Advanced MPI

Slides Available at https://anl.box.com/v/yguo-isc-tutorial-2021

Pavan Balaji
Facebook
Email: pavanbalaji.work@gmail.com
Web: https://pavanbalaji.github.io/

Torsten Hoefler
ETH Zurich
Email: htor@inf.ethz.ch
Web: http://htor.inf.ethz.ch/

Antonio Pena
Barcelona Supercomputing Center
Email: antonio.pena@bsc.es
Web: https://www.bsc.es/pena-antonio

Yanfei Guo
Argonne National Laboratory
Email: yguo@anl.gov
Web: https://www.mcs.anl.gov/~yguo/

About the Speakers

- **Pavan Balaji**: Applied Research Scientist, Facebook
- **Torsten Hoefler**: Professor, ETH Zurich
- **Antonio Pena**: Senior Researcher, Barcelona Supercomputing Center
- **Yanfei Guo**: Assistant Computer Scientist, Argonne National Laboratory

- We are deeply involved in MPI standardization (in the MPI Forum) and in MPI implementation

What this tutorial will cover

- Some advanced topics in MPI
  - Not a complete set of MPI features
  - Will not include all details of each feature
  - Idea is to give you a feel of the features so you can start using them in your applications
- One-sided Communication (Remote Memory Access): MPI-2 and MPI-3
- Hybrid Programming with Threads, Shared Memory, and Accelerators (MPI-2 and MPI-3)
- Nonblocking Collective Communication (MPI-3)
- Topology-aware Communication (MPI-1 and MPI-2.2)
- New features in MPI-4 (might skip depending on time)

What is MPI?

Slides Available at https://anl.box.com/v/yguo-isc-tutorial-2021

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) is the latest version of the standard with minor corrections and features
- 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

Status of MPI-3.1 Implementations

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

Release dates are estimates; subject to change at any time "✗" indicates no publicly announced plan to implement/support that feature

Platform-specific restrictions might apply to the supported features

[1] Open Source but unsupported [2] No MPI_T variables exposed * Under development (*) Partly done

Important considerations while using MPI

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

Web Pointers

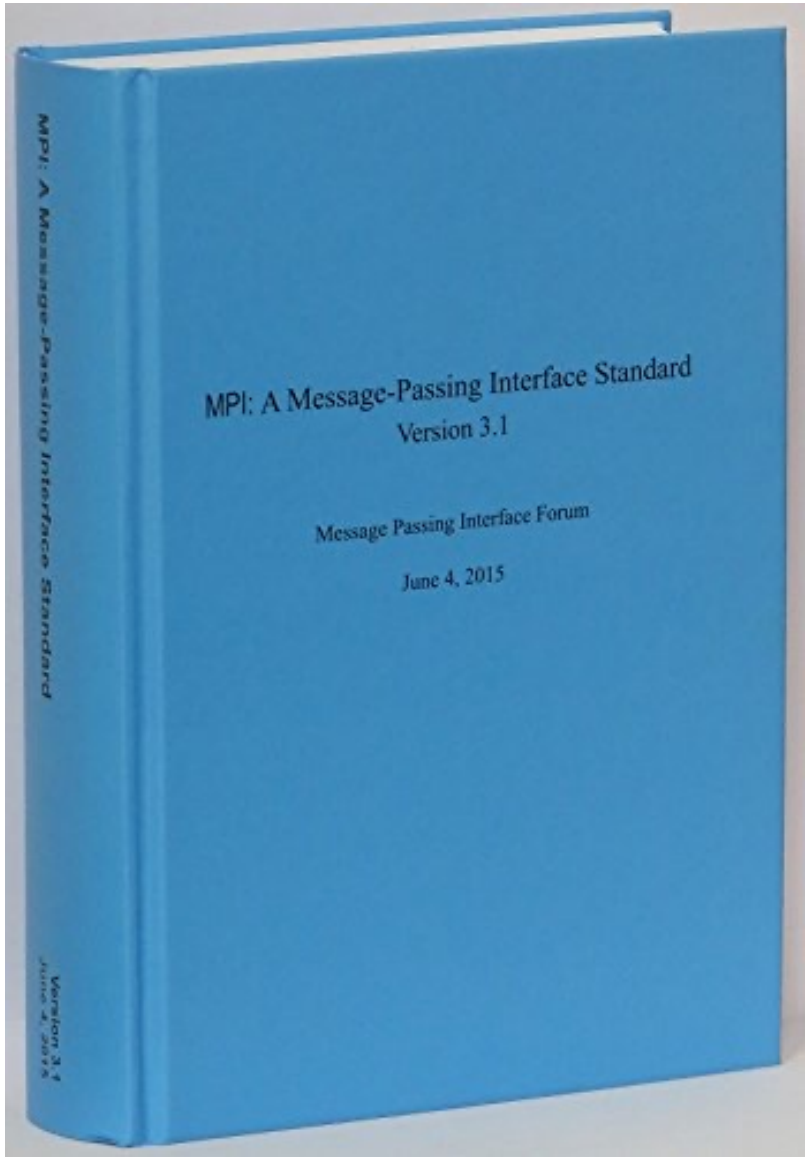
- MPI standard : http://www.mpi-forum.org/docs/docs.html
- MPI Forum : http://www.mpi-forum.org/

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

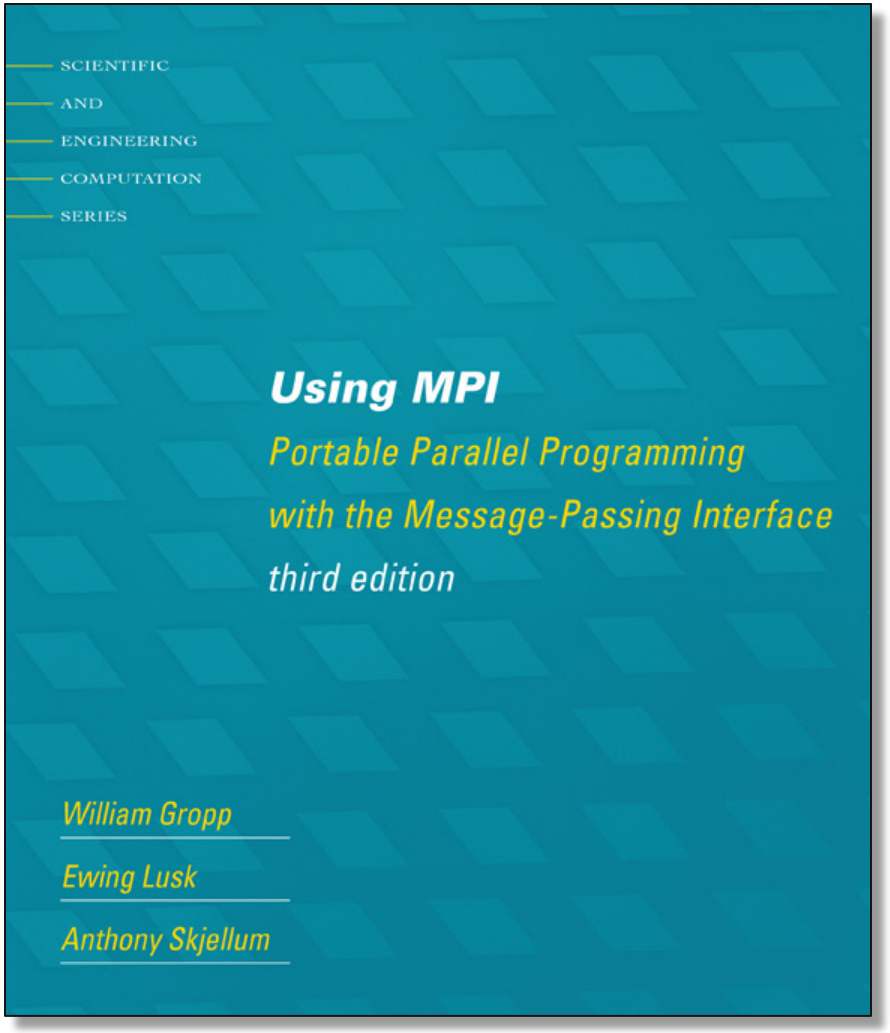
Latest MPI 3.1 Standard in Book Form

Available from amazon.com

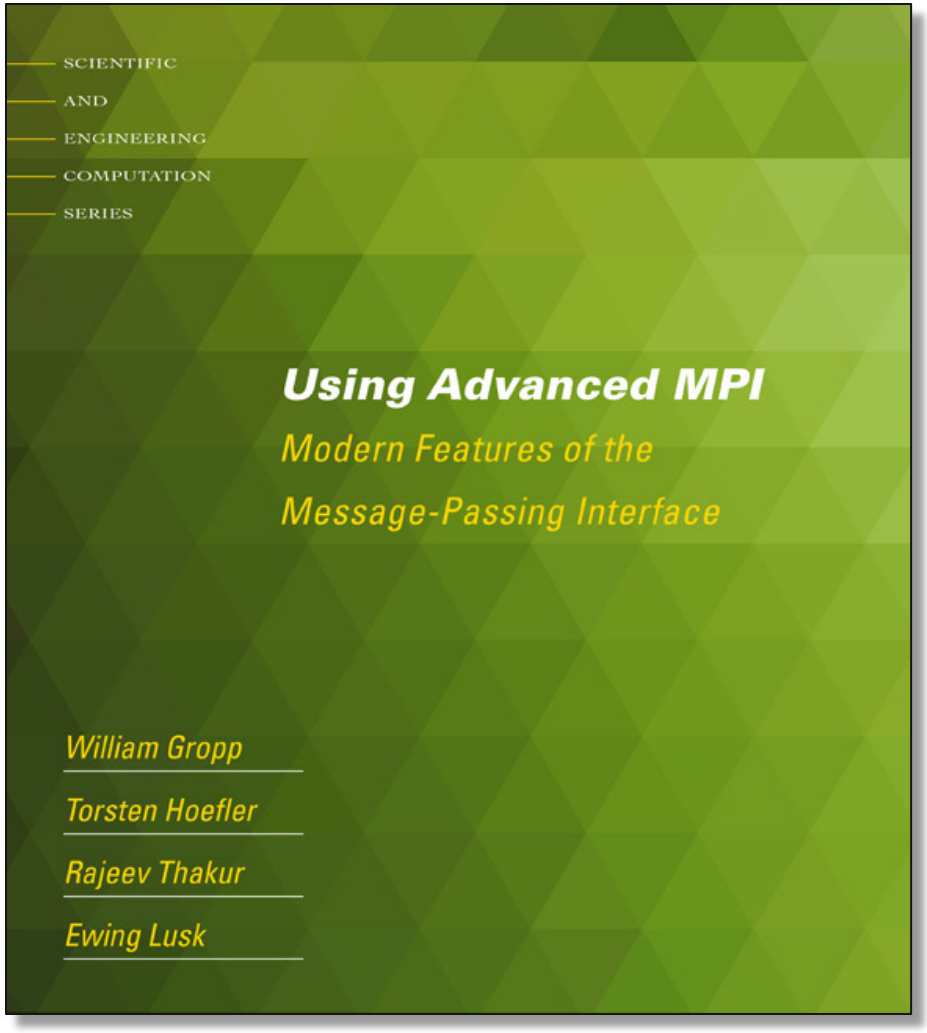
http://www.amazon.com/dp/B015CJ42CU/



Tutorial Books on MPI



Basic MPI

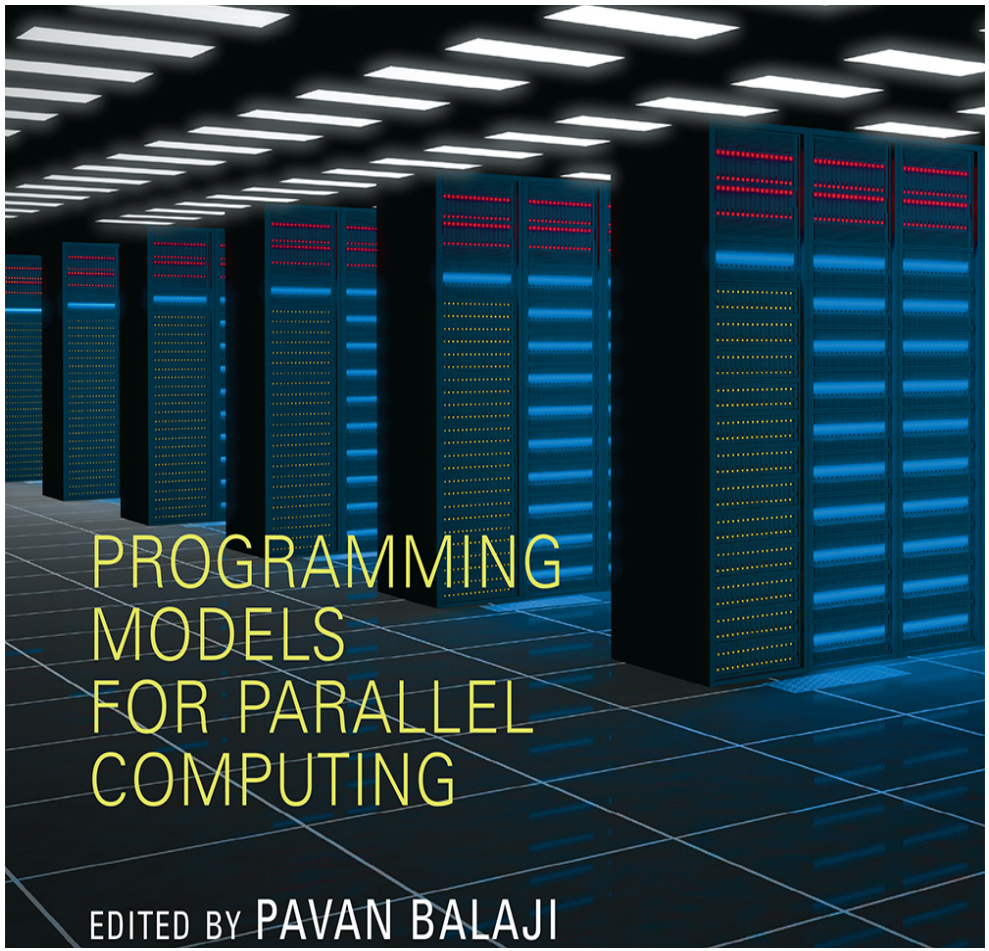


Advanced MPI, including MPI-3

Book on Parallel Programming Models

Edited by Pavan Balaji

- ***MPI:*** W. Gropp and R. Thakur
- ***GASNet:*** P. Hargrove
- ***OpenSHMEM:*** J. Kuehn and S. Poole
- ***UPC:*** K. Yelick and Y. Zheng
- ***Global Arrays:*** S. Krishnamoorthy, J. Daily, A. Vishnu, and B. Palmer
- ***Chapel:*** B. Chamberlain
- ***Charm++:*** L. Kale, N. Jain, and J. Lifflander
- ***ADLB:*** E. Lusk, R. Butler, and S. Pieper
- ***Scioto:*** J. Dinan
- ***SWIFT:*** T. Armstrong, J. M. Wozniak, M. Wilde, and I. Foster
- ***CnC:*** K. Knobe, M. Burke, and F. Schlimbach
- ***OpenMP:*** B. Chapman, D. Eachempati, and S. Chandrasekaran
- ***Cilk Plus:*** A. Robison and C. Leiserson
- ***Intel TBB:*** A. Kukanov
- ***CUDA:*** W. Hwu and D. Kirk
- ***OpenCL:*** T. Mattson



https://mitpress.mit.edu/books/programming-models-parallel-computing

Approach in this Tutorial

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

Access to example materials

Running Example: Stencil

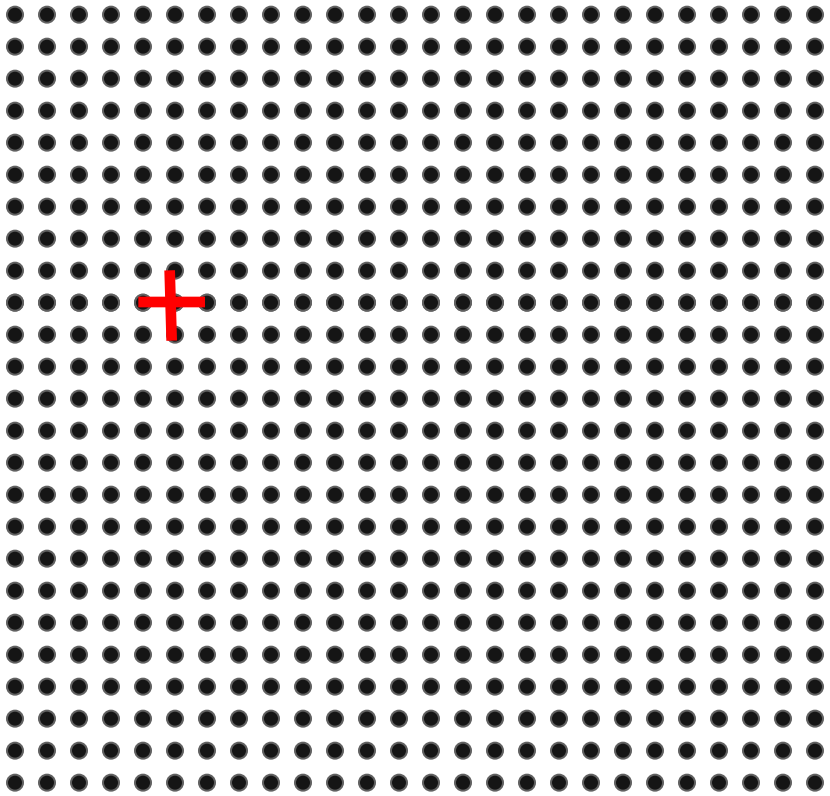
Slides Available at https://anl.box.com/v/yguo-isc-tutorial-2021

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

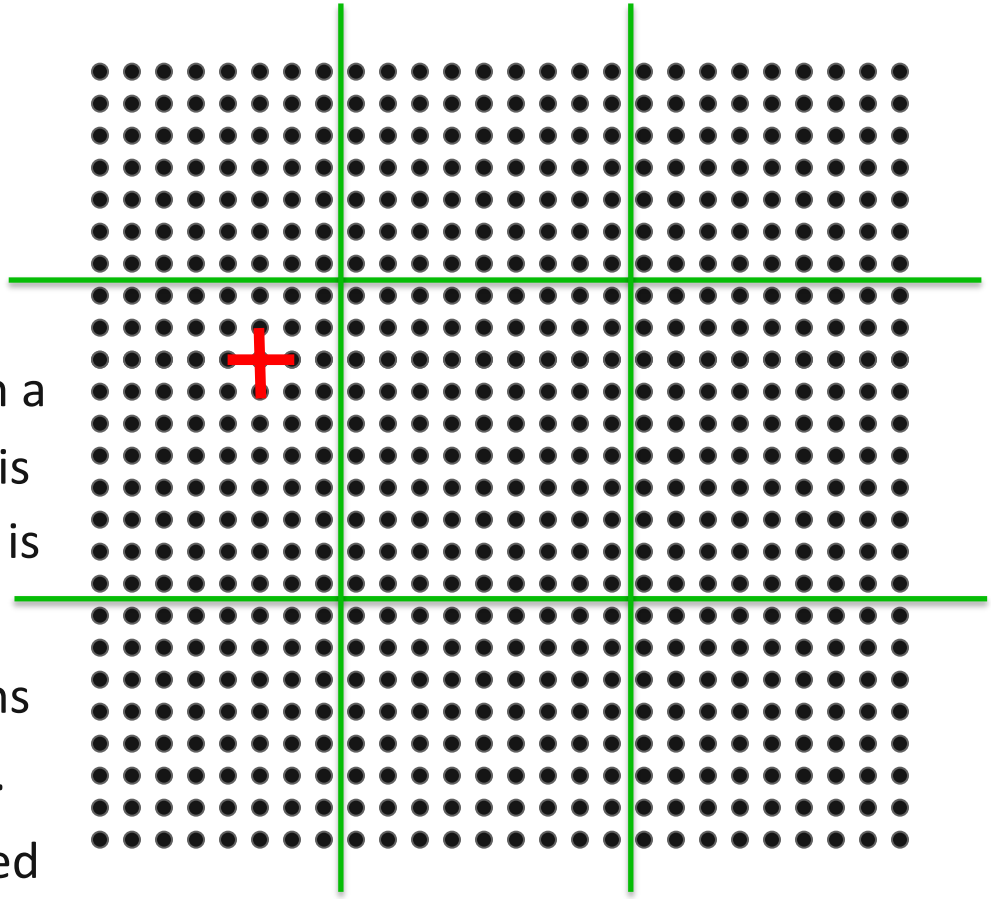
The Global Data Structure

- 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.

