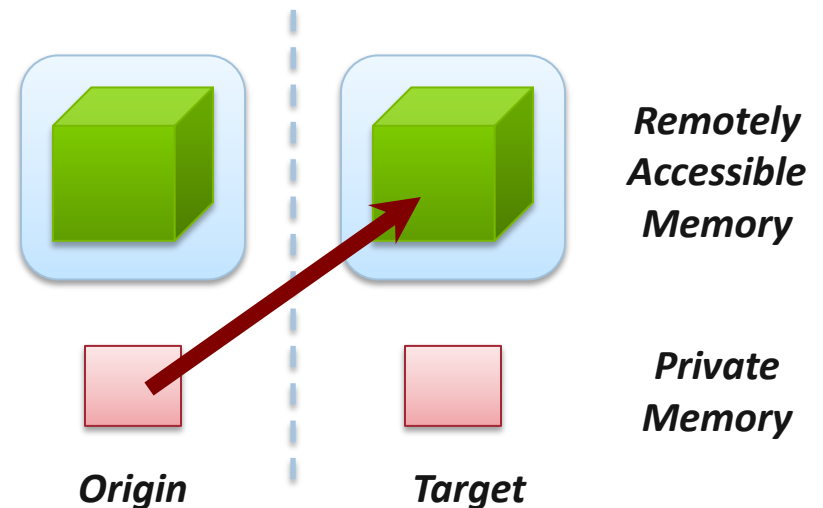
# Data movement

- MPI provides ability to read, write and atomically modify data in remotely accessible memory regions
  - MPI\_PUT
  - MPI\_GET
  - MPI\_ACCUMULATE (atomic)
  - MPI\_GET\_ACCUMULATE (atomic)
  - MPI\_COMPARE\_AND\_SWAP (atomic)
  - MPI\_FETCH\_AND\_OP (atomic)

# Data movement: *Put*

```
MPI_Put(const void *origin_addr, int origin_count,  
        MPI_Datatype origin_dtype, int target_rank,  
        MPI_Aint target_disp, int target_count,  
        MPI_Datatype target_dtype, MPI_Win win)
```

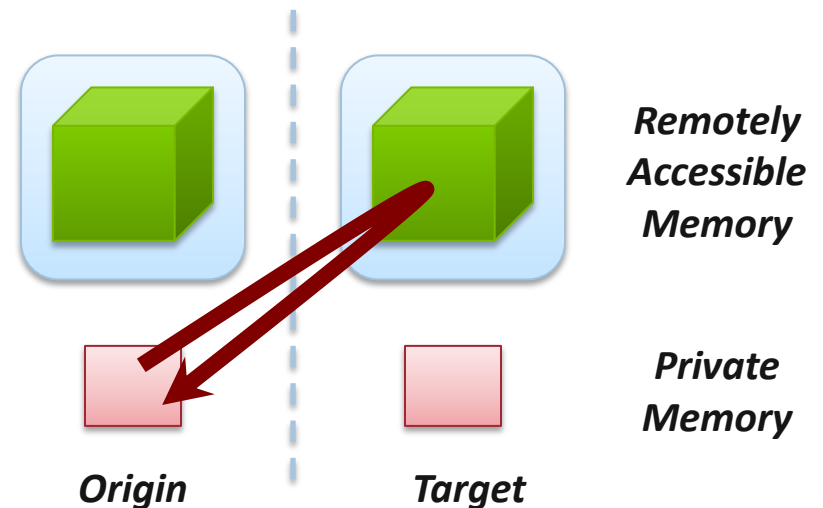
- Move data from origin, to target
- Separate data description triples for **origin** and **target**



# Data movement: *Get*

```
MPI_Get(void *origin_addr, int origin_count,  
        MPI_Datatype origin_dtype, int target_rank,  
        MPI_Aint target_disp, int target_count,  
        MPI_Datatype target_dtype, MPI_Win win)
```

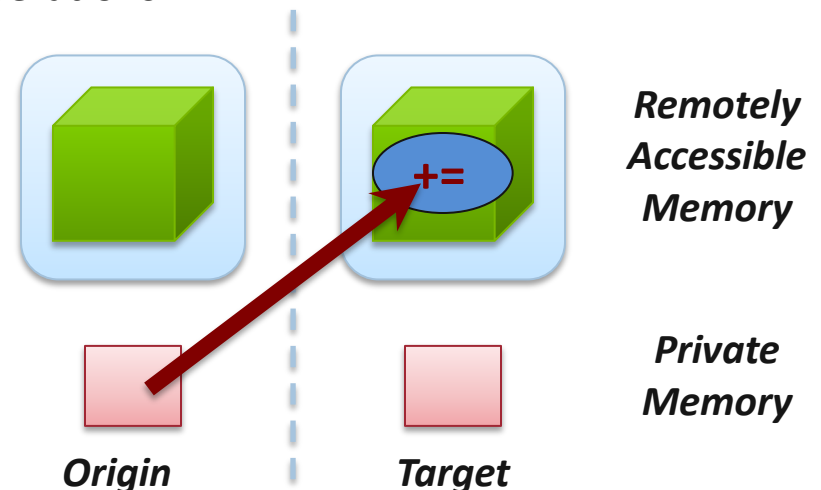
- Move data to origin, from target
- Separate data description triples for **origin** and **target**



# Atomic Data Aggregation: *Accumulate*

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

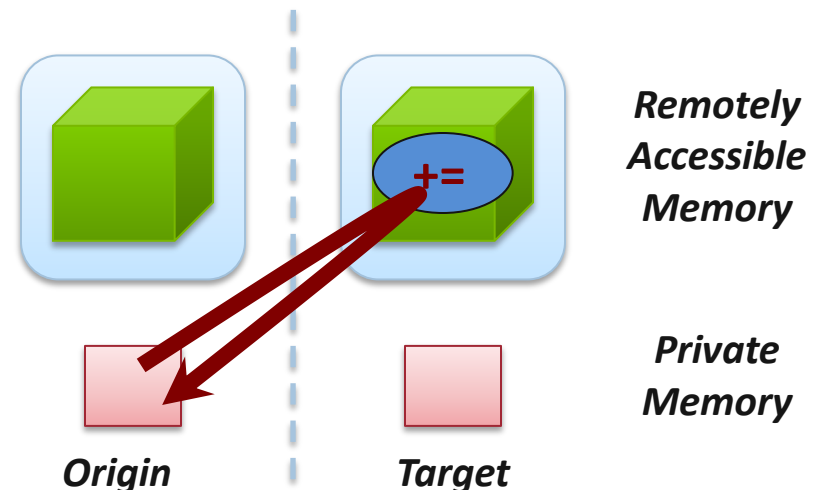
- Atomic update operation, similar to a put
  - Reduces origin and target data into target buffer using op argument as combiner
  - Op = MPI\_SUM, MPI\_PROD, MPI\_OR, MPI\_REPLACE, MPI\_NO\_OP, ...
  - Predefined ops only, no user-defined operations
- Different data layouts between target/origin OK
  - Basic type elements must match
- Op = MPI\_REPLACE
  - Implements  $f(a,b)=b$
  - Atomic PUT



# Atomic Data Aggregation: *Get Accumulate*

```
MPI_Get_accumulate(const void *origin_addr,  
                  int origin_count, MPI_Datatype origin_dtype,  
                  void *result_addr, int result_count,  
                  MPI_Datatype result_dtype, int target_rank,  
                  MPI_Aint target_disp, int target_count,  
                  MPI_Datatype target_dtype, MPI_Op op, MPI_Win win)
```

- Atomic read-modify-write
  - Op = MPI\_SUM, MPI\_PROD, MPI\_OR, MPI\_REPLACE, MPI\_NO\_OP, ...
  - Predefined ops only
- Result stored in target buffer
- Original data stored in result buf
- Different data layouts between target/origin OK
  - Basic type elements must match
- Atomic get with MPI\_NO\_OP
- Atomic swap with MPI\_REPLACE



# Atomic Data Aggregation: *CAS and FOP*

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

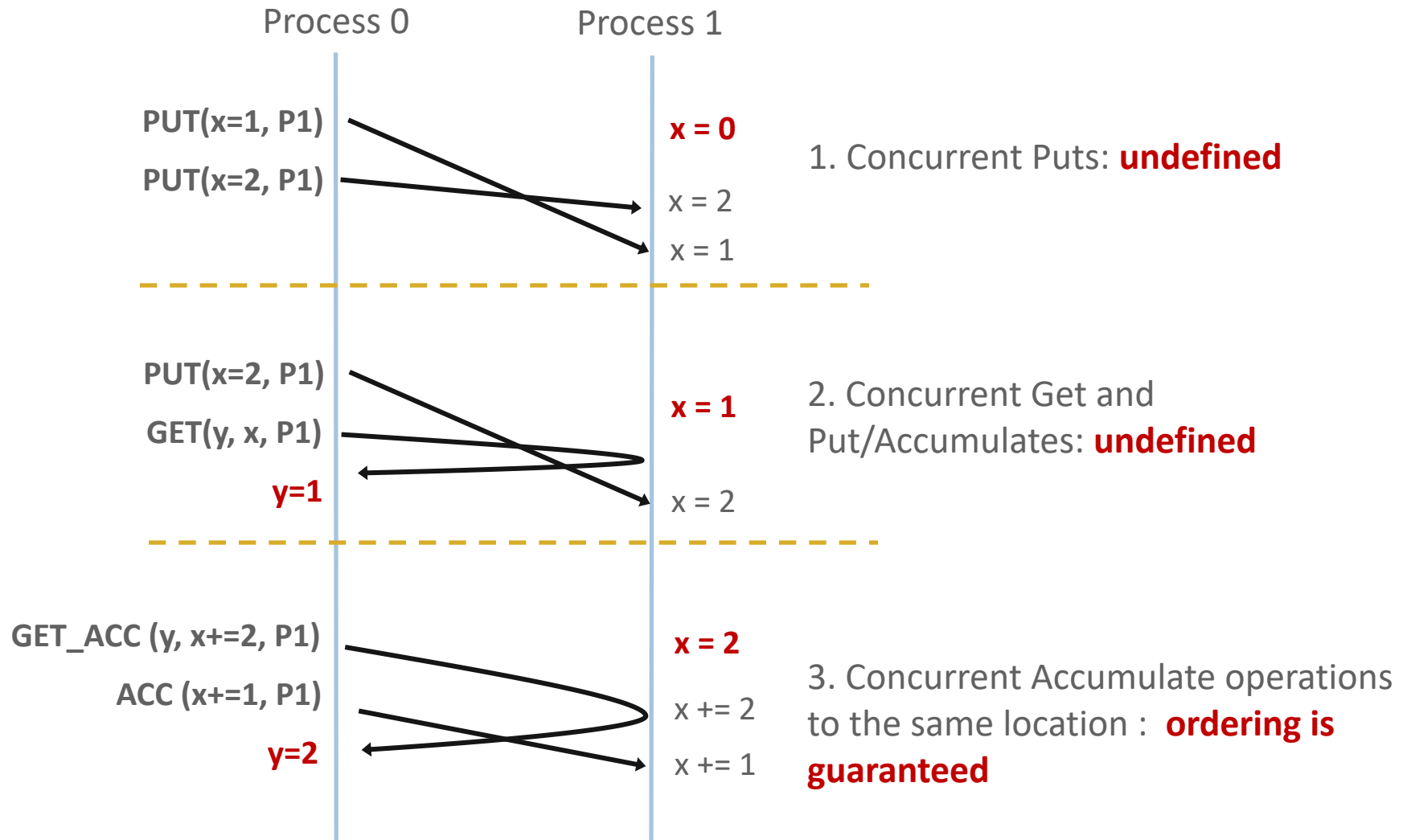
```
MPI_Compare_and_swap(const void *origin_addr,  
                    const void *compare_addr, void *result_addr,  
                    MPI_Datatype dtype, int target_rank,  
                    MPI_Aint target_disp, MPI_Win win)
```

- FOP: Simpler version of MPI\_Get\_accumulate
  - All buffers share a single predefined datatype
  - No count argument (it's always 1)
  - Simpler interface allows hardware optimization
- CAS: Atomic swap if target value is equal to compare value

# Ordering of Operations in MPI RMA

- No guaranteed ordering for Put/Get operations
- Result of concurrent Puts to the same location undefined
- Result of Get concurrent Put/Accumulate undefined
  - Can be garbage in both cases
- Result of concurrent accumulate operations to the same location are defined according to the order in which they occurred
  - Atomic put: Accumulate with `op = MPI_REPLACE`
  - Atomic get: `Get_accumulate` with `op = MPI_NO_OP`
- Accumulate operations from a given process are ordered by default
  - User can tell the MPI implementation that (s)he does not require ordering as optimization hint
  - You can ask for only the needed orderings: RAW (read-after-write), WAR, RAR, or WAW

# Examples with operation ordering





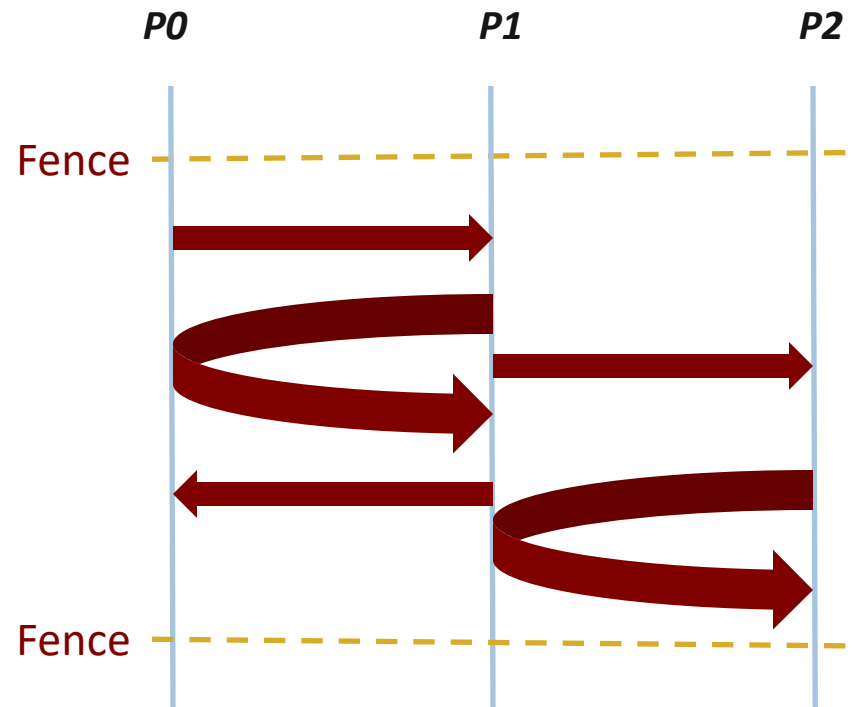
# RMA Synchronization Models

- RMA data access model
  - When is a process allowed to read/write remotely accessible memory?
  - When is data written by process X is available for process Y to read?
  - RMA synchronization models define these semantics
- Three synchronization models provided by MPI:
  - Fence (active target)
  - Post-start-complete-wait (generalized active target; rarely used now)
  - Lock/Unlock (passive target)
- Data accesses occur within “epochs”
  - *Access epochs*: contain a set of operations issued by an origin process
  - *Exposure epochs*: enable remote processes to update a target’s window
  - Epochs define ordering and completion semantics
  - Synchronization models provide mechanisms for establishing epochs
    - E.g., starting, ending, and synchronizing epochs

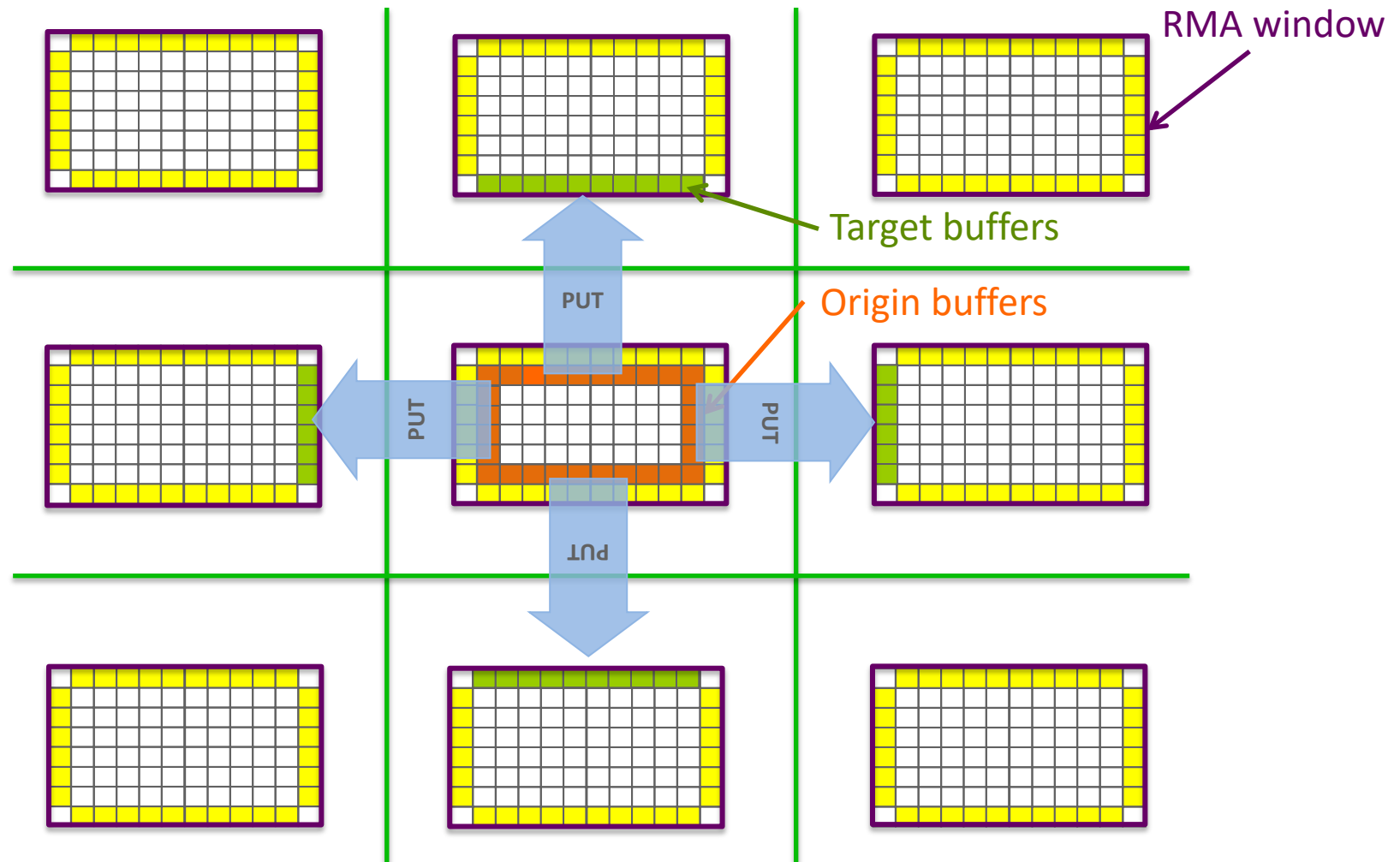
# Fence: Active Target Synchronization

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

- Collective synchronization model
- Starts *and* ends access and exposure epochs on all processes in the window
- All processes in group of “win” do an MPI\_WIN\_FENCE to open an epoch
- Everyone can issue PUT/GET operations to read/write data
- Everyone does an MPI\_WIN\_FENCE to close the epoch
- All operations complete at the second fence synchronization



