MPI for Scalable Computing

Tutorial at ATPESC, August 2023

Latest slides and code examples are available at

https://anl.box.com/v/atpesc2023-mpi-tutorial

Yanfei Guo, Ken Raffenetti, Hui Zhou

Argonne National Laboratory

About the Speakers

- **Ken Raffenetti:** Principal Research Software Engineer, Argonne National Laboratory
- **Yanfei Guo:** Computer Scientist, Argonne National Laboratory
- **Hui Zhou:** Principal Research Software Engineer, Argonne National Laboratory

- All of us are deeply involved in MPI standardization (in the MPI Forum) and in MPI implementation

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 and this tutorial
  - Avoiding unnecessary synchronization
  - Topics in collective communication
  - Stencil example and hands-on exercises
  - Minimizing data motion using MPI derived datatypes
  - Begin one-sided communication (or remote memory access)

- Afternoon
  - One-sided communication contd.
  - Hands-on exercises
  - Hybrid programming
    - MPI + threads
    - MPI + GPUs
  - Hands-on exercises

What is MPI?

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

MPI-1

- MPI-1 supports the classical message-passing programming model: basic point-to-point communication, collectives, datatypes, etc
- MPI-1 was defined (1994) by a broadly based group of parallel computer vendors, computer scientists, and applications developers.
    - 2-year intensive process
- Implementations appeared quickly and now MPI is taken for granted as vendor-supported software on any parallel machine.
- Free, portable implementations exist for clusters and other environments (MPICH, Open MPI)

Following MPI Standards

- MPI-2 was released in 1997
  - Several additional features including MPI + threads, MPI-I/O, remote memory access functionality and many others
- MPI-2.1 (2008) and MPI-2.2 (2009) were released with some corrections to the standard and small features
- MPI-3 (2012) added several new features to MPI
- MPI-3.1 (2015) introduced minor corrections and features
- MPI-4 (June 2021) is the latest version of the standard
- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- Other information on Web:
  - at http://www.mcs.anl.gov/mpi
  - pointers to lots of material including tutorials, a FAQ, other MPI pages

Overview of New Features in MPI-3

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

What’s new in MPI-4

- MPI-4 is official (June 2021)
  - https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf
- Major new features and changes
  - Persistent Collectives
  - Partitioned Communication
  - Sessions
  - Big Count
  - Error Handling Improvement
  - Topology Improvement

Important considerations while using MPI

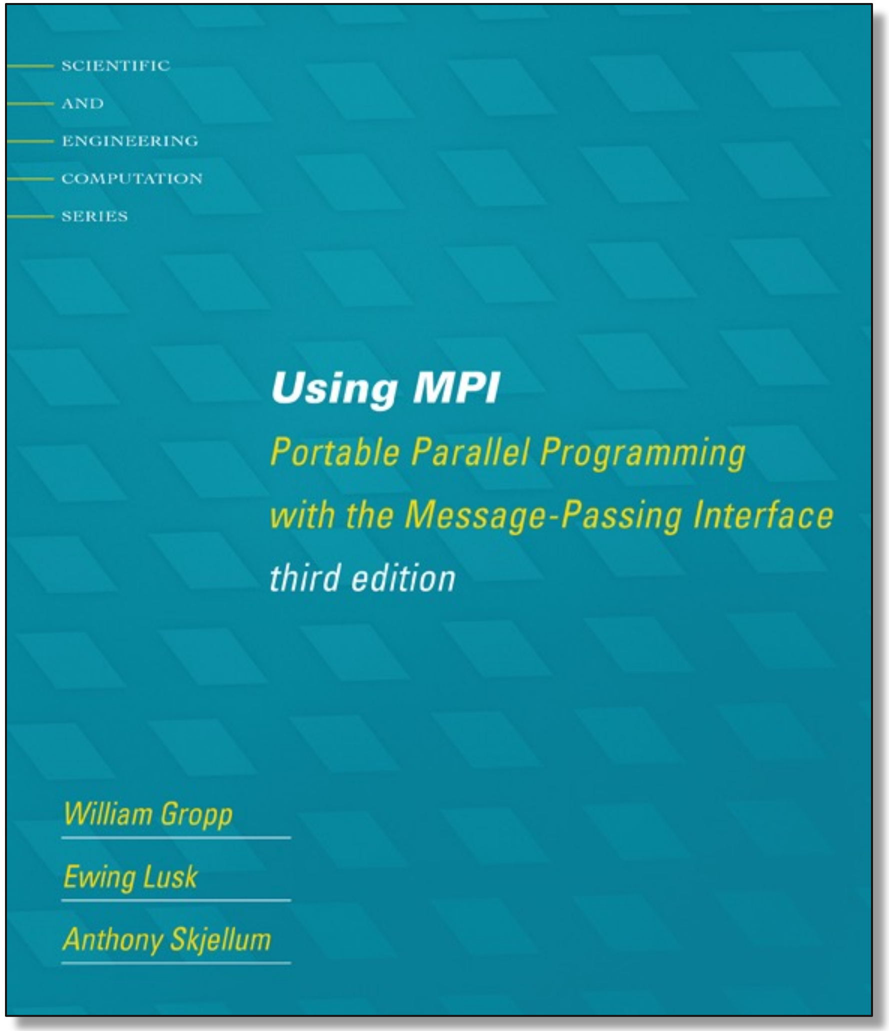
- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs

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://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi
  - 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



SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES

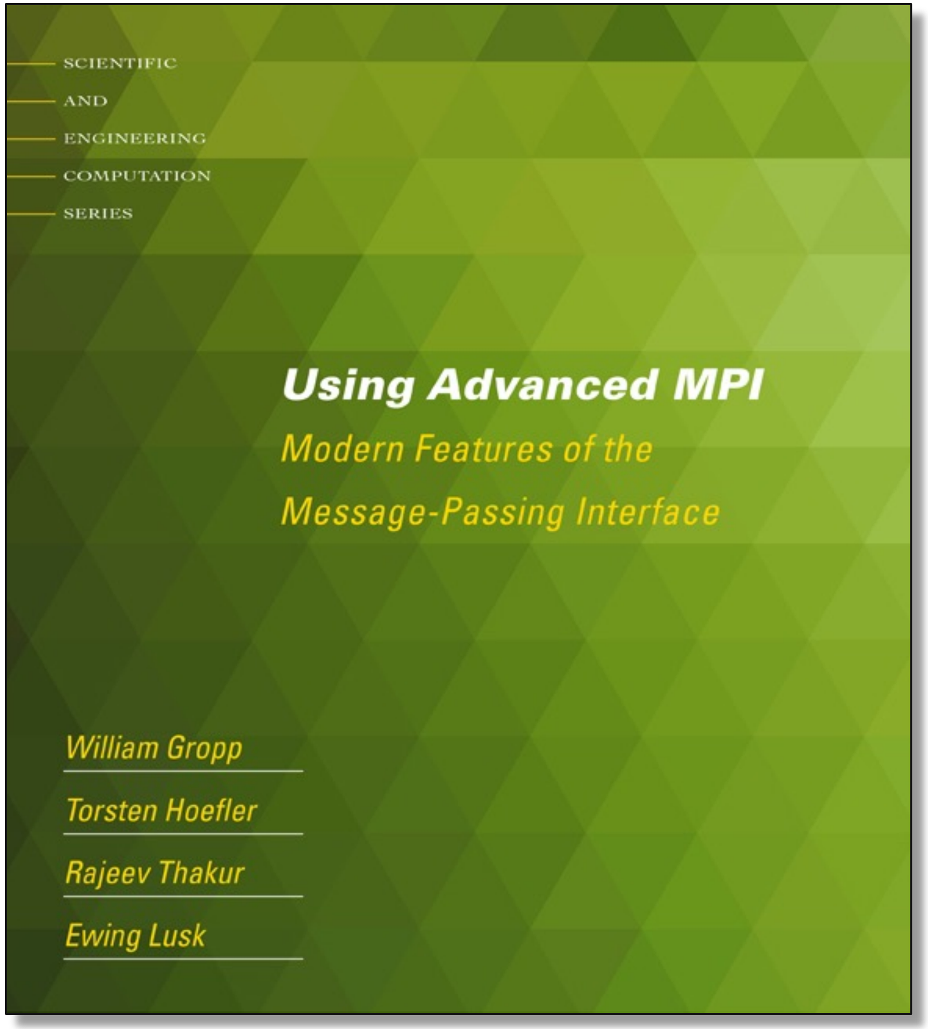
Using MPI

Portable Parallel Programming with the Message-Passing Interface

third edition

William Gropp
Ewing Lusk
Anthony Skjellum

Basic MPI



SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES

Using Advanced MPI

Modern Features of the Message-Passing Interface

William Gropp
Torsten Hoefler
Rajeev Thakur
Ewing Lusk

Advanced MPI, including MPI-3

Approach in this Tutorial

- Example driven
  - A few running examples used throughout the tutorial
  - Other smaller examples used to illustrate specific features

Warm Up Exercise (~5 minutes)

- Copy examples to your directory on Perlmutter
  - `cp ~train202/mpi-examples.zip .`
- Get an interactive allocation
  - `salloc -q shared -C gpu -A ntrain5 -n 16 --reservation=atpesc_aug2 -t 30:00`
- Build and run examples in *blocking_p2p* directory
  - *blocking_p2p/sendrecv.c*
    - Simple messaging between ranks 0 and 1 in MPI_COMM_WORLD
    - `make sendrecv; srun ./sendrecv`
  - *blocking_p2p/manager_worker_simple.c*
    - One process distributes “work” to others
    - `make manager_worker_simple; srun ./manager_worker_simple`
- Explore others on your own!

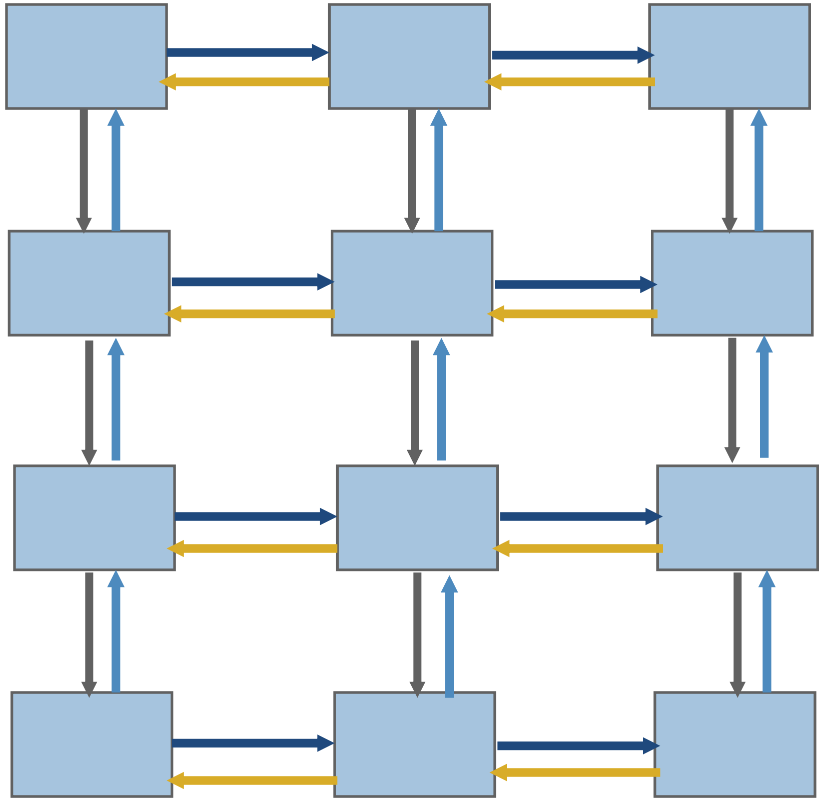
Costs of Unintended Synchronization

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

Time

Start	Start	Start	Start	Start	Start	Send	Recv
Send	Send	Send	Send	Send	Send		
					Send	Recv	
				Send	Recv		
			Send	Recv			
		Send	Recv				
	Send	Recv					
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)
  ```

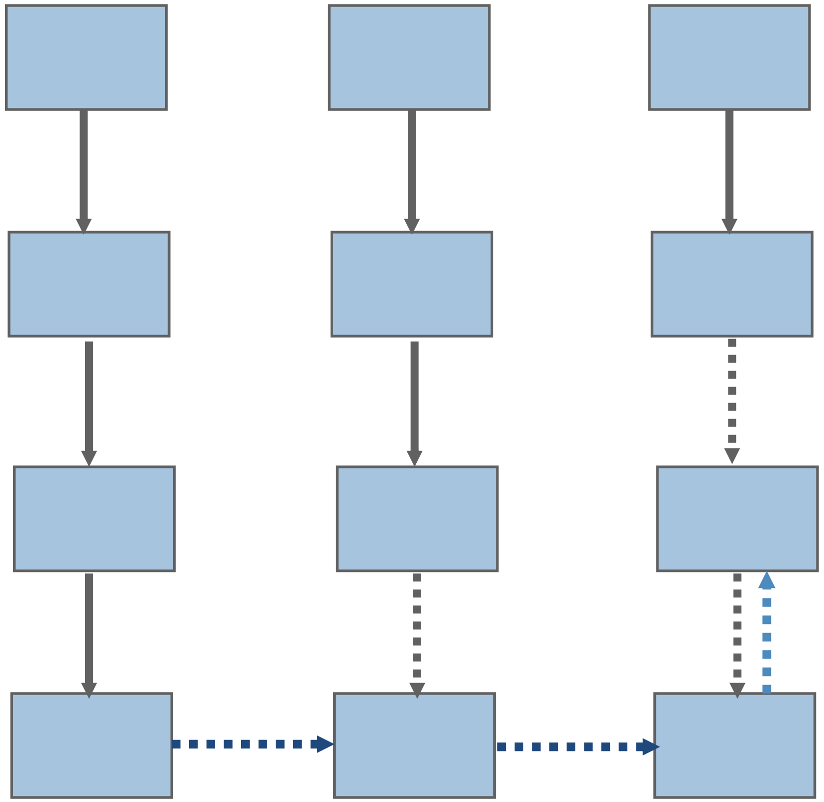
- Does not perform well in practice. Why?

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)

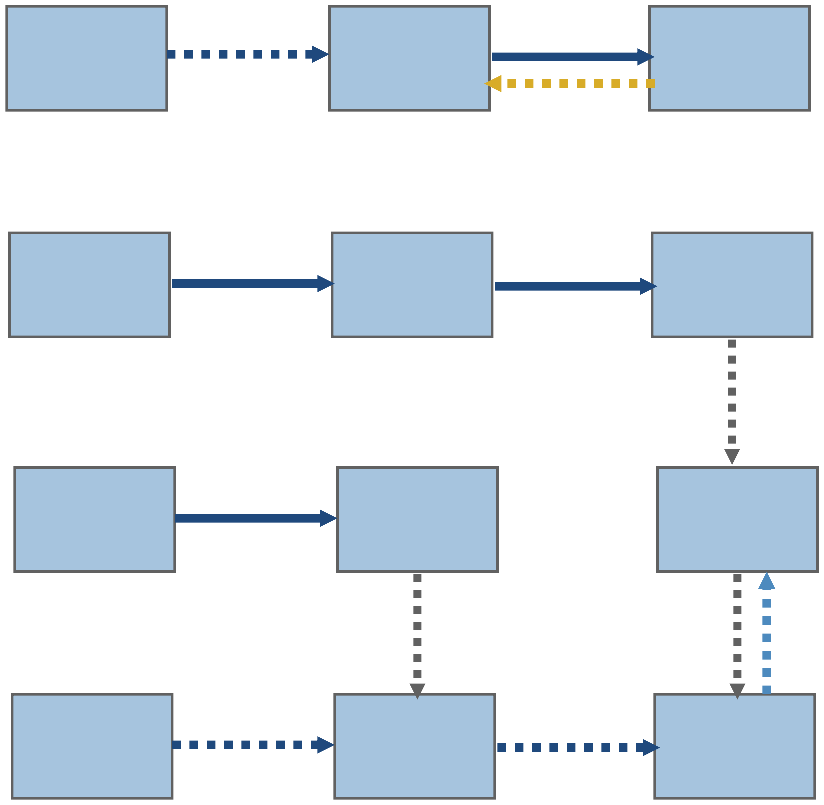
Mesh Exchange - Step 1

- Exchange data on a mesh



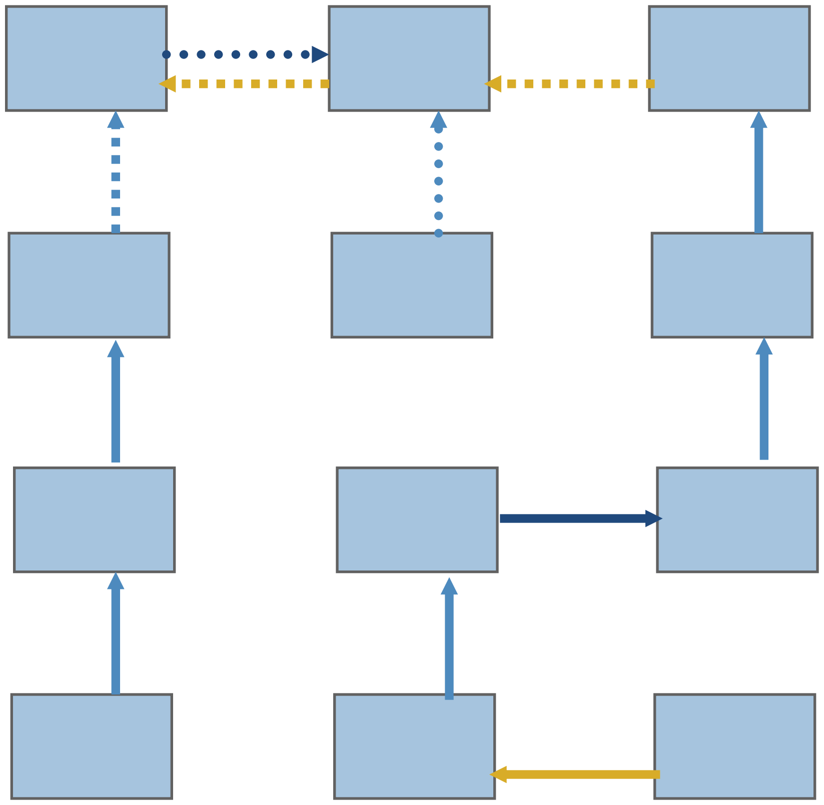
Mesh Exchange - Step 2

- Exchange data on a mesh



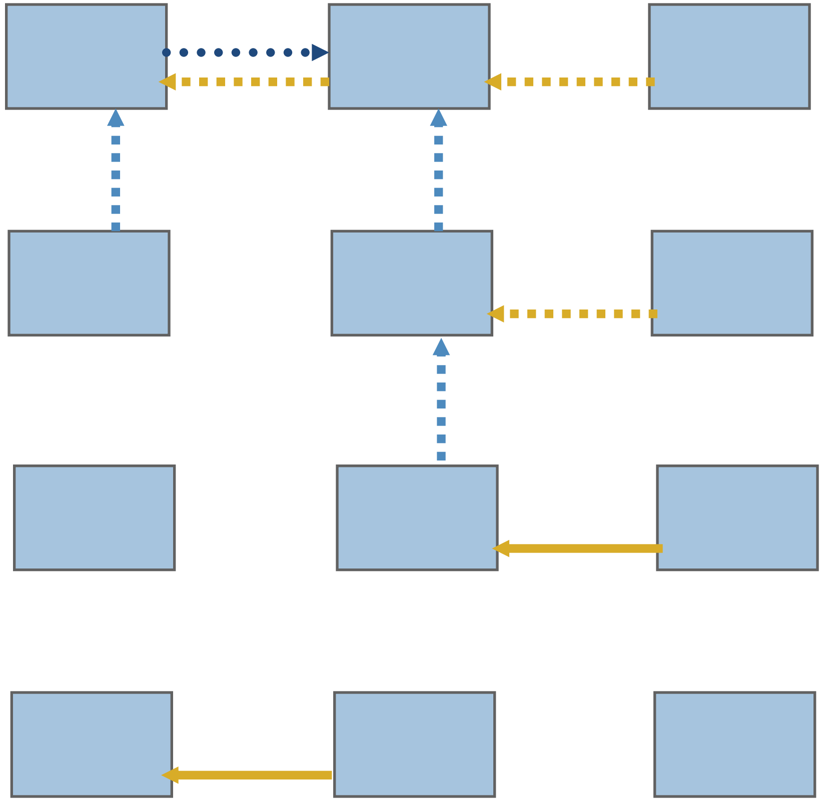
Mesh Exchange - Step 3

- Exchange data on a mesh



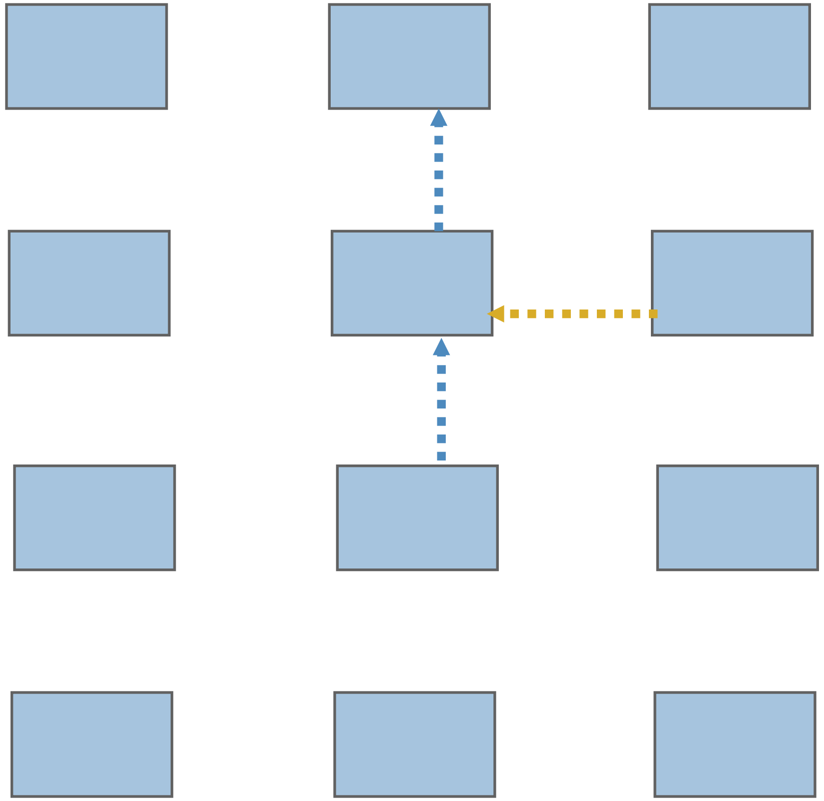
Mesh Exchange - Step 4

- Exchange data on a mesh



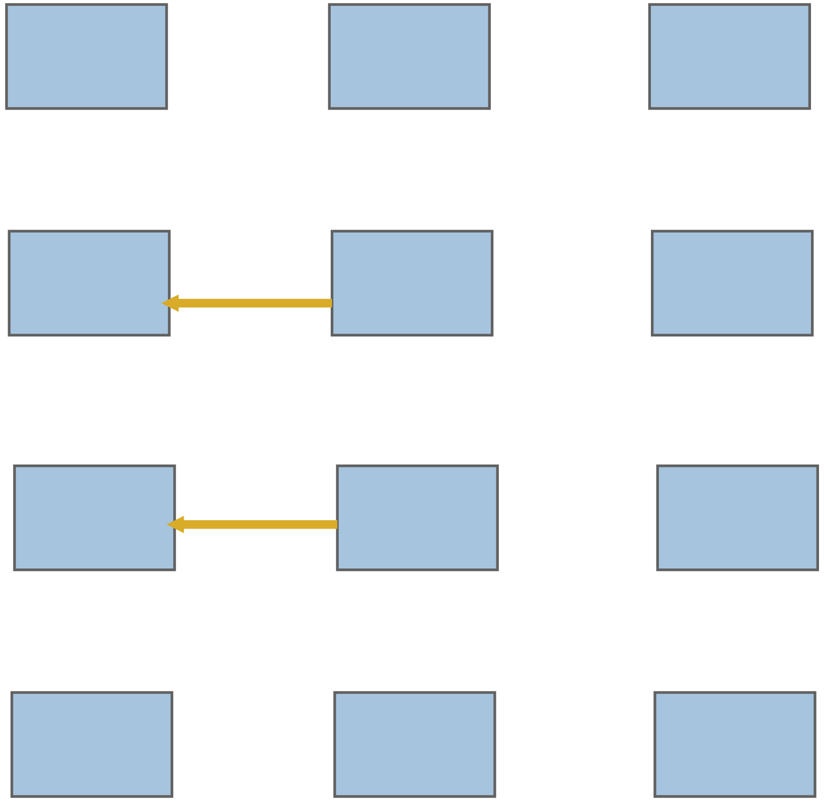
Mesh Exchange - Step 5

- Exchange data on a mesh

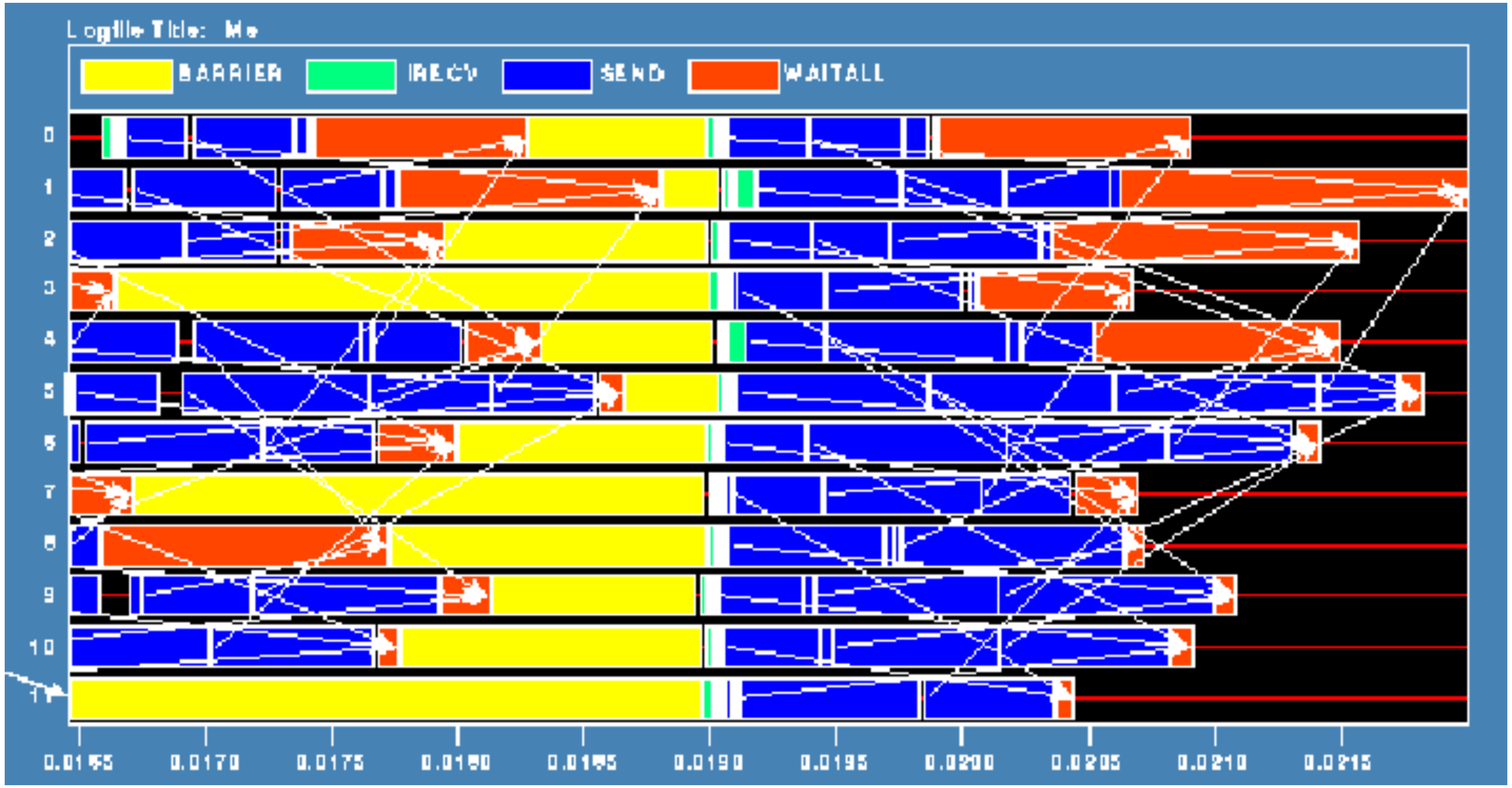


Mesh Exchange - Step 6

- Exchange data on a mesh

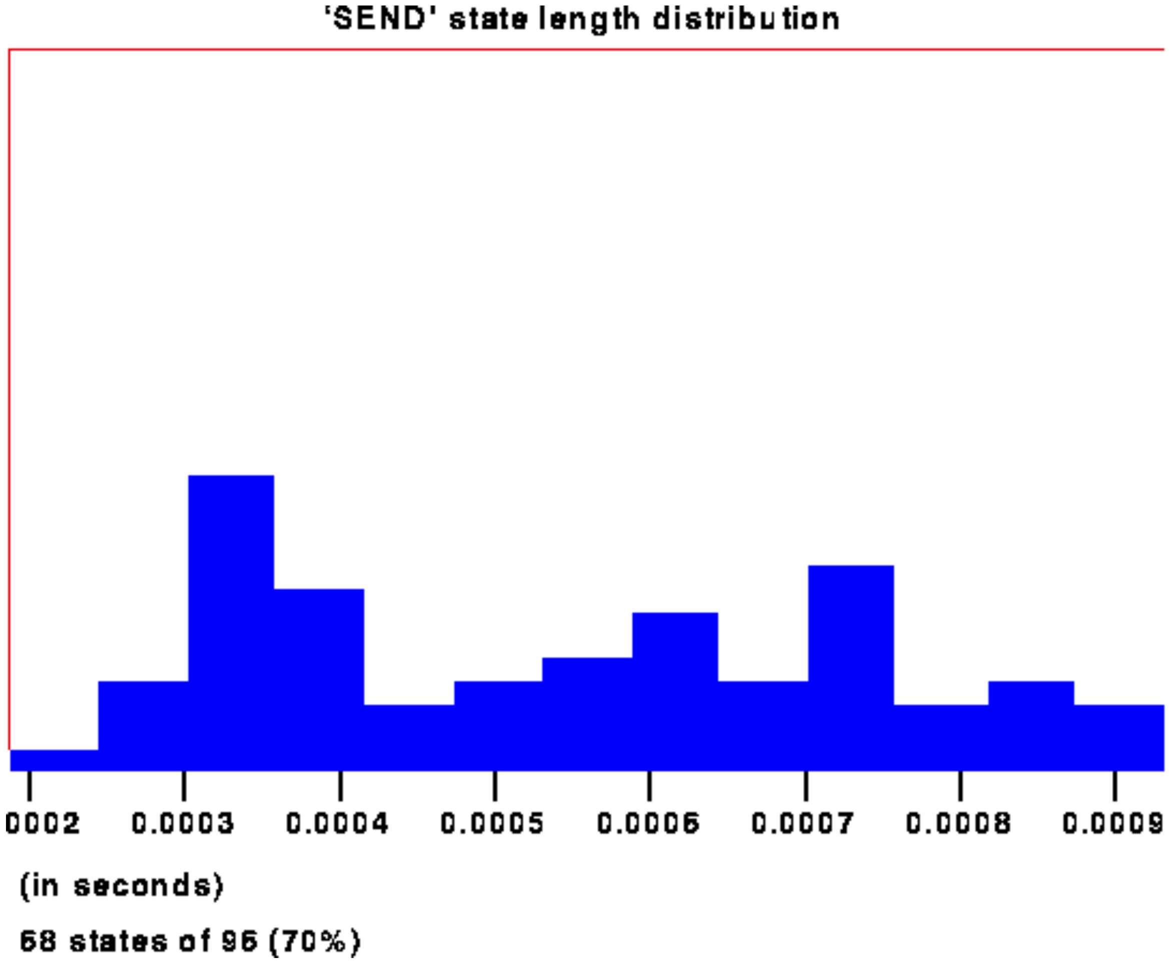


Timeline



- Note that process 1 finishes last, as predicted

Distribution of Sends



Why Six Steps?

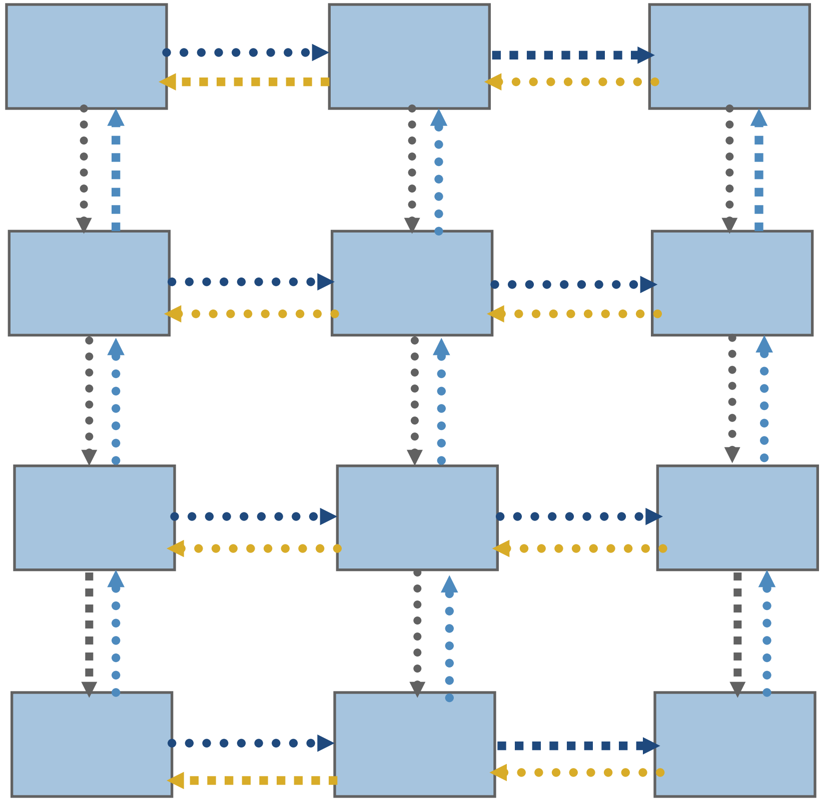
- Ordering of Sends introduces delays when there is contention at the receiver
- Takes roughly twice as long as it should
- Bandwidth is being wasted
- Same thing would happen if using memcpy and shared memory

Fix 2: Use Isend and Irecv

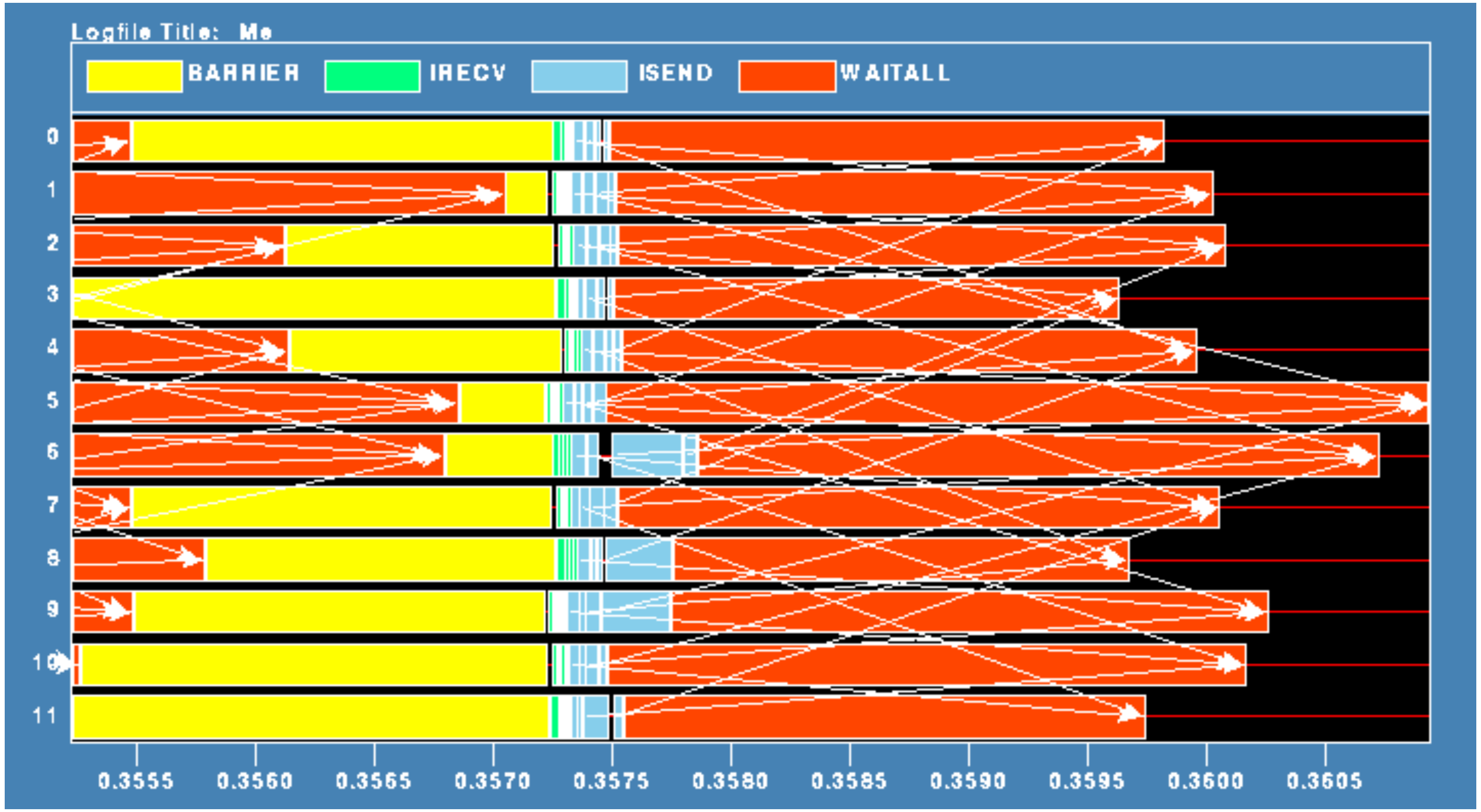
```
Do i=1, n_neighbors
  Call MPI_Irecv(inedge(1,i), len, MPI_REAL, nbr(i), tag, comm, requests(i), ierr)
Enddo
Do i=1,n_neighbors
  Call MPI_Isend(edge(1,i), len, MPI_REAL, nbr(i), tag, comm,
                 requests(n_neighbors+i), ierr)
Enddo
Call MPI_Waitall(2*n_neighbors, requests, statuses, ierr)
```

Mesh Exchange - Steps 1-4

- Four interleaved steps



Timeline with Isend-Irecv



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

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

Collectives: Blocking and Nonblocking

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.

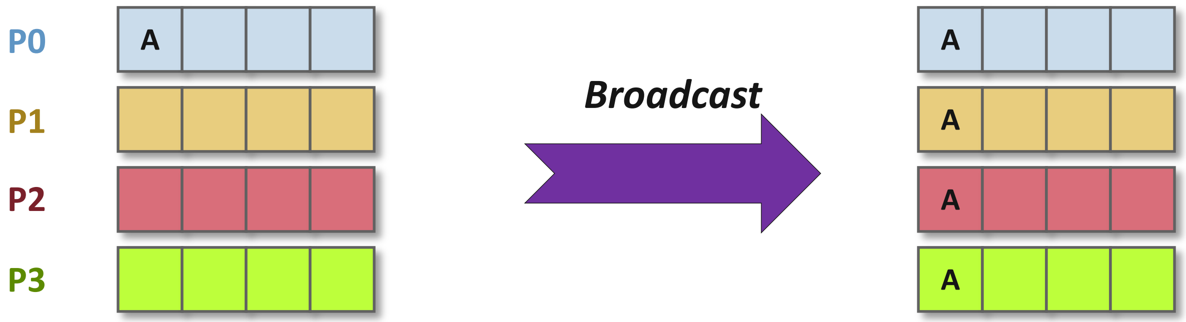
MPI Collective Communication

- Communication and computation is coordinated among a group of processes in a communicator
- Tags are not used; different communicators deliver similar functionality
- Nonblocking 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 the communicator **`comm`** call it
  - A process cannot get out of the barrier until all other processes have reached barrier

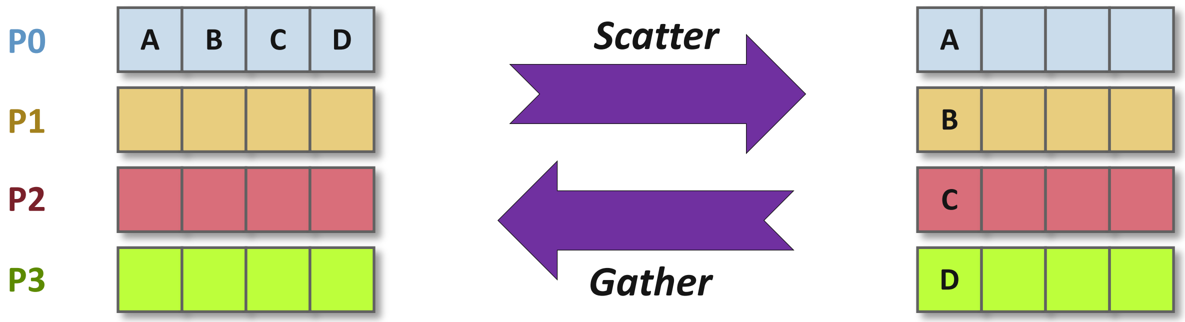
Collective Data Movement



P0	A			
P1				
P2				
P3				

Broadcast

A			
A			
A			
A			



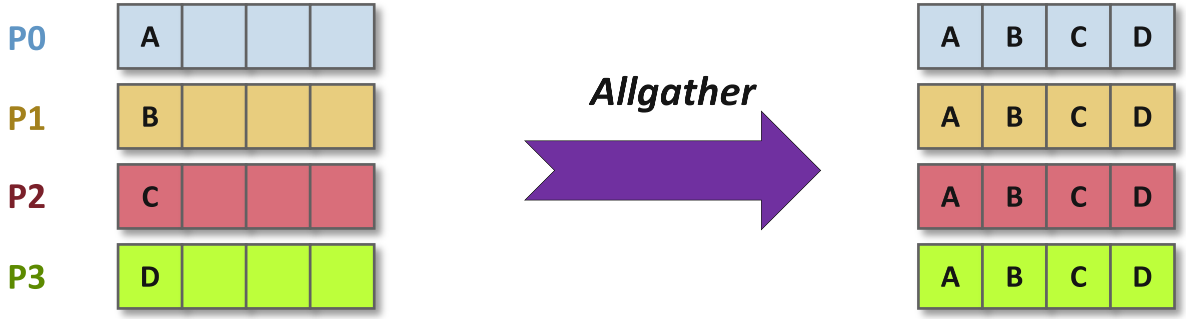
P0	A	B	C	D
P1				
P2				
P3				

Scatter

Gather

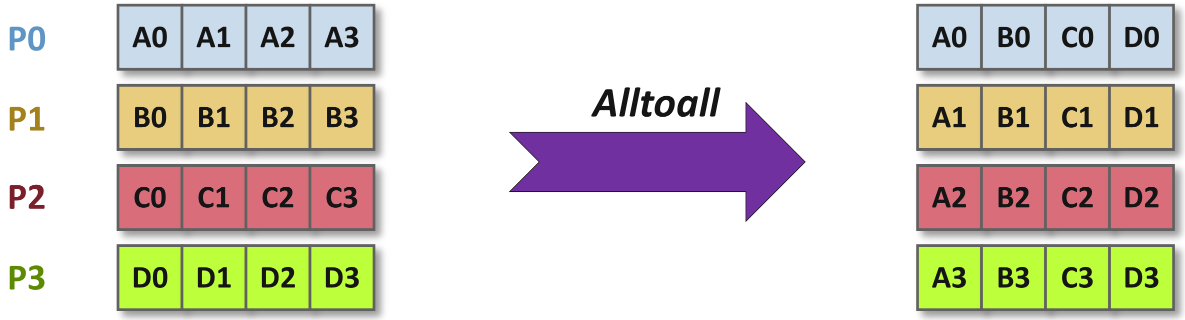
A			
B			
C			
D			

More Collective Data Movement



Allgather

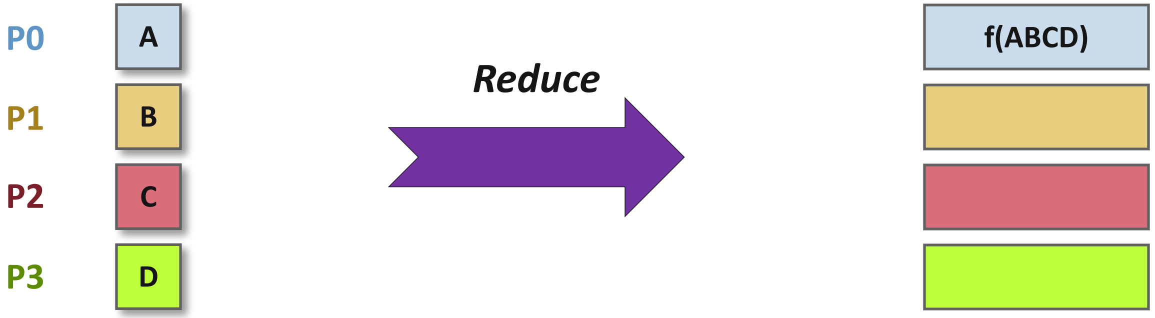
P0	A				A	B	C	D
P1	B				A	B	C	D
P2	C				A	B	C	D
P3	D				A	B	C	D

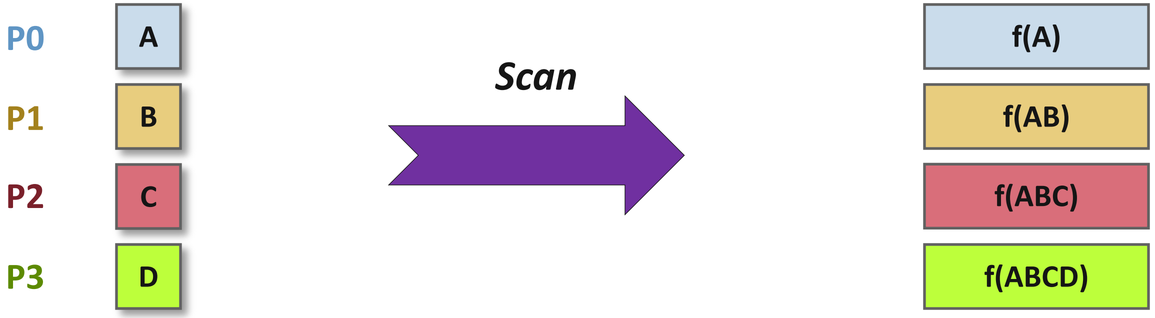


Alltoall

P0	A0	A1	A2	A3	A0	B0	C0	D0
P1	B0	B1	B2	B3	A1	B1	C1	D1
P2	C0	C1	C2	C3	A2	B2	C2	D2
P3	D0	D1	D2	D3	A3	B3	C3	D3

Collective Computation





MPI Collective Routines

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

MPI Built-in Collective Computation Operations

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

Defining your own Collective Operations

- Create (and free) your own collective computations with:

```
MPI_Op_create(user_fn, commutes, &op);
MPI_Op_free(&op);

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

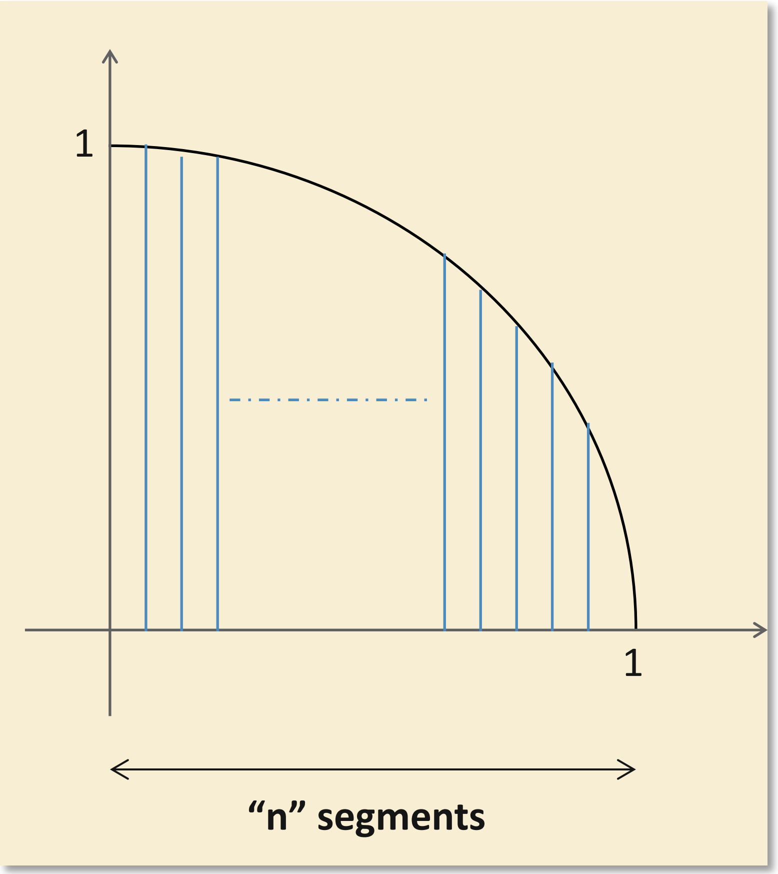
- The user function should perform:

```
for i from 0 to len-1
     inoutvec[i]  =  invec[i]  op  inoutvec[i];
```

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

Example: Calculating Pi (1/3)

- Calculating the value of “pi” via numerical integration
  - Divide interval up into subintervals
  - Assign subintervals to processes
  - Each process calculates partial sum
  - Add all the partial sums together to get pi



1. Width of each segment (w) will be 1/n
2. Distance (d(i)) of segment “i” from the origin will be “i * w”
3. Height of segment “i” will be sqrt(1 – [d(i)]^2)

Example: Calculating Pi (2/3)

```
#include <mpi.h>
#include <math.h>
int main(int argc, char *argv[])
{
    [...snip...]
    /* Tell all processes, the number of segments you want */
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    w   = 1.0 / (double) n;
    mypi = 0.0;
    for (i = rank + 1; i <= n; i += size)
        mypi += w * sqrt(1 - (((double) i / n) * ((double) i / n));
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);
    if (rank == 0)
        printf("pi is approximately %.16f, Error is %.16f\n", 4 * pi,
               fabs((4 * pi) - PI25DT));
    [...snip...]
}
```

Example: Calculating Pi (3/3)

- *blocking_coll/cpi.c*
- Calculating the approximate value of PI in parallel

Nonblocking Collective Communication

- Nonblocking (send/recv) communication
  - Deadlock avoidance
  - Overlapping communication/computation
- Collective communication
  - Collection of pre-defined optimized communication patterns
- → Nonblocking collective communication
  - Combines both techniques
  - System noise/imbalance resiliency
  - Semantic advantages

Nonblocking Collective Communication

- Nonblocking variants of all collectives
  - MPI_Ibcast(<bcast args>, MPI_Request *req);
- Semantics
  - Function returns no matter what
  - No guaranteed progress (quality of implementation)
  - Usual completion calls (wait, test) + mixing
  - Out-of order completion
- Restrictions
  - Send and vector buffers may not be updated during operation (like other nonblocking operations)
  - No tags, in-order matching (like other collective operations)
  - 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 Nonblocking Barrier?

- 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** noncollectively but **synchronize** collectively!

Section Summary

- Collectives are a very powerful feature in MPI
- Optimized heavily in most MPI implementations
  - Algorithmic optimizations (e.g., tree-based communication)
  - Hardware optimizations (e.g., network or switch-based collectives)
- Matches the communication pattern of many applications
- Nonblocking collectives combine the semantics of nonblocking point-to-point and blocking collectives
  - Natural extension to blocking collectives for event-driven programming
  - Hardware implementations already exist for most (but not all) nonblocking collectives

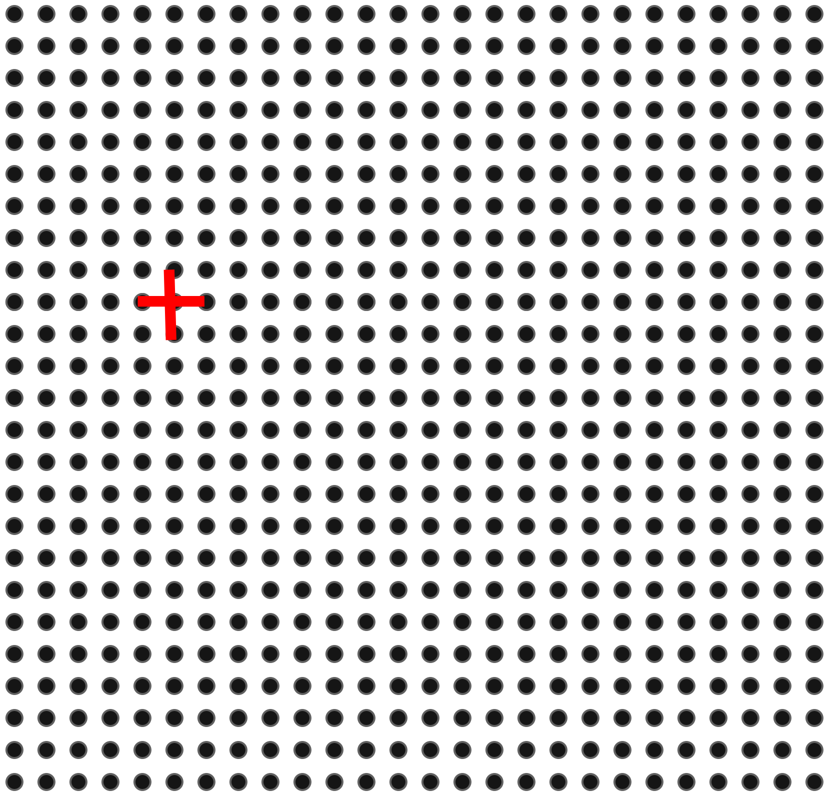
Running Example: Stencil

Running Example: Regular Mesh Algorithms

- Many scientific applications involve the solution of partial differential equations (PDEs)
- Many algorithms for approximating the solution of PDEs rely on forming a set of difference equations
  - Finite difference, finite elements, finite volume
- The exact form of the differential equations depends on the particular method
  - From the point of view of parallel programming for these algorithms, the operations are the same
- Five-point stencil is a popular approximation solution

The Global Data Structure

- Each circle is a mesh point
- Difference equation evaluated at each point involves the four neighbors
- The red “plus” is called the method’s stencil
- Good numerical algorithms form a matrix equation Au=f; solving this requires computing Bv, where B is a matrix derived from A. These evaluations involve computations with the neighbors on the mesh.



MPI Examples

- mpi-examples.zip
- Copy to your own project directory and unzip
- Allocate or submit job using reservation on Perlmutter
  - salloc/sbatch -q shared -C gpu -A ntrain5 -c 1 -n 16 --reservation=atpesc_aug2

Alternative Environment Setup

- Try MPI on your own machine
- Use Docker Image

```
1.   mkdir $HOME/mpi-tutorial
2.   docker pull pmrs/mpi-tutorial
3.   docker run --rm -it -v $HOME/mpi-tutorial:/project pmrs/mpi-tutorial
```

This creates a container and opens a shell.
It mounts local directory $HOME/mpi-tutorial as /project in the container.

```
4.   (in container) cp -r $HOME/examples /project/examples
```

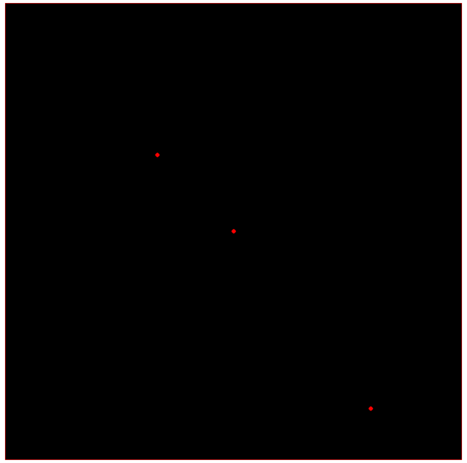
Example: Stencil

- *serial/stencil.c*
- Simple stencil code in single process

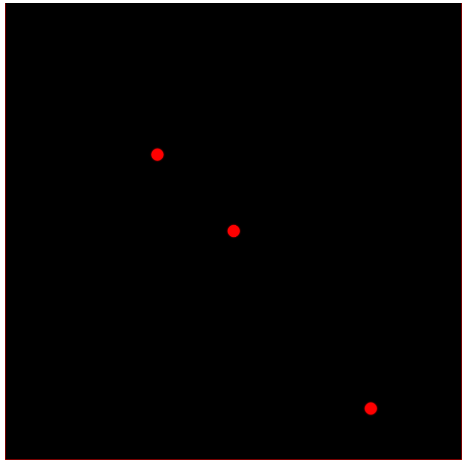
```
# make
# sbatch -q shared -C gpu -A ntrain5 -n 1 --
reservation=atpesc_aug2 -t 30:00 ./job.sh # Perlmutter
```

```
# ./stencil <domain size> <heat source value> <iterations>
```

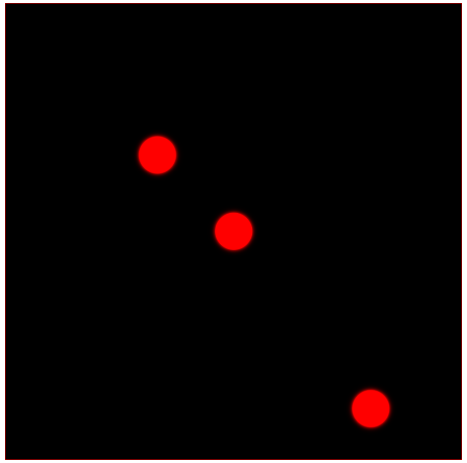
```
 % ./stencil 1000 1000 10
   last heat: 19666.992188 time: 0.045576
```



iter=10



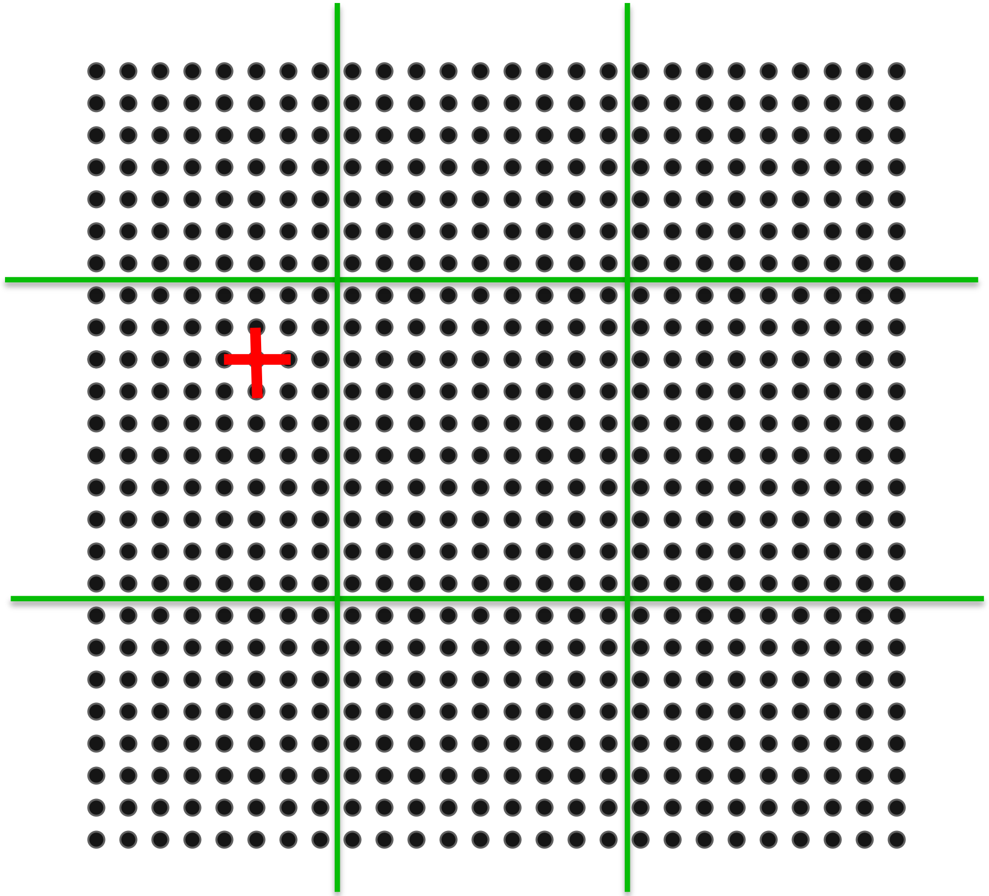
iter=100



iter=1000

The Global Data Structure

- Each circle is a mesh point
- Difference equation evaluated at each point involves the four neighbors
- The red “plus” is called the method’s stencil
- Good numerical algorithms form a matrix equation Au=f; solving this requires computing Bv, where B is a matrix derived from A. These evaluations involve computations with the neighbors on the mesh.
- Decompose mesh into equal sized (work) pieces



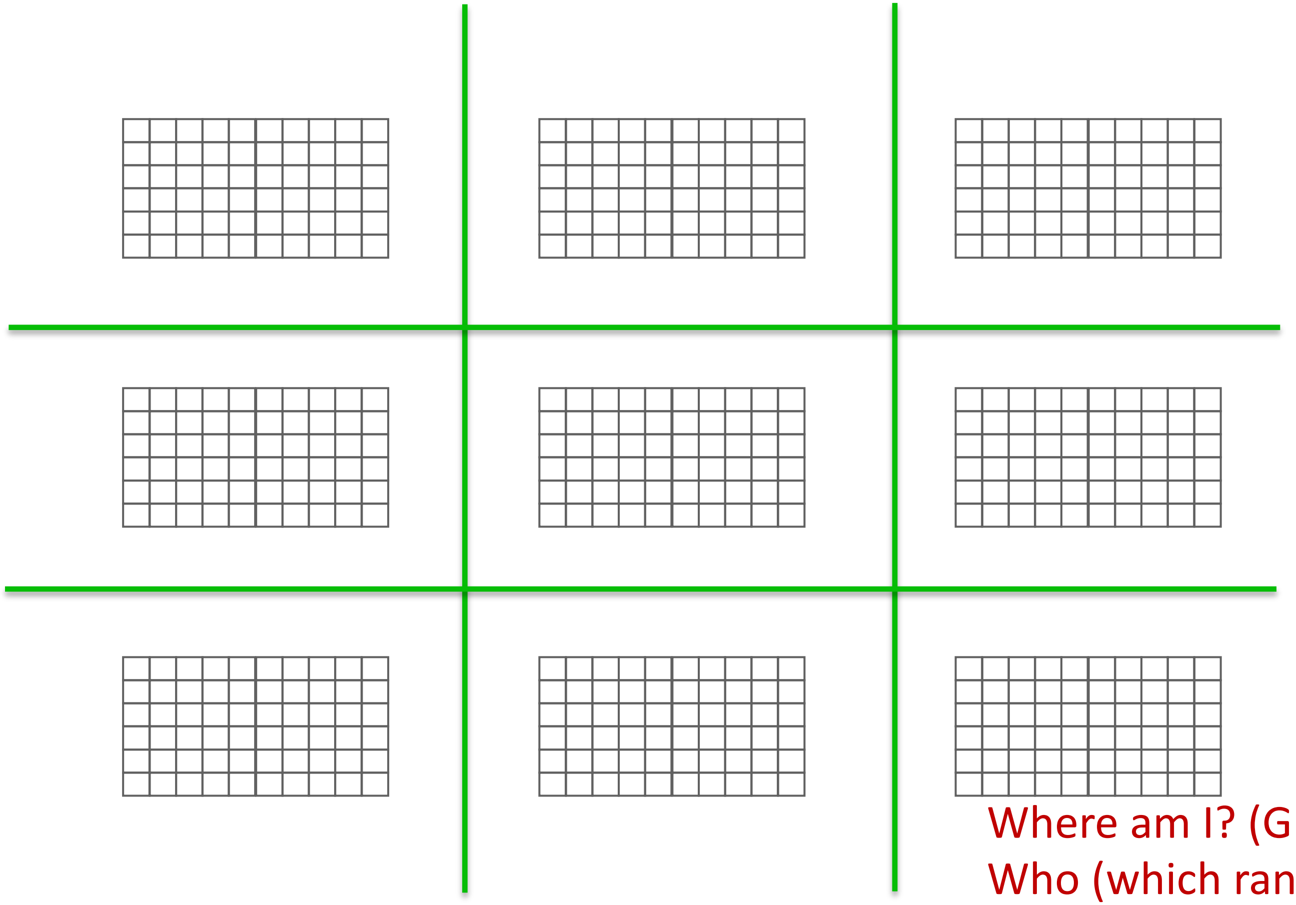
Domain Decompositioin

Parameters for domain decomposition:

N = Size of the edge of the global problem domain (assuming square)

PX, PY = Number of processes in X and Y dimension

N % PX == 0, N % PY == 0

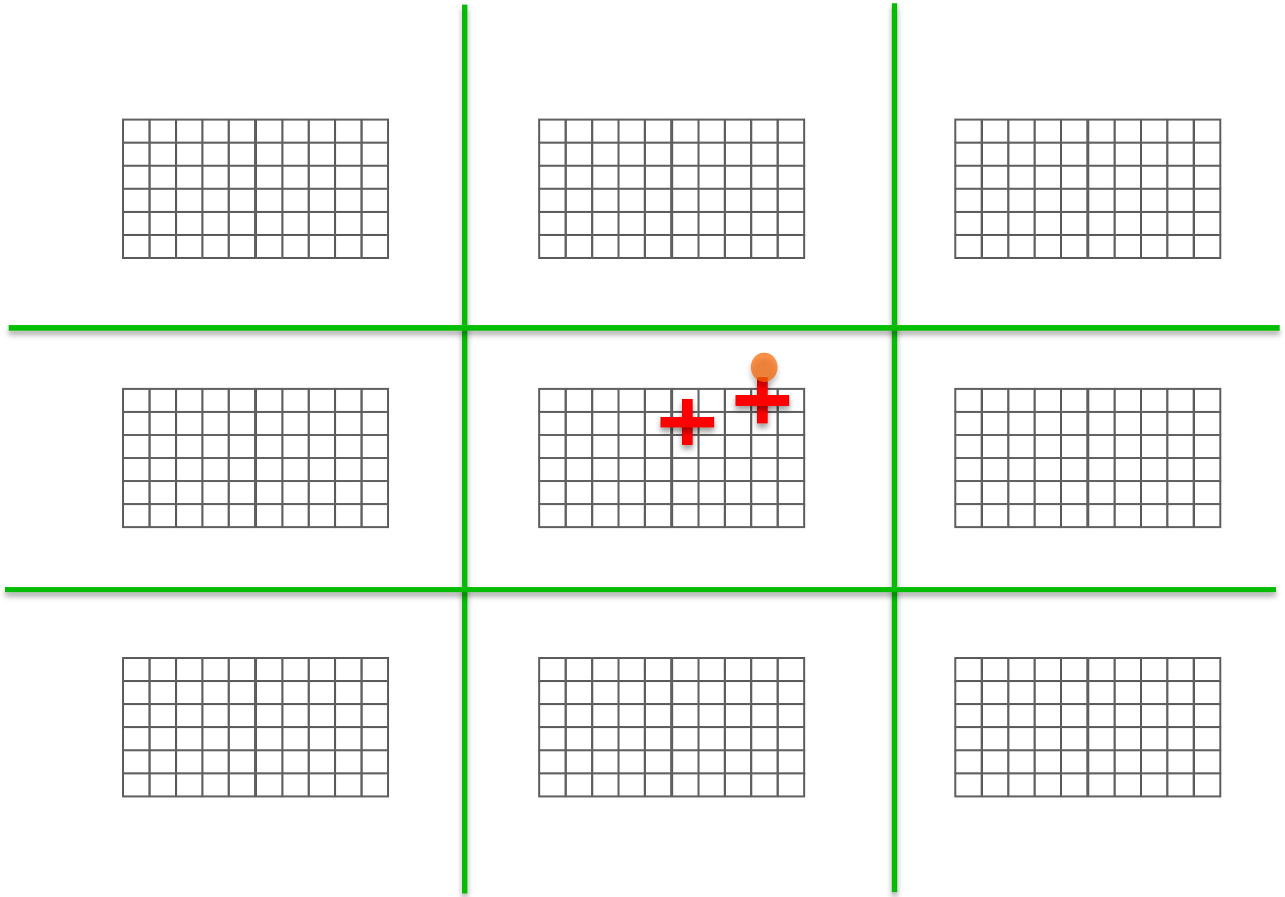


Where am I? (Global offset)

Who (which ranks) are my neigbhors?