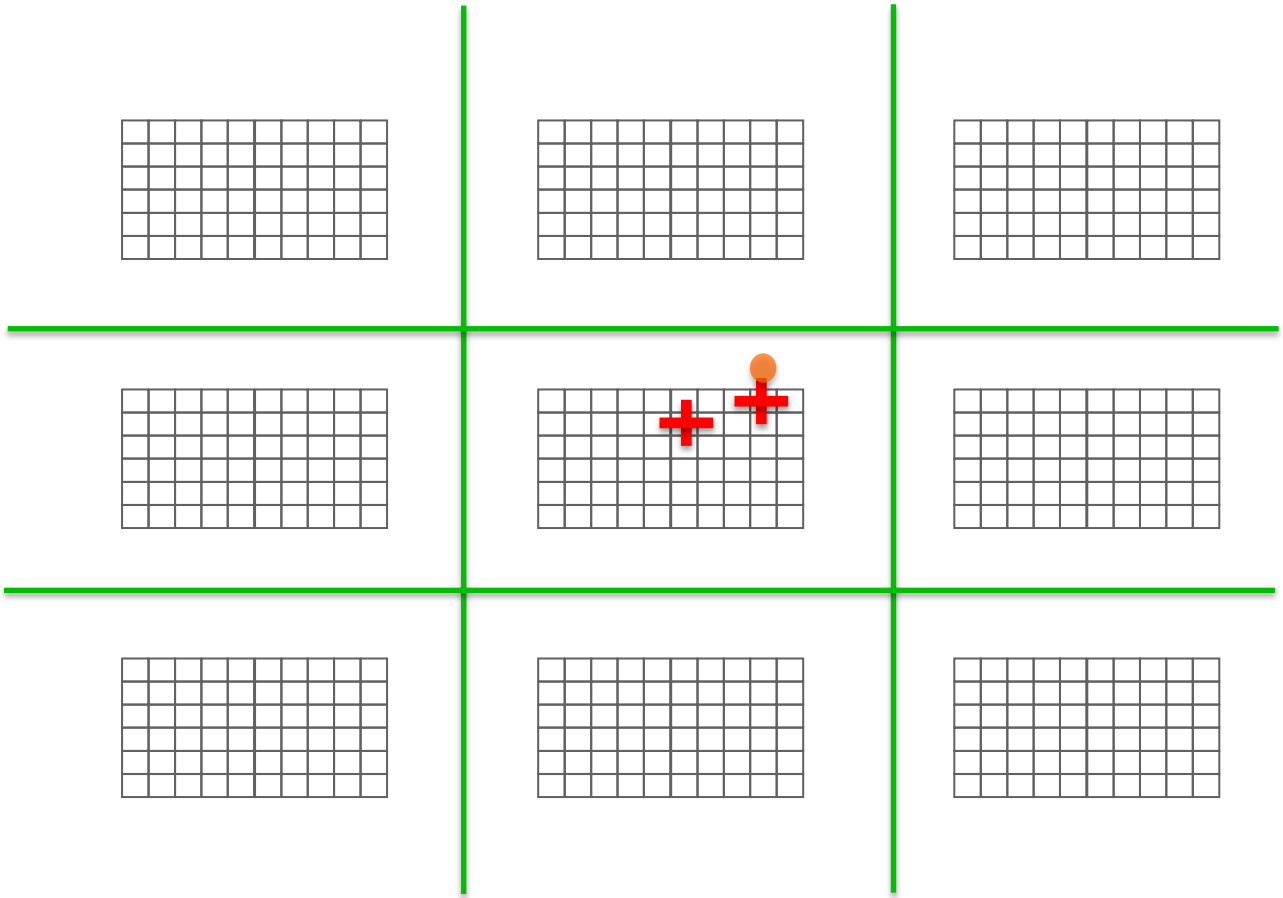


The Global Data Structure

- 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

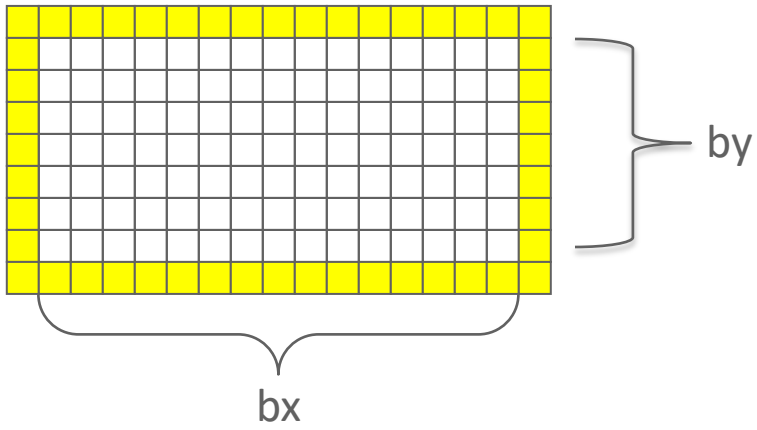


Necessary Data Transfers

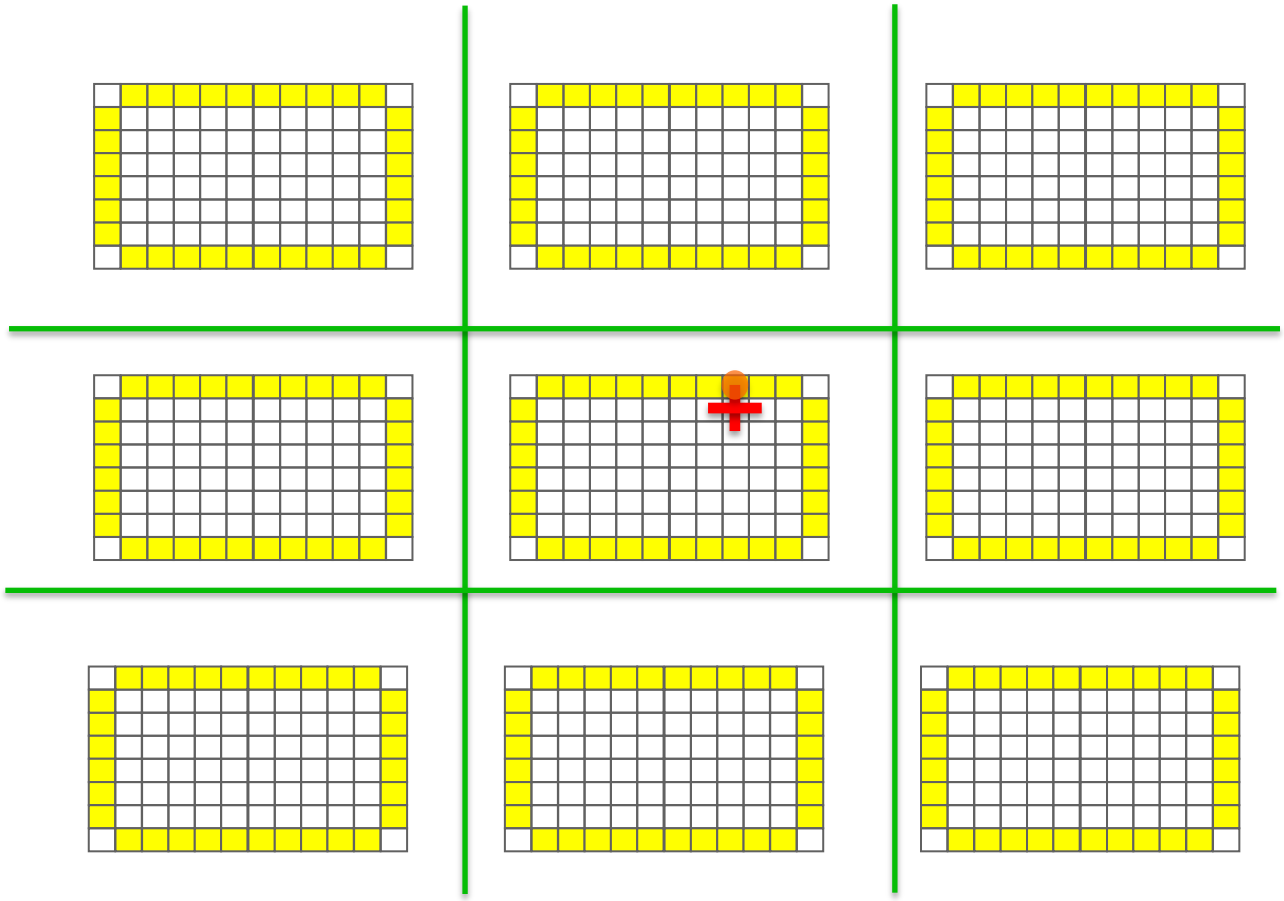


The Local Data Structure

- 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)

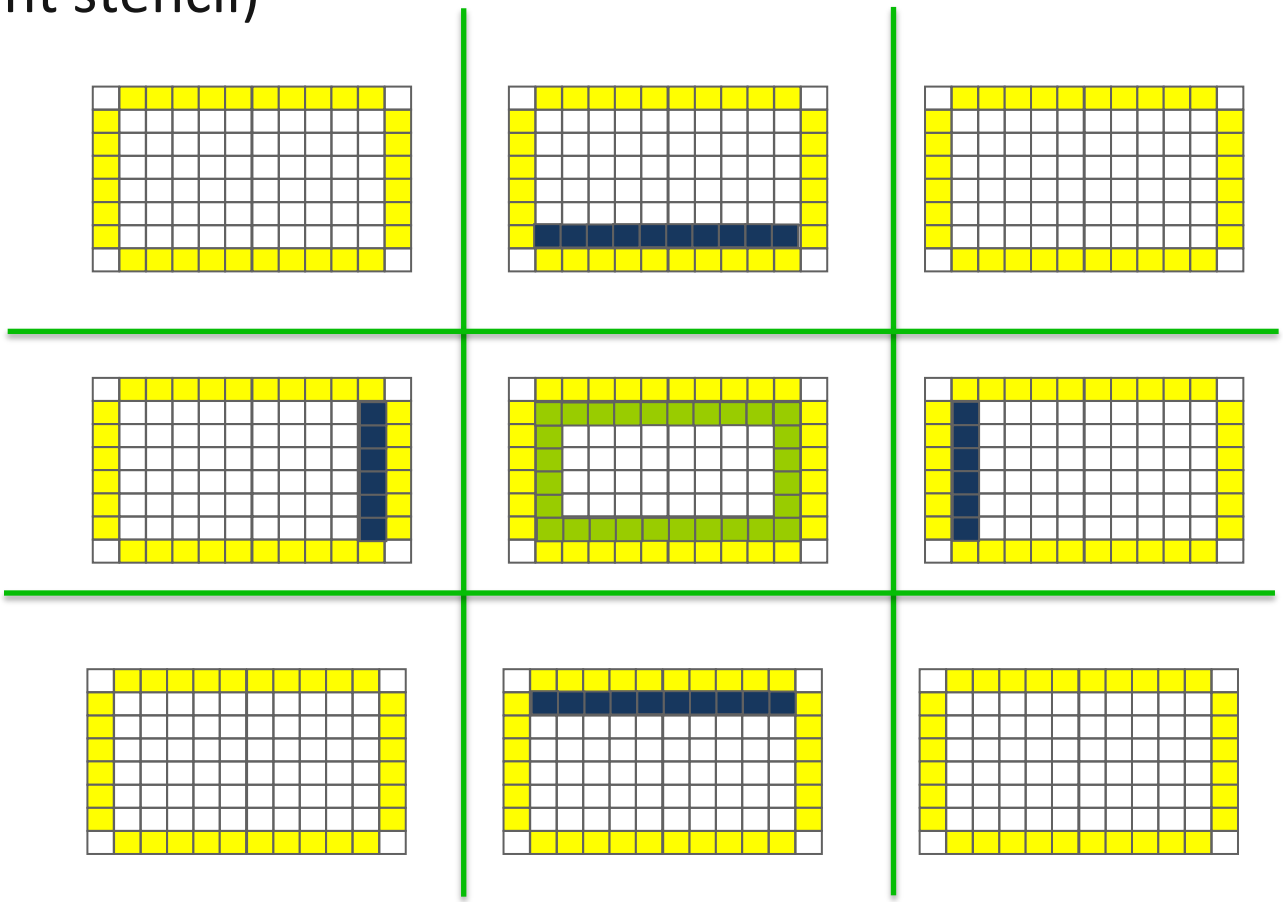


Necessary Data Transfers



Necessary Data Transfers

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



Example: Stencil with Nonblocking Send/recv

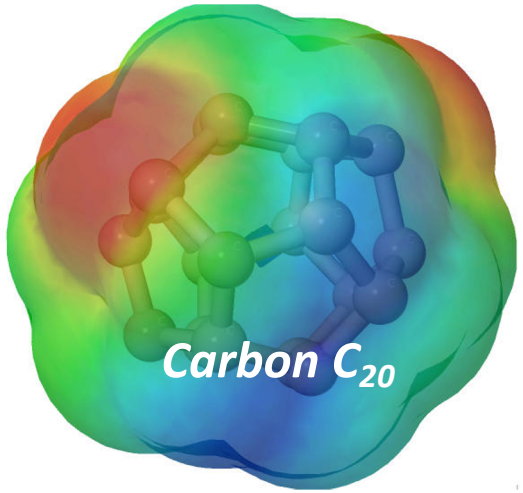
- *nonblocking_p2p/stencil.c*
- Simple stencil code using nonblocking point-to-point operations

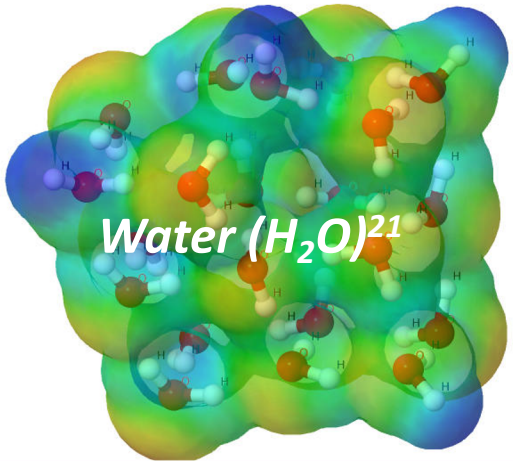
Running Example:
Block Sparse Matrix Multiplication

Slides Available at https://anl.box.com/v/yguo-isc-tutorial-2021

NWChem [1]

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

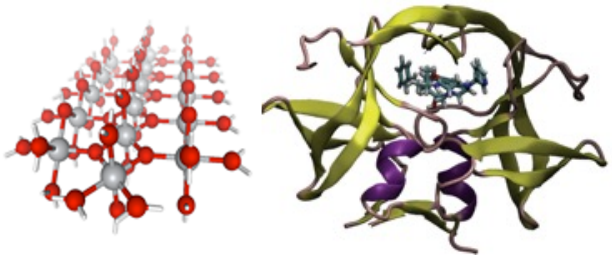




[1] M. Valiev, E.J. Bylaska, N. Govind, K. Kowalski, T.P. Straatsma, H.J.J. van Dam, D. Wang, J. Nieplocha, E. Apra, T.L. Windus, W.A. de Jong, "NWChem: a comprehensive and scalable open-source solution for large scale molecular simulations" Comput. Phys. Commun. 181, 1477 (2010)

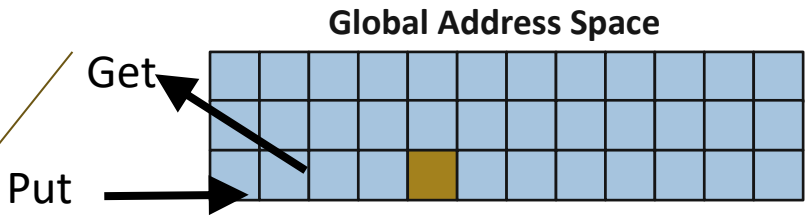
NWChem Communication Runtime

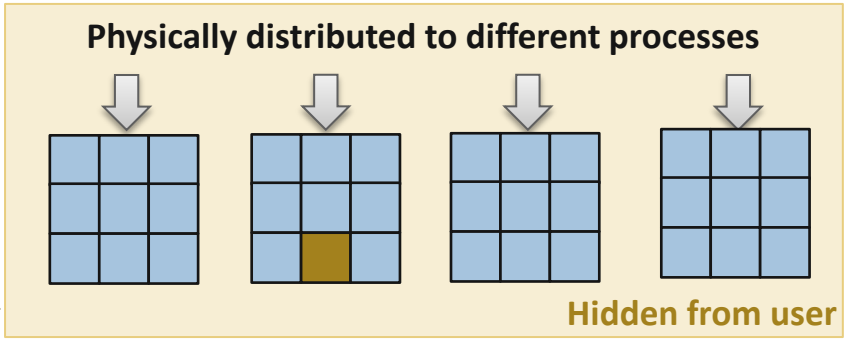
NWChem



Global Arrays [2]

Abstractions for distributed arrays





ARMCI : Communication interface for RMA[3]

ARMCI native ports

IB

DMMAP

...

ARMCI-MPI

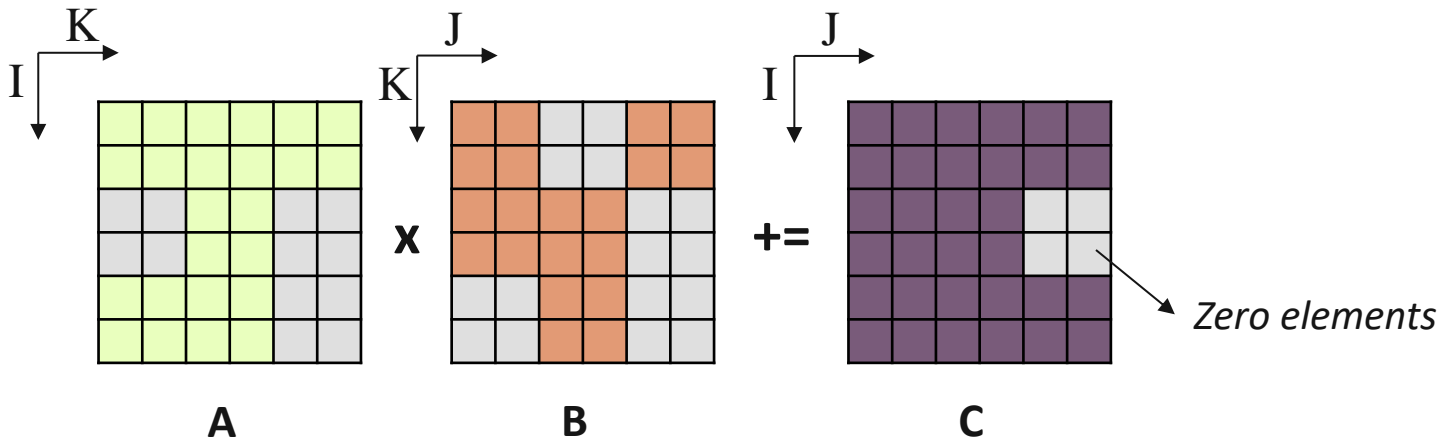
MPI RMA

Irregularly access large amount of remote memory regions

[2] http://hpc.pnl.gov/globalarrays
[3] http://hpc.pnl.gov/armci

Block Sparse Matrix Multiplication (BSPMM)

- Computing block sparse matrix multiplication: C = $\alpha A \times B + \beta C$



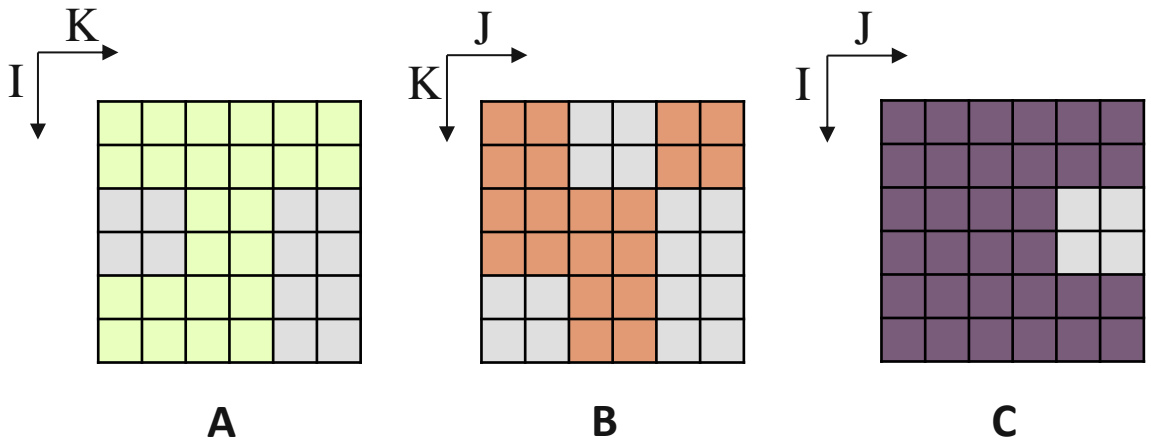
$$\mathrm{A} = \begin{bmatrix} a11 & \cdots & a1k \\ \vdots & \ddots & \vdots \\ ai1 & \cdots & aik \end{bmatrix}; B = \begin{bmatrix} b11 & \cdots & b1j \\ \vdots & \ddots & \vdots \\ bk1 & \cdots & bkj \end{bmatrix}; C = \begin{bmatrix} c11 & \cdots & c1j \\ \vdots & \ddots & \vdots \\ ci1 & \cdots & cij \end{bmatrix}$$

$$c_{ij} = \alpha \sum_{k=1}^{K} a_{ik} b_{kj} + \beta c_{ij}, \qquad for\ i = 1, \ldots, I, and\ j = 1, \ldots, J.$$

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

The Global Data Structure

- Applications operate on large data sets that easily exceed the capacity of a single node
  - Resource sharing across nodes becomes necessary
  - "Allocate" matrices in the global address space that is physically distributed onto different nodes
- In the tutorial, the first process entirely allocates all three matrices for simplicity



The Parallel Approach

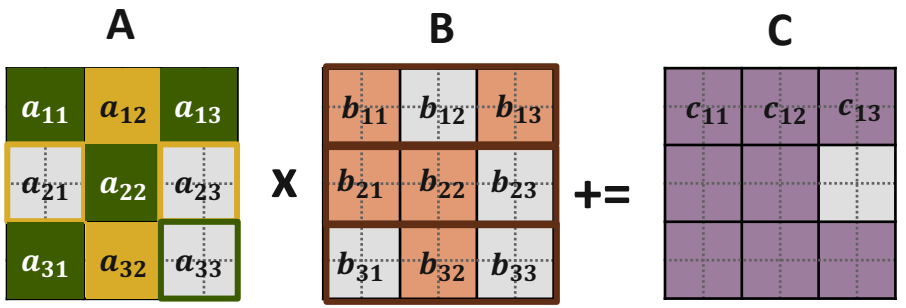
- Divide to independent subblocks and distribute over processes
- Compute dgemm of subblocks locally
- For each c_{ij}, sum up the results of corresponding subblock multiplication

rank 0

```
c11+=a11*b11
Skip a11*(b12)
c13+=a11*b13
Skip a13*(b31)
c12+=a13*b32
Skip a13*(b33)
...
```

rank 1

```
c11+=a12*b21
c12+=a12*b22
Skip a12*(b23)
Skip (a21)
...
```

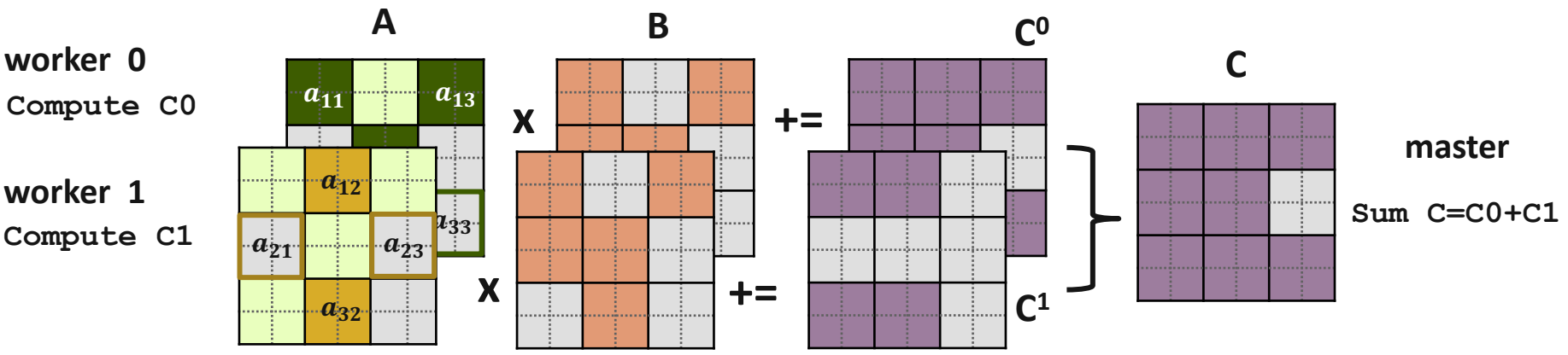


A-based parallelism

Set α=1, β=0 for simplicity, compute $C = AB$

Example: BSPMM with full matrices

- *blocking_p2p/bspmm_simple.c*
- Simple bspmm code using blocking point-to-point operations
  - Rank 0 (master) initializes all matrices and sends the entire A and B matrices to other processes (workers)
  - Each worker computes different blocks and sends the local matrix C to master
  - Master sums up the received results