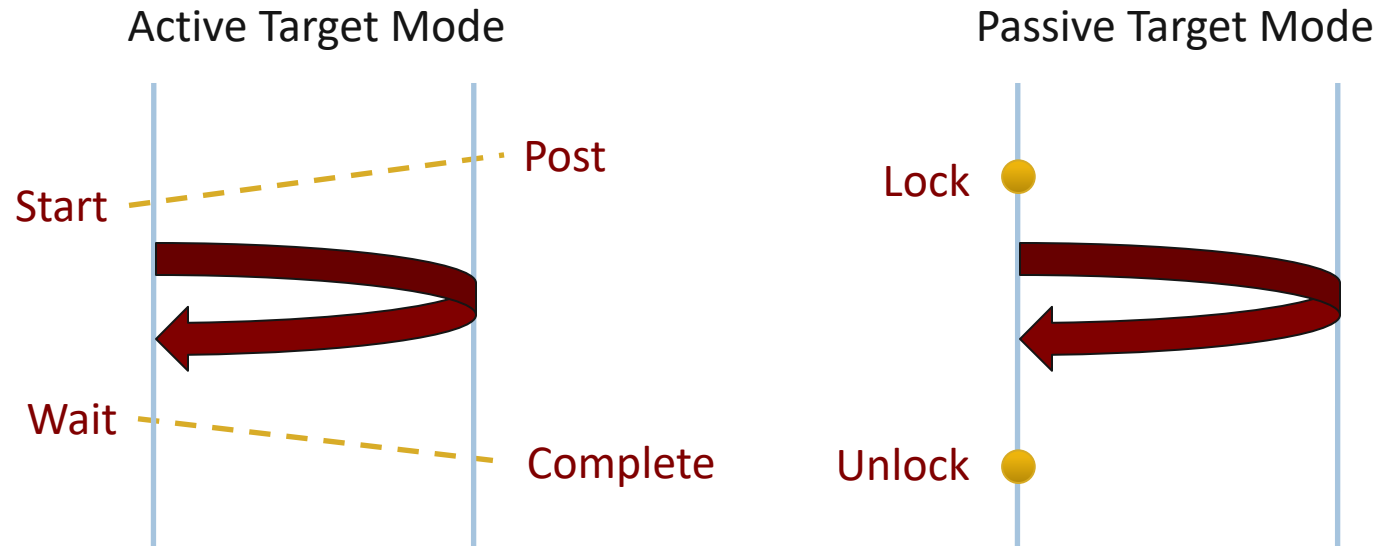
# Implementing Stencil Computation with RMA Fence



# Code Example

- *Code example from the examples set*

# Lock/Unlock: Passive Target Synchronization



- Passive mode: One-sided, *asynchronous* communication
  - Target does **not** participate in communication operation
- Shared memory-like model

# Passive Target Synchronization

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

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

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

- Lock/Unlock: Begin/end passive mode epoch
  - Target process does not make a corresponding MPI call
  - Can initiate multiple passive target epochs to different processes
  - Concurrent epochs to same process not allowed (affects threads)
- Lock type
  - SHARED: Other processes using shared can access concurrently
  - EXCLUSIVE: No other processes can access concurrently
- Flush: Remotely complete RMA operations to the target process
  - After completion, data can be read by target process or a different process
- Flush\_local: Locally complete RMA operations to the target process

# Advanced Passive Target Synchronization

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

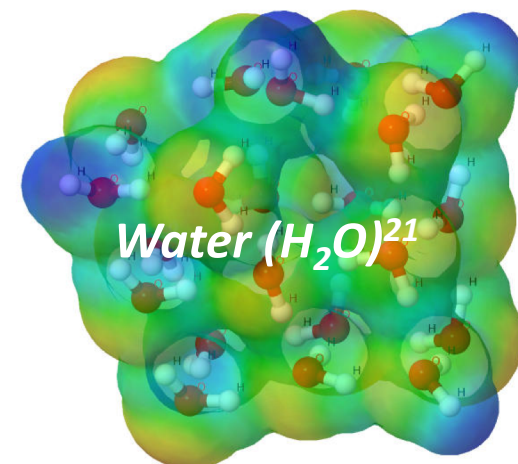
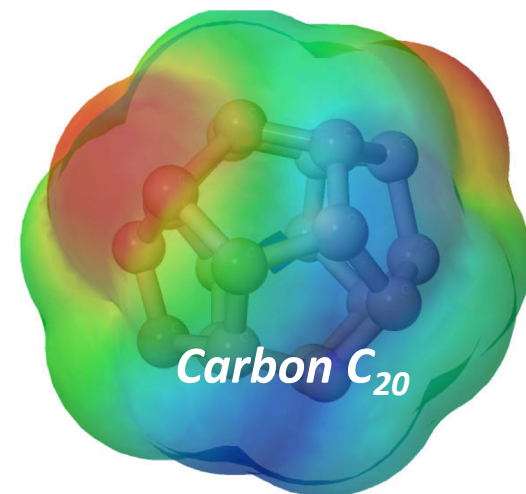
```
MPI_Win_unlock_all(MPI_Win win)
```

```
MPI_Win_flush_all/flush_local_all(MPI_Win win)
```

- Lock\_all: Shared lock, passive target epoch to all other processes
  - Expected usage is long-lived: lock\_all, put/get, flush, ..., unlock\_all
- Flush\_all – remotely complete RMA operations to all processes
- Flush\_local\_all – locally complete RMA operations to all processes

# NWChem [1]

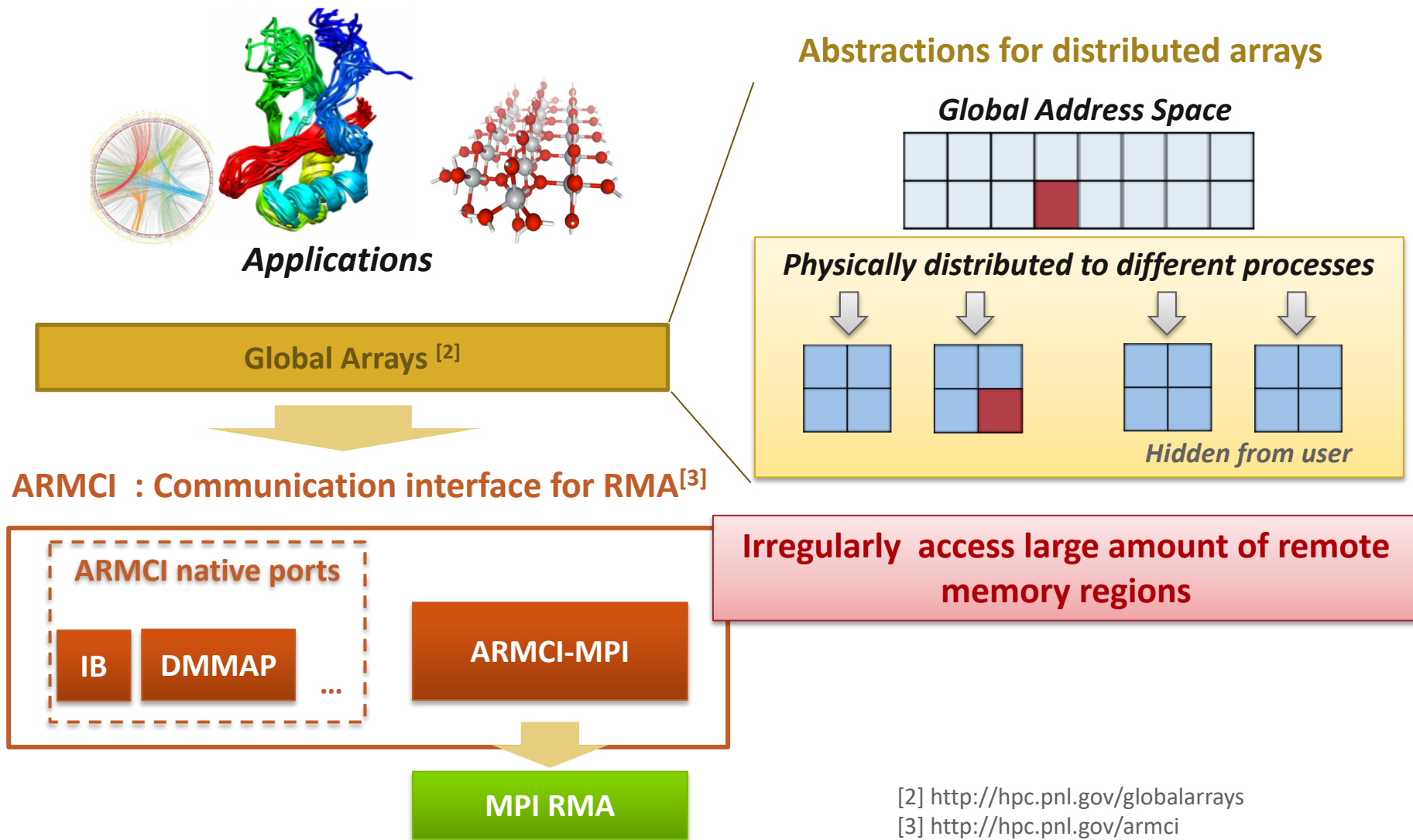
- High performance computational chemistry application suite
- Quantum level simulation of molecular systems
  - Very expensive in computation and data movement, so is used for small systems
  - Larger systems use molecular level simulations
- Composed of many simulation capabilities
  - Molecular Electronic Structure
  - Quantum Mechanics/Molecular Mechanics
  - Pseudo potential Plane-Wave Electronic Structure
  - Molecular Dynamics
- Very large code base
  - 4M LOC; Total investment of ~200M \$ to date



[1] M. Valiev, E.J. Bylaska, N. Govind, K. Kowalski, T.P. Straatsma, H.J.J. van Dam, D. Wang, J. Nieplocha, E. Apra, T.L. Windus, W.A. de Jong, "NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations" Comput. Phys. Commun. 181, 1477 (2010)



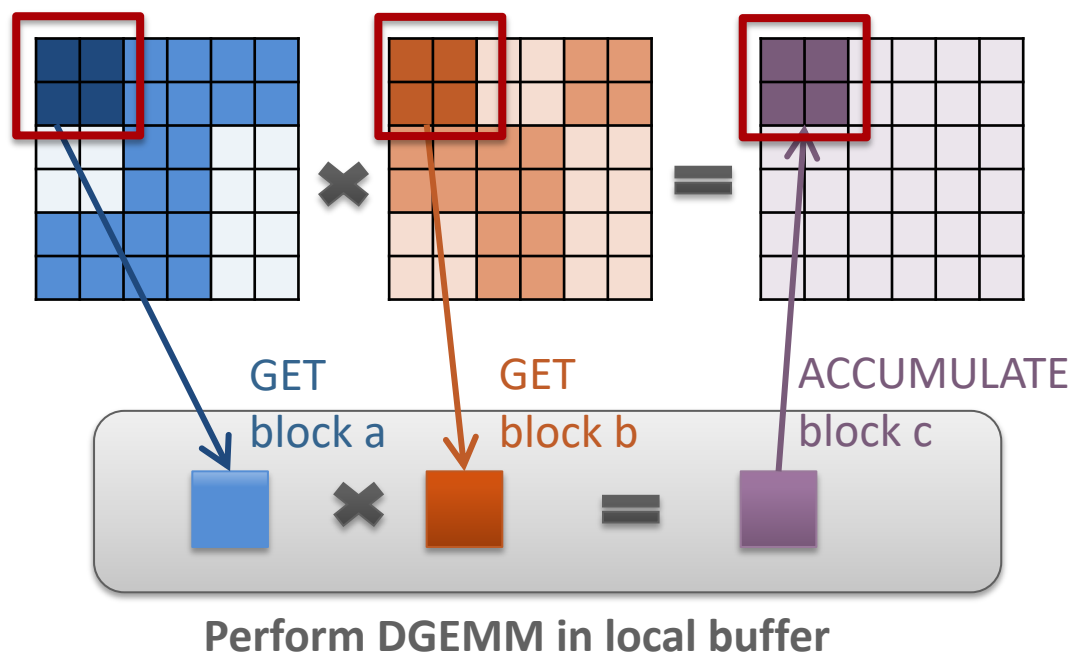
# NWChem Communication Runtime



# Get-Compute-Update

- Typical Get-Compute-Update mode in GA programming

All of the blocks are non-contiguous data



*Pseudo code*

```
for i in I blocks:
  for j in J blocks:
    for k in K blocks:
      GET block a from A
      GET block b from B
      c += a * b /*computing*/
    end do
    ACC block c to C
    NXTASK
  end do
end do
```

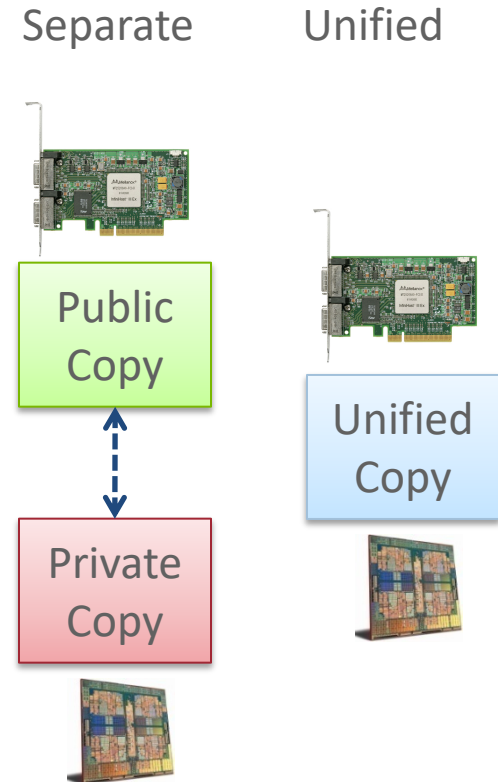
*Mock figure showing 2D DGEMM with block-sparse computations. In reality, NWChem uses 6D tensors.*