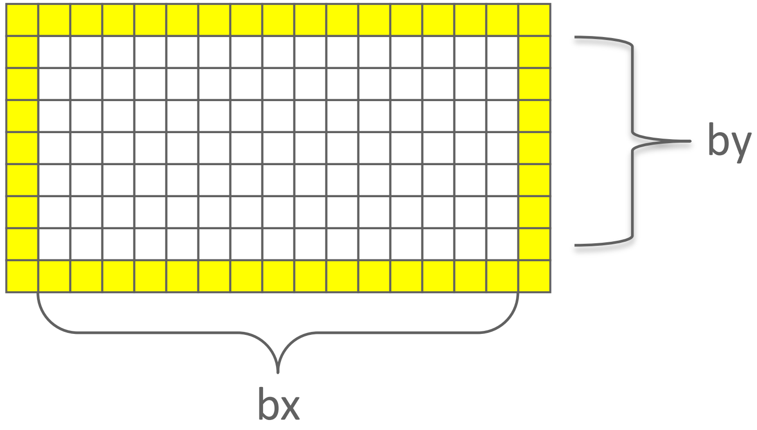
Use MPI_PROC_NULL for boundary

Necessary Data Transfers



The Local Data Structure

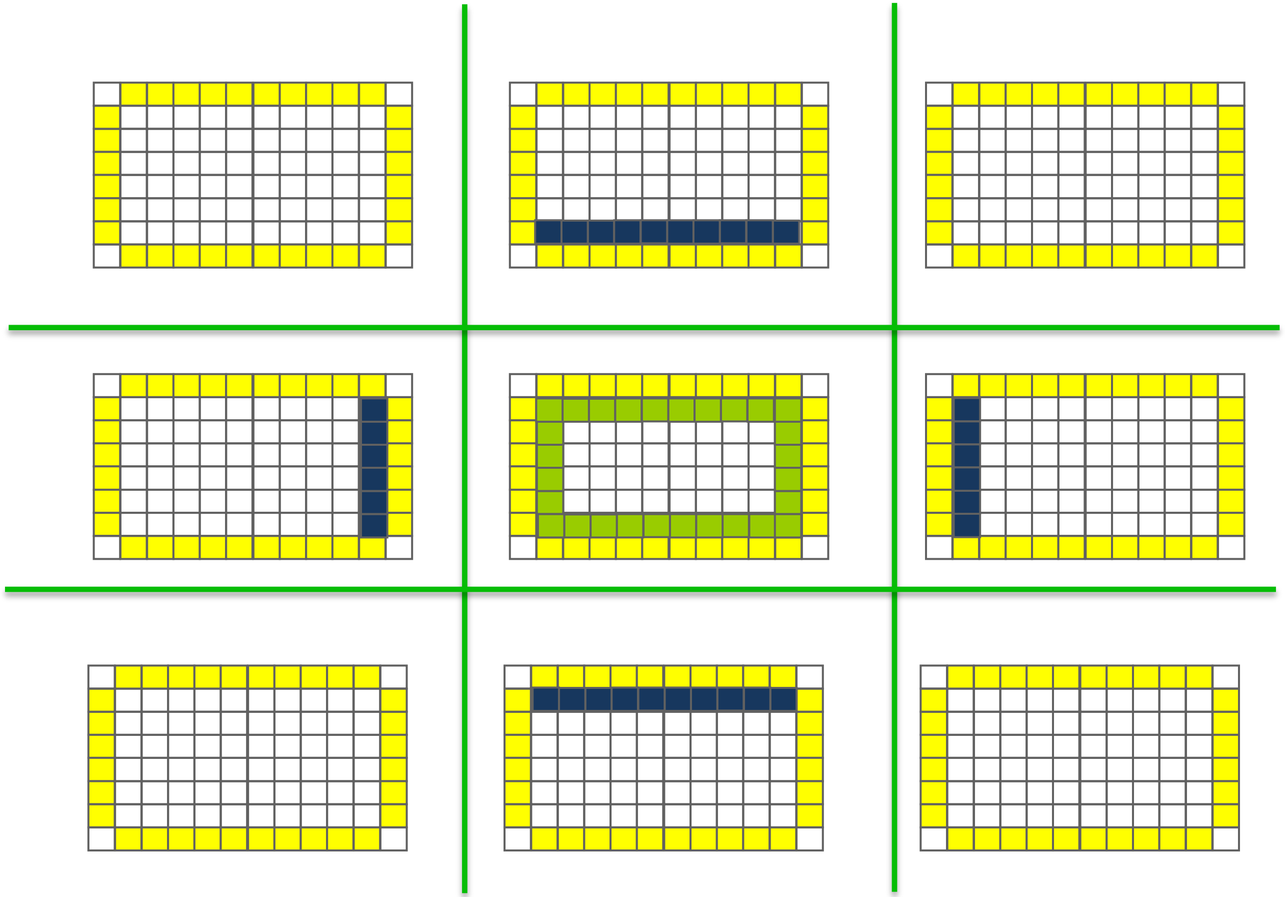
- Each process has its local “patch” of the global array
  - “bx” and “by” are the sizes of the local array
  - Always allocate a halo around the patch
  - Array allocated of size (bx+2)x(by+2)



Check the **`alloc_bufs`** function to see how buffers are allocated

Necessary Data Transfers

- Provide access to remote data through a halo exchange (5 point stencil)



Check the **`update_grid`** function to see how it is done

Example: Stencil with Nonblocking Send/recv

- *nonblocking_p2p/stencil.c*
- Simple stencil code using nonblocking point-to-point operations

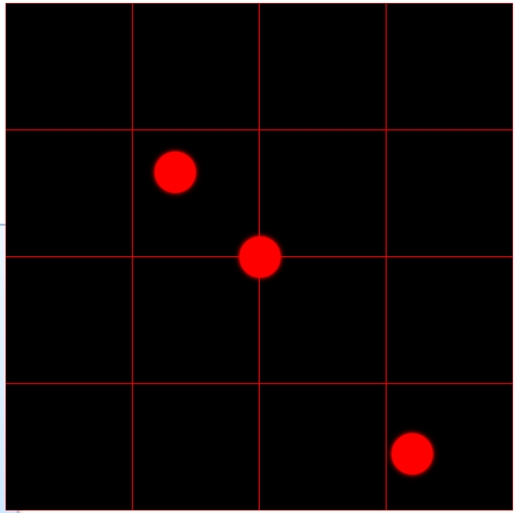
```
# make
# sbatch -q shared -C gpu -A ntrain5 -n 16 --reservation=atpesc_aug2 -t
30:00 ./job.sh # Perlmutter
```

For Cray systems

```
# srun/aprun -n <nproc> ./stencil <domain size> <heat source value> <iteration> <px> <py>
```

For most other systems (or local machine)

```
# mpirun -n <nproc> ./stencil <domain size> <heat source value> <iterations> <px> <py>

<nproc> == <px> * <py>
```



iter=1000

```
 % mpirun -n 16./stencil 1000 1000 10 4 4
   [0] last heat: 19666.992188 time: 0.045576
```

Derived Datatypes

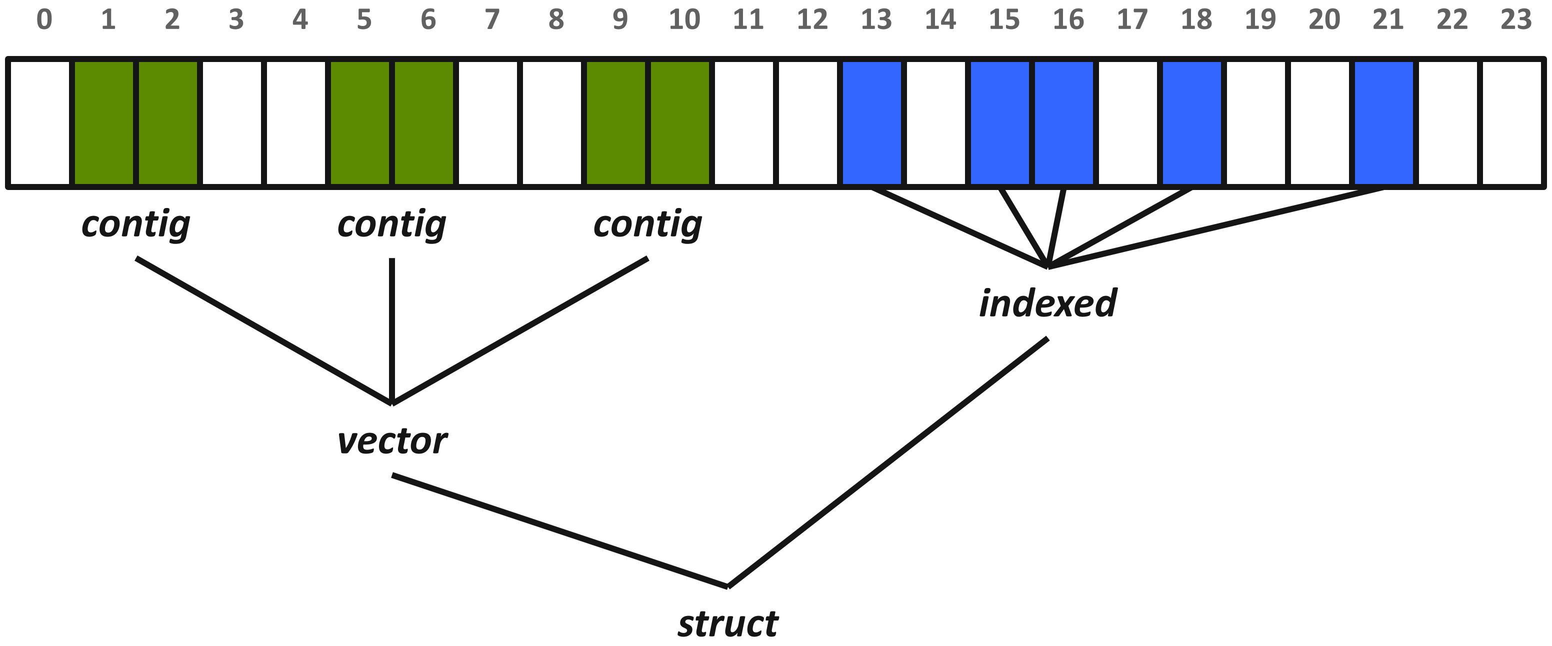
Introduction to Datatypes in MPI

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

Simple/Predefined Datatypes

- Equivalents exist for all C, C++ and Fortran native datatypes
  - C int → MPI_INT
  - C float → MPI_FLOAT
  - C double → MPI_DOUBLE
  - C uint32_t → MPI_UINT32_T
  - Fortran integer → MPI_INTEGER
- For more complex or user-created datatypes, MPI provides routines to represent them as well
  - Contiguous
  - Vector/Hvector
  - Indexed/Indexed_block/Hindexed/Hindexed_block
  - Struct
  - Some convenience types (e.g., subarray)

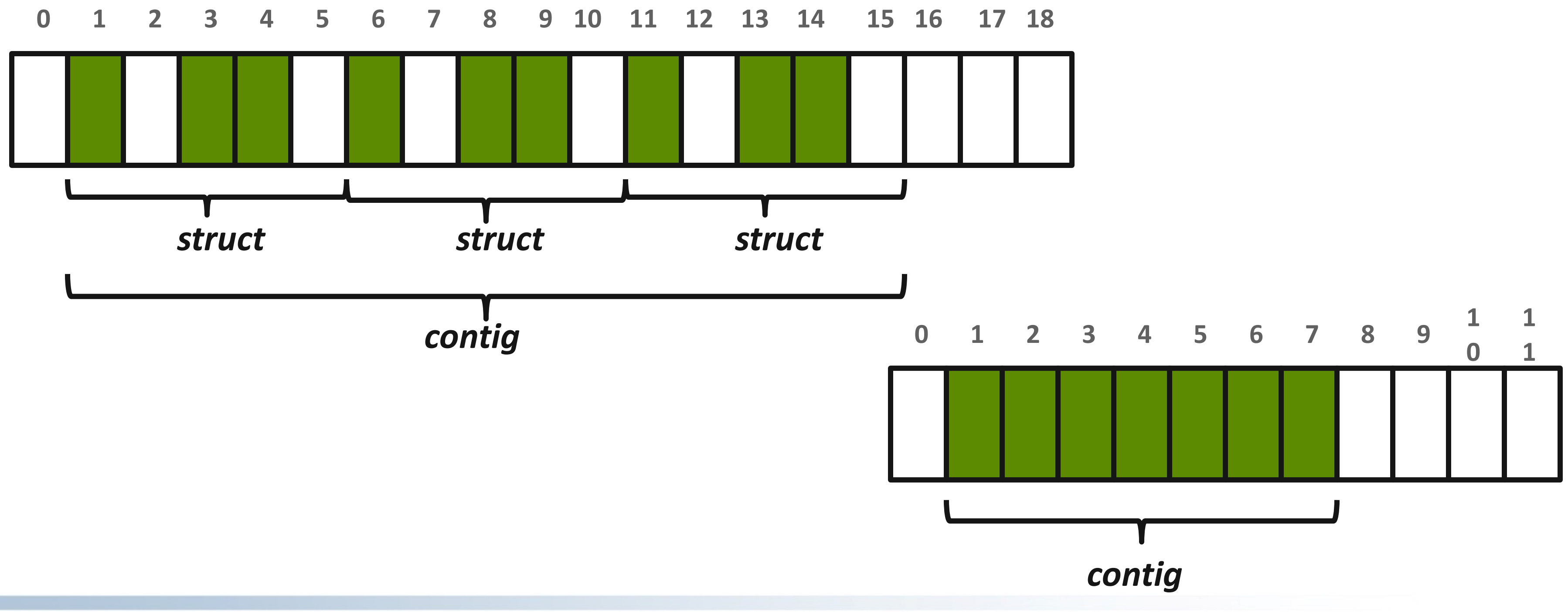
Derived Datatype Example



MPI_Type_contiguous

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

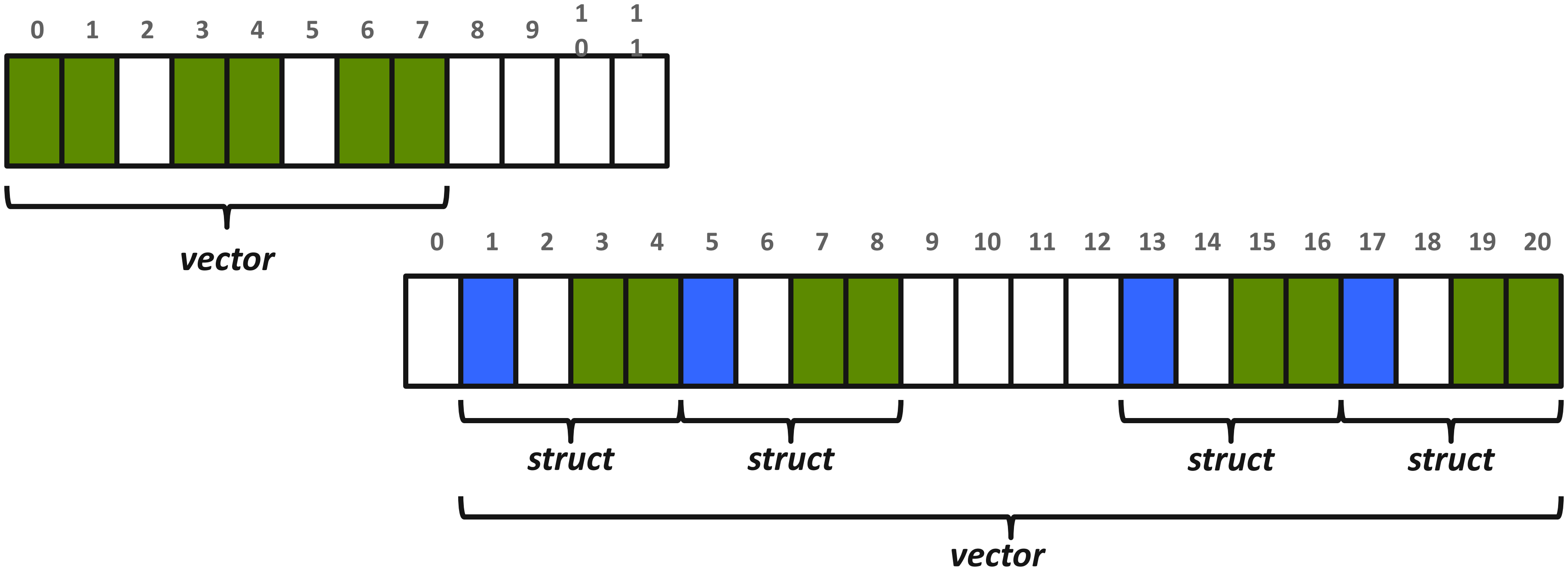
- Contiguous array of oldtype
- Should not be used as last type (can be replaced by count)



MPI_Type_vector

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

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



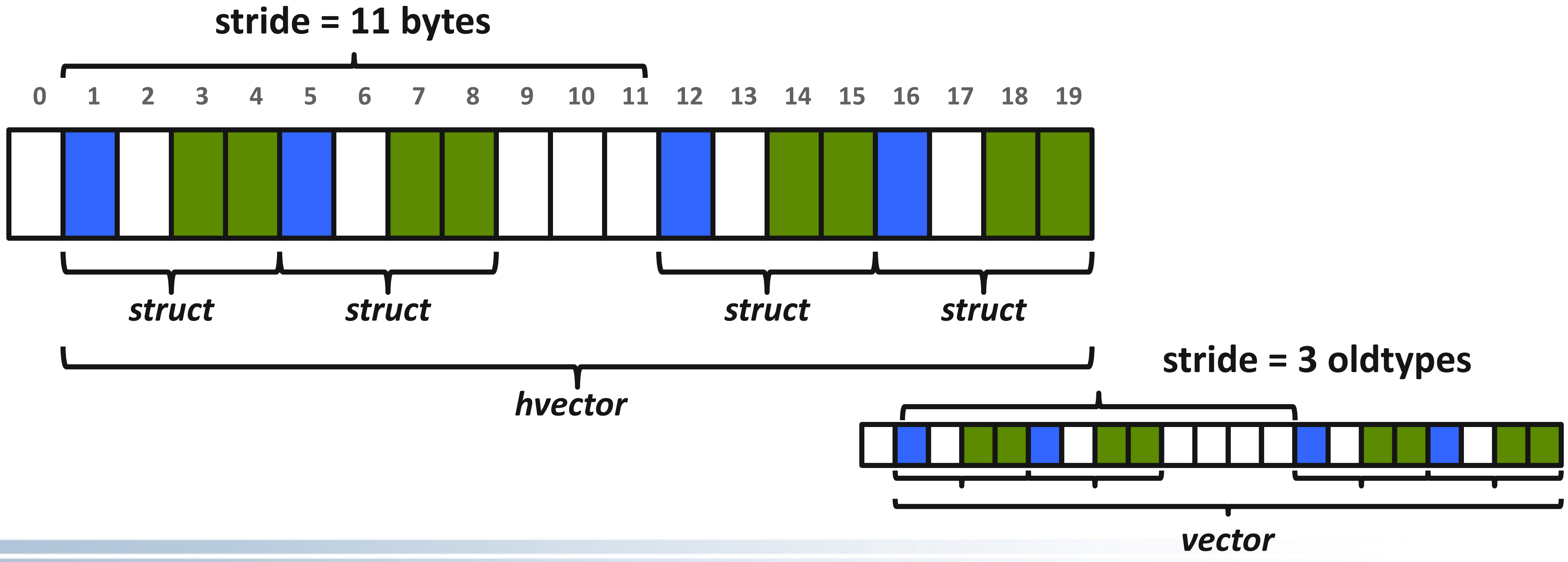
Commit, Free, and Dup

- Types must be committed before use
  - Only the ones that are used!
  - MPI_Type_commit may perform heavy optimizations (and hopefully will)
- MPI_Type_free
  - Free MPI resources of datatypes
  - Does not affect types built from it
- MPI_Type_dup
  - Duplicates a type
  - Library abstraction (composability)

MPI_Type_create_hvector

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

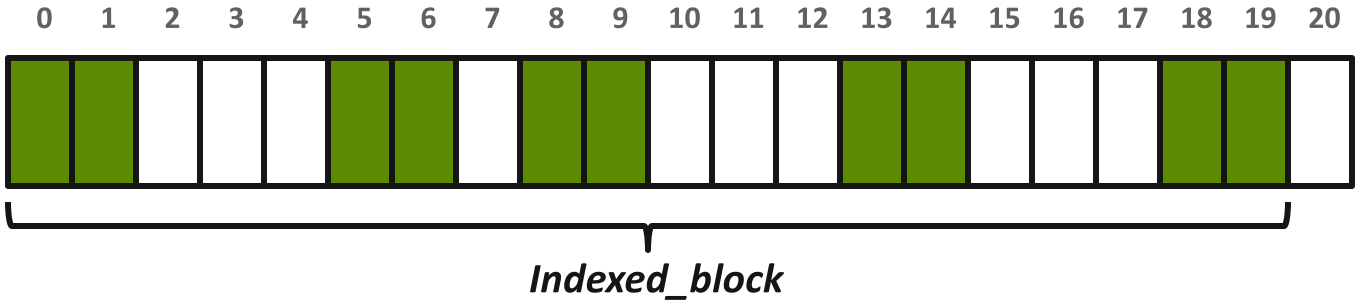
- Create byte strided vectors
- Useful for composition, e.g., vector of structs



MPI_Type_create_indexed_block

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

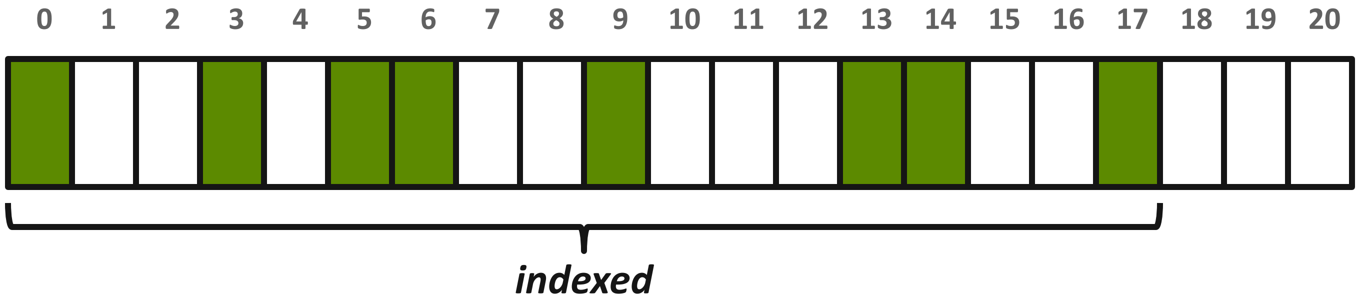
- Pulling irregular subsets of data from a single array
  - dynamic codes with index lists, expensive though!
  - blen=2
  - displs={0,5,8,13,18}



MPI_Type_indexed

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

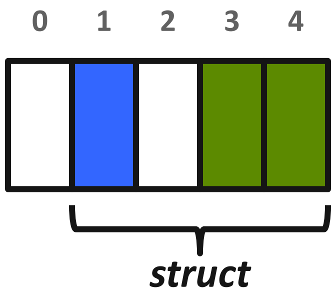
- Like indexed_block, but can have different block lengths
  - blen={1,1,2,1,2,1}
  - displs={0,3,5,9,13,17}



MPI_Type_create_struct

```
MPI_Type_create_struct(int count, int *array_of_blocklens, int *array_of_displacements,
                       MPI_Datatype *array_of_types, MPI_Datatype *newtype)
```

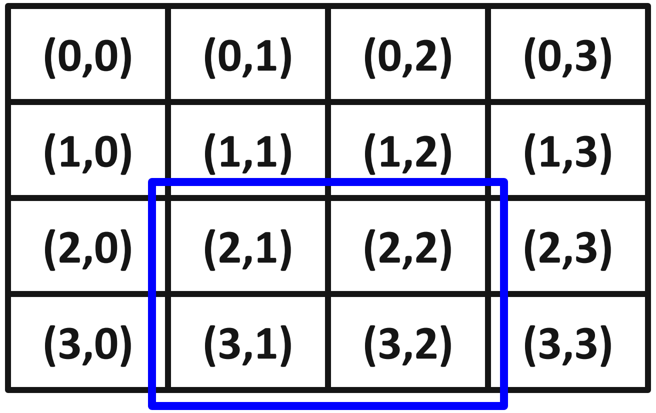
- Most general constructor, allows different types and arbitrary arrays (also most costly)



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)
```

- Convenience function for creating datatypes for array segments
- Specify subarray of n-dimensional array (sizes) by start (starts) and size (subsize)



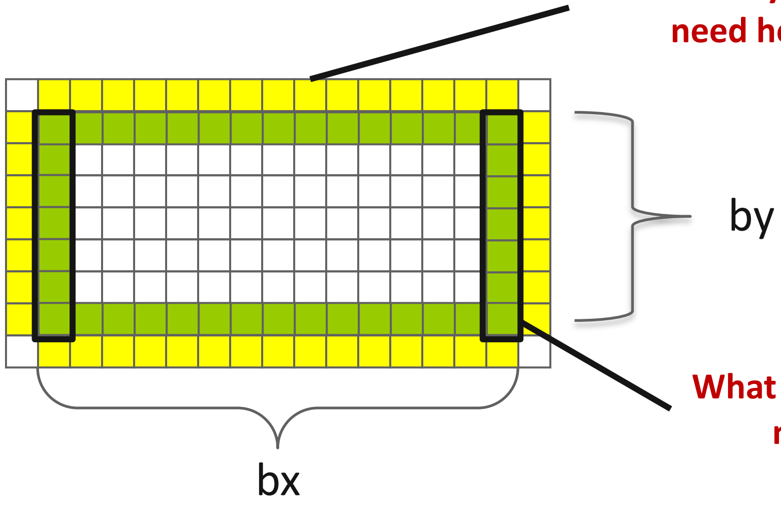
(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

Section Summary

- Derived datatypes are a sophisticated mechanism to describe ANY layout in memory
  - Hierarchical construction of derived datatypes allows them to be just as complex as the data layout is
  - More complex layouts require more complex datatype constructions
- Current state of MPI implementations might be a bit lagging in performance, but it is improving
  - Increasing amount of hardware support to process derived datatypes on the network hardware
  - If the performance is lagging when you try it out, complain to the MPI implementer, don’t just stop using it!

Exercise: Stencil with Derived Datatypes (1/2)

- In the basic version of the stencil code
  - Used nonblocking communication 👍
  - Used manual packing/unpacking of data 👎
- Let’s try to use derived datatypes
  - Specify the locations of the data instead of manually packing/unpacking



What datatype do we need here?

What datatype do we need here?

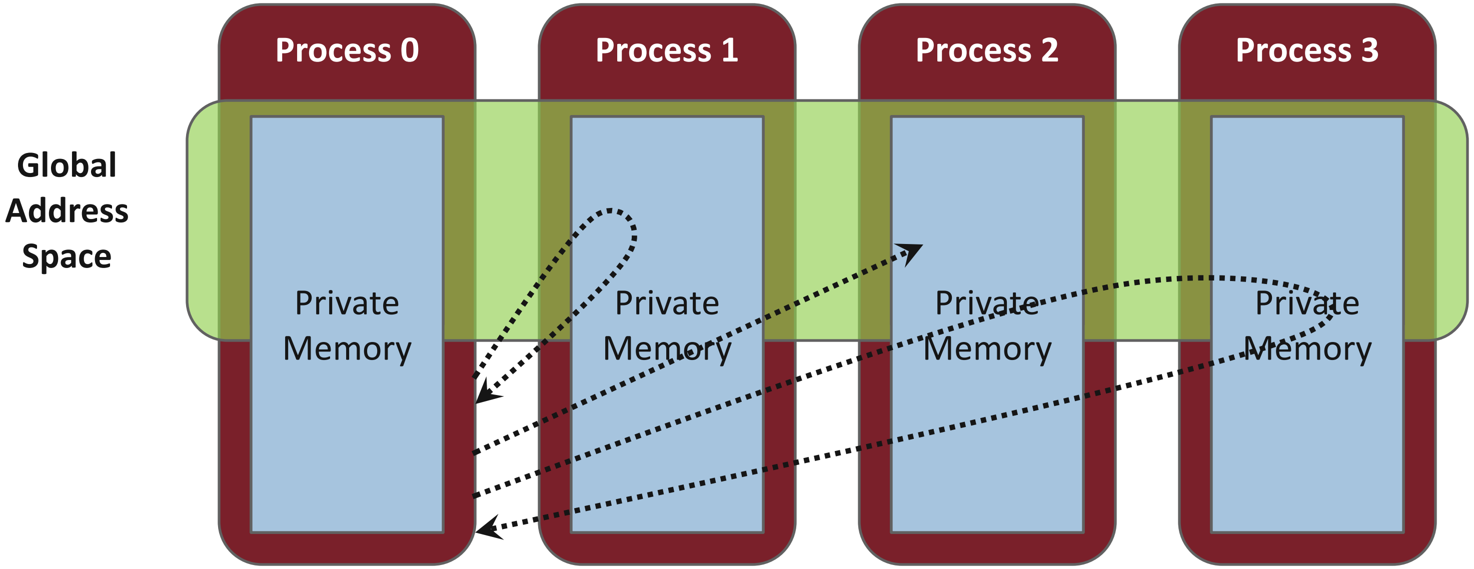
Exercise: Stencil with Derived Datatypes (2/2)

- Nonblocking sends and receives
- Data location specified by MPI datatypes
- Manual packing of data no longer required
- *Start from nonblocking_p2p/stencil.c*
- *Solution in derived_datatype/stencil.c*

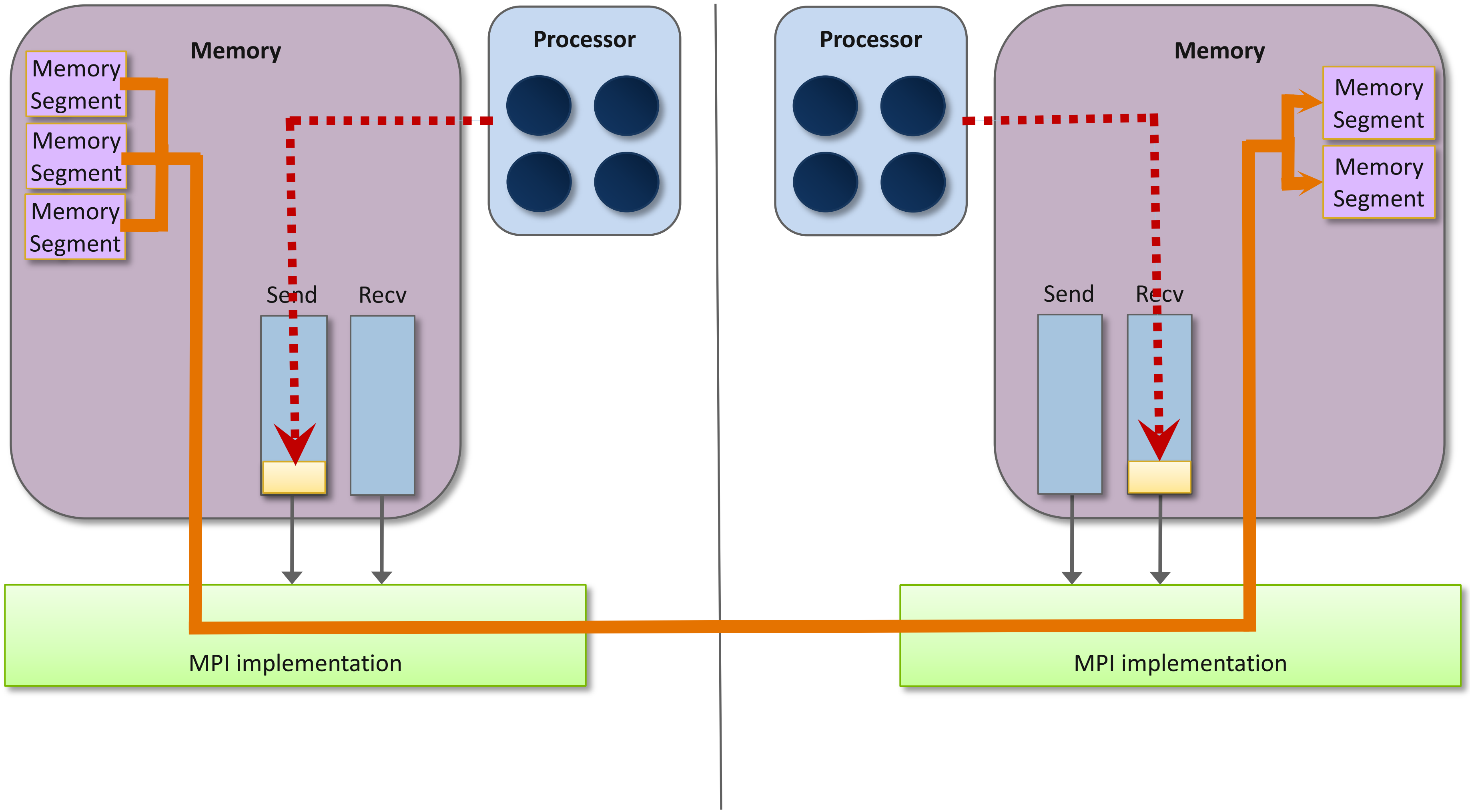
MPI One-sided Communication

One-sided Communication

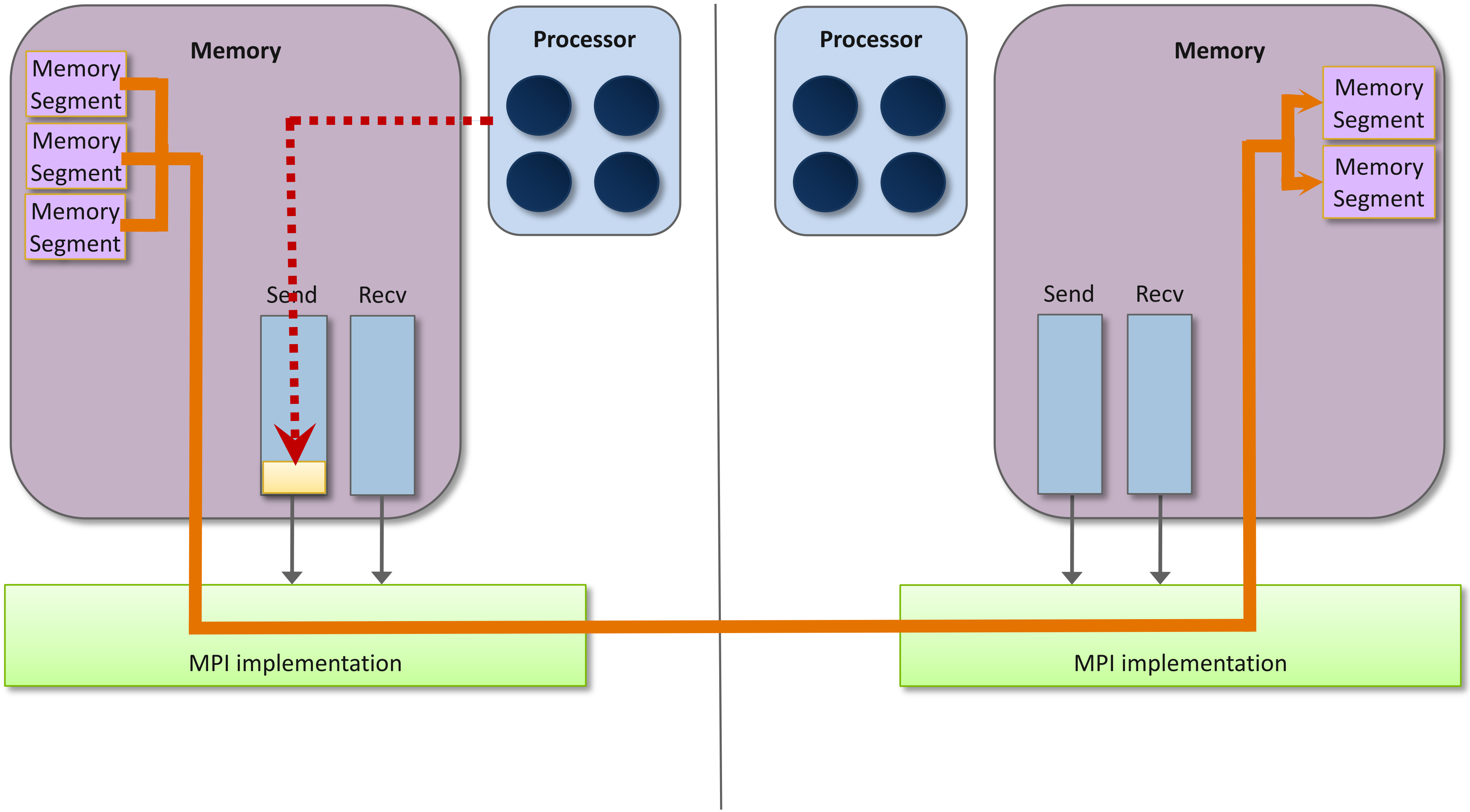
- The basic idea of one-sided communication models is to decouple data movement with process synchronization
  - Should be able to move data without requiring that the remote process synchronize
  - Each process exposes a part of its memory to other processes
  - Other processes can directly read from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



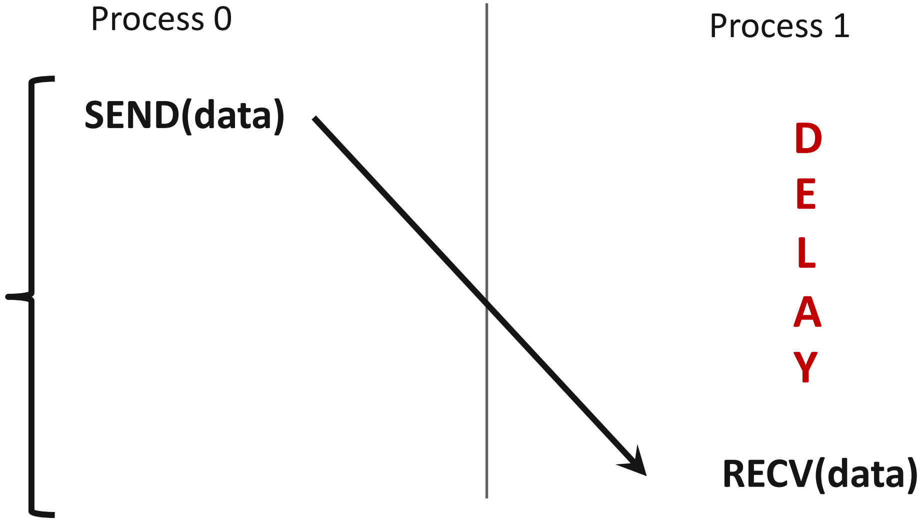
Two-sided Communication Example



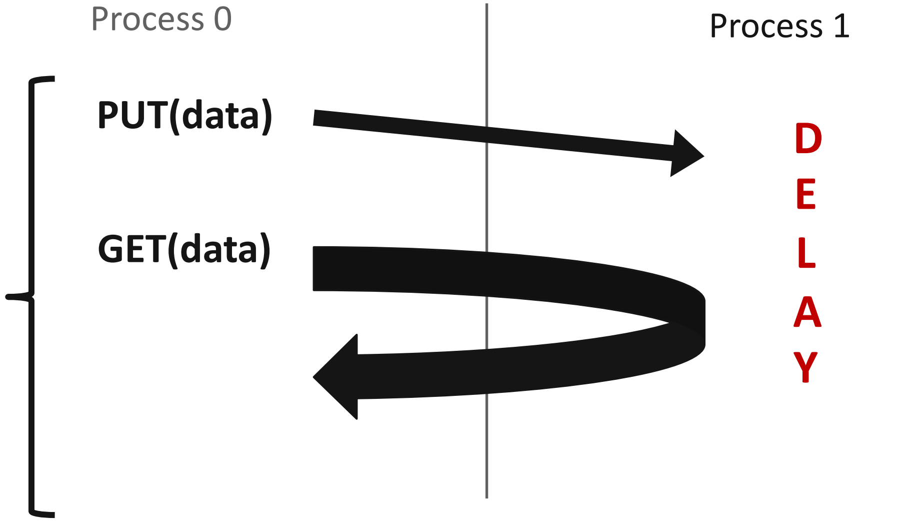
One-sided Communication Example



Comparing One-sided and Two-sided Programming



Even the sending process is delayed



Delay in process 1 does not affect process 0

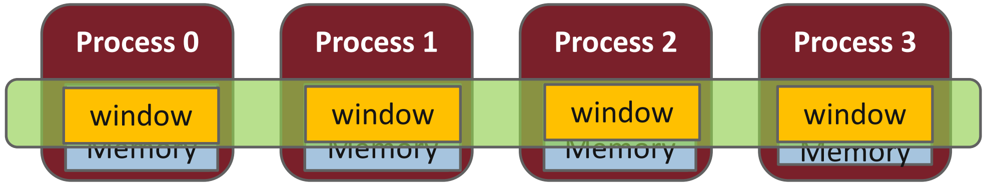
What we need to know in MPI RMA

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

- MPI RMA has a large number of functions, supporting many options
  - We will concentrate on a core that provides most of the power of RMA
  - You can refer to *Using Advanced MPI* or the MPI 4.0 standard for more on RMA

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 via MPI RMA functions

Window creation models

- Four models exist
  - `MPI_WIN_ALLOCATE`
    - You want to create a buffer and directly make it remotely accessible
  - `MPI_WIN_CREATE`
    - You already have an allocated buffer that you would like to make remotely accessible
  - `MPI_WIN_CREATE_DYNAMIC`
    - You don’t have a buffer yet, but will have one in the future
    - You may want to dynamically add/remove buffers to/from the window
  - `MPI_WIN_ALLOCATE_SHARED`
    - You want multiple processes on the same node share a buffer (not covered in this tutorial)

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 */
    a = (int *) malloc(1000*sizeof(int));
    /* use private memory like you normally would */
    for (int i = 0; i < 1000; i++) a[i] = i + 1;

    /* 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);
    free(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 */
    for (int i = 0; i < 1000; i++) a[i] = i + 1;

    /* 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;
}
```