Argonne NATIONAL LABORATORY

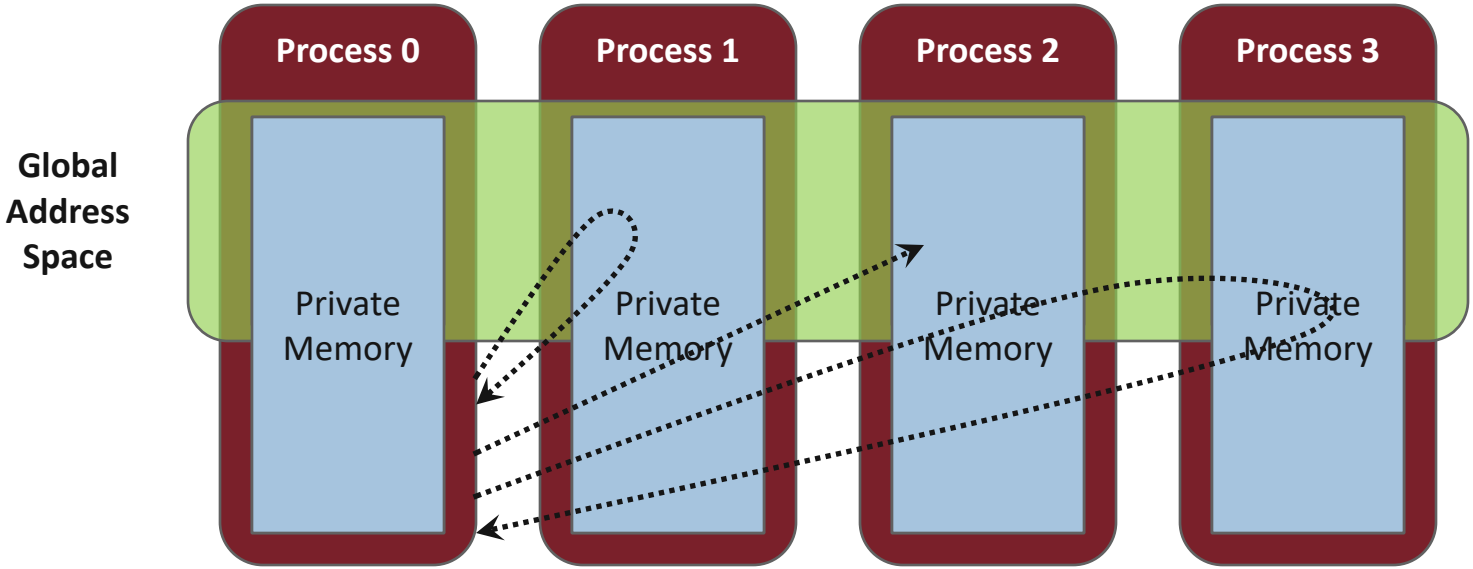
MPI One-sided Communication

Slides Available at https://anl.box.com/v/yguo-isc-tutorial-2021

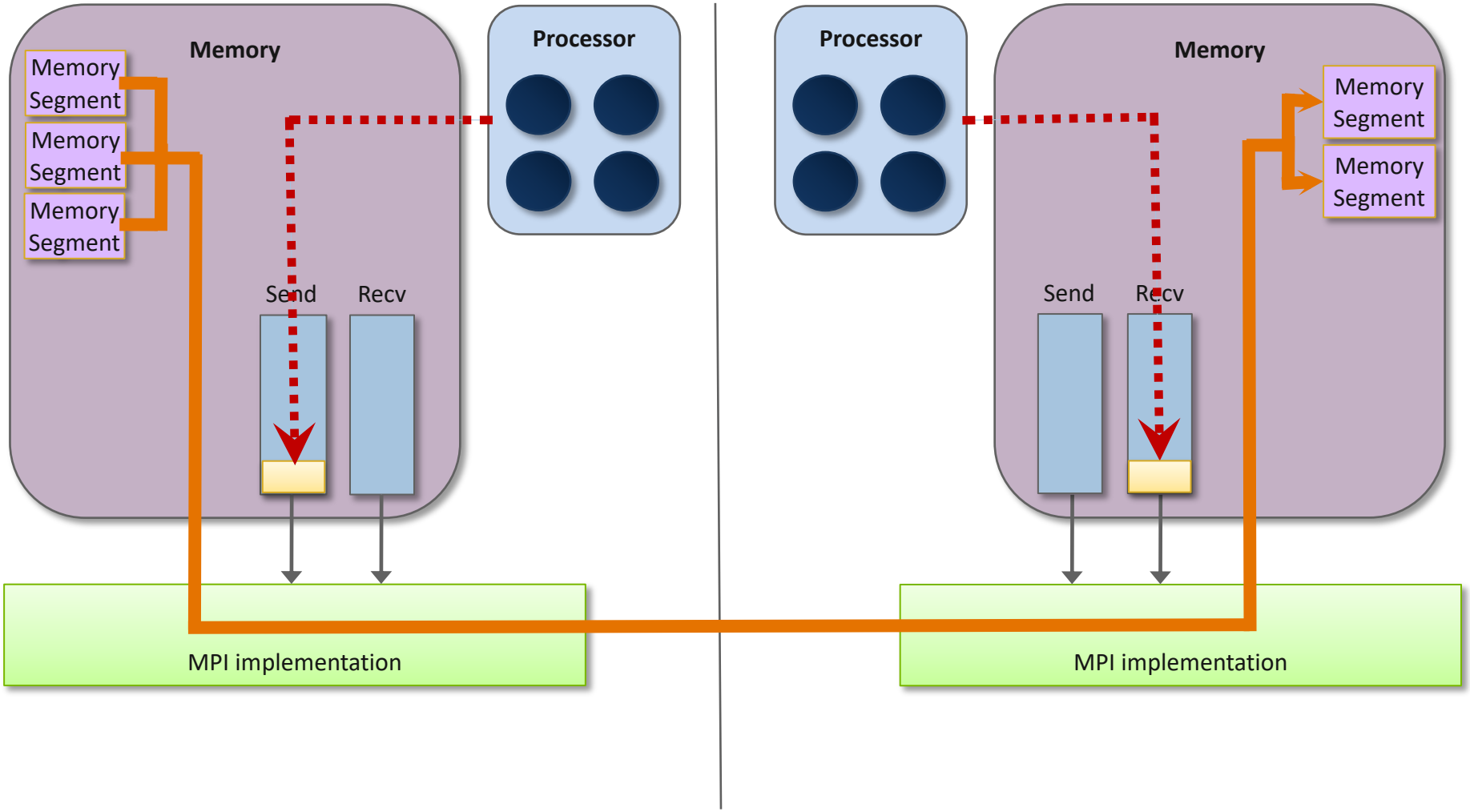
U.S. DEPARTMENT OF ENERGY

One-sided Communication

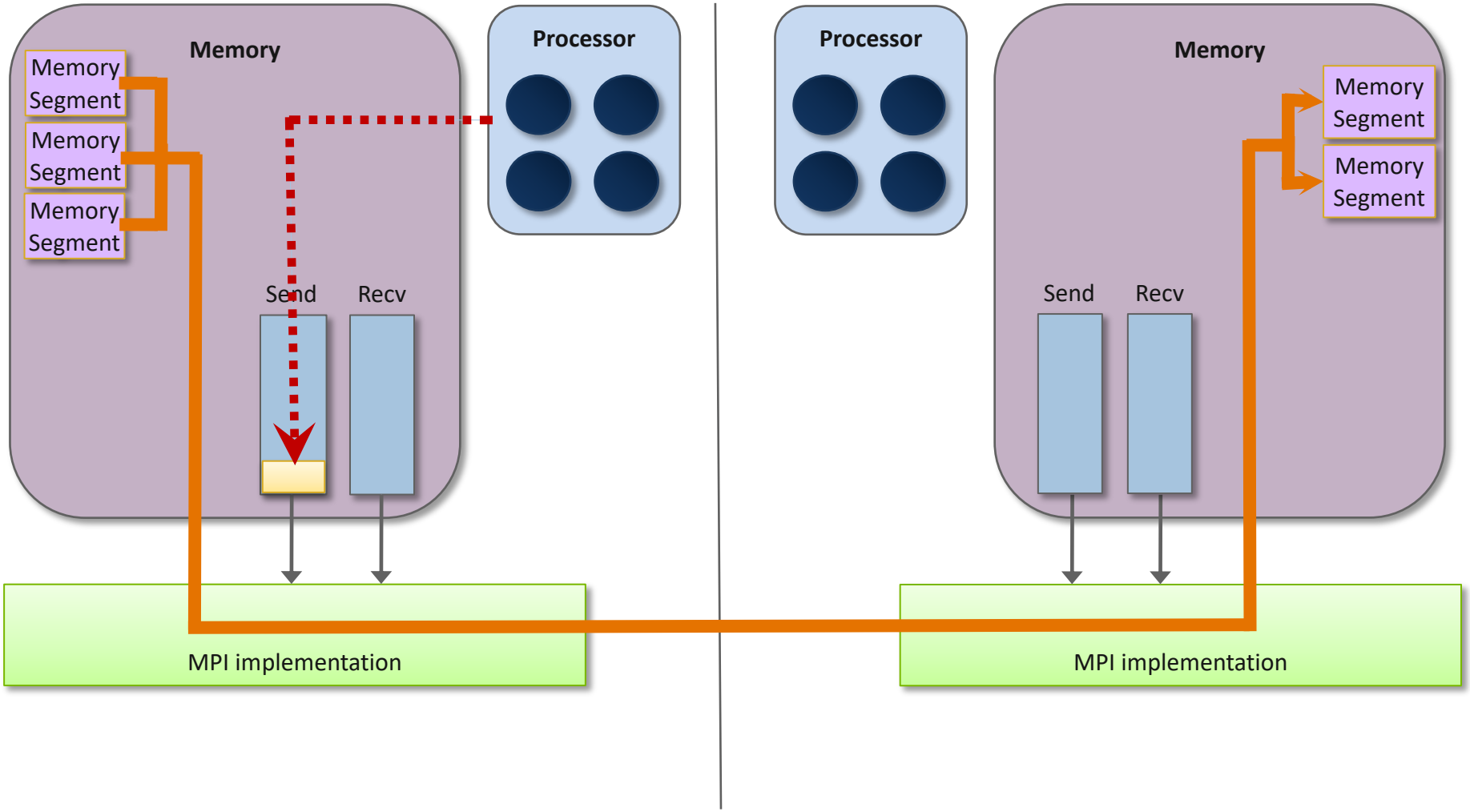
- The basic idea of one-sided communication models is to decouple data movement with process synchronization
  - Should be able to move data without requiring that the remote process synchronize
  - Each process exposes a part of its memory to other processes
  - Other processes can directly read from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



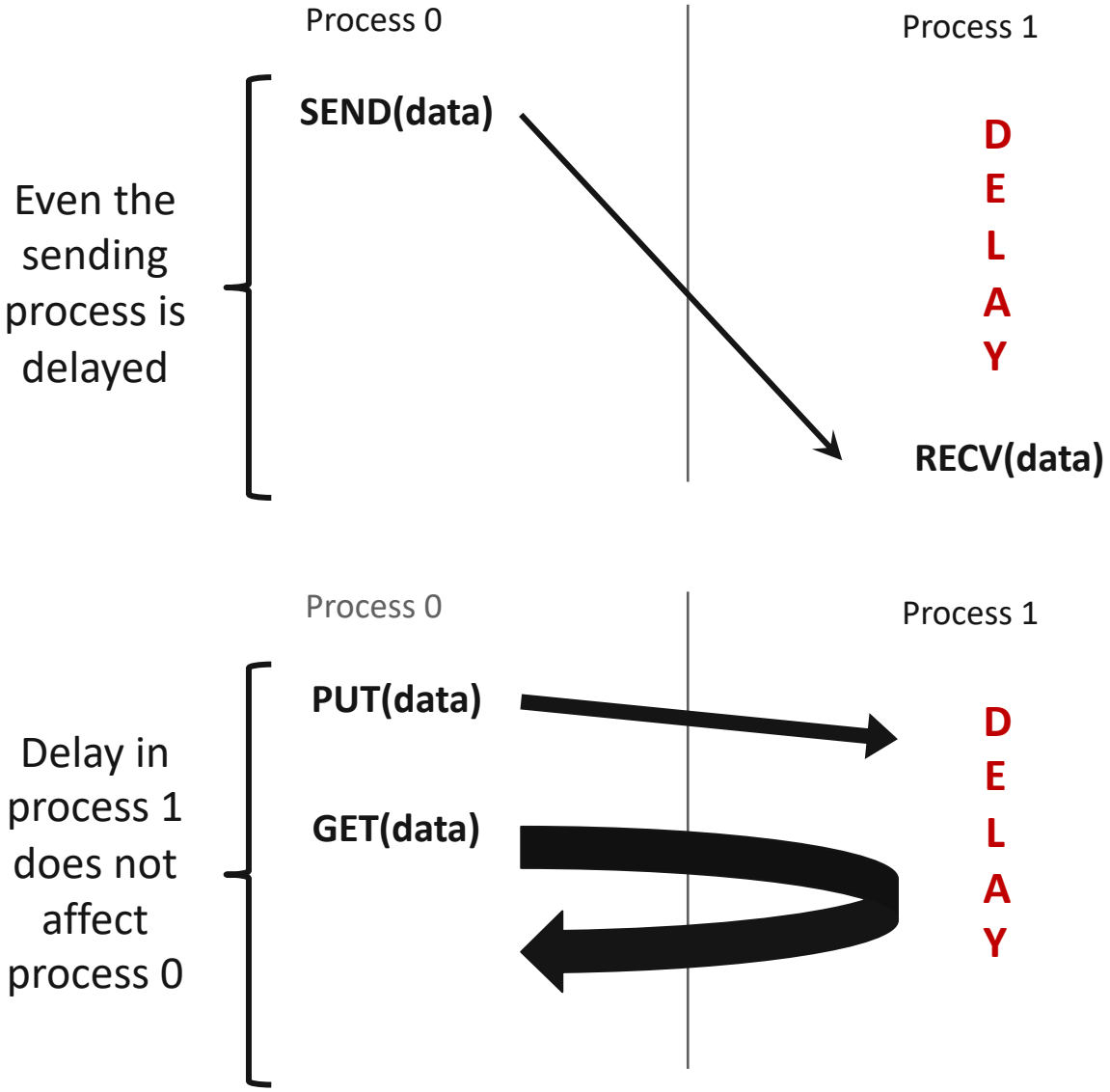
Two-sided Communication Example



One-sided Communication Example



Comparing One-sided and Two-sided Programming

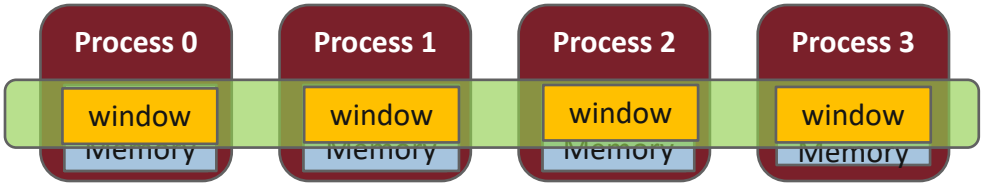


What we need to know in MPI RMA

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

Creating Public Memory

- Any memory used by a process is, by default, only locally accessible
  - X = malloc(100);



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

Window creation models

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

MPI_WIN_ALLOCATE

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

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

Example with MPI_WIN_ALLOCATE

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

    MPI_Init(&argc, &argv);

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

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

    MPI_Win_free(&win);

      MPI_Finalize(); return 0;
}
```

MPI_WIN_CREATE

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

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

Example with MPI_WIN_CREATE

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

    MPI_Init(&argc, &argv);

    /* create private memory */
    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;
}
```

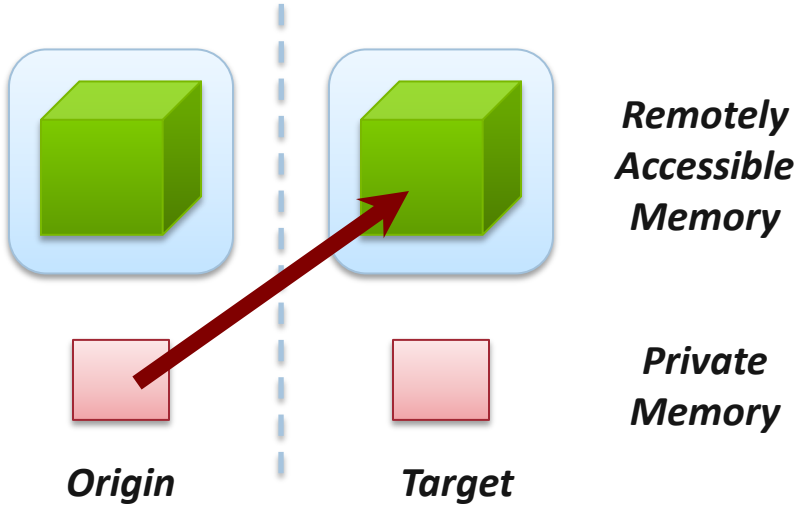
Data movement

- 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)

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)
```

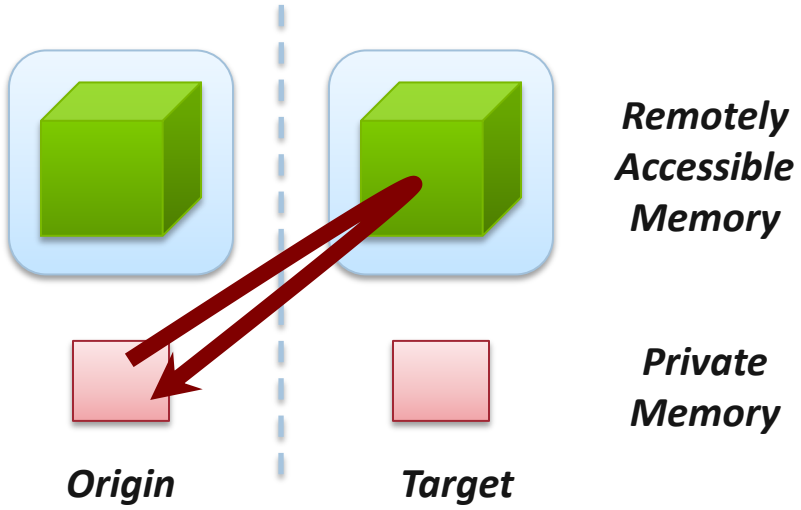
- Move data from origin, to target
- Separate data description triples for **origin** and **target**



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)
```

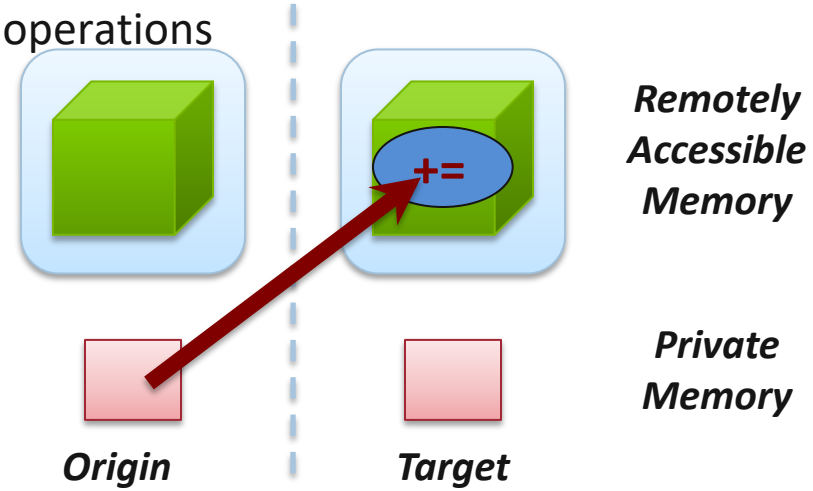
- Move data to origin, from target
- Separate data description triples for **origin** and **target**