# Which synchronization mode should I use, when?

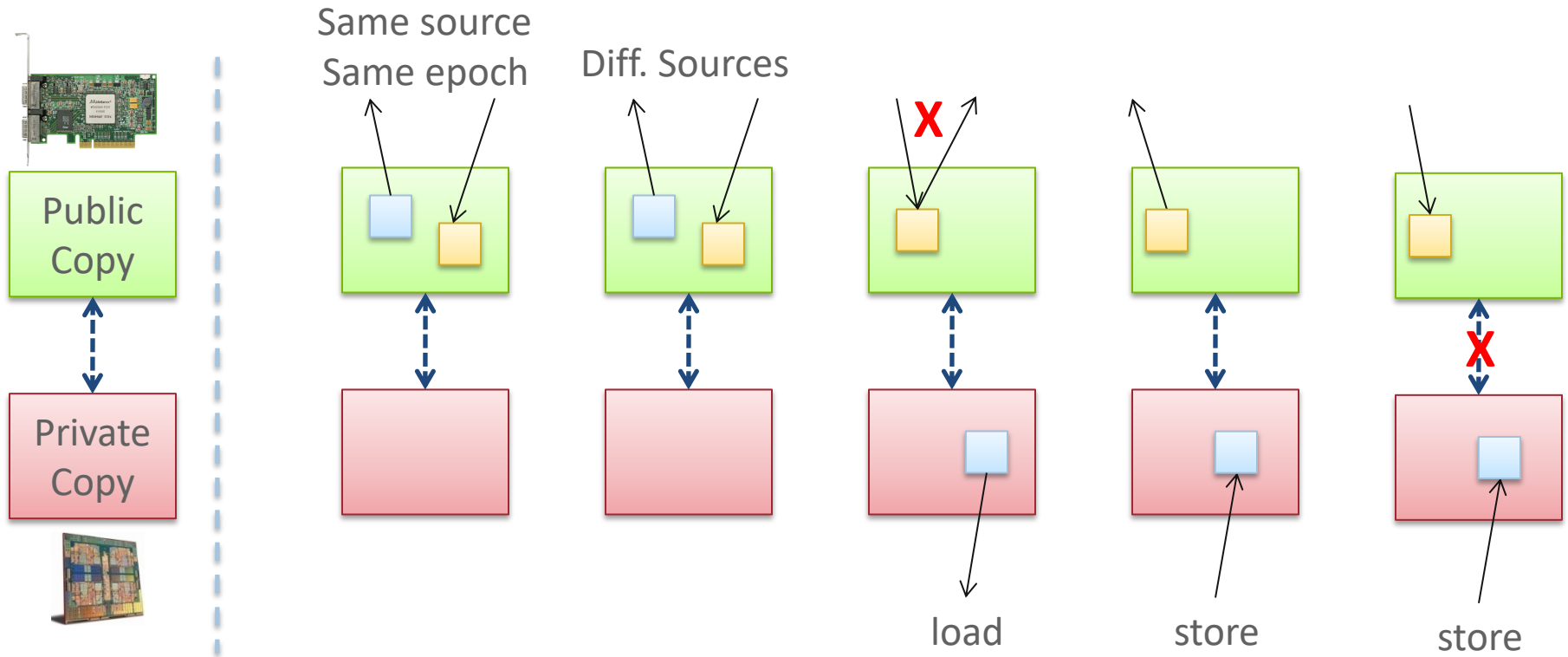
- RMA communication often has low overheads versus send/recv
  - Two-sided: Matching, queuing, buffering, unexpected receives, etc...
  - One-sided: No matching, no buffering, always ready to receive (but must separately sync the communication)
  - Direct use of RDMA provided by high-speed interconnects (e.g. InfiniBand)
    - Good two-sided implementations will also use RDMA, but must first match messages
- Active mode: bulk synchronization
  - E.g. ghost cell exchange
- Passive mode: asynchronous data movement
  - Useful when dataset is large, requiring memory of multiple nodes
  - Also, when data access and synchronization pattern is dynamic
  - Common use case: distributed, shared arrays
- Passive target locking mode
  - Lock/unlock – Useful when exclusive epochs are needed
  - Lock\_all/unlock\_all – Useful when only shared epochs are needed

# MPI RMA Memory Model

- MPI-3 provides two memory models: separate and unified
- MPI-2: Separate Model
  - Logical public and private copies
  - MPI provides software coherence between window copies
  - Extremely portable, to systems that don't provide hardware coherence
- MPI-3: New Unified Model
  - Single copy of the window
  - System must provide coherence
  - Superset of separate semantics
    - E.g. allows concurrent local/remote access
  - Provides access to full performance potential of hardware
  - Matches typical SMP nodes

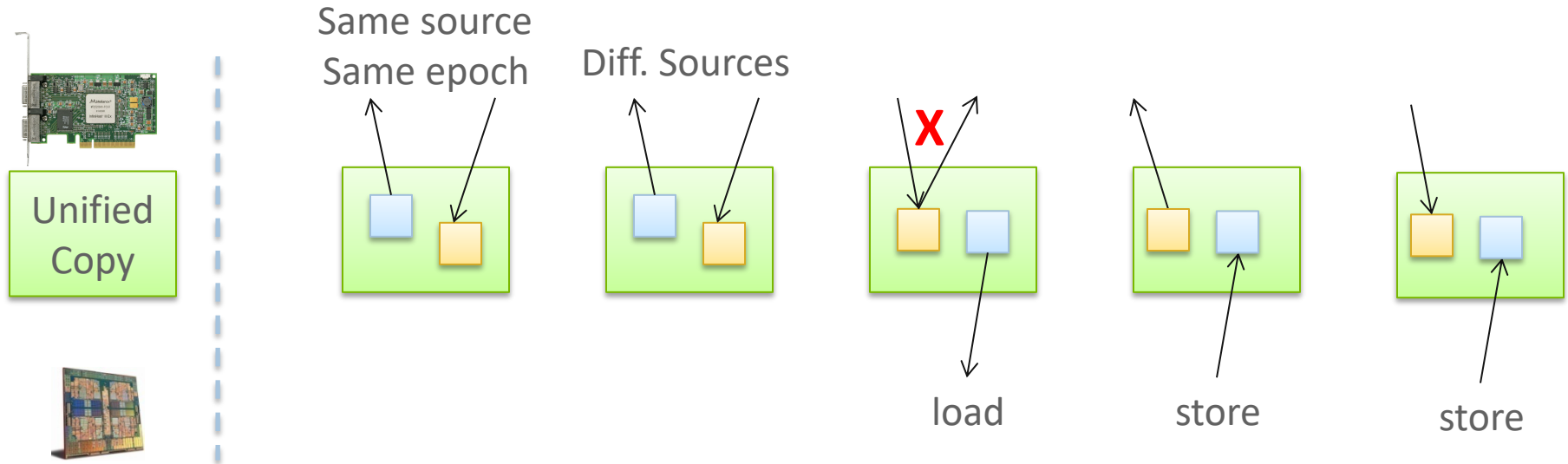


# MPI RMA Memory Model (separate windows)



- Very portable, compatible with non-coherent memory systems
- Limits concurrent accesses to enable software coherence

# MPI RMA Memory Model (unified windows)



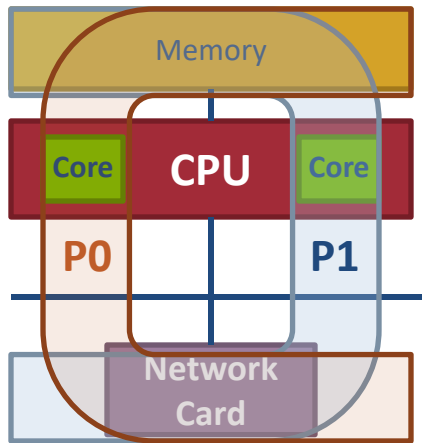
- Allows concurrent local/remote accesses
- Concurrent, conflicting operations are allowed (not invalid)
  - Outcome is not defined by MPI (defined by the hardware)
- Can enable better performance by reducing synchronization

# Advanced Topics: Hybrid Programming with Threads, Shared Memory, and Accelerators

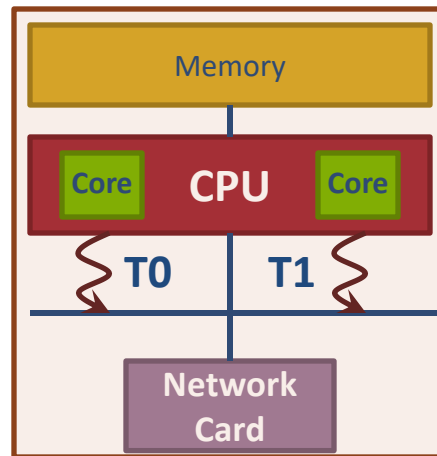
# Hybrid MPI + X : Most Popular Forms

## MPI + X

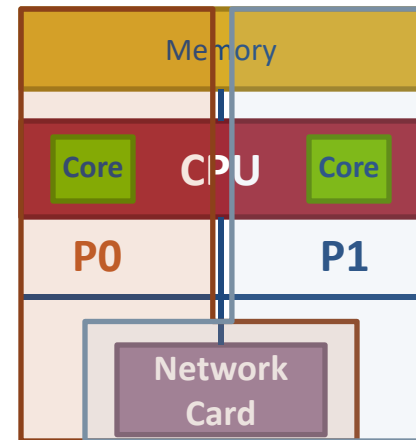
### MPI Process



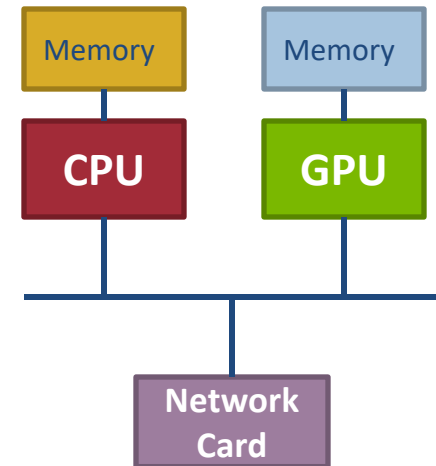
MPI + 0



MPI + Threads



MPI +  
Shared Memory



MPI + ACC



# MPI + Threads

# Why Hybrid MPI+X? Towards Strong Scaling (1/3)

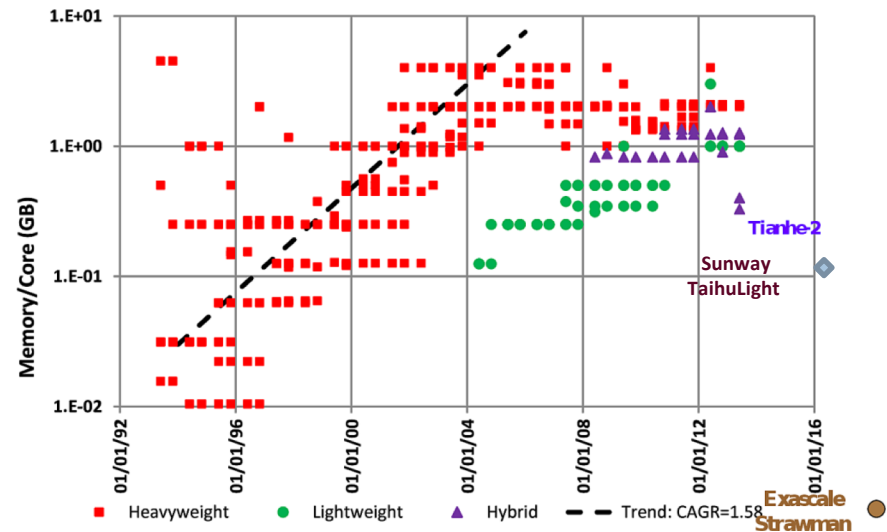
- Strong scaling applications is increasing in importance

- Hardware limitations: not all resources scale at the same rate as cores (e.g., memory capacity, network resources)
- Desire to solve the same problem faster on a bigger machine

- Nek5000, HACC, LAMMPS

- Strong scaling pure MPI applications is getting harder

- On-node communication is costly compared to load/stores
- $O(Px)$  communication patterns (e.g., All-to-all) costly

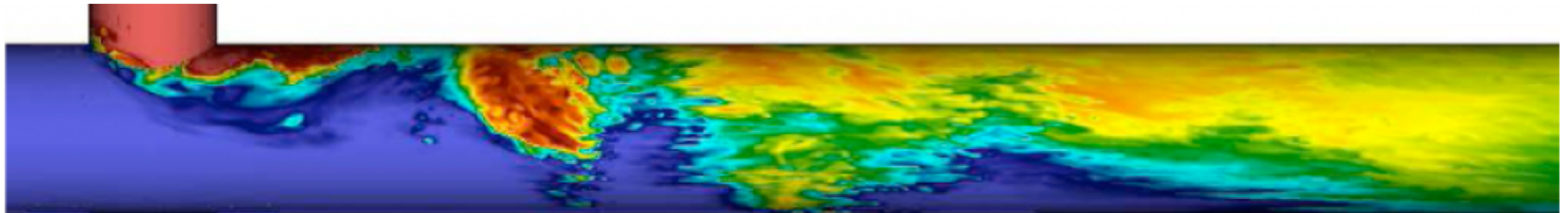


Evolution of the memory capacity per core in the Top500 list (Peter Kogge. PIM & memory: The need for a revolution in architecture.)

## Why Hybrid MPI+X? Towards Strong Scaling (2/3)

- MPI+X benefits ( $X = \{\text{threads, MPI shared-memory, etc.}\}$ )
  - Less memory hungry (MPI runtime consumption,  $O(P)$  data structures, etc.)
  - Load/stores to access memory instead of message passing
  - $P$  is reduced by constant  $C$  (#cores/process) for  $O(Px)$  communication patterns
- Example 1: the Nek5000 team is working at the strong scaling limit

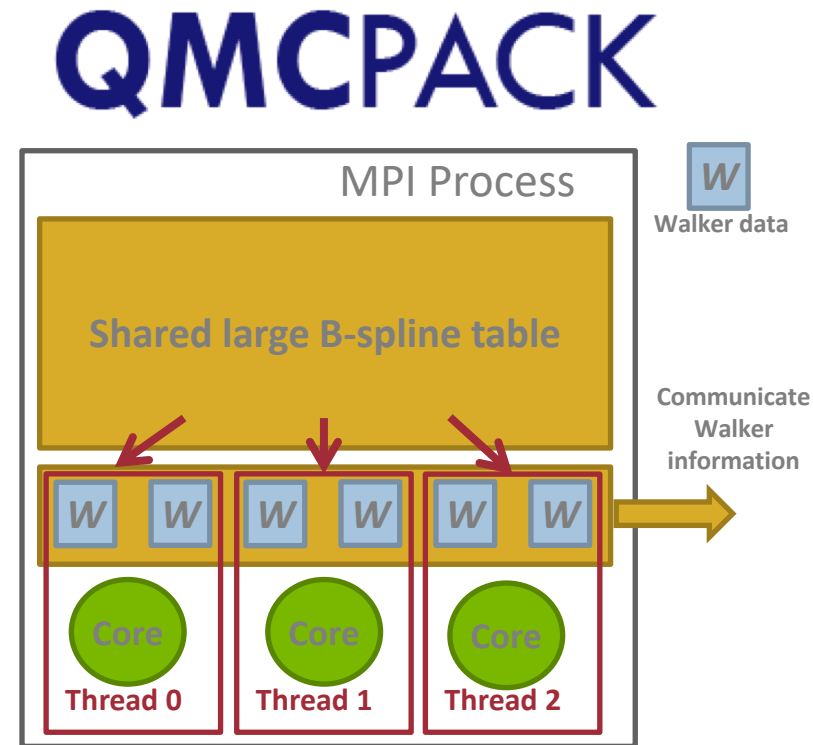
### Nek5000



# Why Hybrid MPI+X? Towards Strong Scaling (3/3)

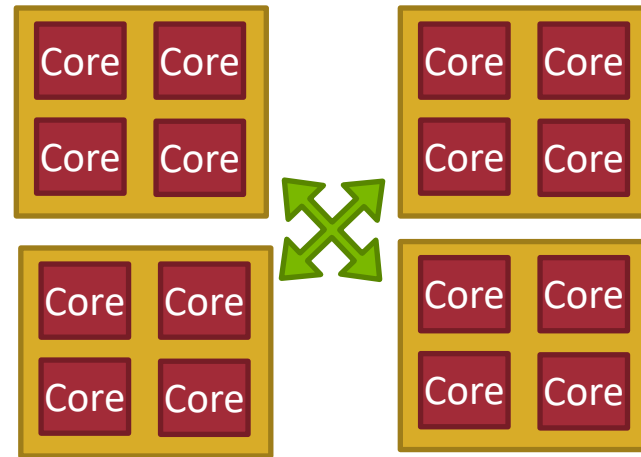
## ■ Example 2: Quantum Monte Carlo Simulation (QMCPACK)

- Size of the physical system to simulate is bound by memory capacity [1]
- Memory space dominated by large interpolation tables (typically several GB of storage)
- Threads are used to share those tables
- Memory for communication buffers must be kept low to allow simulation of larger and highly detailed simulations.



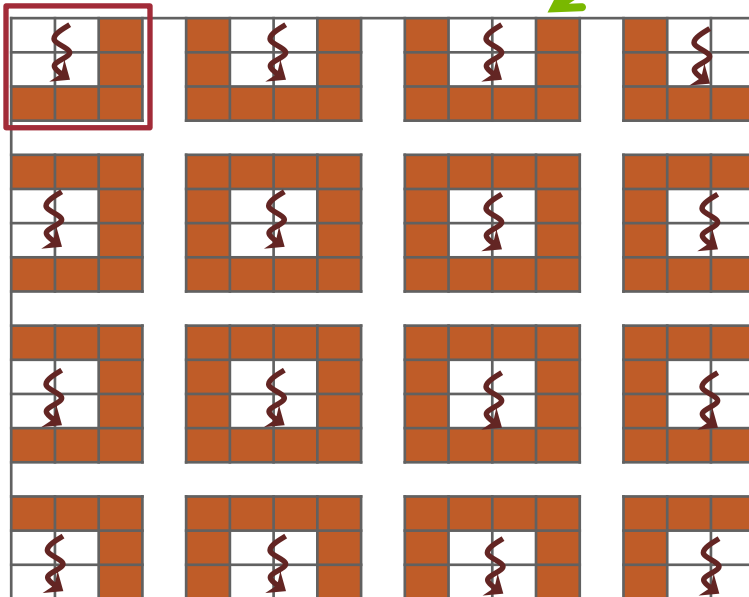
[1] Kim, Jeongnim, et al. "Hybrid algorithms in quantum Monte Carlo." Journal of Physics, 2012.

