Argonne NATIONAL LABORATORY

MPI for Scalable Computing

Tutorial at ATPESC, July 2025

Links to latest slides and code examples are posted in Slack

Yanfei Guo, Ken Raffenetti, Hui Zhou, Mike Wilkins

Argonne National Laboratory

U.S. DEPARTMENT OF ENERGY

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
- **Mike Wilkins:** Maria Goeppert Mayer Fellow, 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 on understanding MPI concepts and use hands-on code examples to illustrate them
- Emphasis will be on issues affecting scalability and performance
- There will be code walkthroughs and hands-on exercises

Outline

- Morning
  - Introduction
  - MPI Fundamentals Refresher
  - Avoiding Unintended Synchronization
  - Collective Communication
  - Derived Datatypes
  - One-sided Communication (RMA)
- Afternoon
  - Hybrid programming
    - MPI + threads
    - MPI + GPUs
  - New features in MPI
    - MPI Sessions
    - Large Count
    - ABI

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
- 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 new features including MPI + threads, MPI-I/O, remote memory access functionality, C++ bindings, 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) several new features
- MPI-3.1 (2015) minor corrections and features
- MPI-4 (June 2021) several new features
- MPI-4.1 (November 2023) minor corrections and features,
- MPI-5.0 (June 2025) standardization of application binary interface (ABI)
- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both PDF and HTML

Overview of 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

Overview of MPI-4

- Major new features and changes
  - Persistent Collectives
  - Partitioned Communication
  - Sessions
  - Large Count
  - Error Handling Improvement
  - Topology-aware Communicator Creation
- MPI 4.1 added corrections and some deprecations
  - MPI_TYPE_SIZE_X, etc. deprecated in favor or “large count” versions (MPI_TYPE_SIZE_C)
  - mpif.h Fortran binding deprecated

Overview of MPI-5

- Major new features
  - Application Binary Interface (ABI) for C interface. Build applications once and run with any compliant implementation. Improved support for packaging and third-party libraries/language bindings.

Important considerations while using MPI

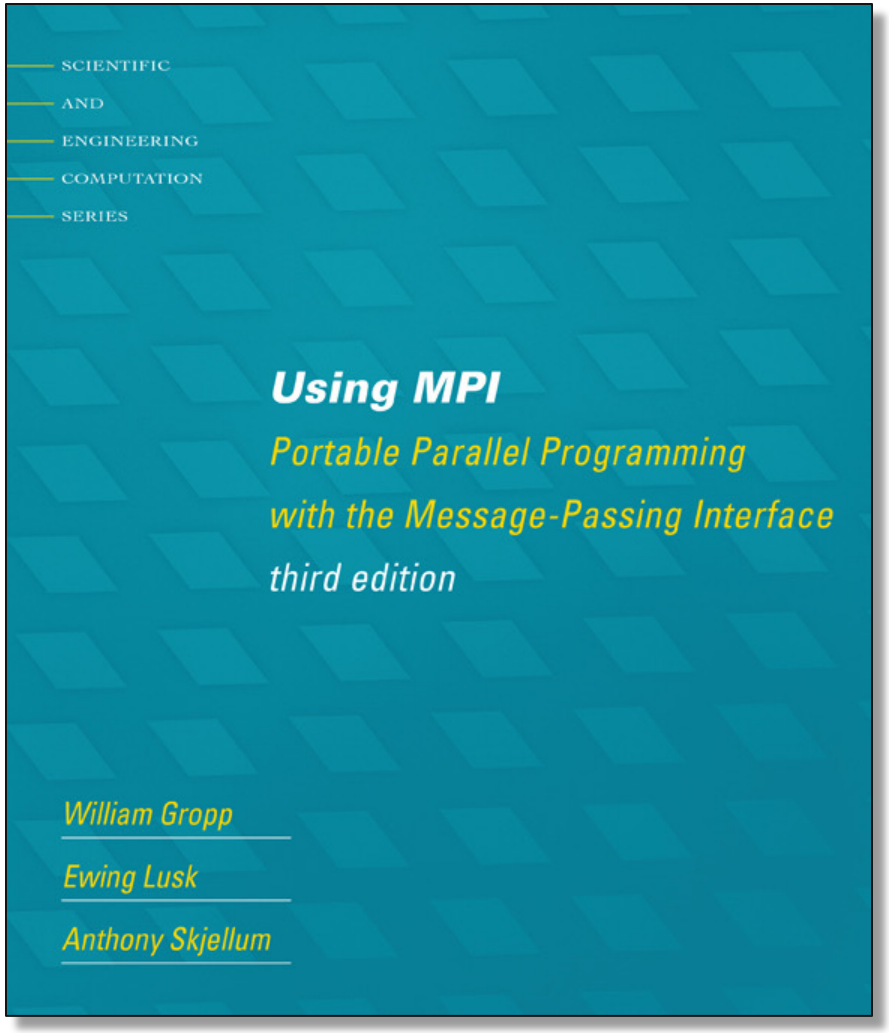
- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs

Web Pointers

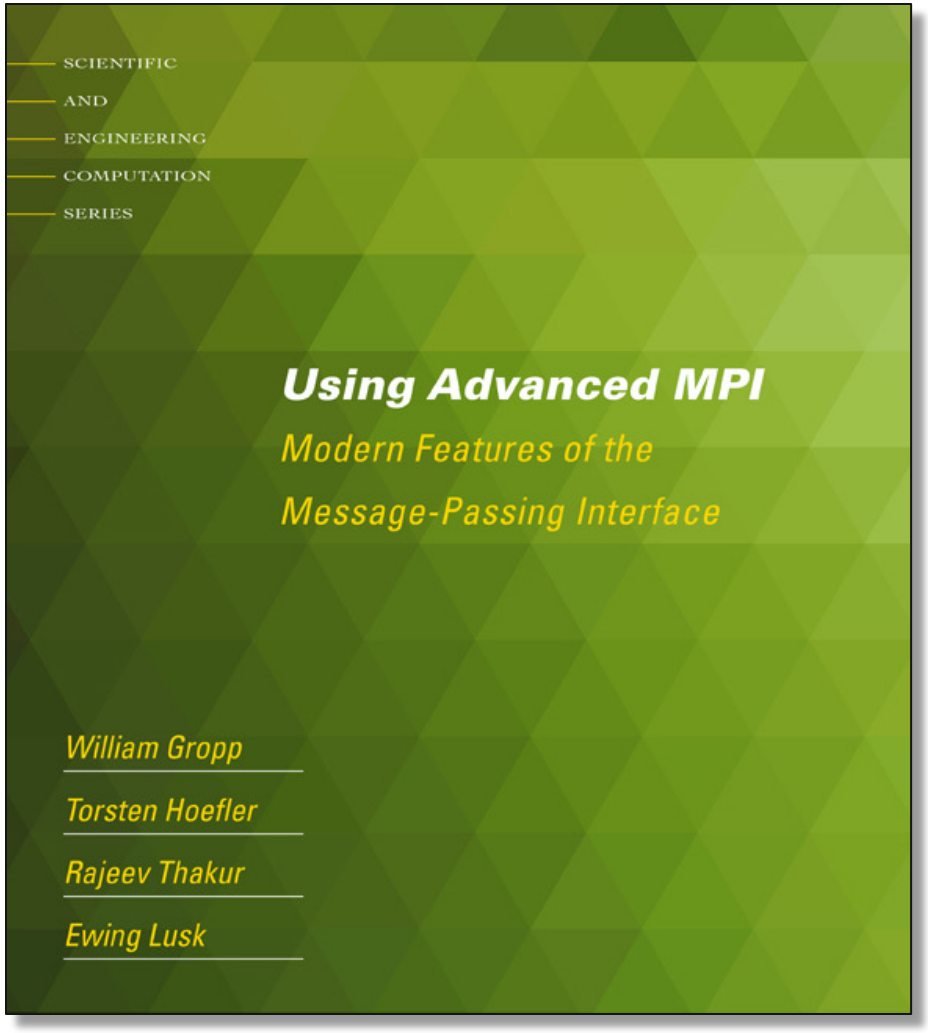
- MPI Standard : http://www.mpi-forum.org/docs/
- MPI Forum : http://www.mpi-forum.org/

- 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 Spectrum MPI, Cray MPICH, ParaStation MPI, …
- General MPI Education
  - https://rookiehpc.org/mpi/
  - https://mpitutorial.com/tutorials/

Tutorial Books on MPI



Basic MPI



Advanced MPI, including MPI-3

Approach in this Tutorial

- Example driven
  - Small examples used to illustrate specific features
  - Running examples used throughout the tutorial

MPI Fundamentals Refresher

Communicators, Messages, Blocking vs Nonblocking

MPI Communicators

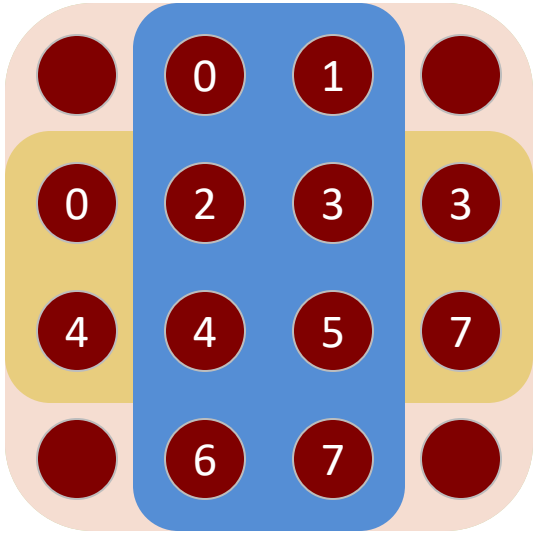
- MPI processes are members of one or more groups
  - Each group can have multiple colors (sometimes called context)
  - *Group + color == communicator (it is like a name for the group)*
  - When an MPI application starts, the group of all processes is initially given a predefined name called **`MPI_COMM_WORLD`**
    - The same group can have many names, but simple programs do not have to worry about multiple names
- A process is identified by a unique number within each communicator, called *rank*
  - For two different communicators, the same process can have two different ranks: the meaning of a “rank” is only defined when you specify the communicator

Communicators

```
% mpiexec -n 16 ./hello
```

Communicators do not need to contain all processes in the system

Every process in a communicator has an ID called a “rank”



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

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

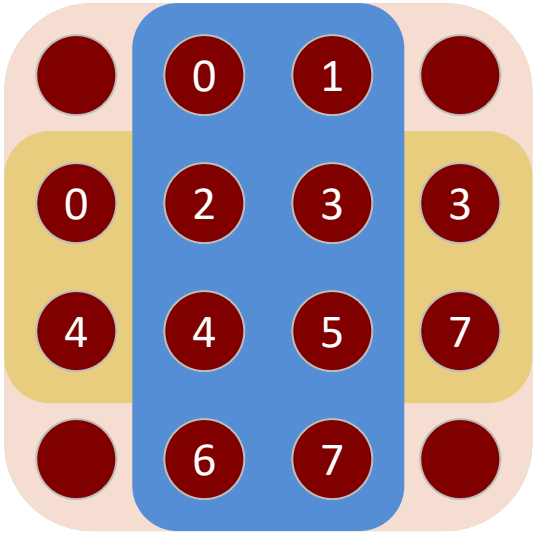
The same process might have different ranks in different communicators

Communicators can be created “by hand” or using tools provided by MPI (not discussed in this tutorial)

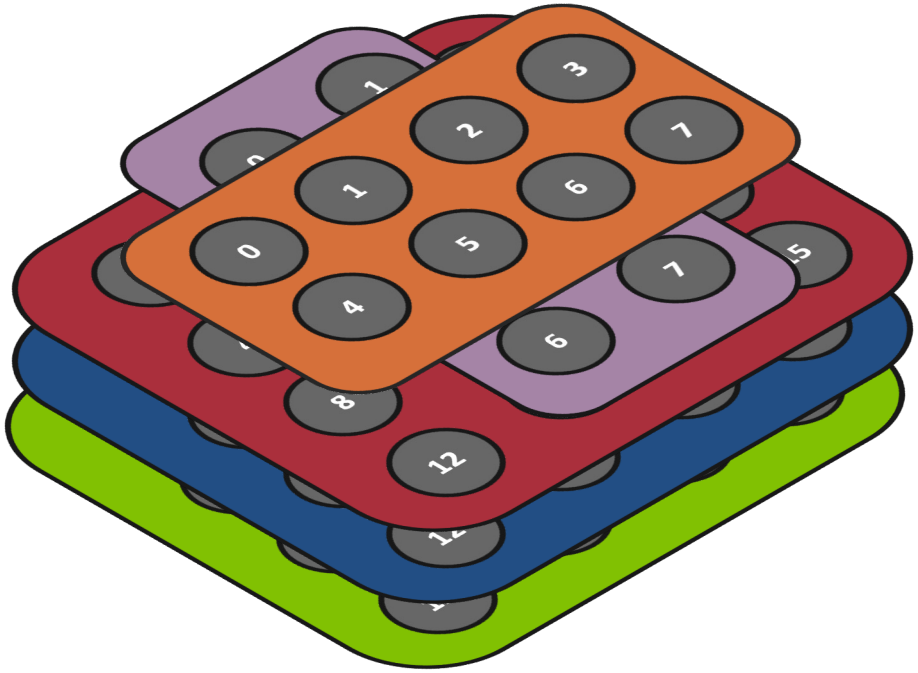
Simple programs typically only use the predefined communicator `MPI_COMM_WORLD`

Communicators

Think of independent communication layers over a group of processes



Messages in one layer will not affect messages in another



Message-based Communication

- Point-to-point messaging in MPI
  - One process sends a copy of its data to another process which receives it
  - Message boundaries are well-defined, unlike pipes or sockets
- Communication requires the following information:
  - Sender has to know:
    - Whom to send the data to (receiver process's rank in communicator)
    - What kind of data to send (100 integers or 200 characters, etc)
    - A user-defined "tag" for the message (additional context for message)
  - Receiver "might" have to know:
    - Who is sending the data (OK if the receiver does not know; in this case sender rank will be **`MPI_ANY_SOURCE*`**, meaning anyone can send)
    - What kind of data is being received (partial information is OK: I might receive *up to* 1000 integers)
    - What the user-defined "tag" of the message is (OK if the receiver does not know; in this case tag will be **`MPI_ANY_TAG*`**)

*use of wildcards may incur a performance penalty

More Details on Describing Data for Communication

- 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`
  - `C++ std::complex<float>` → `MPI_CXX_FLOAT_COMPLEX`
  - `Fortran integer` → `MPI_INTEGER`
- More complex datatypes are also possible (covered later in tutorial)
- The "count" in `MPI_SEND` and `MPI_RECV` refers to how many datatype elements should be communicated

Message Ordering

- MPI specifies that messages are **nonovertaking**
  - Messages sent to the same destination with the same matching information (comm, rank, tag) will be matched in the order they were issued by the sender
- Messages match in order but are not required to complete in order
  - Completion order could be affected by:
    - message size
    - network routing
    - user-level wait behavior

Blocking vs. Nonblocking Communication

- **`MPI_SEND/MPI_RECV`** are blocking communication calls
  - Return of the routine implies completion
  - When these calls return the memory locations used in the message transfer can be safely accessed
  - For “send” completion implies buffer can be reused/modified
    - Modifications will not affect data intended for the receiver
  - For “receive” completion implies buffer can be read
- **`MPI_ISEND/MPI_IRECV`** are nonblocking variants
  - Routine returns immediately – completion is separately tested/waited
  - Routines are local. They do not depend on the action of another process.

Blocking vs. Nonblocking Completion Semantics

- A send has completed when the user supplied buffer can be reused
- Just because the send completes does not mean that the receive has completed
  - Message may be buffered by the system
  - Message may still be in transit

```
*buf = 3;
MPI_Send(buf, 1, MPI_INT …)
*buf = 4; /* OK, receiver will always
             receive 3 */
```

```
*buf = 3;
MPI_Isend(buf, 1, MPI_INT …)
*buf = 4; /* NOT OK, receiver may get
             3, 4, or garbage */
MPI_Wait(…);
```

- A receive has completed when the user supplied buffer can be read

```
*buf = -1;
MPI_Recv(buf, 1, MPI_INT …)
assert(*buf >= 0); /* OK */
```

```
*buf = -1;
MPI_Irecv(buf, 1, MPI_INT …)
assert(*buf >= 0); /* NOT OK */
MPI_Wait(…);
assert(*buf >= 0); /* OK */
```

Nonblocking Communication Overlap

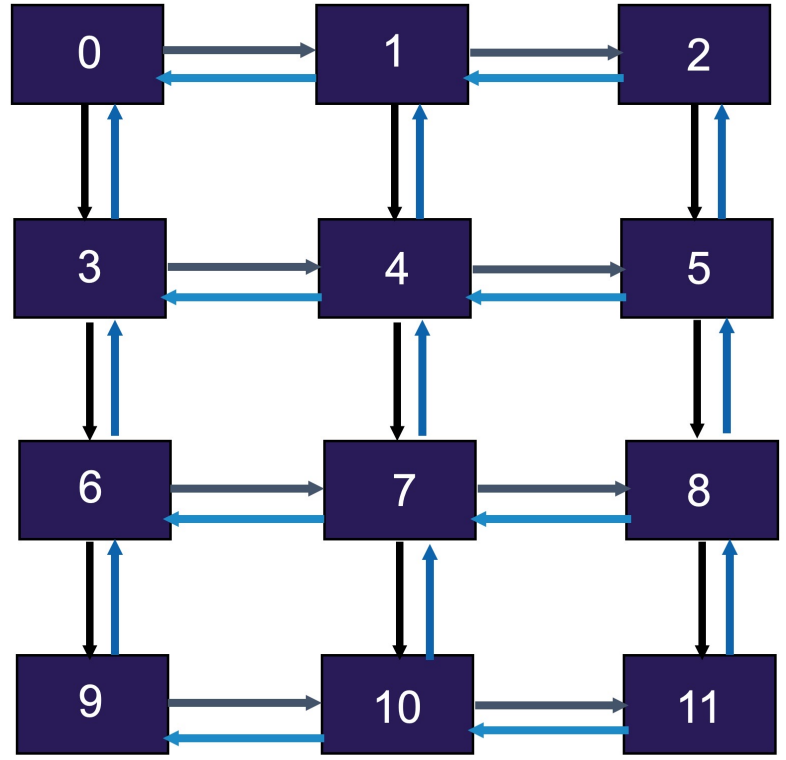
- Nonblocking communication allows for *overlap*
  - Overlap with other communication
  - Overlap with computation

- Overlap with communication
  - Useful for deadlock avoidance. No need to carefully order operations.
  - MPI can better utilize communication resources for better performance
- Overlap with computation
  - If application threads are busy doing computation, communication might not progress in the background. It is not required in the MPI specification. Referred to as the “weak progress model” of MPI.
  - Communication progress characteristics are dependent on many factors, but not limited to:
    - System architecture
    - MPI library configuration and implementation
    - Runtime parameters
    - Process locality
    - Communication arguments, e.g. datatypes
    - …
  - If possible, periodically call MPI_Test[any|some|all] during computation phase to ensure progress

Costs of Unintended Synchronization

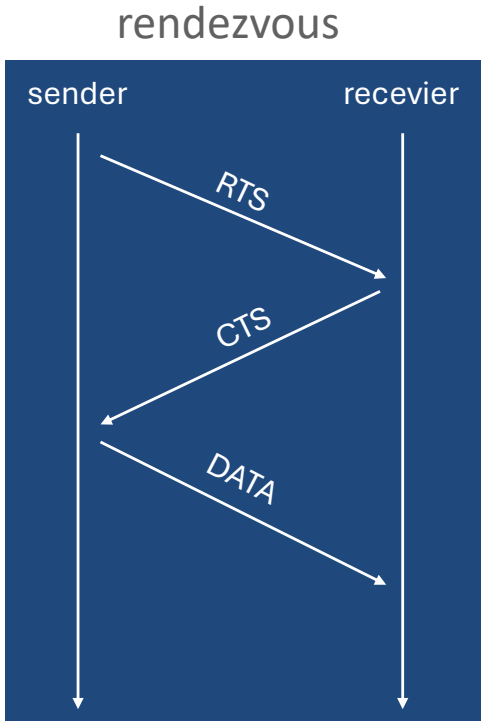
Unintended Synchronization

- MPI send and receive messaging combines data movement and synchronization
  - In many typical data exchange patterns, unnecessary synchronization can lead to poor performance
  - An example of unnecessary synchronization is sending and receiving messages one-at-a-time using blocking communication primitives.
  - Commonly used rendezvous protocol dictates that sends only complete with involvement (read: synchronization) of the receiver



Neighbor Exchange Example Code

- https://github.com/pmodels/mpi-tutorial-examples/tree/main/unintended-sync
  - 4 versions of 2-dimensional nearest neighbor exchange
    - NOTE: examples are hardcoded for n=12
  - Build/run them and look at performance characteristics of each



2d exchange (blocking)

- Unsafe use of blocking send. Risk of deadlock!

```
MPI_Send(sbufsouth, COUNT, MPI_INT, south, 0, MPI_COMM_WORLD);
MPI_Send(sbufeast, COUNT, MPI_INT, east, 0, MPI_COMM_WORLD);
MPI_Send(sbufnorth, COUNT, MPI_INT, north, 0, MPI_COMM_WORLD);
MPI_Send(sbufwest, COUNT, MPI_INT, west, 0, MPI_COMM_WORLD);

MPI_Recv(rbufnorth, COUNT, MPI_INT, north, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Recv(rbufwest, COUNT, MPI_INT, west, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Recv(rbufsouth, COUNT, MPI_INT, south, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Recv(rbufeast, COUNT, MPI_INT, east, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

2d exchange (blocking + ordering)

- Fixes deadlock, but communication is sequential. Low resource utilization.

```
if (south ≠ MPI_PROC_NULL) {
    MPI_Send(sbufsouth, COUNT, MPI_INT, south, 0, MPI_COMM_WORLD);
}
if (north ≠ MPI_PROC_NULL) {
    MPI_Recv(rbufnorth, COUNT, MPI_INT, north, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
if (east ≠ MPI_PROC_NULL) {
    MPI_Send(sbufeast, COUNT, MPI_INT, east, 0, MPI_COMM_WORLD);
}
if (west ≠ MPI_PROC_NULL) {
    MPI_Recv(rbufwest, COUNT, MPI_INT, west, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
<...>
```

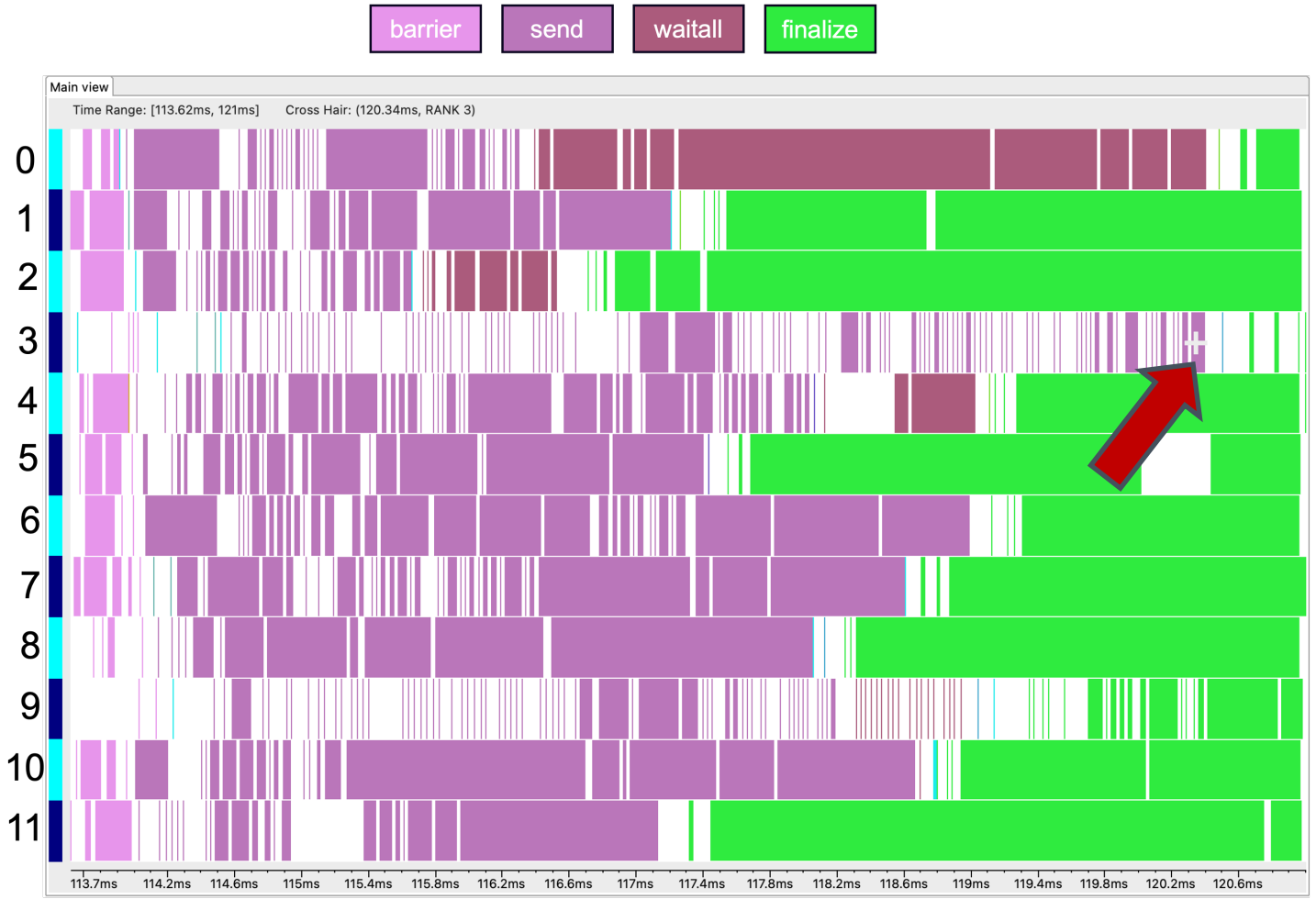
2d exchange (nonblocking receives)

- Better utilization, but congestion at receivers can delay sends, which propagates through the exchange.