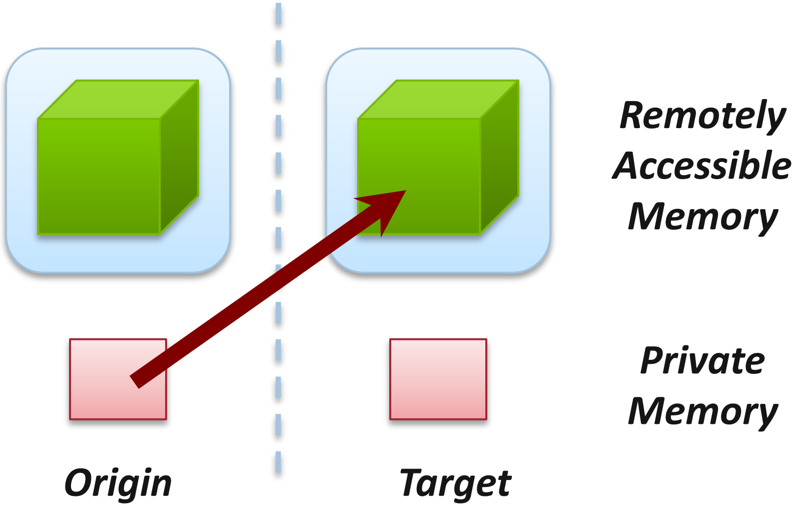
Data movement

- 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)
- There are variations of these as well, include versions with requests.

Data movement: *Put*

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

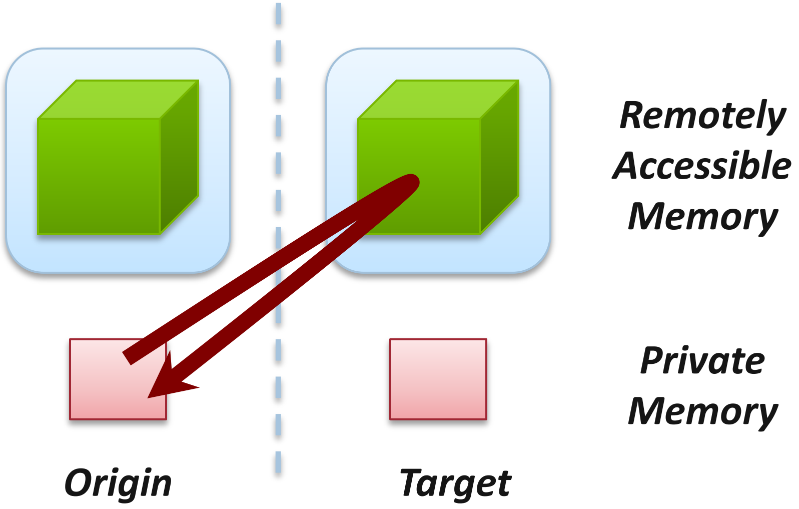
- Move data from origin, to target
- Separate data description triples for **origin** and **target**



Data movement: *Get*

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

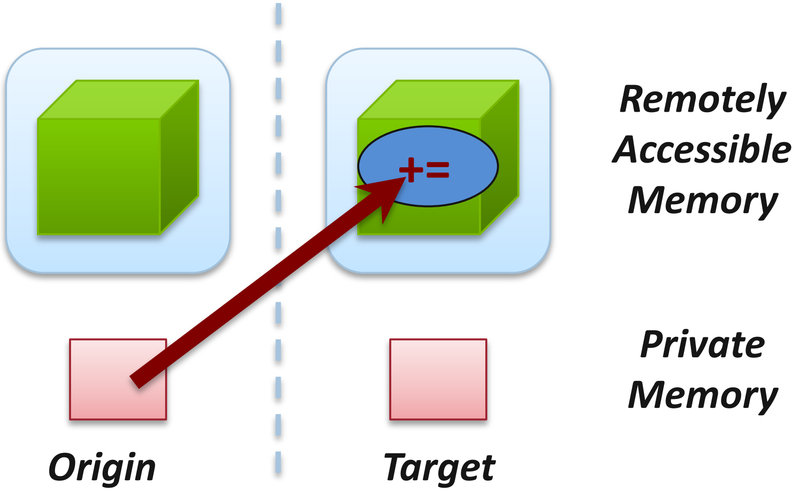
- Move data to origin, from target
- Separate data description triples for **origin** and **target**



Atomic Data Aggregation: *Accumulate*

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

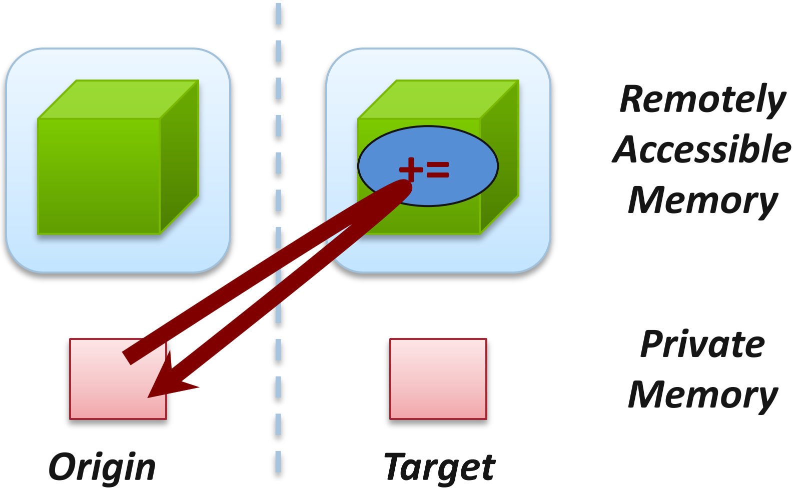
- Atomic update operation, similar to a put
  - Reduces origin and target data into target buffer using op argument as combiner
  - Op = `MPI_SUM`, `MPI_PROD`, `MPI_OR`, `MPI_REPLACE`, `MPI_NO_OP`, …
  - Predefined ops only, no user-defined operations
- Different data layouts between target/origin OK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



Atomic Data Aggregation: *Get Accumulate*

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

- Atomic read-modify-write
  - Op = **MPI_SUM**, **MPI_PROD**, **MPI_OR**, **MPI_REPLACE**, **MPI_NO_OP**, …
  - Predefined ops only
- Result stored in target buffer
- Original data stored in result buf
- Different data layouts between target/origin OK
  - Basic type elements must match
- Atomic get with **MPI_NO_OP**
- Atomic swap with **MPI_REPLACE**



Atomic Data Aggregation: *FOP and CAS*

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

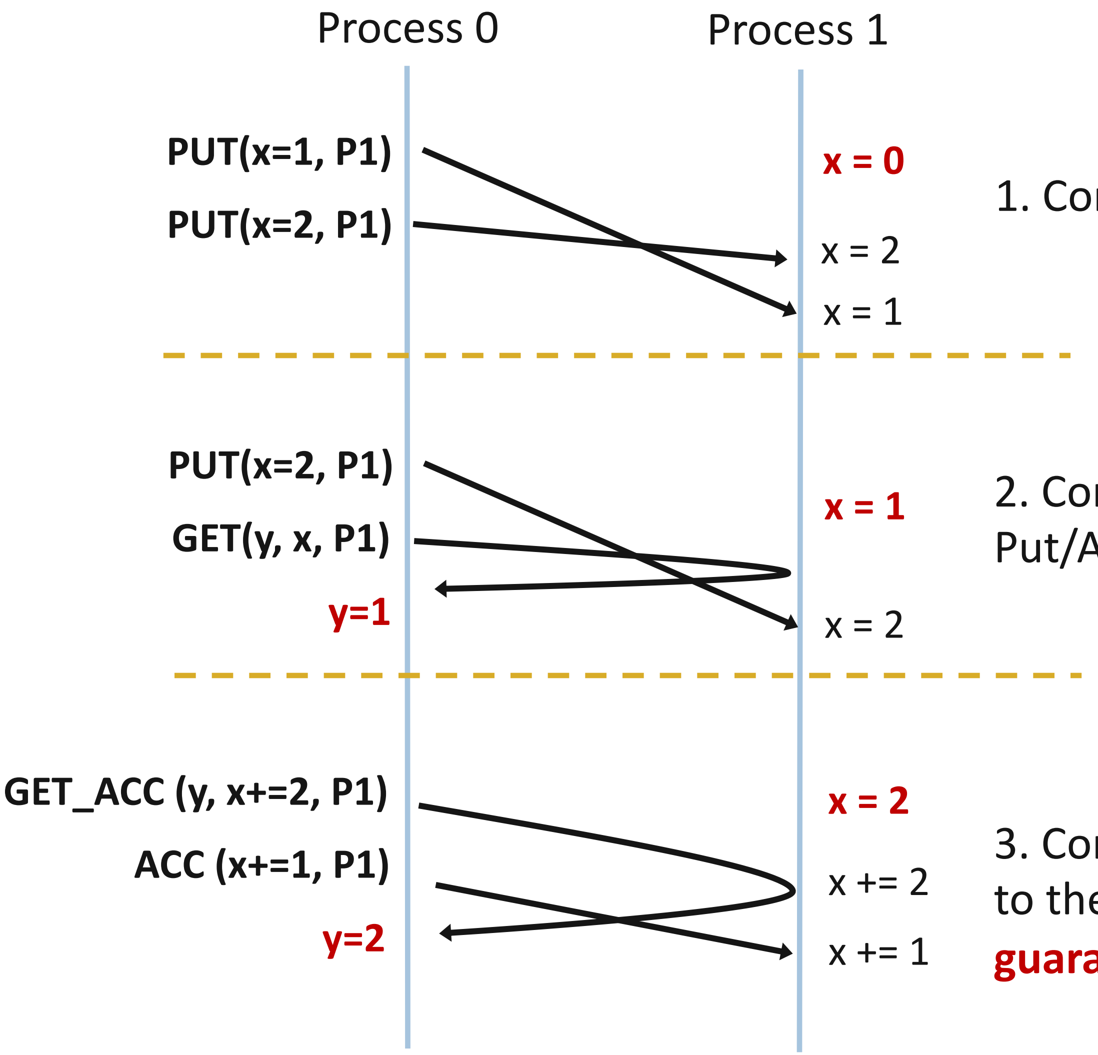
```
MPI_Compare_and_swap(const void *origin_addr, const void *compare_addr,
                     void *result_addr, MPI_Datatype dtype, int target_rank,
                     MPI_Aint target_disp, MPI_Win win)
```

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

Ordering of Operations in MPI RMA

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

Examples with operation ordering



1. Concurrent Puts: **undefined**

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

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

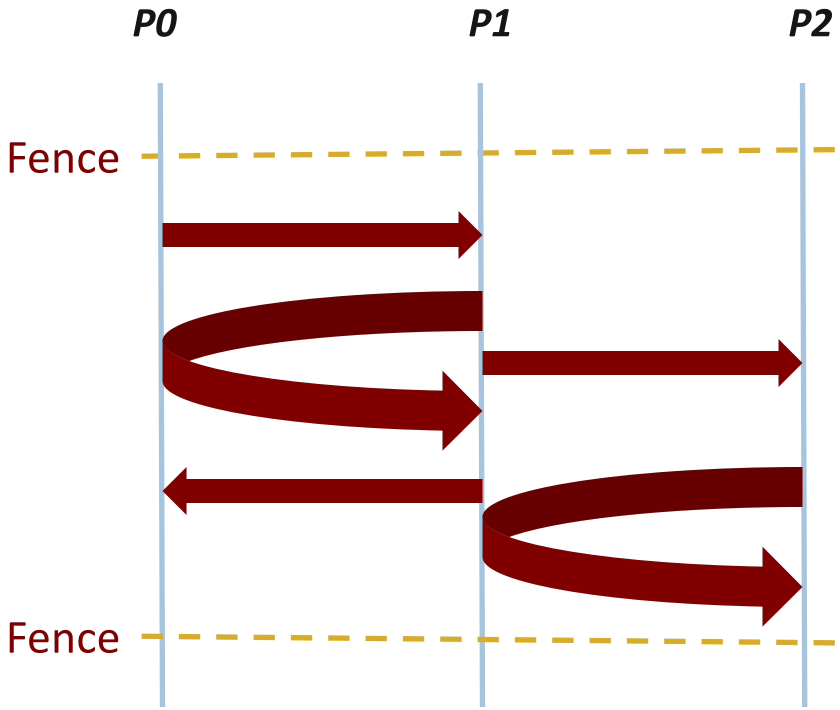
RMA Synchronization Models

- RMA data access model
  - When is a process allowed to read/write remotely accessible memory?
  - When is data written by process X is available for process Y to read?
  - RMA synchronization models define these semantics
- Three synchronization models provided by MPI:
  - Fence (active target)
  - Post-start-complete-wait (generalized active target)
  - Lock/Unlock (passive target) <- preferred for all RMA since MPI 3.0
- 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

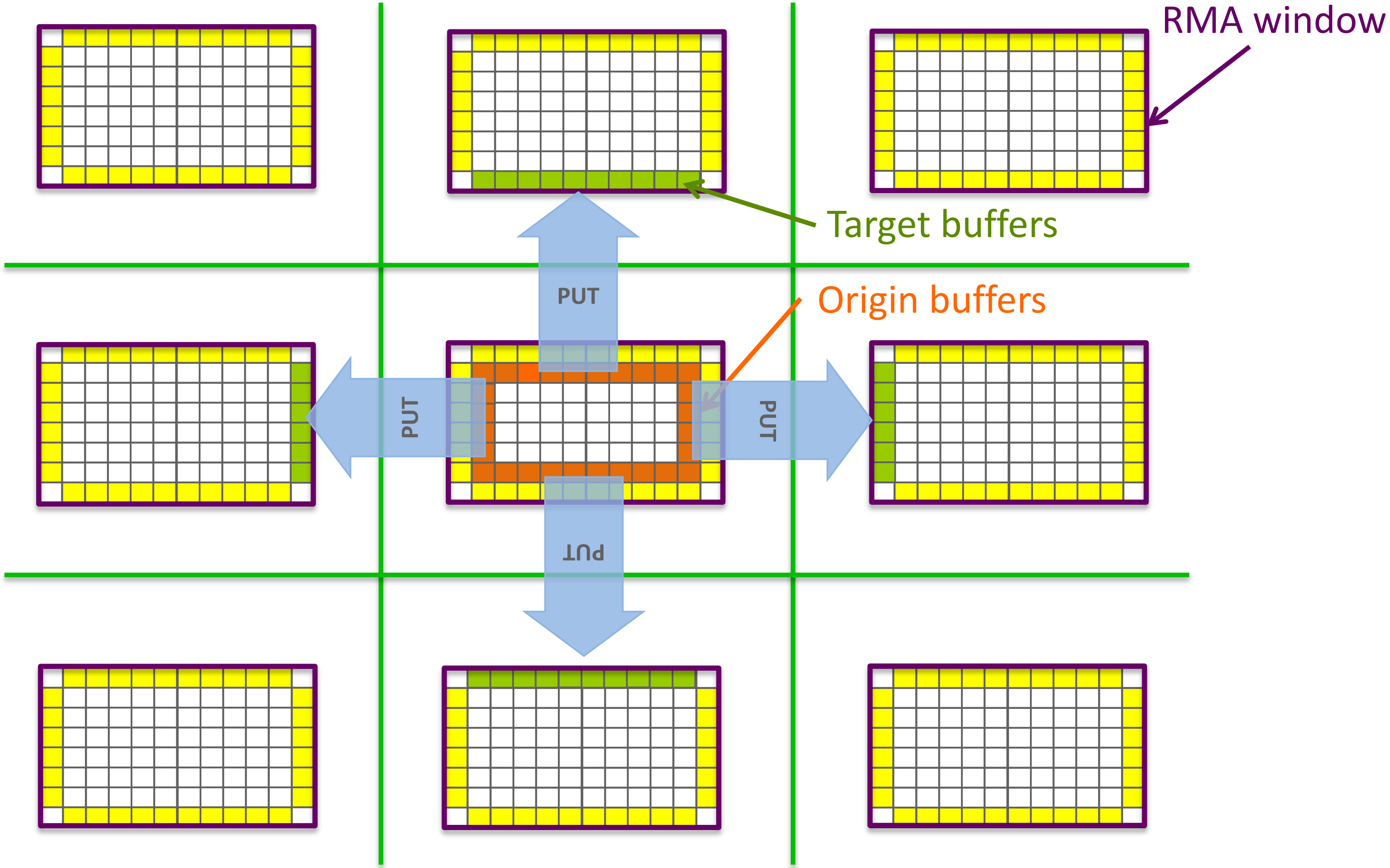
Fence: Active Target Synchronization

```
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



Exercise 1: Stencil with RMA Fence (1/2)



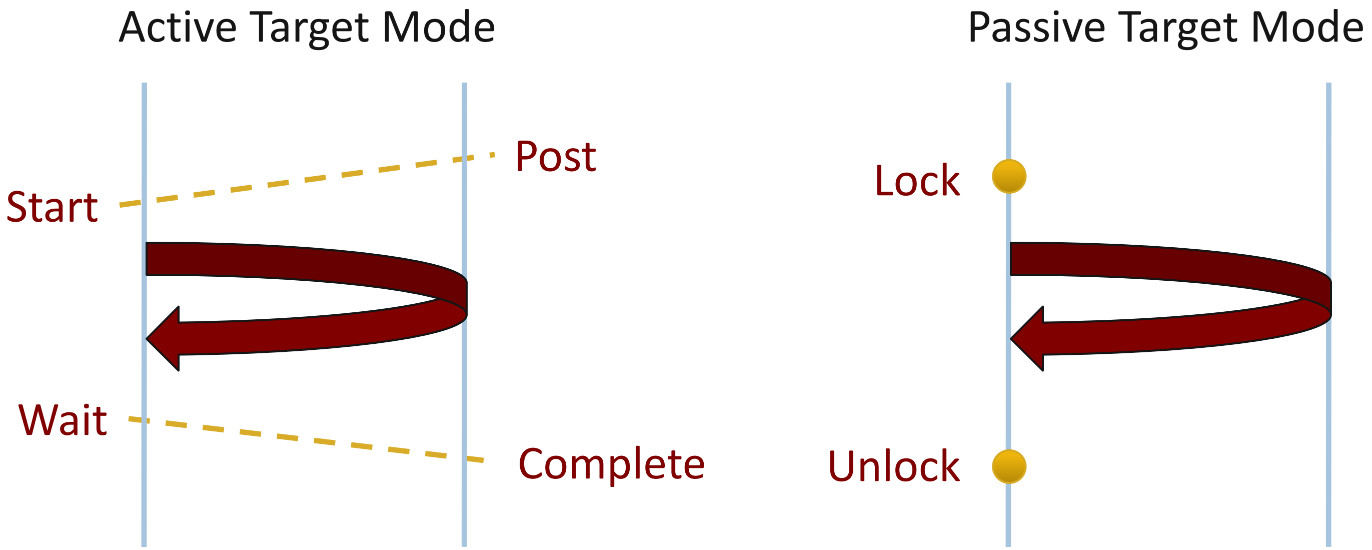
Exercise 1: Stencil with RMA Fence (2/2)

- In the derived datatype version of the stencil code
  - Used nonblocking communication
  - Used derived datatypes
- Let’s try to use RMA fence
  - Move data with PUT instead of send/recv
- *Start from derived_datatype/stencil.c*
- *Solution available in rma/stencil_fence_put.c*

Exercise 2: Stencil with RMA Fence (GET model)

- In the derived datatype version of the stencil code
  - Used nonblocking communication
  - Used derived datatypes
- Let’s try to use RMA fence
  - Move data with GET instead of send/recv
- *Start from rma/stencil_fence_put.c*
- *Solution available in rma/stencil_fence_get.c*

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

Passive Target Synchronization

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

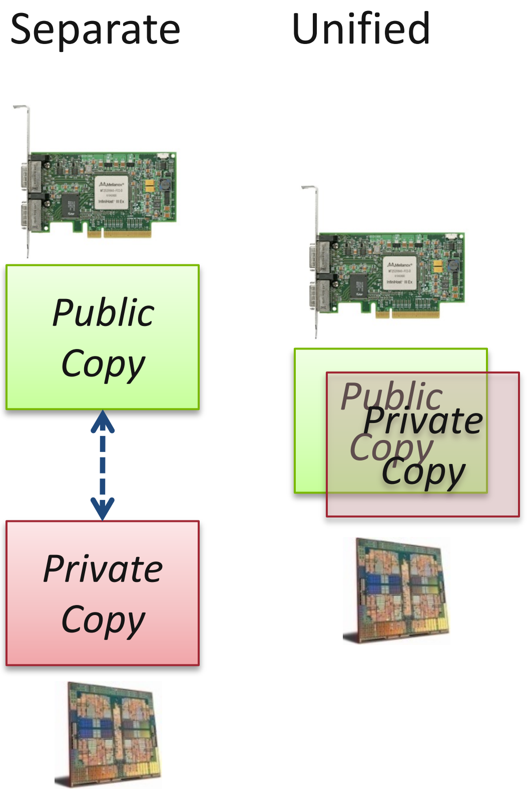
```
MPI_Win_unlock_all(MPI_Win win)
```

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

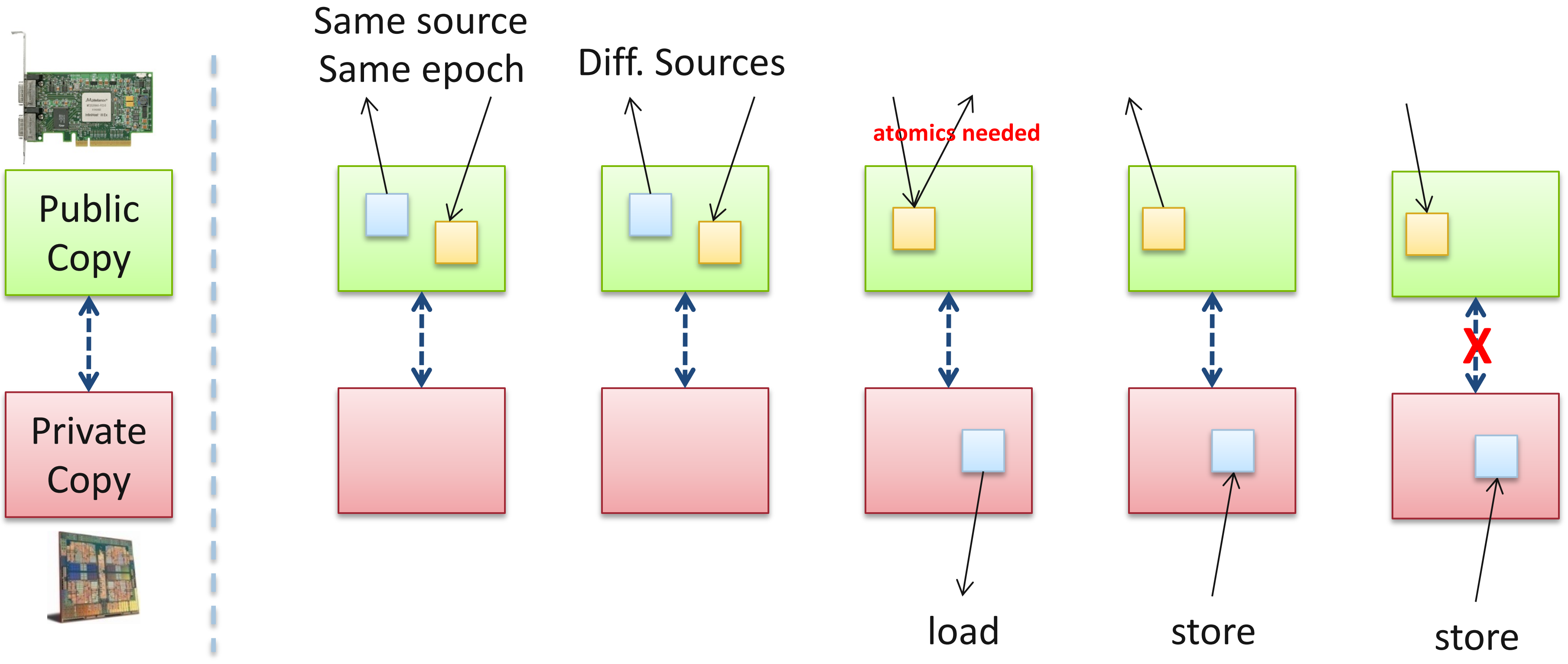
- **`Lock_all`**: Shared lock, passive target epoch to all other processes
  - Expected usage is long-lived: **`lock_all`**, **`put/get`**, **`flush`**, …, **`unlock_all`**
- **`Flush_all`** – remotely complete RMA operations to all processes
- **`Flush_local_all`** – locally complete RMA operations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

MPI RMA Memory Model

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

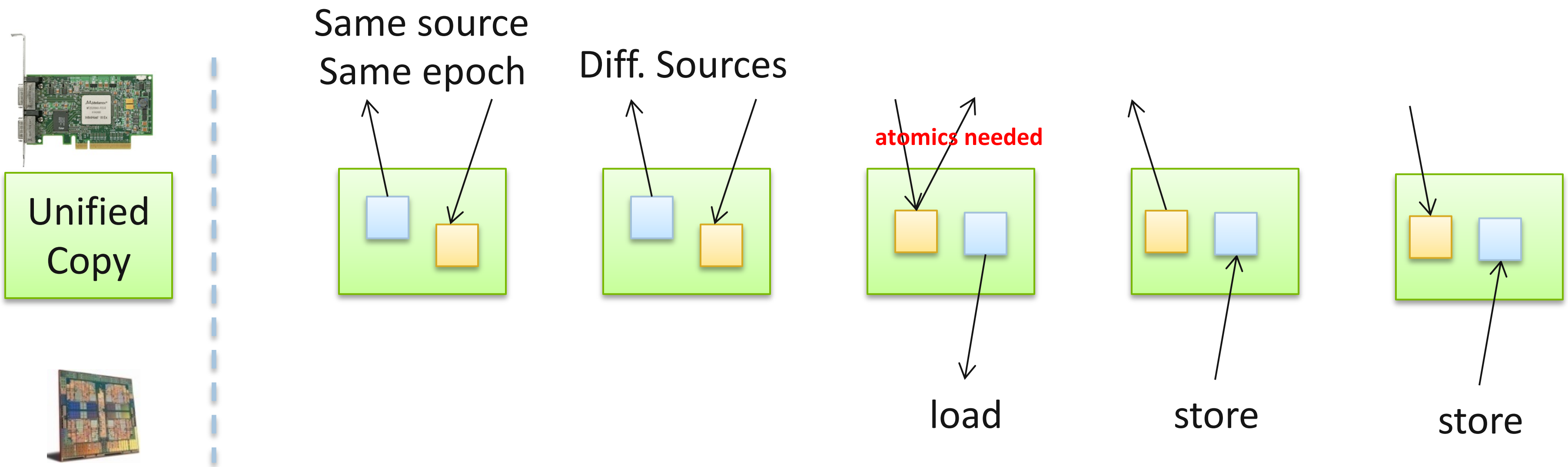


MPI RMA Memory Model (separate windows)



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

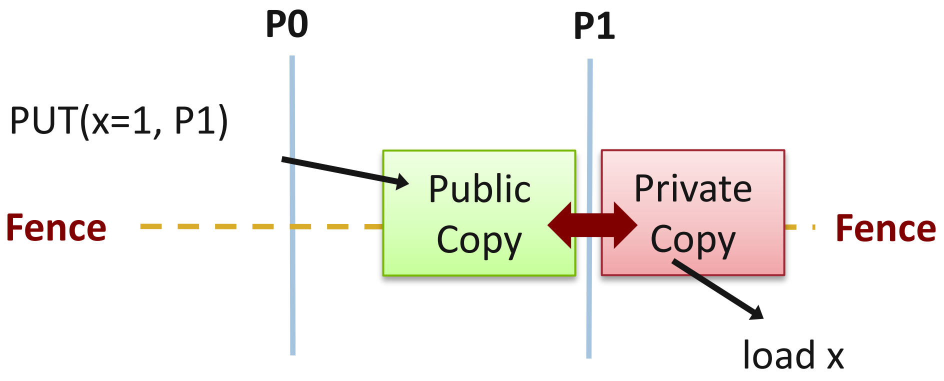
MPI RMA Memory Model (unified windows)



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

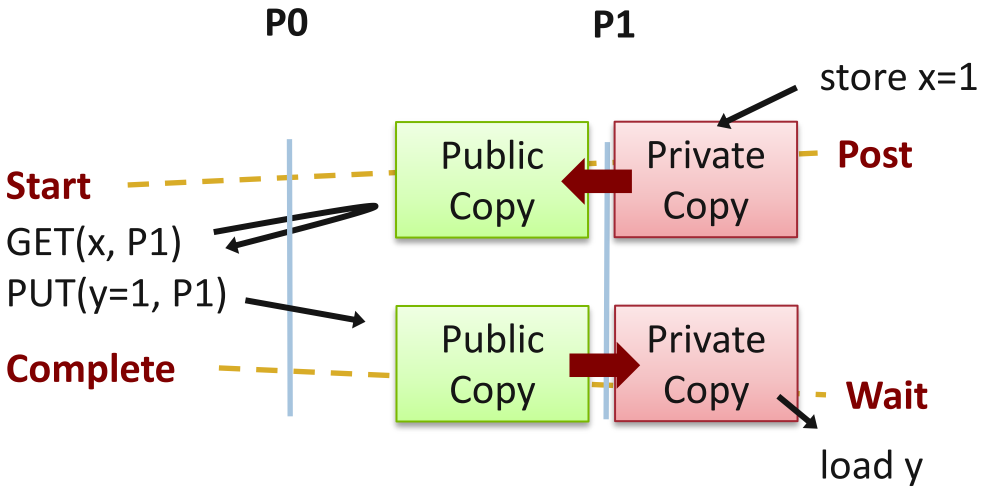
Synchronizing Local and RMA Access (1/2)

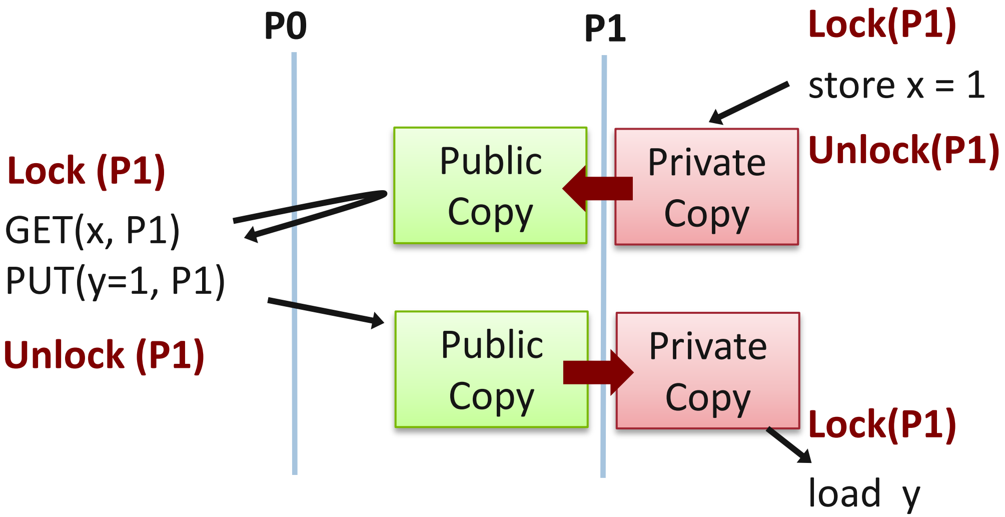
- RMA operations access the public copy of window
- Local load/store update the private copy
  - Including using as MPI send/receive buffers
- Ensure memory synchronization for portability
- Implicit memory synchronization (i.e., memory barrier) in RMA synchronization calls
  - **Fence:** Synchronize private and public copies of local window



Synchronizing Local and RMA Access (2/2)

- PSCW active target epoch
  - **Post**: Updates in private copy becomes visible in public copy
  - **Wait**: Updates in public copy becomes visible in private copy
- Passive target epoch
  - **Lock/Lock_all**: Updates in public copy becomes visible in private
  - **Unlock/Unlock_all**: Updates in private copy becomes visible in public

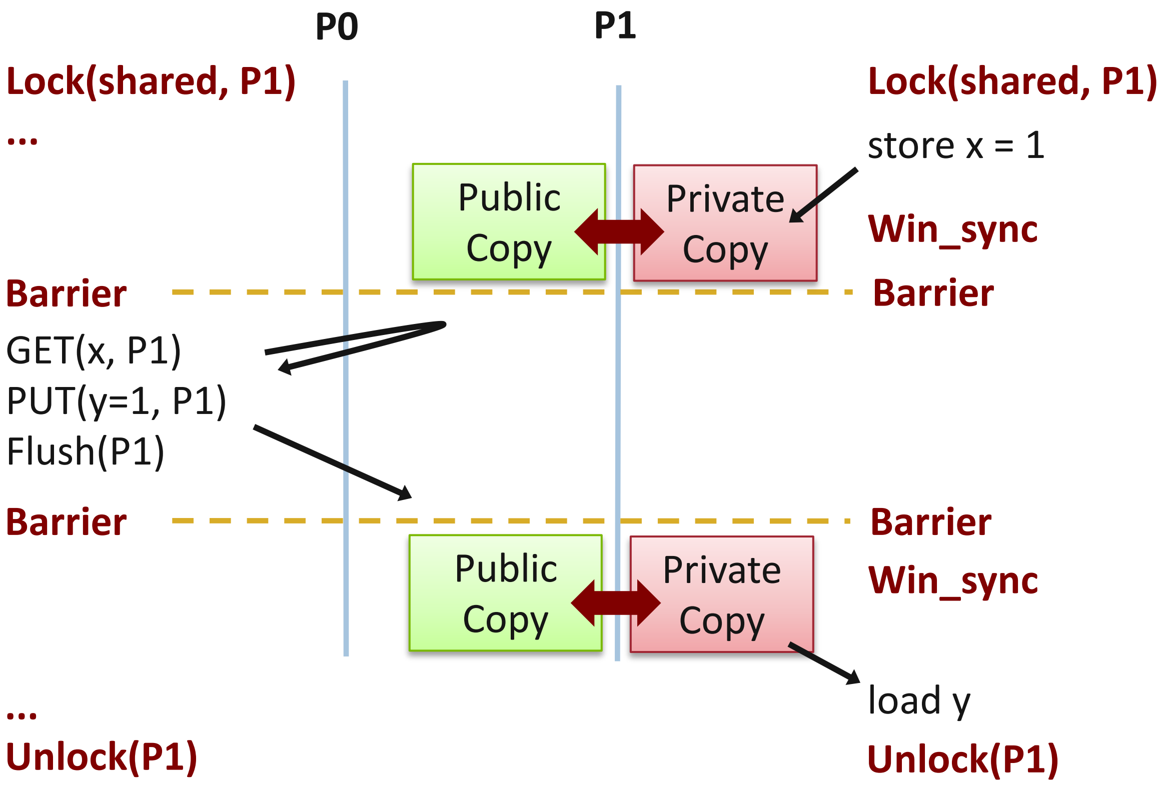




Window synchronization: MPI_WIN_SYNC