# MPI + Threads: How To? (1/3)



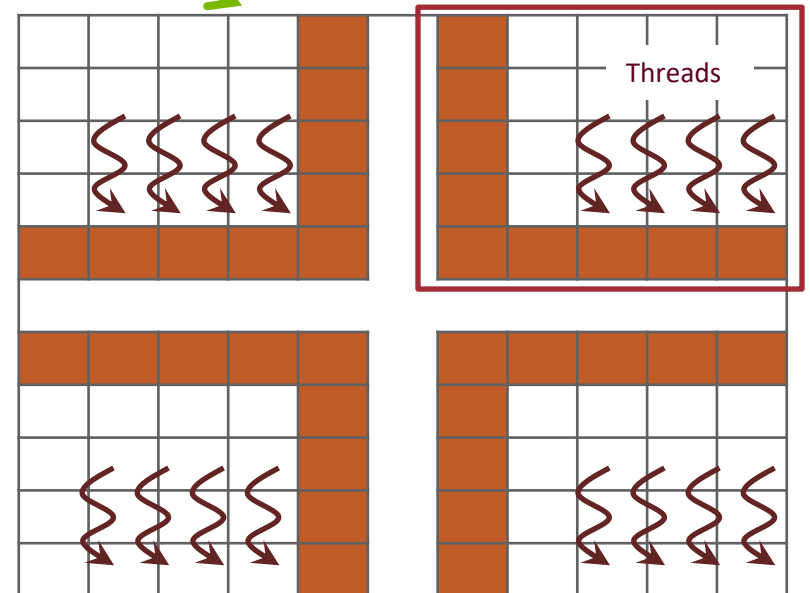
Multi- or Many-core Nodes

MPI Process



MPI only

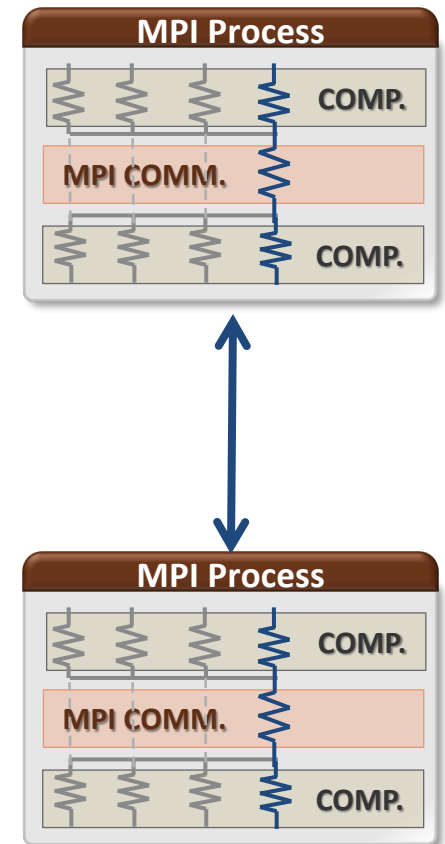
MPI Process



MPI + Threads

## MPI + Threads: How To? (2/3)

- 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.



# MPI + Threads: How To? (3/3)

MPI + Threads



**Interoperability**

Interoperation or thread levels:

- MPI\_THREAD\_SINGLE
  - No additional threads
- MPI\_THREAD\_FUNNELED
  - Master thread communication only
- MPI\_THREAD\_SERIALIZED
  - Threaded communication serialized
- MPI\_THREAD\_MULTIPLE
  - No restrictions

**•Restriction**

**•Low  
Thread-  
Safety Costs**

**•Flexibility**

**•High  
Thread-  
Safety Costs**

# MPI's Four Levels of Thread Safety

- MPI defines four levels of thread safety -- these are commitments the application makes to the MPI
- Thread levels are in increasing order
  - If an application works in FUNNELED mode, it can work in SERIALIZED
- MPI defines an alternative to MPI\_Init
  - **MPI\_Init\_thread**(int argc, char \*\*argv, int requested, int \*provided):  
*Application specifies level it needs; MPI implementation returns level it supports*



# MPI\_THREAD\_SINGLE

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

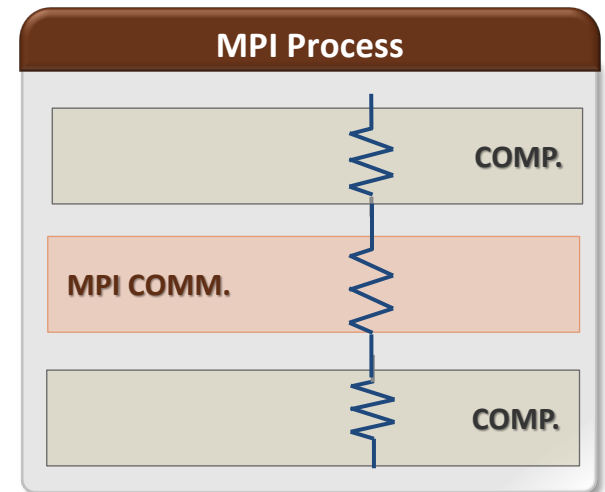
```
int buf[100];
int main(int argc, char ** argv)
{
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

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

    /* Do MPI stuff */

    MPI_Finalize();

    return 0;
}
```



# MPI\_THREAD\_FUNNELED

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

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

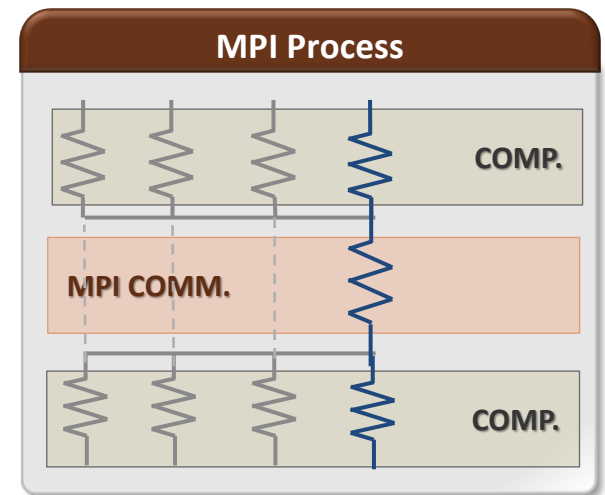
    MPI_Init_thread(&argc, &argv,
        MPI_THREAD_FUNNELED, &provided);
    if (provided < MPI_THREAD_FUNNELED)
        MPI_Abort(MPI_COMM_WORLD, 1);

    for (i = 0; i < 100; i++)
        pthread_create(..., func, (void*)i);
    for (i = 0; i < 100; i++)
        pthread_join(...);

    /* Do MPI stuff */

    MPI_Finalize();
    return 0;
}
```

```
void* func(void* arg) {
    int i = (int)arg;
    compute(buf[i]);
    return 0;
}
```



# MPI\_THREAD\_SERIALIZED

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

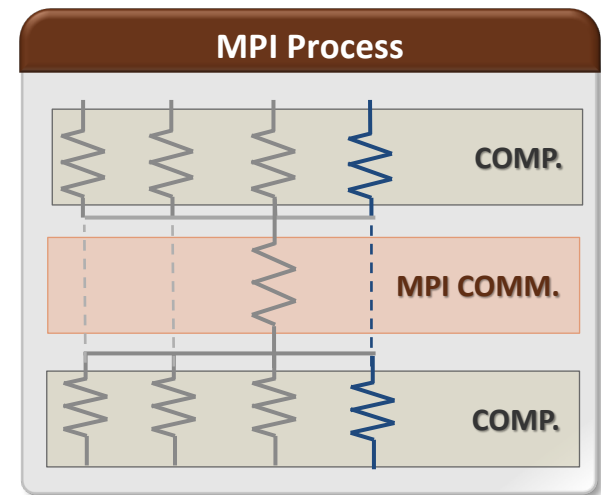
```
int buf[100];
int main(int argc, char ** argv)
{
    int provided;
    pthread_mutex_t mutex;

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

    for (i = 0; i < 100; i++)
        pthread_create(..., func, (void*)i);
    for (i = 0; i < 100; i++)
        pthread_join(...);

    MPI_Finalize();
    return 0;
}
```

```
void* func(void* arg) {
    int i = (int)arg;
    compute(buf[i]);
    pthread_mutex_lock(&mutex);
    /* Do MPI stuff */
    pthread_mutex_unlock(&mutex);
    return 0;
}
```



# MPI\_THREAD\_MULTIPLE

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

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

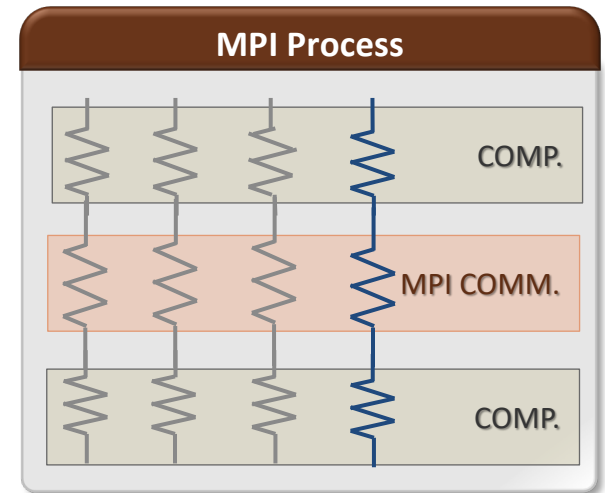
    MPI_Init_thread(&argc, &argv,
        MPI_THREAD_MULTIPLE, &provided);
    if (provided < MPI_THREAD_SERIALIZED)
        MPI_Abort(MPI_COMM_WORLD, 1);

    for (i = 0; i < 100; i++)
        pthread_create(..., func, (void*)i);

    MPI_Finalize();
    return 0;
}
```

```
void* func(void* arg) {
    int i = (int)arg;
    compute(buf[i]);

    /* Do MPI stuff */
    ...
    return 0;
}
```



# Threads and MPI

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

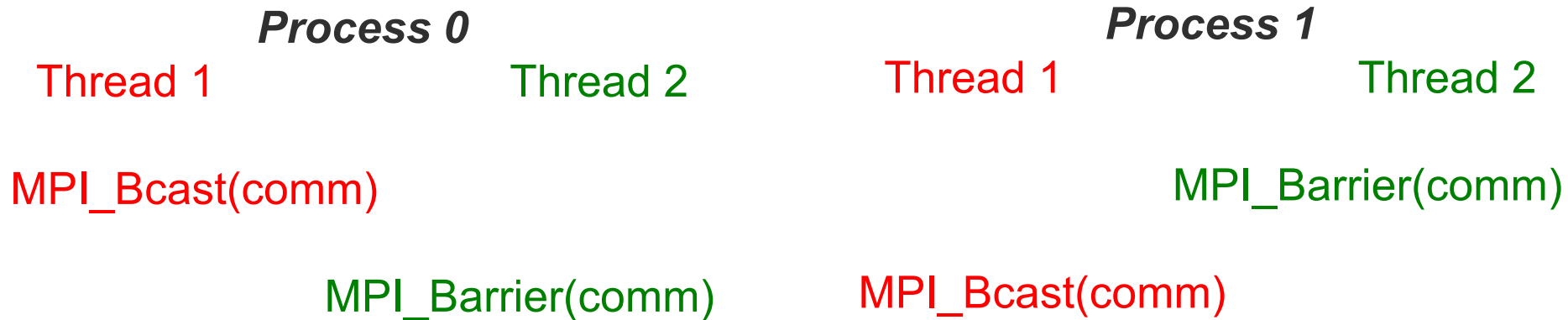
# MPI Semantics and MPI\_THREAD\_MULTIPLE

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

# Ordering in MPI\_THREAD\_MULTIPLE: Incorrect Example with Collectives

	<i>Process 0</i>	<i>Process 1</i>
<i>Thread 0</i>	MPI_Bcast(comm)	MPI_Bcast(comm)
<i>Thread 1</i>	MPI_Barrier(comm)	MPI_Barrier(comm)

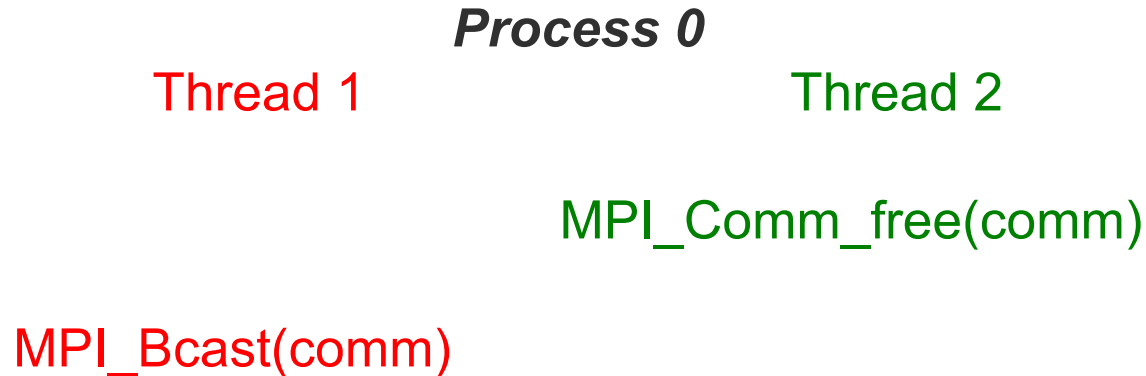
# Ordering in MPI\_THREAD\_MULTIPLE: Incorrect Example with Collectives



- P0 and P1 can have different orderings of Bcast and Barrier
- 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



# Ordering in MPI\_THREAD\_MULTIPLE: Incorrect Example with Object Management



- The user has to make sure that one thread is not using an object while another thread is freeing it
  - This is essentially an ordering issue; the object might get freed before it is used

# Blocking Calls in MPI\_THREAD\_MULTIPLE: Correct Example

	<i>Process 0</i>	<i>Process 1</i>
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
- They probably support MPI\_THREAD\_FUNNELED even if they don't admit it.
  - Does require thread-safety for some system routines (e.g. malloc)
  - On most systems `-pthread` will guarantee it (OpenMP implies `-pthread`)
- Many (but not all) implementations support THREAD\_MULTIPLE
  - Hard to implement efficiently though (thread synchronization issues)
- Bulk-synchronous OpenMP programs (loops parallelized with OpenMP, communication between loops) only need FUNNELED
  - So don't need “fully thread-safe” MPI for many hybrid programs
  - But watch out for Amdahl's Law!

# Hybrid Programming: Correctness Requirements

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

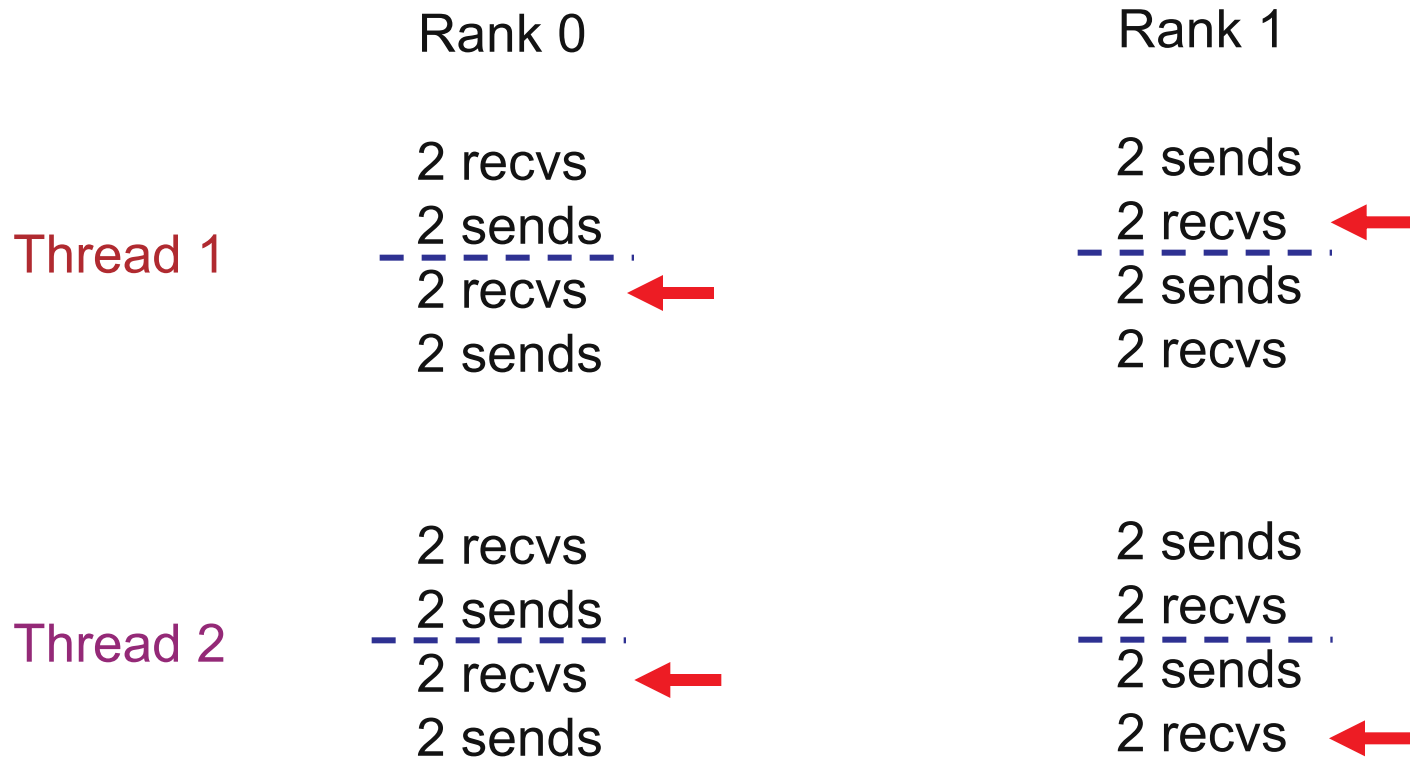
## An Example we encountered

- 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
- We spent several hours trying to debug MPICH 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 < 2; i++)  
            MPI_Send(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD);  
        for (i = 0; i < 2; i++)  
            MPI_Recv(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &stat);  
    }  
    else { /* rank == 0 */  
        for (i = 0; i < 2; i++)  
            MPI_Recv(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD, &stat);  
        for (i = 0; i < 2; i++)  
            MPI_Send(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD);  
    }  
}
```

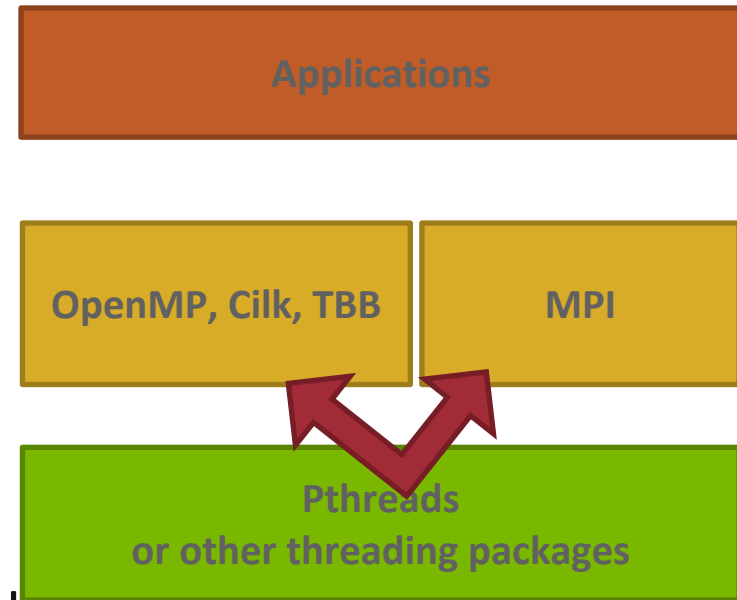
## What Happened



- 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

# MPI+OpenMP correctness semantics

- For OpenMP threads, the MPI+OpenMP correctness semantics are similar to that of MPI+threads
  - Caution: OpenMP iterations need to be carefully mapped to which thread executes them (some schedules in OpenMP make this harder)
- For OpenMP tasks, the general model to use is that an OpenMP thread can execute one or more OpenMP tasks
  - An MPI blocking call should be assumed to block the entire OpenMP thread, so other tasks might not get executed





# OpenMP threads: MPI blocking Calls (1/2)