```
MPI_Irecv(rbufnorth, COUNT, MPI_INT, north, 0, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(rbufwest, COUNT, MPI_INT, west, 0, MPI_COMM_WORLD, &reqs[1]);
MPI_Irecv(rbufsouth, COUNT, MPI_INT, south, 0, MPI_COMM_WORLD, &reqs[2]);
MPI_Irecv(rbufeast, COUNT, MPI_INT, east, 0, MPI_COMM_WORLD, &reqs[3]);

MPI_Send(sbufsouth, COUNT, MPI_INT, south, 0, MPI_COMM_WORLD);
MPI_Send(sbufeast, COUNT, MPI_INT, east, 0, MPI_COMM_WORLD);
MPI_Send(sbufnorth, COUNT, MPI_INT, north, 0, MPI_COMM_WORLD);
MPI_Send(sbufwest, COUNT, MPI_INT, west, 0, MPI_COMM_WORLD);

MPI_Waitall(4, reqs, MPI_STATUSES_IGNORE);
```

Timeline of nonblocking receive exchange

- Some processes finish much earlier than others. Long delays the result of congestion propagation.



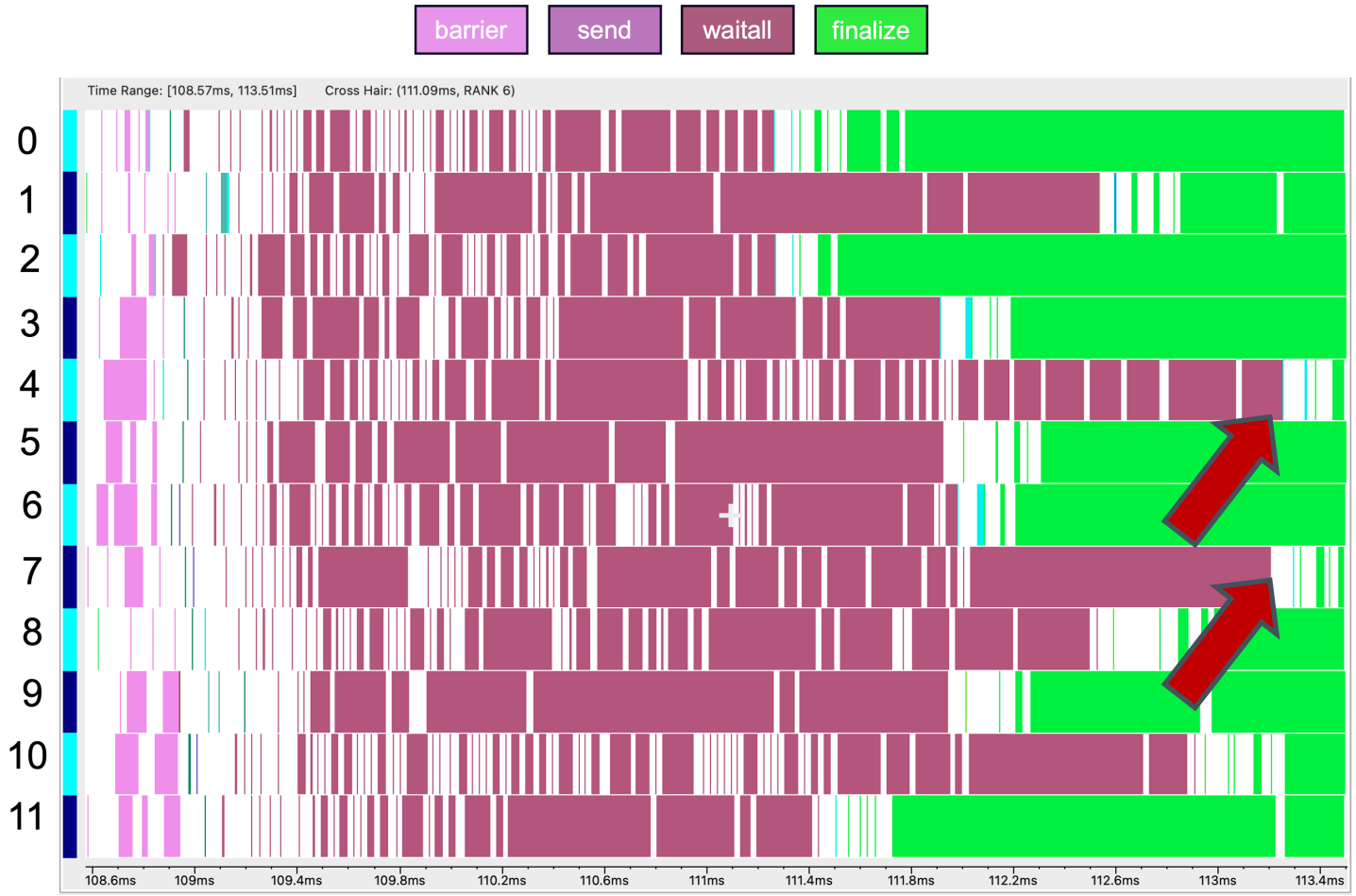
2d exchange (fully nonblocking)

- Best utilization. No arbitrary or unintended synchronization.

```
MPI_Irecv(rbufnorth, COUNT, MPI_INT, north, 0, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(rbufwest, COUNT, MPI_INT, west, 0, MPI_COMM_WORLD, &reqs[1]);
MPI_Irecv(rbufsouth, COUNT, MPI_INT, south, 0, MPI_COMM_WORLD, &reqs[2]);
MPI_Irecv(rbufeast, COUNT, MPI_INT, east, 0, MPI_COMM_WORLD, &reqs[3]);

MPI_Isend(sbufsouth, COUNT, MPI_INT, south, 0, MPI_COMM_WORLD, &reqs[4]);
MPI_Isend(sbufeast, COUNT, MPI_INT, east, 0, MPI_COMM_WORLD, &reqs[5]);
MPI_Isend(sbufnorth, COUNT, MPI_INT, north, 0, MPI_COMM_WORLD, &reqs[6]);
MPI_Isend(sbufwest, COUNT, MPI_INT, west, 0, MPI_COMM_WORLD, &reqs[7]);

MPI_Waitall(8, reqs, MPI_STATUSES_IGNORE);
```

Timeline of fully nonblocking exchange

- Interior processes take longer because they have the most neighbors. More balanced completion. Overall communication time reduced 30%.



Takeaway: 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
- Gives MPI the best chance to utilize communication resources effectively and results in best performance
  - High amounts of memory and network bandwidth in modern hardware. Do not waste it!

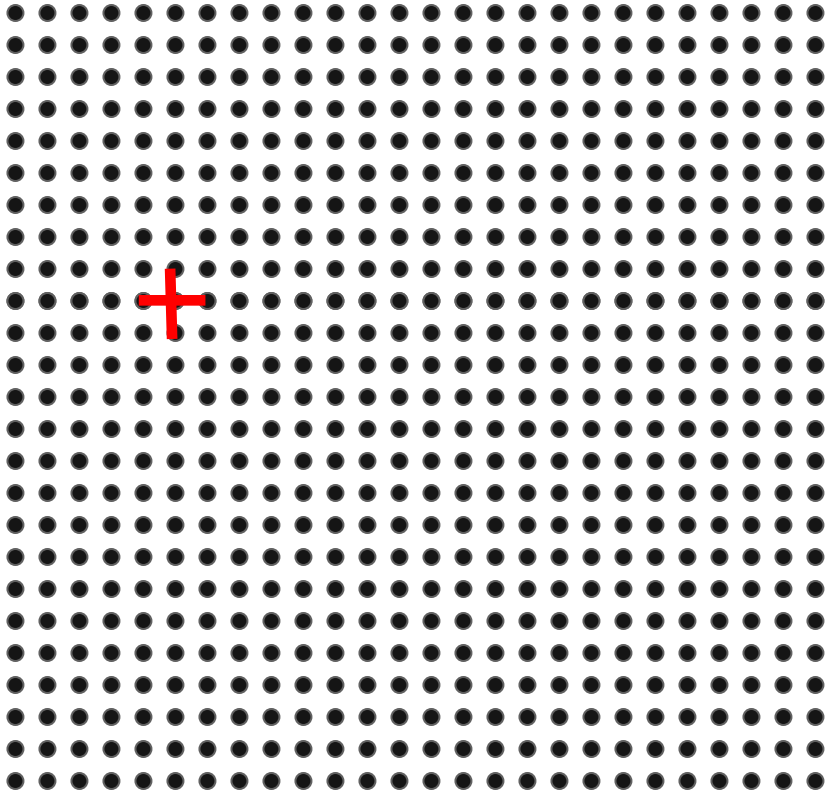
Running Example: Stencil

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.
- Example uses “fixed boundary conditions”. Heat is not lost through the boundary, but it is no longer measured by the system.



MPI Examples

- https://github.com/pmodels/mpi-tutorial-examples/
- Clone repo in your home or project directory
- Allocate or submit job using reservation on Aurora
  - `qsub -q ATPESC -l select=1,walltime=1:00:00,filesystems=home -A ATPESC2025`

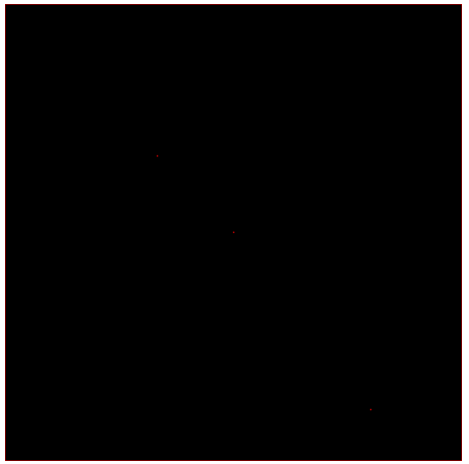
Example: Stencil

- *hands-on-examples/stencil_serial.c*
- 2D stencil code in single process

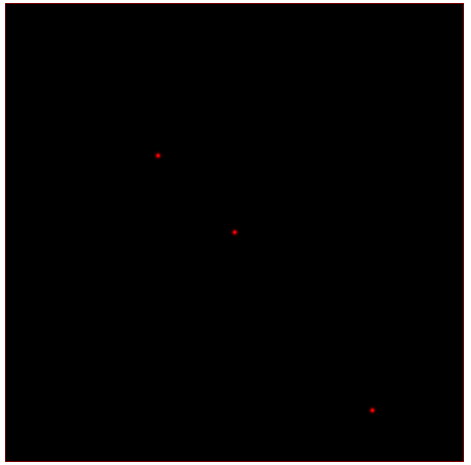
```
# mpicc –o stencil_serial stencil_serial.c
# qsub -q ATPESC –l
select=1,walltime=1:00:00,filesystems=home –A
ATPESC2025 -I # Aurora
```

```
# ./stencil_serial <domain size> <iterations>
```

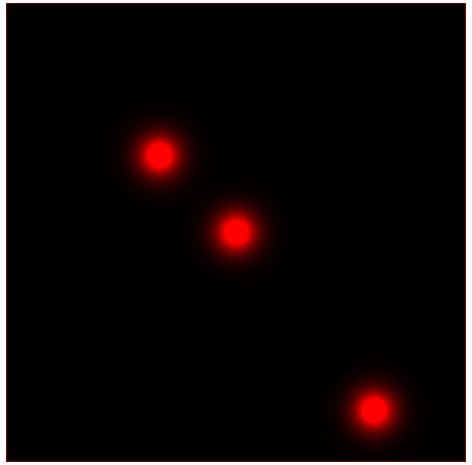
```
# ./stencil_serial 1000 10
 last heat: 30.000000
```



iter=10



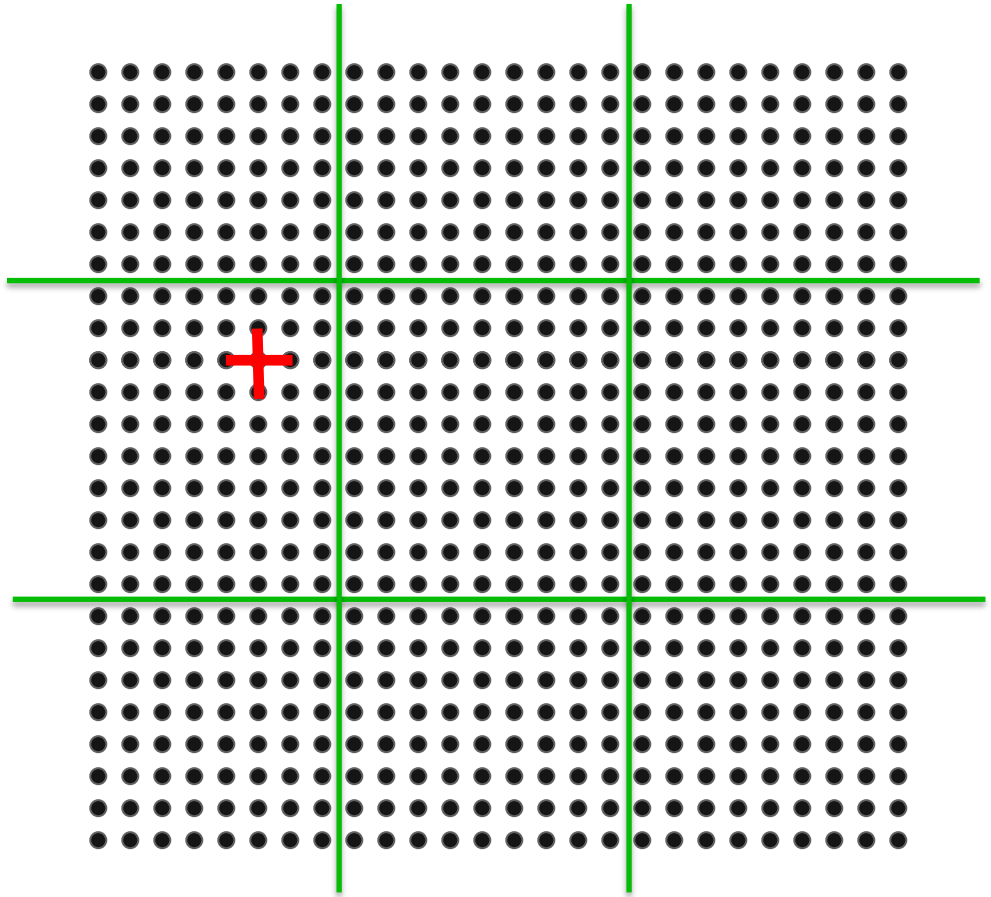
iter=100



iter=10000

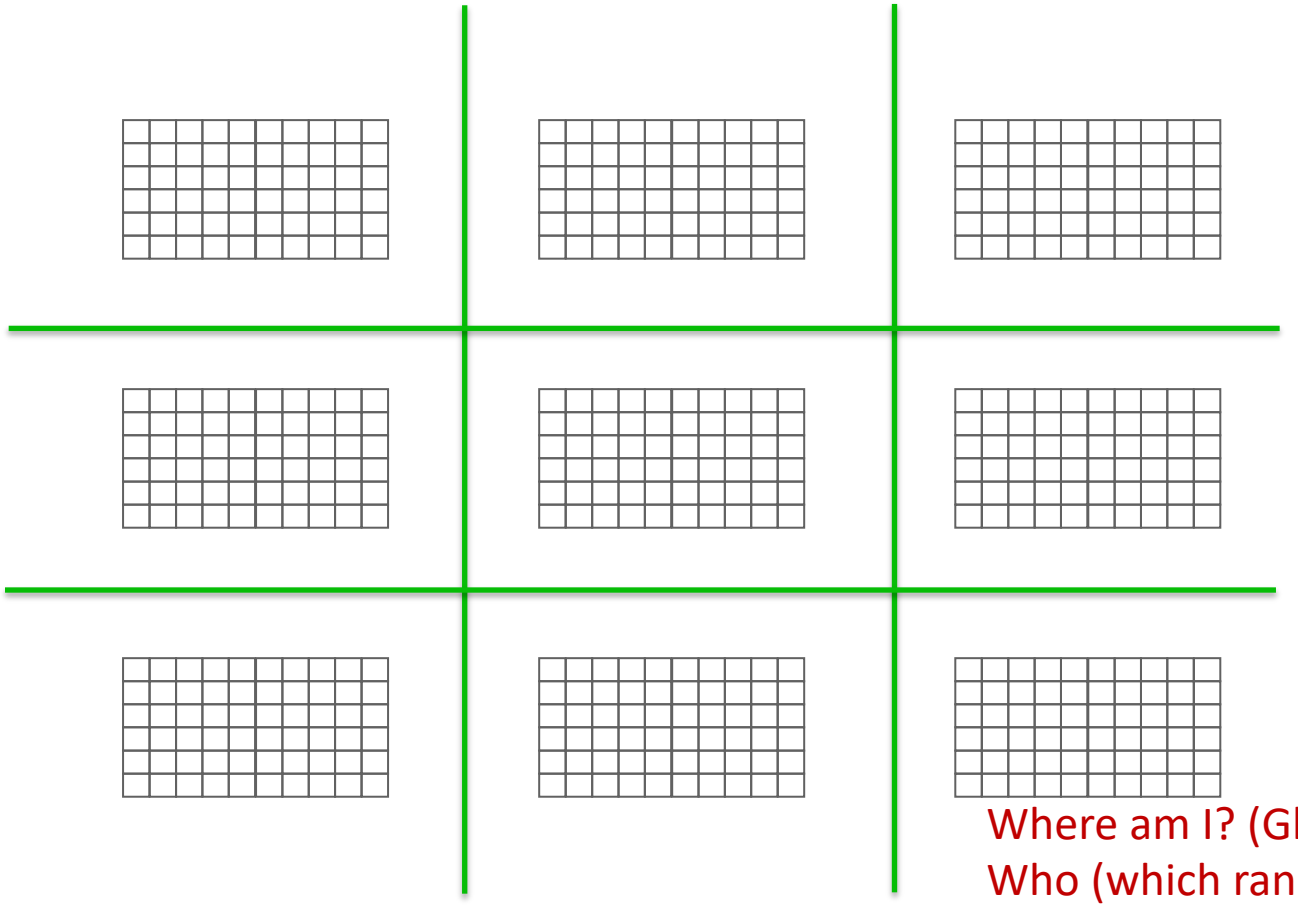
The Global Data Structure

- 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



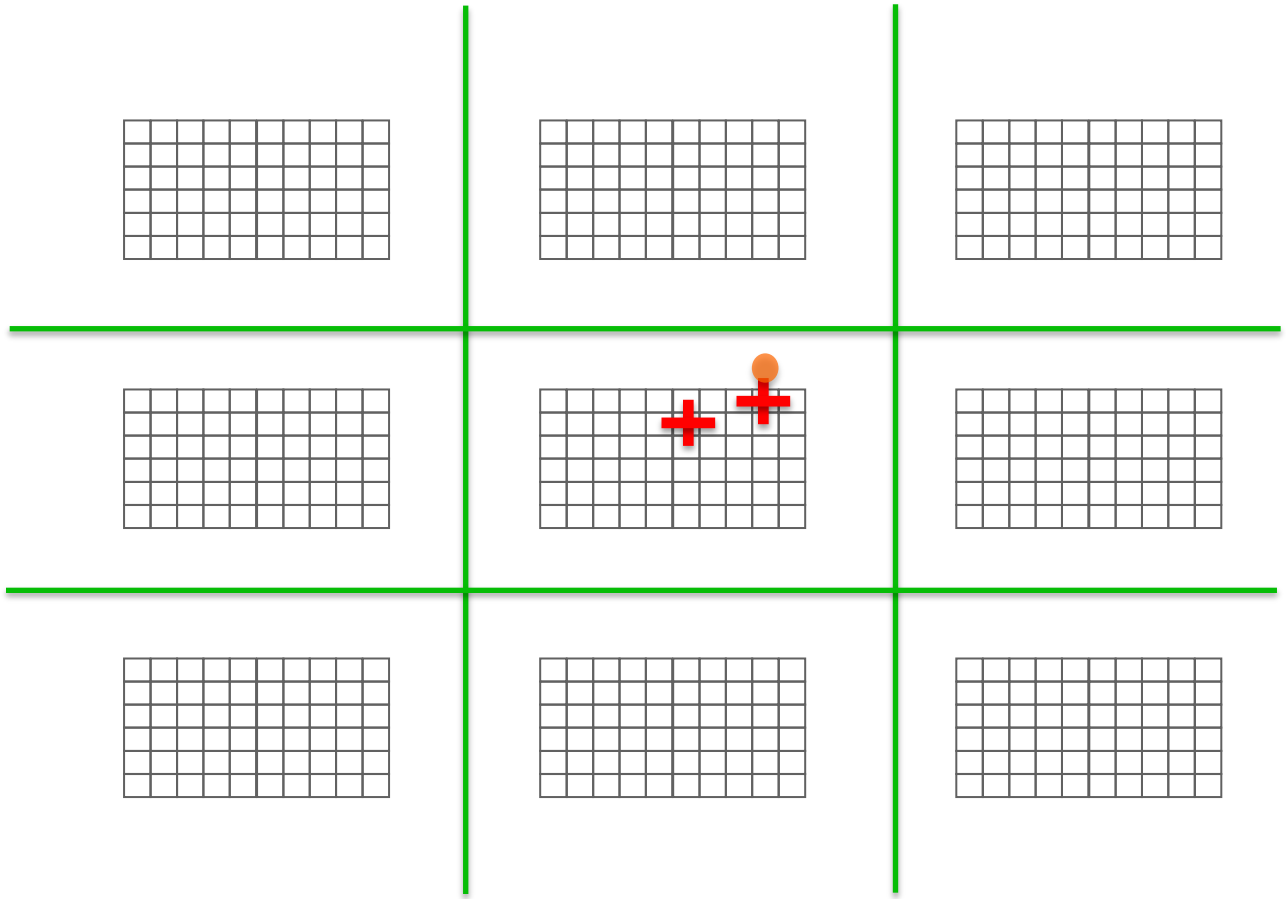
Domain Decompositioin

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



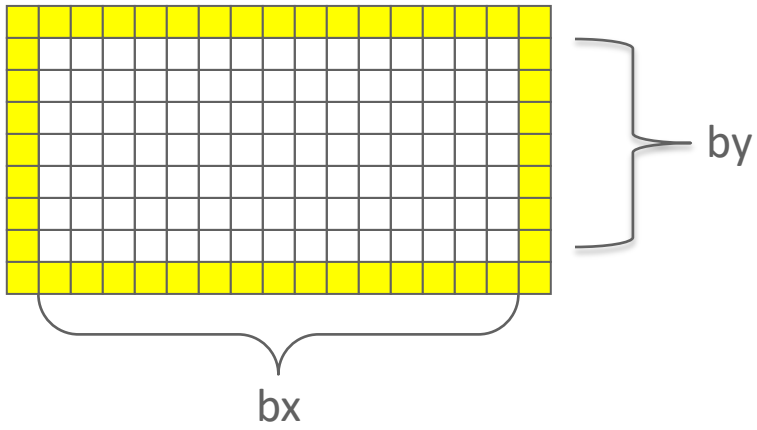
Where am I? (Global offset)
Who (which ranks) are my neigbhors?
Use MPI_PROC_NULL for exterior boundaries

Necessary Data Transfers



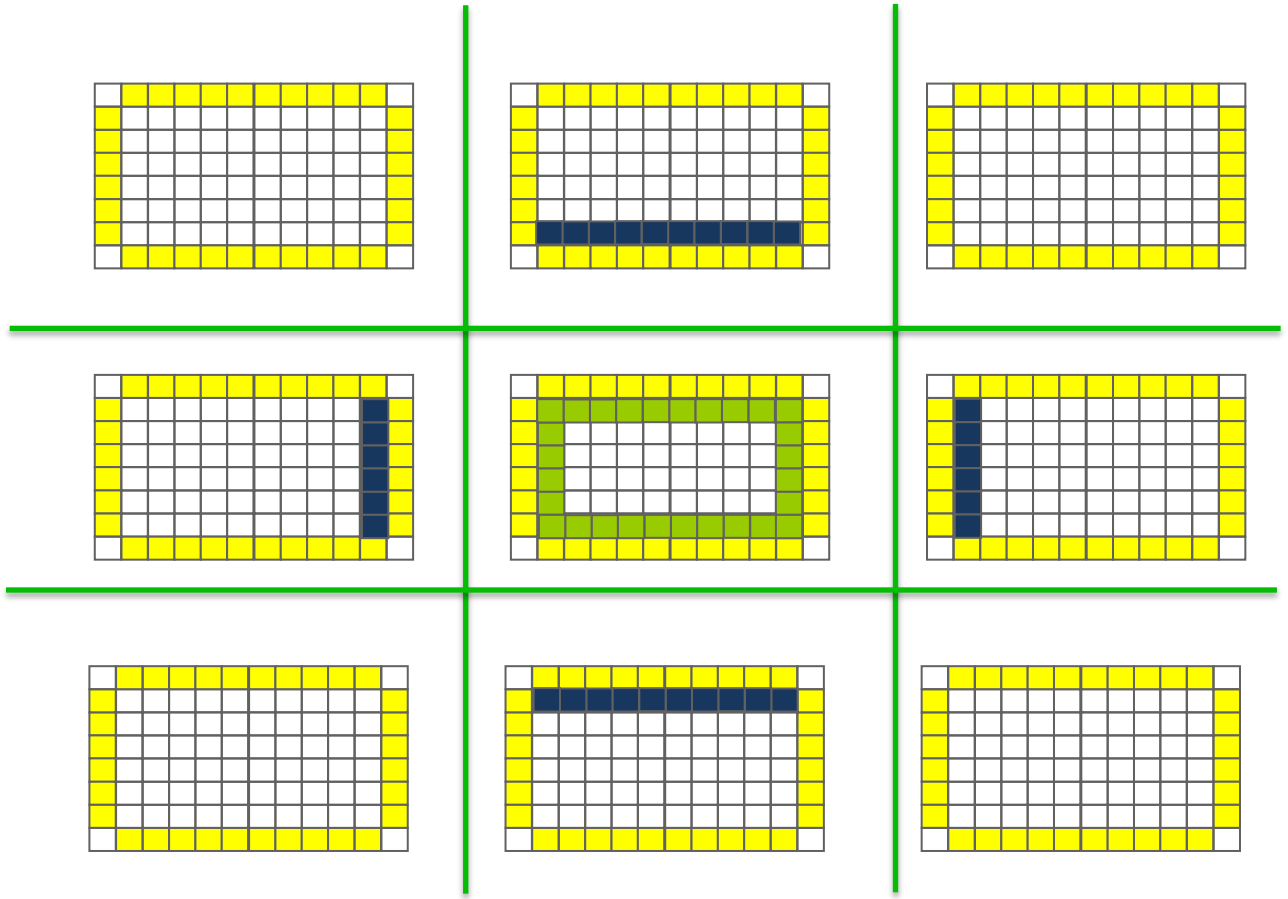
The Local Data Structure

- 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)



Necessary Data Transfers

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



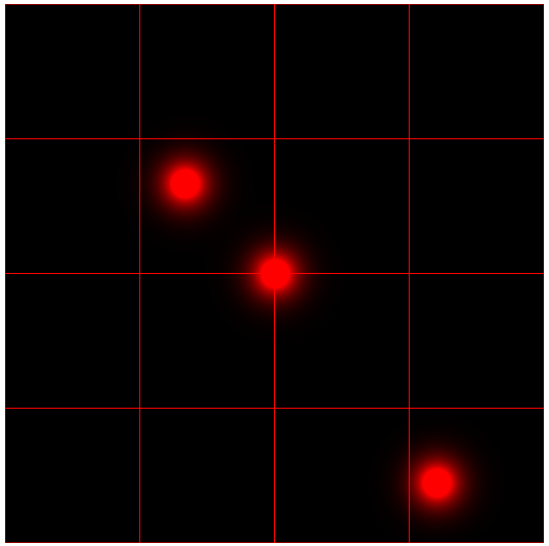
Example: Stencil with Nonblocking Send/recv

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

```
# qsub -q ATPESC -l select=1,walltime=1:00:00,filesystems=home –A
ATPESC2025 -I # Aurora
```

```
For most systems (or local machine)
# mpiexec -n <nproc> ./stencil <domain size> <iterations> <px> <py>

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

```
% mpiexec –n 16 ./stencil 1000 10 4 4
[0] last heat: 30.000000
```



iter=10000

Concluding Remarks

- Parallelism is critical today. Necessary to achieve performance improvement with the modern hardware.
- MPI is an industry standard model for parallel programming
  - Many implementations of MPI exist (both commercial and community supported)
  - Virtually every system in the world supports MPI
- Gives user explicit control on data management
- Widely used by many scientific applications with great success

Argonne NATIONAL LABORATORY

Collectives: Blocking and Nonblocking

U.S. DEPARTMENT OF ENERGY

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 collectives, improving both simplicity and efficiency.