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)
```

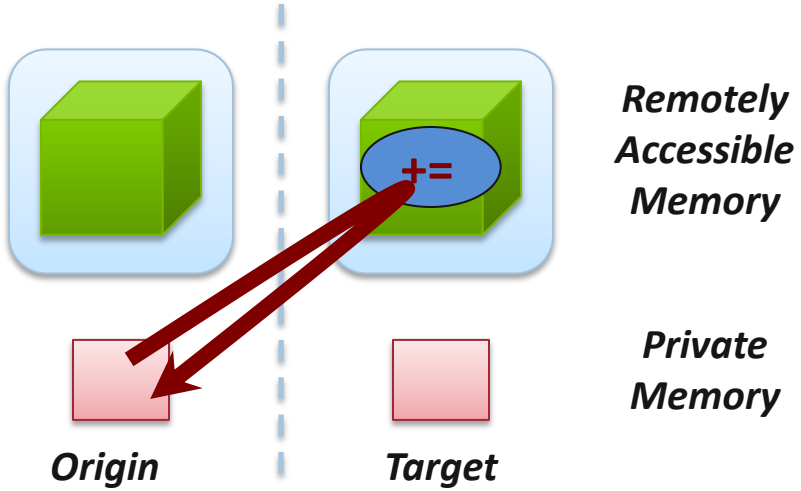
- Atomic update operation, similar to a put
  - Reduces origin and target data into target buffer using op argument as combiner
  - Op = **MPI_SUM, MPI_PROD, MPI_OR, MPI_REPLACE, MPI_NO_OP**, …
  - Predefined ops only, no user-defined operations
- Different data layouts between target/origin OK
  - Basic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

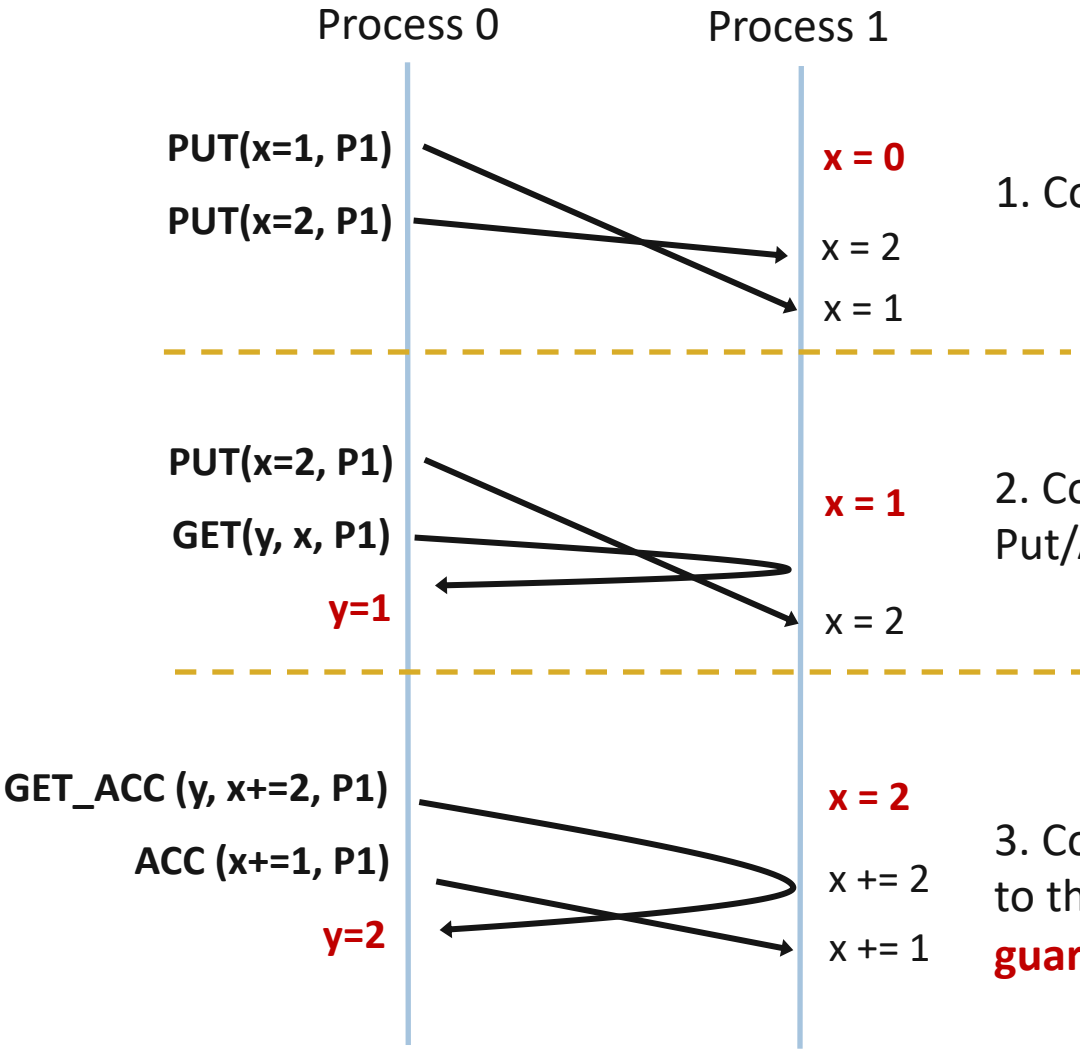
```
MPI_Compare_and_swap(const void *origin_addr,
        const void *compare_addr, void *result_addr,
        MPI_Datatype dtype, int target_rank,
        MPI_Aint target_disp, MPI_Win win)
```

- 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**

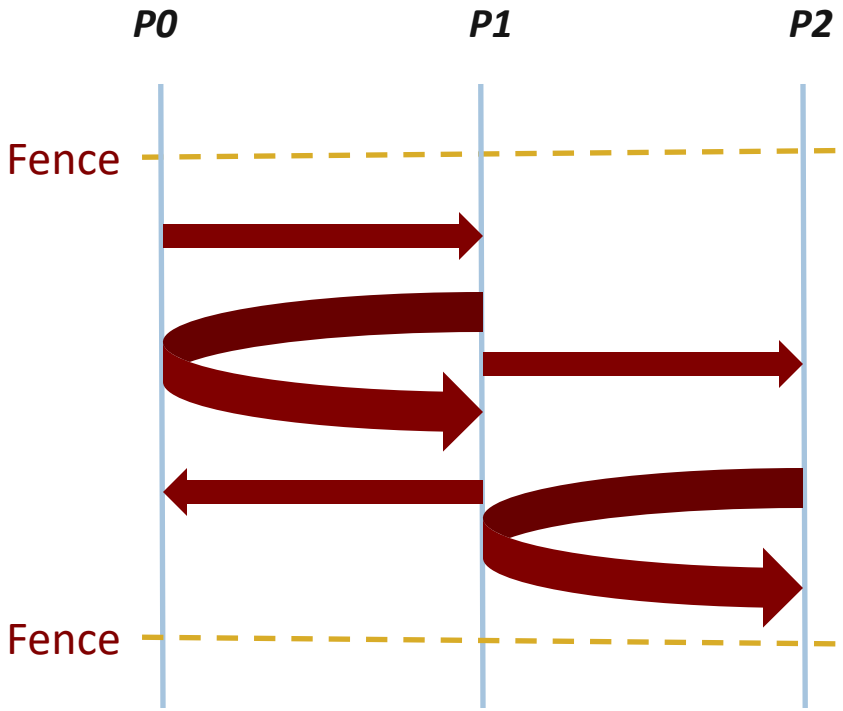
RMA Synchronization Models

- 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)
- 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

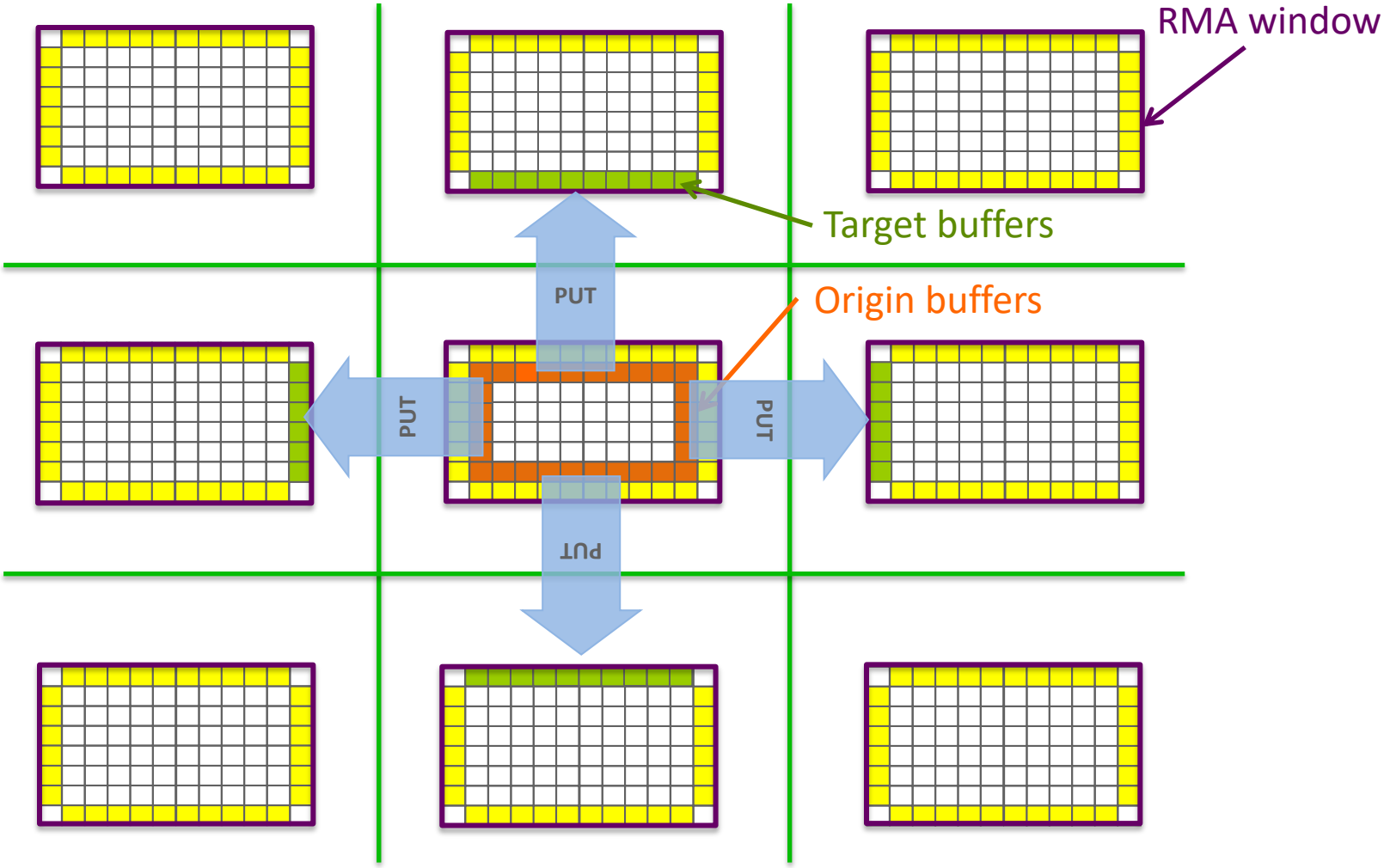
Fence: Active Target Synchronization

```
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*

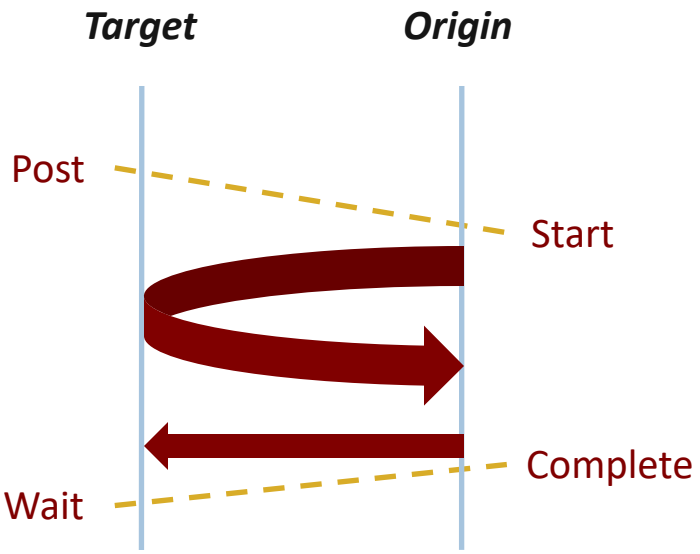
Exercise 2: Stencil with RMA Fence (GET model)

- 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*

PSCW: Generalized Active Target Synchronization

```
MPI_Win_post/start(MPI_Group grp, int assert, MPI_Win win)
MPI_Win_complete/wait(MPI_Win win)
```

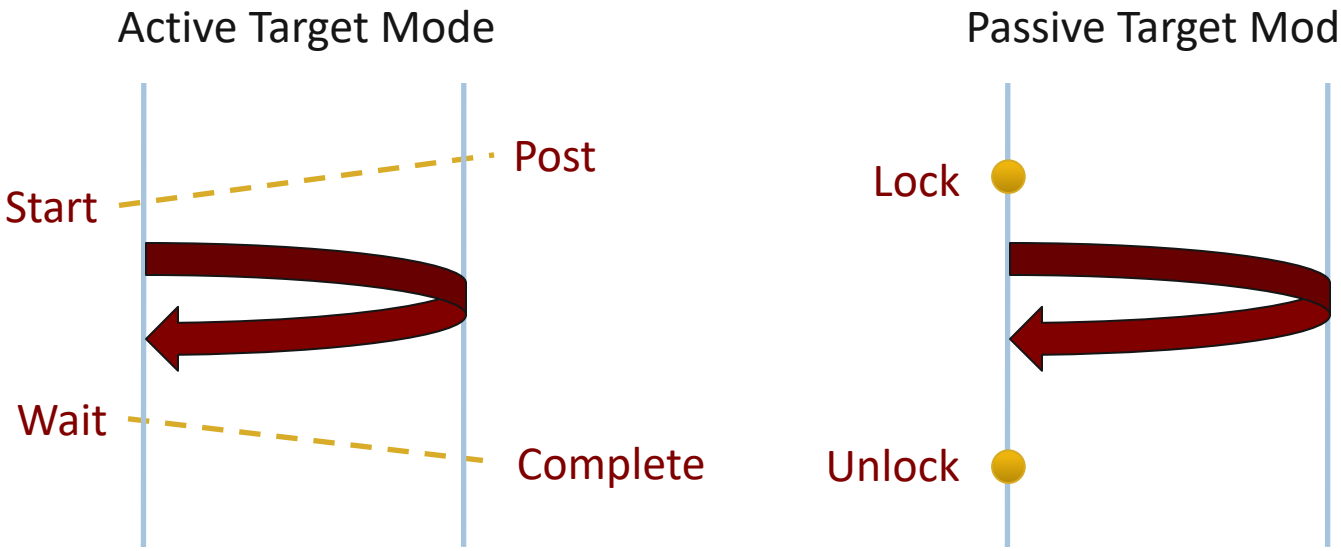
- Like FENCE, but origin and target specify who they communicate with
- Target: Exposure epoch
  - Opened with **`MPI_Win_post`**
  - Closed by **`MPI_Win_wait`**
- Origin: Access epoch
  - Opened by **`MPI_Win_start`**
  - Closed by **`MPI_Win_complete`**
- All synchronization operations may block, to enforce P-S/C-W ordering
  - Processes can be both origins and targets



Exercise 3: Stencil with RMA PSCW (PUT model)

- In the fence version of the stencil code
  - Unnecessary synchronization between all processes
- Let's try to use RMA PSCW
  - Synchronize with PSCW instead of Fence
- *Start from rma/stencil_fence_put.c*
- *Solution available in rma/stencil_pscw_put.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

Advanced Passive Target Synchronization

```
MPI_Win_lock_all(int assert, MPI_Win win)
MPI_Win_unlock_all(MPI_Win win)
MPI_Win_flush_all/flush_local_all(MPI_Win win)
```

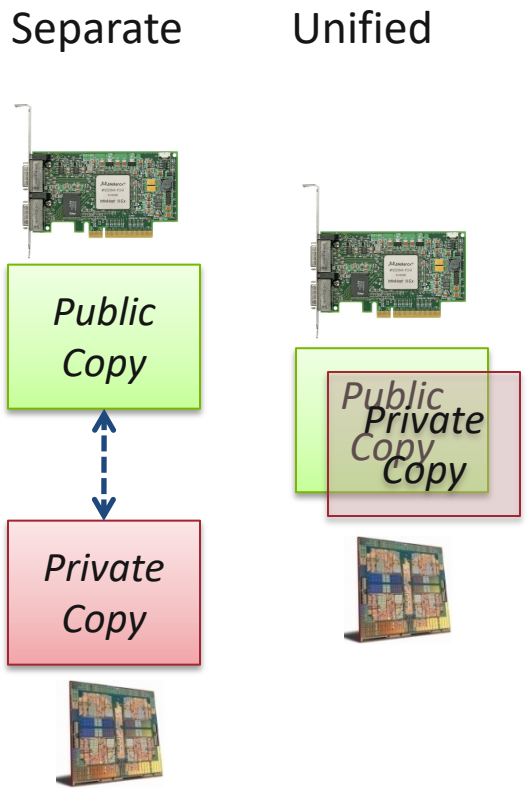
- **`Lock_all`**: Shared lock, passive target epoch to all other processes
  - Expected usage is long-lived: **`lock_all`**, **`put/get`**, **`flush`**, …, **`unlock_all`**
- **`Flush_all`** – remotely complete RMA operations to all processes
- **`Flush_local_all`** – locally complete RMA operations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

Which synchronization mode should I use, when?

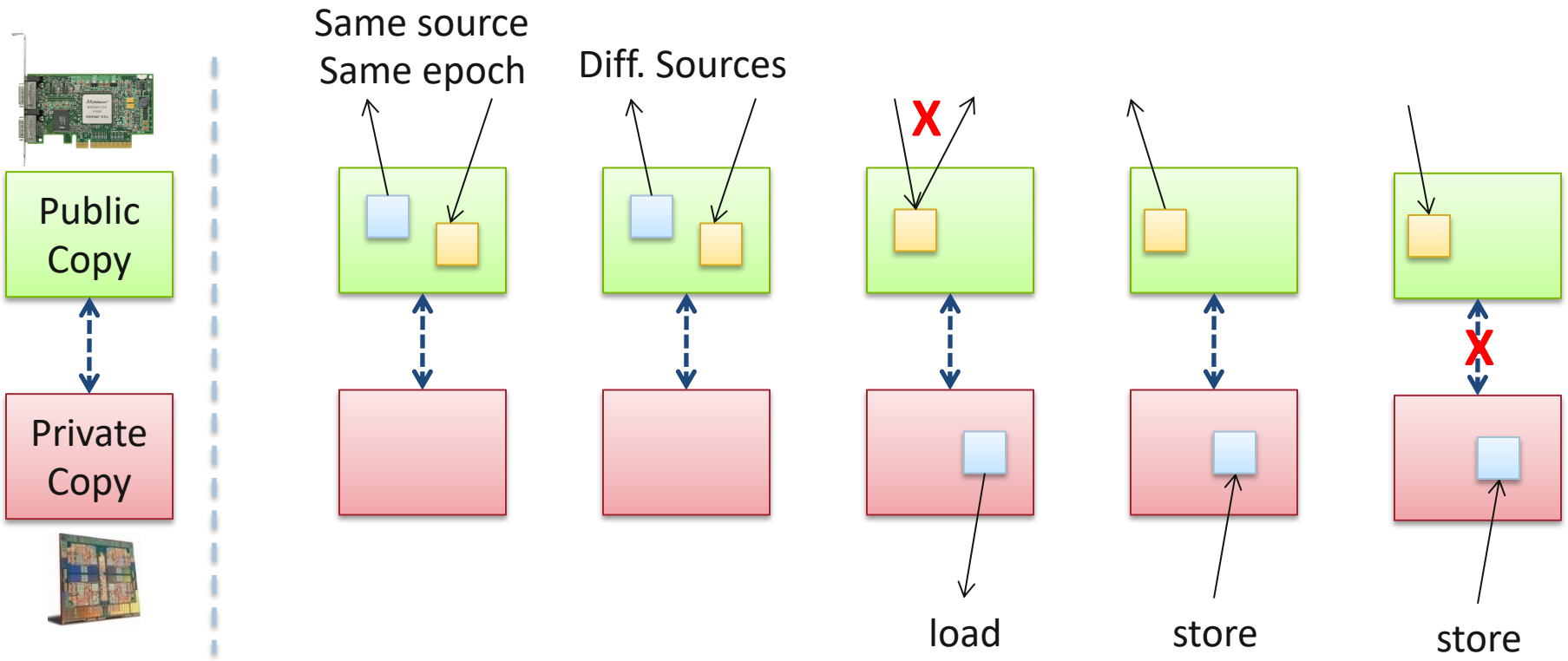
- RMA communication has low overheads versus send/recv
  - Two-sided: Matching, queuing, buffering, unexpected receives, etc…
  - One-sided: No matching, no buffering, always ready to receive
  - Utilize RDMA provided by high-speed interconnects (e.g. InfiniBand)
- Active mode: bulk synchronization
  - E.g. ghost cell exchange
- Passive mode: asynchronous data movement
  - Useful when dataset is large, requiring memory of multiple nodes
  - Also, when data access and synchronization pattern is dynamic
  - Common use case: distributed, shared arrays
- Passive target locking mode
  - Lock/unlock – Useful when exclusive epochs are needed
  - Lock_all/unlock_all – Useful when only shared epochs are needed

MPI RMA Memory Model

- MPI-3 provides two memory models: separate and unified
- 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

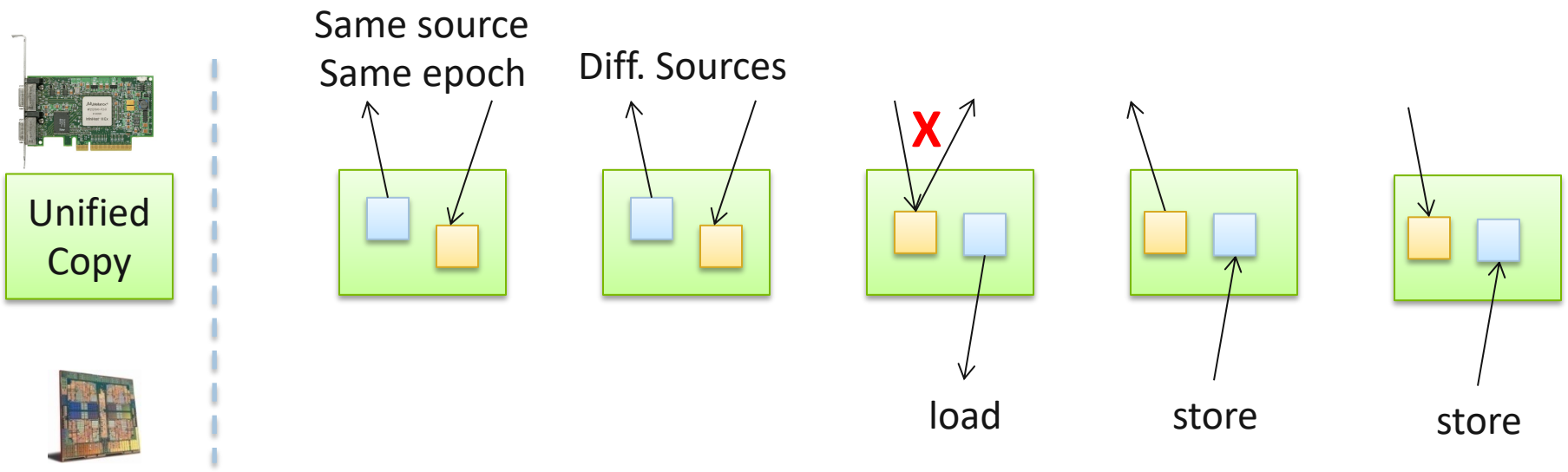


MPI RMA Memory Model (separate windows)



- 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

MPI RMA Operation Compatibility (Separate)

	Load	Store	Get	Put	Acc
Load	OVL+NOVL	OVL+NOVL	OVL+NOVL	NOVL	NOVL
Store	OVL+NOVL	OVL+NOVL	NOVL	X	X
Get	OVL+NOVL	NOVL	OVL+NOVL	NOVL	NOVL
Put	NOVL	X	NOVL	NOVL	NOVL
Acc	NOVL	X	NOVL	NOVL	OVL+NOVL

This matrix shows the compatibility of MPI-RMA operations when two or more processes access a window at the same target concurrently.

OVL – Overlapping operations permitted

NOVL – Nonoverlapping operations permitted

X – Combining these operations is OK, but data might be garbage

MPI RMA Operation Compatibility (Unified)

	Load	Store	Get	Put	Acc
Load	OVL+NOVL	OVL+NOVL	OVL+NOVL	NOVL	NOVL
Store	OVL+NOVL	OVL+NOVL	NOVL	NOVL	NOVL
Get	OVL+NOVL	NOVL	OVL+NOVL	NOVL	NOVL
Put	NOVL	NOVL	NOVL	NOVL	NOVL
Acc	NOVL	NOVL	NOVL	NOVL	OVL+NOVL

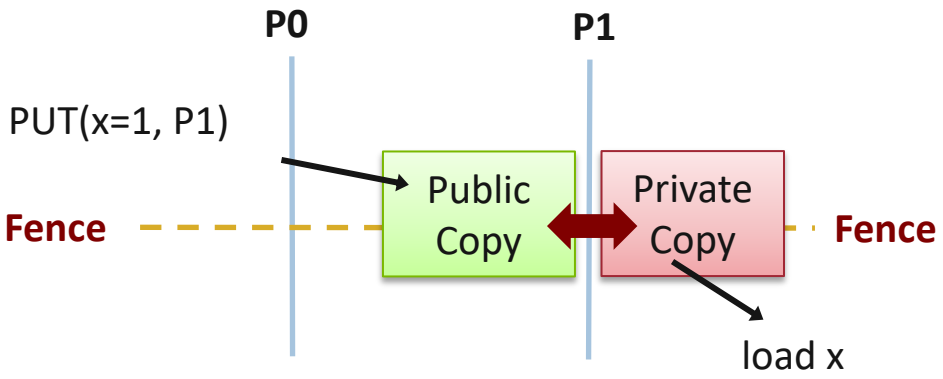
This matrix shows the compatibility of MPI-RMA operations when two or more processes access a window at the same target concurrently.