```
int main(int argc, char ** argv)
{
    MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, &provided);

    #pragma omp parallel for
    for (i = 0; i < 100; i++) {
        if (i % 2 == 0)
            MPI_Send(..., to_myself, ..);
        else
            MPI_Recv(..., from_myself, ..);
    }

    MPI_Finalize();

    return 0;
}
```

*Iteration to OpenMP thread mapping needs to explicitly be handled by the user; otherwise, OpenMP threads might all issue the same operation and deadlock*

## OpenMP threads: MPI blocking Calls (2/2)

```
int main(int argc, char ** argv)
{
    MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, &provided);

#pragma omp parallel
{
    assert(omp_get_num_threads() > 1)
#pragma omp for schedule(static, 1)
    for (i = 0; i < 100; i++) {
        if (i % 2 == 0)
            MPI_Send(.., to_myself, ..);
        else
            MPI_Recv(.., from_myself, ..);
    }
}

MPI_Finalize();

return 0;
}
```

*Either explicit/careful mapping of iterations to threads, or using nonblocking versions of send/rcv would solve this problem*

# OpenMP tasks: MPI blocking Calls (1/5)

```
int main(int argc, char ** argv)
{
    MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, &provided);

#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i < 100; i++) {
        #pragma omp task
        {
            if (i % 2 == 0)
                MPI_Send(.., to_myself, ..);
            else
                MPI_Recv(.., from_myself, ..);
        }
    }
}

MPI_Finalize();
return 0;
}
```

***This can lead to deadlocks. No ordering or progress guarantees in OpenMP task scheduling should be assumed; a blocked task blocks it's thread and tasks can be executed in any order.***

## OpenMP tasks: MPI blocking Calls (2/5)

```
int main(int argc, char ** argv)
{
    MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, &provided);

#pragma omp parallel
{
    #pragma omp taskloop
    for (i = 0; i < 100; i++) {
        if (i % 2 == 0)
            MPI_Send(.., to_myself, ..);
        else
            MPI_Recv(.., from_myself, ..)
    }
}

    MPI_Finalize();
    return 0;
}
```

*Same problem as before.*

## OpenMP tasks: MPI blocking Calls (3/5)

```
int main(int argc, char ** argv)
{
    MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, &provided);

#pragma omp parallel
{
    #pragma omp taskloop
    for (i = 0; i < 100; i++) {
        MPI_Request req;
        if (i % 2 == 0)
            MPI_Isend(.., to_myself, .., &req);
        else
            MPI_Irecv(.., from_myself, .., &req);
        MPI_Wait(&req, ..);
    }
}

MPI_Finalize();
return 0;
}
```

***Using nonblocking operations but with MPI\_Wait inside the task region does not solve the problem***

# OpenMP tasks: MPI blocking Calls (4/5)

```
int main(int argc, char ** argv)
{
    MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, &provided);

#pragma omp parallel
{
    #pragma omp taskloop
    for (i = 0; i < 100; i++) {
        MPI_Request req; int done = 0;
        if (i % 2 == 0)
            MPI_Isend(.., to_myself, .., &req);
        else
            MPI_Irecv(.., from_myself, .., &req);
        While (!done) {
            #pragma omp taskyield
            MPI_Test(&req, &done, ..);
        }
    }
}

MPI_Finalize();
return 0;
}
```

***Still incorrect; taskyield does not guarantee a task switch***

# OpenMP tasks: MPI blocking Calls (5/5)

```
int main(int argc, char ** argv)
{
    MPI_Init_thread(NULL, NULL, MPI_THREAD_MULTIPLE, &provided);
    MPI_Request req[100];

#pragma omp parallel
{
    #pragma omp taskloop
    for (i = 0; i < 100; i++) {
        if (i % 2 == 0)
            MPI_Isend(.., to_myself, .., &req[i]);
        else
            MPI_Irecv(.., from_myself, .., &req[i]);
    }
}

MPI_Waitall(100, req, ..);
MPI_Finalize();
return 0;
}
```

***Correct example. Each task is nonblocking.***

# Ordering in MPI\_THREAD\_MULTIPLE: Incorrect Example with RMA

```
int main(int argc, char ** argv)
{
    /* Initialize MPI and RMA window */

    #pragma omp parallel for
    for (i = 0; i < 100; i++) {
        target = rand();
        MPI_Win_lock(MPI_LOCK_EXCLUSIVE, target, 0, win);
        MPI_Put(..., win);
        MPI_Win_unlock(target, win);
    }

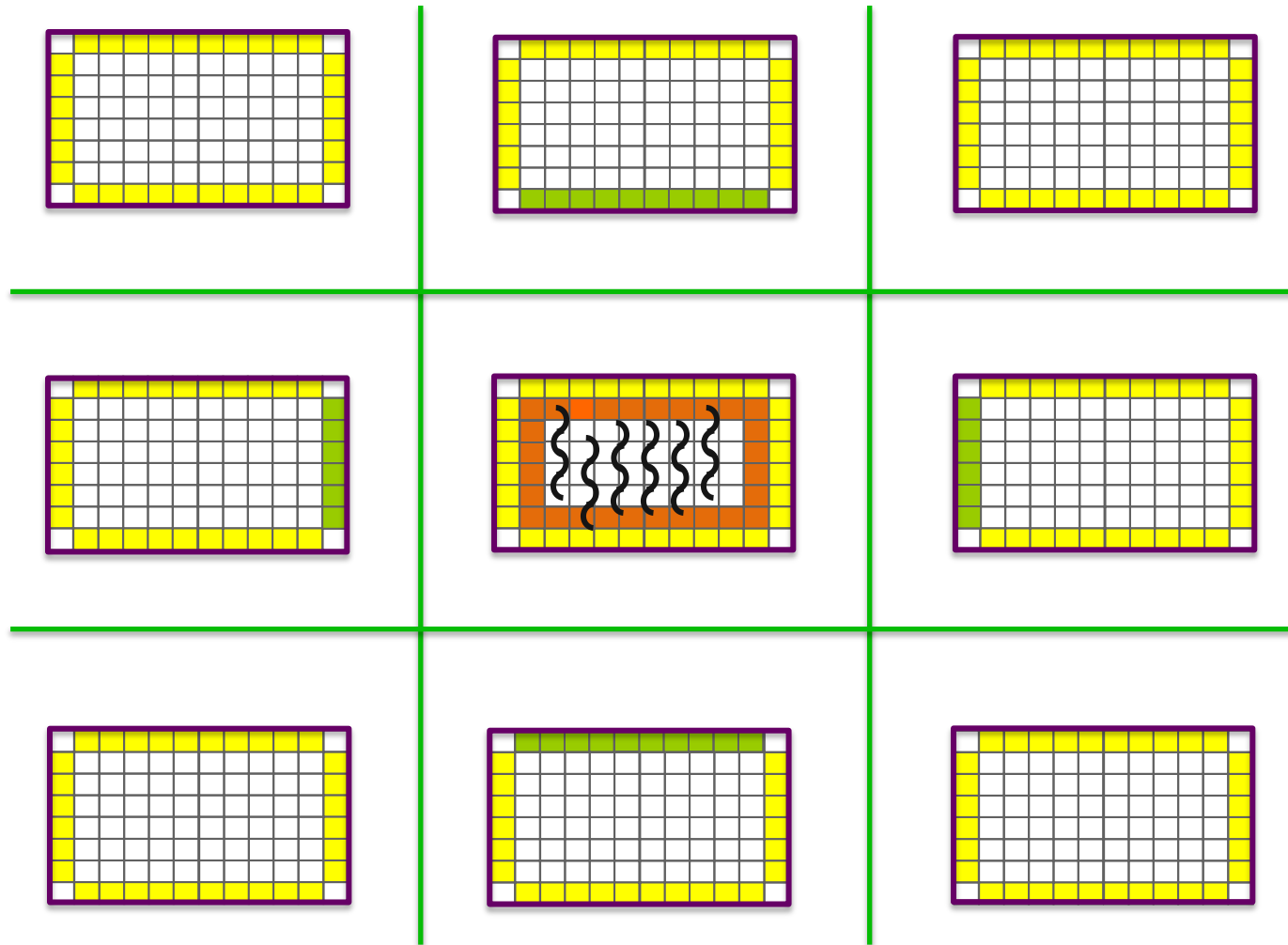
    /* Free MPI and RMA window */

    return 0;
}
```

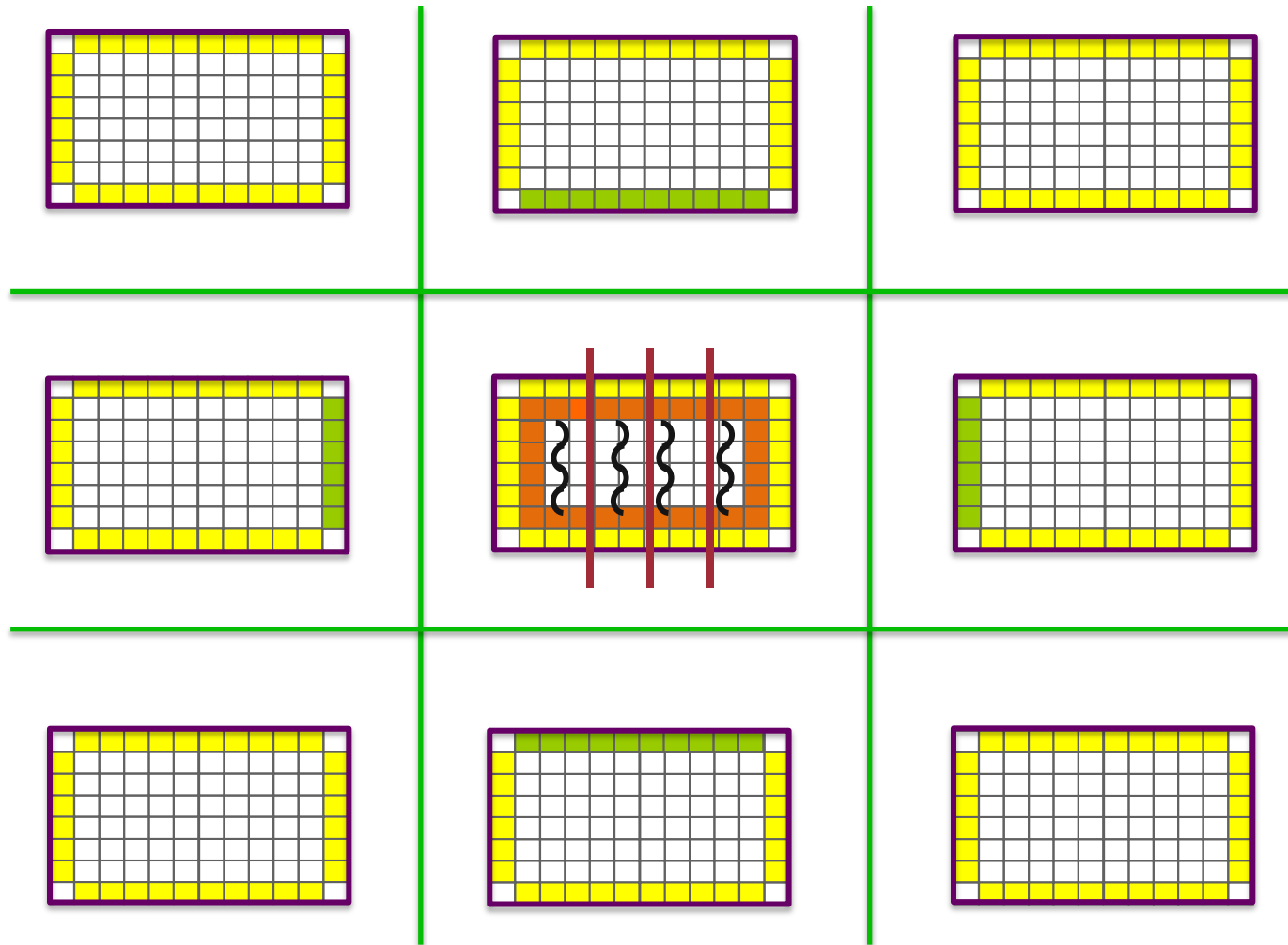
***Different threads can lock the same process causing multiple locks to the same target before the first lock is unlocked***



# Implementing Stencil Computation using MPI\_THREAD\_FUNNELED



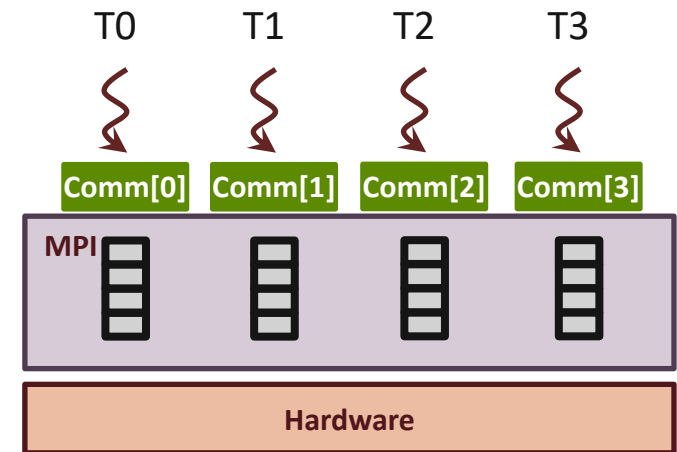
# Implementing Stencil Computation using MPI\_THREAD\_MULTIPLE



# Recommendation: Maximize independence between threads with communicators

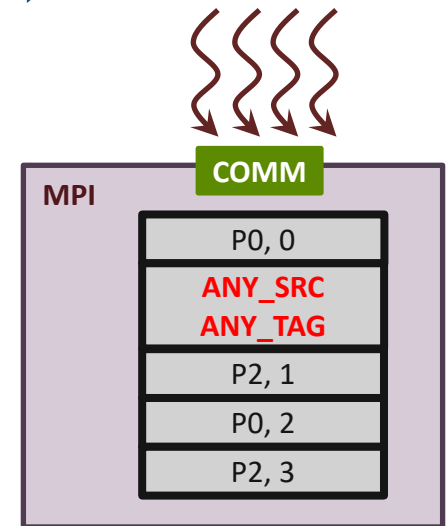
- Each thread accesses a **different communicator**
  - Each communicator may be associated with isolated resource in an MPI implementation

```
MPI_Comm *comms;  
int nthreads = omp_get_num_threads();  
comms = malloc(sizeof(MPI_Comm) * nthreads);  
  
for (i = 0; i < nthreads; i++)  
    MPI_Comm_dup(MPI_COMM_WORLD, &comms[i]);  
  
#pragma omp parallel  
{  
    int tid = omp_get_thread_num();  
    #pragma omp taskloop  
    for (i = 0; i < 100; i++)  
        MPI_Isend(..., comm[tid], &req[i]);  
}  
MPI_Waitall(100, req, ..);
```



# Recommendation: Maximize independence between threads with ranks or tags (1/2)

- Threads have to match all receive messages in sequential (e.g., a single receive-queue) if a **wildcard receive** may be posted
  - Ensure ordering of message matching
- **Let MPI know if you do not use wildcard receive**
  - Info hints **no\_any\_source**, **no\_any\_tag** (accepted for inclusion in MPI-4)
  - MPI can get rid of the single receive-queue for the communicator

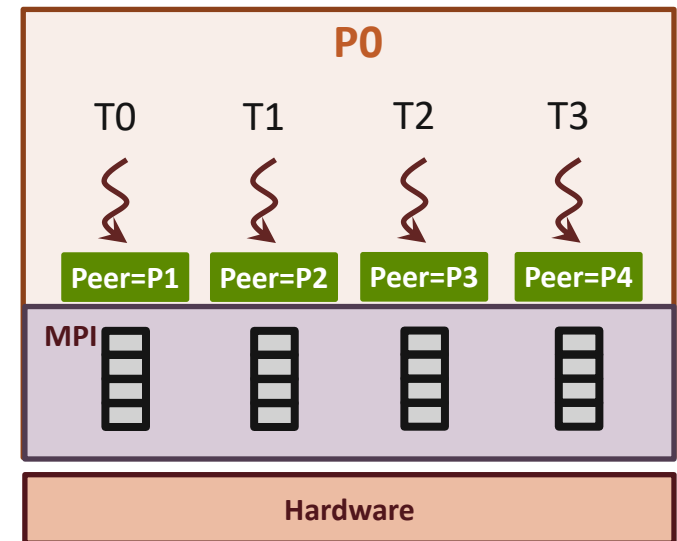