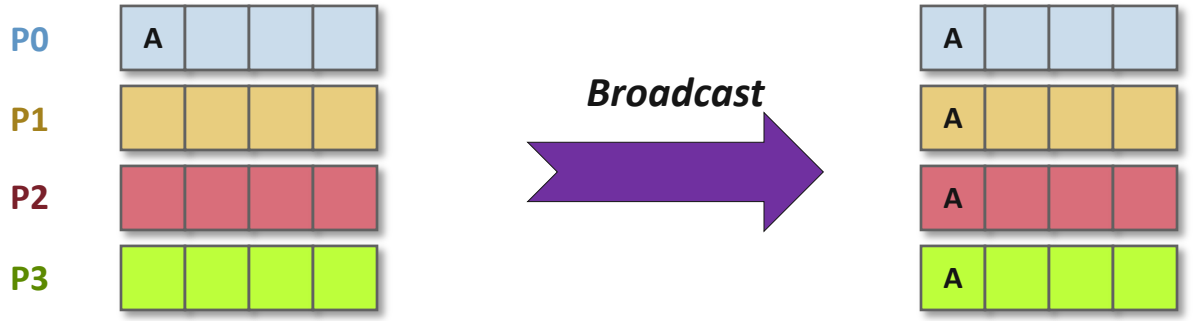
MPI Collective Communication Basics

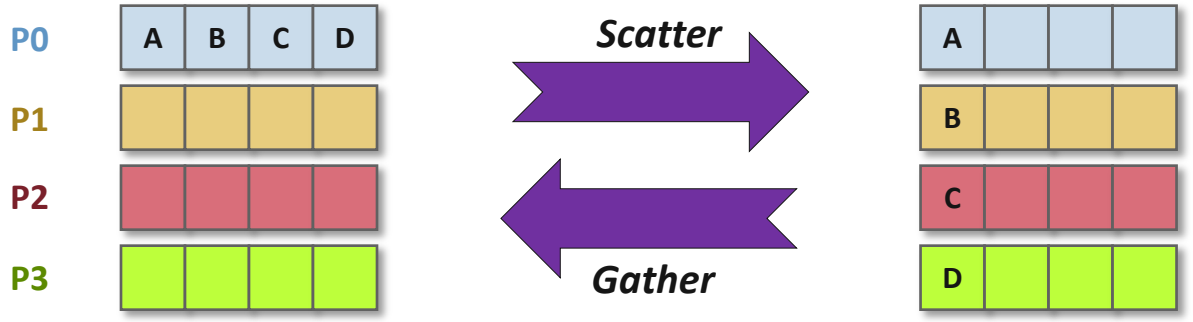
- Communication and computation is coordinated among a group of processes in a communicator
- Tags are not used
  - Use multiple communicators to overlap collectives
- Nonblocking collective operations added in MPI-3
- Three classes of operations: synchronization, data movement, collective computation

Types of Collectives: Synchronization

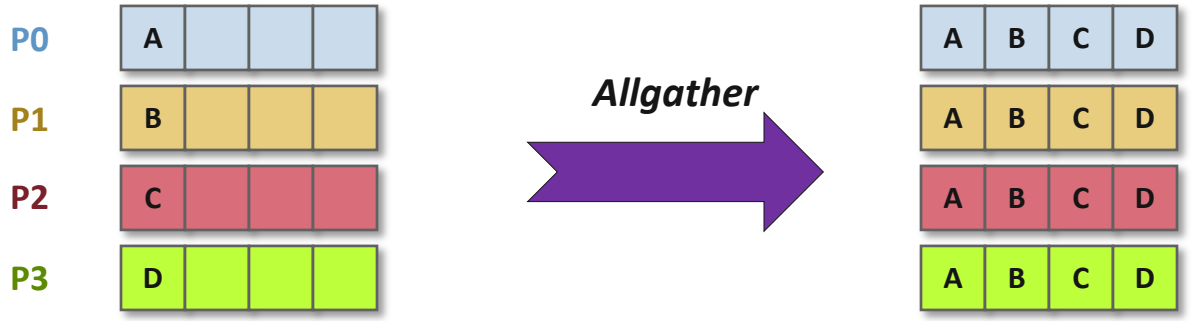
- `MPI_BARRIER(comm)`
  - Blocks until all processes in the group of the communicator `comm` call it
  - A process cannot get out of the barrier until all other processes have reached barrier

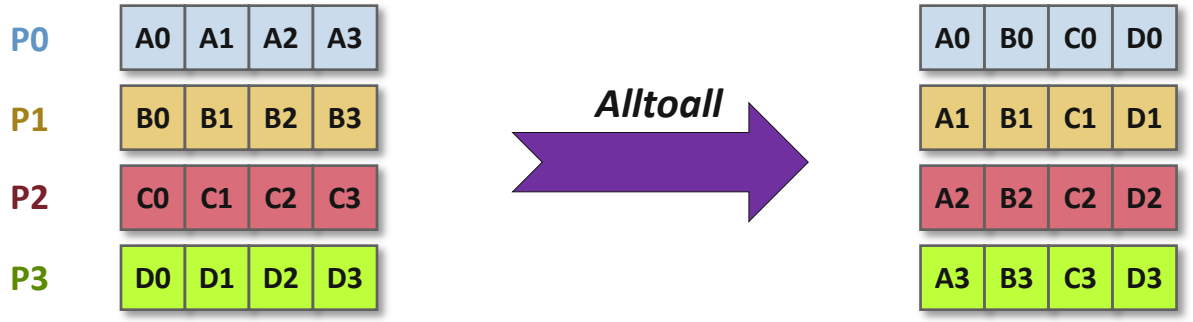
Types of Collectives: One-Sided/Unbalanced Data Movement



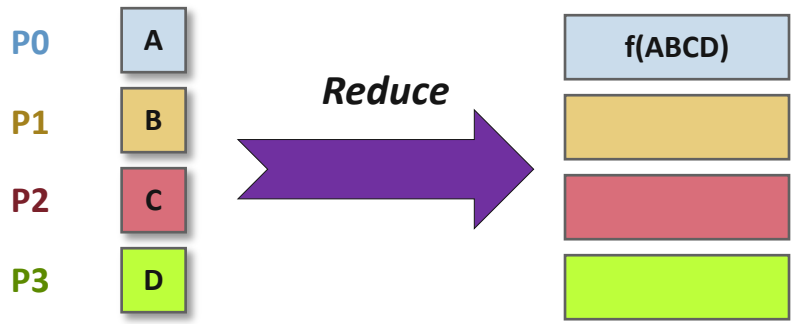


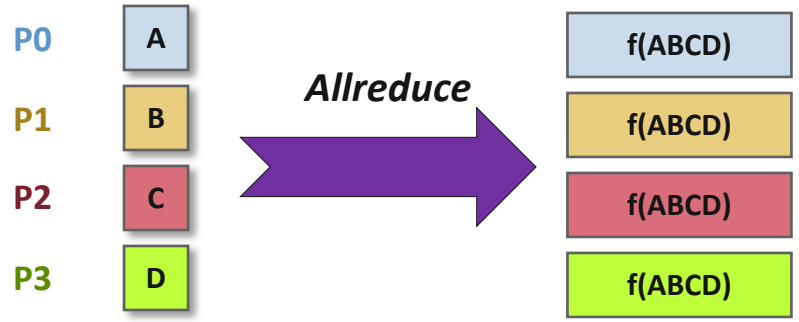
Types of Collectives: Two-Sided/Balanced Data Movement

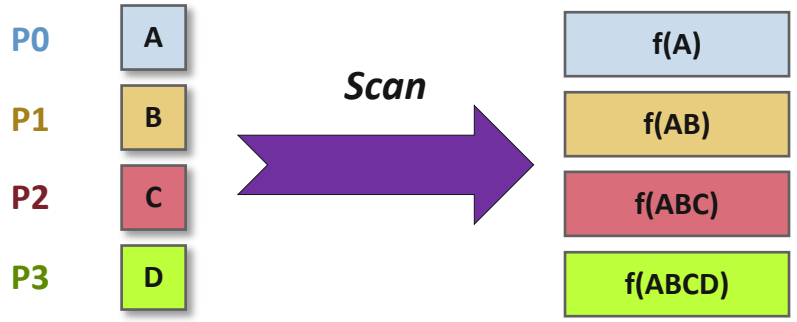


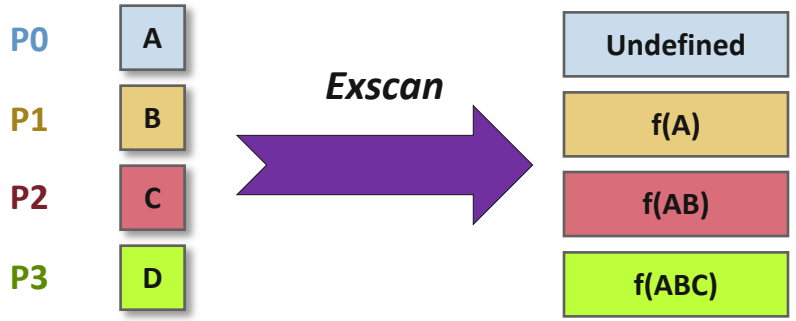


Types of Collectives: Collective Computation









MPI Collective Routine Nomenclature

- "**All**" versions deliver results to all participating processes
  - E.g., **MPI_ALLREDUCE**
- "***V***" versions (stands for vector) allow the chunks to have different sizes
  - E.g., **MPI_ALLTOALLV**
  - **AllTOALL**"***W***" allows different datatypes!
- **MPI_ALLREDUCE**, **MPI_REDUCE**, **MPI_REDUCESCATTER**, and **MPI_SCAN** take both built-in and user-defined computation operations (i.e., "ops")

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 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: Stencil with Blocking Collectives

- *blocking_coll/stencil.c*

- Use *MPI_Alltoallv* to do data exchanges all in one MPI call!

```
/* exchange data with neighbors */
MPI_Alltoallv(exchange_sbuf, send_sizes, send_start_indices, MPI_DOUBLE, exchange_rbuf,
              recv_sizes, recv_start_indices, MPI_DOUBLE, MPI_COMM_WORLD);
```

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 (e.g., MPI_Bcast -> MPI_Ibcast)
  - System noise/imbalance resiliency
  - **Semantic advantages**

Nonblocking Collective Communication Basics

- Semantics
  - Function returns immediately
  - 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

Semantic Advantages of Nonblocking Collectives

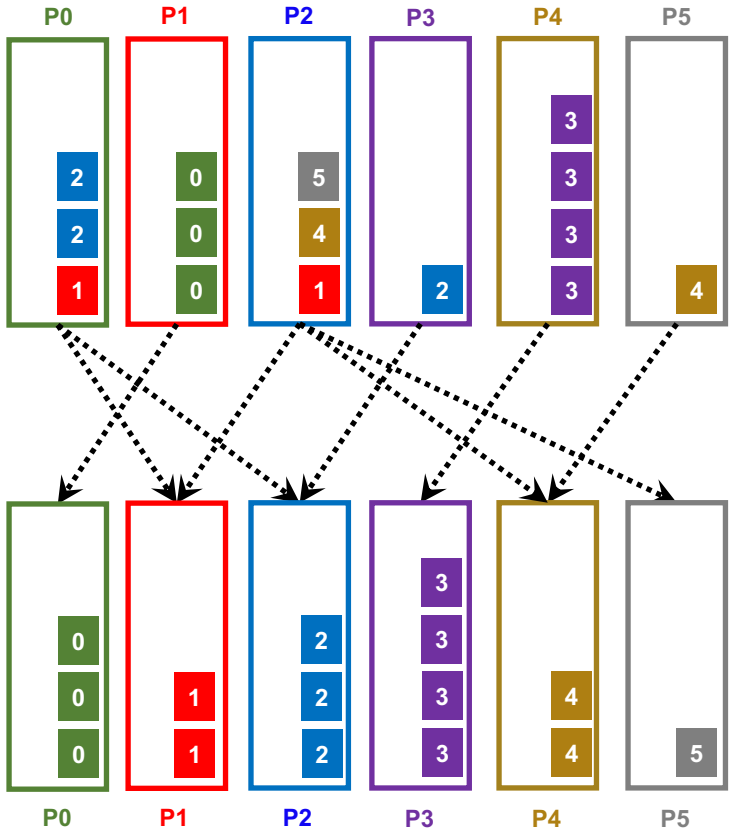
- Asynchronous progression
  - Pipelining + Communication/Computation Overlap
- Overlapping collectives
- Decouple data transfer and synchronization

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** non-collectively but **synchronize** collectively!

A Semantics Example: DSDE

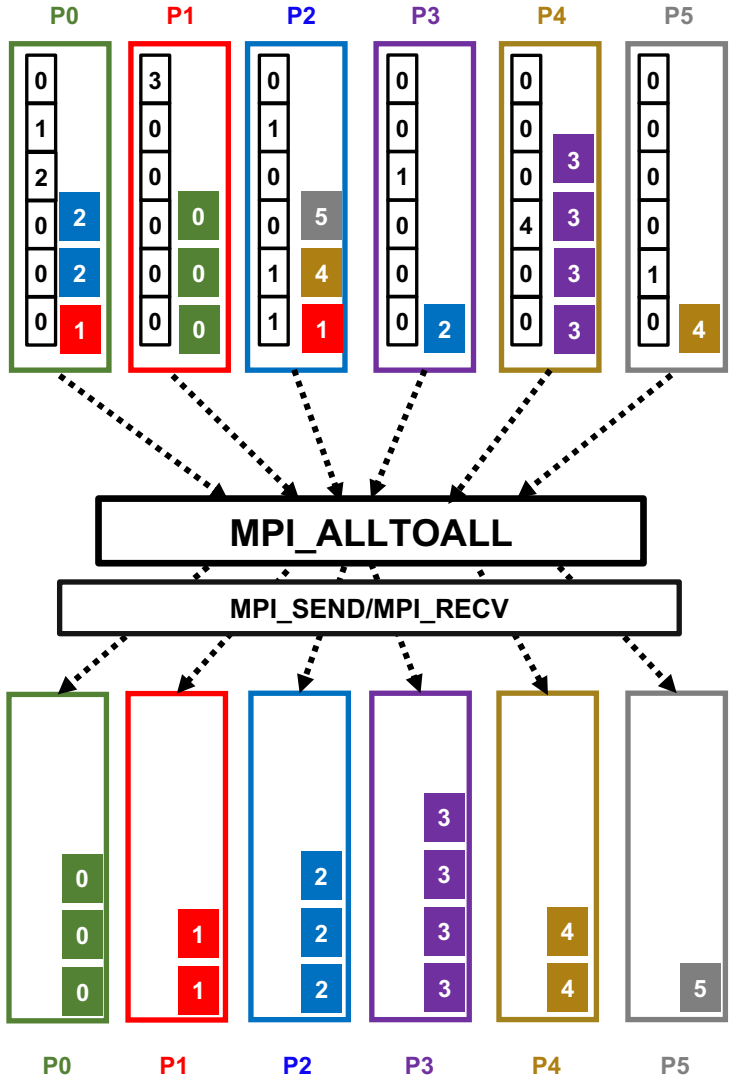
- Dynamic Sparse Data Exchange
  - Dynamic: comm. pattern varies across iterations
  - Sparse: number of neighbors is limited (O(log P))
  - Data exchange: only senders know neighbors
- Main Problem: metadata
  - Determine who wants to send how much data to me
    (I must post receive and reserve memory)



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

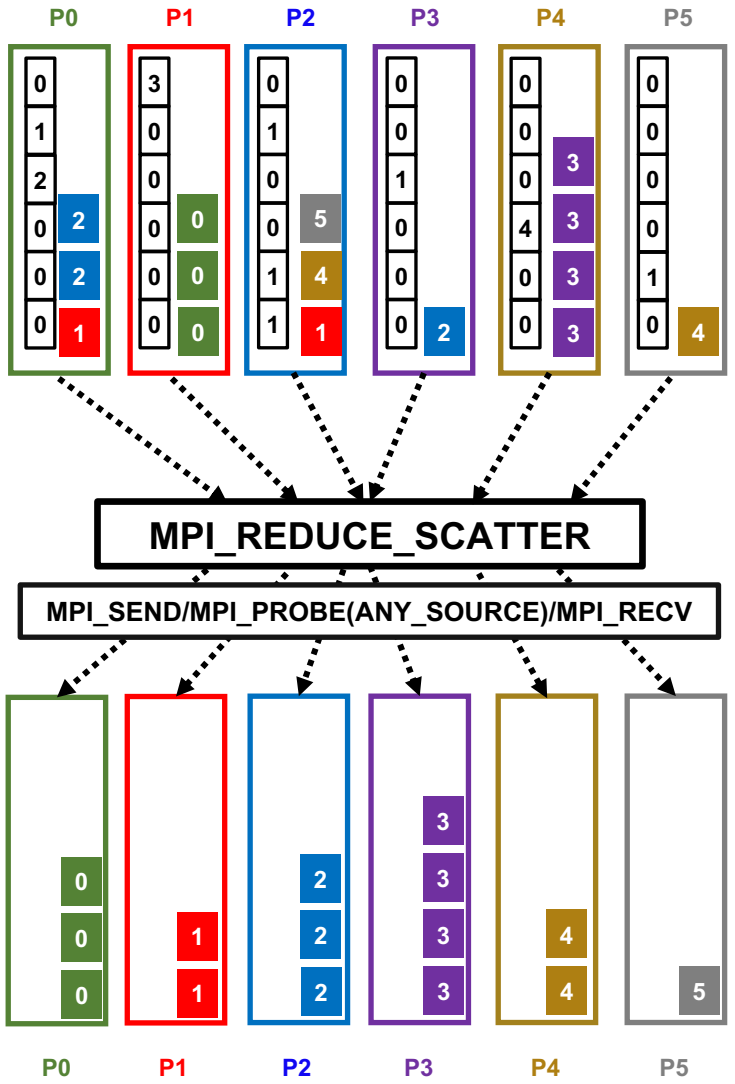
Using Alltoall (PEX)

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



Reduce_scatter (PCX)

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

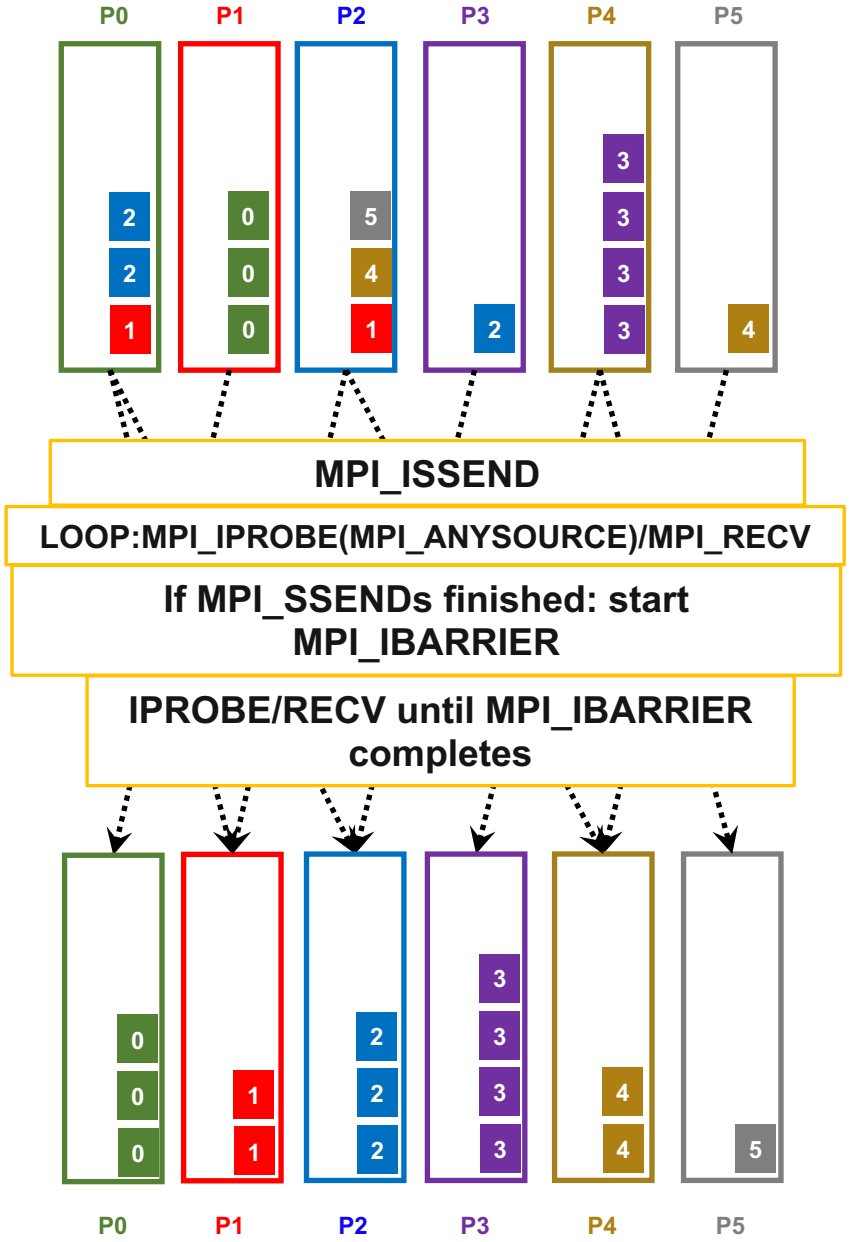


MPI_IBARRIER (NBX)

- Complexity - census (barrier):

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

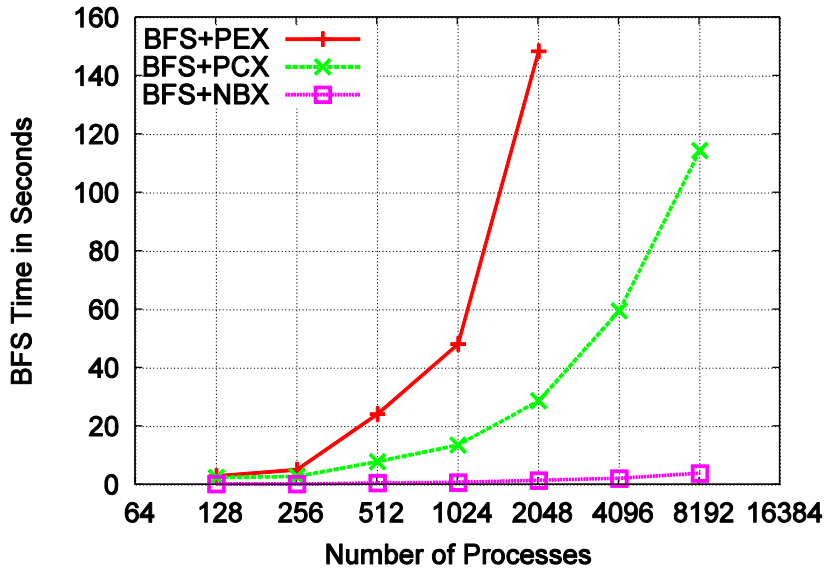
  - Combines metadata with actual transmission
  - Point-to-point synchronization
  - Continue receiving until barrier completes
  - Processes start collective synchronization (ibarrier) when p2p phase ended
    - barrier = distributed marker!
  - Better than Alltoall, reduce-scatter!