```
MPI_Win_sync(MPI_Win win)
```

- Synchronizes the public and private copies of local window in passive target epoch

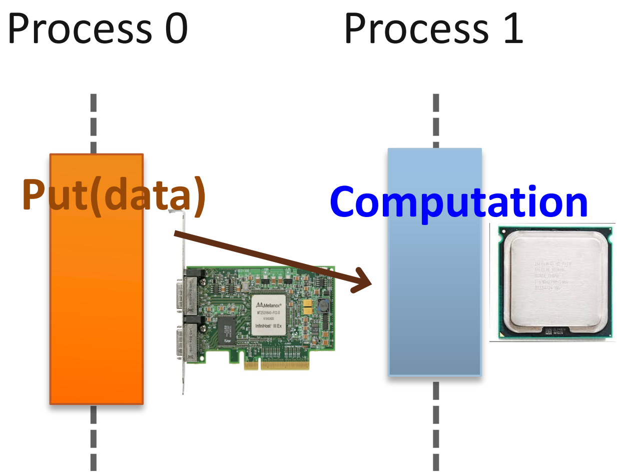


Exercise 3: Stencil with RMA Lock_all/Unlock_all (PUT model)

- In the fence versions of the stencil code, RMA synchronization involves the target processes
- Let’s try to use RMA Lock_all/Flush_all/Unlock_all
  - Only the origin processes call RMA synchronization
  - Still need **Barrier** for process synchronization (e.g., ensure neighbors have completed data update to my local window)
  - Need **Win_sync** for memory synchronization
- *Start from rma/stencil_fence_put.c*
- *Solution available in rma/stencil_lock_put.c*

Hardware-Offloaded Communication

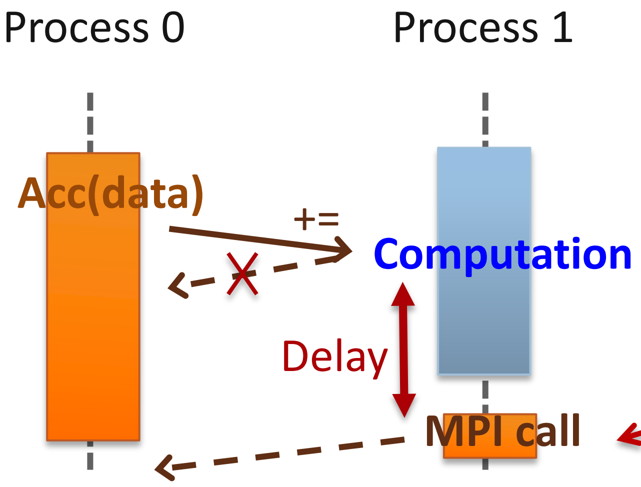
- Offloading data transfer to hardware is ideal for performance
  - Two-sided (e.g., SEND/RECEIVE):
    - Require **complex message matching** (rank + tag + comm), especially for wildcard receives (MPI_ANY_TAG|ANY_SOURCE)
    - Supported HW: Mellanox ConnectX-5 (support HW tag matching)
  - One-sided (e.g., PUT/GET/ACCUMULATE):
    - **No matching** requirement, easier for hardware offloading
    - Natively supported on various RDMA networks such as Mellanox InfiniBand, Cray Aries, and Fujitsu Tofu



Hardware-offloaded data transfer can be fully overlapped with computation

Asynchronous Execution of MPI RMA

- Asynchronous execution of RMA depends on the MPI implementation, which in turn depends on what the network hardware provides
- Most common situation on current network hardware:
  - Some operations are natively supported in hardware (e.g., contiguous PUT/GET)
  - Other operations need to be **emulated in software**



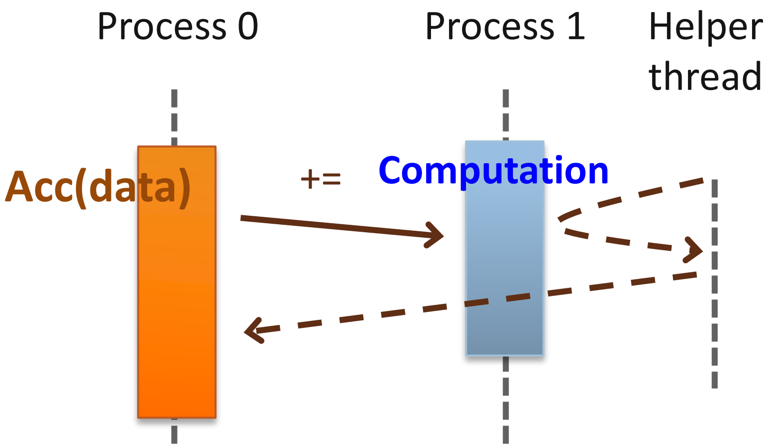
Software-handled MPI Accumulate

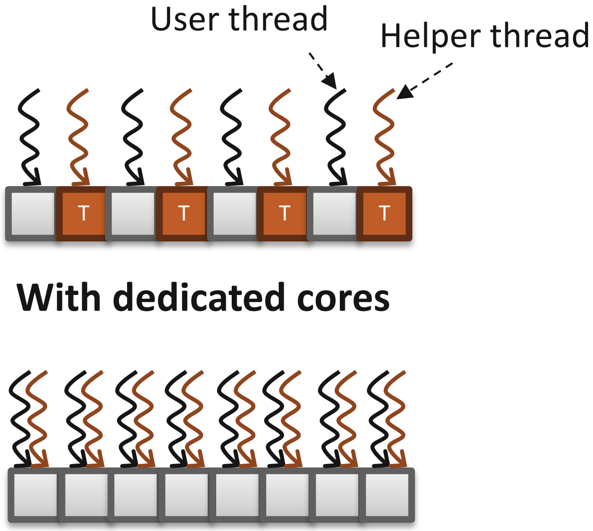
Software implementation of one-sided operations means that **the target process has to run code to complete the MPI operation**

The common case for many MPI implementations is that this is done within some MPI routine

Possible Solution 1: Thread-based Progress (1/2)

- Every MPI process creates a **dedicated helper thread** at MPI_Init_thread
  - Supported by most MPI implementations, but unlikely to be default
  - Might need to turn on some environment variable (check the documentation)
- The thread polls MPI progress for the process while the process is computing
- May dedicate some number of computing cores per process for this helper thread
- Multithreading safety overhead (i.e., MPI internal lock contention between threads, memory barriers)





With dedicated cores

With oversubscribed cores

Possible Solution 1: Thread-based Progress (2/2)

- Available in many mainstream MPI implementations
- Core binding is important! See vendor documentation

MPICH:

```
% export MPIR_CVAR_ASYNC_PROGRESS=1
```

Intel MPI:

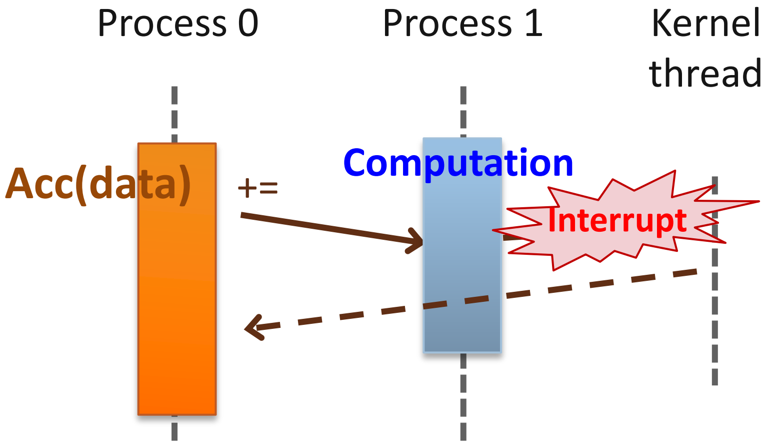
```
% export I_MPI_ASYNC_PROGRESS=1
```

Cray MPI:

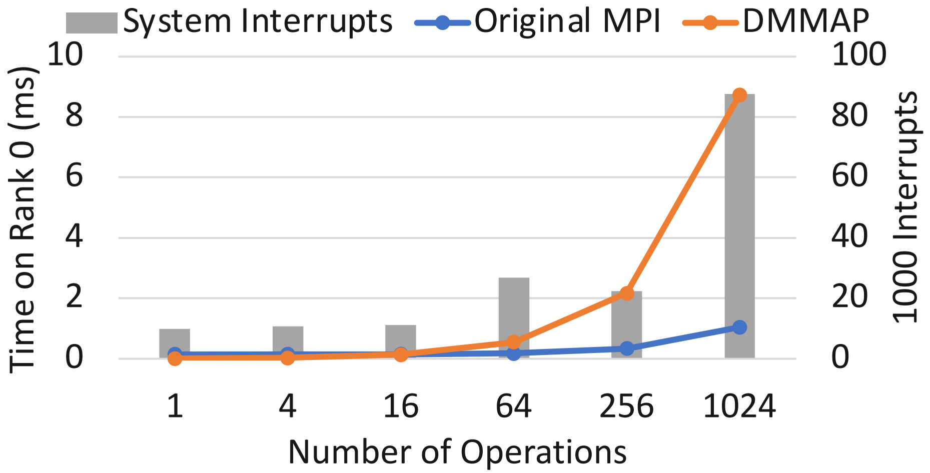
```
% export MPICH_NEMESIS_ASYNC_PROGRESS=1
% export MPICH_MAX_THREAD_SAFETY=multiple
```

Possible Solution 2: Interrupt-based Progress

- Utilize **hardware interrupts** to awaken a kernel thread when new message arrives
- Examples: Cray MPI DMAPP mode (deprecated from v7.6.0), IBM MPI on Blue Gene/P
- Overhead of frequent interrupts, need special hardware support
  - Not a common model for most current networks

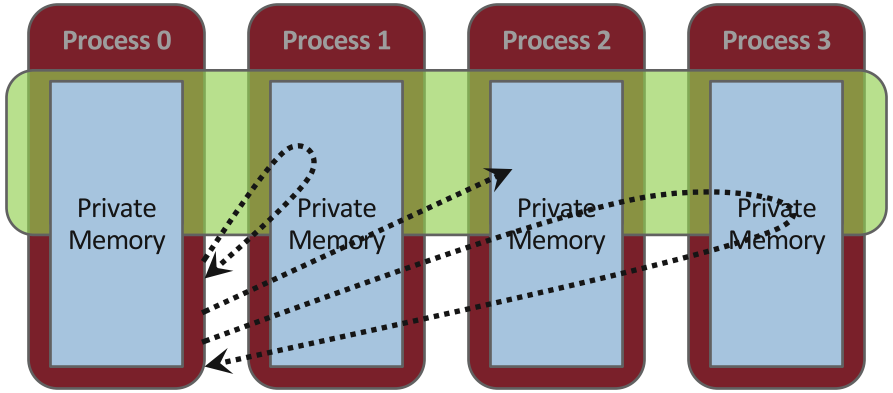


Interrupt-based asynchronous progress with CrayMPI7.2.1 on Cray XC30



Section Summary

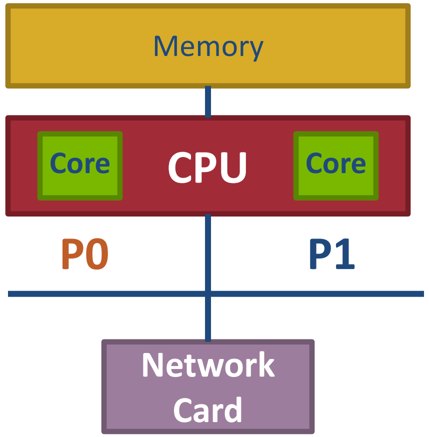
- MPI one-sided communication is associated with **windows**
- **Operations** include basic **PUT**, **GET**, and **Atomic** operations
- **Synchronization** modes
  - Active-target (similar to two-sided) : FENCE, PSCW
  - Passive-target: LOCK-UNLOCK, FLUSH, FLUSH_LOCAL…
- Enable **asynchronous progress** for performance



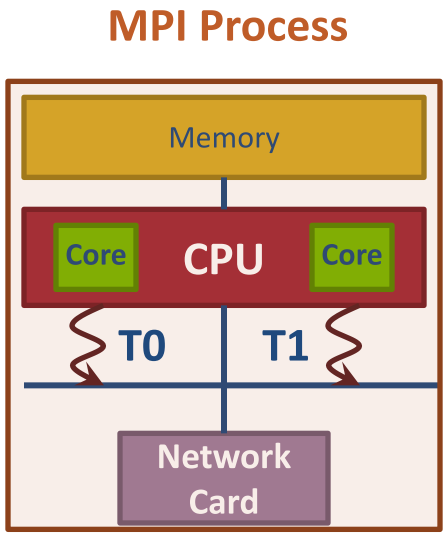
MPI Hybrid Programming: Threads

Hybrid MPI + X : Most Popular Forms

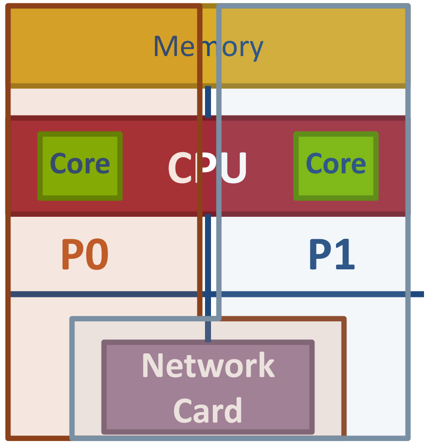
MPI + X



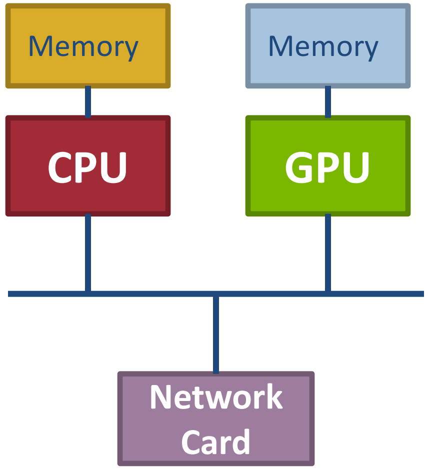
Flat MPI



MPI + Threads



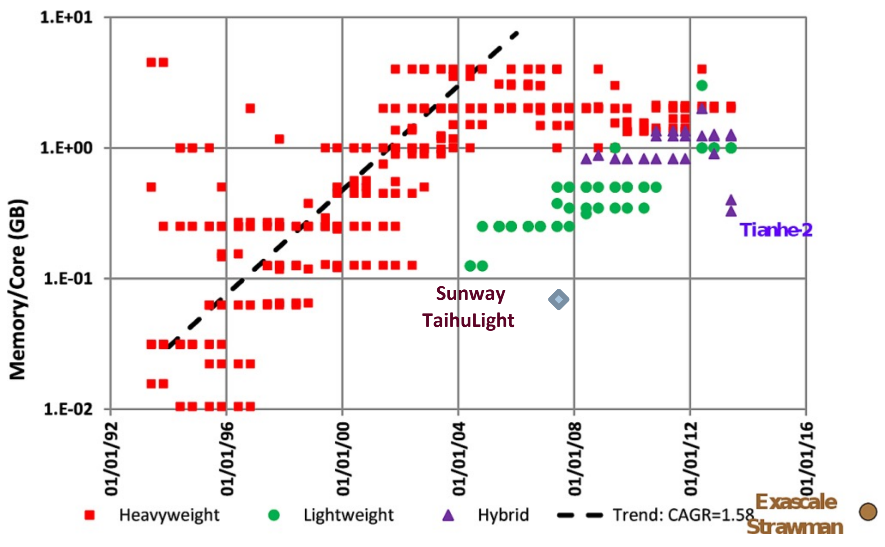
MPI + Shared Memory



MPI + ACC

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

- MPI's distributed process model lacks facility for thread and GPU programming
  - Significant effort needed for a pure MPI solution
- Strong scaling applications is increasing in importance
  - 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



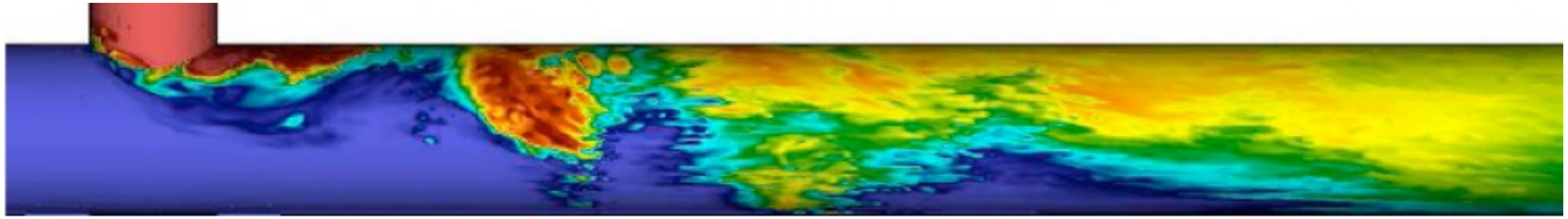
Evolution of the memory capacity per core in the Top500 list (Peter Kogge. PIM & memory: The need for a revolution in architecture.)

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

- MPI+X benefits (X= {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

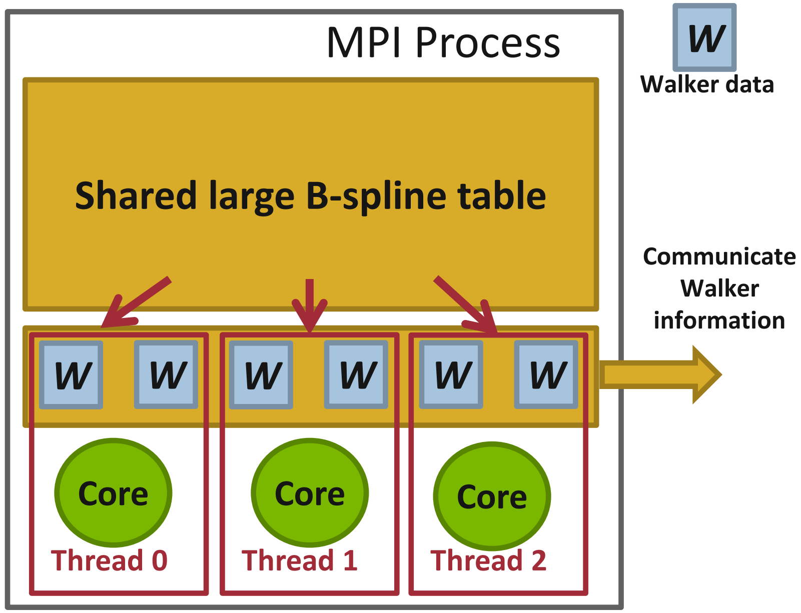
Nek5000



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

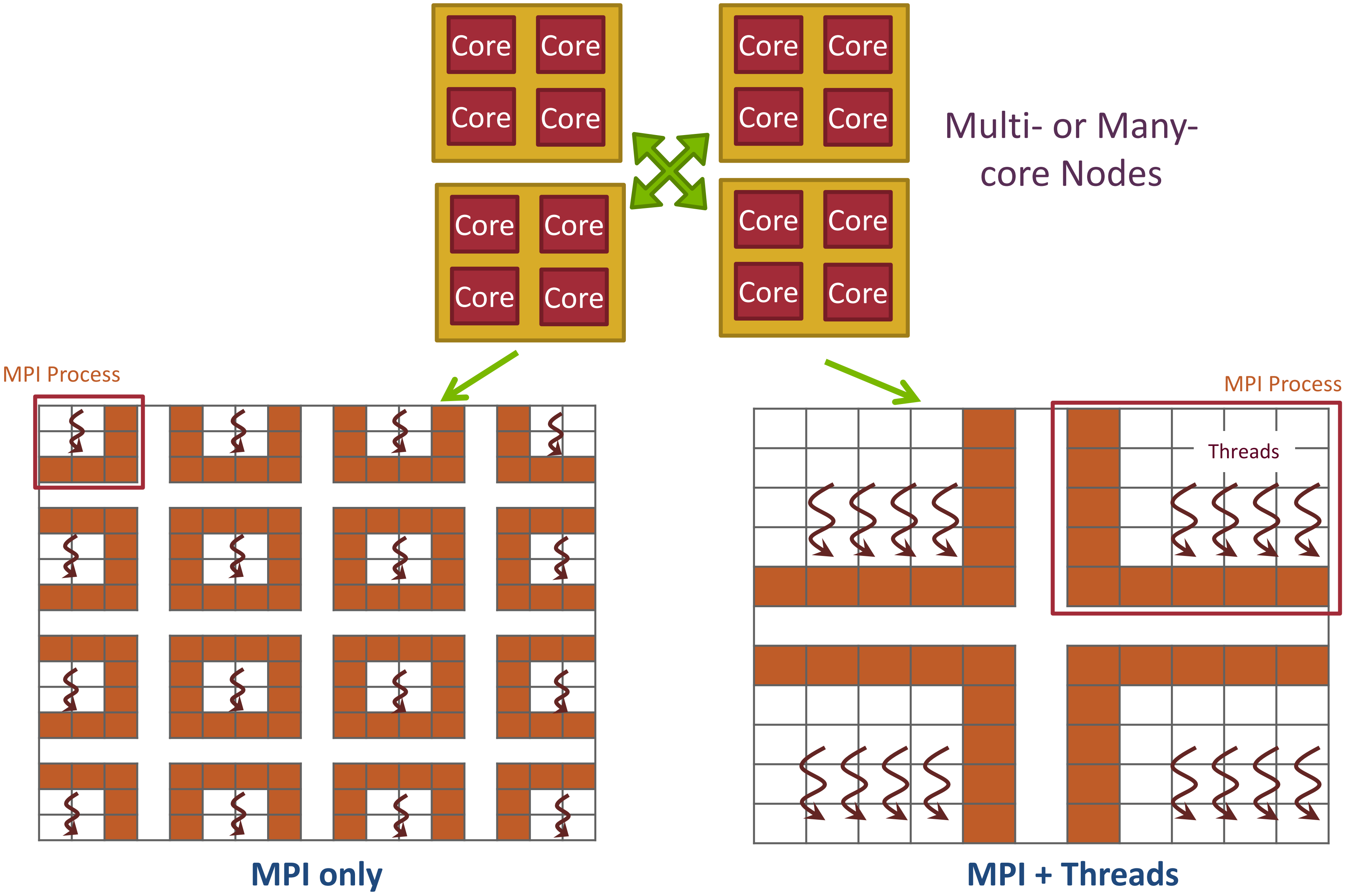
- Example 2: Quantum Monte Carlo Simulation (QMCPACK)
  - Size of the physical system to simulate is bound by memory capacity [1]
  - Memory space dominated by large interpolation tables (typically several Giga Bytes of storage)
  - Threads are used to share those tables
  - Memory for communication buffers must be kept low to be allow simulation of larger and highly detailed simulations.

QMCPACK



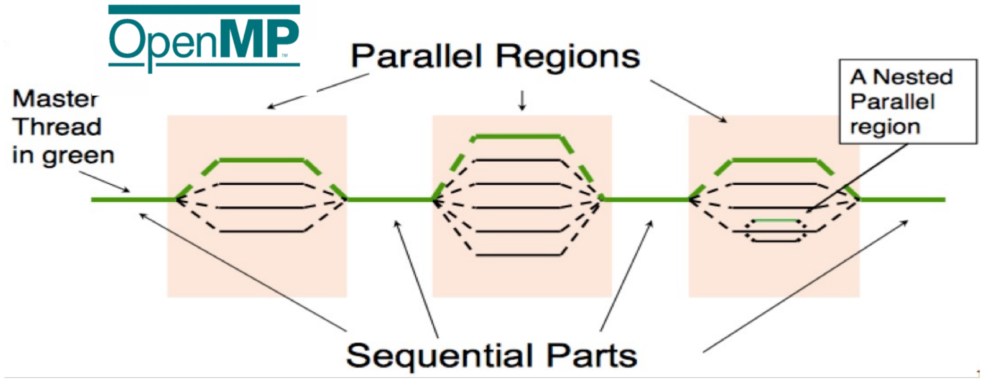
[1] Kim, Jeongnim, et al. "Hybrid algorithms in quantum Monte Carlo." Journal of Physics, 2012.

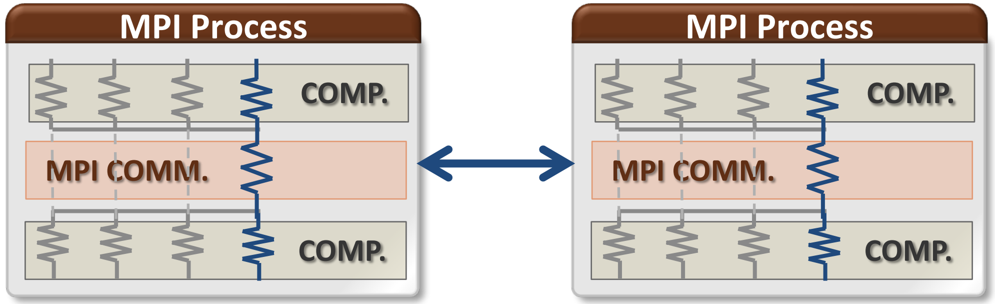
MPI + Threads: How To? (1/3)



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

- OpenMP and Pthreads are common models
  - OpenMP creates thread parallel regions implicitly. Threads are created and managed by the compiler, based on user directives.
  - Pthreads creates and manages threads explicitly, provides more complex and dynamic approaches.





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

MPI + Threads

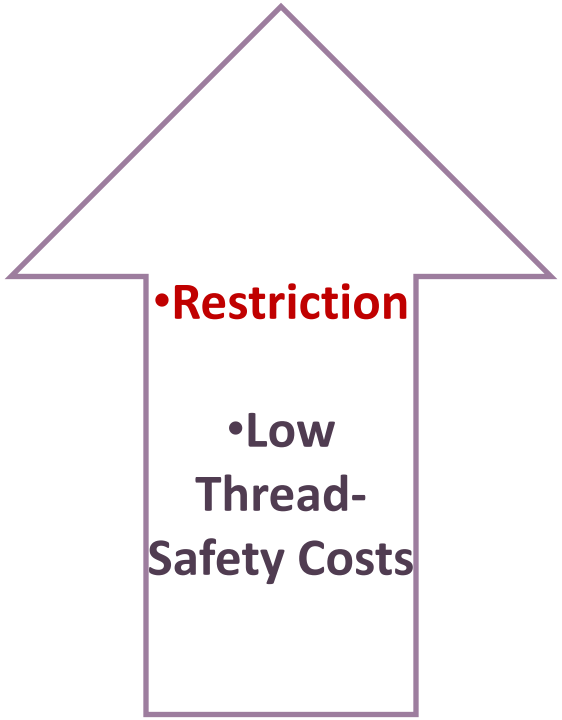
↓

Interoperability

Data Safety In MPI

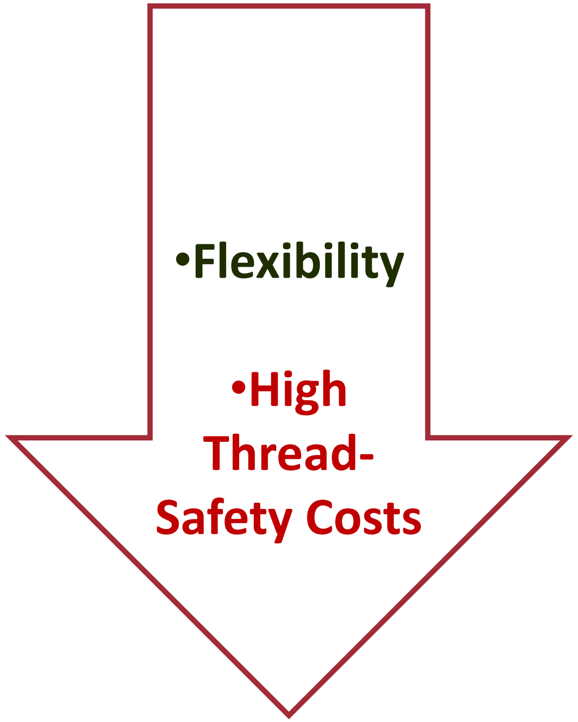
Interoperation or thread levels:

- 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

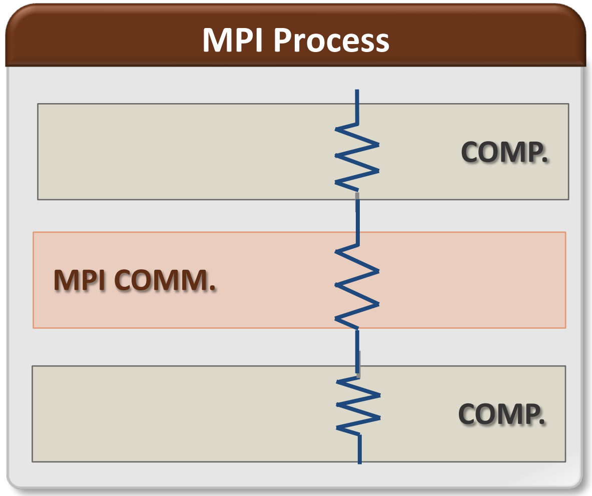
MPI's Four Levels of Thread Safety

- MPI defines four levels of thread safety
  - These are *commitments* the application makes to the MPI
  - MPI use them as *hints* for performance optimization
- Thread levels are in increasing order
  - If an application works in FUNNELED mode, it can work in SERIALIZED
  - If an MPI library provide MULTIPLE mode, it will work for application that require SERIALIZED
- MPI defines an alternative to MPI_Init
  - **MPI_Init_thread**(requested, 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;
}
```