```
MPI_Info info;
info = MPI_Info_create();
MPI_Info_set(info, "no_any_source",
              "true");
MPI_Comm_set_info(comm, info);
MPI_Info_free(&info);
/* Communicate without
   MPI_ANY_SOURCE */
```

## Recommendation: Maximize independence between threads with ranks or tags (2/2)

- Each thread communicates with **different peer\_rank or tag**
  - MPI may assign isolated resource for different set of [peer\_rank + tag]

```
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    #pragma omp taskloop
    for (i = 0; i < 100; i++)
        MPI_Isend(..., peer_ranks[tid], tid,
                  comm, &req[i]);
}
MPI_Waitall(100, req, ..);
```

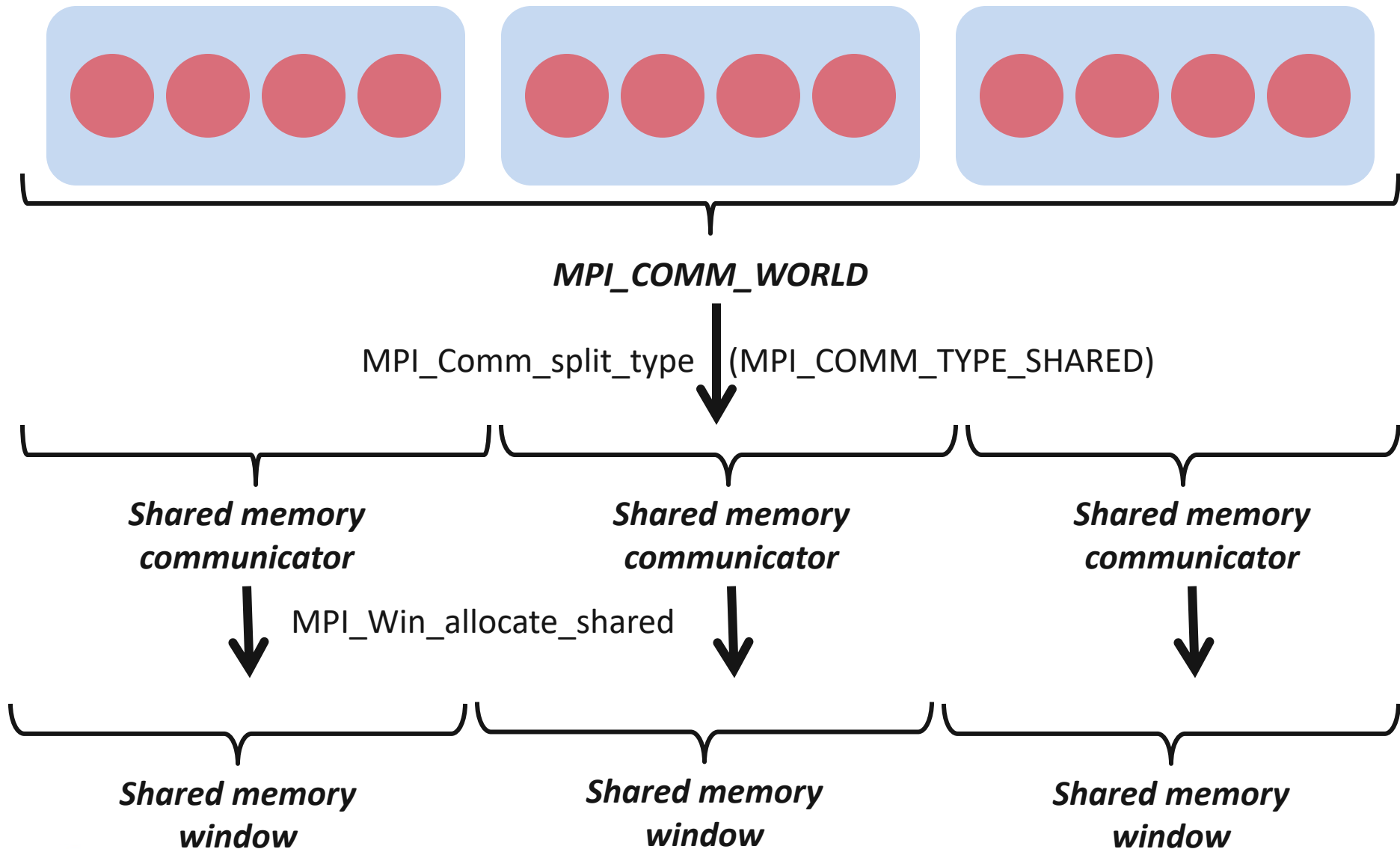


# MPI + Shared-Memory

# Hybrid Programming with Shared Memory

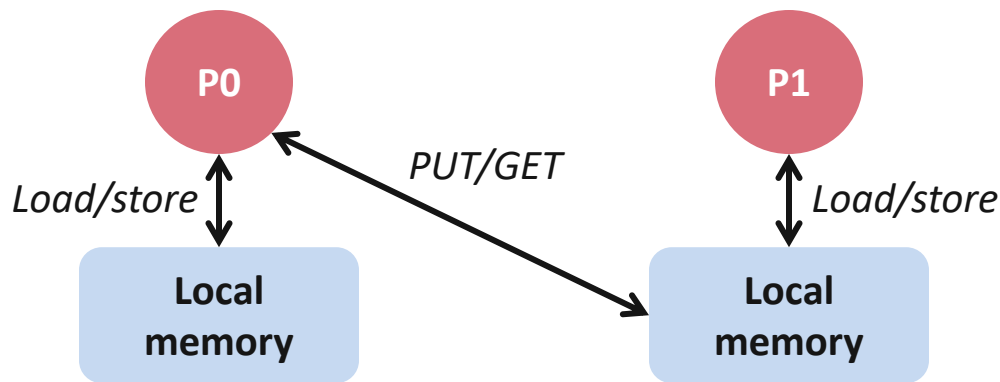
- MPI-3 allows different processes to allocate shared memory through MPI
  - `MPI_Win_allocate_shared`
- Uses many of the concepts of one-sided communication
- Applications can do hybrid programming using MPI or load/store accesses on the shared memory window
- Other MPI functions can be used to synchronize access to shared memory regions
- Can be simpler to program than threads
  - Because memory locality is clear (needed for performance) and data sharing is explicit

# Creating Shared Memory Regions in MPI

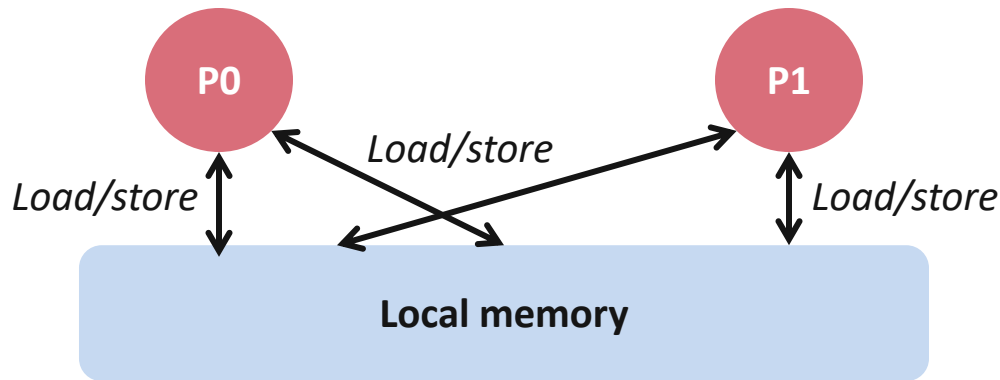




# Regular RMA windows vs. Shared memory windows



***Traditional RMA windows***



***Shared memory windows***

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

# MPI\_COMM\_SPLIT\_TYPE

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

- Create a communicator where processes “share a property”
  - Properties are defined by the “split\_type”
  - In MPI 3.1, only split\_type is MPI\_COMM\_TYPE\_SHARED
- Arguments:
  - comm - input communicator (handle)
  - Split\_type - property of the partitioning (integer)
  - Key - Rank assignment ordering (nonnegative integer)
  - info - info argument (handle)
  - newcomm- output communicator (handle)

# MPI\_WIN\_ALLOCATE\_SHARED

```
MPI_Win_allocate_shared(MPI_Aint size, int disp_unit,  
                        MPI_Info info, MPI_Comm comm, void *baseptr,  
                        MPI_Win *win)
```

- Create a remotely accessible memory region in an RMA window
  - Data exposed in a window can be accessed with RMA ops or load/store
- Arguments:
  - size - size of local data in bytes (nonnegative integer)
  - disp\_unit - local unit size for displacements, in bytes (positive integer)
  - info - info argument (handle)
  - comm - communicator (handle)
  - baseptr - pointer to exposed local data
  - win - window (handle)

# Shared Arrays with Shared memory windows

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

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

    MPI_Win_lockall(win);

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

    /* use shared memory */

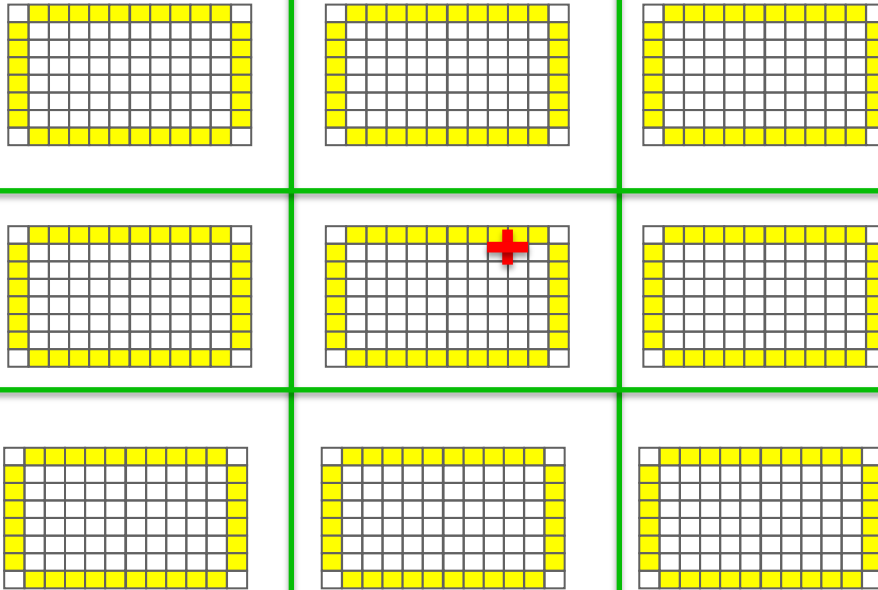
    MPI_Win_unlock_all(win);

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

# Memory allocation and placement

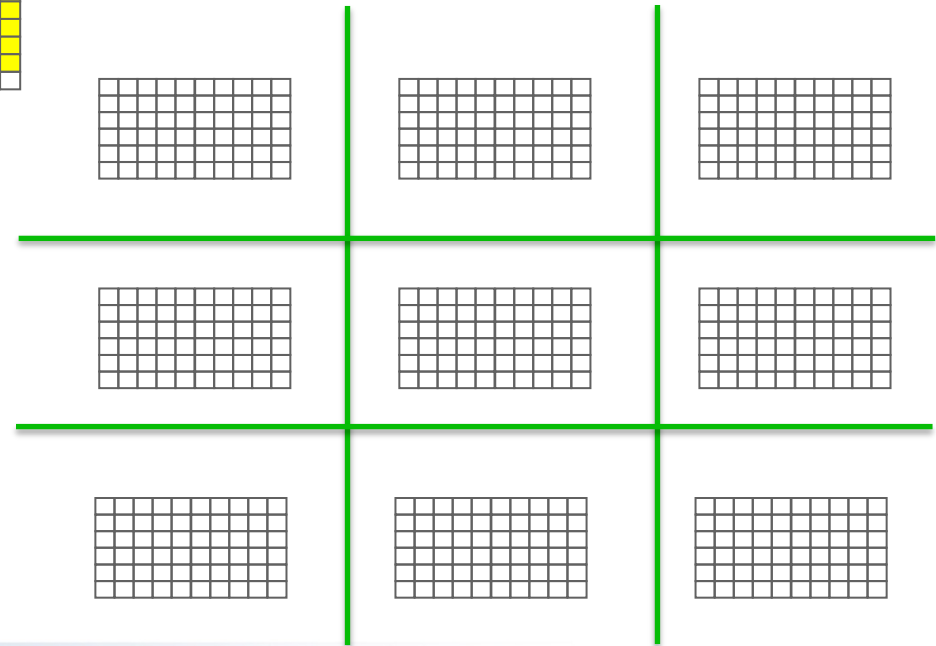
- Shared memory allocation does not need to be uniform across processes
  - Processes can allocate a different amount of memory (even zero)
- The MPI standard does not specify where the memory would be placed (e.g., which physical memory it will be pinned to)
  - Implementations can choose their own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

# Example Computation: Stencil



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

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



# What should you use: Threads or Process Shared Memory

- It depends on the application, target machine, and MPI implementation
- When should I use process shared memory?
  - The only resource that needs sharing is memory
  - Few allocated objects need sharing (easy to place them in a public shared region)
- When should I use threads?
  - More than memory resources need sharing (e.g., TLB)
  - Many application objects require sharing
  - Application computation structure can be easily parallelized with high-level OpenMP loops

# Shortcomings: Restricted Allocation Methods

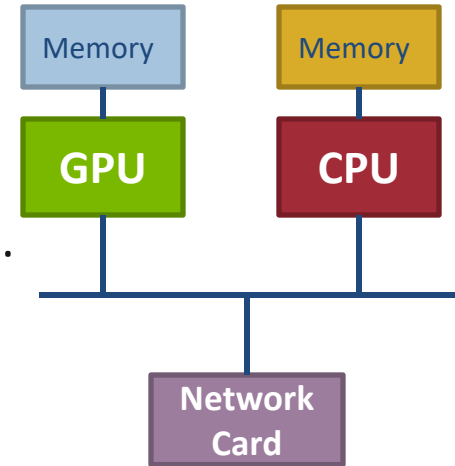
- In MPI-3 shared memory, memory allocation is restrictive
  - Allocation has to be done using the MPI call
  - Cannot use the plethora of other memory allocation libraries out there, e.g., cannot allocate aligned memory (important for vectorization)
- With threads, most of those other memory allocation techniques are directly usable



# MPI + Accelerators

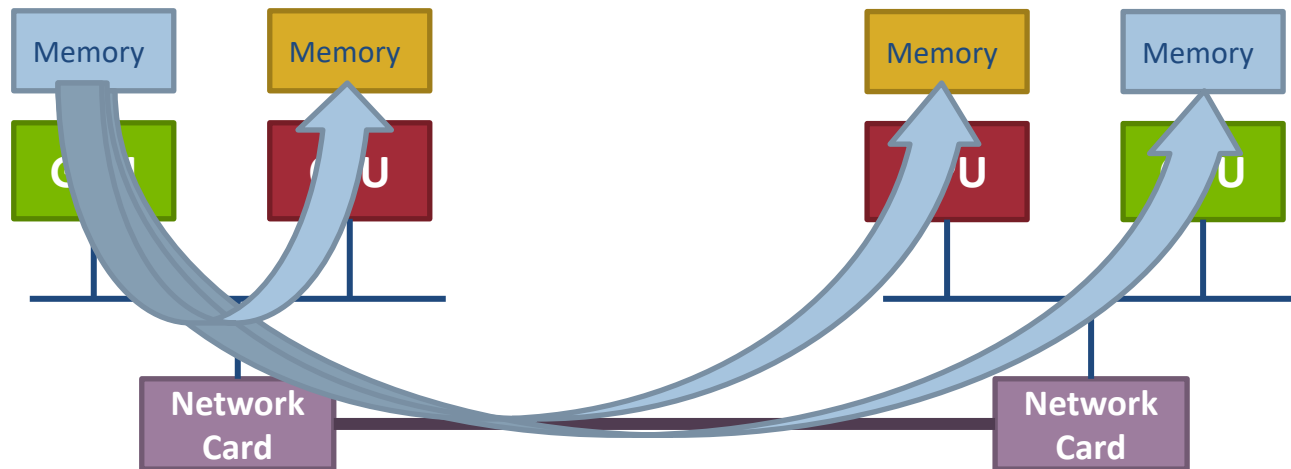
# Accelerators in Parallel Computing

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



# MPI + Accelerator Programming Examples

How to move data between GPUs with MPI?



**Real answer:** It depends on what GPU library, what hardware and what MPI implementation you are using

**Simple answer:** For modern GPUs, “just like you would with a non-GPU machine”

# CUDA Awareness in MPI

- The MPI standard does not explicitly require GPU support
  - Each MPI implementation can choose whether or not it wants to support GPUs
- Current status: Many, but not all, MPI implementations support CUDA
  - Already supported by MVAPICH, Open MPI, Spectrum MPI
- You can use GPUs even with MPI implementations that do not support CUDA, but data movement will need to be explicit
  - MPI does not understand data residing on GPUs
- With CUDA-aware MPI implementations, some things are automatically handled by the MPI library

# Non-CUDA-aware MPI implementations: Programmability Limitations (1/2)

CUDA

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

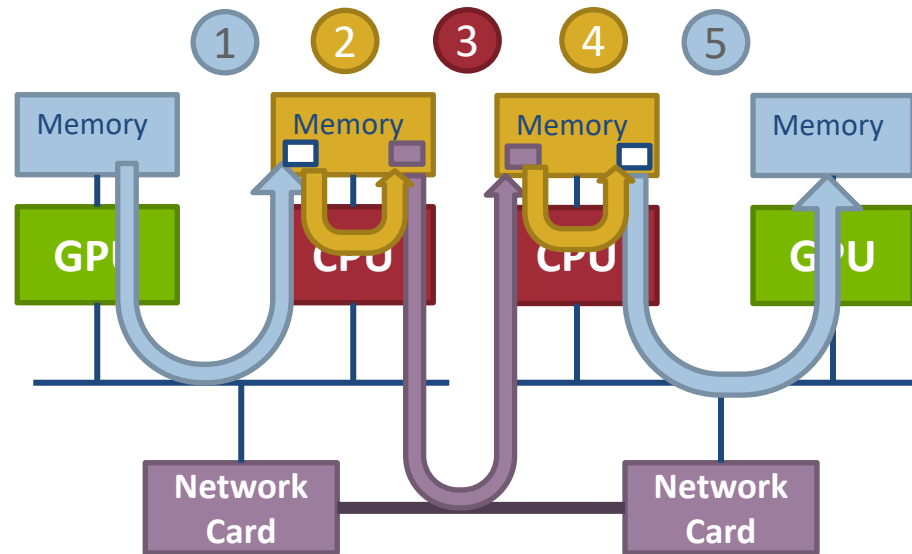
if(my_rank == sender) {
    computation_on_GPU(dev_buf);
    cudaMemcpy(host_buf, dev_buf, size, ...);
    MPI_Isend(host_buf, size, ...);
} else {
    MPI_Recv(host_buf, size, ...);
    cudaMemcpy(dev_buf, host_buf, size, ...);
    computation_on_GPU(dev_buf);
}
```