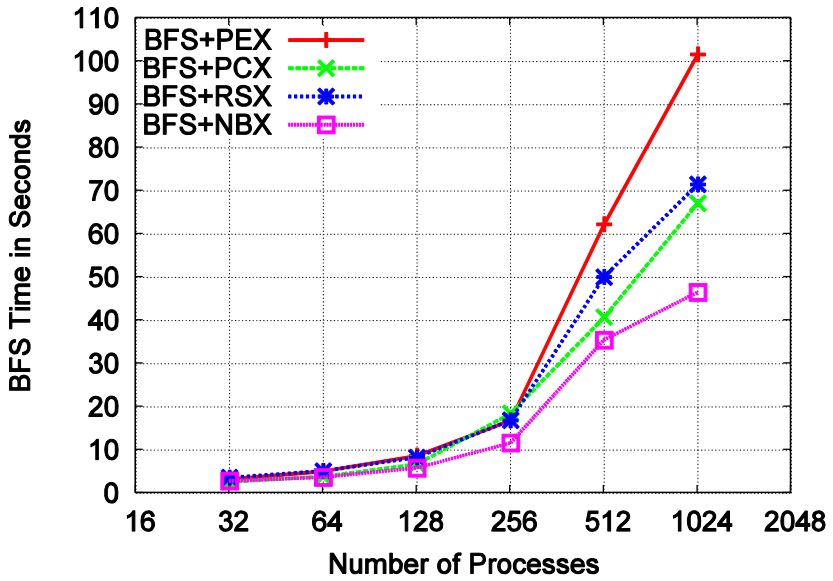
Parallel Breadth First Search

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

BlueGene/P – with HW barrier!



Myrinet 2000 with LibNBC



Impact of HW barrier is significant at large scale!

Example: Stencil with Non-Blocking Collectives

- *nonblocking_coll/stencil.c*

- Use *MPI_Alltoallw* to do data exchanges

- Overlap communication with computation
  - Compute “inner” grid points that don’t require halo zones while collective is running

Section Summary

- Collectives are a very powerful and popular 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

Derived Datatypes

Introduction to Datatypes in MPI

- Automatically serialize **arbitrary** data layouts into a message stream
  - Networks provide serial channels
  - Same for block devices and I/O

- Several constructors to 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

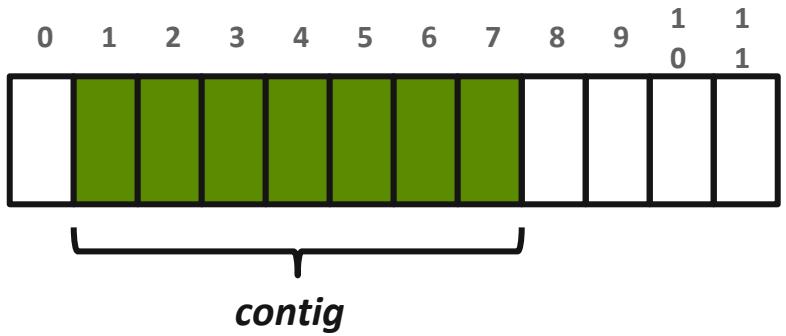
- Equivalents exist for all C, C++ and Fortran native datatypes
  - C int → MPI_INT
  - C float → MPI_FLOAT
  - C double → MPI_DOUBLE
  - C uint32_t → MPI_UINT32_T
  - Fortran integer → MPI_INTEGER

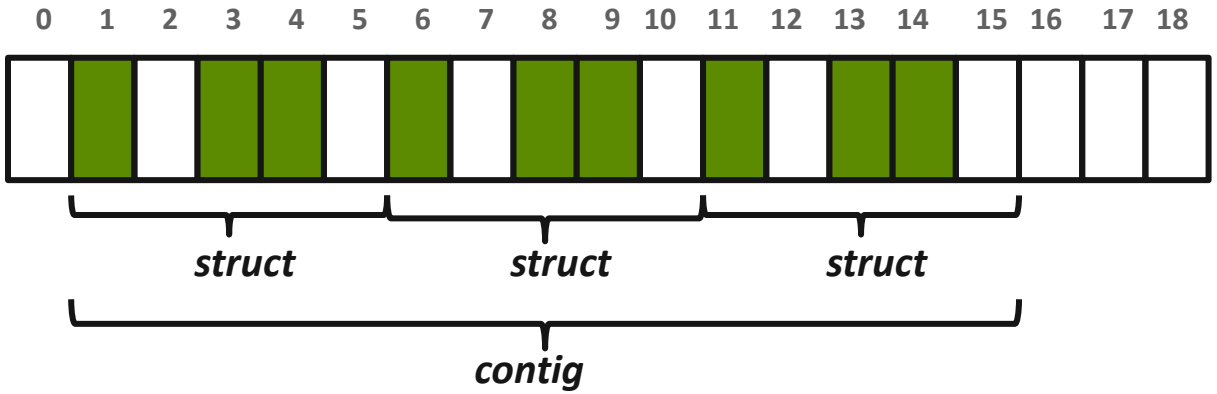
- MPI provides routines to represent more complex, user-defined datatypes
  - Contiguous
  - Vector/Hvector
  - Indexed/Indexed_block/Hindexed/Hindexed_block
  - Struct
  - Some convenience types (e.g., subarray)

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” (final call in nested set), can be replaced by count

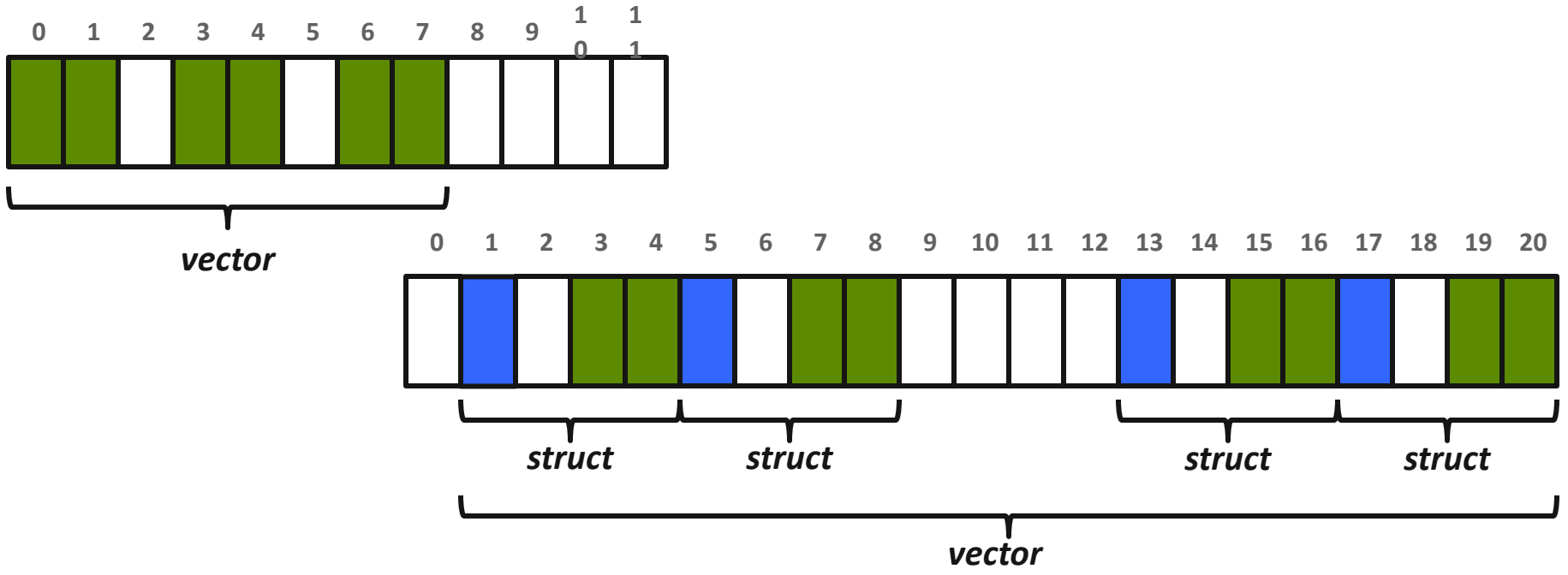




MPI_Type_vector

```
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)

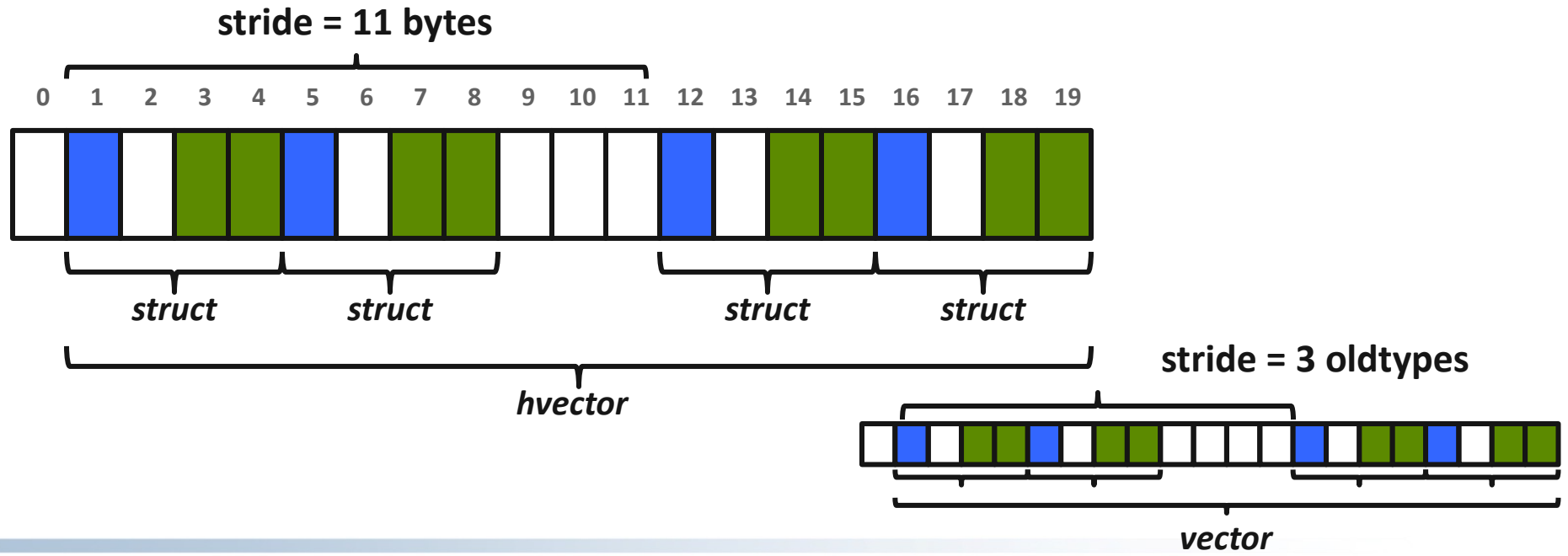
Example: Noncontiguous Send and Recv

- *derived_datatype/sendrecv_nc.c*
- *3 methods for sending and receiving noncontiguous data*
  1. *Multiple messages*
  2. *Manual packing*
  3. *MPI Datatypes – 1 message, no packing!*

MPI_Type_create_hvector

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

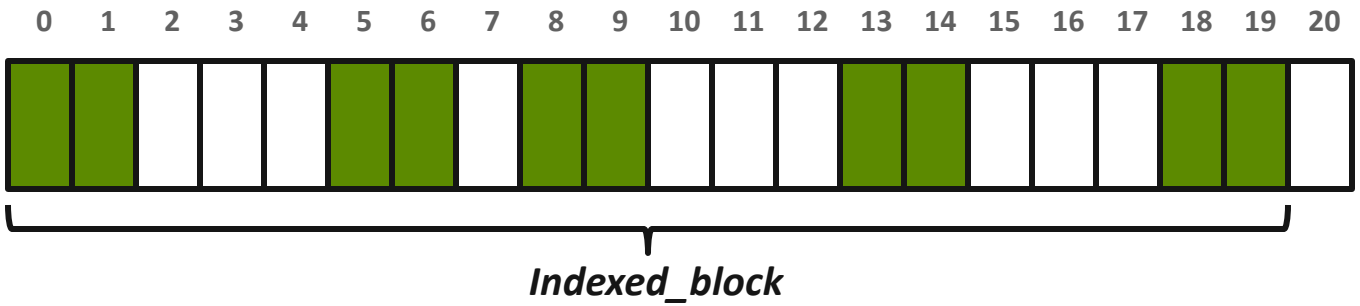
- Create *byte* strided vectors
- Useful for composition, e.g., vector of structs



MPI_Type_create_indexed_block

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

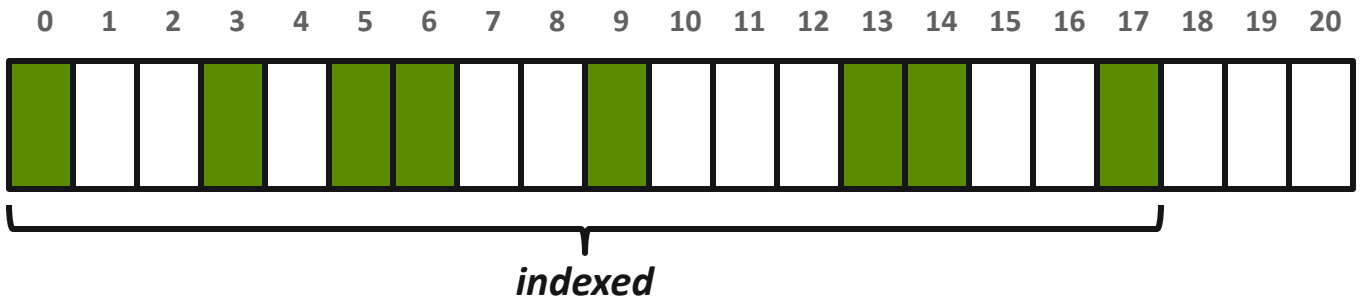
- Pulling irregular subsets of data from a single array
  - dynamic codes with index lists, expensive though!
  - blen=2
  - displs={0,5,8,13,18}



MPI_Type_indexed

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

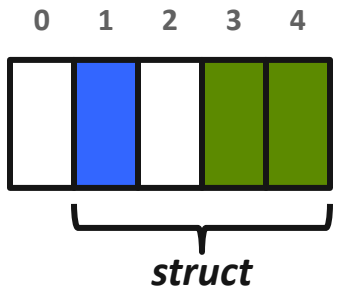
- Like indexed_block, but can have different block lengths
  - blen={1,1,2,1,2,1}
  - displs={0,3,5,9,13,17}



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)
```

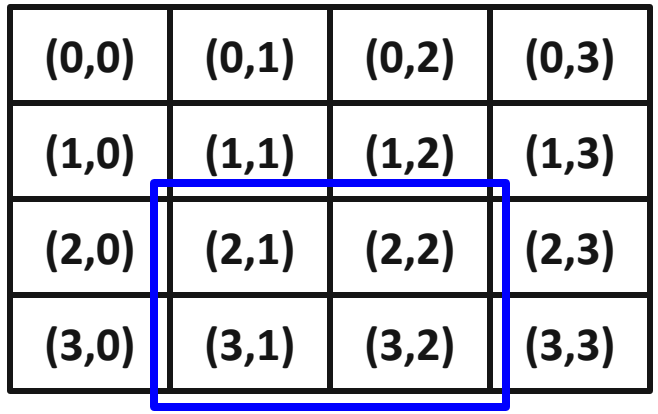
- Most general constructor, allows different types and arbitrary arrays (also most costly)



MPI_Type_create_subarray

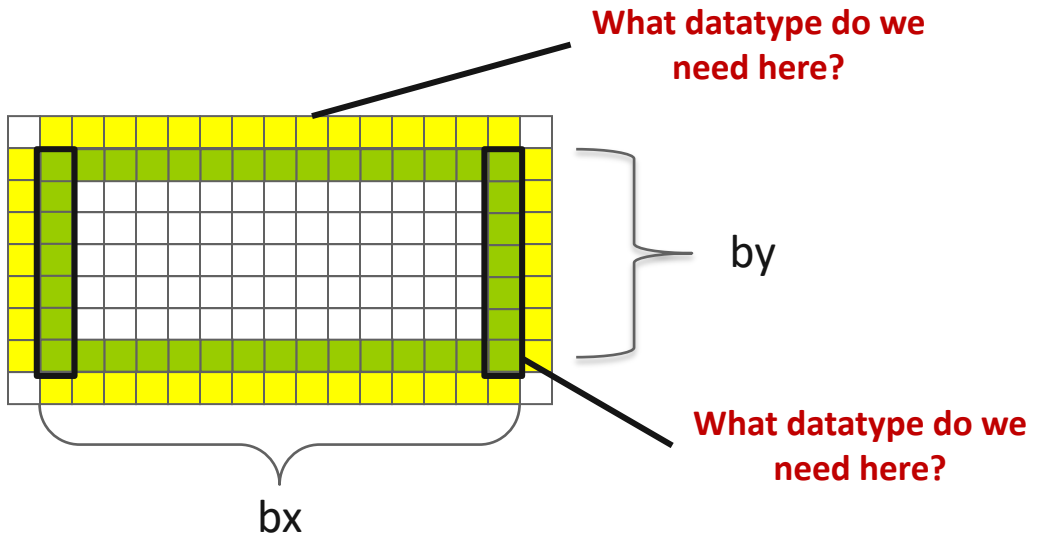
```
MPI_Type_create_subarray(int ndims, int* array_of_sizes, int *array_of_subsizes,
                         int *array_of_starts, int order, MPI_Datatype oldtype,
                         MPI_Datatype *newtype)
```

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



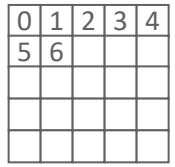
Exercise: Stencil with Derived Datatypes (1/3)

- In the previous versions of the stencil code
  - Used manual packing/unpacking of data 👎
- Let's try to use derived datatypes
  - Specify the locations of the data instead of manually packing/unpacking



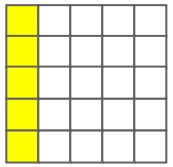
Exercise: Stencil with Derived Datatypes (2/3)
Memory Layout of 2D Array

- Buffers allocated with malloc are contiguous in memory - the addresses are sequential.

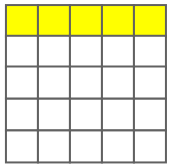


2d array

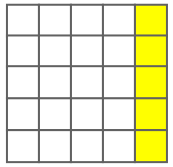
2d array in sequential memory



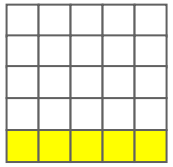
west edge



north edge



east edge



south edge

Example: Stencil with Derived Datatypes

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

Section Summary

- Derived datatypes are a mechanism to describe ANY layout in memory
  - Hierarchical construction allows them to be as complex as the data layout
  - More complex layouts require more complex datatype constructions

- Current MPI implementations are lagging in performance, but it is improving
  - Increasing hardware support to process derived datatypes on the network hardware
  - If you run into performance issues, complain to the MPI implementer, don’t stop using it!

Bonus Exercise: Stencil using Derived Datatype and Collectives

- As time allows/homework

- Goal: Use collective communication with derived datatype

- *Start from derived_datatype/stencil.c*
- *Solution in derived_datatype/ directory*

Argonne NATIONAL LABORATORY

MPI Remote Memory Access (RMA, a.k.a One-Sided) Communication

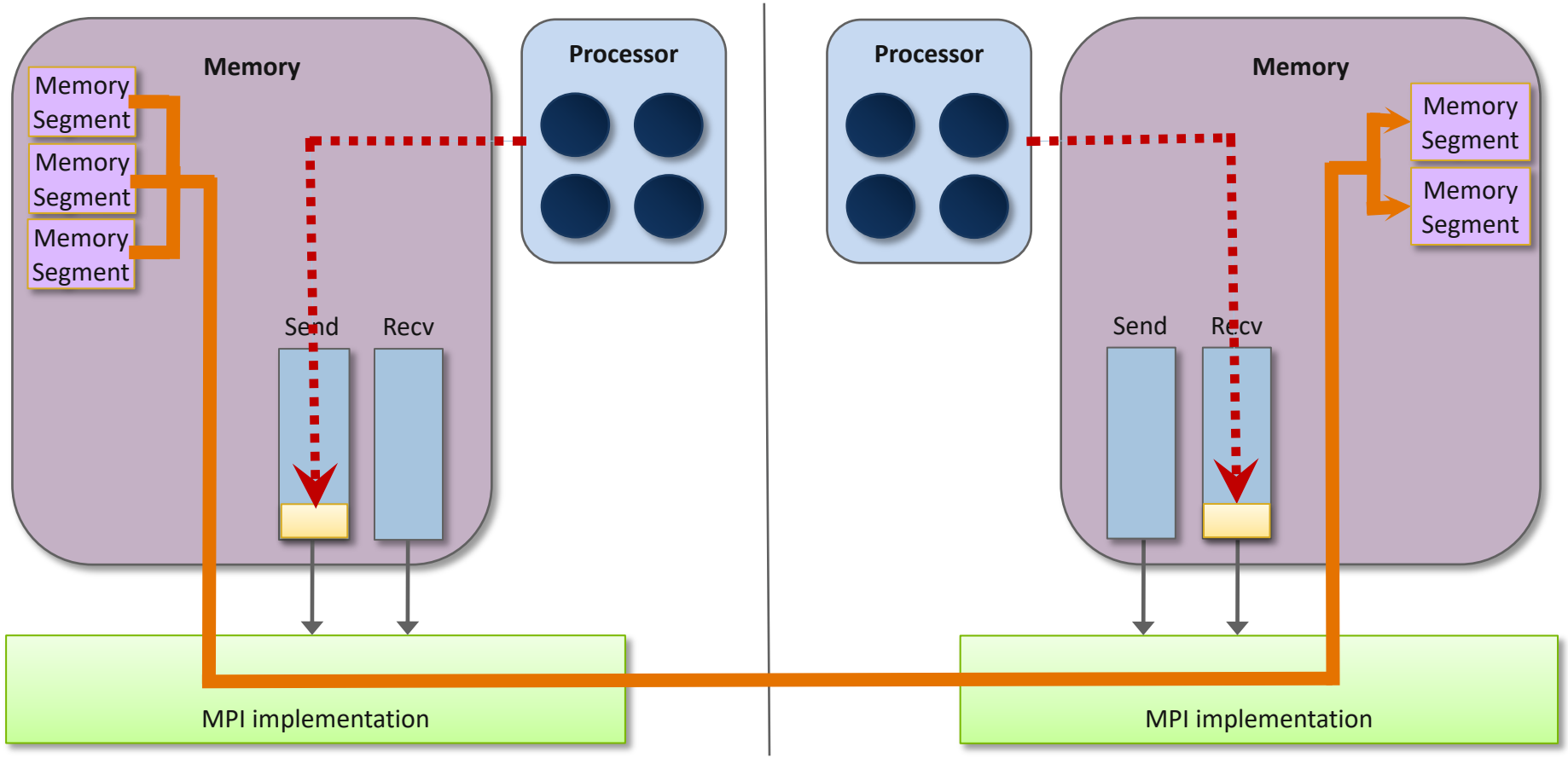
U.S. DEPARTMENT OF ENERGY

Process Provide Memory Isolation

- One process does not simply access another process's memory
  - Operating system ensures that
  - But parallel programs need to exchange data
- Using shared memory for parallel programs
  - Within one node
- Multithreaded programming
  - Within one node

Limitation of Point-to-Point Communcation

- Point-to-point is a *two-sided* communication
  - Communication requires a SEND to match with a RECV, developer must to arrange that
  - Sender is delayed if Receiver is slow, especially for large message