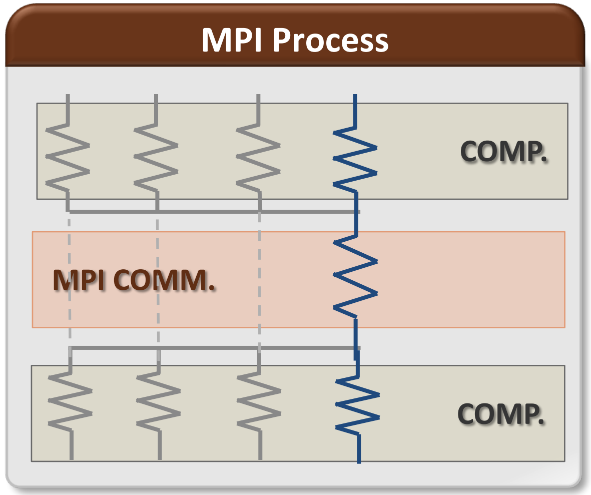


MPI_THREAD_FUNNELED

- 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]);
}
```

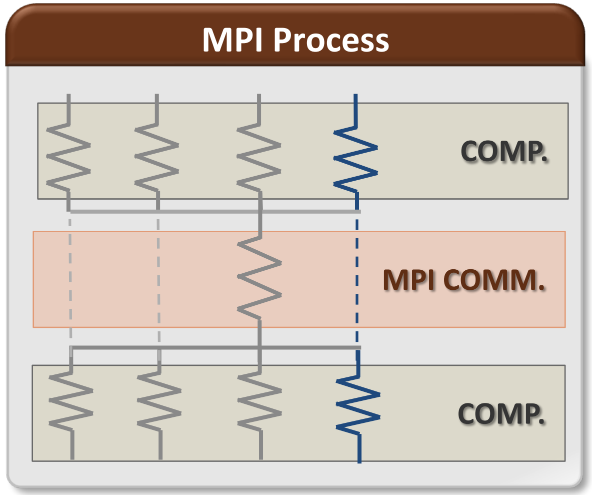


MPI_THREAD_SERIALIZED

- 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);
}
```

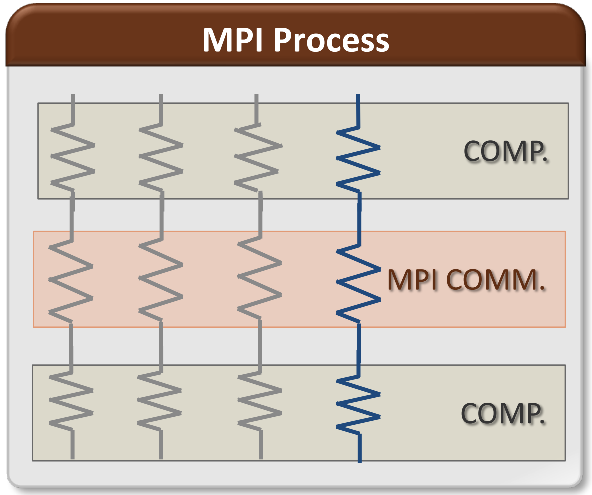


MPI_THREAD_MULTIPLE

- **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_MULTIPLE)
       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]);

   /* Do MPI stuff */
}
```



Important Notes

- MPI does not require an implementation to support levels higher than MPI_THREAD_SINGLE
  - *Always check the provided level!*
- An implementation may provide both thread-single verson and thread-safe version
  - Avoids branches and locking/atomics overhead
- A program that calls MPI_Init (instead of MPI_Init_thread) should assume that only MPI_THREAD_SINGLE is supported
  - MPI Standard *mandates* MPI_THREAD_SINGLE for MPI_Init
  - *A threaded MPI program that does not call MPI_Init_thread is an incorrect program (common user error we see)*

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

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
  - Send/Recv can be used for synchronization
  - 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

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

Ordering in MPI_THREAD_MULTIPLE: Incorrect Example with Collectives

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

- 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

Process 0

Thread 1	Thread 2
	MPI_Comm_free(comm)
MPI_Bcast(comm)	

- 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

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

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

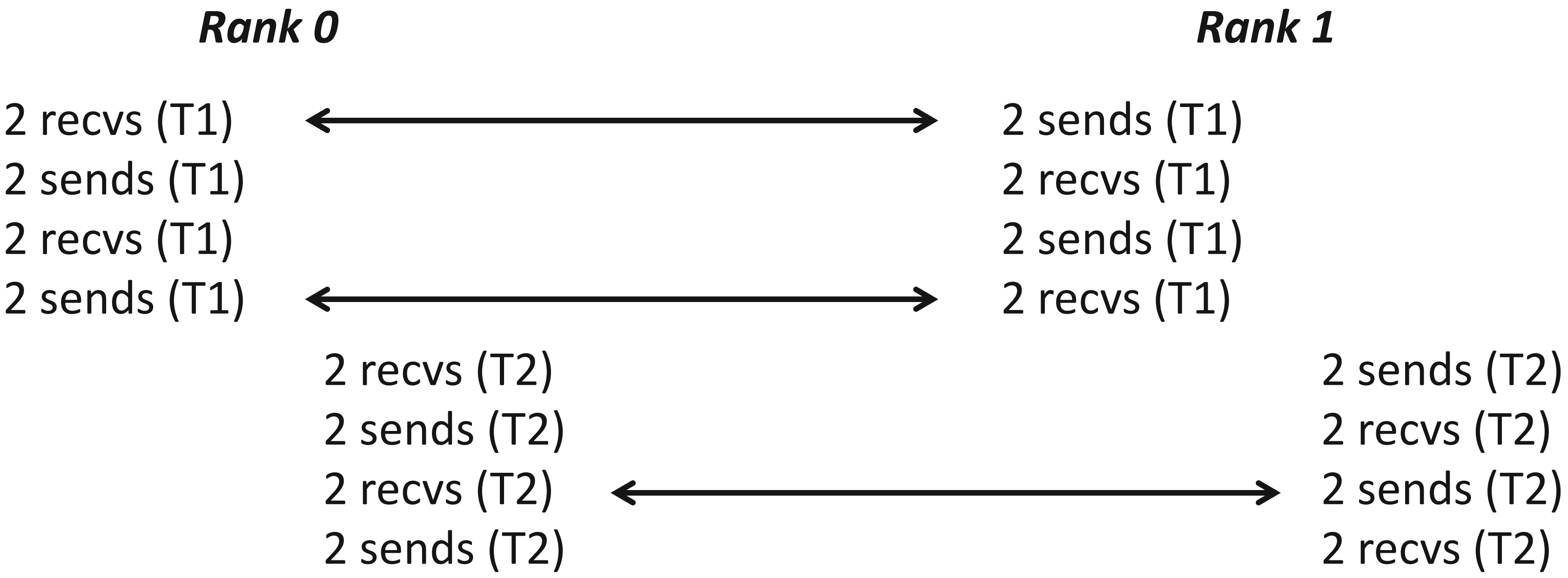
An Example we encountered

- 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 Processes, 2 Threads (each thread executes this code)

```
if (rank == 1) {
   MPI_Send(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
   MPI_Send(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
   MPI_Recv(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &stat);
   MPI_Recv(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &stat);

   MPI_Send(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
   MPI_Send(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
   MPI_Recv(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &stat);
   MPI_Recv(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &stat);
} else {  /* rank == 0 */
   MPI_Recv(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD, &stat);
   MPI_Recv(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD, &stat);
   MPI_Send(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
   MPI_Send(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD);

   MPI_Recv(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD, &stat);
   MPI_Recv(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD, &stat);
   MPI_Send(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
   MPI_Send(NULL, 0, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
}
```

Intended Ordering of Operations



Rank 0

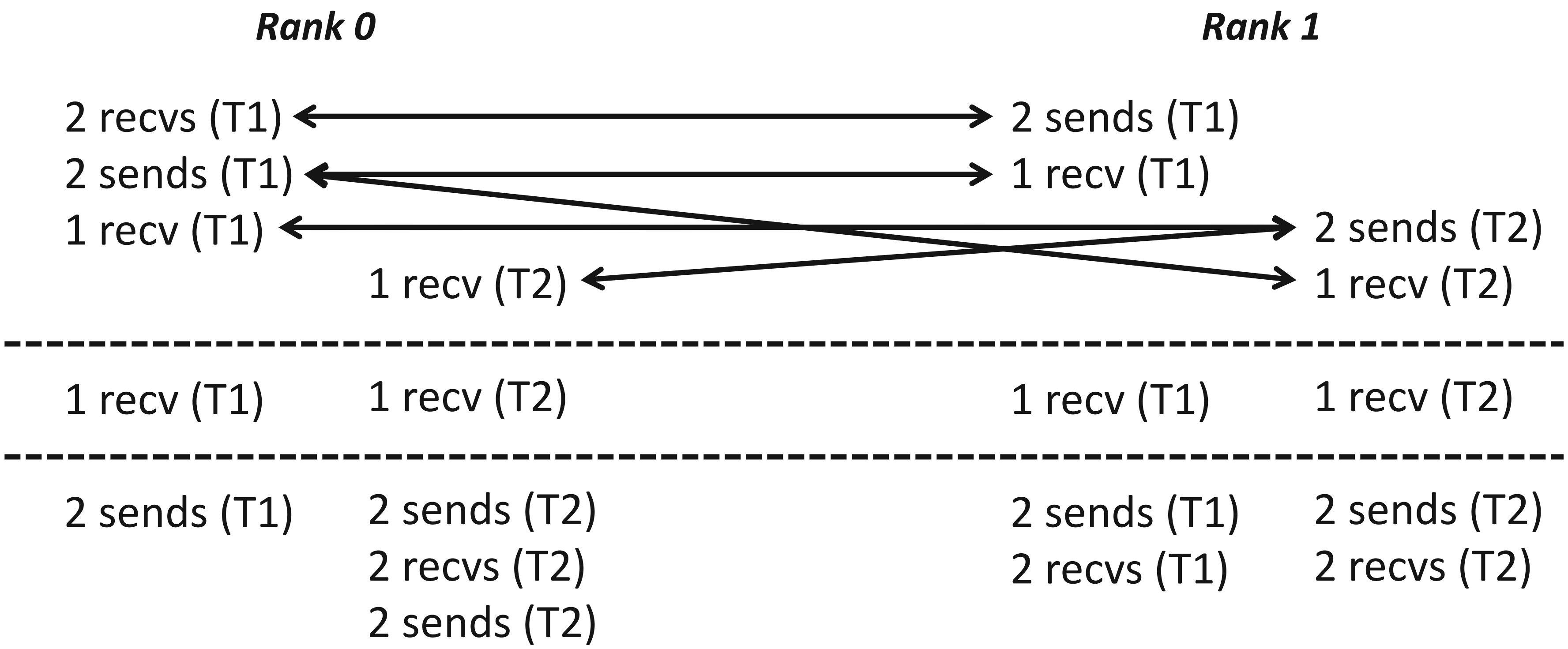
2 recvs (T1)
2 sends (T1)
2 recvs (T1)
2 sends (T1)
2 recvs (T2)
2 sends (T2)
2 recvs (T2)
2 sends (T2)

Rank 1

2 sends (T1)
2 recvs (T1)
2 sends (T1)
2 recvs (T1)
2 sends (T2)
2 recvs (T2)
2 sends (T2)
2 recvs (T2)

- Every send matches a receive on the other rank

Possible Ordering of Operations in Practice



Rank 0

2 recvs (T1)
2 sends (T1)
1 recv (T1)
1 recv (T2)

1 recv (T1)
1 recv (T2)

2 sends (T1)
2 sends (T2)
2 recvs (T2)
2 sends (T2)

Rank 1

2 sends (T1)
1 recv (T1)
2 sends (T2)
1 recv (T2)

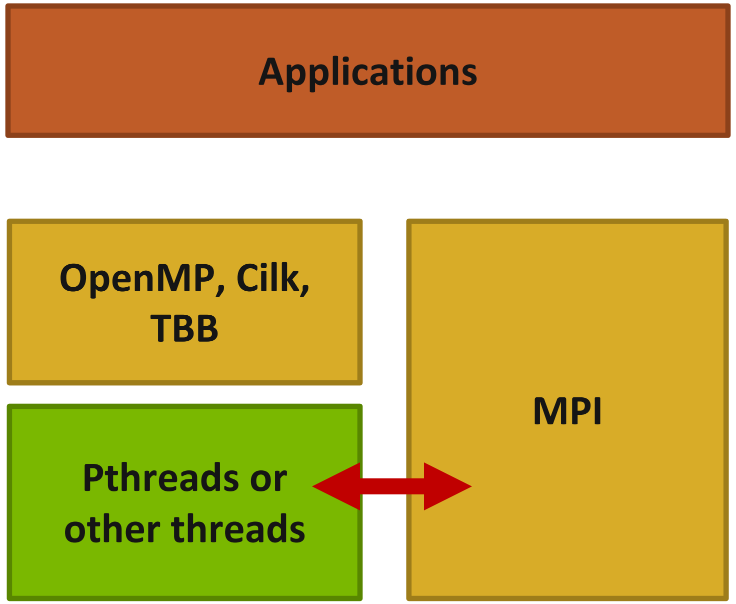
1 recv (T1)
1 recv (T2)

2 sends (T1)
2 recvs (T1)
2 sends (T2)
2 recvs (T2)

- Because the MPI operations can be issued in an arbitrary order across threads, all threads could block in a RECV call

MPI+OpenMP correctness semantics

- MPI only specifies interoperability with threads, not with OpenMP (or any other high-level programming model using threads)
  - 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/recv 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;
}
```

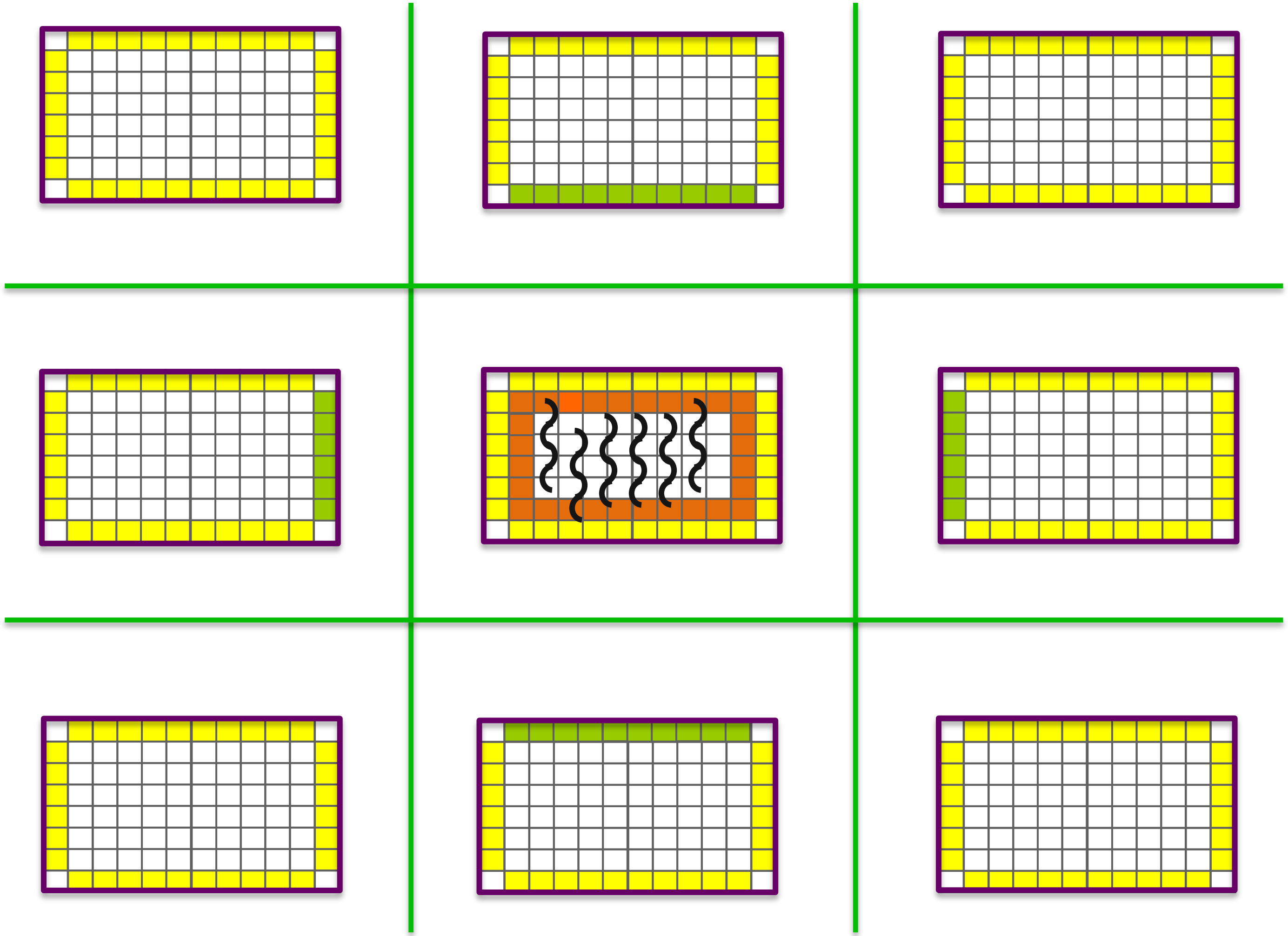
Correct example. Each task is nonblocking.

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

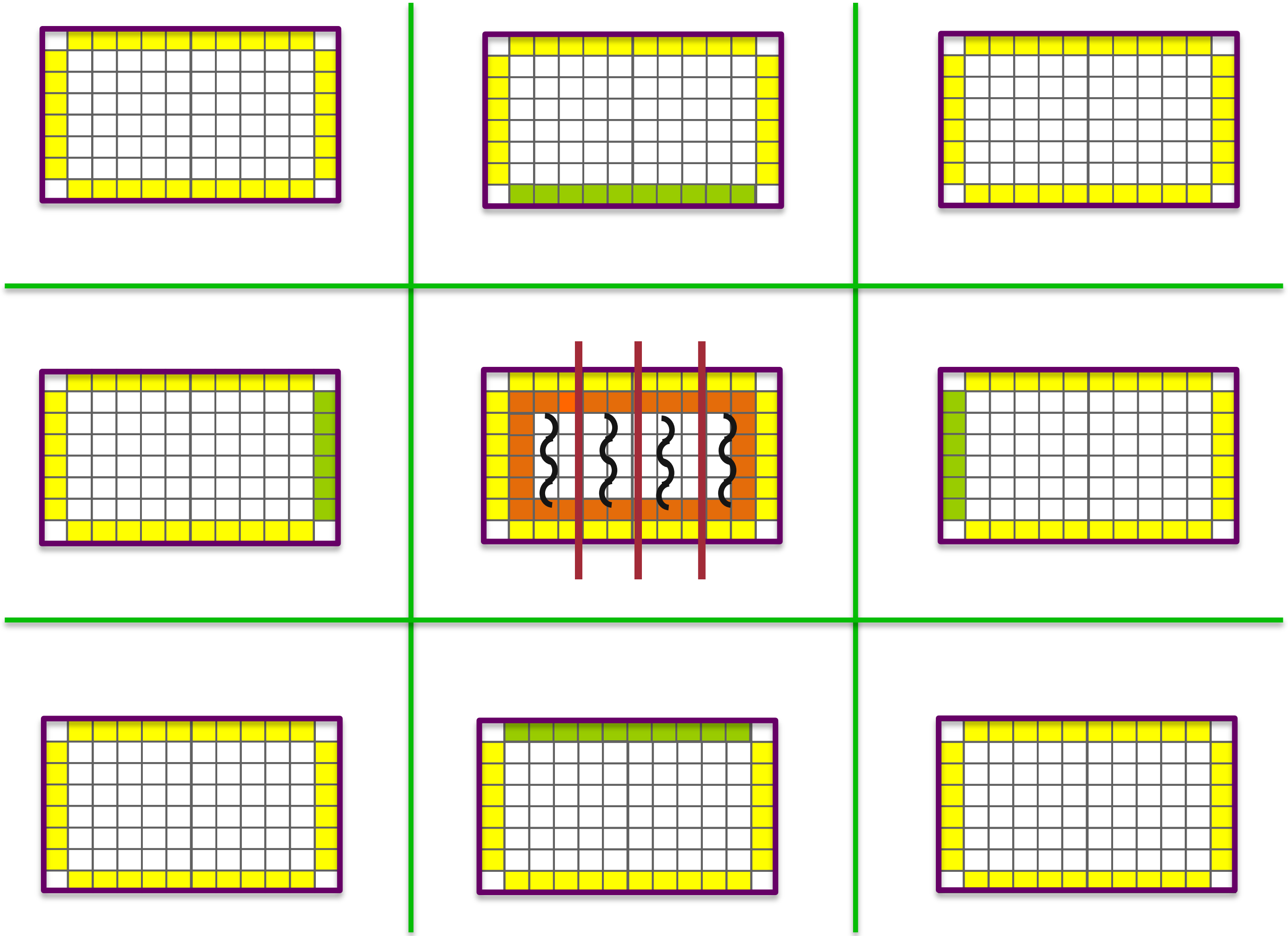
Exercise 1: Stencil in Funneled mode (1/2)



Exercise 1: Stencil in Funneled mode (2/2)

- Parallelize computation (OpenMP parallel for)
- Main thread does all communication
- *Start from derived_datatype/stencil.c*
- *Solution available in threads/stencil_funneled.c*

Exercise 2: Stencil in Multiple mode (1/2)



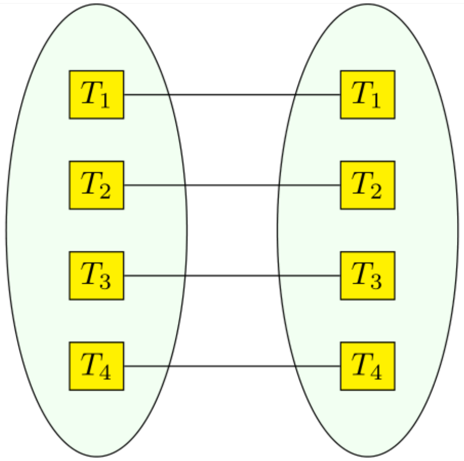
Exercise 2: Stencil in Multiple mode (2/2)

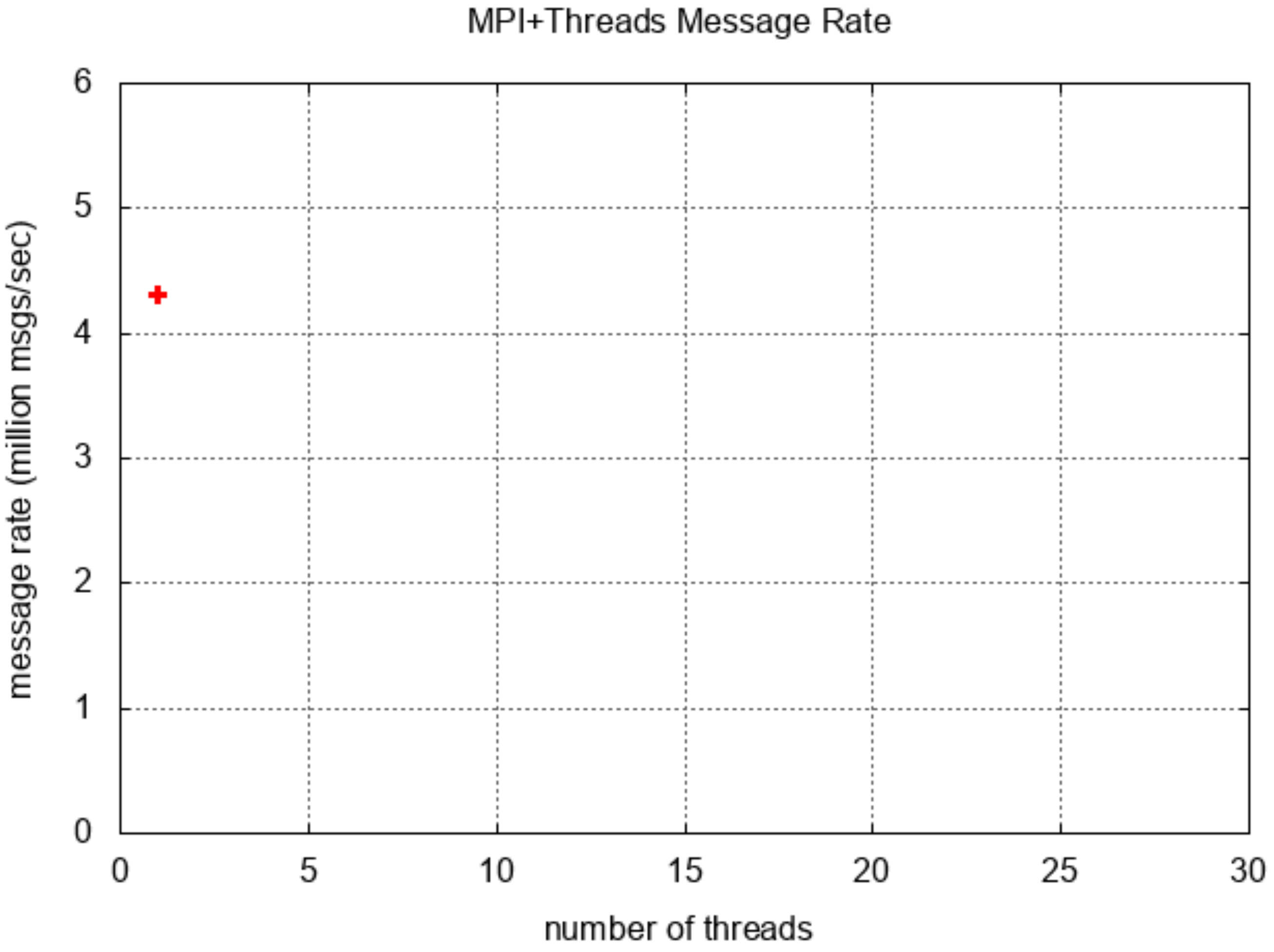
- Divide the process memory among OpenMP threads
- Each thread responsible for communication and computation
- *Start from threads/stencil_funneled.c*
- *Solution available in threads/stencil_multiple.c*

MPI+threads performance

MPI_THREAD_MULTIPLE performance

Latency-sensitive small messages

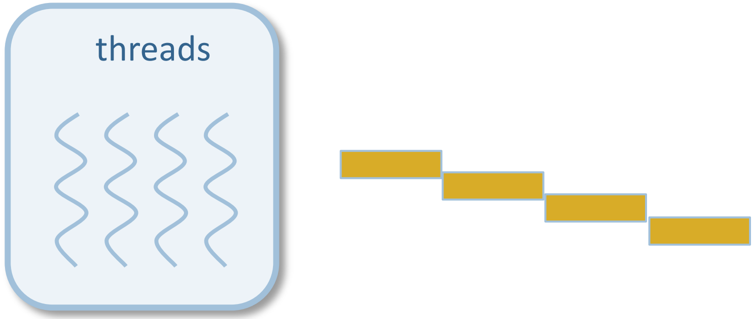


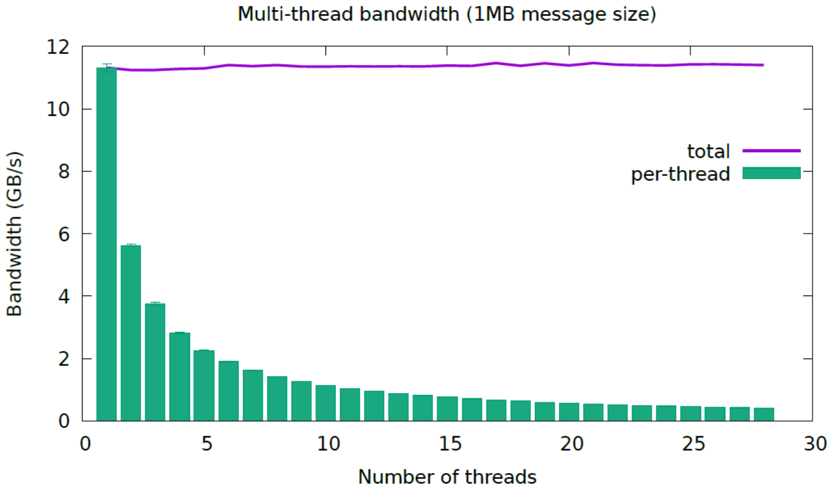


MPI_THREAD_MULTIPLE performance

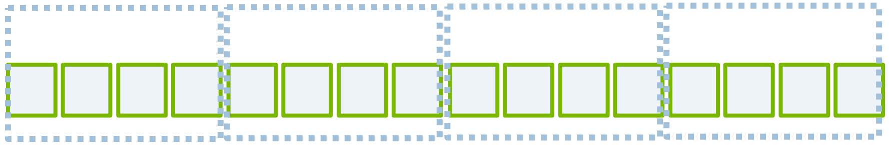
Comparisons

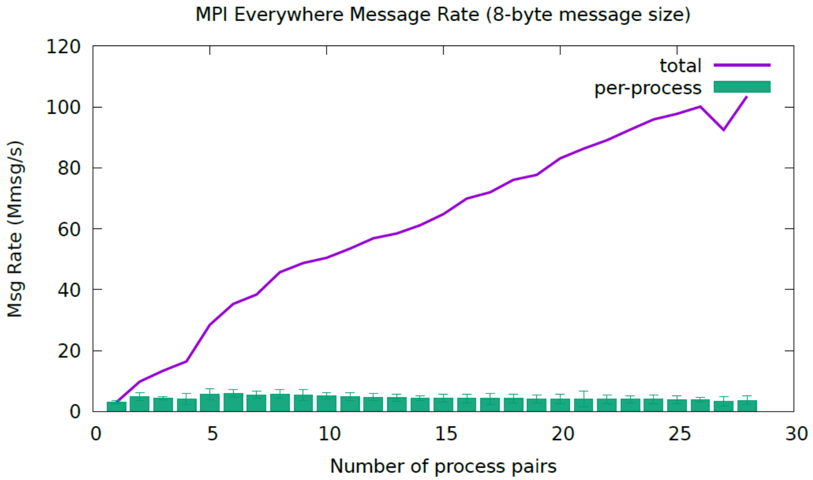
bandwidth-dominated
large messages





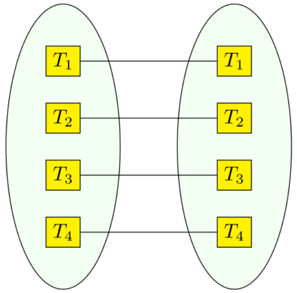
Flat-MPI
small messages



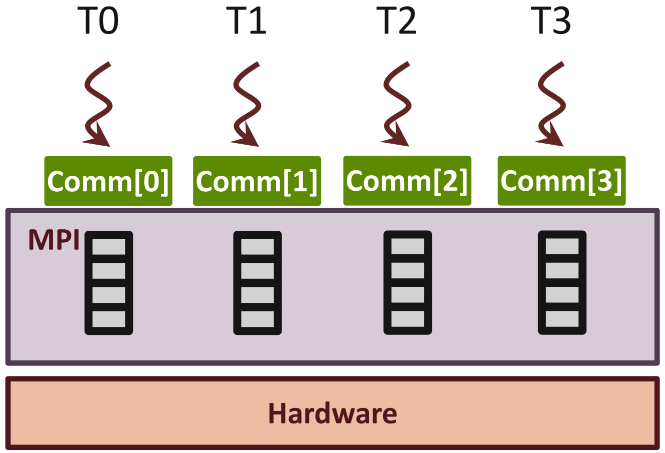


Recommendation: Maximize independence between threads with communicators

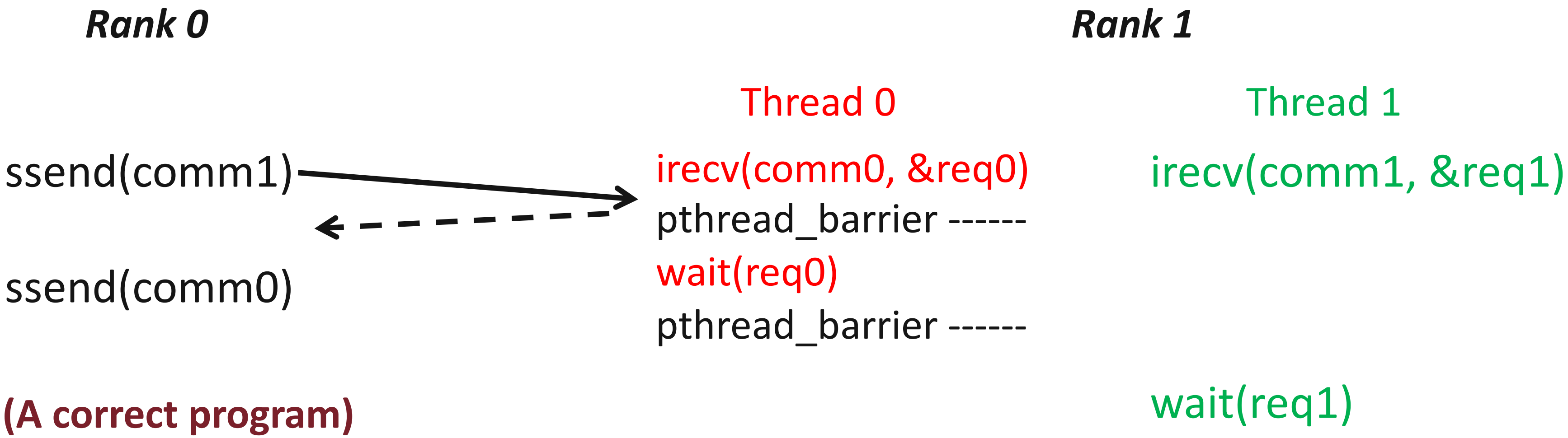
- Each thread accesses to a **different communicator**
  - Each communicator may be associated with isolated resource in an MPI implementation



```
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, ..);
```