OpenACC

```
double *buf;
buf = (double*)malloc(size * sizeof(double));
#pragma acc enter data create(buf[0:size])

if(my_rank == sender) {
    computation_on_GPU(buf);
    #pragma acc update host (buf[0:size])
    MPI_Isend(buf, size, ...);
} else {
    MPI_Recv(buf, size, ...);
    #pragma acc update device (buf[0:size])
    computation_on_GPU(buf);
}
```

# Non-CUDA-aware MPI implementations: Programmability Limitations (2/2)



```
computation_on_GPU(dev_buf);  
cudaMemcpy(host_buf, dev_buf, size, ...);  
MPI_Isend(host_buf, size, ...);  
  
MPI_Recv(host_buf, size, ...);  
cudaMemcpy(dev_buf, host_buf, size, ...);  
computation_on_GPU(dev_buf);
```

CUDA

```
computation_on_GPU(buf);  
#pragma acc update host (buf[0:size])  
MPI_Isend(buf, size, ...);  
  
MPI_Recv(buf, size, ...);  
#pragma acc update device (buf[0:size])  
computation_on_GPU(buf);
```

OpenACC

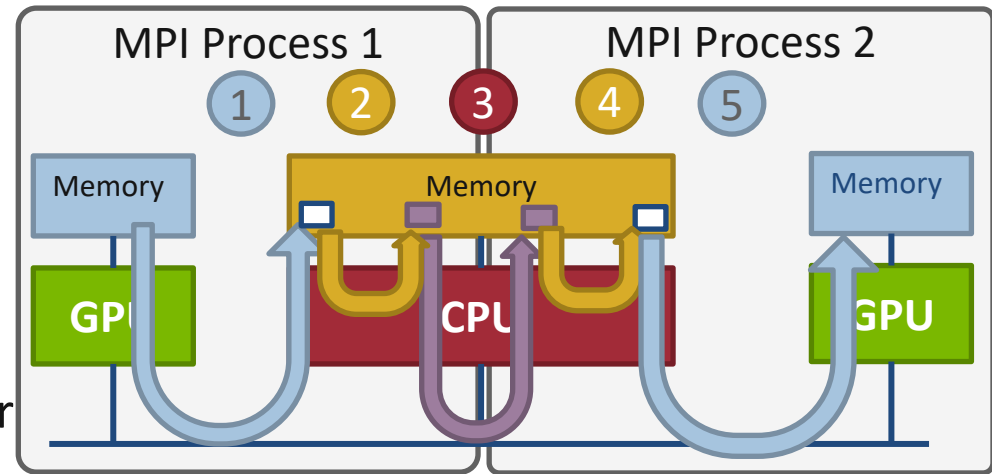
MPI assumes host memory

The user ensures that host memory is synchronized

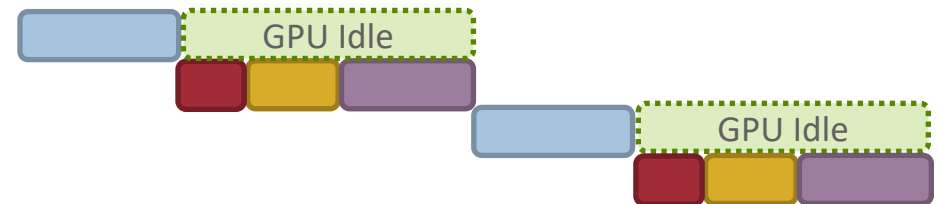
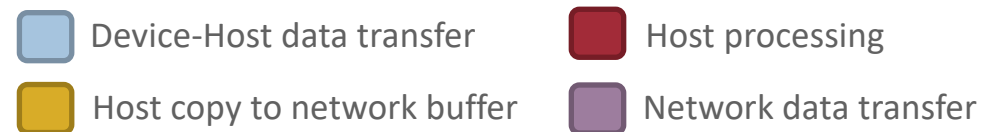
**Using cudaMemcpyAsync before MPI\_Isend would be incorrect**

# Non-CUDA-aware MPI implementations: Performance Limitations

- Inefficient intranode GPU-GPU data transfer between MPI processes
  - Several DMA and memory copies on the critical path
- Inefficient bulk-synchronous transfer model
  - The CPU cannot trigger the MPI data transfer until the GPU completed the device-host data transfer
- Inefficient GPU resource utilization
  - The GPU could potentially be idle while the host handles MPI communication



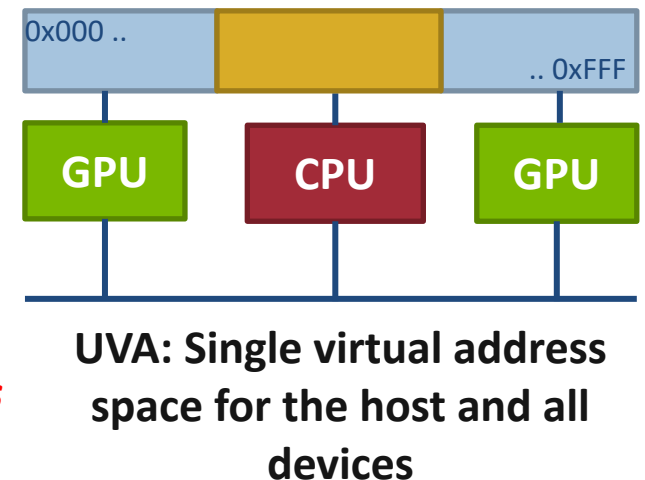
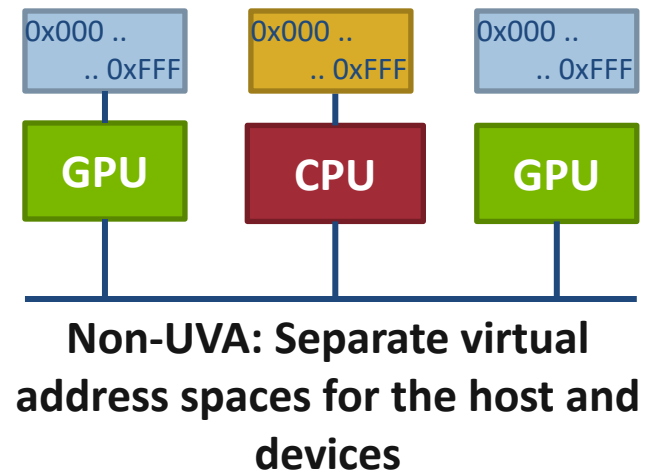
**Inefficient intra-node GPU data transfer**



**Inefficient bulk-synchronous and GPU-wasteful data transfer model**

# CUDA-aware MPI implementation requirements

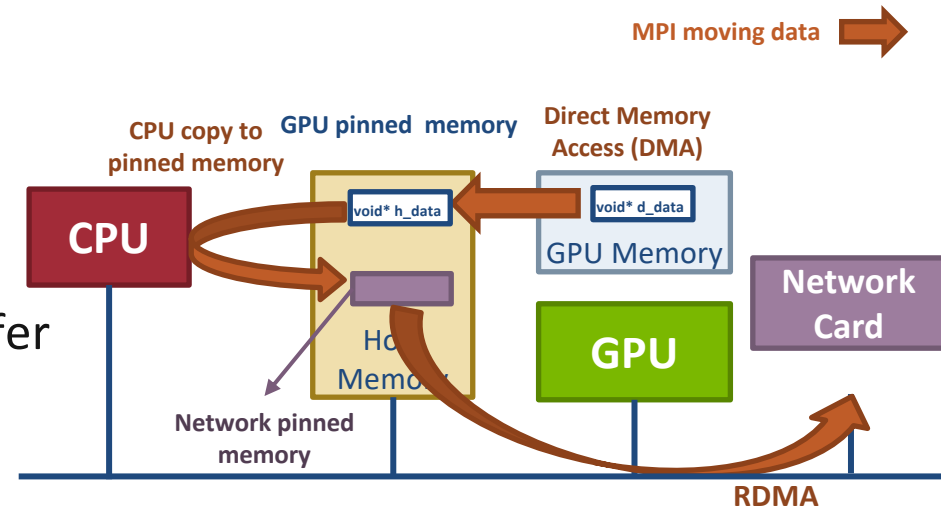
- CUDA-awareness in MPI requires the Unified Virtual Address (UVA) feature of GPUs, at the very least
  - Introduced in CUDA-4.0
  - Host memory and all GPUs share the same virtual address space
  - The user can query the location of the data allocation given a pointer in the unified address space with **cuPointerGetAttribute()**
- GPU Direct 1.0, GPU Direct 2.0 and GPU Direct RDMA are not required for correctness, but improve performance
  - Needs to be supported by the GPU and the network
  - ***This is the state-of-the-art for modern NVIDIA GPUs and Mellanox InfiniBand, but might not be supported by other GPUs or other networks***





# CUDA-aware MPI implementations: Programmability

- User can pass device pointer to MPI
- MPI implementation can query for the owner (host or device) of the data
- If the data is on the device, **the MPI implementation can handle** data transfer from GPU to the network



Example of MPI moving data from the GPU device to the network

CUDA

```
computation_on_GPU(dev_buf);  
MPI_Isend(dev_buf, size, ...);  
  
MPI_Recv(dev_buf, size, ...);  
computation_on_GPU(dev_buf);
```

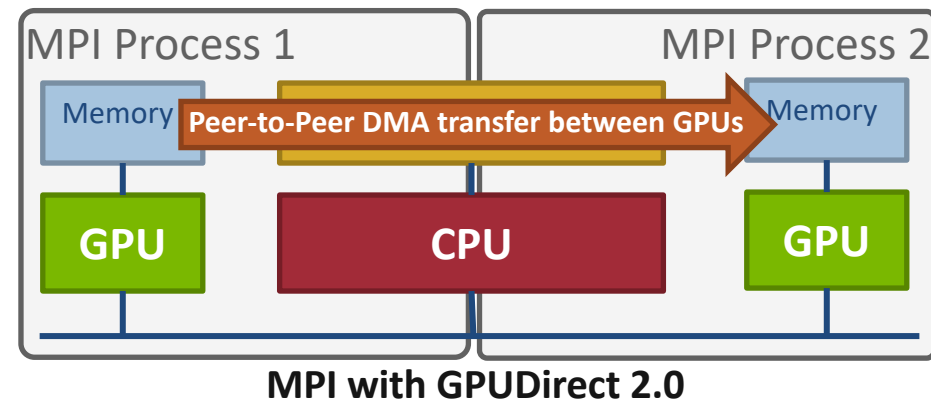
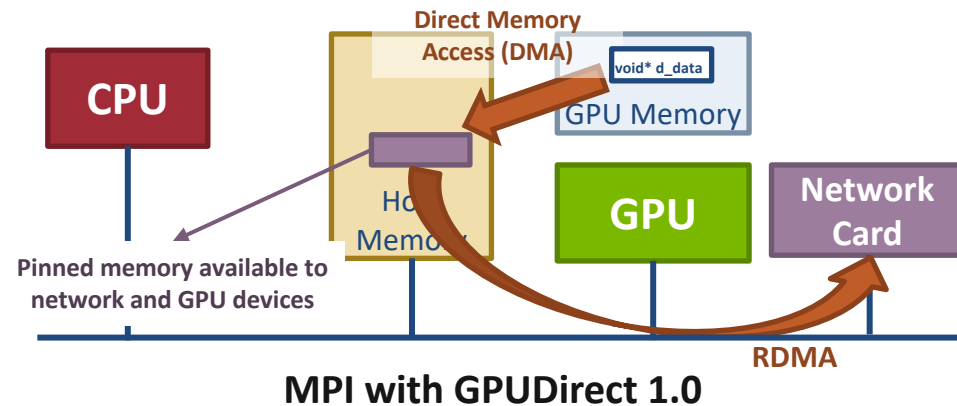
OpenACC

```
computation_on_GPU(buf);  
#pragma acc host_data use_device (buf)  
MPI_Isend(buf, size, ...);  
  
#pragma acc host_data use_device (buf)  
MPI_Recv(buf, size, ...);  
computation_on_GPU(buf);
```

MPI can transparently figure out the physical location of the data

# CUDA-aware MPI implementations: Performance (2/3)

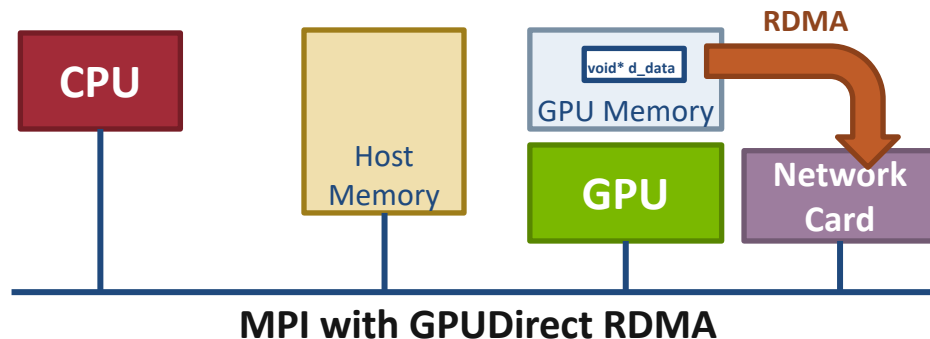
- GPUDirect 1.0 (Q2' 2010)
  - Avoid unnecessary system memory copies copying data directly to/from pinned CUDA host memory
  - RDMA can use directly the CUDA pinned memory
  - Required kernel driver updates
- GPUDirect 2.0 (Peer-to-Peer, 2011)
  - GPU peer-to-peer data transfers are possible
  - MPI can directly move data between GPU devices



# CUDA-aware MPI implementations: Performance (3/3)

## ■ GPUDirect RDMA

- CUDA  $\geq 5$ , 2013
- Technology introduced in Kepler-class GPUs and CUDA-5
- GPU memory is directly accessible to third-party devices, including network interfaces
- RDMA operations to/from the device memory are possible and completely bypass the host memory



## Section Summary

- Programming with accelerators is becoming increasingly important
- MPI is playing its role in enabling the usage of accelerators across distributed memory nodes
- The situation with MPI + GPU support is improving in both MPI implementations and in GPU hardware/software capabilities

# Process Topologies and Neighborhood Collectives

# Topology Mapping Basics

- First type: Allocation mapping (when job is submitted)
  - 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 maximization”
  - Also problematic to specify communication pattern upfront, not always possible (or static)

# Topology Mapping Basics contd.

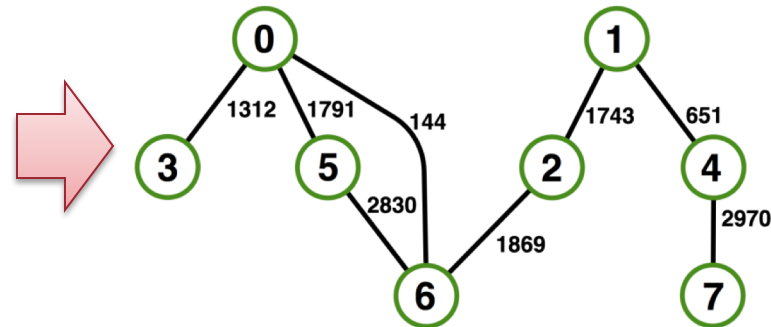
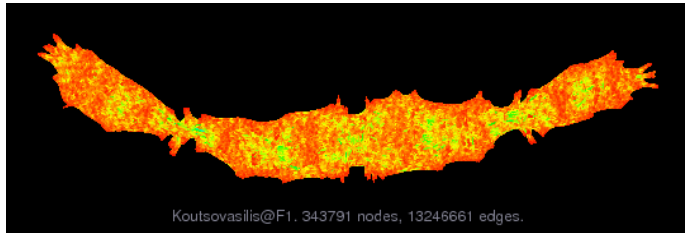
- Rank reordering