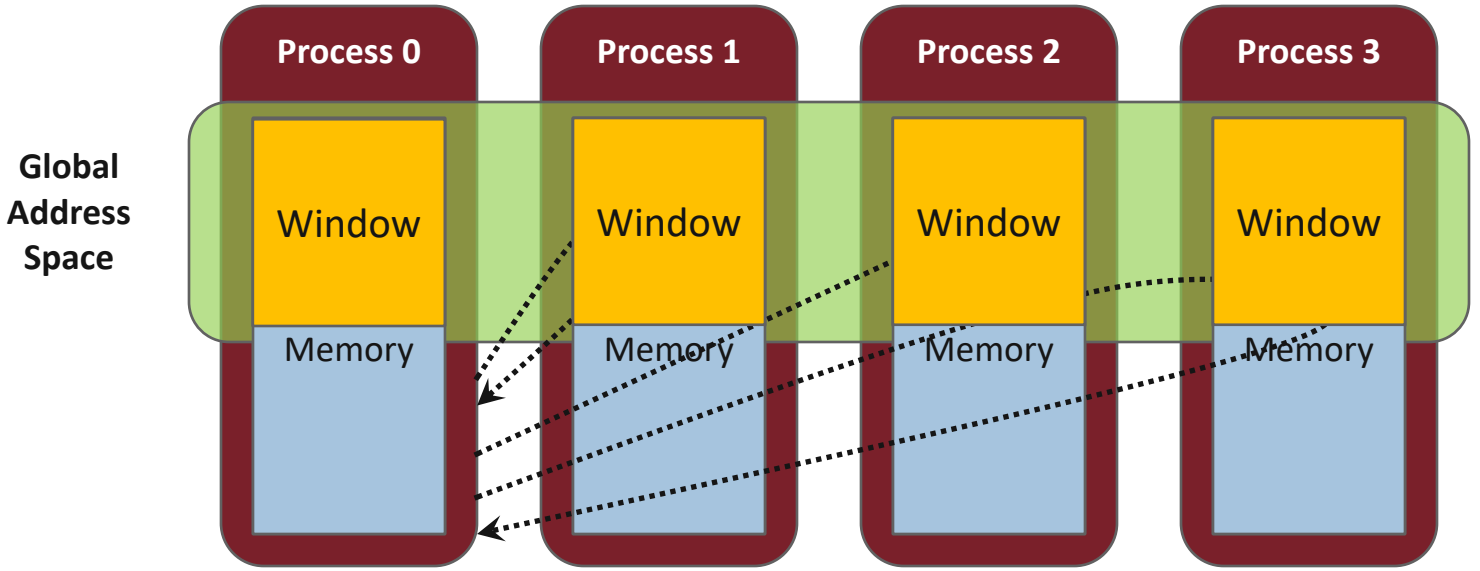


Process Synchronization as A Side Effect of Data Movement

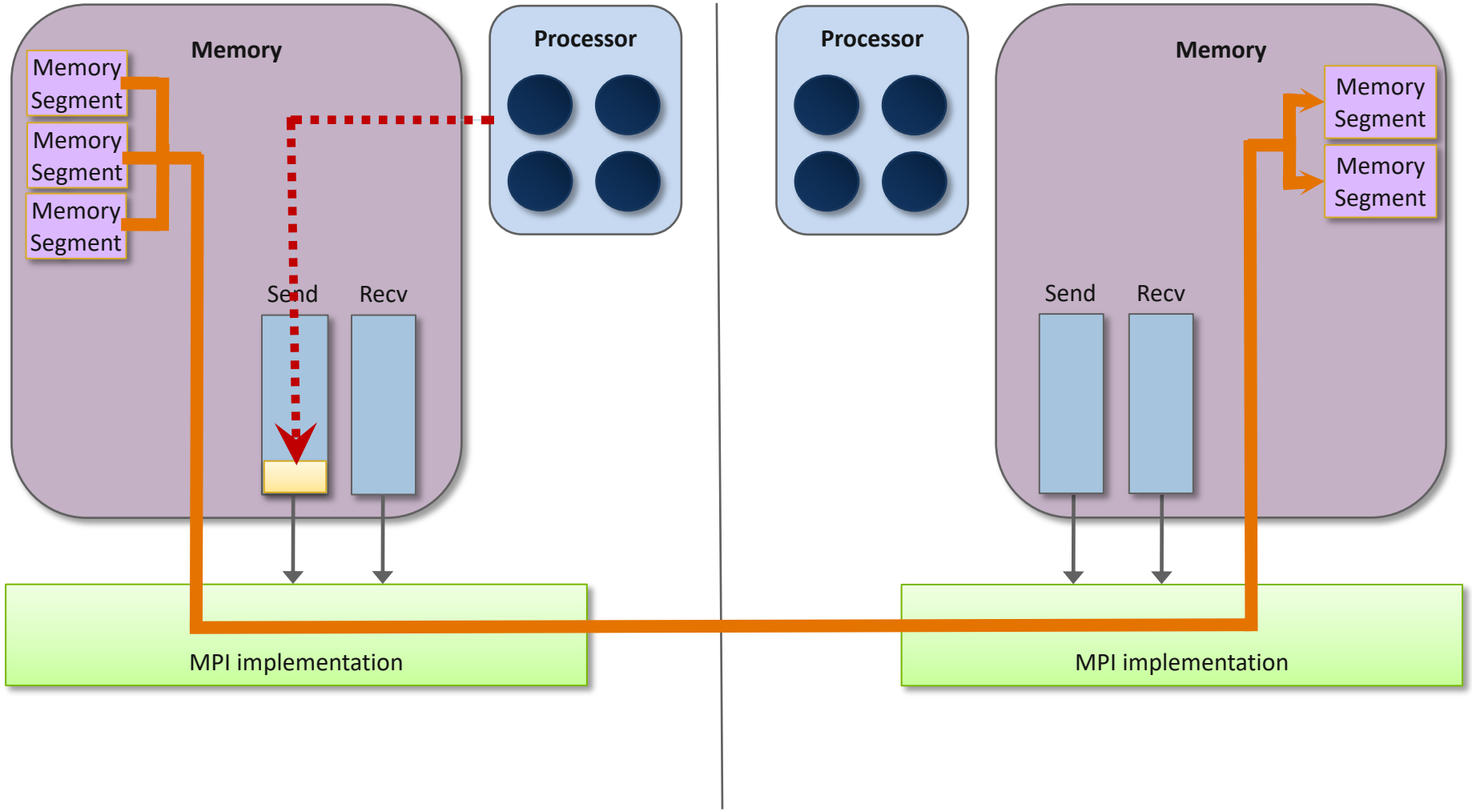
- Point-to-point: synchronizes between a pair of processes
- Collective: synchronizes among a group of processes
  - All processes in the group must participate
  - Everybody waiting for the straggler
- Nonblocking is a Remedy
  - Overlaps communication with computation
  - Still stuck at MPI_Wait/Test if run out of work to overlap
  - Load unbalance

Remote Memory Access (RMA)/One-sided Communication

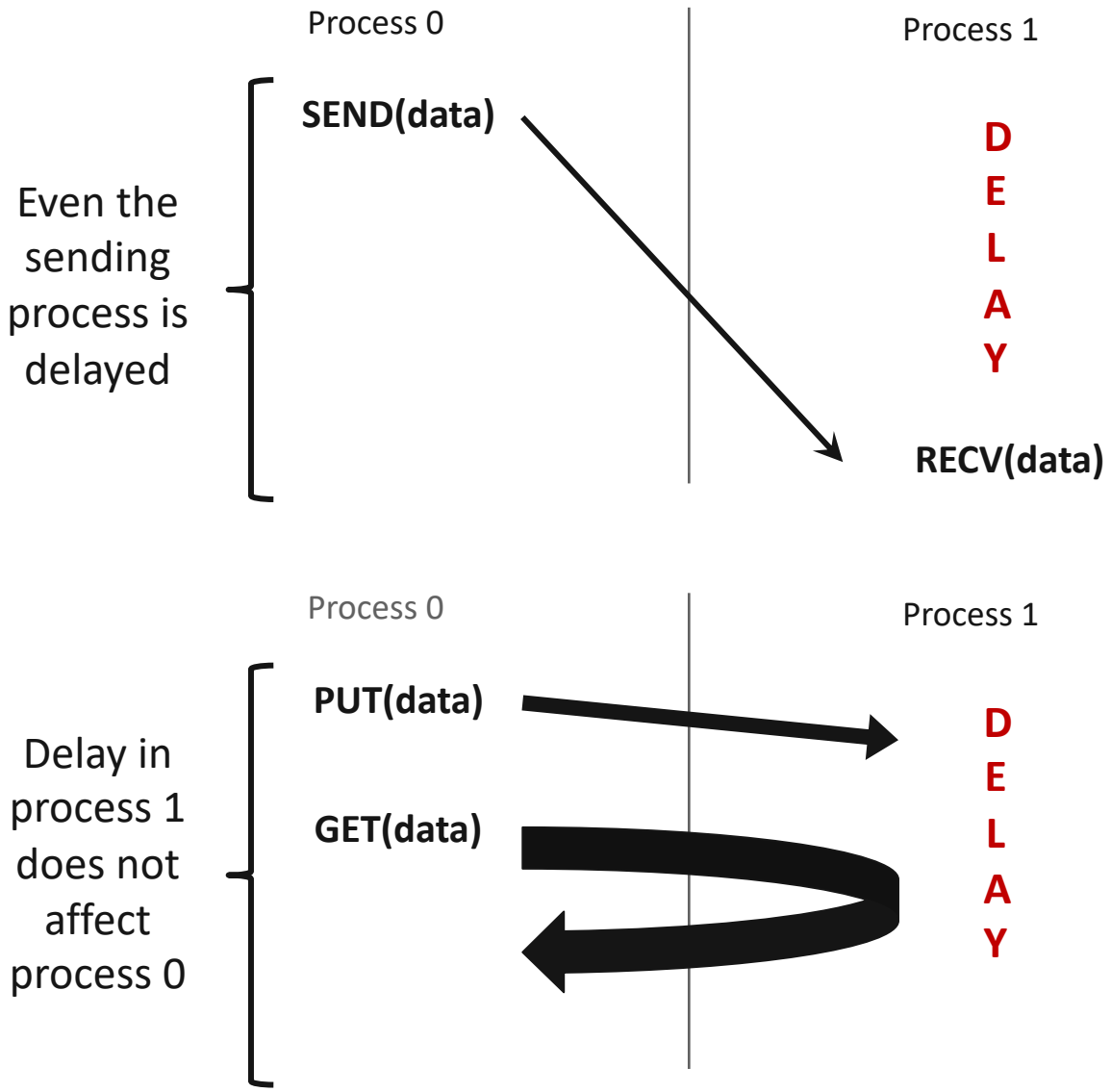
- The basic idea is global address space that makes process's private memory public
  - Each process can exposes a part of its memory to other processes
  - Other processes can directly read from or write to this memory, even across nodes
  - One process should be able to move data without requiring that the remote process to participate/synchronize



RMA Communication Example



Comparing RMA and Point-to-Point Programming



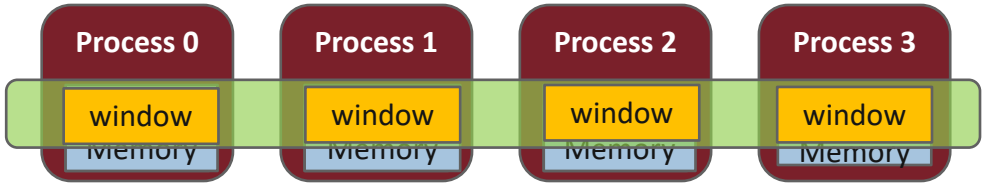
What we need to know in MPI RMA

- Allocation: How to create remote accessible memory?
- Operation: Reading, Writing and Updating remote memory
- Coordination: Data Synchronization
- Information: 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

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 existing 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)

win = WIN_CREATE(base, size, comm);

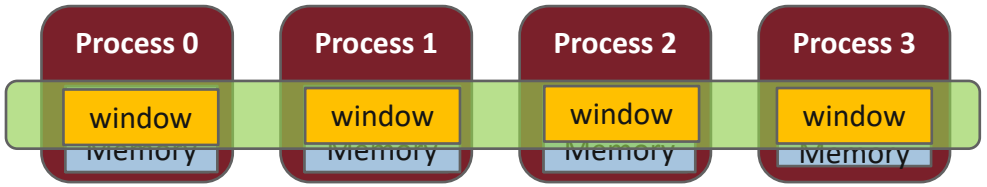
Example with MPI_WIN_CREATE

```
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_ALLOCATE

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

- Allocate a new memory region and expose it globally via 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** - returned pointer to exposed local data
  - **win** - returned window (handle)



baseptr, win = WIN_ALLOCATE(size, comm);

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;
}
```

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
    - (not covered in this tutorial, but slides available)
  - **`MPI_WIN_ALLOCATE_SHARED`**
    - You want multiple processes on the same node share a buffer (covered in later section)

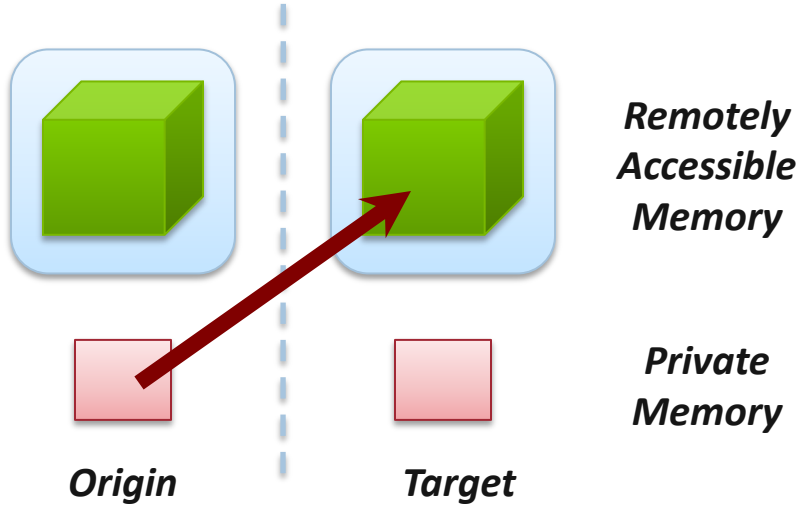
Data movement

- 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.
- All these operations are nonblocking!

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)
```

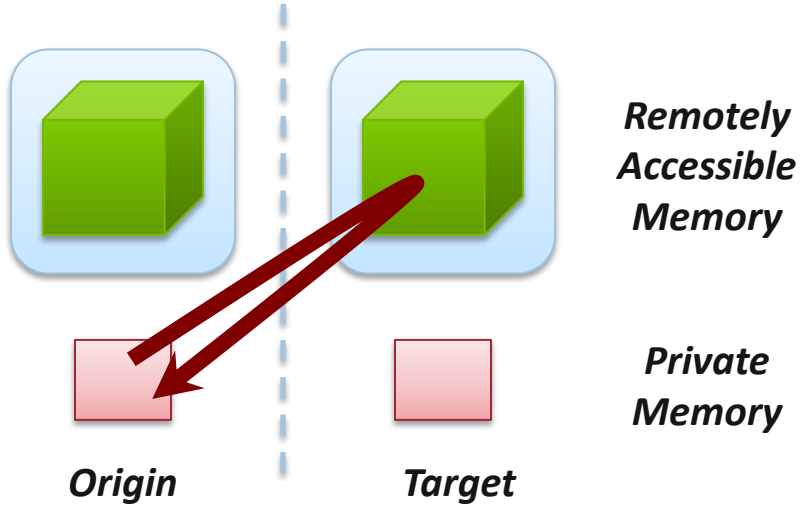
- Move data from origin, to target
- Separate data description triples for **origin** and **target**



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)
```

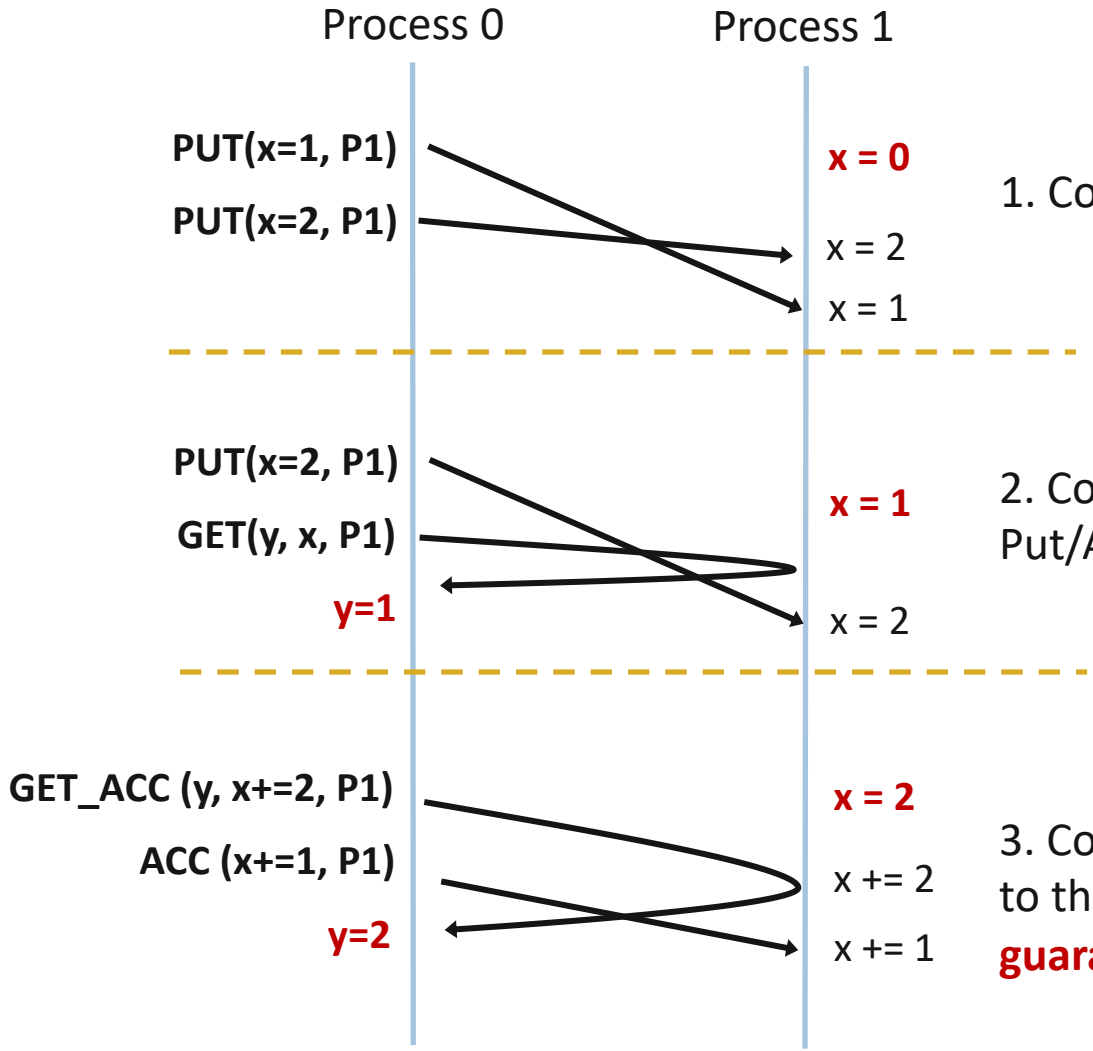
- Move data to origin, from target
- Separate data description triples for **origin** and **target**



Ordering of Non-atomic Operations in MPI RMA

- 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?
  - What is the value of M if both process X and process Y put into it?
  - What data will be read if process X do a put then get to M?
- No guaranteed ordering for Put/Get operations
- Result of concurrent Puts to the **same** location is undefined (different locations are fine)
- Result of Get concurrent Put undefined
  - Can be garbage
- What is “concurrent” here?
  - RMA operations are nonblocking
  - Sequentially issued operations can happen to update target memory at the same time
  - Sequentially issued operations can seemed to be reordered
  - Memory can be in corrupted states

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**

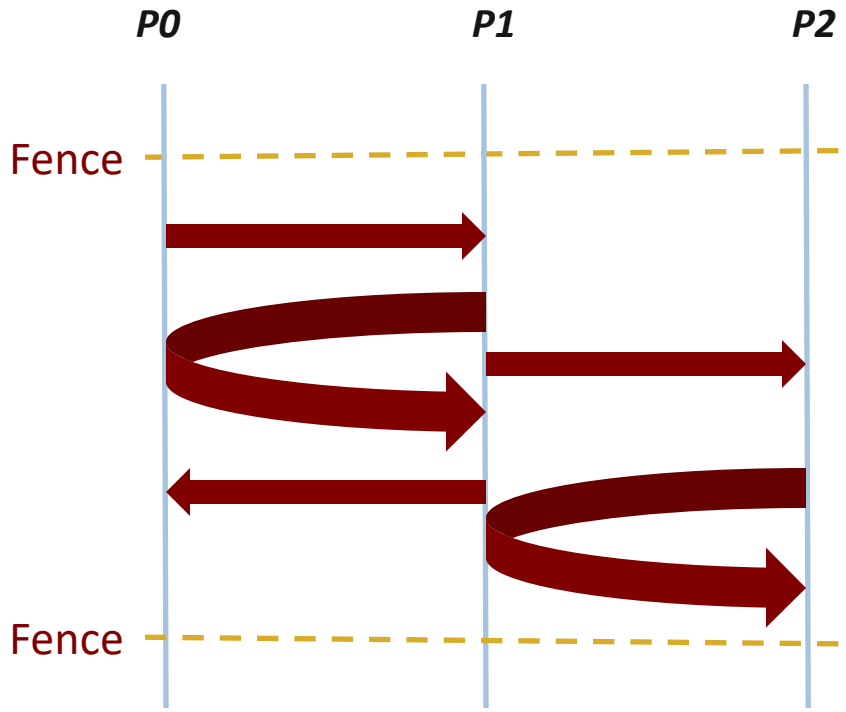
RMA Synchronization Models

- RMA data access model
  - Epochs define ordering and completion semantics
  - Open a epoch to allow RMA operations
  - Close a epoch to make sure memory changes in the epoch is visible to other processes
- Three synchronization models provided by MPI:
  - **Lock/Unlock (passive target)** <- preferred for all RMA since MPI 3.0
  - **Fence (active target)**
  - Post-start-complete-wait (PSCW) (generalized active target)
- They are memory synchronizations, not process synchronizations
  - Memory states are consistent
  - Not synchronizing processes is the point

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
- This is not a MPI_Barrier

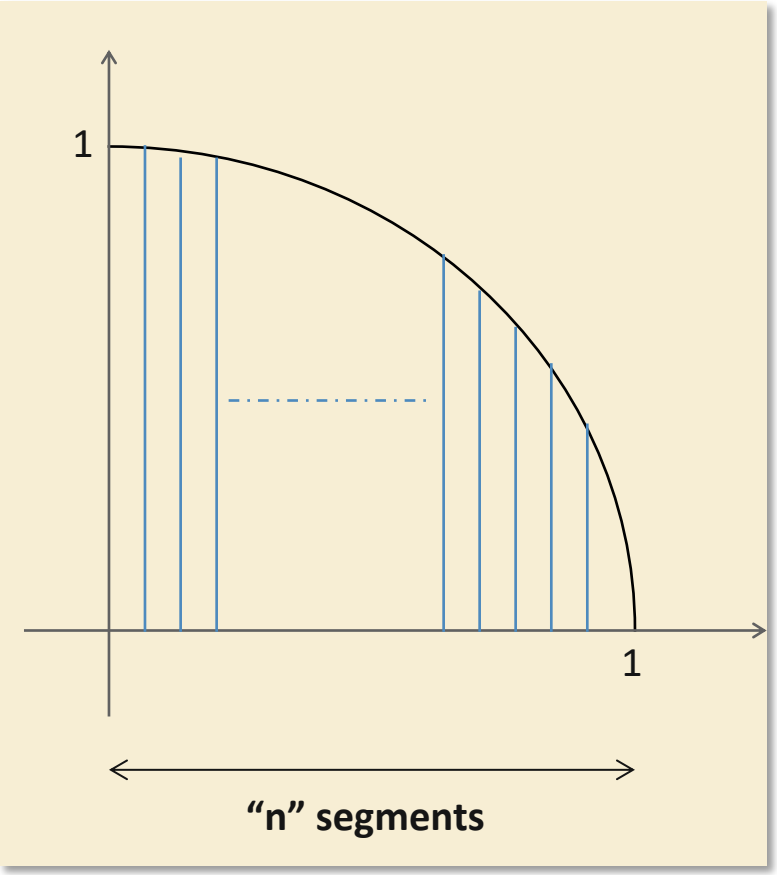


Example with MPI_PUT and MPI_FENCE

```
int main(int argc, char ** argv)
{
    int *a, value = 1;
    MPI_Win win;

    /* Init and allocation omitted */

    /* starting an epoch */
    MPI_Fence(0, win);

    if (rank == 0) {
        MPI_Put(&value, 1, MPI_INT, 1, 0, 1, MPI_INT, win);
    }

    MPI_Fence(0, win);

    /* cleanup omitted */
}
```

Hands-on Example: Calculating Pi with RMA

- 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
- Blocking Collective Version
  - Bcast to distribute n
  - Reduce to calculate result
  - See blocking_coll/cpi.c



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)

Hands-on Task 1: Replace Collective Communication with RMA

- Replace MPI_Bcast and MPI_Reduce with RMA
- Starting from rma/cpi.c
- TODO
  - Create windows to expose memory
  - Use MPI_Put/MPI_Get to move data (n and mypi)
  - Use MPI_Win_fence for synchronization
  - Cleanup at the end

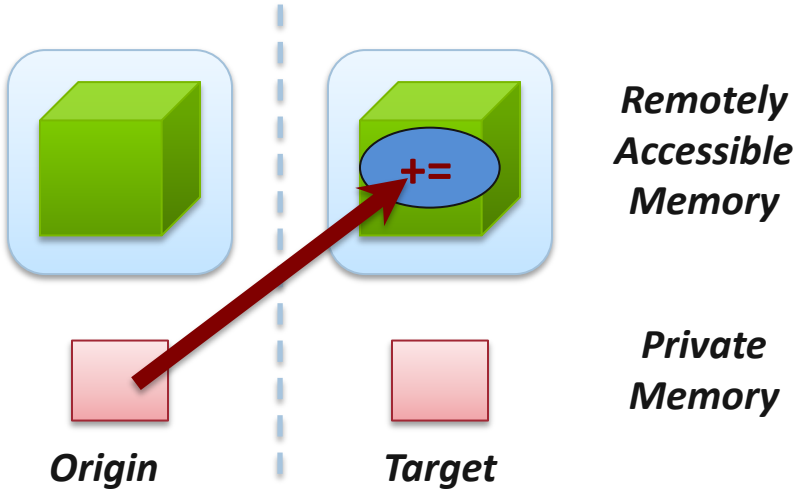
Atomic Data movement

- 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.
- All these operations are nonblocking!