OVL – Overlapping operations permitted

NOVL – Nonoverlapping operations permitted

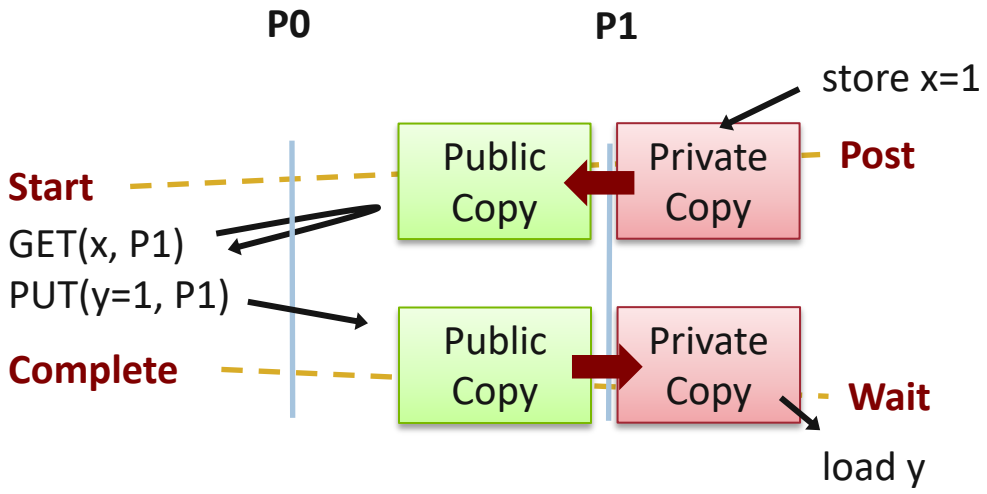
Synchronizing Local and RMA Access (1/2)

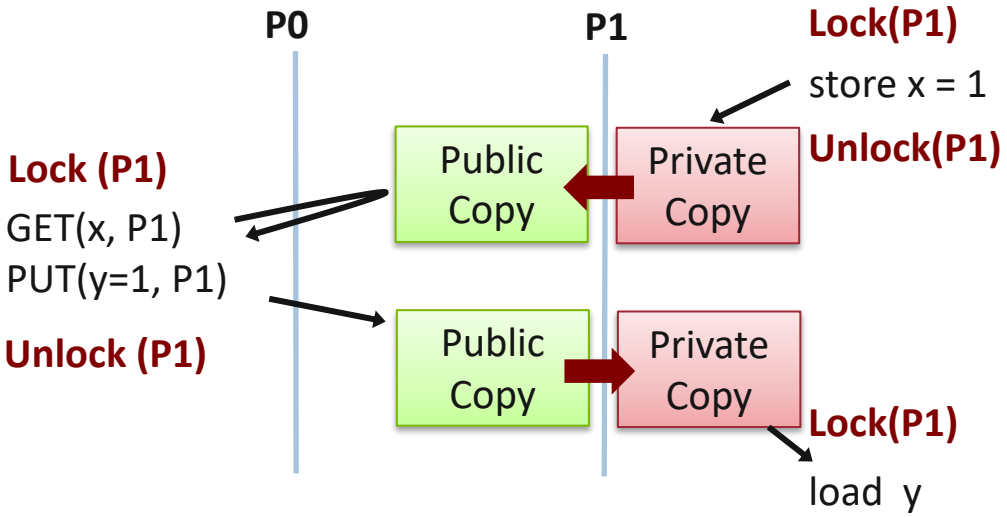
- RMA operations access the public copy of window
- Local load/store update the private copy
  - Including using as MPI send/receive buffers
- Ensure memory synchronization for portability
- Implicit memory synchronization (i.e., memory barrier) in RMA synchronization calls
  - **Fence:** Synchronize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

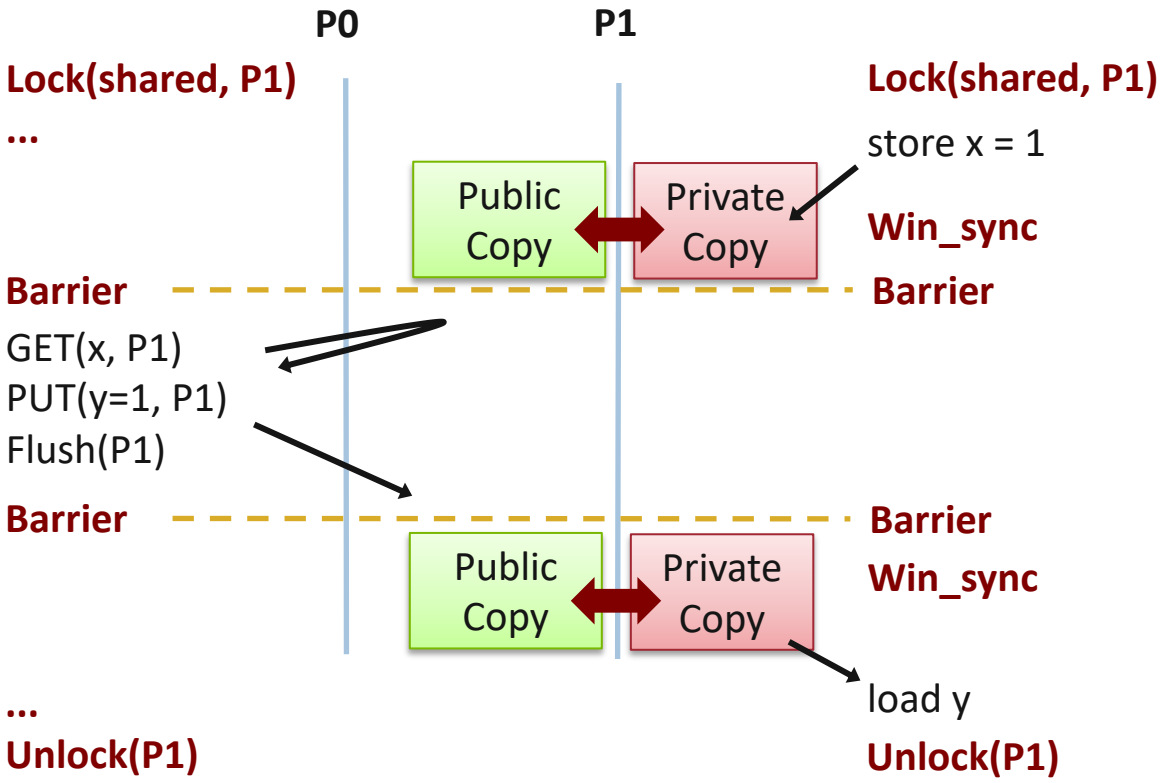




Window synchronization: MPI_WIN_SYNC

```
MPI_Win_sync(MPI_Win win)
```

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

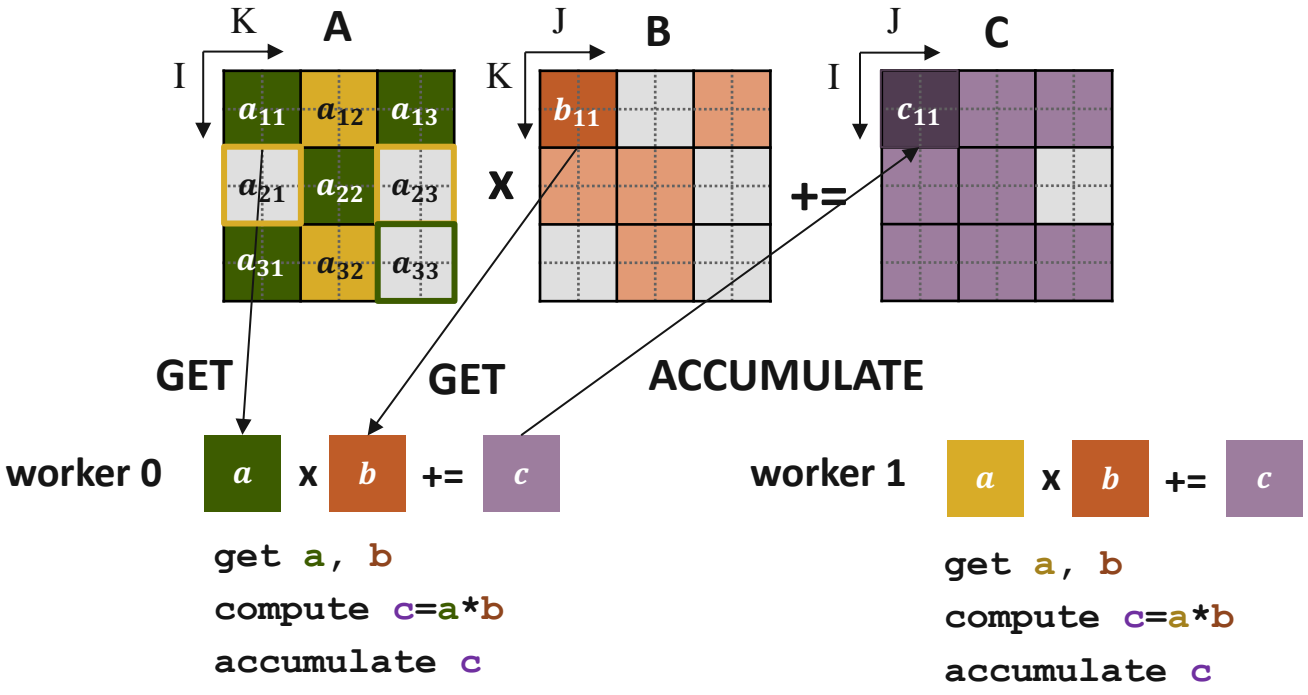


Exercise 4: Stencil with RMA Lock_all/Unlock_all (PUT model)

- In the fence and PSCW 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*

Example: BSPMM with RMA

- *rma/bspmm.c*
- Only synchronization from origin processes (workers), no synchronization from target processes (implicit master)
- Replace send/receive with RMA Get and Accumulate



Exercise 5: BSPMM using global counter (atomics)

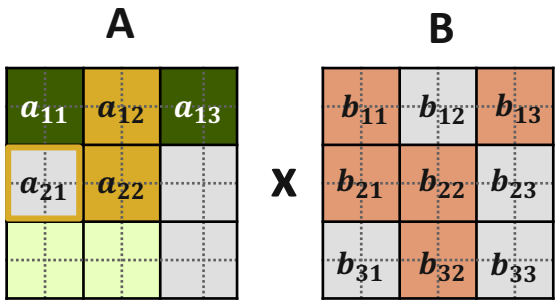
- In the basic RMA version of BSPMM, we statically assigned the ownership of blocks for each process
  - Load imbalance: e.g., one process may get more zero-blocks than others
- Let us implement a dynamic task secluding
  - Each process dynamically "queries" the next available block computation once finished the previous one (A-based parallelism)
  - Use atomic **Fetch_and_op(SUM)** with a **"global counter"**
- *Start from rma/bspmm.c*
- *Solution available in rma/bspmm_counter.c*

rank 0

```
FOP(counter)-> 0
compute a11*b1j



FOP(counter)-> 2
compute a13*b3j
...
```

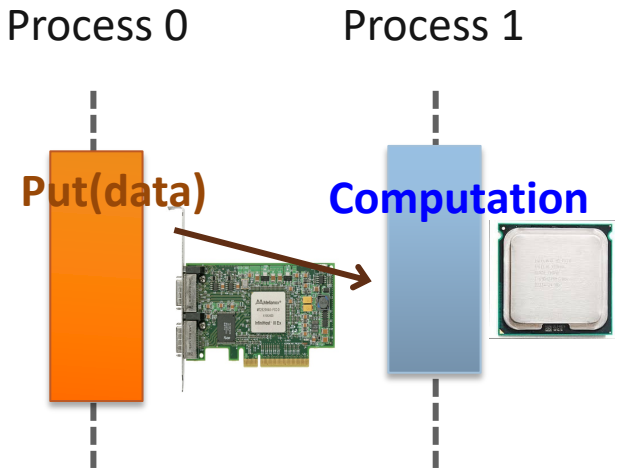
rank 1

```
FOP(counter)-> 1
compute a12*b2j
FOP(counter)-> 3
Skip (a21)
FOP(counter)-> 4
compute a22*b2j
...
```



Hardware-Offloaded Communication

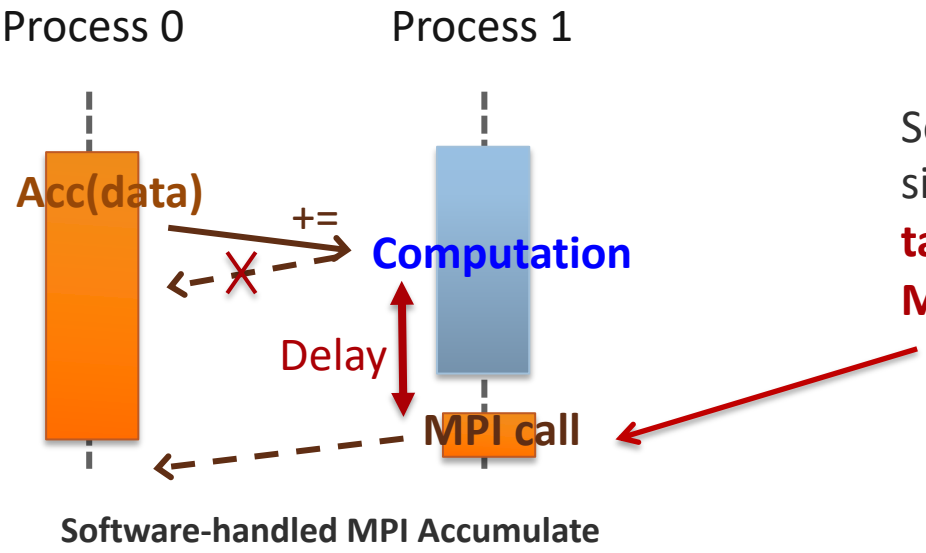
- Offloading data transfer to hardware is ideal for performance
  - Two-sided (e.g., SEND/RECEIVE):
    - Require **complex message matching** (rank + tag + comm), especially for wildcard receives (MPI_ANY_TAG|ANY_SOURCE)
    - Supported HW: Mellanox ConnectX-5 (support HW tag matching)
  - One-sided (e.g., PUT/GET/ACCUMUALTE):
    - **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**

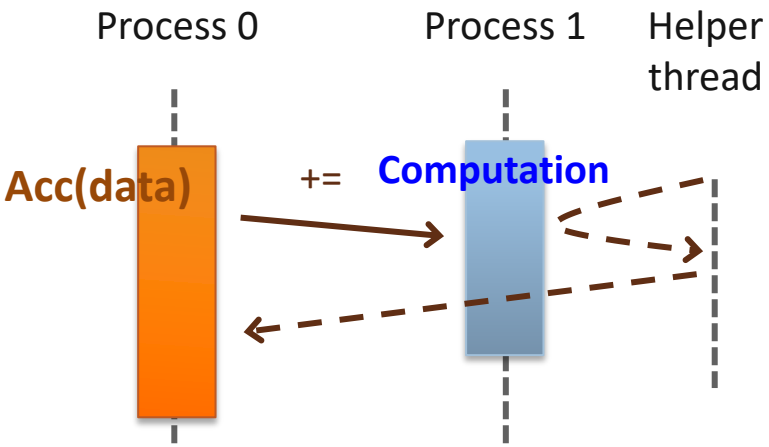


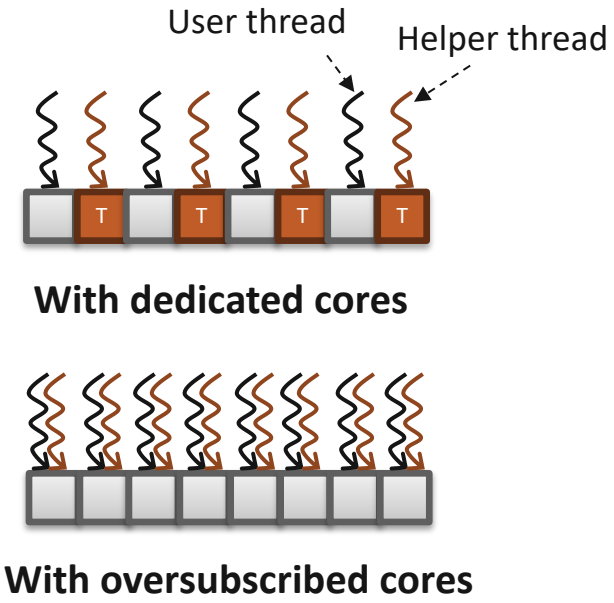
Software-handled MPI Accumulate

Software implementation of one-sided operations means that **the target process has to make an MPI call to make progress**

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

- Every MPI process creates a **dedicated helper thread** at MPI_Init
  - 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
- Dedicates some number of computing cores per process
- Multithreading safety overhead (i.e., MPI internal lock contention between threads, memory barriers)





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:

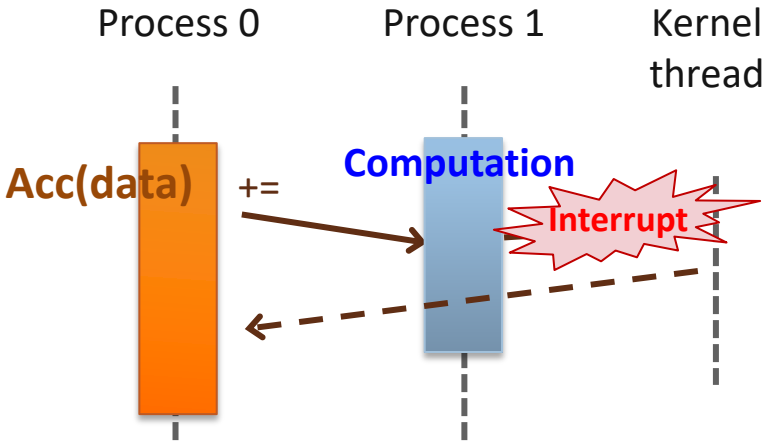
```
% export I_MPI_ASYNC_PROGRESS=1
```

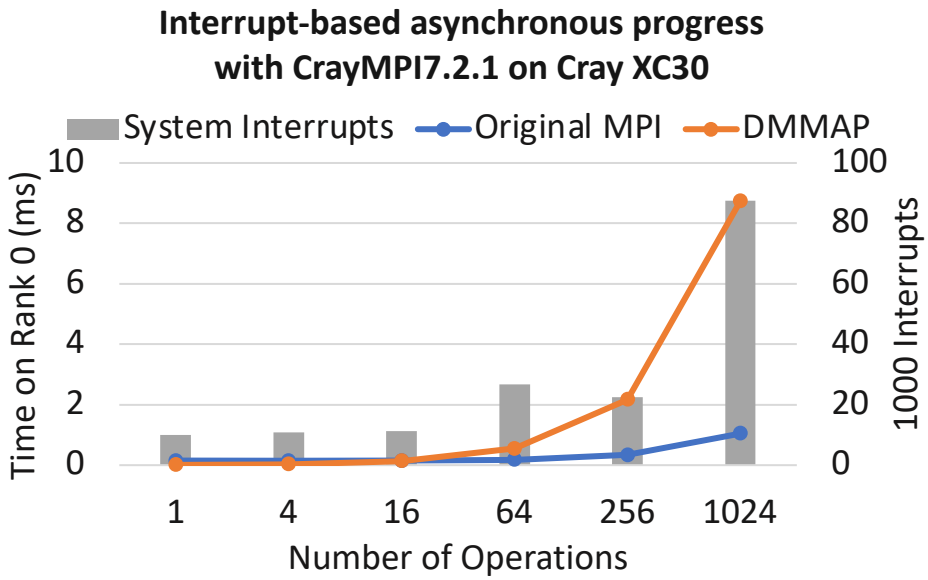
Cray MPI:

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

Possible Solution 2: Interrupt-based Progress

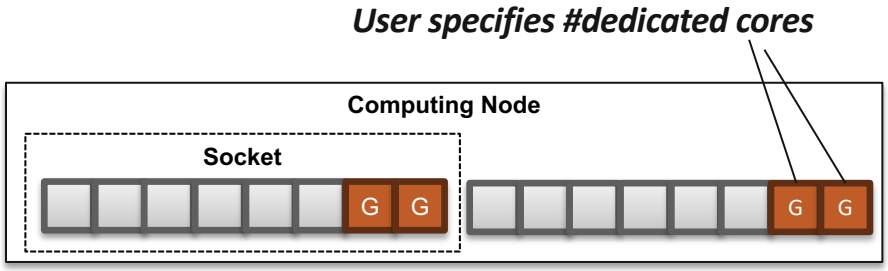
- Utilize **hardware interrupts** to awaken a kernel thread when new message arrives
- Examples: Cray MPI DMAPP mode (deprecated from v7.6.0), IBM MPI on Blue Gene/P
- Overhead of frequent interrupts, need special hardware support
  - Not a common model for most current networks

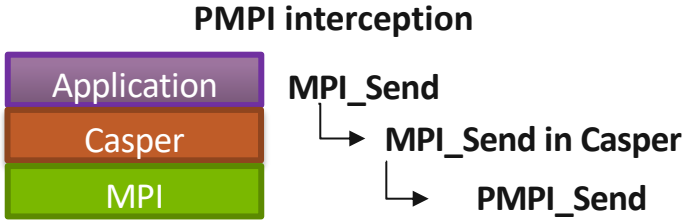


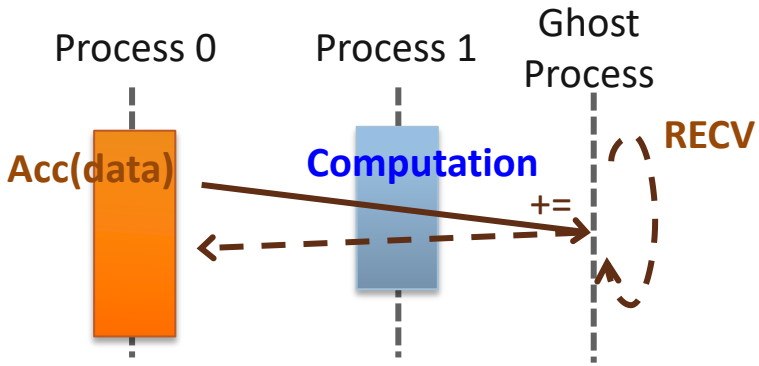


Possible Solution 3: Process-based Progress (1/4)

- Casper
  - Dedicating **arbitrary number of cores** (usually 1-4 per multicore node) to **"ghost processes"**
  - **Portable PMPI profiling interface based** design allows integration with any MPI-3 implementations *
  - Transparently replace MPI_COMM_WORLD by **COMM_USER_WORLD**
  - **Shared memory mapping** between local user and ghost processes
  - Redirect RMA operations to ghost process, thus ghost process ensures communication progress
  - Casper can redirect operations only for **MPI_WIN_ALLOCATE** windows







* MPI-3 MPI_Win_allocate_shared function is used to enable shared memory mapping.