- Change numbering in a given allocation to reduce congestion or dilation
- Sometimes automatic (early IBM SP machines)

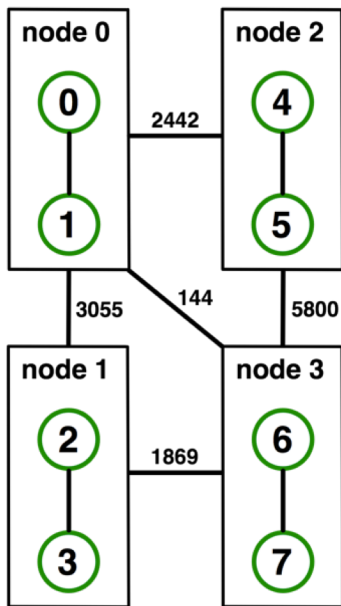
- Properties

- 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

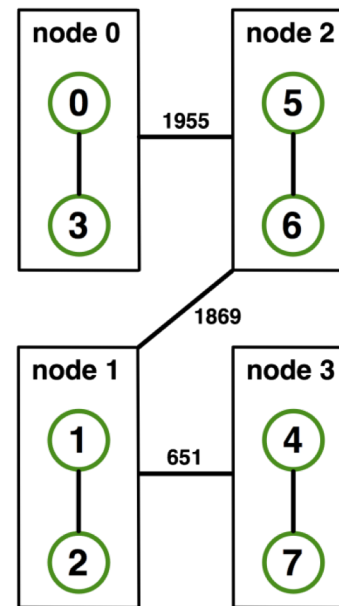
# On-Node Reordering



Naïve Mapping



Optimized Mapping

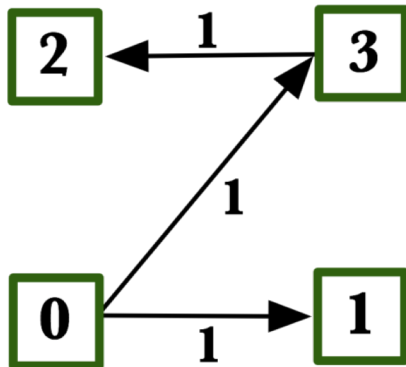


Topomap

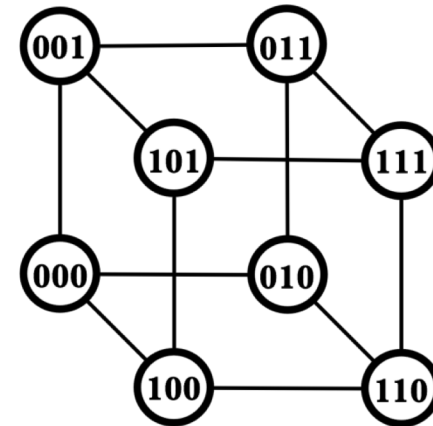


# Off-Node (Network) Reordering

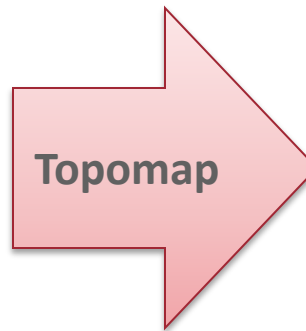
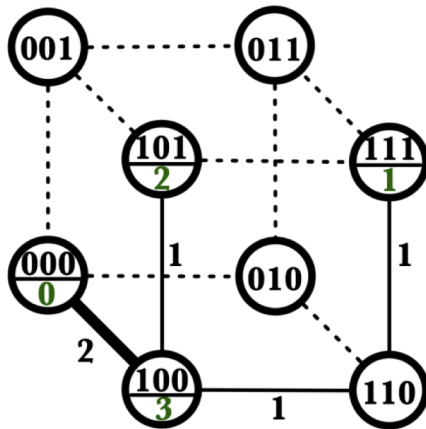
Application Topology



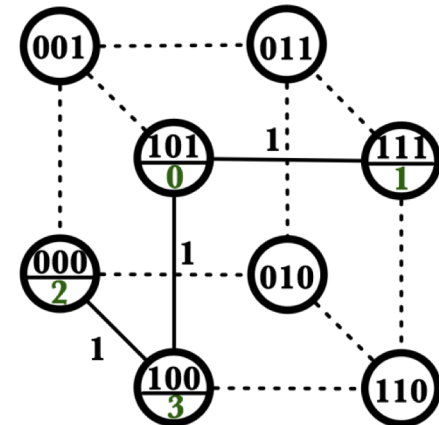
Network Topology



Naïve Mapping



Optimal Mapping



# 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 Topology Realities

## ■ Cartesian Topologies

- MPI\_Dims\_create is required to provide a “square” decomposition
  - May not match underlying physical network
  - Even if it did, hard to define unless physical network is mesh or torus
- MPI\_Cart\_create is supposed to provide a “good” remapping (if requested)
  - But implementations are poor and may just return the original mapping

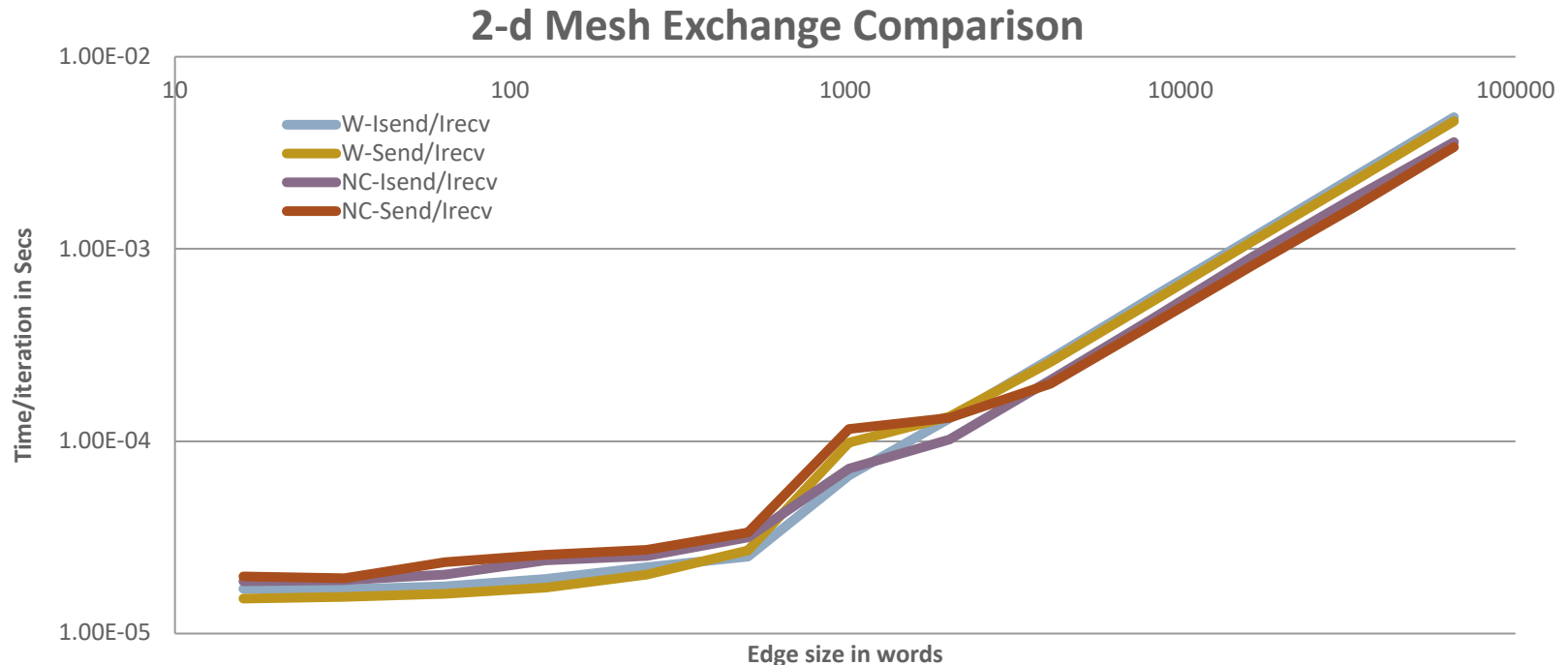
## ■ Graph Topologies

- The general process mapping problem is very hard
- Most (all?) MPI implementations are poor
- Some research work has developed tools to create better mappings
  - You can use them with MPI\_Comm\_dup to create a “well ordered” communicator

## ■ Neighbor collectives

- MPI 3 introduced these; permit collective communication with just the neighbors as defined by the MPI process topology
- Offers opportunities for the MPI implementation to optimize; not realized yet

# Hotspot results for Theta



- Communicator from MPI\_Cart\_create has same order as MPI\_COMM\_WORLD
- For 2-d mesh exchange with 512 processes and 64 processes/node, MPI\_Cart\_create has an average of 28 target off-node processes.
- The “node-cart” communicator has an average of 20 target off-node processes
  - Communication time is 26% faster with the hand-optimized communicator

# 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 in dims

## MPI\_Dims\_create Example

```
int p;  
int dims[3] = {0,0,0};  
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

# 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 of dims must be  $\leq P$
- 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!



# 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

# Some Stencil Results

Type	COMM_WORLD	Cart Comm	NodeCart Comm
Blocking pt2pt	2.97	2.93	3.20
Nonblocking pt2pt	3.14	3.17	3.42
NB pt2pt w datatype	3.00	3.00	3.16
NB pt2pt w DT + Overlap	2.93	2.99	3.24
RMA	0.174	0.172	0.177
Collective	--	2.96	--
Nonblocking w Sharedmem	1.83	2.20	1.98

4096 MPI processes on 64 x 64 processor mesh; 16 processes/node

4k x 4k global grid. On Blue Waters, a Cray XE6/XK7

Rates in TF/sec. Used stencil test in mpi-patterns from baseenv. RMA with datatypes too slow and removed from test

# Dynamic Workloads Require New, More Integrated Approaches

- Performance irregularities mean that classic approaches to decomposition are increasingly ineffective
  - Irregularities come from OS, runtime, process/thread placement, memory, heterogeneous nodes, power/clock frequency management
- Static partitioning tools can lead to persistent load imbalances
  - Mesh partitioners have incorrect cost models, no feedback mechanism
  - “Regrid when things get bad” won’t work if the cost model is incorrect; also costly
- Basic building blocks must be more dynamic without introducing too much overhead

## Section Summary

- MPI does not expose information about the network topology (would be very complex)
- Topology functions allow users to specify application communication patterns/topology
  - Convenience functions (e.g., Cartesian)
  - Storing neighborhood relations (Graph)
- Neighborhood collectives allow user virtual topologies to be exploited in collective communication

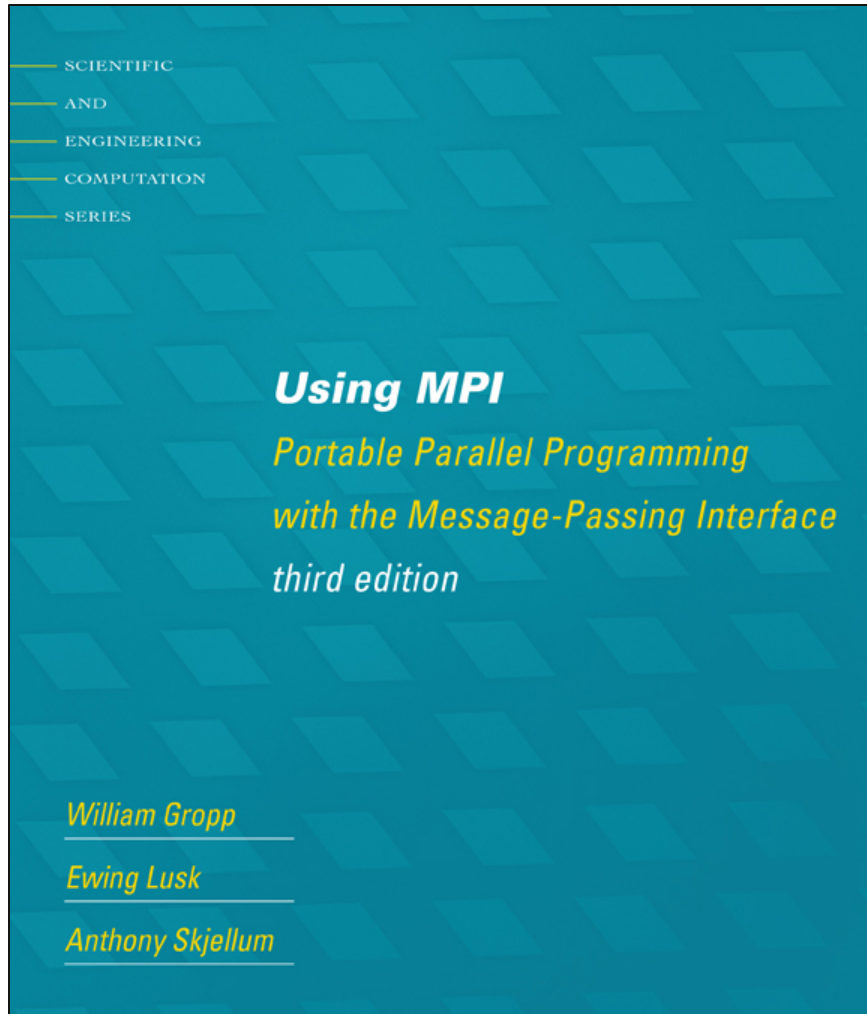
# Concluding Remarks

- 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 scientific applications with great success

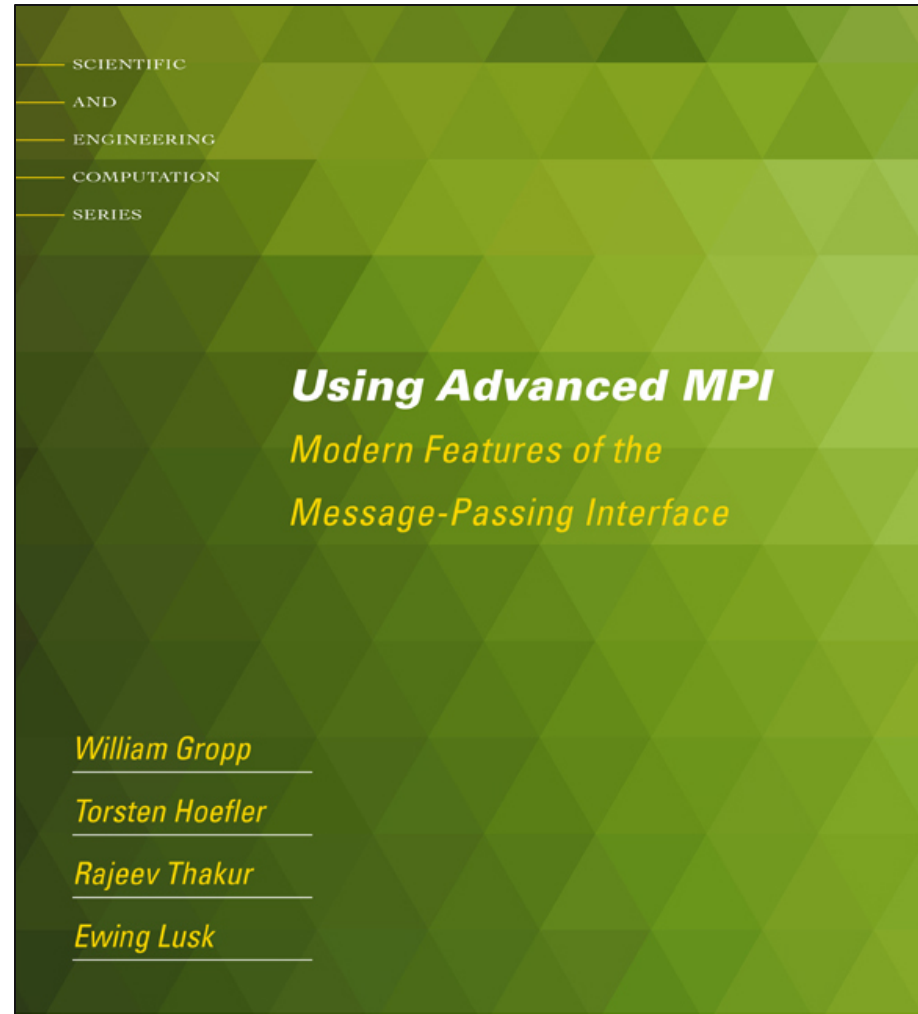
# Web Pointers

- MPI standard : <http://www.mpi-forum.org/docs/docs.html>
- MPI Forum : <http://www.mpi-forum.org/>
- MPI implementations:
  - MPICH : <http://www.mpich.org>
  - MVAPICH : <http://mvapich.cse.ohio-state.edu/>
  - Intel MPI: <http://software.intel.com/en-us/intel-mpi-library/>
  - Microsoft MPI: [www.microsoft.com/en-us/download/details.aspx?id=39961](http://www.microsoft.com/en-us/download/details.aspx?id=39961)
  - Open MPI : <http://www.open-mpi.org/>
  - IBM MPI, Cray MPI, HP MPI, TH MPI, NEC MPI, Fujitsu MPI, ...
- Several MPI tutorials can be found on the web

# Tutorial Books on MPI



**Basic MPI**



**Advanced MPI, including MPI-3**