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)
```

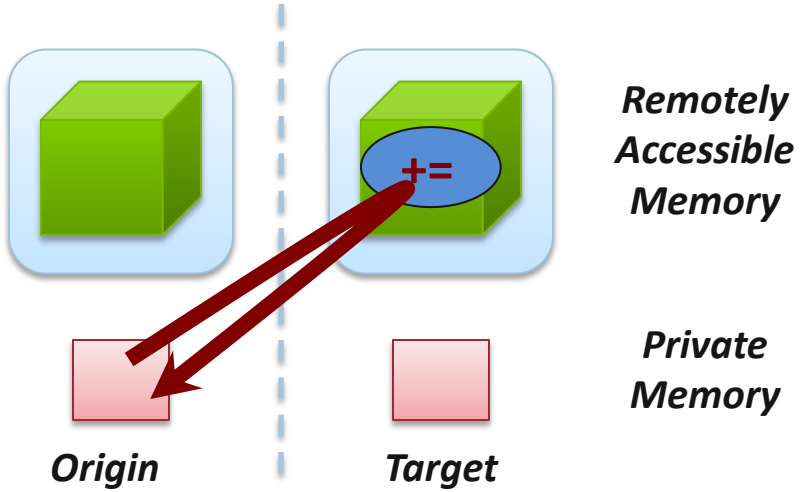
- Atomic update operation, similar to a put
  - Reduces origin and target data into target buffer using op argument as combiner
  - Op = **MPI_SUM, MPI_PROD, MPI_OR, MPI_REPLACE, MPI_NO_OP**, …
  - Predefined ops only, no user-defined operations
- Different data layouts between target/origin OK
  - Basic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

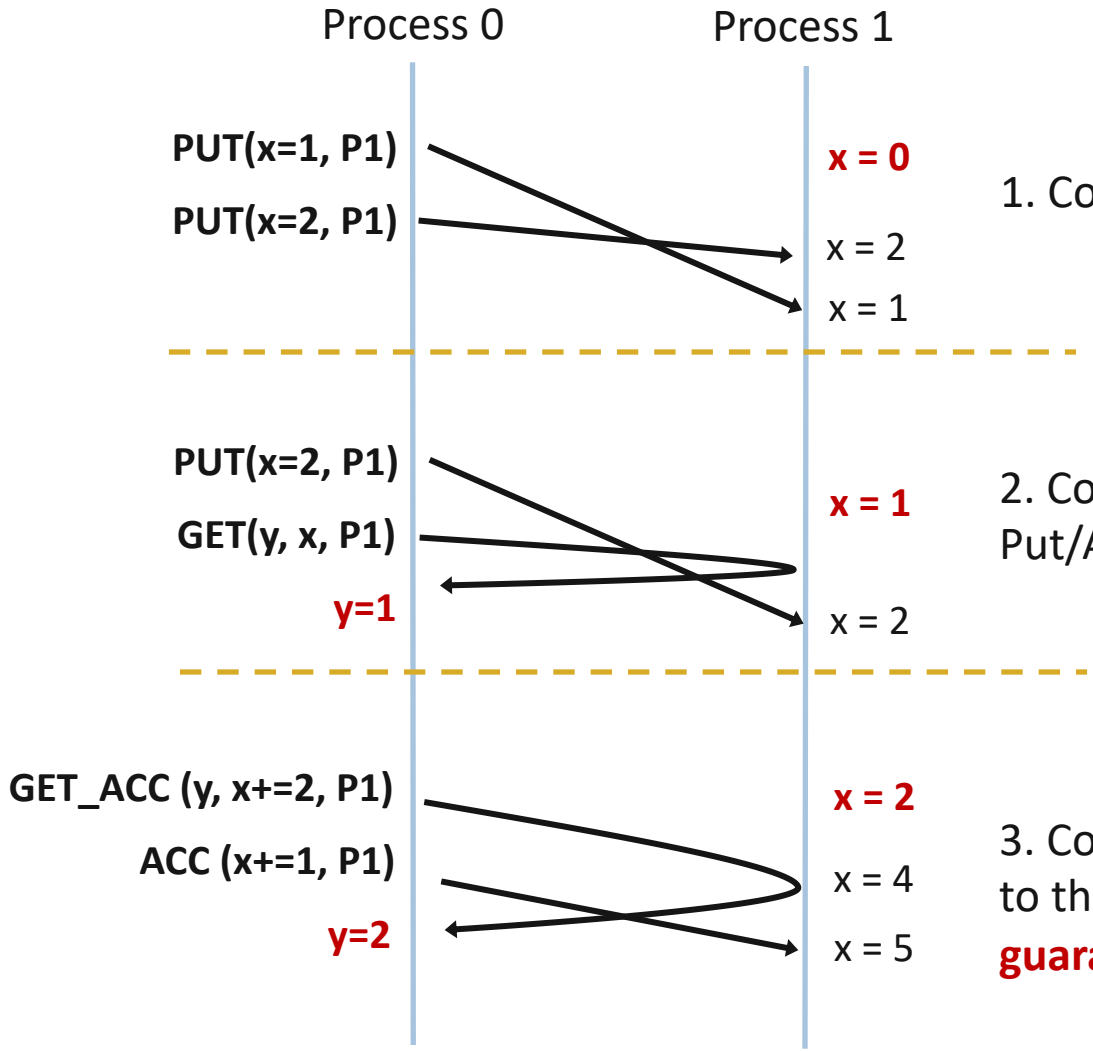
- FOP: Simpler version of MPI_Get_accumulate
  - All buffers share a single predefined datatype
  - No count argument (it's always 1)
  - Simpler interface allows hardware optimization
- CAS: Atomic swap if target value is equal to compare value

Ordering of Operations in MPI RMA

- 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
  - This is where info_hint in window creation is useful

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

Examples with operation ordering with atomic



1. Concurrent Puts: **undefined**

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

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

Hands-on Task 2: Improving with RMA Atomic

- Use MPI_Accumulate to calculate the sum
  - Each process ”push” partial sum to rank 0
  - Avoids the loop and multiple MPI_Win_fence

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)
```

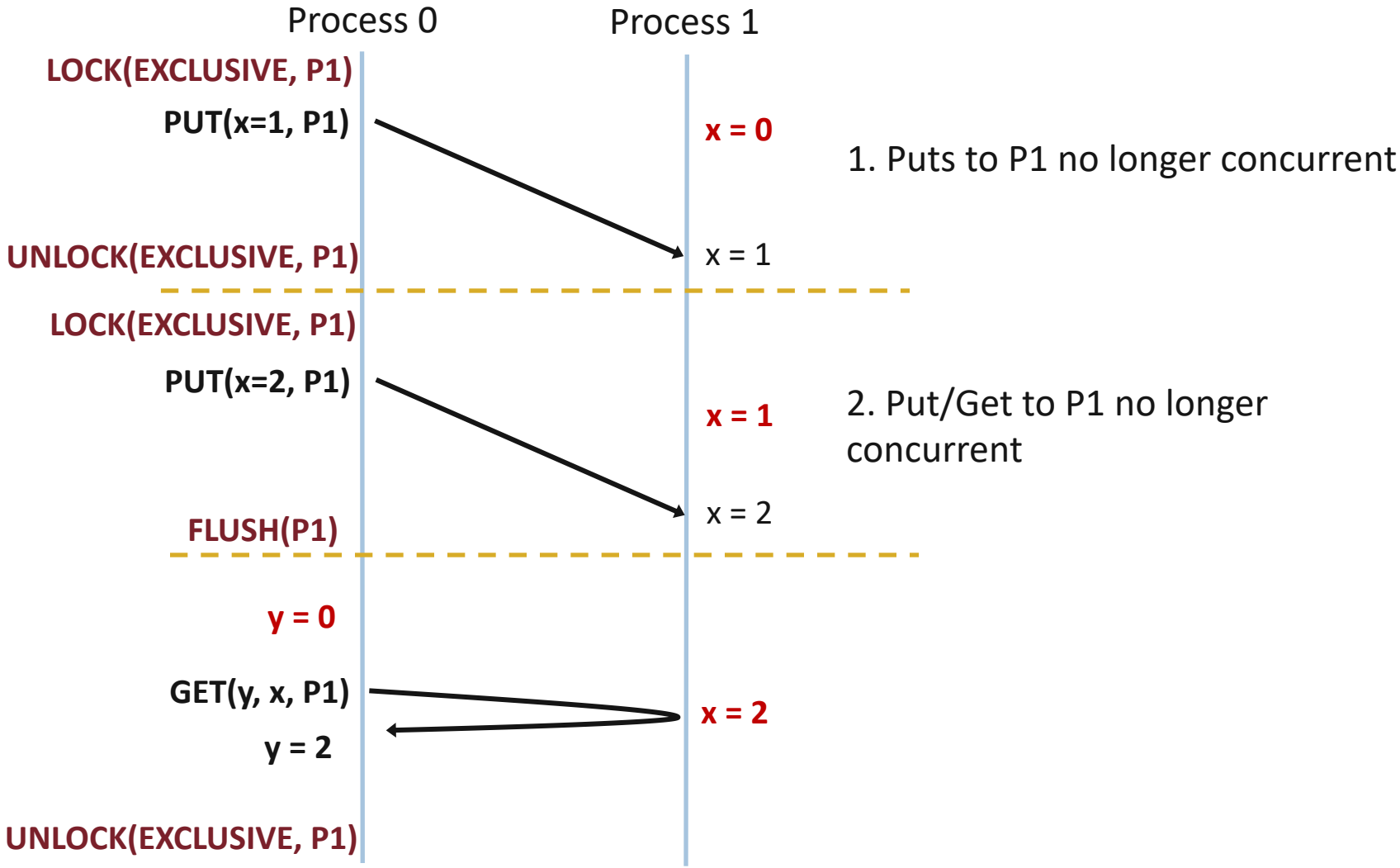
- Lock/Unlock: Begin/end passive mode epoch
  - Target process does not make a corresponding MPI call
  - Can initiate multiple passive target epochs to different processes
  - Concurrent epochs to same process not allowed (affects threads)
- Lock type
  - SHARED: Other processes using shared can access 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

Examples with operation ordering

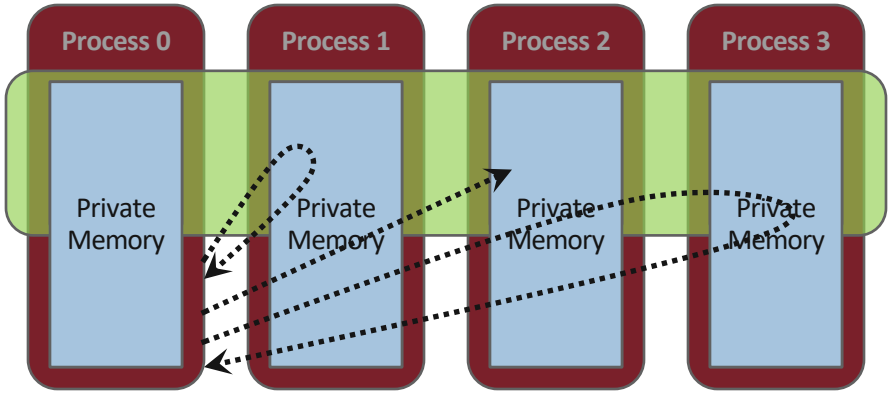


Hands-on Task 3: Change MPI_Win_fence to MPI_Win_lock/unlock

- Use MPI_Win_lock/unlock for fine-grained synchronization
  - Avoids most collective synchronization
  - Caveat: at one collective synchronization is needed

Section Summary

- MPI RMA communication is provides virtual global arrays across processes
- Exposes memory through **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…



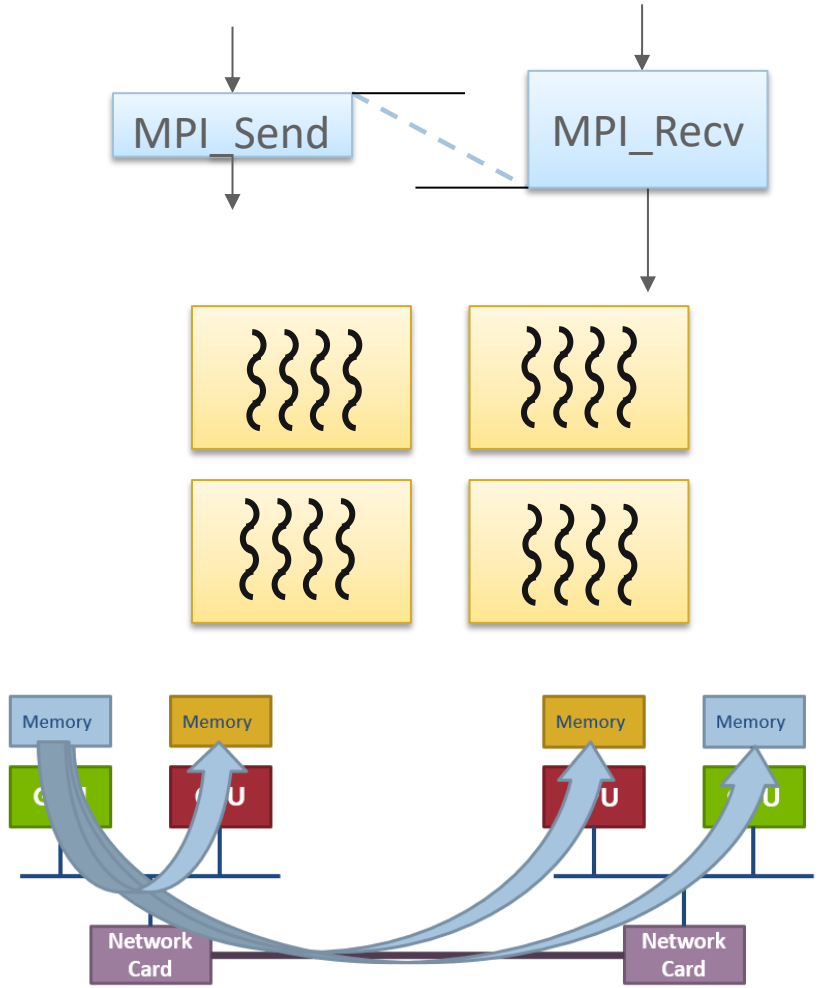
Argonne NATIONAL LABORATORY

Hybrid Programming: MPI+Threads & MPI+GPU

U.S. DEPARTMENT OF ENERGY

MPI Hybrid Programming - Overview

- What makes MPI hybrid programming “difficult” is performance.
- More hands-on exercises
  - *Performance is all about expectation.*
- Topics:
  - Basic MPI messaging
  - MPI+Threads
  - MPI+GPU



- Grab an interactive session on aurora:

```
qsub –l select=2,walltime=4:00:00 –l filesystems=home –A ATPESC2025 –q ATPESC
```

Warm-up Exercise: Basic Send/Recv

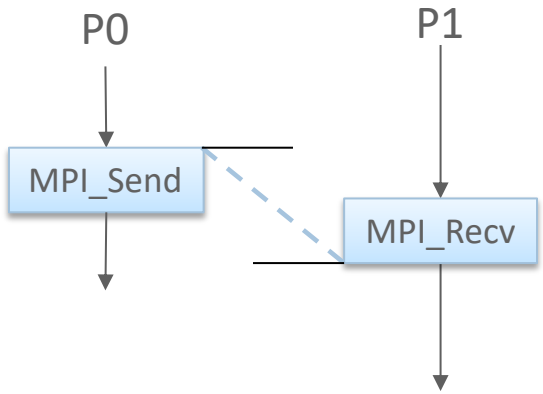
1. Start with a MPI skeleton
2. Setup variables
3. Perform Send/Recv
4. Compile and test

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    MPI_Init(0, 0);
    /* Send/Recv */
    MPI_Finalize();
    return 0;
}
```

```
MPI_Comm comm = MPI_COMM_WORLD;
int rank; MPI_Comm_rank(comm, &rank);
int tag = 0;
int count = 1024;
void *buf = malloc(count);
```

```
if (rank == 0) {
    MPI_Send(buf, count, MPI_INT, 1, tag, comm);
} else {
    MPI_Recv(buf, count, MPI_INT, 0, tag, comm, MPI_STATUS_IGNORE);
}
```

```
$ mpicc t.c && mpirun -n 2 ./a.out
```



Optional:

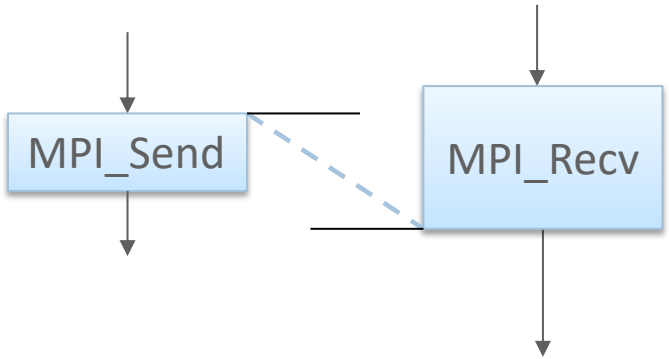
- Check MPI_Comm_size.
- Initialize send buffer and verify received data.
- Check MPI_Status.
- Use MPI_BYTE to count message size in bytes.

Reference: hybrid/basic_sendrecv.c

Semantics of MPI Messaging

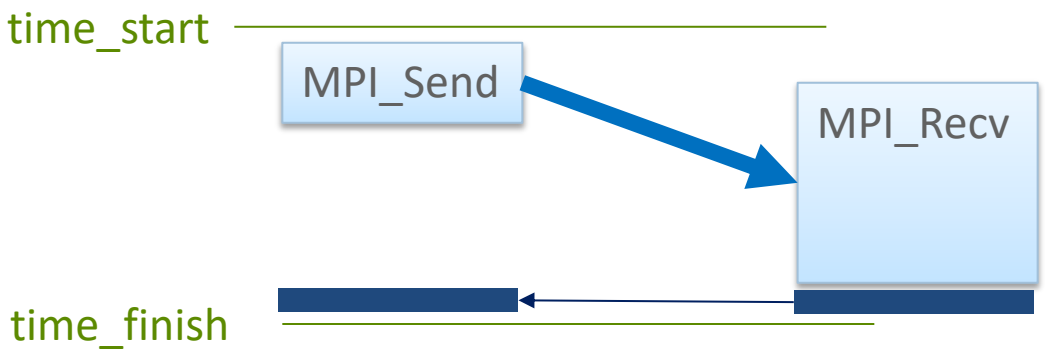
- What does an MPI message accomplish?
  - Data movement
  - Synchronization
- Synchronization
  - Performance metric
    - Latency
- Data transfer
  - Performance metric
    - Bandwidth

```
if (rank == 0) {
    MPI_Send(buf, count, MPI_INT, tag, comm);
} else {
    MPI_Recv(buf, count, MPI_INT, tag, comm,
&status);
}
```



Exercise: Measure Bandwidth

$$Bandwidth = \frac{MessageSize}{ElapasedTime}$$
(GB/sec)



- Use a large message size

```
int count = 1000000000; /* 1GB message size */
void *buf = malloc(count);
```

- Use MPI_Wtime to measure time

```
double time_start = MPI_Wtime();
```

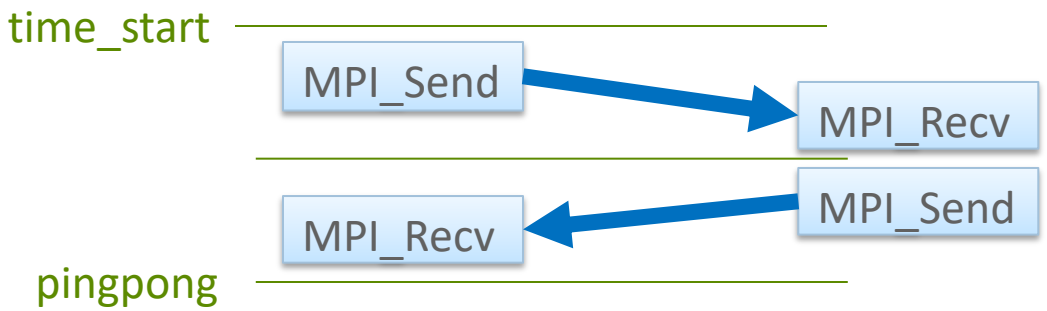
- Use a zero-sized message to synchronize completion

Reference: hybrid/bandwidth.c

Optional:
- Repeat to observe variations.
- Vary message size.
- Intranode vs. internode.
- Intranuma vs. internuma.

Exercise: Measure Latency

$$Latency = Elapased\ Time$$
(µs)



- Use *small* message sizes
- Ping-pong to measure two-way latency;

  divide by 2 for one-way latency
- Aggregate many, many rounds to improve accuracy

Optional:
- Observe measurement variations.
- Vary message size.
- Intranode vs. internode.
- Intranuma vs. internuma.

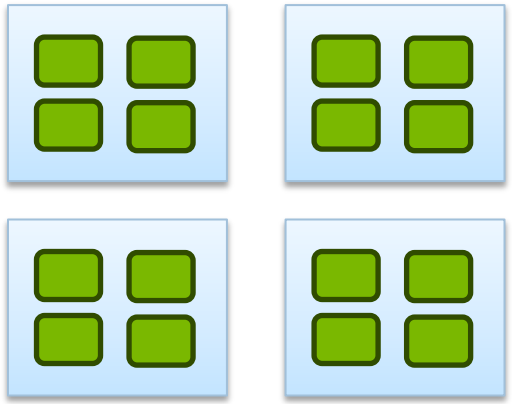
Reference: hybrid/latency.c

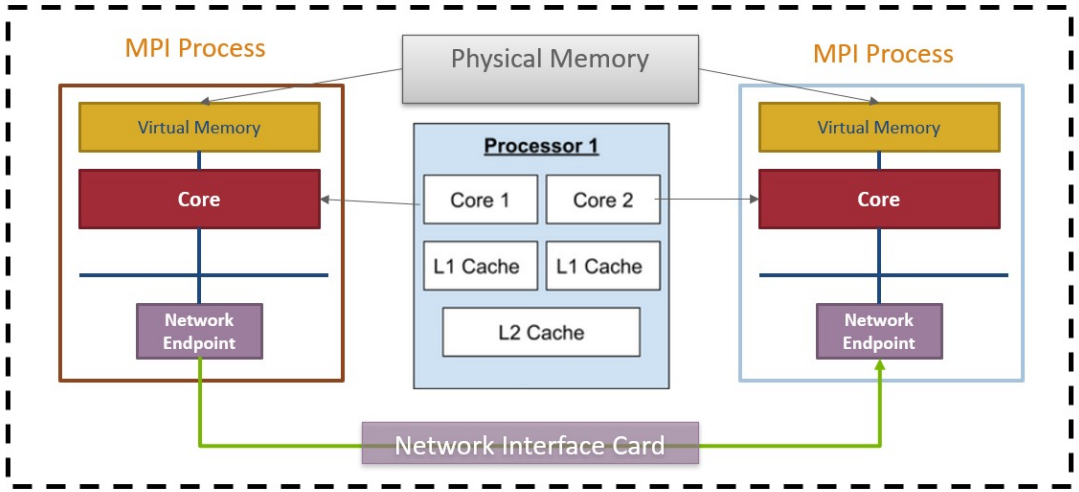
MPI+Threads

Review: The Basic MPI Model

- Execution context
- Memory context
- Semantics
  - Data Transfer
  - Synchronization

- The flat MPI model (one process per core)

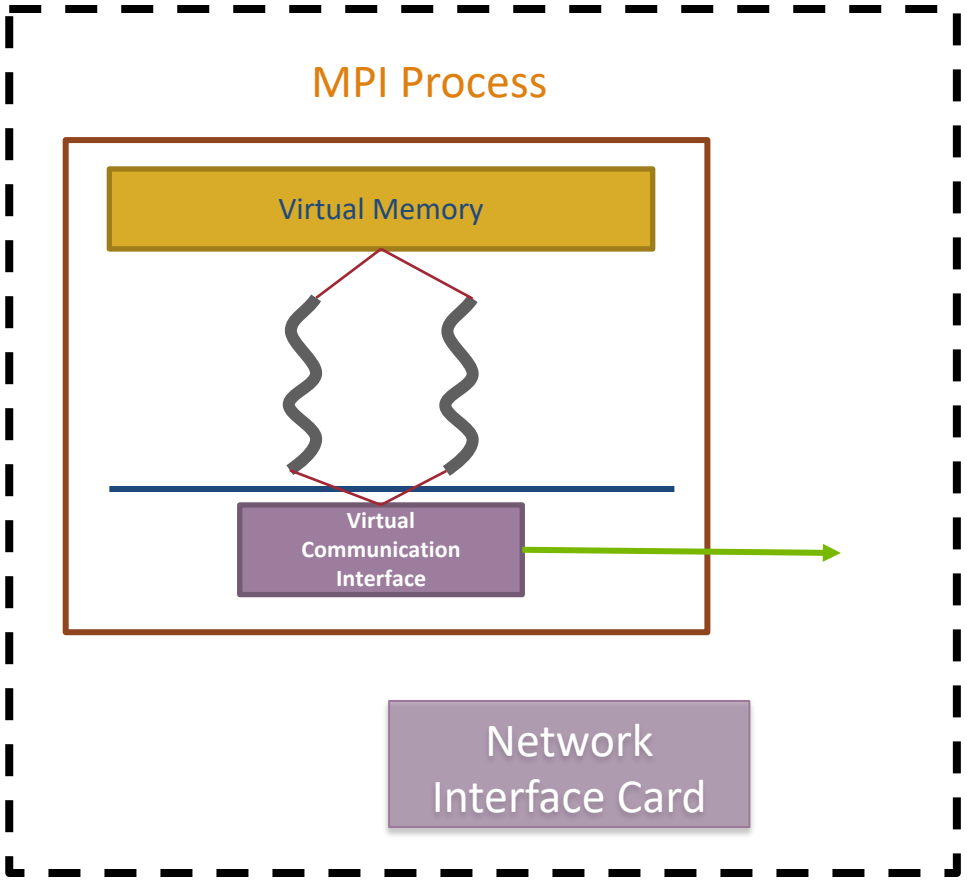




- Homework: Measure intra-node and inter-node performance separately

MPI+Threads

- Threads vs. Process
  - Shared memory
  - Shared communication interface
- Semantics
  - Require Synchronizations
- Advantage
  - Flexible
- Performance Impact



Exercise: MPI+Threads

- Just add an OpenMP parallel region, RIGHT?