Possible Solution 3: Process-based Progress (2/4)

- Download Casper at
  http://www.mcs.anl.gov/project/casper/downloads
- How to use Casper?
  1. Load Casper between application and MPI library:
  - **Option 1**: dynamic load Casper at execution time

```
% mpicc -o app app.c
% export LD_PRELOAD=<casper_install_dir>/lib/libcasper.so
```

  - **Option 2**: link with Casper (ensure link order: app.o -lcasper -lmpi)

```
% mpicc -o app app.c –lcasper
```

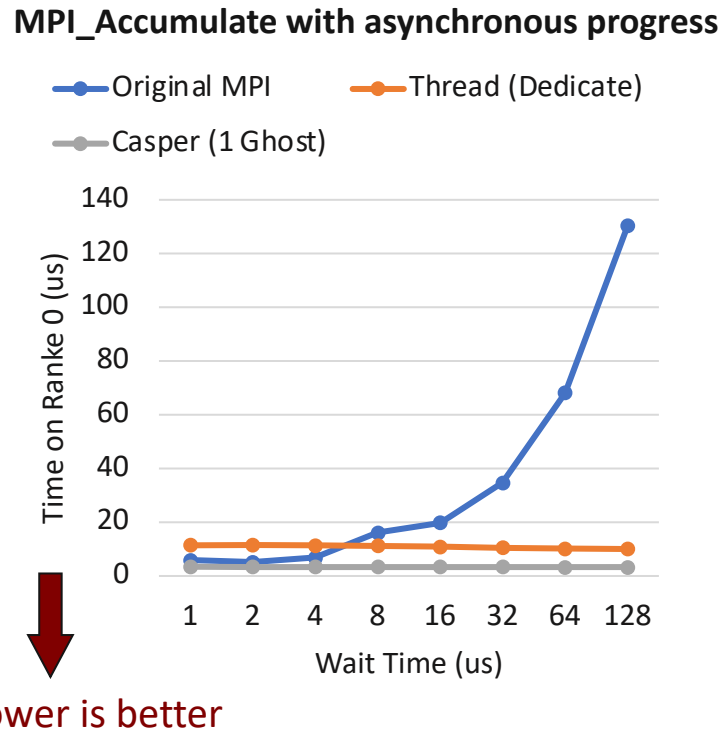
  2. Set number of ghost processes (dedicated progress core)

```
% export CSP_NG=2
% mpiexec -np 48 -ppn 24 ./app
/* 48 processes are running on two nodes,
 * each node has 22 user processes and 2 ghost processes*/
```

Possible Solution 3: Process-based Progress (3/4)

- Demonstrating communication progress

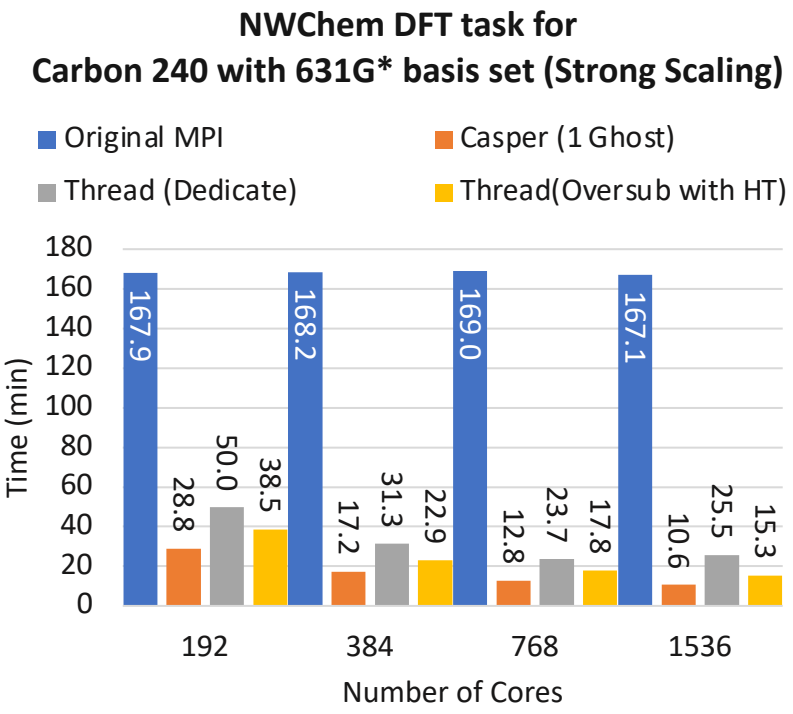
```
MPI_Win_allocate(sizeof(int), sizeof(int),
    MPI_INFO_NULL, MPI_COMM_WORLD, &a, &win);

if (rank == 0) {
    MPI_Win_lock(MPI_LOCK_SHARED, 1, 0, win);
    for (int x = 0; x < ITER; x++) {
        MPI_Accumulate(…, target = 1, …);
        MPI_Win_flush(1, win);
    }
    MPI_Win_unlock(1, win);
    MPI_Send(…, dst = 1, …);
} else {
    MPI_Irecv(…, src = 0, …, &req);
    for (int x = 0; x < ITER; x++) {
        busy_wait(time);
        MPI_Test(&req, &flag, &stat);
    }
}
MPI_Win_free(&win);
```

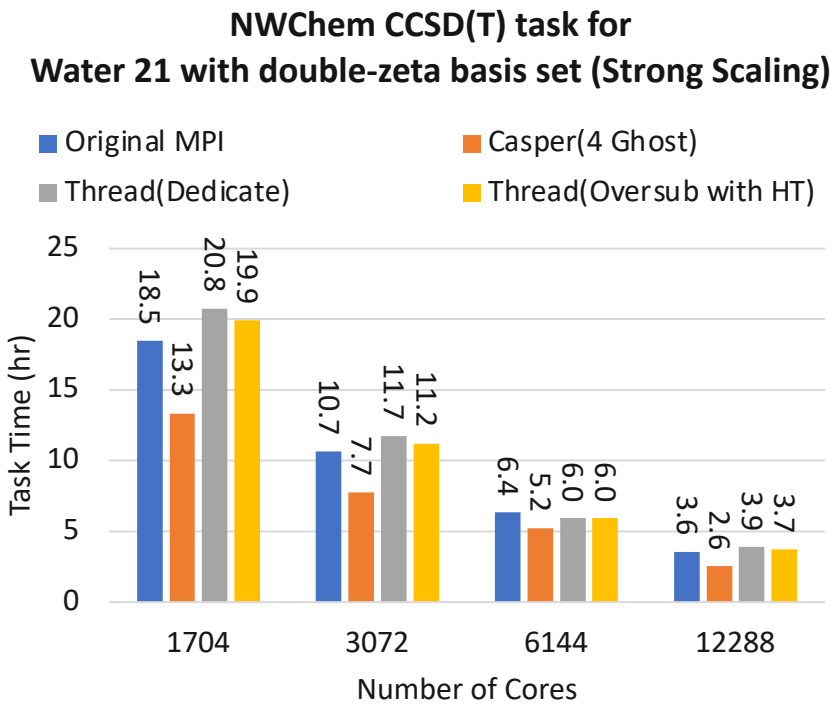


* *Measured on NERSC Edison Cray XC30 with CrayMPI 6.3.1.*

Possible Solution 3: Process-based Progress (4/4)

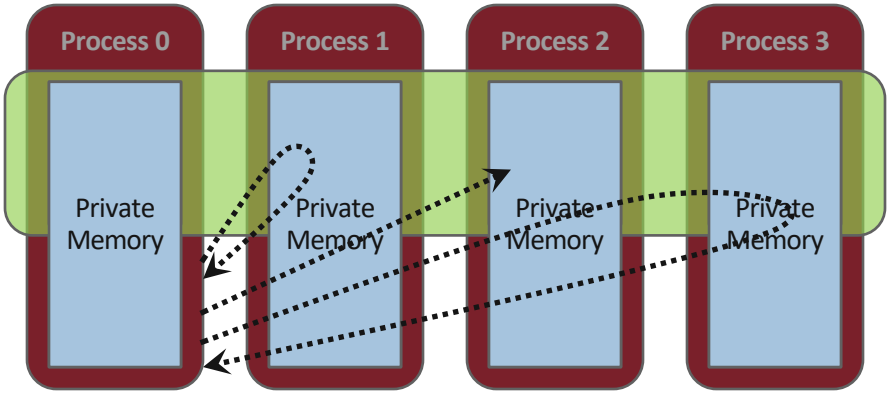
- Performance improvement of NWChem with RMA asynchronous progress





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



Argonne NATIONAL LABORATORY

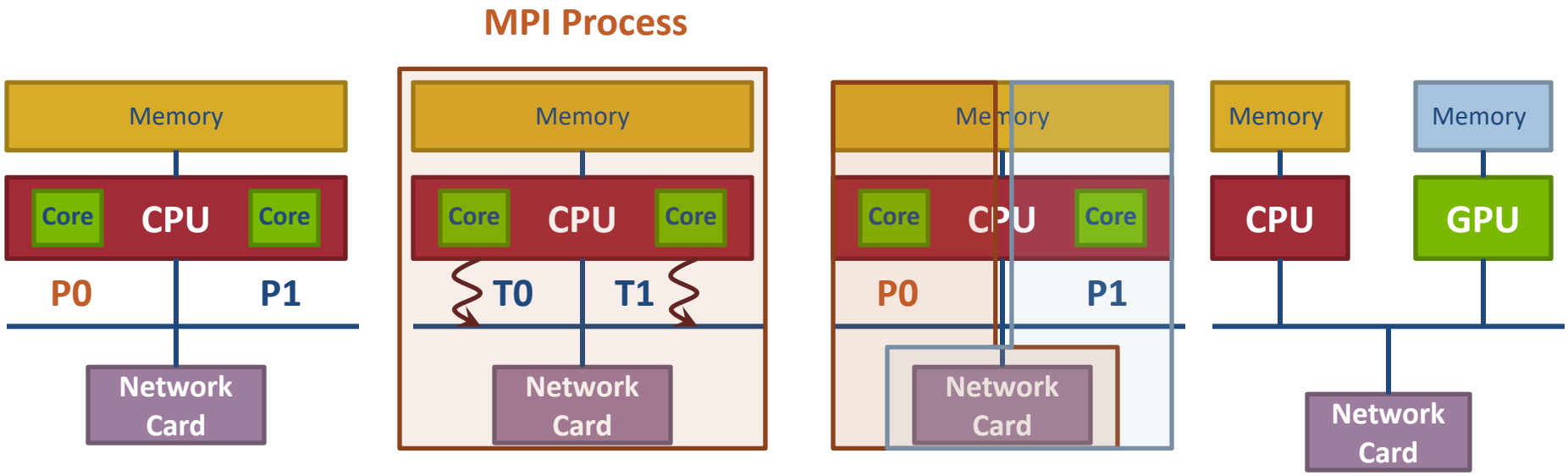
MPI Hybrid Programming: Threads

Slides Available at https://anl.box.com/v/yguo-isc-tutorial-2021

U.S. DEPARTMENT OF ENERGY

Hybrid MPI + X : Most Popular Forms

MPI + X



Flat MPI

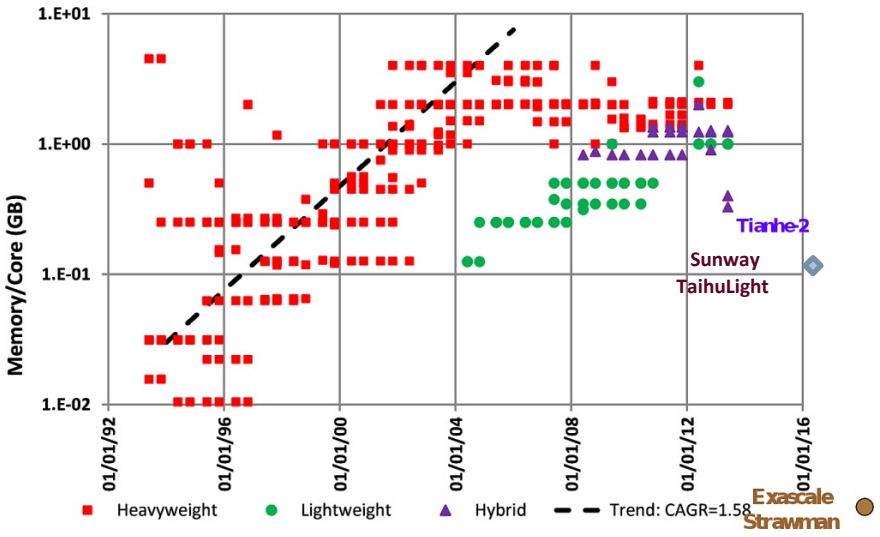
MPI + Threads

MPI + Shared Memory

MPI + ACC

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

- Strong scaling applications is increasing in importance
  - Hardware limitations: not all resources scale at the same rate as cores (e.g., memory capacity, network resources)
  - Desire to solve the same problem faster on a bigger machine
    - Nek5000, HACC, LAMMPS



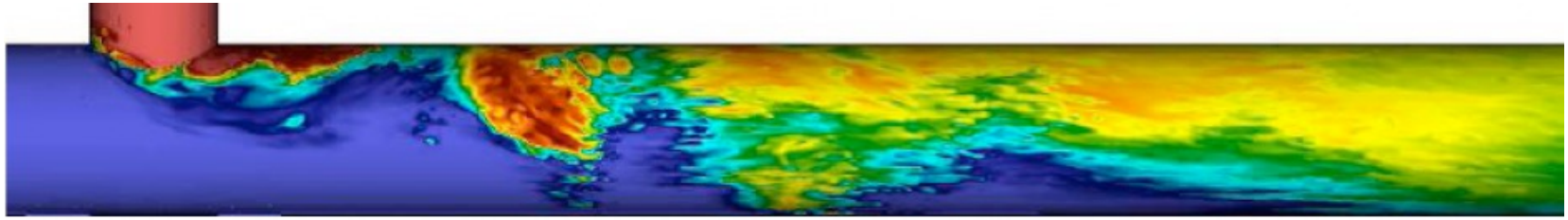
Evolution of the memory capacity per core in the Top500 list (Peter Kogge. Pim & memory: The need for a revolution in architecture.)

- 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

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

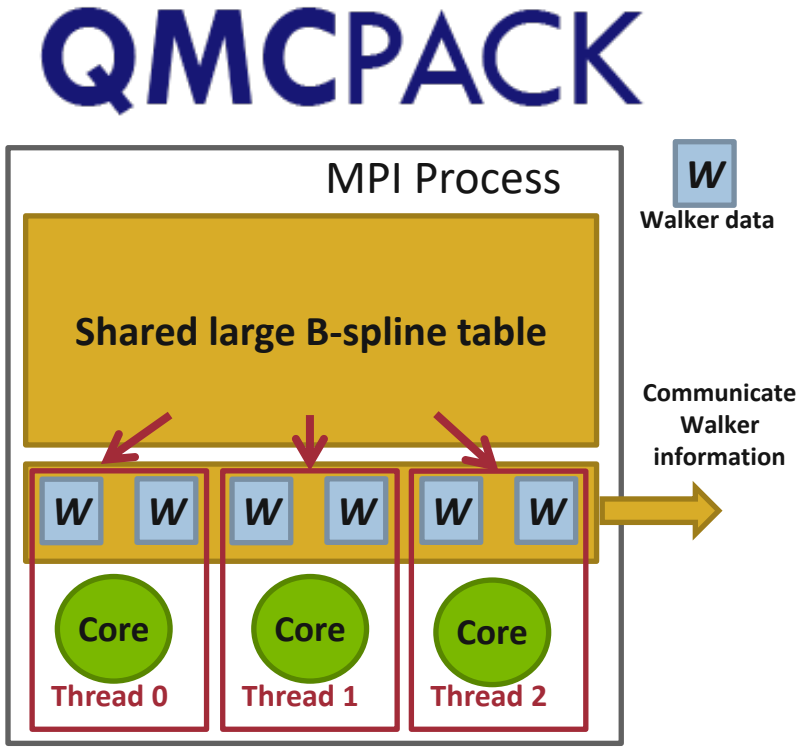
- MPI+X benefits (X= {threads,MPI shared-memory, etc.})
  - Less memory hungry (MPI runtime consumption, O(P) data structures, etc.)
  - Load/stores to access memory instead of message passing
  - P is reduced by constant C (#cores/process) for O(Px) communication patterns
- Example 1: the Nek5000 team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

Nek5000



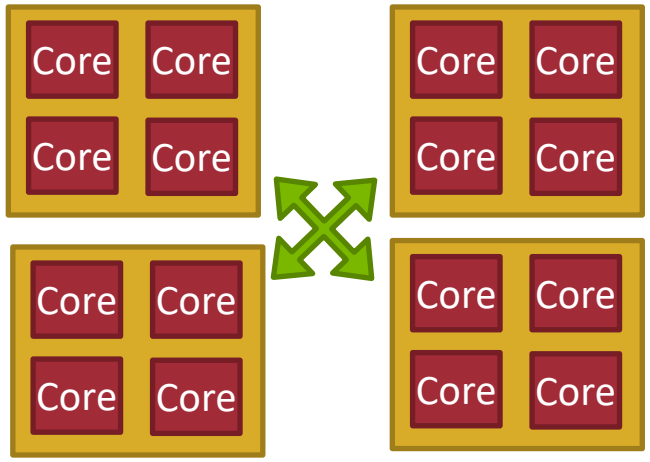
Why Hybrid MPI+X? Towards Strong Scaling (3/3)

- 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.

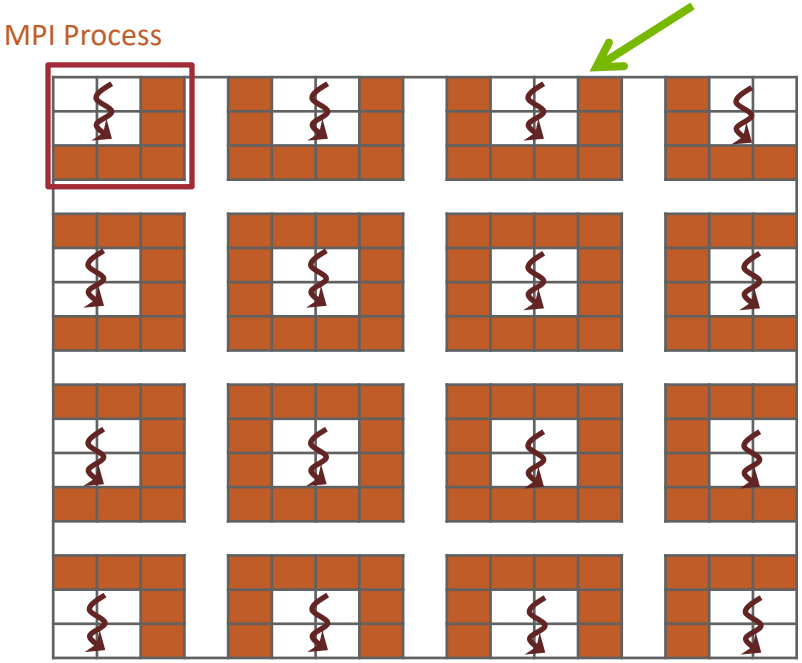


[1] Kim, Jeongnim, et al. "Hybrid algorithms in quantum Monte Carlo." Journal of Physics, 2012.

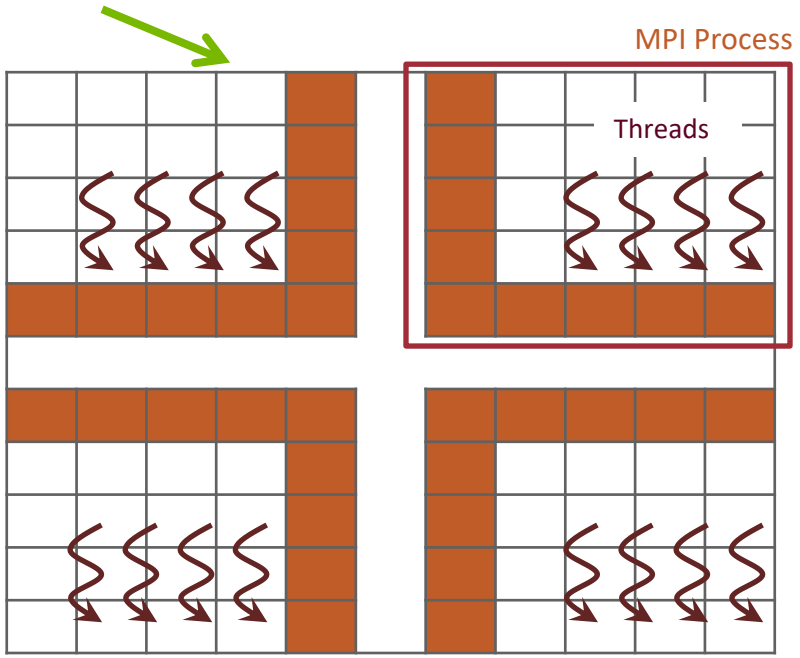
MPI + Threads: How To? (1/3)



Multi- or Many-core Nodes



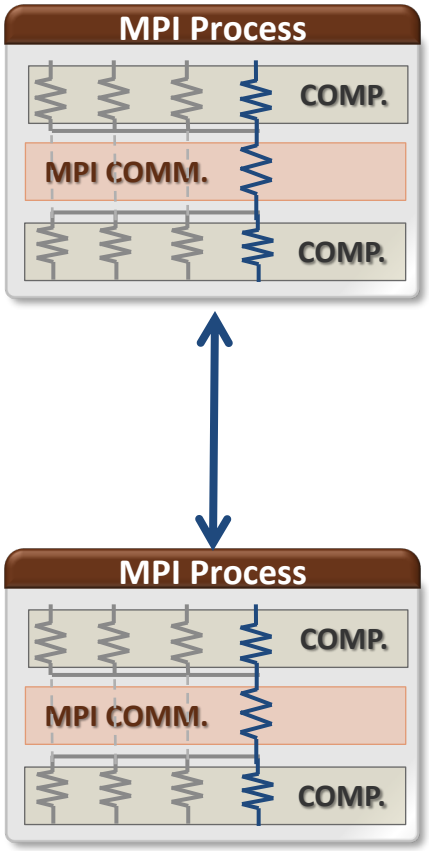
MPI only



MPI + Threads

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

- MPI describes parallelism between *processes* (with separate address spaces)
- *Thread* parallelism provides a shared-memory model within a process
- OpenMP and Pthreads are common models
  - OpenMP provides convenient features for loop-level parallelism. Threads are created and managed by the compiler, based on user directives.
  - Pthreads provide more complex and dynamic approaches. Threads are created and managed explicitly by the user.



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

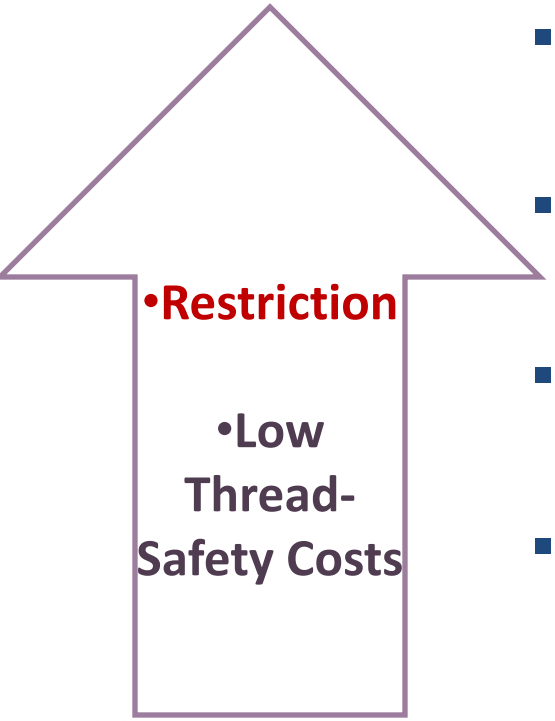
MPI + Threads

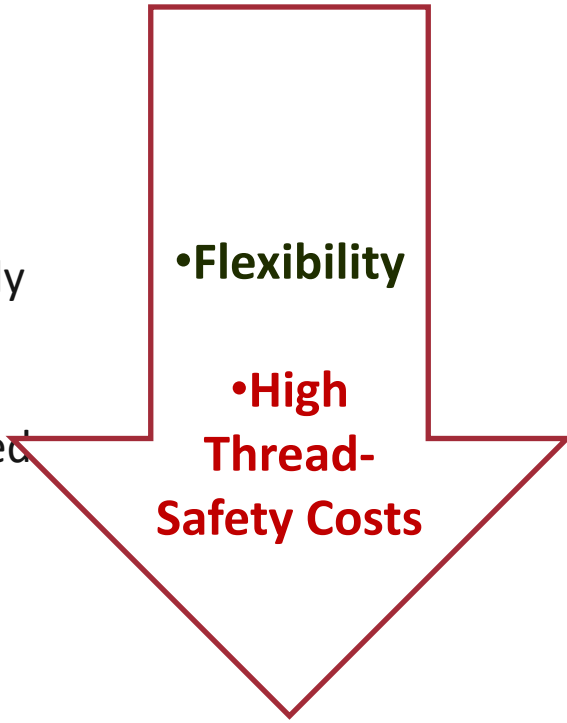
↓

Interoperability

Interoperation or thread levels:

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





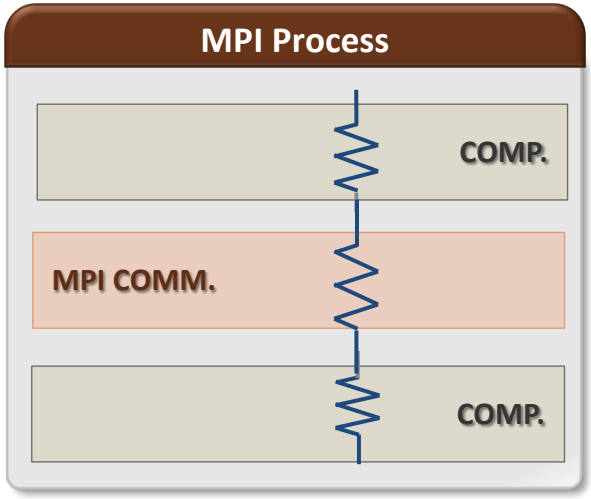
MPI's Four Levels of Thread Safety

- MPI defines four levels of thread safety -- these are commitments the application makes to the MPI
- Thread levels are in increasing order
  - If an application works in FUNNELED mode, it can work in SERIALIZED
- MPI defines an alternative to MPI_Init
  - **MPI_Init_thread**(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;
}
```

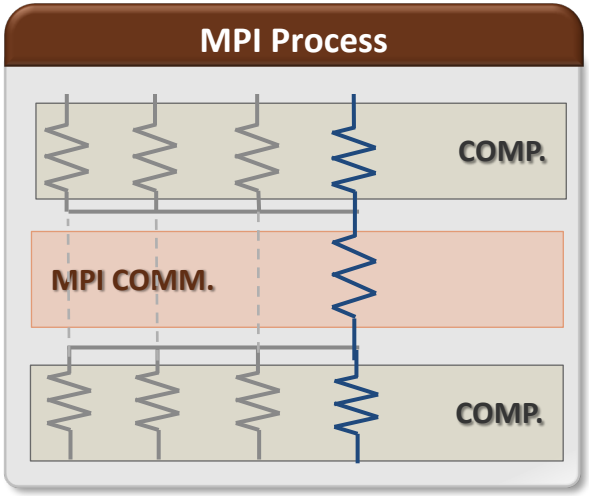


MPI_THREAD_FUNNELED

- 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]);
}
```