Communication Isolation Limitations



Rank 0

ssend(comm1)

ssend(comm0)

(A correct program)

Rank 1

Thread 0

irecv(comm0, &req0)
pthread_barrier ------
wait(req0)
pthread_barrier ------

Thread 1

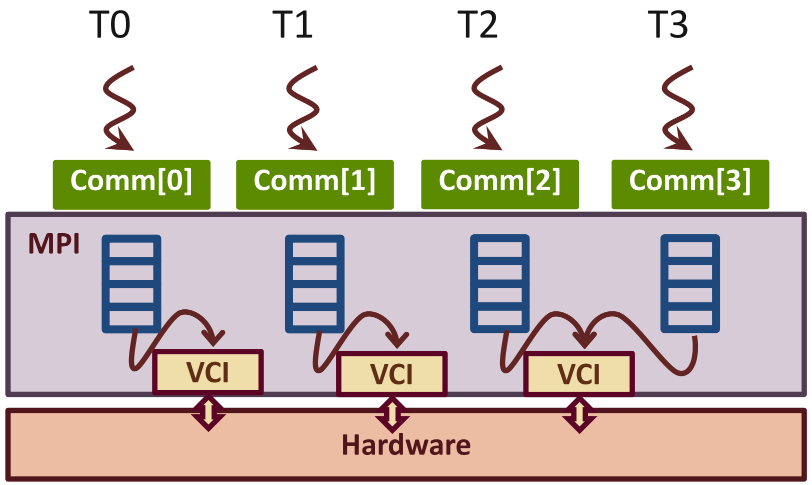
irecv(comm1, &req1)

wait(req1)

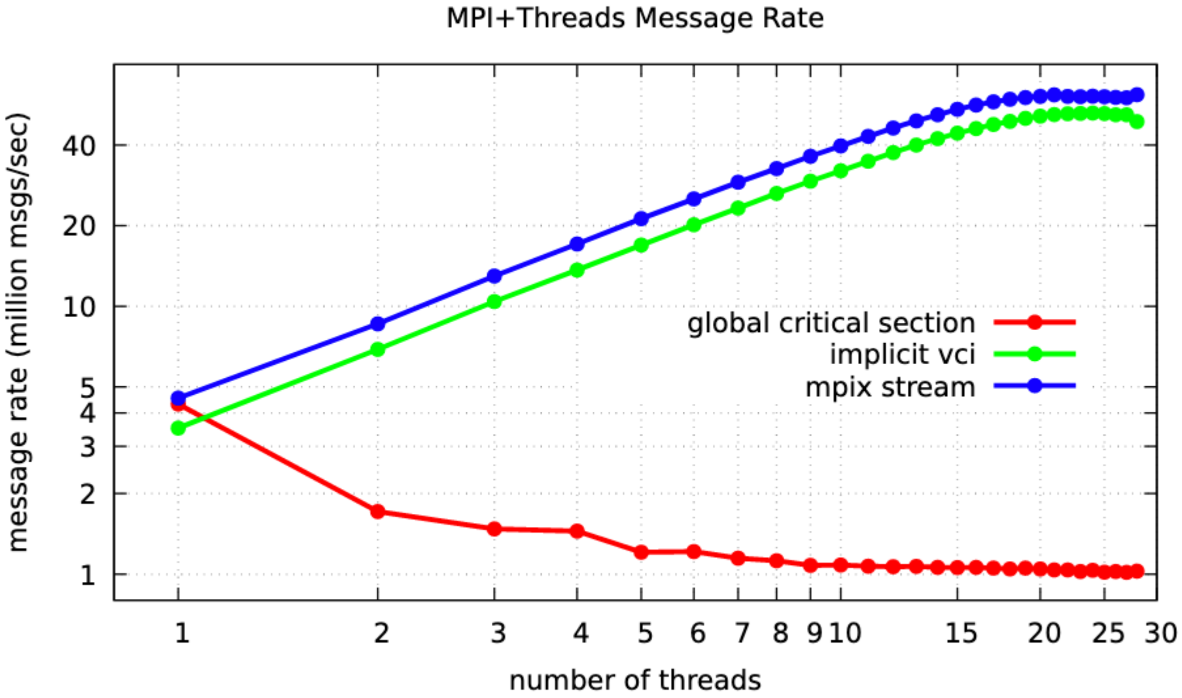
- ***Progress:*** A blocked thread will not prevent progress of other runnable threads on the same process
  - ssend(comm1) returns only after irecv(comm1) is posted
    - MPI may internally send handshake messages to synchronize
  - Thread 0 has to make progress for comm1 in wait(req0) (e.g., **access comm1’s receive-queue**), to ensure ssend(comm1) can complete

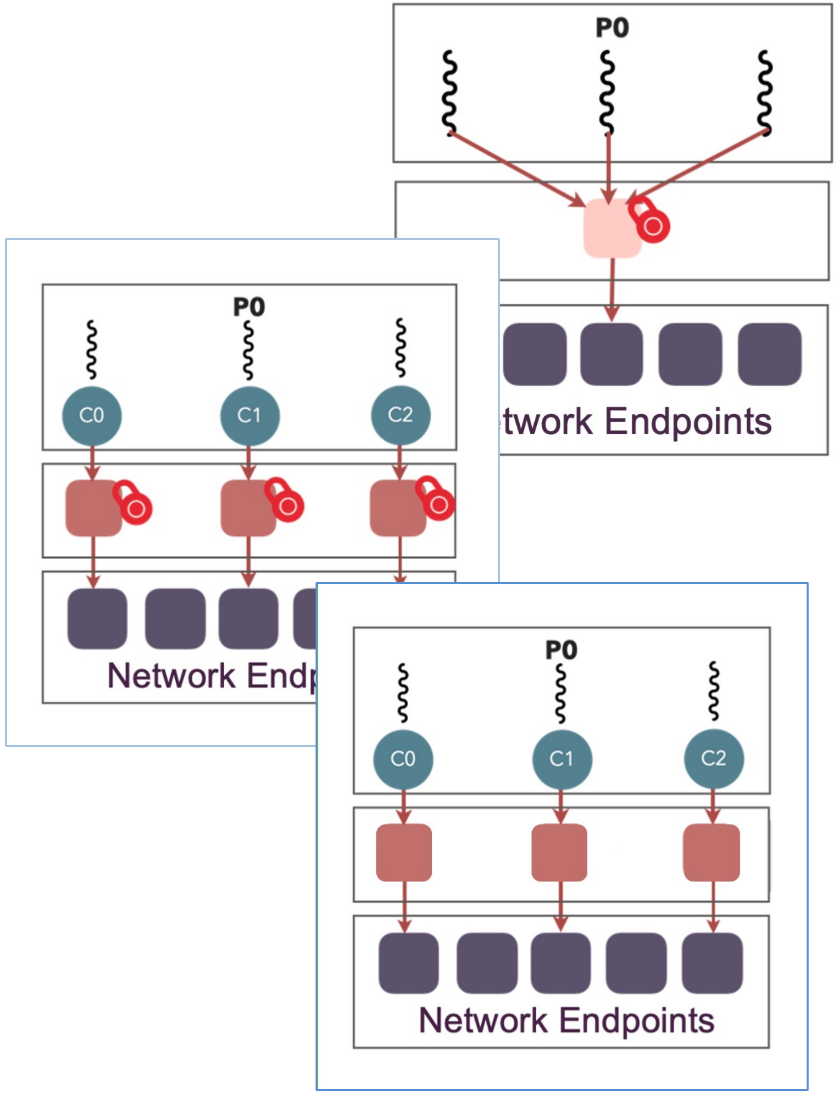
Possible Optimizations MPI libraries can do

- Virtual Communication Interface (VCI)
  - Each VCI abstracts a set of network/shared-memory resources
  - Some networks support multiple VCIs: InfiniBand contexts, scalable endpoints over Intel Omni-Path
  - Traditional MPI implementation uses single VCI
    - Serializes all traffic
    - Does not fully exploit network hardware resources
- **Utilizing multiple VCIs to maximize independence** in communication
  - Separate VCIs per communicator or per RMA window
  - Distribute traffic between VCIs with respect to ranks, tags, and generally out-of-order communication
  - M-N mapping between Work-Queues and VCIs



THREAD_MULTIPLE performance





Exercise 5: Stencil with Independent Communicators

- Divide the process memory among OpenMP threads
- Each thread responsible for communication and computation
- Each thread uses a different communicator
- *Start from threads/stencil_multiple.c*
- *Solution available in threads/stencil_multiple_ncomms.c*

Section Summary

- Hybrid MPI + “X” is a promising approach for large scale programming (e.g., MPI+Threads)
  - Less memory consumption
  - More efficient on-node data movement (load/store)
- MPI thread safety: SINGLE, FUNNELED, SERIALIZED, MULTIPLE
- Use MPI_Init_thread for threaded programs (i.e., not SINGLE)
- THREAD_MULTIPLE ordering & progress semantics
- Always maximize independence between threads in your program
  - Independent communicators, no wildcard, independent peer_ranks and tags

MPI Hybrid Programming: Accelerators

(Thanks to Jiri Kraus @ NVIDIA for several corrections and comments)

Introduction

Accelerators are becoming increasingly popular in parallel computing

- **CPUs**
  - Task-sequential execution model (focus on latency)
  - Small # of complex compute cores (out-of-order execution)
  - Deep pipelines
  - Large caches
  - Branch prediction hardware
- **GPUs**
  - Data-parallel execution model (focus on throughput)
  - Large # of simple compute elements (in-order execution)
  - Small caches
  - Shallow pipelines
  - Large off-chip global High-Bandwidth Memory (HBM)
  - High FLOPs/W and FLOPs/$

Top500 Accelerators Based Systems (June 2023)

- #1 - Frontier (ORNL USA)
  - AMD MI250X
- #3 - LUMI (CSC Finland)
  - AMD MI250X
- #4 - Leonardo (EuroHPC/CINECA Italy)
  - NVIDIA A100
- #5 - Summit (ORNL USA)
  - NVIDIA V100
- #6 - Sierra (LLNL USA)
  - NVIDIA V100
- #8 - Perlmutter (LBNL/NERSC USA)
  - NVIDIA A100
- #9 – Selene (NVIDIA Corporation USA)
  - NVIDIA A100

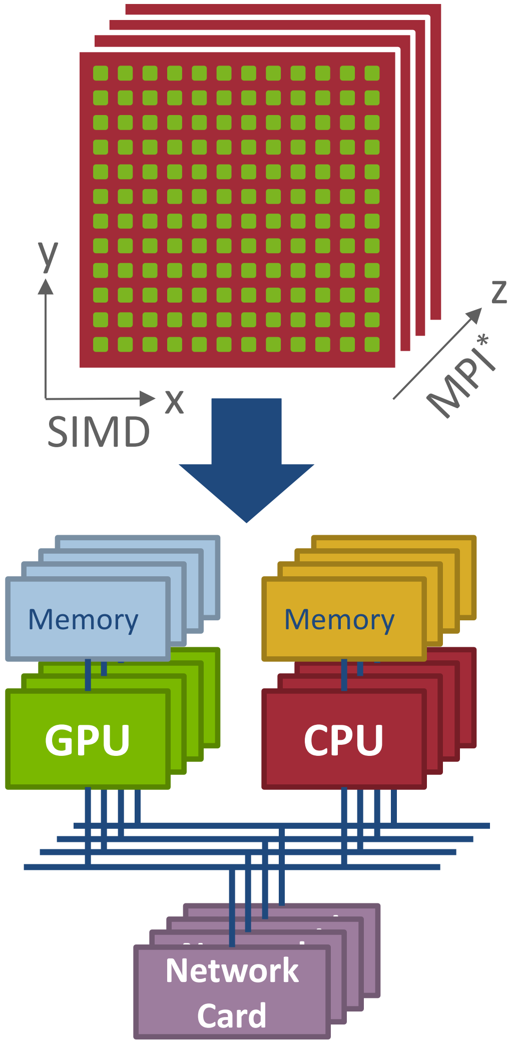
Upcoming Exascale Accelerators Based Systems

- Aurora (ANL USA)
  - Intel based technology
    - https://www.anl.gov/article/us-department-of-energy-and-intel-to-deliver-first-exascale-supercomputer
- El Capitan (LLNL USA)
  - AMD based GPU technology
    - https://www.llnl.gov/news/llnl-and-hpe-partner-amd-el-capitan-projected-worlds-fastest-supercomputer

Programming Model for Accelerators

- **GPUs** are well suited for fine grain data level parallelism
- Shared Memory, Single Instruction Multiple Data (SIMD) model
- Many available compute platforms and programming frameworks (focus on their memory model and interaction with MPI)
  - NVIDIA CUDA (NVIDIA platform only)
  - AMD ROCm & HIP
  - OpenMP
  - OpenACC (mentioned but not covered)
  - OpenCL & SYCL

Multi-dimensional Dataset

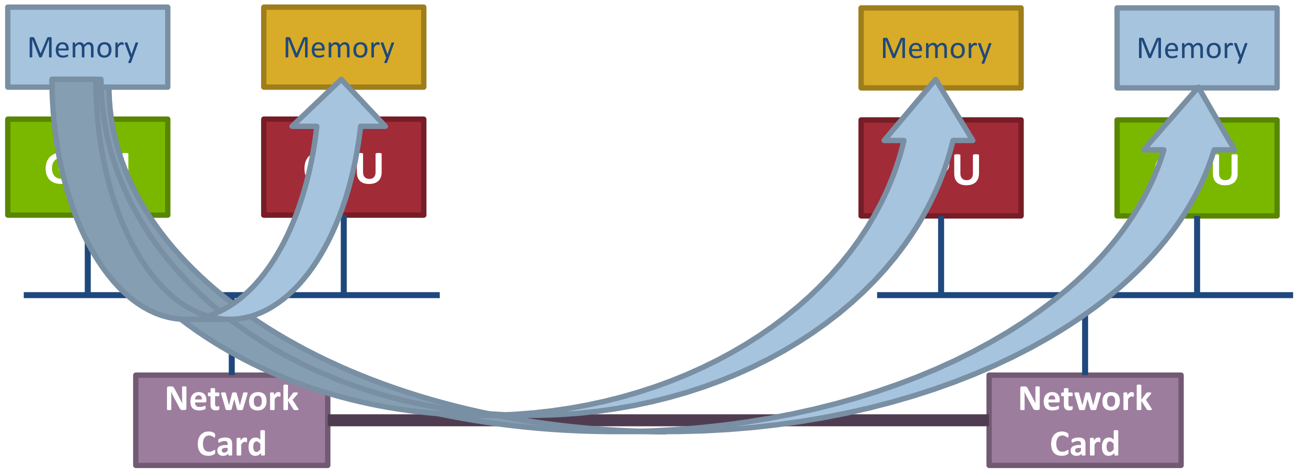


(*)Single Program Multiple Data

Interoperability with MPI

GPUs have separate physical memory subsystem

How to move data between GPUs with MPI?

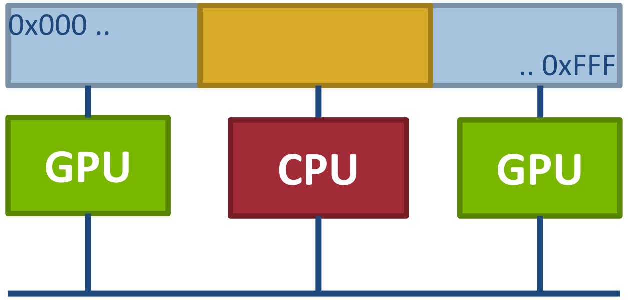


Real answer: It depends on what GPU library, what hardware and what MPI implementation you are using

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

Unified Virtual Addressing (UVA)

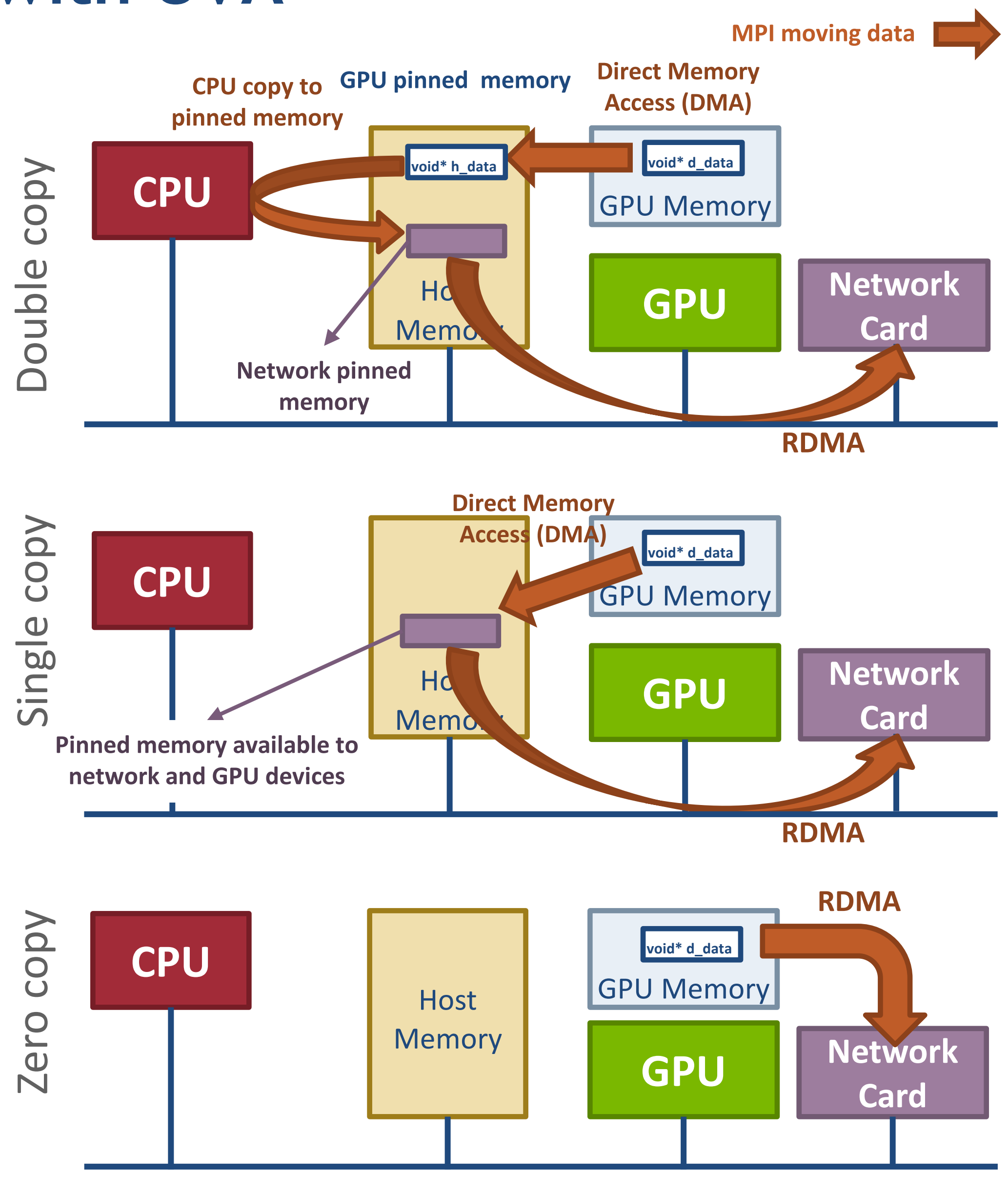
- UVA is a memory address management system supported in modern 64-bit architectures
  - Requires device driver support
- The same virtual address space is used for all processors, host or devices
- No distinction between host and device pointers
- The user can query the location of the data allocation given a pointer in the unified virtual address space and the appropriate GPU runtime library query APIs (“GPU-aware” MPI library)



UVA: Single virtual address space for the host and all devices

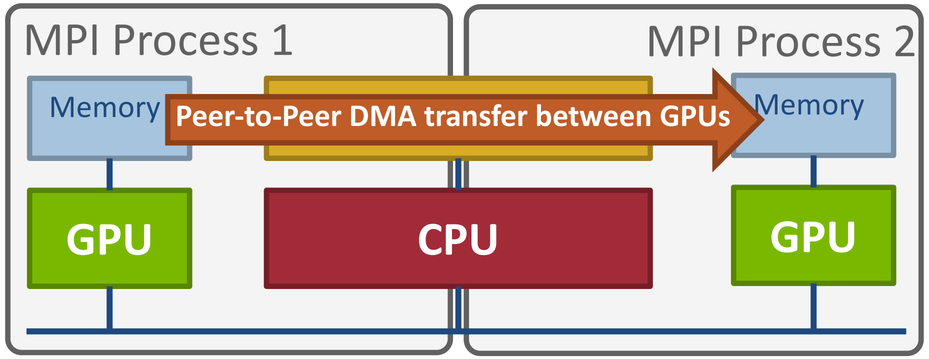
Remote Direct Memory Access with UVA

- Only GPU-enabled MPI implementations can take advantage of UVA
- 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 optimize** data transfers



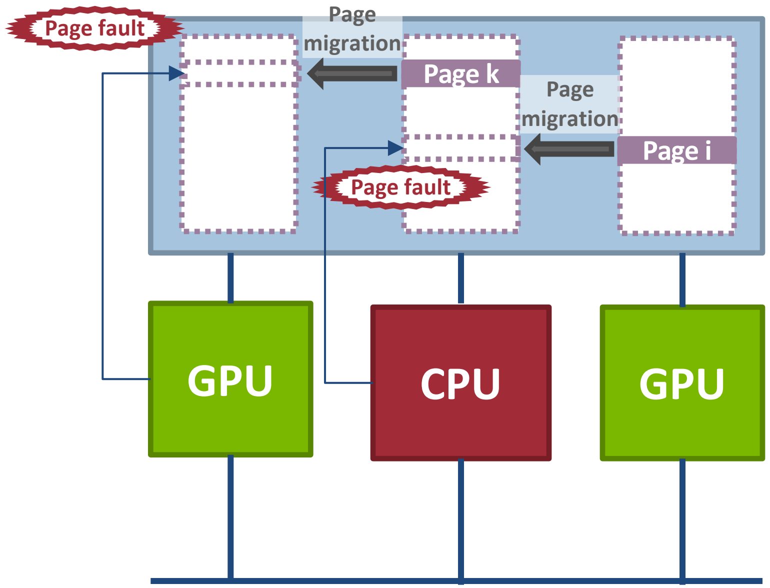
Intranode Communication with UVA

- Intranode Optimization
  - GPU peer-to-peer data transfers are possible
  - MPI can directly move data between GPU devices



Heterogeneous Memory Management (HMM)

- Next step towards the unification of heterogeneous memory spaces in the Linux Kernel (not yet available)
- Support started with version 4.14 through helper functions to be used by device drivers
  - Support paging in device for migrating memory between host and GPU
- Automatic data movement between host and GPU memories (called Unified Memory in CUDA)
  - Data is automatically migrated between host and GPU on page faults
  - Moving pages to GPU and back to host is similar to swap-out and swap-in of pages to and from disk



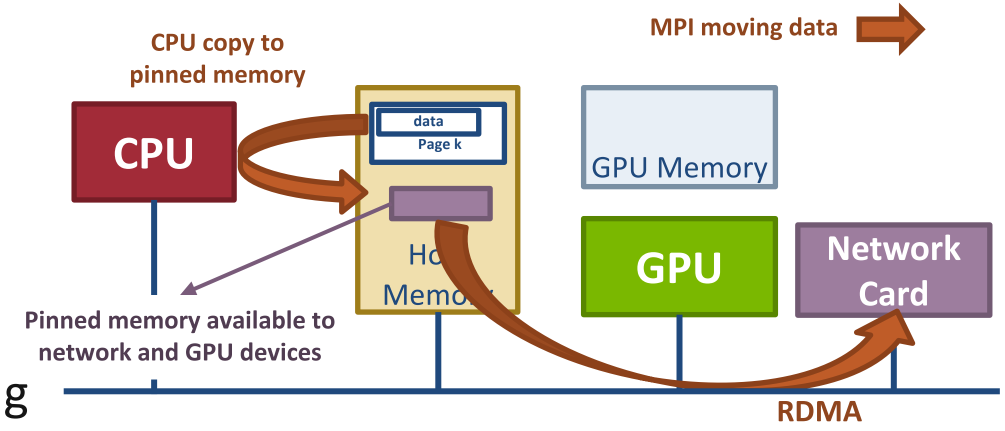
Single memory space accessible to all devices and host. Transparently managed heterogeneous memory.

MPI + HMM in a Nutshell

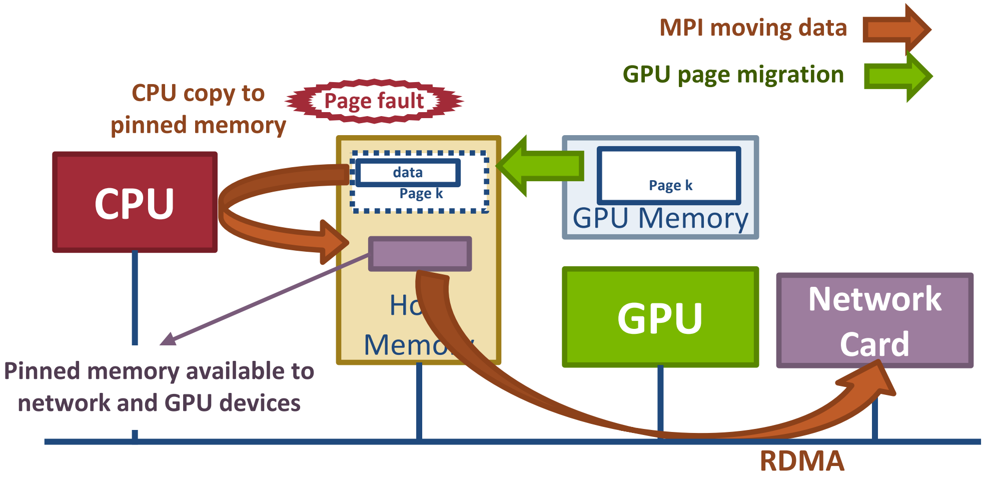
- In theory, any MPI implementation can transparently work with HMM
  - The MPI implementation can always assume that the data is on the host (or device)
  - GPUs take care of moving data between device and host memories
  - *Trying to register memory on the wrong device from the network should simply fail for HMM, but there have been reports of silent failures in this regard for CUDA, so you might need to be careful*
  - *Data in HMM can be corrupted if GPU updates pages during network transfer*
- In practice MPI implementations should never use HMM directly
  - Managed heterogeneous memory cannot be directly accessed by the network (correctness issue)
    - Virtual address cannot be pinned to a fixed physical memory region since GPU might need to migrate the data
    - Intermediate buffer is needed to copy data from HMM
- In any case MPI can never know in which device data is physically located
  - Data management is completely handled by GPU and can cause unnecessary data movement (performance issue)

MPI + HMM Assuming Data on Host

- MPI can assume data is on host memory
- MPI copies data to network pinned memory
  - Network registration will fail
- On a correct guess
  - The copy will not trigger a page fault to bring data from GPU
- On incorrect guess
  - An **expensive page fault** will occur


MPI moving data
CPU copy to pinned memory
CPU
data
Page k
GPU Memory
Host Memory
GPU
Network Card
Pinned memory available to network and GPU devices
RDMA

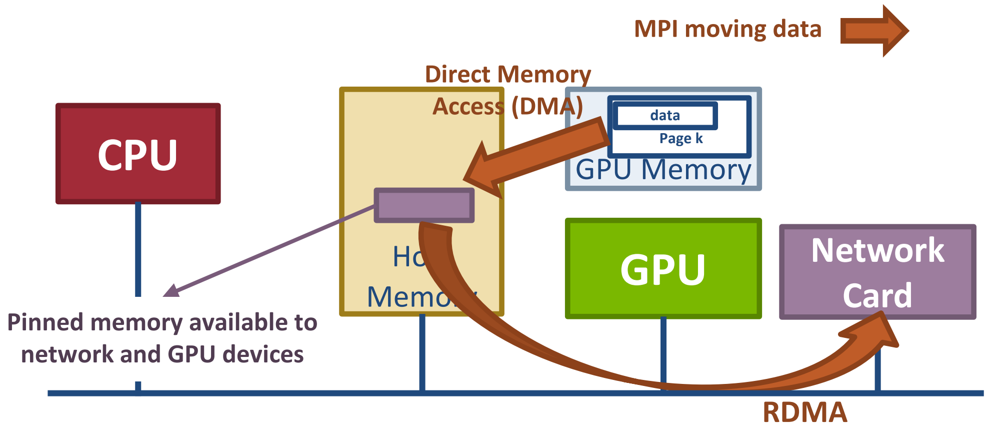
Correct guess: data is on host memory



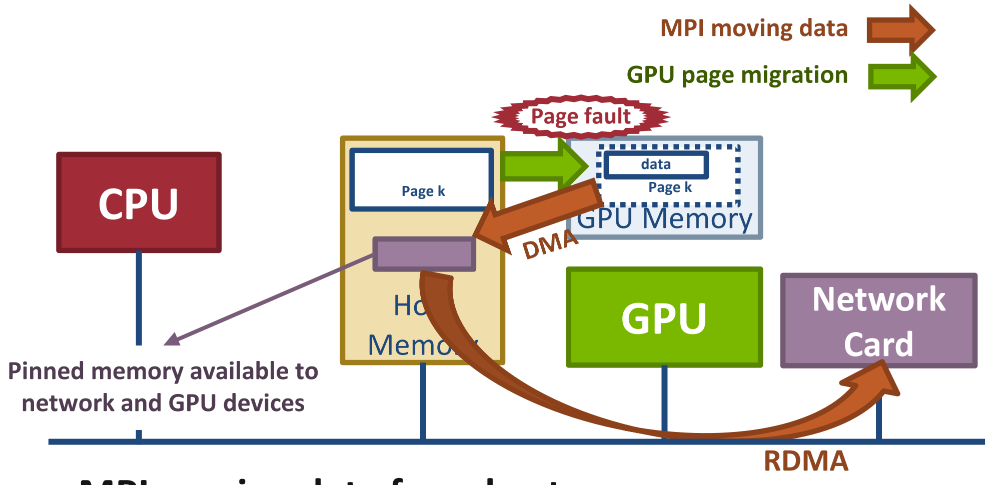
MPI moving data from host memory on a wrong guess

MPI + HMM Assuming Data on GPU

- MPI can assume data is on some GPU device memory
- MPI would need to move data from the GPU device memory to network pinned memory
  - This can be either host or GPU memory (but not unified memory)
- On a correct guess
  - The copy will not trigger a page fault
- On incorrect guess
  - An **expensive page fault** will occur when accessing data on the GPU device memory
- Most MPI implementations assume memory to reside on the GPU


MPI moving data
Direct Memory Access (DMA)
CPU
data
Page k
GPU Memory
Host Memory
GPU
Network Card
Pinned memory available to network and GPU devices
RDMA

Correct guess: data is on the target GPU



MPI moving data from host memory on a wrong guess

Compute Unified Device Architecture (CUDA)

(*Thanks to CJ Newburn from NVIDIA for review and comments*)

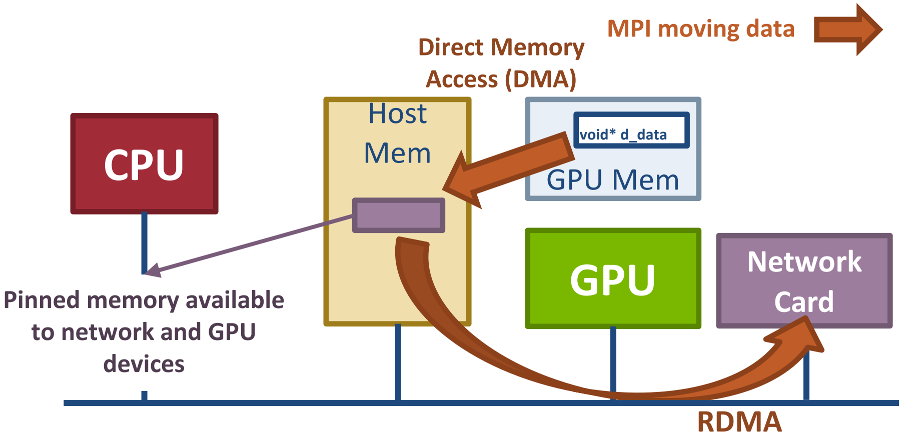
Overview

- General-purpose parallel computing platform and programming model released by NVIDIA in 2006
- Provides a user library (*libcuda*), runtime (*libcudart*), device drivers and C/C++ compiler (NVCC)
- Programming language extensions for C/C++
  - Define C functions to run on the GPU (kernels) using the **__global__** declaration specifier
  - Kernels can be launched with different number of threads using the **<<<…>>>** execution syntax
  - Each thread executing the kernel is given a unique thread ID accessible from inside the kernel using the built-in **threadIdx** variable
- Support other languages such as FORTRAN

```
/* Kernel definition */
__global__ void gpu_kernel(double *in,
                           double *out)
{
  /* get indices from thread id */
  int i = threadIdx.x;
  int j = threadIdx.y;

  /* each thread performs work */
  out[i][j] = f(in, i, j);
}

int main()
{
  double *in_h, *out_h, *in_d, *out_d;
  in_h  = malloc(size);
  out_h = malloc(size);
  cudaMalloc(&in_d, size);
  cudaMalloc(&out_d, size);

  cudaMemcpy(in_d, in_h, size,
             cudaMemcpyHostToDevice);

  /* kernel invocation with N threads */
  gpu_kernel<<<2,N>>>(in_d, out_d);

  cudaMemcpy(out_h, out_d, size,
             cudaMemcpyDeviceToHost);

  [...snip...]

  return 0;
}
```