```
#pragma omp parallel num_threads(8) {
    if (rank == 0) {
        MPI_Send(buf, count, MPI_INT, tag, comm);
    } else {
        MPI_Recv(buf, count, MPI_INT, tag, comm, &status);
    }
}
```

What could go wrong?

MPI+Threads - What could go wrong - 1

```
Thread 4 received from thread 7
Thread 1 received from thread 7
Thread 7 received from thread 7
Assertion failed in file ./src/mpid/ch4/src/ch4_request.h at line 79: ctr_ >= 1
Thread 5 received from thread 7
Assertion failed in file ./src/mpid/ch4/src/ch4_request.h at line 79: ctr_ >= 1
Thread 6 received from thread 7
/home/hzhou/MPI/lib/libmpi.so.0(MPL_backtrace_show+0x39) [0x7f7070e3d7a9]
/home/hzhou/MPI/lib/libmpi.so.0(+0x416998) [0x7f7070df0998]
/home/hzhou/MPI/lib/libmpi.so.0(+0x3de27c) [0x7f7070db827c]
/home/hzhou/MPI/lib/libmpi.so.0(+0x3e127d) [0x7f7070dbb27d]
/home/hzhou/MPI/lib/libmpi.so.0(+0x3285d0) [0x7f7070d025d0]
/home/hzhou/MPI/lib/libmpi.so.0(+0x32ae23) [0x7f7070d04e23]
/home/hzhou/MPI/lib/libmpi.so.0(+0x32de59) [0x7f7070d07e59]
/home/hzhou/MPI/lib/libmpi.so.0(+0x32e1e0) [0x7f7070d081e0]
/home/hzhou/MPI/lib/libmpi.so.0(MPI_Recv+0x476) [0x7f7070b59126]
./t(+0x141c) [0x55cf5e4b841c]
/lib/x86_64-linux-gnu/libgomp.so.1(+0x1dc0e) [0x7f707098dc0e]
/lib/x86_64-linux-gnu/libc.so.6(+0x94ac3) [0x7f70707dbac3]
/lib/x86_64-linux-gnu/libc.so.6(+0x126850) [0x7f707086d850]
Abort(1) on node 1: Internal error
/home/hzhou/MPI/lib/libmpi.so.0(MPL_backtrace_show+0x39) [0x7f7070e3d7a9]
/home/hzhou/MPI/lib/libmpi.so.0(+0x416998) [0x7f7070df0998]
/home/hzhou/MPI/lib/libmpi.so.0(+0x3de27c) [0x7f7070db827c]
/home/hzhou/MPI/lib/libmpi.so.0(+0x3e127d) [0x7f7070dbb27d]
/home/hzhou/MPI/lib/libmpi.so.0(+0x3285d0) [0x7f7070d025d0]
/home/hzhou/MPI/lib/libmpi.so.0(+0x32ae23) [0x7f7070d04e23]
/home/hzhou/MPI/lib/libmpi.so.0(+0x32de59) [0x7f7070d07e59]
/home/hzhou/MPI/lib/libmpi.so.0(+0x32e1e0) [0x7f7070d081e0]
/home/hzhou/MPI/lib/libmpi.so.0(MPI_Recv+0x476) [0x7f7070b59126]
./t(+0x141c) [0x55cf5e4b841c]
/lib/x86_64-linux-gnu/libgomp.so.1(GOMP_parallel+0x46) [0x7f7070984a16]
./t(+0x1382) [0x55cf5e4b8382]
/lib/x86_64-linux-gnu/libc.so.6(+0x29d90) [0x7f7070770d90]
```

problem:
MPI is not **thread-safe**!

MPI+Threads: Thread Levels

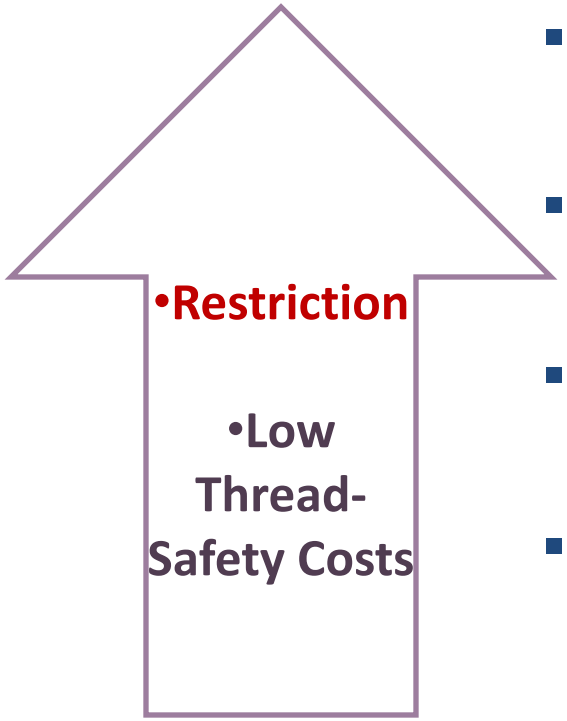
MPI_Init_thread(requested, provided)

MPI + Threads

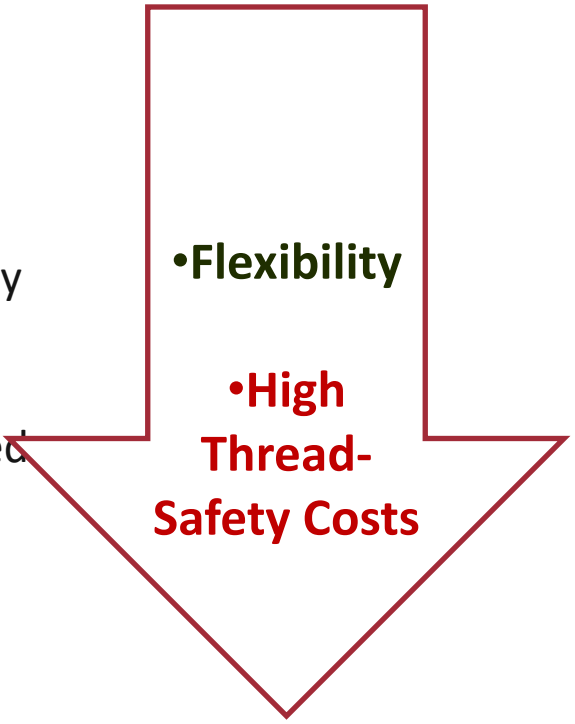
↓

Interoperability

Interoperation or thread levels:



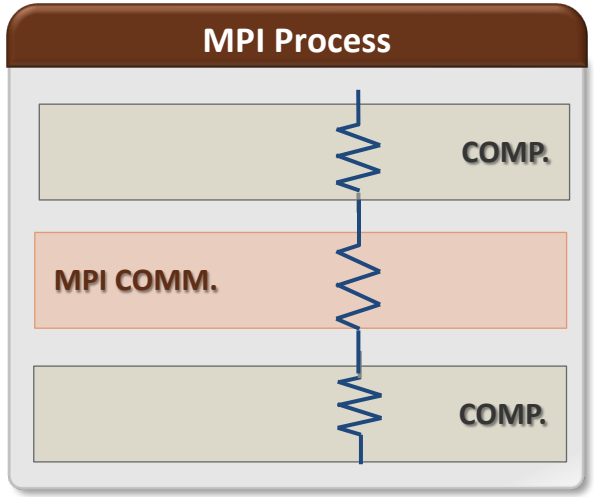
- MPI_THREAD_SINGLE
  - No additional threads
- MPI_THREAD_FUNNELED
  - Master thread communication only
- MPI_THREAD_SERIALIZED
  - Threaded communication serialized
- MPI_THREAD_MULTIPLE
  - No restrictions



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;
}
```

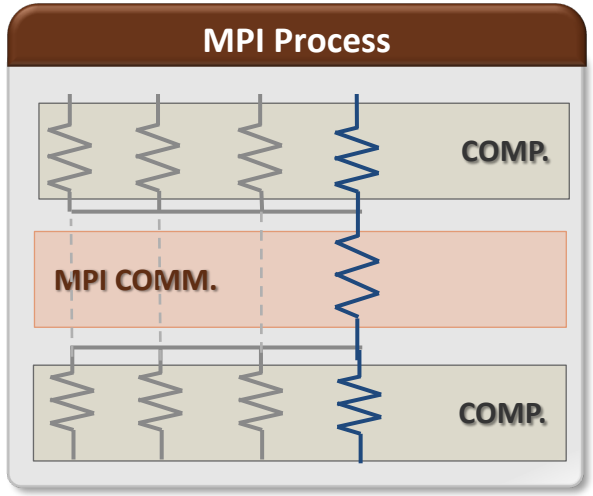


MPI_THREAD_FUNNELED

- 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]);
}
```

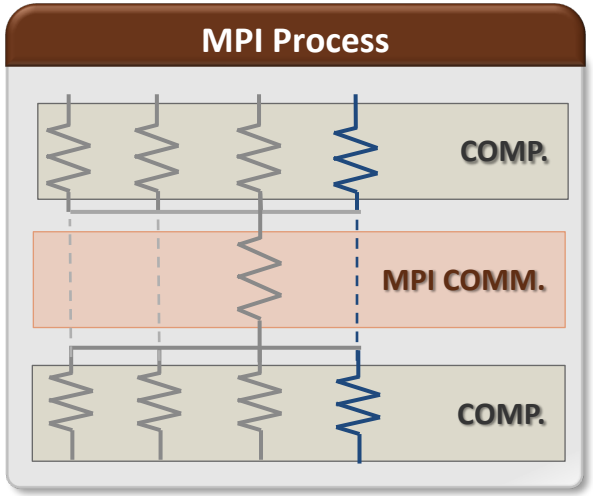


MPI_THREAD_SERIALIZED

- 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);
}
```

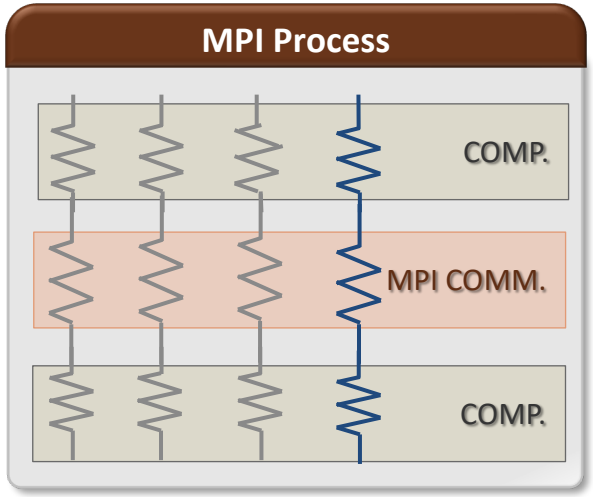


MPI_THREAD_MULTIPLE

- **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 */
}
```



Threads and MPI

- An implementation is not required to support levels higher than MPI_THREAD_SINGLE; that is, an implementation is not required to be thread safe
- A fully thread-safe implementation will support MPI_THREAD_MULTIPLE
- A program that calls MPI_Init (instead of MPI_Init_thread) should assume that only MPI_THREAD_SINGLE is supported
  - 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)*

MPI+Threads - What could go wrong - 2

- Use MPI_THREAD_MULTIPLE, our code is no longer crashing ✓
- Let's check the message matching

```
#pragma omp parallel num_threads(8) {
    if (rank == 0) {
        buf[0] = thread_id;
        MPI_Send(buf, count, MPI_INT, tag, comm);
    } else {
        MPI_Recv(buf, count, MPI_INT, tag, comm, &status);
        printf("Thread %d received from thread %d\n",
                thread_id, buf[0]);
    }
}
```

```
Thread 6 received from thread 7
Thread 7 received from thread 7
Thread 2 received from thread 7
Thread 0 received from thread 7
Thread 4 received from thread 7
Thread 1 received from thread 7
Thread 3 received from thread 7
Thread 5 received from thread 7
```

problem:
Race conditions on message buffers!

MPI+Threads - What could go wrong - 3

```
#pragma omp parallel num_threads(8) {
    void *buf = buffer_pool[thread_id];
    if (rank == 0) {
        buf[0] = thread_id;
        MPI_Send(buf, count, MPI_INT, tag, comm);
    } else {
        MPI_Recv(buf, count, MPI_INT, tag, comm, &status);
        printf("Thread %d received from thread %d\n",
               thread_id, buf[0]);
    }
}
```

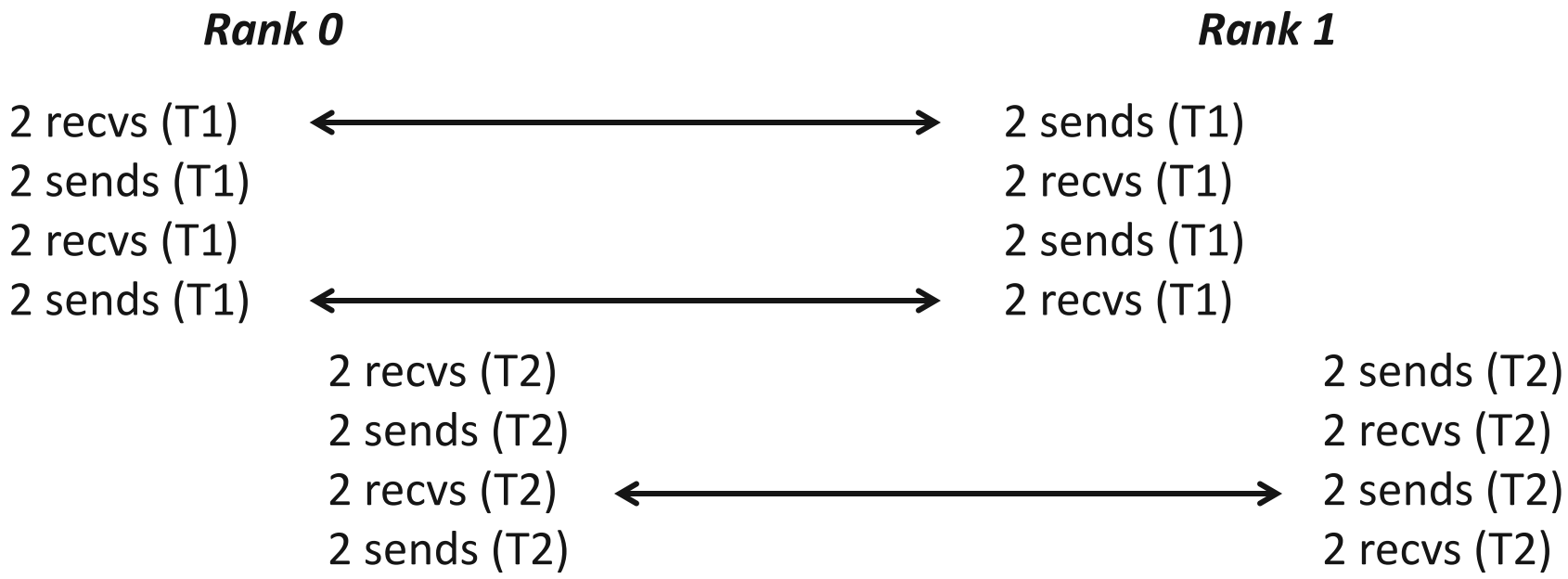
```
Thread 2 received from thread 0
Thread 6 received from thread 5
Thread 0 received from thread 6
Thread 5 received from thread 2
Thread 7 received from thread 3
Thread 1 received from thread 7
Thread 3 received from thread 1
Thread 4 received from thread 4
```

problem:
Indeterministic message order!

What could go wrong: 2 Processes, 2 Threads

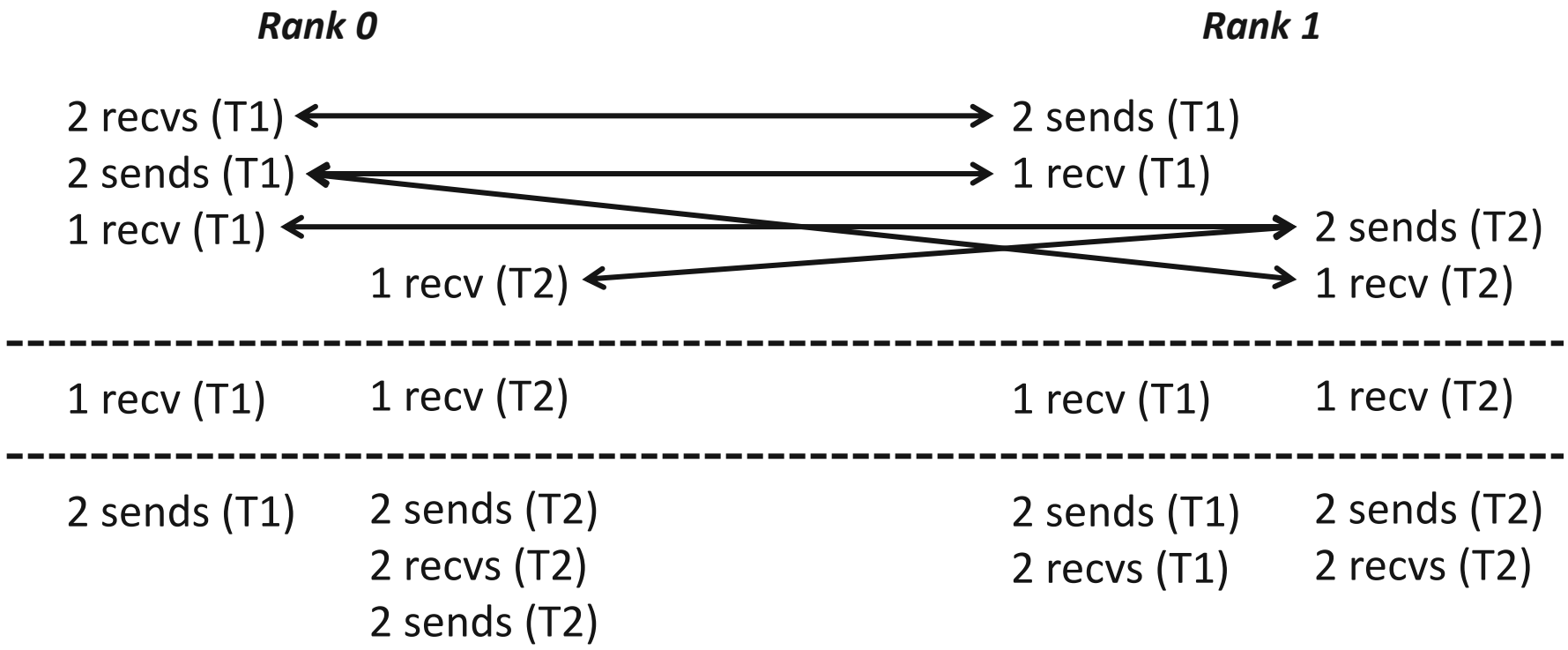
```
if (rank == 1) {
    MPI_Send(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    MPI_Send(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    MPI_Recv(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &stat);
    MPI_Recv(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &stat);

    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



- Every send matches a receive on the other rank

Possible Ordering of Operations in Practice

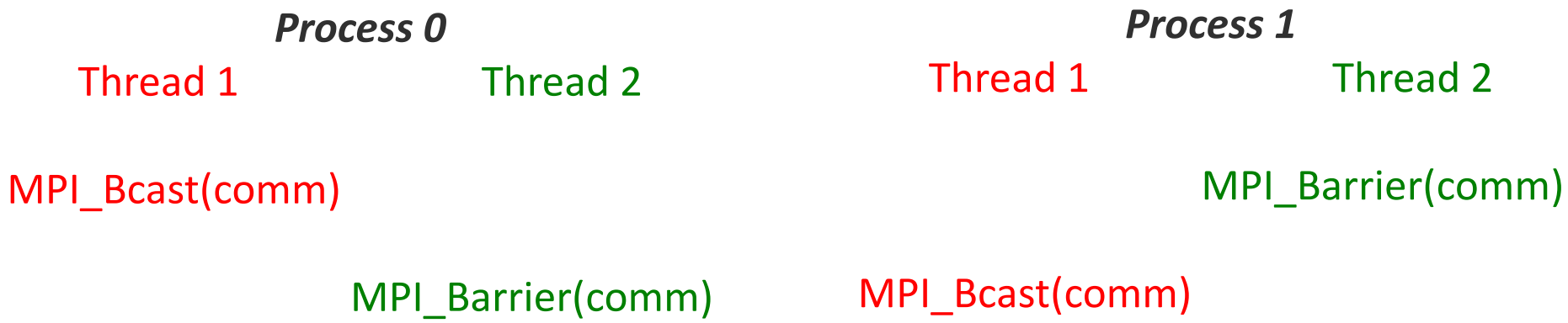


- Because the MPI operations can be issued in an arbitrary order across threads, all threads could block in a RECV call

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)

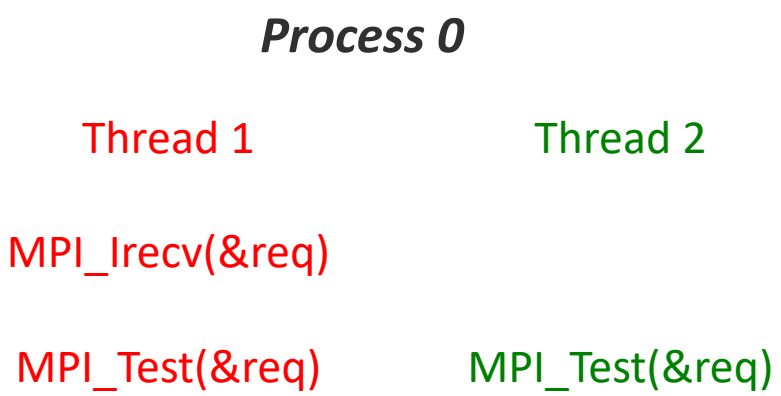
Ordering in MPI_THREAD_MULTIPLE: Incorrect Example with Collectives



- 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

Hands-on: try yourself

Ordering in MPI_THREAD_MULTIPLE: Incorrect Example with Object Management - request object

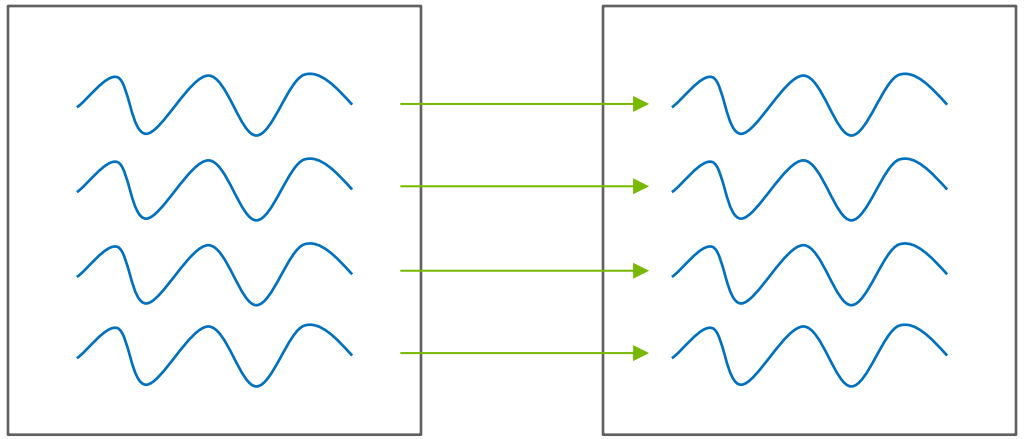


- Calling a completion function transfers the ownership of the request objects.
  - While MPI can make concurrent progress, concurrent setting the same request objects may corrupt it.