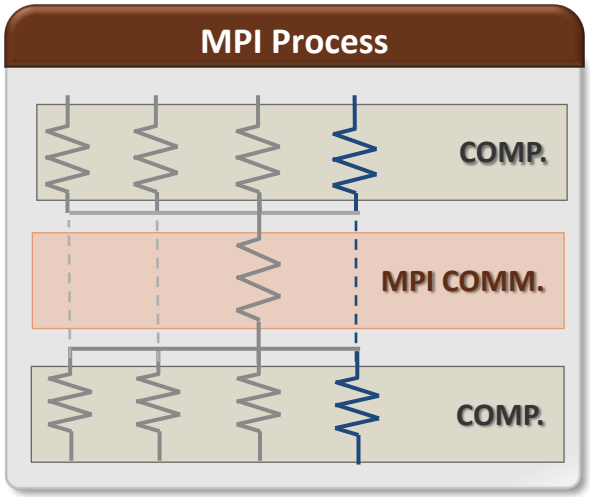


MPI_THREAD_SERIALIZED

- 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);
}
```

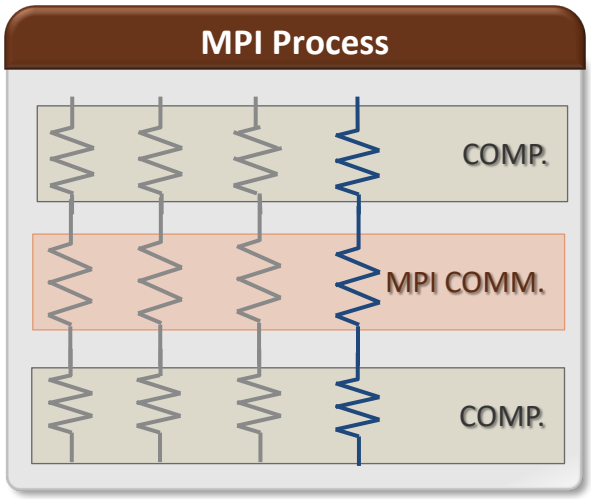


MPI_THREAD_MULTIPLE

- **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 */
}
```



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)*

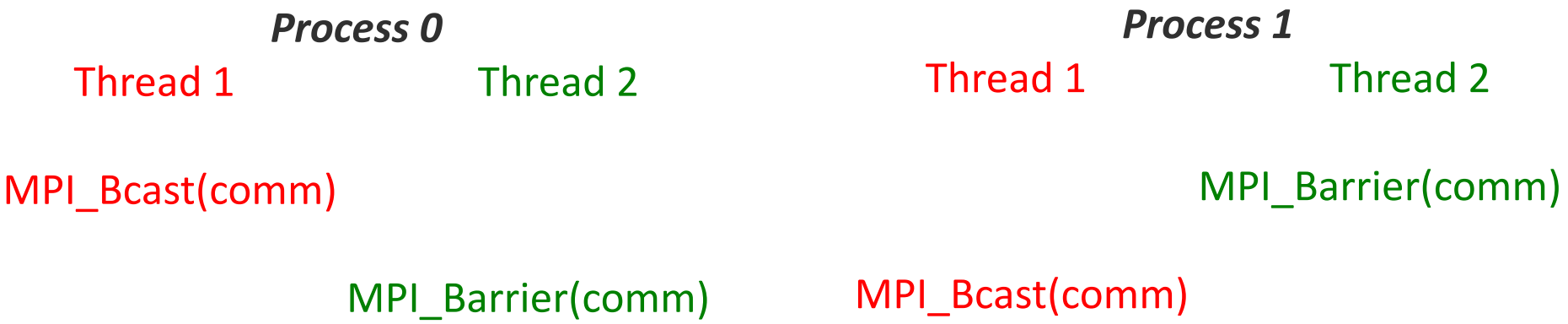
MPI Semantics and MPI_THREAD_MULTIPLE

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

Ordering in MPI_THREAD_MULTIPLE: Incorrect Example with Collectives

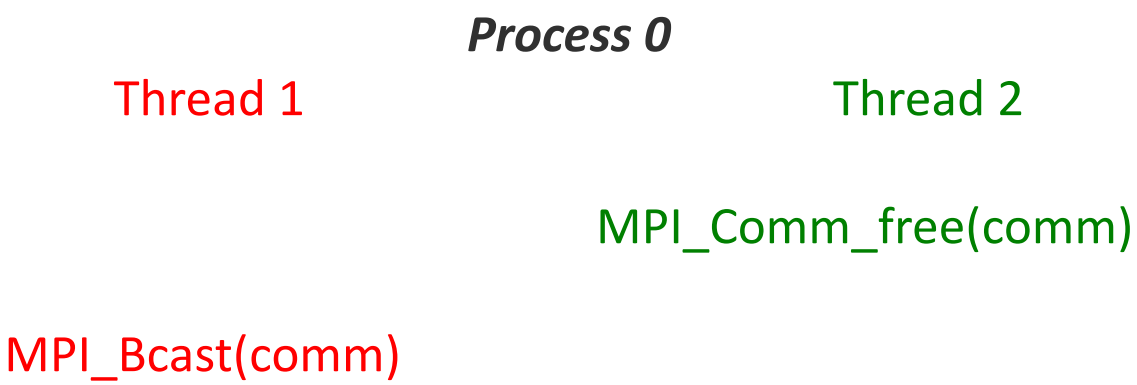
	Process 0	*Process 1*
Thread 0	MPI_Bcast(comm)	MPI_Bcast(comm)
Thread 1	MPI_Barrier(comm)	MPI_Barrier(comm)

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

Ordering in MPI_THREAD_MULTIPLE: Incorrect Example with Object Management



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

Blocking Calls in MPI_THREAD_MULTIPLE: Correct Example

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

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

The Current Situation

- All MPI implementations support MPI_THREAD_SINGLE
- They probably support MPI_THREAD_FUNNELED even if they don't admit it.
  - Does require thread-safety for some system routines (e.g. malloc)
  - On most systems `-pthread` will guarantee it (OpenMP implies `-pthread`)
- Many (but not all) implementations support THREAD_MULTIPLE
  - Hard to implement efficiently though (thread synchronization issues)
- Bulk-synchronous OpenMP programs (loops parallelized with OpenMP, communication between loops) only need FUNNELED
  - So don't need "thread-safe" MPI for many hybrid programs
  - But watch out for Amdahl's Law!

Hybrid Programming: Correctness Requirements

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

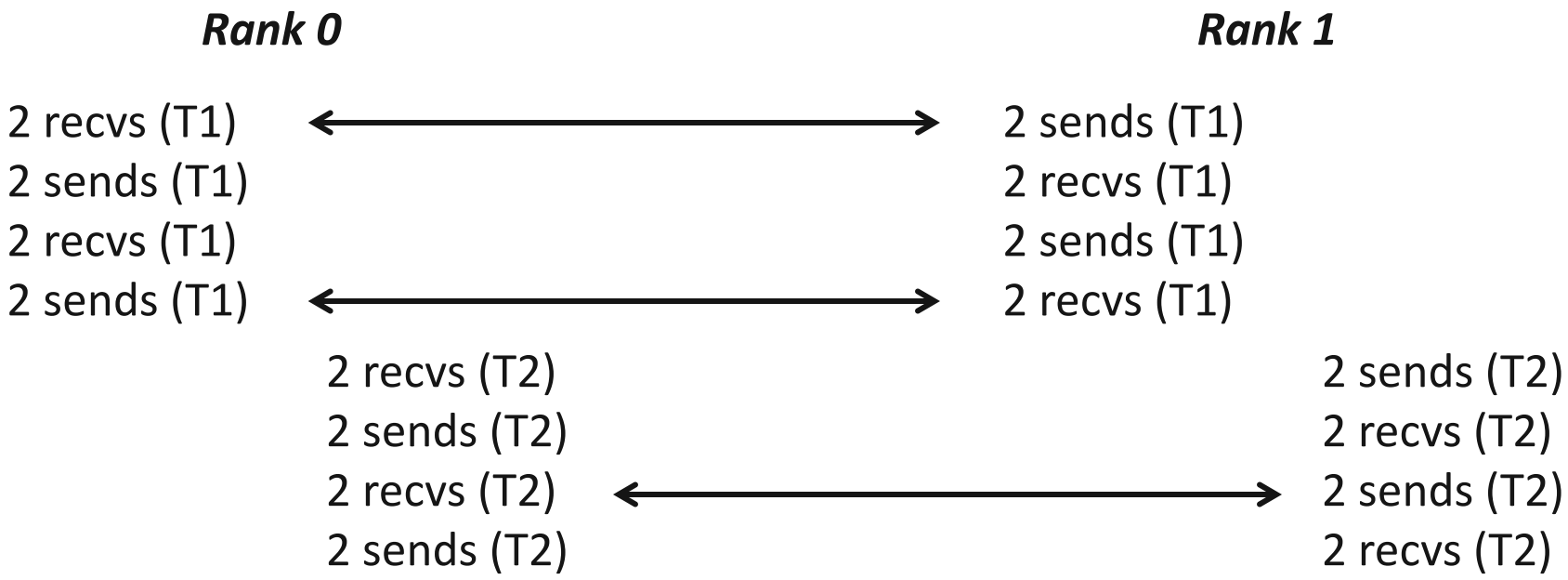
An Example we encountered

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

2 Processes, 2 Threads, Each Thread Executes this Code

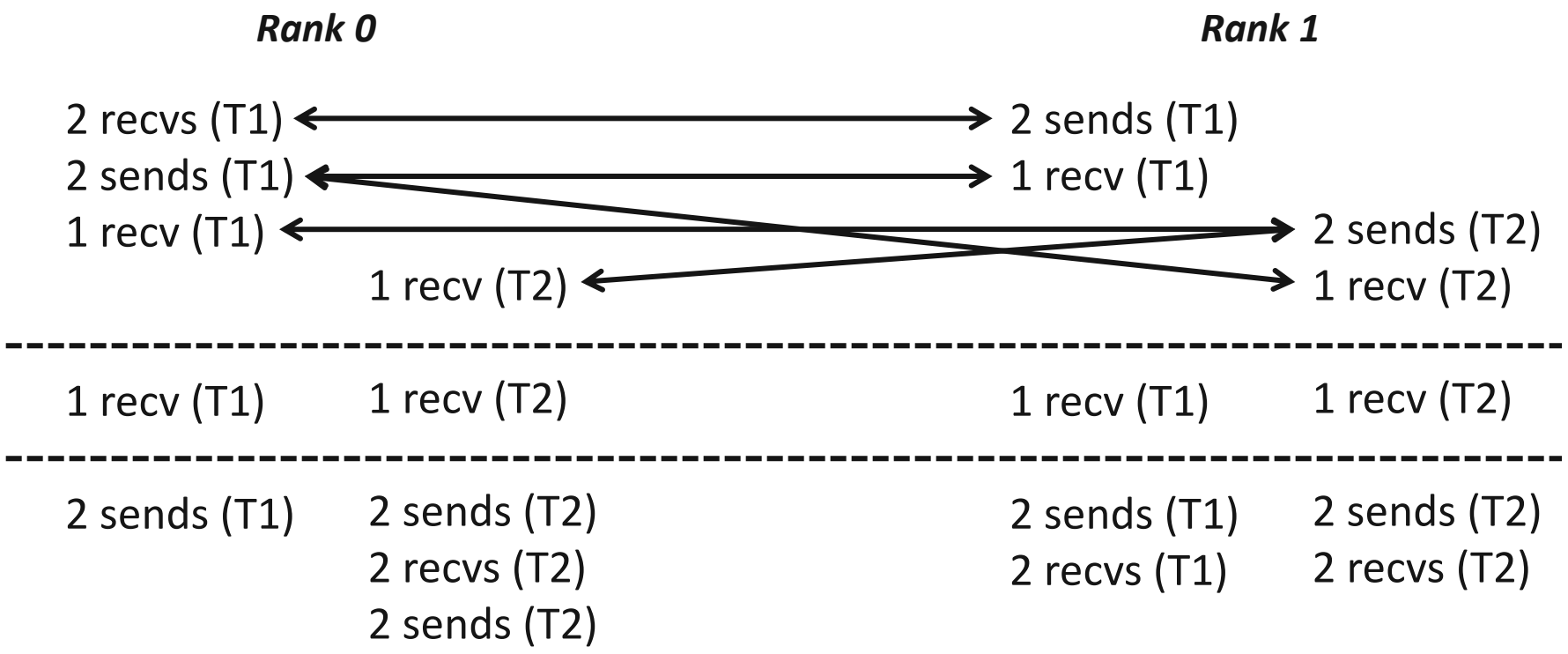
```
for (j = 0; j < 2; j++) {
     if (rank == 1) {
          for (i = 0; i < 2; i++)
          MPI_Send(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
          for (i = 0; i < 2; i++)
          MPI_Recv(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &stat);
     }
    else {  /* rank == 0 */
          for (i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

Intended Ordering of Operations



- Every send matches a receive on the other rank

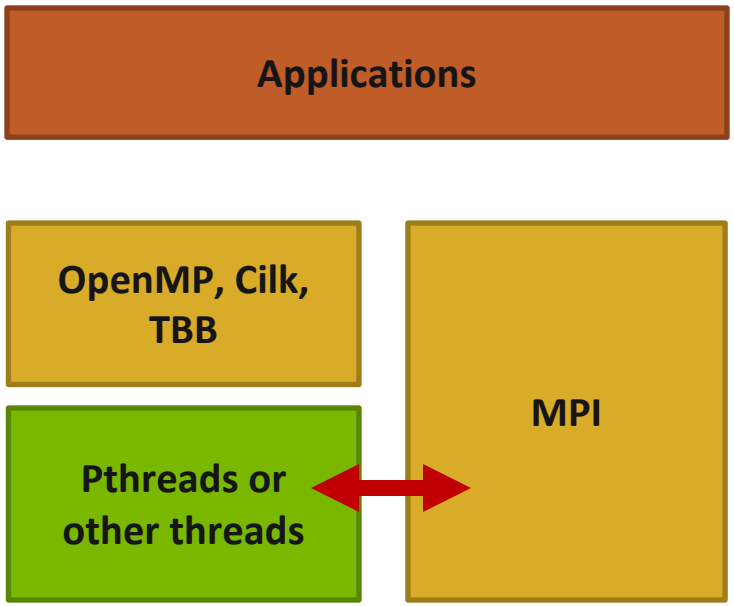
Possible Ordering of Operations in Practice



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

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;
}
```

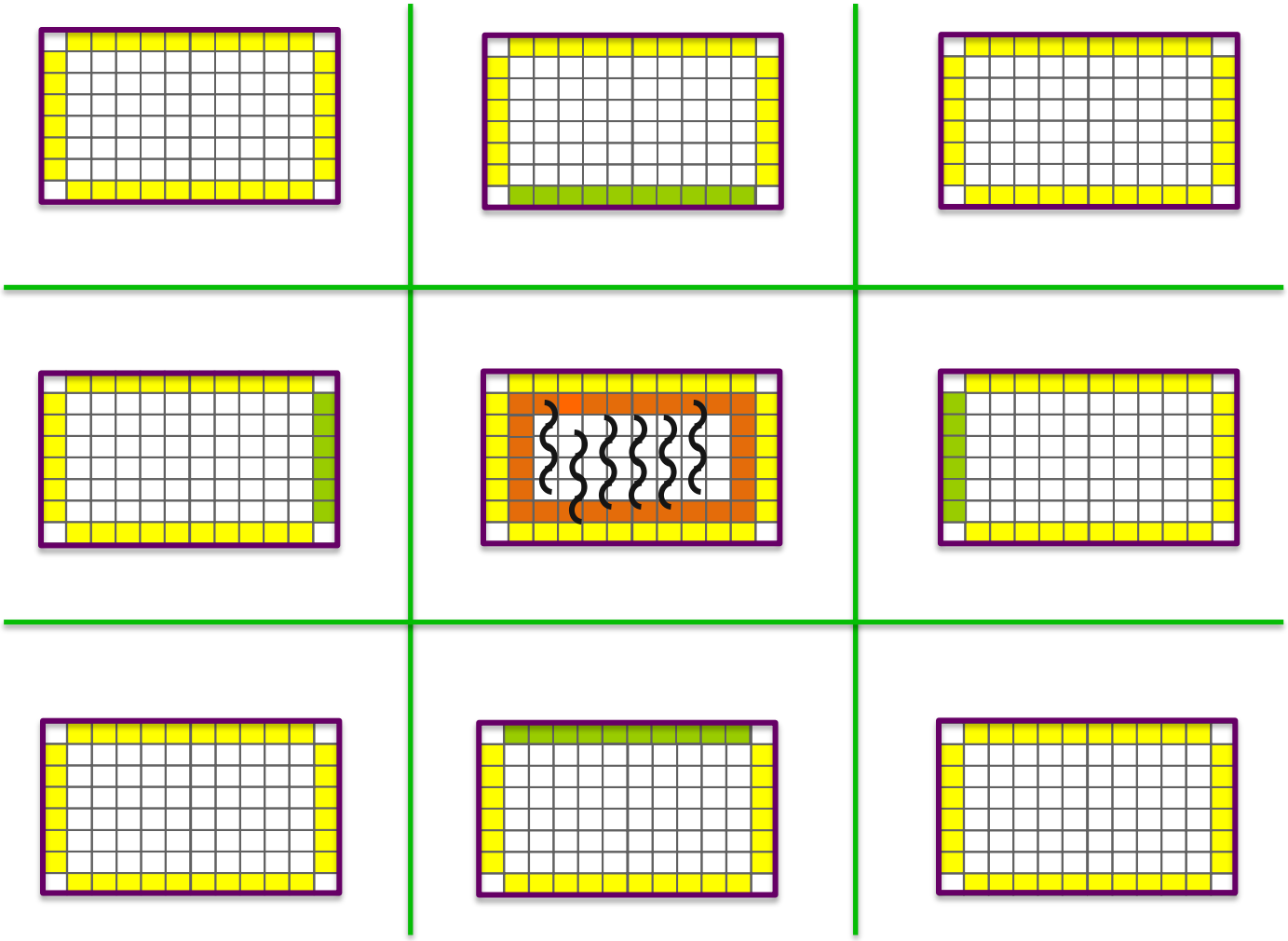
Correct example. Each task is nonblocking.

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

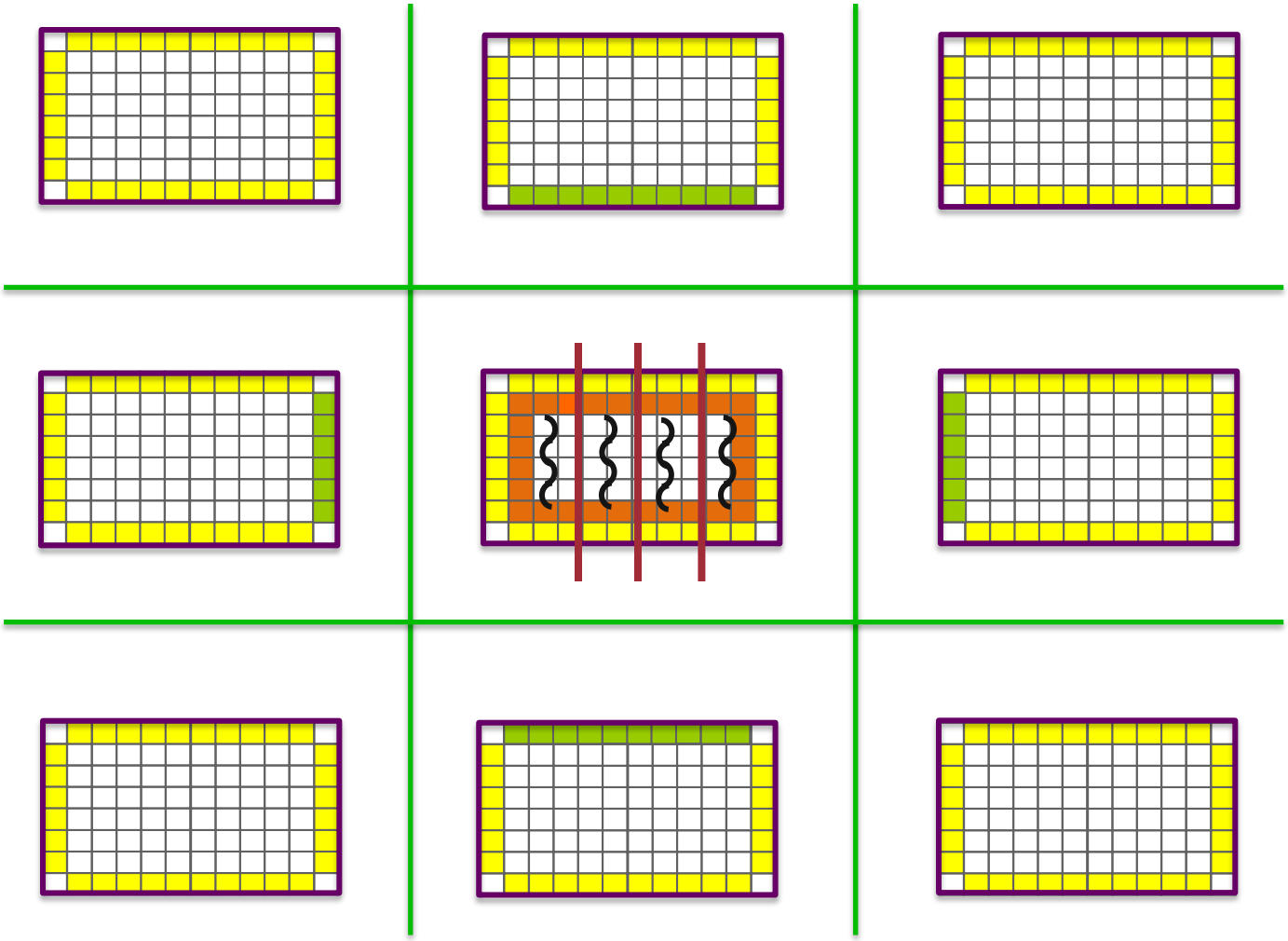
Exercise 1: Stencil in Funneled mode (1/2)



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*

Exercise 3: BSPMM in Funneled mode

- Parallelize dgemm computation (OpenMP parallel for)
- Main thread does all communication
- *Start from rma/bspmm_counter.c*
- *Solution available in threads/bspmm_funneled.c*