MPI + GPUDirect (CUDA ≥ 4)

- GPUDirect 1.0 (Q2’ 2010) allows pinned memory to be shared by GPU and NIC such that GPU can directly copy data in/out pinned memory and NIC can DMA data from it
- GPUDirect 2.0 (Peer-to-peer 2011) extends UVA support by allowing direct memory transfers between GPUs in the same node bypassing host completely


MPI moving data
Direct Memory Access (DMA)
CPU
Host Mem
void* d_data
GPU Mem
GPU
Network Card
Pinned memory available to network and GPU devices
RDMA

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

if(my_rank == sender) {
    gpu_kernel<<<grid,block,0,stream0>>>(dev_buf);
     cudaStreamSynchronization(stream0);
    MPI_Isend(dev_buf, size, MPI_DOUBLE, receiver, 0, comm, req);
} else {
    MPI_Recv(dev_buf, size, MPI_DOUBLE, sender, 0, comm, &status);
    gpu_kernel<<<..>>>(dev_buf);
}
```

- MPI_Isend is not aware of which stream the kernel runs on, could issue the cudaMemcpy on a different stream. Explicit synchronization needed to avoid async access to dev_buf

MPI + GPUDirect RDMA (CUDA ≥ 5)

- Technology introduced in 2013 with Kepler-class GPUs and CUDA-5
- GPU memory is directly accessible to third-party devices, including network interfaces (NIC driver talks to CUDA driver to register GPU memory)
- RDMA operations to/from the device memory are possible and completely bypass the host memory (zero copy)

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

if(my_rank == sender) {
    gpu_kernel<<<grid,block,0,stream0>>>(dev_buf);
    cudaStreamSynchronization(stream0);
    MPI_Isend(dev_buf, size, MPI_DOUBLE, receiver, 0, comm, req);
} else {
    MPI_Recv(dev_buf, size, MPI_DOUBLE, sender, 0, comm, &status);
    gpu_kernel<<<..>>>(dev_buf);
}
```

MPI + Unified Memory (CUDA ≥ 6)

- Unified Memory (UM) between host and device
  - CUDA kernel driver has a copy of the page table of the host and can handle pagefaults by migrating pages from host to device & vice versa
- UM-aware MPI implementations stage data into tmp buffer
- As mentioned before, performance can be bad
  - MPI never knows if pages are on host or device (can only guess)

```
double *buf;
cudaMallocManaged(&buf, size);

/* initialize buf in host … */

if(my_rank == sender) {
    gpu_kernel<<<grid,block,0,stream0>>>(buf);
    cudaStreamSynchronize(stream0);
    MPI_Isend(buf, size, MPI_DOUBLE, receiver, 0, comm, req);
} else {
    MPI_Recv(buf, size, MPI_DOUBLE, sender, 0, comm, &status);
    gpu_kernel<<<..>>>(buf);
}
```

MPI + CUDA Optimizations Historical Summary

Period	CUDA version	Major Features	MPI Optimization Space	MPI Implementation Requirements
After 2011	>= 4.0 < 5.0	• GPUDirect 1.0: RDMA can use GPU pinned memory • GPUDirect 2.0: GPU peer-to-peer DMA possible	• Use DMA and RDMA without extra memory copies to temporary buffers	GPU-aware MPI implementations
After 2012	>= 5.0	• GPUDirect RDMA: GPU memory is directly accessible to third-party devices	• Completely bypass host memory through RDMA to/from GPU memory	GPU-aware MPI implementations
After 2014	>= 6.0	• Unified Memory: shared memory between host and devices and automatic page migration	• The hardware takes care of moving data between host and device memories • MPI optimizations are limited and need user hints	GPU-aware MPI implementations needed for performance

MPI + GPUDirect RDMA (Supported HW & SW)

- Mellanox Host Channel Adapters (HCA):
  - ConnectX-3, ConnectX-3 Pro, Connect IB, ConnectX-4, ConnectX-5, or newer
  - MLNX_OFED v2.1-x.x.x or later (compatible with HW, support for older HW may be dropped in newer MLNX_OFED)
  - Plugin Module to enable GPUDirect RDMA
- NVIDIA GPUs:
  - NVIDIA Tesla, Quadro K-Series or Tesla/Quadro P-Series, or newer
  - NVIDIA Driver and Kernel Modules
  - NVIDIA Runtime and Toolkit

Future directions in MPI + GPU

- General purpose GPU programming is still evolving
  - I am betting on some sorts of unification
- Synchronization between GPUs and CPUs/NICs is a bottleneck
  - Pinning memory incurs huge overhead (up to millisecond)
  - cudaStreamSynchronization et.al. are very expensive
- Potential solutions
  - Memory type hints
  - Stream-aware MPI
  - In-kernel triggering

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

if(my_rank == sender) {
    gpu_kernel<<<grid,block,0,stream0>>>(dev_buf);
    cudaStreamSynchronization(stream0);
    MPI_Isend(dev_buf, size, MPI_DOUBLE, receiver, 0, comm, req);
} else {
    MPI_Recv(dev_buf, size, MPI_DOUBLE, sender, 0, comm, &status);
    gpu_kernel<<<..>>>(dev_buf);
}
```

Summary

- Accelerators are becoming increasingly important in HPC
- 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
  - For CUDA and ROCm P2P/RDMA support from GPU memory is enabled for contiguous datatypes through the UCX driver
  - For OpenCL/SYCL SVM can potentially enable similar optimizations as CUDA/ROCm but at the current state no such support is available and explicit data movement between host and device memory is still required
  - For OpenMP/OpenACC data movement is managed through directives but compilers can decide to leverage HMM

Example: Stencil with GPU

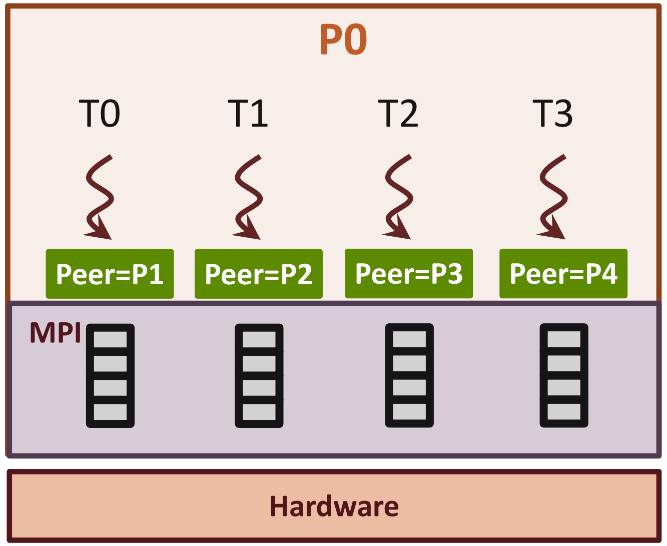
- Use GPU for updating the grid
  - Compute the grid on GPU
  - Pack/Unpack Halo regions to host memory for communication
- *Code in accelerators/stencil_cuda.cu*
- Special Environment Setup on Perlmutter

```
$ module swap PrgEnv-gnu PrgEnv-nvhpc
```

Other methods of expressing independence (1/2)

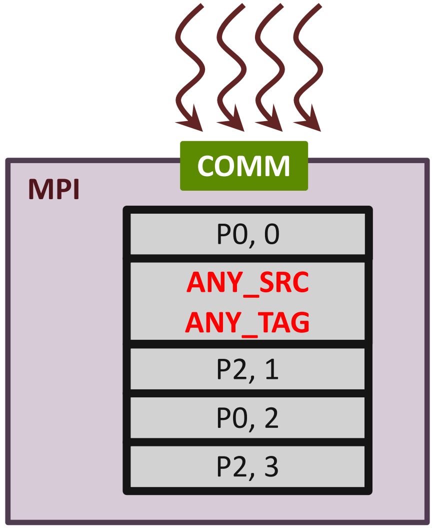
- Each thread communicates with **different peer_rank or tag**
  - MPI may assign isolated resource for different set of [peer_rank + tag]

```
#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, ..);
```



Other method of expressing independence (2/2)

- 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 **mpi_assert_no_any_source, mpi_assert_no_any_tag** (already proposed to MPI standard)
  - MPI can get rid of the single receive-queue for the communicator



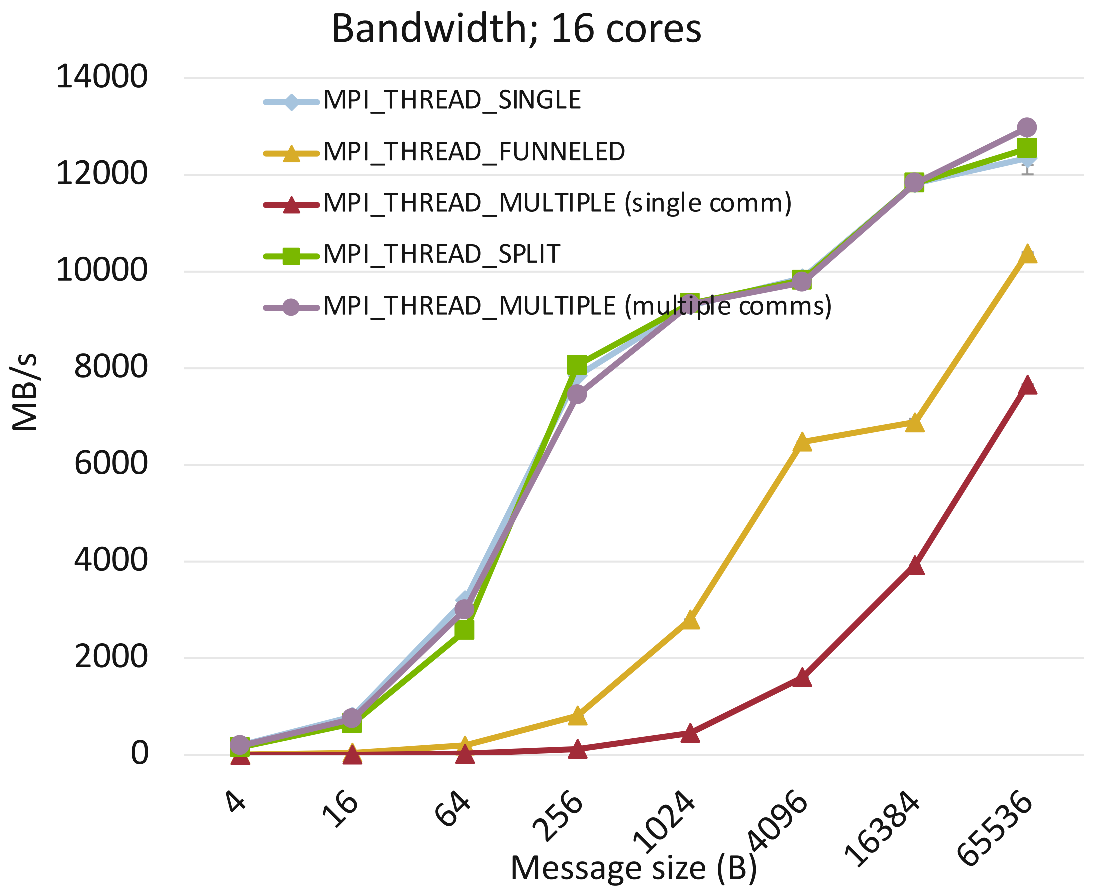
```
MPI_Info info;
info = MPI_Info_create();
MPI_Info_set(info,
   "mpi_assert_no_any_source", "true");
MPI_Comm_set_info(comm, info);
MPI_Info_free(&info);
/* Communicate without ANY_SOURCE */
```

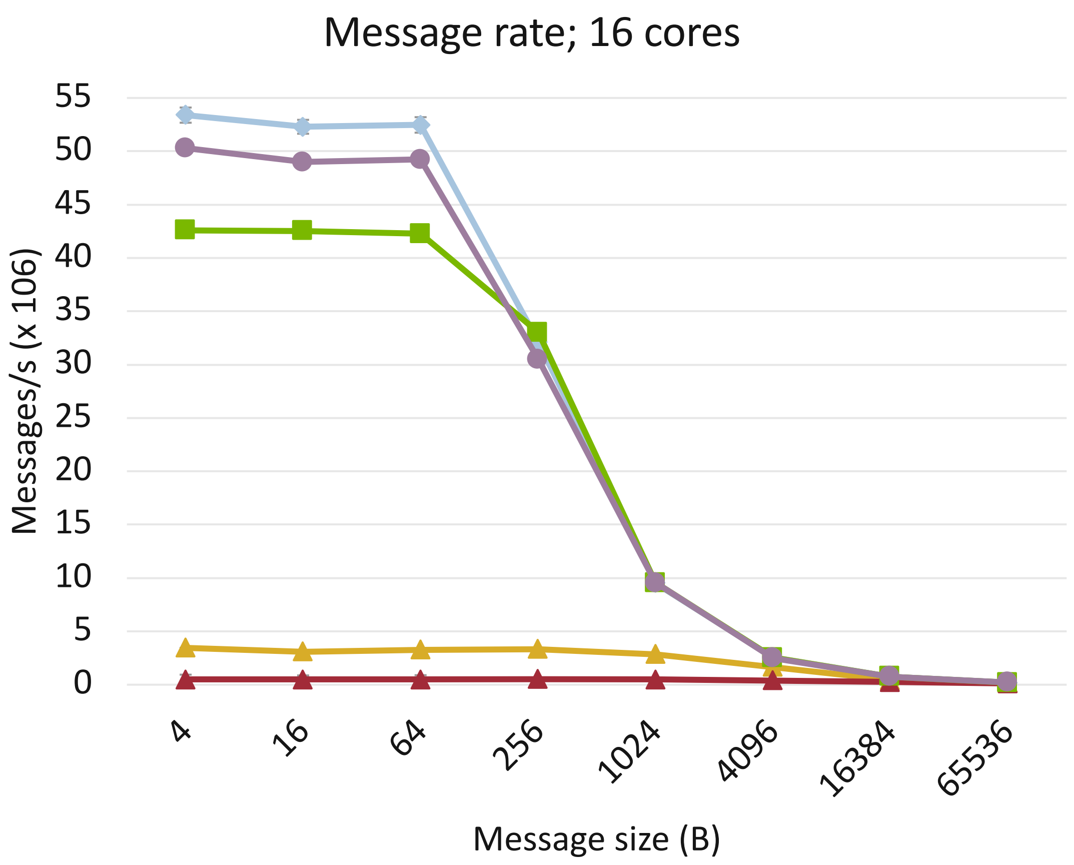
MPI+threads optimizations in Intel MPI 2019

- Supported starting Intel MPI Library 2019 (for Linux).
- MPI_Init_thread(MPI_THREAD_MULTIPLE)
- Environment variables
  - I_MPI_THREAD_SPLIT=1
  - I_MPI_THREAD_RUNTIME=openmp
- MPI_THREAD_SPLIT
  - Only threads with the same thread id can communicate using distinct communicators, limiting space of possible communication patterns.
- Known issues
  - Using MPI_PROC_NULL with I_MPI_THREAD_SPLIT=1 causes errors.
  - MPI_Finalize can take long with I_MPI_THREAD_SPLIT=1.

For more information: https://software.intel.com/en-us/mpi-developer-guide-linux-multiple-endpoints-support

Exposing parallelism in micro-benchmarks

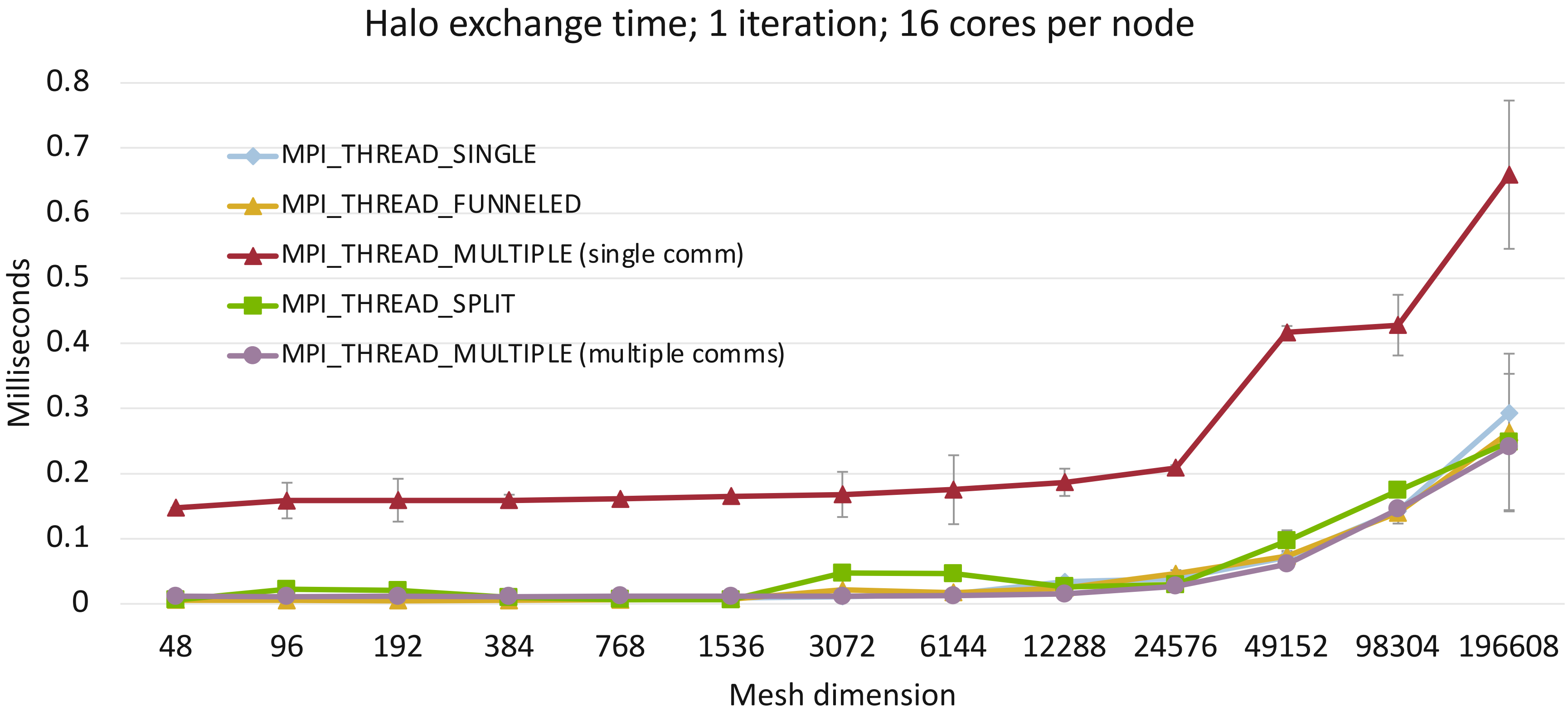




- All results, except “MPI_THREAD_MULTIPLE (multiple comms)” use Intel MPI 2019. The multiple comms version will be included into Intel MPI soon.

Exposing parallelism in stencil example

- Using 16 cores per node; 3 x 3 node grid.
- Halo exchange time of a rank on the central node.
- All results, except "MPI_THREAD_MULTIPLE (multiple comms)" use Intel MPI 2019. The multiple comms version will be included into Intel MPI soon.



What’s new in MPI-4

Introduction

- MPI-4 is official
  - https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf
- Major new features and changes
  - Persistent Collectives
  - Partitioned Communication
  - Sessions
  - Big Count
  - Error Handling Improvement
  - Topology Improvement

Persistent Collectives

- Similar to, but not exactly the same as regular nonblocking collective operations
- For each nonblocking MPI collective, add a persistent variant
- For every MPI_I<coll>, add MPI_<coll>_init
- Parameters are identical to the corresponding nonblocking variant – plus additional MPI_INFO parameter
- All arguments “fixed” for subsequent uses
- Persistent collective operations cannot be matched with blocking or nonblocking collective calls

Slide courtesy of Tony Skjellum

Persistent Collectives Example

Nonblocking collectives API

```
for (i = 0; i < MAXITER; i++) {
    compute(bufA);
    MPI_Ibcast(bufA, …, rowcomm, &req[0]);
    compute(bufB);
    MPI_Ireduce(bufB, …, colcomm, &req[1]);
    MPI_Waitall(2, req, …);
}
```

Persistent collectives API

```
MPI_Bcast_init(bufA, …, rowcomm, &req[0]);
MPI_Reduce_init(bufB, …, colcomm, &req[1]);
for (i = 0; i < MAXITER; i++) {
    compute(bufA);
    MPI_Start(req[0]);
    compute(bufB);
    MPI_Start(req[1]);
    MPI_Waitall(2, req, …);
}
```

Slide courtesy of Tony Skjellum

Partitioned Communication

- A MPI operation performed by multiple actors
  - Actors could be threads, or GPUs
  - Send with a big buffer that each actor “own” it a part of it
  - Differentiating different actors as partitions

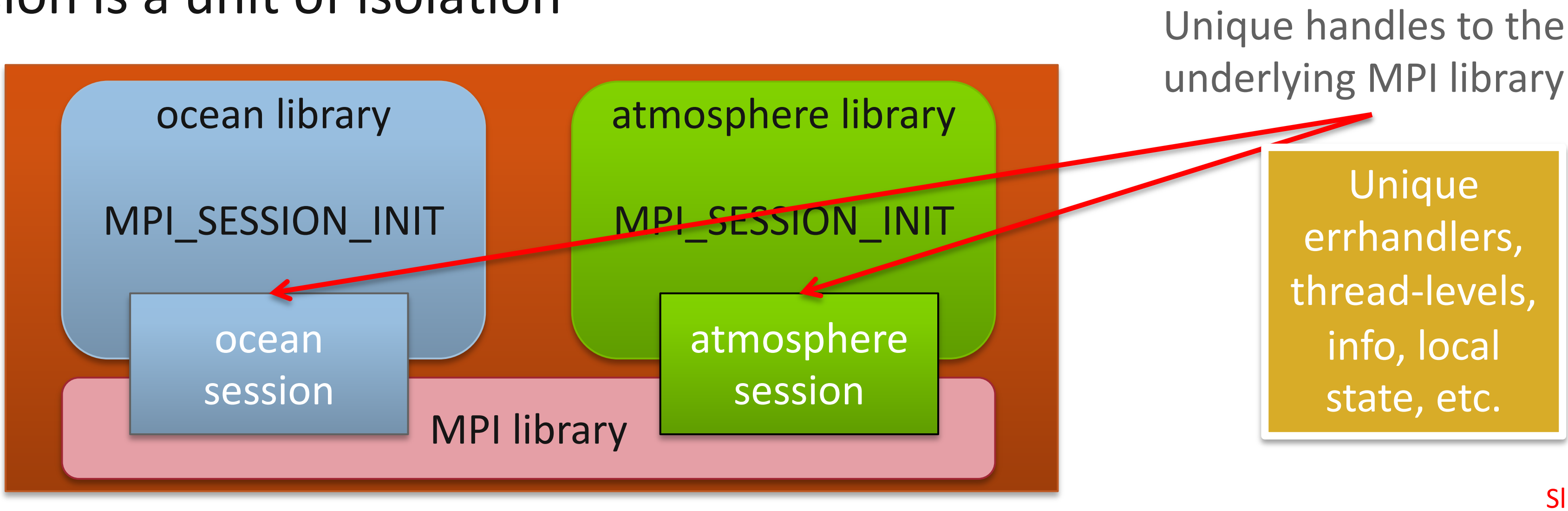
```
MPI_PSEND_INIT(buf, partitions, count, datatype, dest, tag, comm, info, request)
MPI_PREADY(partition, request)
MPI_PARRIVED(request, partition, flag)
```

Partitioned Communication Example

```
if (myrank == 0) {
    MPI_Psend_init(message, partitions, count, xfer_type, dest, tag,info, MPI_COMM_WORLD, &request);
    MPI_Start(&request);
#pragma omp parallel for shared(request) num_threads(NUM_THREADS)
    for (int i=0; i<partitions; i++)
    {
        /* compute and fill partition #i, then mark ready: */
        MPI_Pready(i, request);
    }
    while(!flag) {
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
    }
    MPI_Request_free(&request);
} else if (myrank == 1) {
    MPI_Precv_init(message, partitions, count, xfer_type, source, tag,info, MPI_COMM_WORLD, &request);
    MPI_Start(&request);
    while(!flag) {
        /* Do useful work */
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
    }
    MPI_Request_free(&request);
}
```

New Concept: “Session”

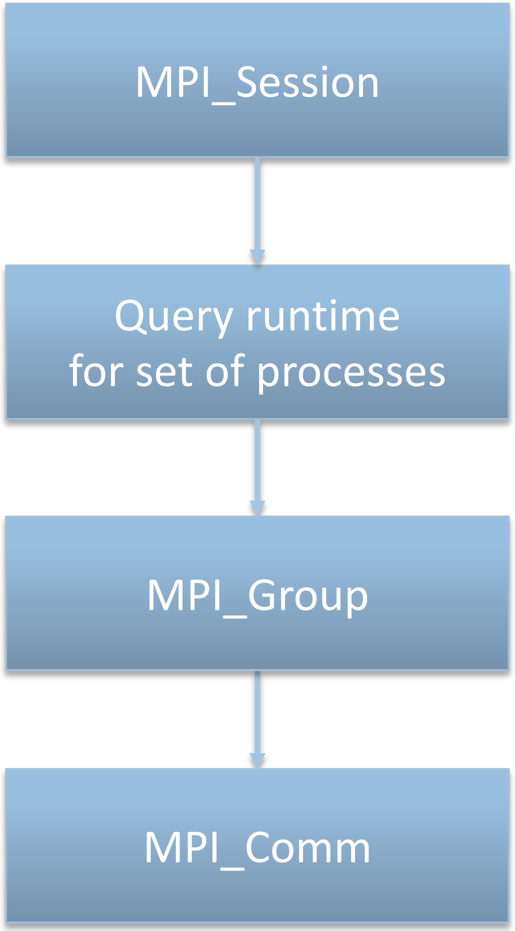
- A local handle to the MPI library
  - Implementation intent: lightweight / uses very few resources
  - Can also cache some local state
- Can have multiple sessions in an MPI process
  - MPI_Session_init(…, &session);
  - MPI_Session_finalize(…, &session);
- Each session is a unit of isolation



Slide courtesy of Tony Skjellum

Overview

- General scheme:
  - Query the underlying run-time system
    - Get a “set” of processes
  - Determine the processes you want
    - Create an MPI_Group
  - Create a communicator with just those processes
    - Create an MPI_Comm



Big Count

- Support count larger than INT_MAX
- Using MPI_Count type
- New interfaces with ”_c” suffix

```
MPI_Send(void *buf, int count, …)
MPI_Send_c(void *buf, MPI_Count count, …)
```

Noncatastrophic Errors

- Currently the state of MPI is undefined if any error occurs
- Even simple errors, such as incorrect arguments, can cause the state of MPI to be undefined
- Noncatastrophic errors are an opportunity for the MPI implementation to define some errors as “ignorable”
- For an error, the user can query if it is catastrophic or not
- If the error is not catastrophic, the user can simply pretend like (s)he never issued the operation and continue

Topology Improvements

- Hierarchical Mapping
- Feature Based MPI_COMM_SPLIT_TYPE
  - Guided Mode
    - Info keys to specify hardware level
  - Unguided Mode
    - Start from an input COMM
    - Split down step-wise

Communication Relaxation Hints

- mpi_assert_no_any_tag
  - The process will not use MPI_ANY_TAG
- mpi_assert_no_any_source
  - The process will not use MPI_ANY_SOURCE
- mpi_assert_exact_length
  - Receive buffers must be correct size for messages
- mpi_assert_overtaking_allowed
  - All messages are logically concurrent

MPI_T Events: Callback-driven event information

- Motivation
  - PMPI does not provide access to MPI internal state information
  - MPI_T performance variables only show aggregated information
- New interface to query available runtime event types
  - Follows the MPI_T variable approach
  - No specific event types mandated
  - Event structure can be inferred at runtime
- Register callback functions to be called by the MPI runtime
  - Runtime may defer callback invocation (tool can query event time)
  - Runtime may reduce restrictions on callback functions per invocation
  - Callback can query event information individually or copy data en bloc

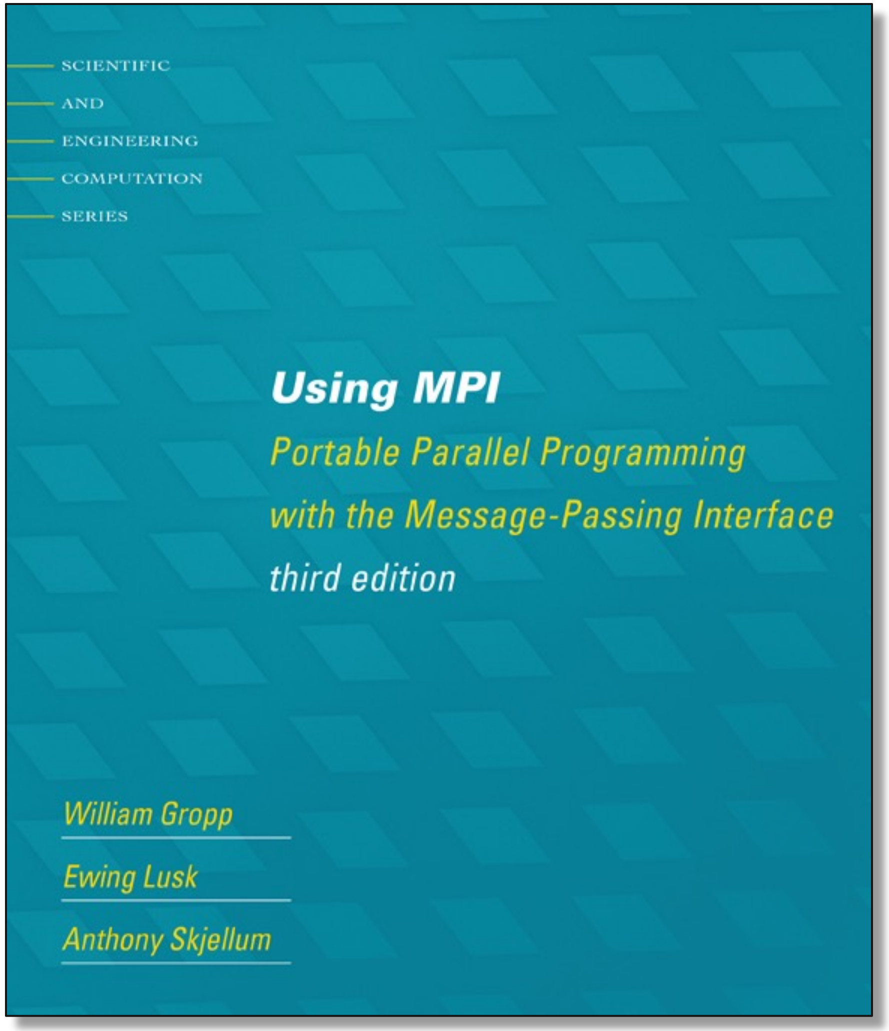
Concluding Remarks

- 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
- Your application can be next!

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://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi
  - 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



SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES

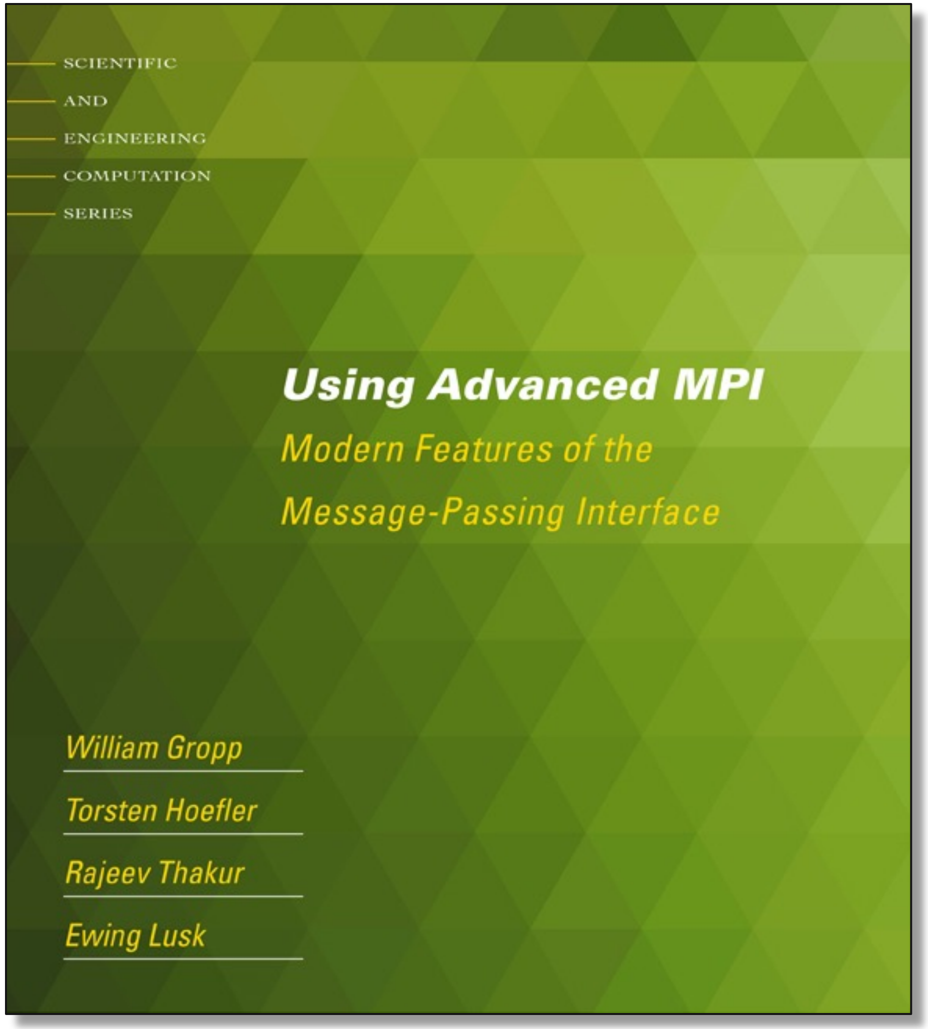
Using MPI

Portable Parallel Programming with the Message-Passing Interface

third edition

William Gropp
Ewing Lusk
Anthony Skjellum

Basic MPI



SCIENTIFIC
AND
ENGINEERING
COMPUTATION
SERIES

Using Advanced MPI

Modern Features of the Message-Passing Interface

William Gropp
Torsten Hoefler
Rajeev Thakur
Ewing Lusk

Advanced MPI, including MPI-3

MPI for Scalable Computing

Tutorial at ATPESC, August 2023

Latest slides and code examples are available at

https://anl.box.com/v/atpesc2023-mpi-tutorial

Yanfei Guo, Ken Raffenetti, Hui Zhou

Argonne National Laboratory