Exercise: Measure MPI+Threads Performance

- Benchmark: send messages concurrently in multiple threads
  - Use 2 nodes to measure internode performance
  - Per-thread latency
  - Per-thread bandwidth
  - Aggregate bandwidth
  - Vary the number of threads

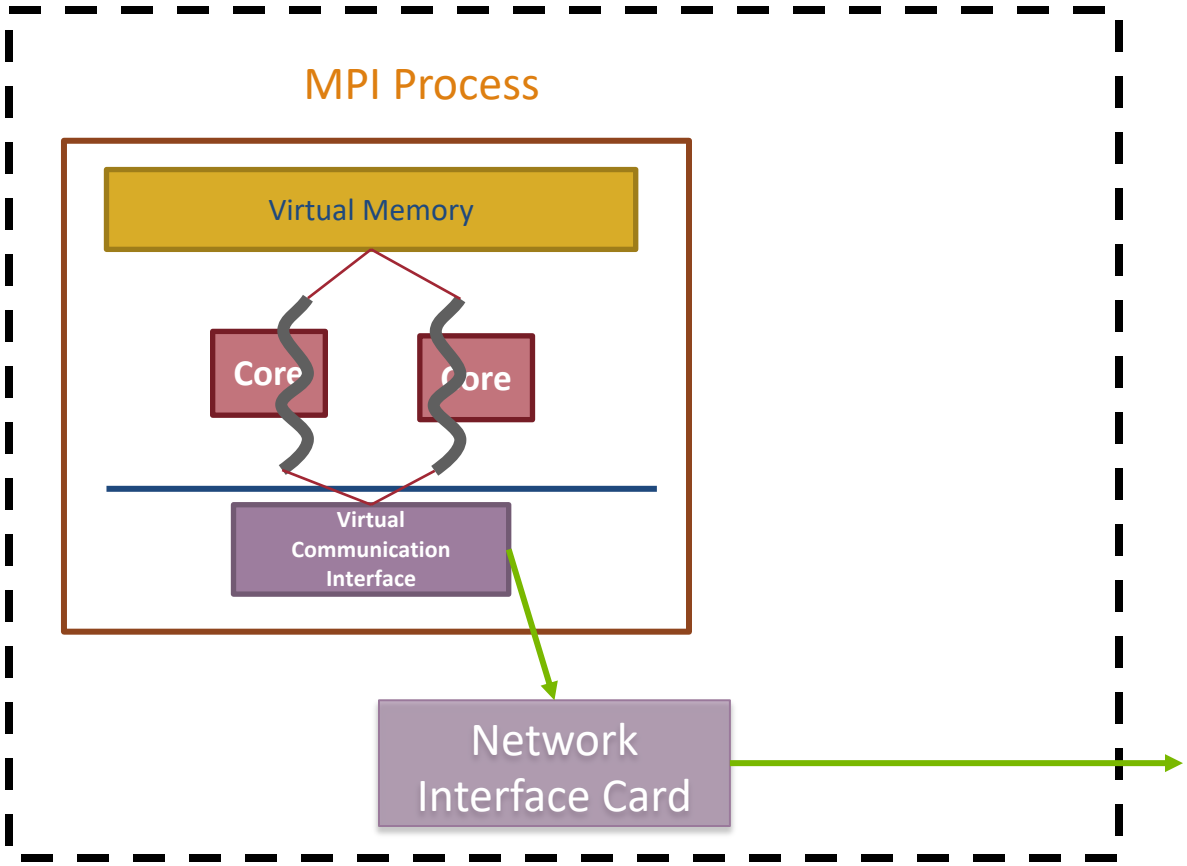
Pair-wise Messaging



Reference: hybrid/bandwidth.c, hybrid/latency.c

How to improve MPI+Threads Performance

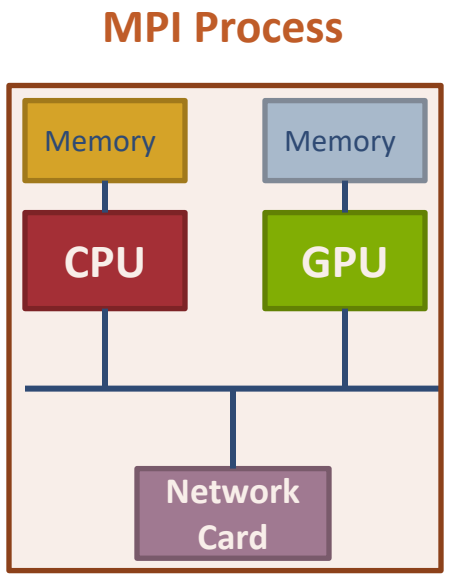
- Two contention areas
  1. Internal contexts
  2. Communication interface
- Ideas
  - Separate communication contexts via separate communicators
  - Enable multiple “Virtual Communication Interfaces”
    - MPIR_CVAR_CH4_NUM_VCIS=N
- Exercise:
  - Improve MPI+Threads performance



MPI+GPU

MPI + GPU

- Execution Context
  - Asynchronous
- Memory
  - Hybrid
- Advantage: Throughput
- Disadvantage: Synchronization
- Communication
  - CPU driven
  - Latency optimization – GDRCopy or Fast memcpy
  - Bandwidth optimization - IPC or RDMA

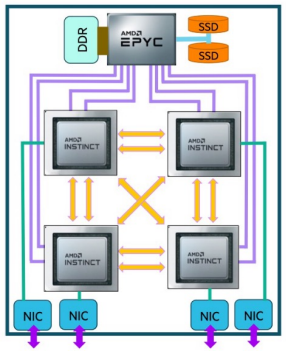


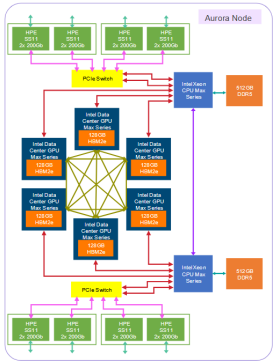
Exascale Supercomputers

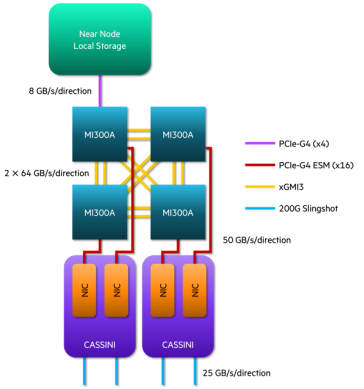
1 x AMD EPYC CPU
4 x AMD MI250X GPU

2 x Intel Xeon CPU
6 x Intel Data Center GPU

4 x AMD MI300A APU







Exercise: MPI+GPU

- It just works
- But complicated for performance

Allocating memory:

```
int dev_id = omp_get_default_device();

void *buf_gpu = omp_target_alloc(nbytes, dev_id);
/* intel extension */
void *buf_gpu = omp_target_alloc_device(nbytes, dev_id);
void *buf_host = omp_target_alloc_host(nbytes, dev_id);
Void *buf_shared = omp_target_alloc_shared(nbytes, dev_id);
```

MPI:

```
if (rank == 0) {
    MPI_Send(buf, count, MPI_INT, tag, comm);
} else {
    MPI_Recv(buf, count, MPI_INT, tag, comm, &status);
}
```

Compile:

```
$ mpicc -fiopenmp -fopenmp-targets=spir64 …
```

Hybrid Memory Types

Type of allocation	Initial location	Accessible on host?	Accessible on device?
Host	Host	Yes	Yes
Device	Device	No	Yes
Shared *	Host, Device, or Unspecified	Yes	Yes

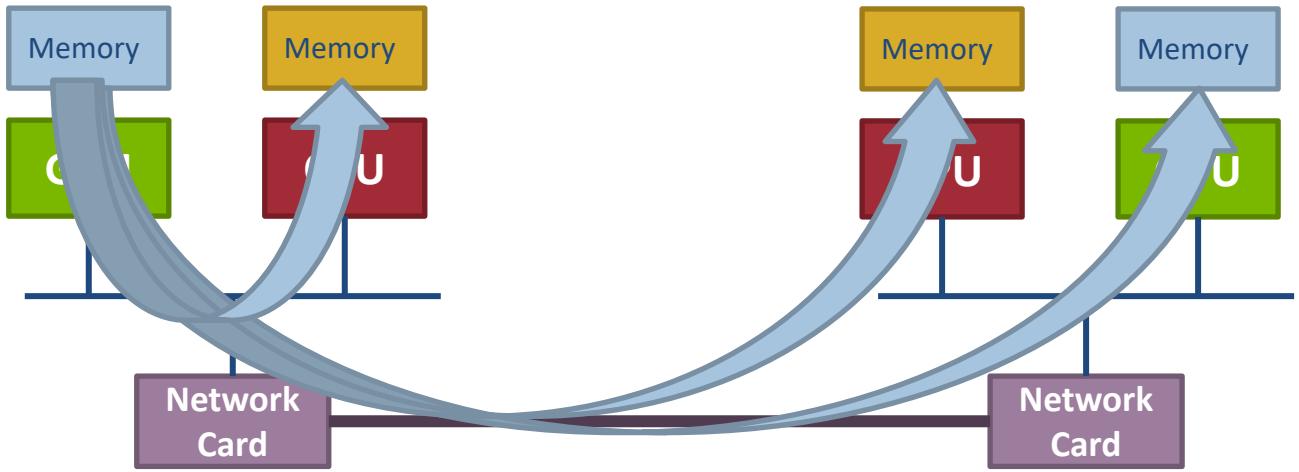
OpenMP memory allocation routine	Intel extension?	Type of allocation
`omp_target_alloc`	No	Device
`omp_target_alloc_device`	Yes	Device
`omp_target_alloc_host`	Yes	Host
`omp_target_alloc_shared`	Yes	Shared

* Shared memory may limit MPI messaging performance

How MPI Moves Data Between Different Types Of Memory?

GPUs have separate physical memory subsystem

How to move data between GPUs with MPI?



Simple answer: For modern GPUs and GPU-aware MPI implementations, “just like you would with a non-GPU machine”

Exercise: GPU buffers with GPU-Unaware MPI implementations

- It is still common for older systems to use GPU-Unaware MPI by default
  - GPU-awareness may come with a cost of lowering CPU-only performance
  - GPU architecture is still evolving
- Fallback strategy: copy to host bounce buffer.
  - Extra latency
  - Wasted bandwidth
  - Loss of optimization opportunity
- It serve as a base-line performance check for MPI implementations.

```
Omp_target_memcpy(dst, src, len, src_off, dst_off, dst_dev, src_dev);
```

```
int host_id = omp_get_initial_device();
```

Hands on: disable MPI GPU support: MPIR_CVAR_ENABLE_GPU=0

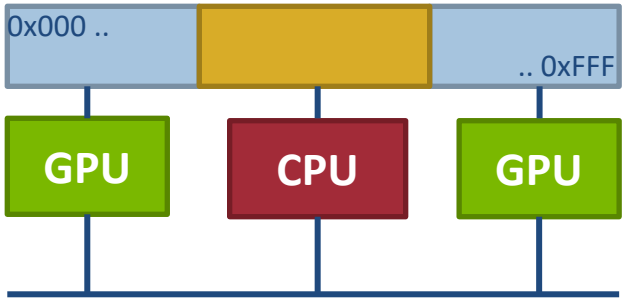
How MPI Moves Data Between Different Types Of Memory?

- Contiguous data
  - Intra-node
    - Small messages - sender copy to shared memory, receiver copy from shared memory
    - Large messages – Inter-Process Communication (IPC)
      - IPC is fast because it avoids double copy
  - Inter-node
    - Small messages – sender copy to NIC, NIC to NIC, receiver copy from NIC
    - Large message – GPU Remote Direct Memory Access (GPU RDMA)
      - RDMA is fast because it un-involves the CPU and GPU
- Non-contiguous data
  - Large segments – send the data piece-by-piece, receive the data piece-by-piece
  - Fragmented – pack and send as contig host memory, receive and unpack
- Techniques: staging buffers, pipelining, copy kernels

- Ideally implementations should do best under each scenarios.
- In reality, hardware architecture and software stacks keep evolving.
- Lean on MPI, but measure for performance.

Unified Virtual Addressing (UVA)

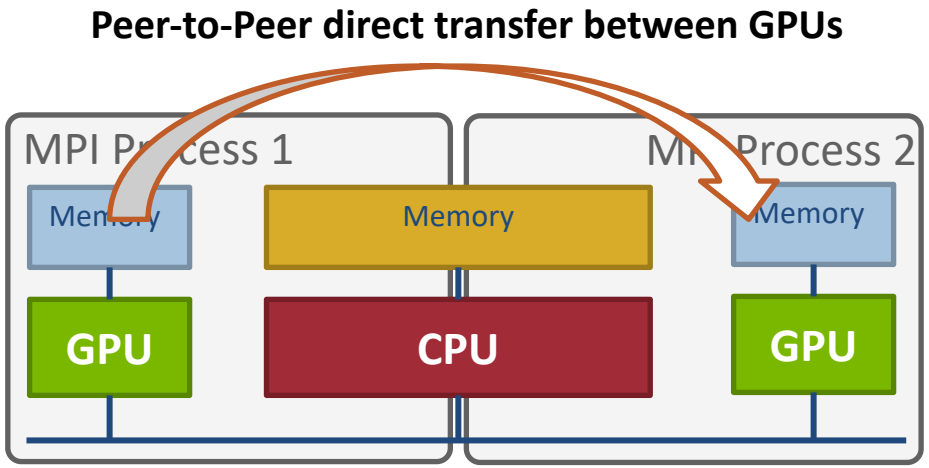
- UVA is a memory address management system supported in modern 64-bit architectures
  - Requires device driver support
- The same virtual address space is used for all processors, host or devices
- No distinction between host and device pointers
- The user can query the location of the data allocation 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

Intranode Communication with UVA

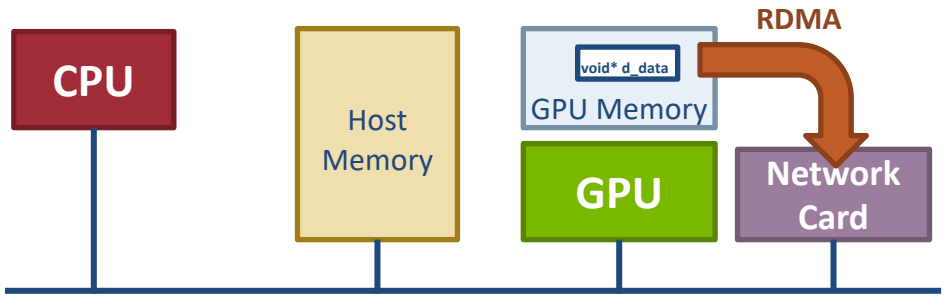
- Intranode Optimization
  - GPU peer-to-peer data transfers are possible
  - MPI can directly move data between GPU devices



GPU Direct IPC

Internode Communication with UVA

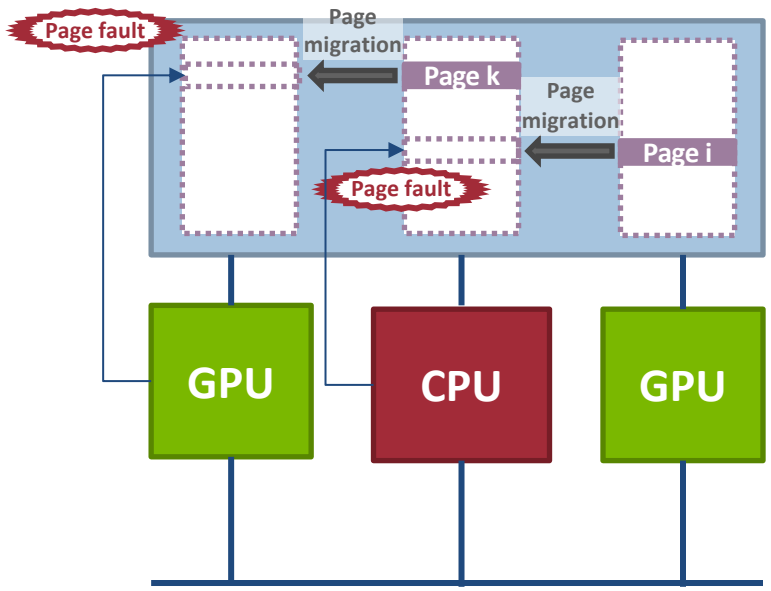
- Internode Optimization
  - GPUDirect RDMA enables network card transfer data without involving CPU or GPU



GPU Direct RDMA

A Note On Shared Memory

- Shared memory allows access from either host or device without explicit copy.
  - Intel: Unified Shared Memory (USM)
  - Nvidia: Managed Memory or Unified Shared Memory
  - AMD: Unified Memory
- 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
- Performance
  - Implicit hidden page migration cost
  - Good for programming model that neglects latency
  - Prevents MPI from optimizing



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

Exercise: MPI+GPU performance

* MPI+GPU
  * [ ] Allocate host buffer, device buffer, registered host buffer, and shared buffer
  * [ ] Send/recv between various types of buffers
  * [ ] Inter-node, intra-node, inter-device, inter-tiles

Q: How does the gpu messaging performance compared with host messaging?

Interoperability with MPI - Memory Allocation Kinds

- Standardized in MPI 4.1
- "mpi_memory_alloc_kinds"
  - Supplied via mpiexec, or MPI_Session_init.
  - It allows MPI implementation to optimize
  - Similar to MPI thread level
- "mpi_assert_memory_kinds"
  - Set to a communicator, a window, or a file
  - As a promise from user
  - Allows per-object-scope optimizations

```
% mpiexec -memory-alloc-kinds=mpi:system
-n 4 my_app
```

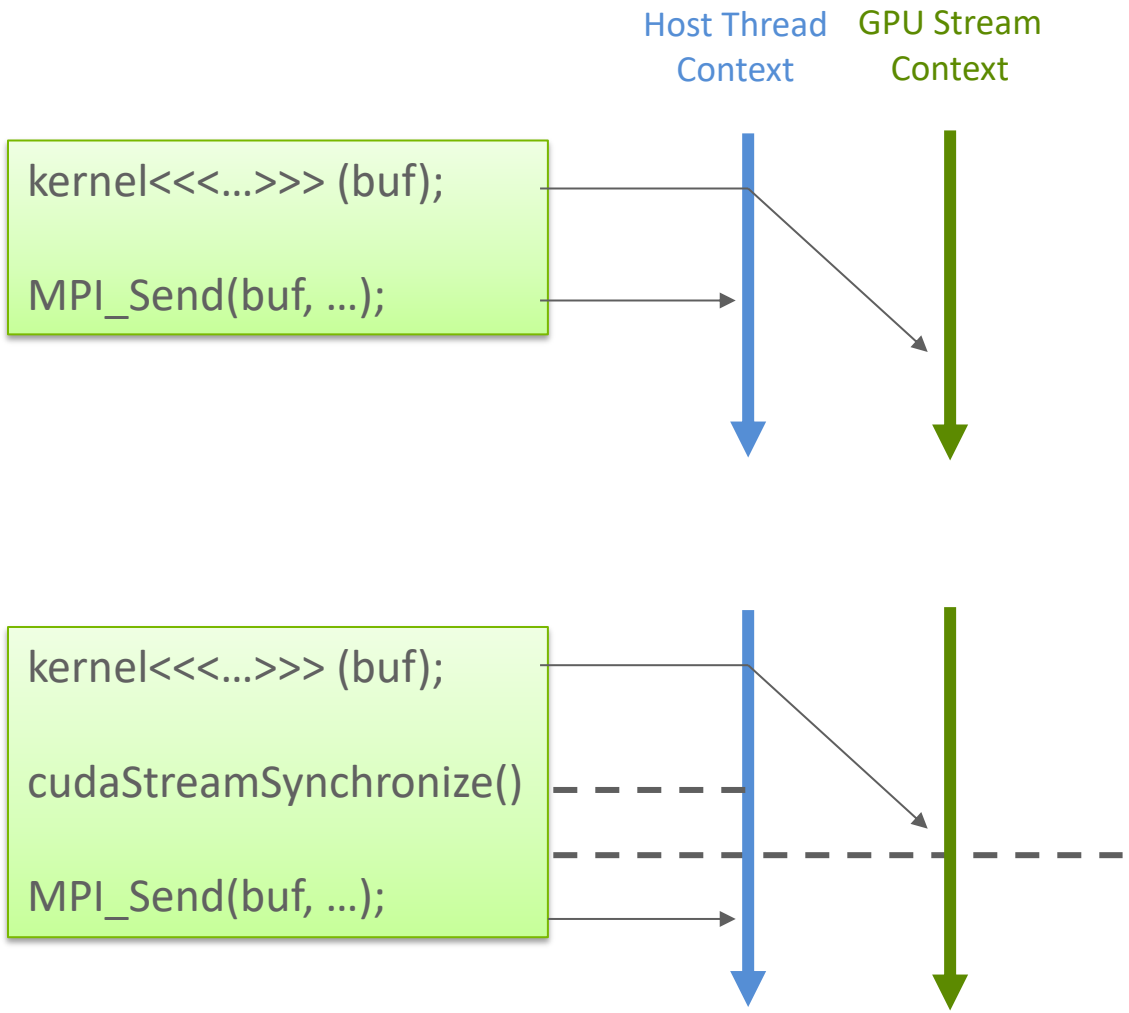
We do not need GPU support

Interoperability with MPI - Execution Context

- Explicit synchronization between the host and GPU execution is required.
- Stream synchronizations are expensive in performance.

Future MPI

- Stream-aware MPI functions
  - Enqueue MPI operations and triggered by the stream
- In-kernel MPI functions
  - e.g. MPI_Pready in kernel



Additional Exercises:

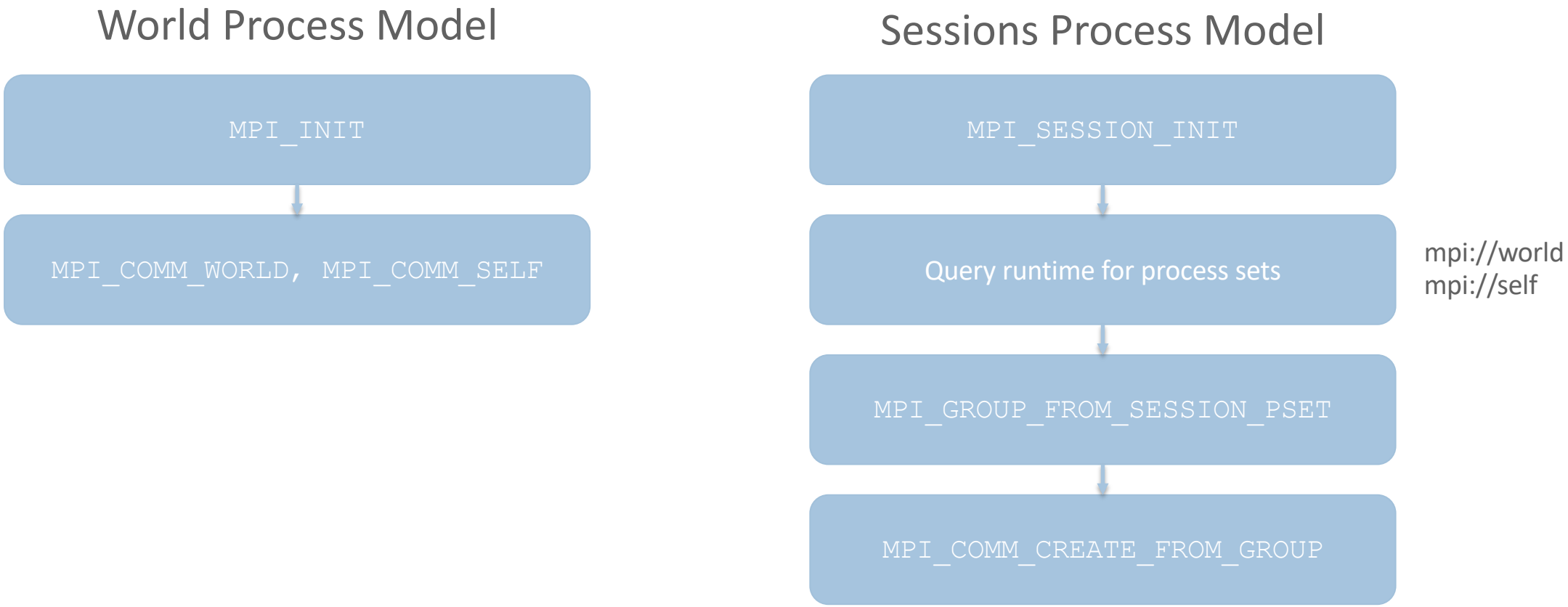
- Explore performance factors
  - CPU binding
  - Varying message sizes
  - For intra-node, compare against OpenMP-equivalent
- Benchmark collective performance
  - MPI_Bcast
  - MPI_Allreduce
  - Which collectives are performance-critical in your app?
- Stencil example using MPI+Threads
- Stencil example using MPI+GPU

New MPI Features

Sessions, Large Count, Partitioned Communication, ABI

MPI Sessions

An alternative method to initializing and creating MPI resources



ref: hello/hello_world.c, hello/hello_session.c

MPI Sessions

An alternative method to initializing and creating MPI resources

- Why?
  - More explicit initialization and object construction
    - No predefined MPI objects
  - Composability
    - Libraries can manage their own MPI resources by creating independent sessions. No need to check if MPI is initialized/finalized.
  - Tighter integration with resource manager
    - Job information, such as placement, can be queried. Process sets can be defined by system resources such as racks and switches
  - Fault tolerance? Malleability? MPI+X? Many open research areas to explore.
  - (Holmes et al., EuroMPI 2016)[1]

- Notes
  - Sessions and the WPM can coexist in the same process. Sessions also requires support for multiple init/finalize cycles. Some bugs may still be hiding in implementations ☺.

1. MPI Sessions: Leveraging Runtime Infrastructure to Increase Scalability of Applications at Exascale https://doi.org/10.1145/2966884.2966915

Large Count

Support for counts > INT_MAX

- `int -> MPI_Count` for count arguments
  - `MPI_Send -> MPI_Send_c`
  - Fortran interface polymorphism in `mpi_f08` module. Just use `MPI_SEND` and it will work.
- Why?
  - Common request over the years. Users were told to workaround by creating large datatypes instead.
  - Creating large datatypes workaround had known issues, particularly with ”v” and “w” collectives (Hammond et al., ExaMPI, 2014)[1]
- Notes
  - Led MPI Forum to “pythonize” the document to generate language bindings in the document.
  - Led the MPICH team to generate C language bindings and assiocated boiler plate (Zhou et al., EuroMPI, 2021)[2]
    - Profiling interface (PMPI)
    - Argument checking

1. To INT_MAX... and Beyond! Exploring Large-Count Support in MPI https://doi.org/10.1109/ExaMPI.2014.5/
2. Generating Bindings in MPICH https://arxiv.org/abs/2401.16547

Partitioned Communication

A new approach to MPI+X

- Send and receive buffer partitioning
  - Partitioned requests are persistent and match once during initialization
    - Data generation for partitions can be delegated to threads or accelerators
  - Individual partitions marked ready to send (`MPI_PREADY`)
  - Receiver can optionally check for arrived partitions (`MPI_PARRIVED`)
- Why?
  - One-sided model wrapped in familiar two-sided API
  - Potential benefits include reduced resource contention, message aggregation, “early-bird” effect (Gillis et al., ICPP, 2023)[1]
  - Ongoing work to define support for `MPI_PREADY` in a GPU context

1. Quantifying the Performance Benefits of Partitioned Communication in MPI https://dl.acm.org/doi/10.1145/3605573.3605599

A Standard MPI ABI

Build Once, Run Many

- Common definitions across implementations (Hammond et al., EuroMPI, 2023)[1]
  - `MPI_Aint` and other integer types
  - Handle types – `MPI_Comm`, …
  - Constant Values – `MPI_STATUS_IGNORE`, …
  - Data structures – `MPI_Status`, …
- Why?
  - Better support third-party languages – Python, Julia, Rust, etc.
  - Improved package management and distribution.
  - Easier to write tools
  - Prior efforts wi4mpi and MPI Trampoline demonstrated utility

1. MPI Application Binary Interface Standardization https://doi.org/10.1145/3615318.3615319

Thank You!