Exercise 4: BSPMM in Multiple mode

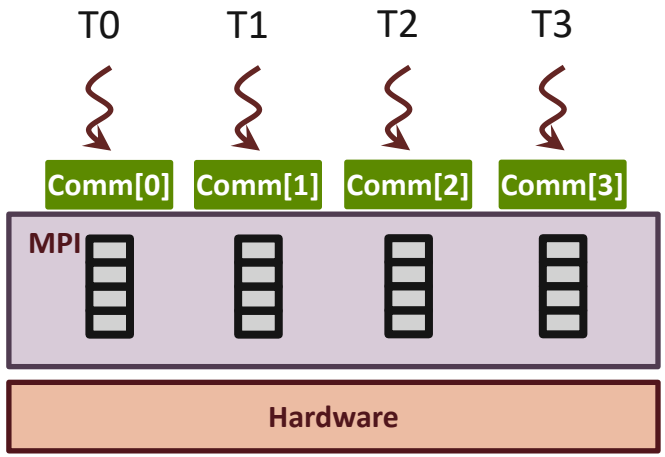
- Each thread queries the next available block multiplication
- *Start from threads/bspmm_funneled.c*
- *Solution available in threads/bspmm_multiple.c*

MPI+threads performance recommendations

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

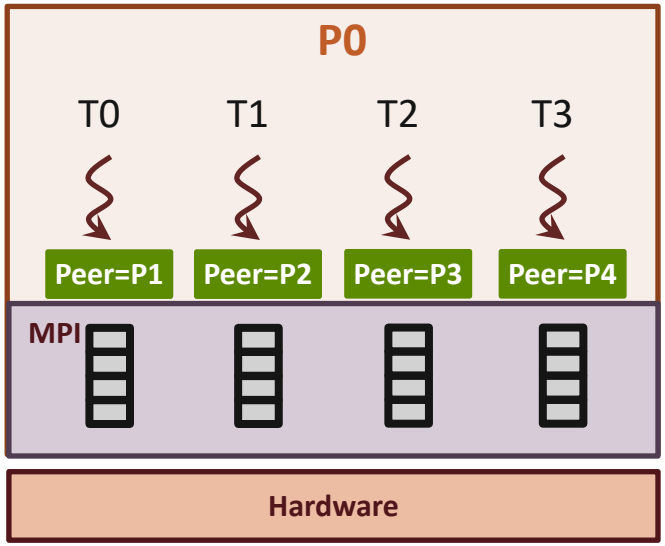
```
MPI_Comm *comms;
int nthreads = omp_get_num_threads();
comms = malloc(sizeof(MPI_Comm) * nthreads);

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, ..);
```



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

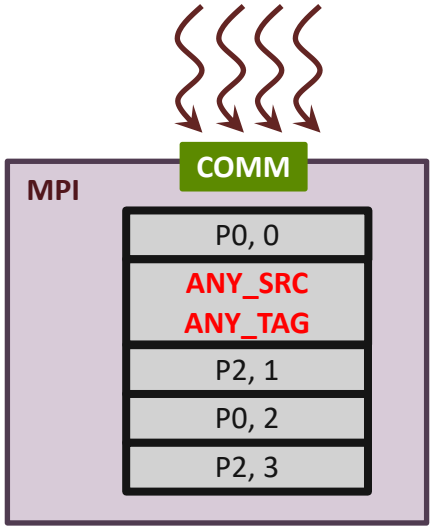
- Each thread communicates with **different peer_rank or tag**
  - MPI may assign isolated resource for different set of [peer_rank + tag]

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



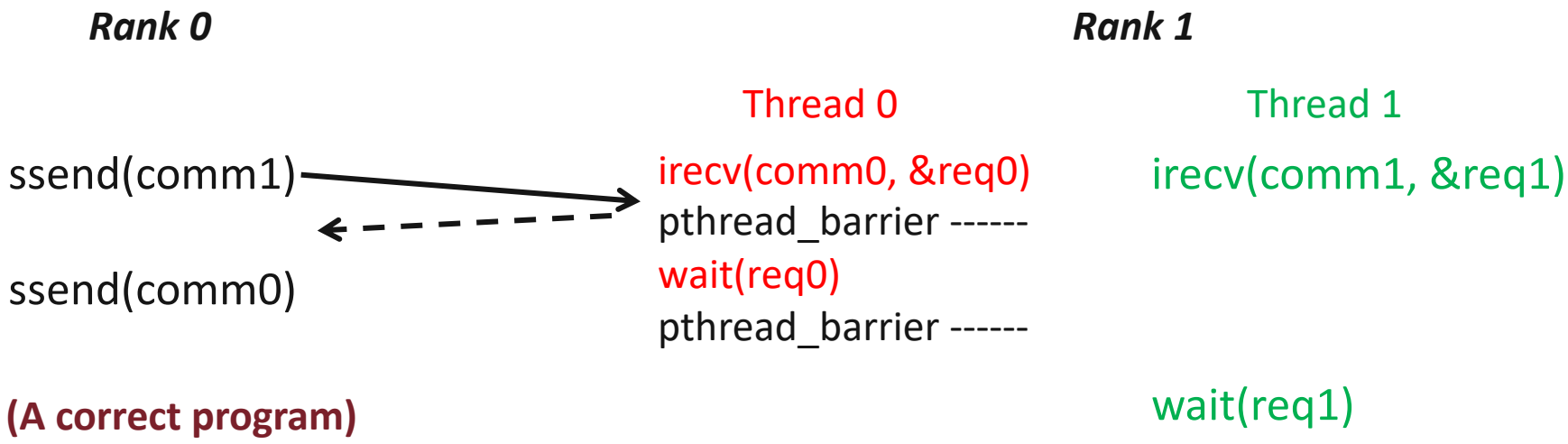
Recommendation: Maximize independence between threads with ranks or tags (2/2)

- 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 */
```

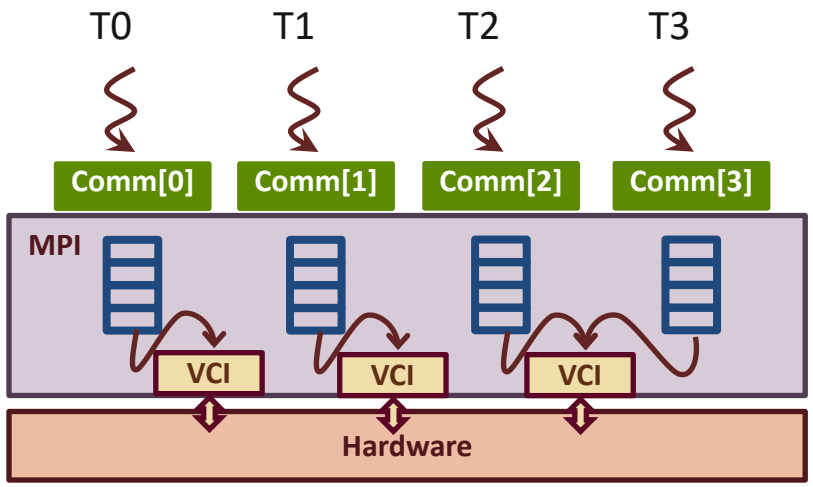
Communication Isolation Limitations



- ***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



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*

Exercise 6: BSPMM with overlapping windows

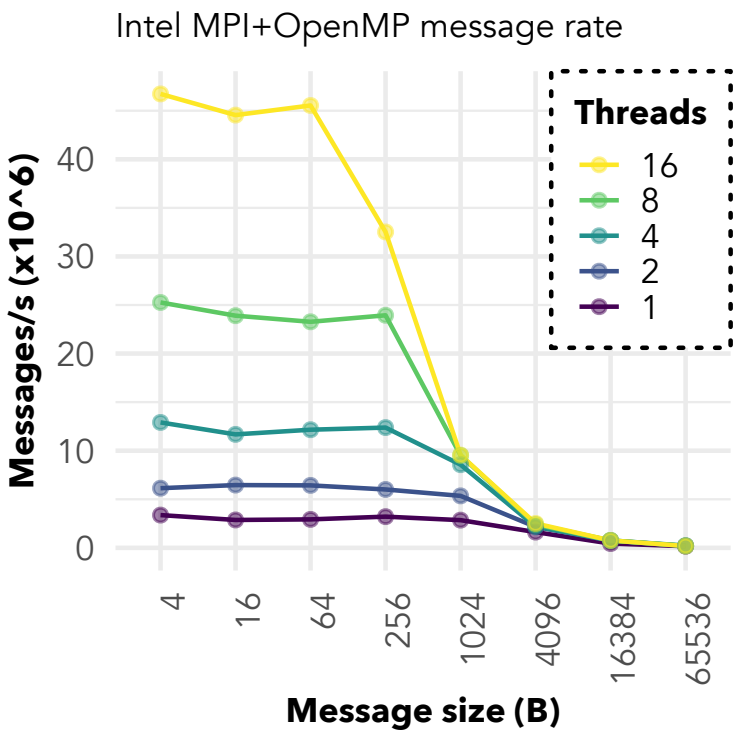
- Each thread queries the next available block multiplication
- Each thread uses a different window (overlapped) for accessing A and B global matrices
  - Have to use a single window for C matrix and global counter in order to ensure atomicity
- *Start from threads/bspmm_multiple.c*
- *Solution available in threads/bspmm_multiple_nwins.c*

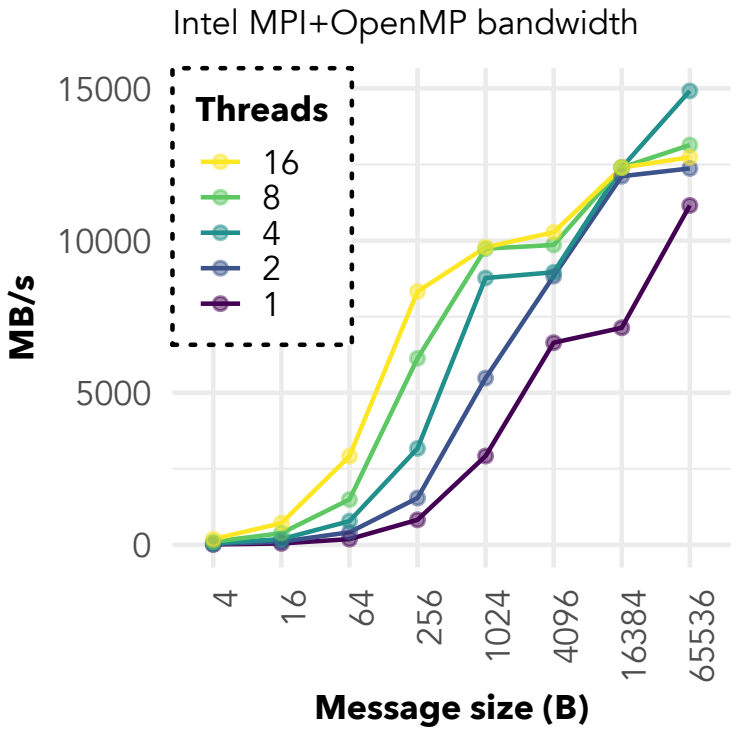
MPI+threads optimizations in Intel MPI 2019

- Very restricted version
- MPI_Init_thread(MPI_THREAD_MULTIPLE)
- Environment variables
  - I_MPI_THREAD_SPLIT=1
  - I_MPI_THREAD_RUNTIME=openmp
- Restriction
  - Communicators shared only by threads with the same thread ID on all processes
- 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

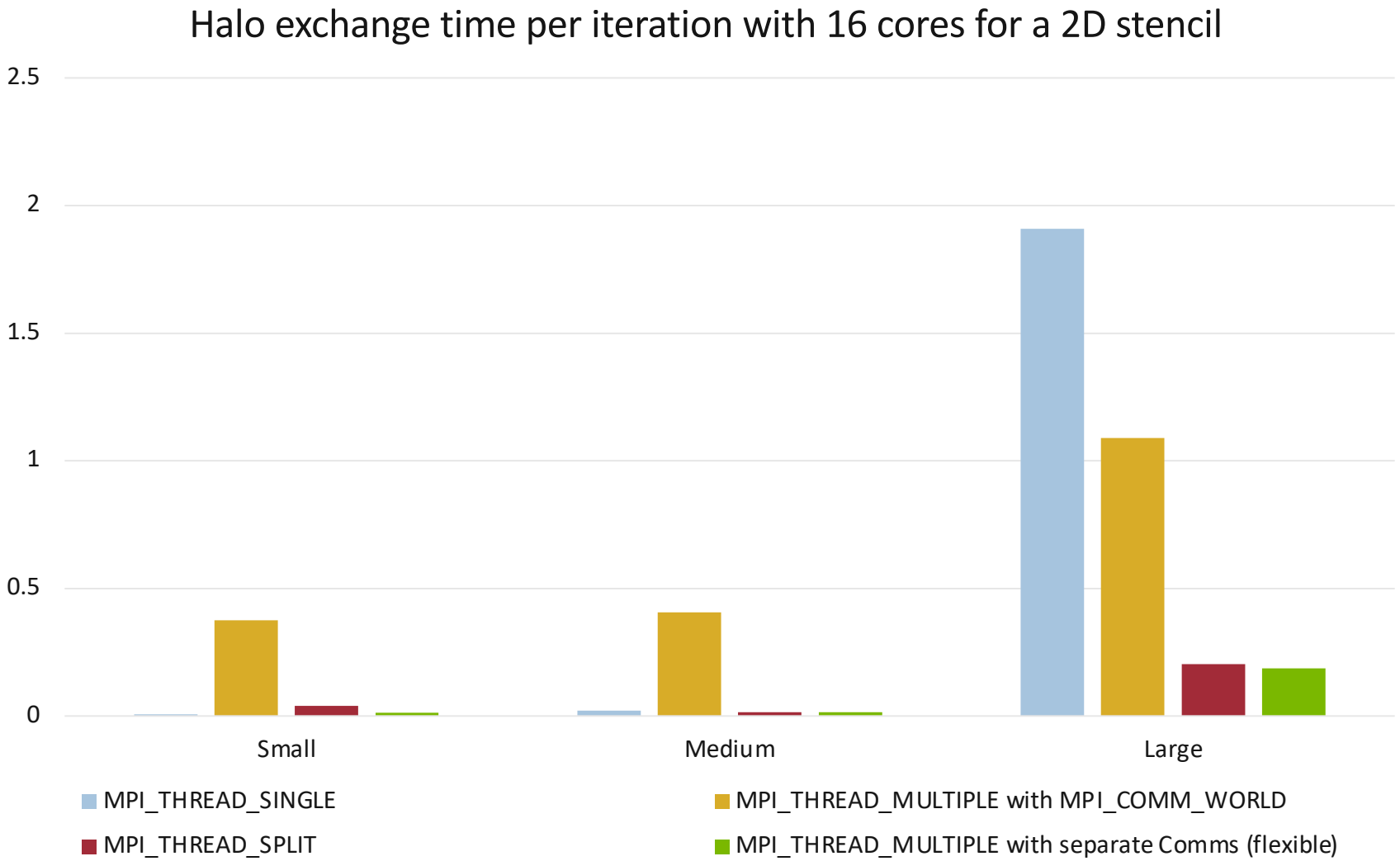
Message-rate and Bandwidth with MPI_THREAD_SPLIT





- Each thread communicating with another using separate communicators.

MPI libraries can exploit exposed communication parallelism



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

Argonne NATIONAL LABORATORY

MPI Hybrid Programming: Shared Memory

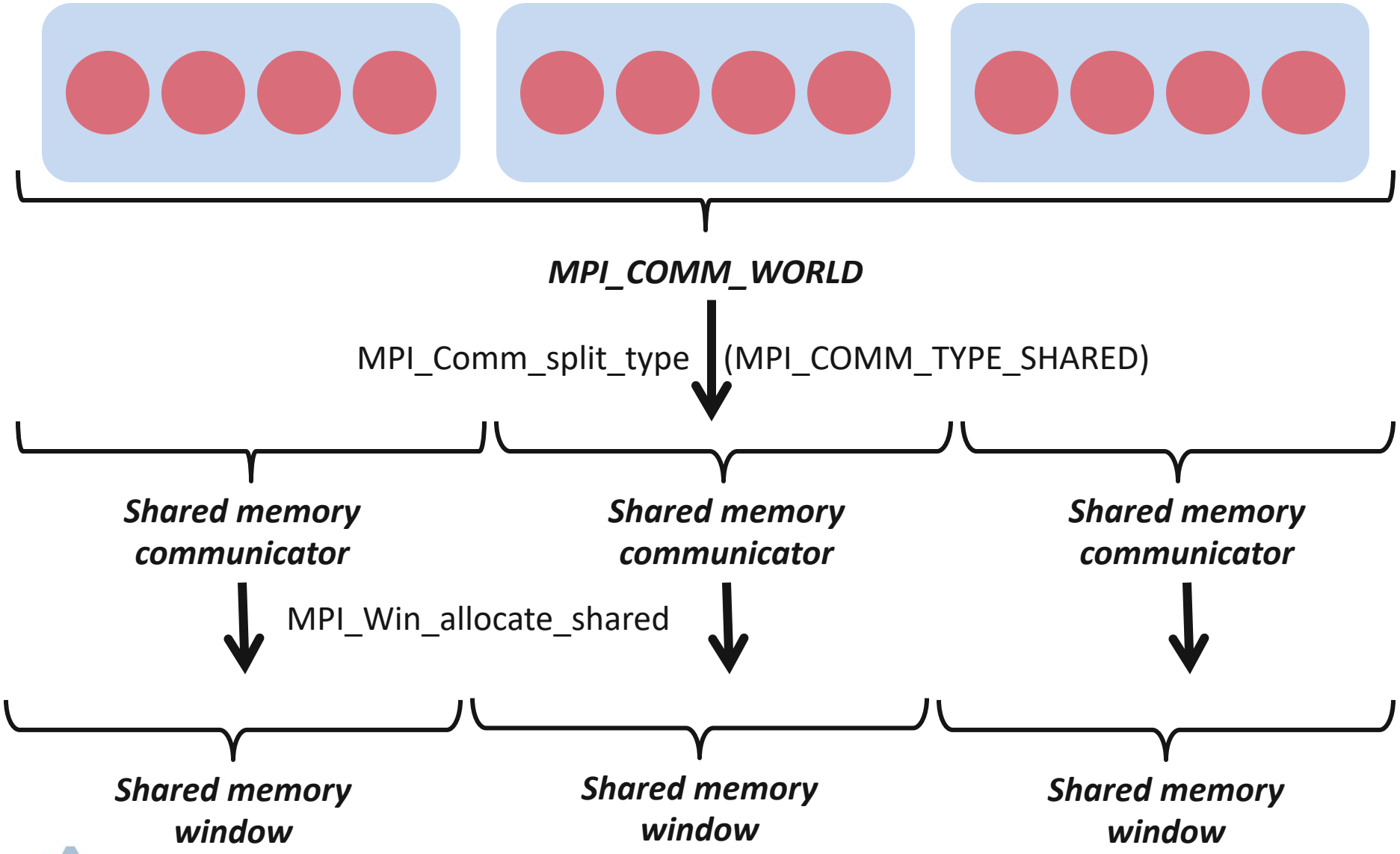
Slides Available at https://anl.box.com/v/yguo-isc-tutorial-2021

U.S. DEPARTMENT OF ENERGY

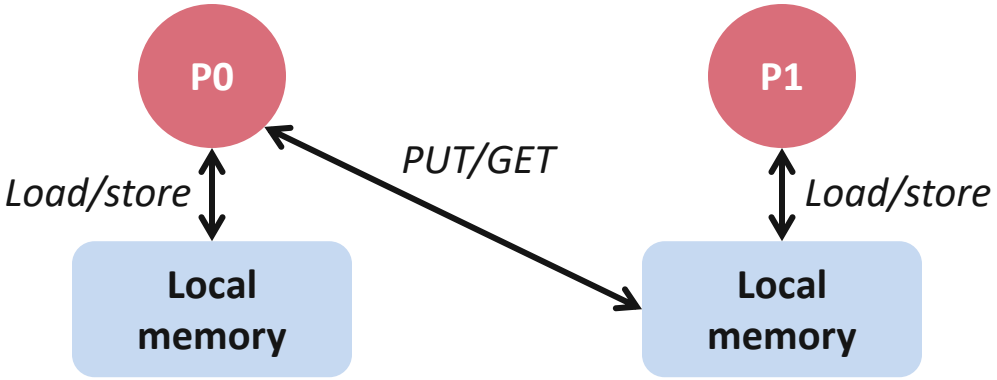
Hybrid Programming with Shared Memory

- MPI-3 allows different processes to allocate shared memory through MPI
  - MPI_Win_allocate_shared
- Uses many of the concepts of one-sided communication
- Applications can do hybrid programming using MPI or load/store accesses on the shared memory window
- Other MPI functions can be used to synchronize access to shared memory regions
- Can be simpler to program than threads

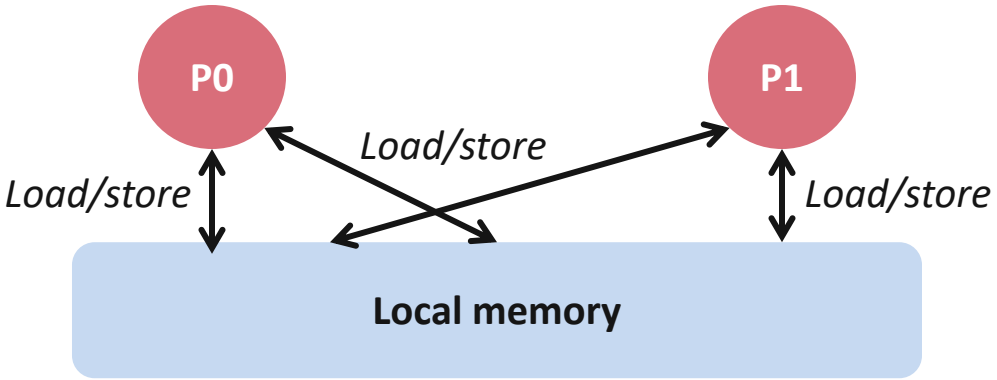
Creating Shared Memory Regions in MPI



Regular RMA windows vs. Shared memory windows



Traditional RMA windows



Shared memory windows

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

MPI_COMM_SPLIT_TYPE

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

- Create a communicator where processes “share a property”
  - Properties are defined by the “split_type”
- Arguments:
  - comm - input communicator (handle)
  - split_type - property of the partitioning (integer)
  - key - rank assignment ordering (nonnegative integer)
  - info - info argument (handle)
  - newcomm - output communicator (handle)

MPI_WIN_ALLOCATE_SHARED

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

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

Shared Arrays with Shared memory windows

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

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

    MPI_Win_lockall(win);

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

    /* use shared memory */

    MPI_Win_unlock_all(win);

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

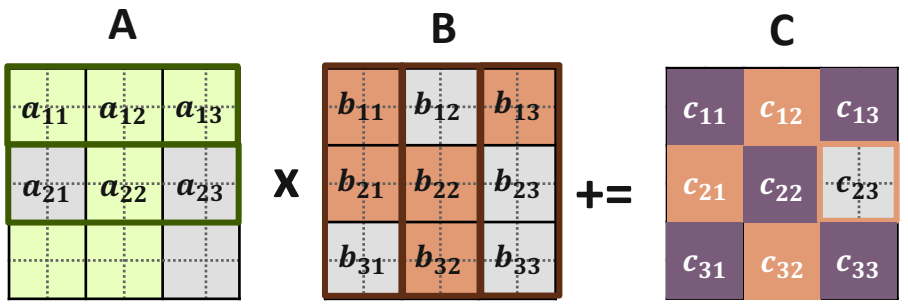
Memory allocation and placement

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

Example: BSPMM with Shared Memory (1/2)

- *shared_mem/bspmm_counter.c*
- Shift to C-based parallelism to eliminate atomic updates to C
  - Every process computes different C block

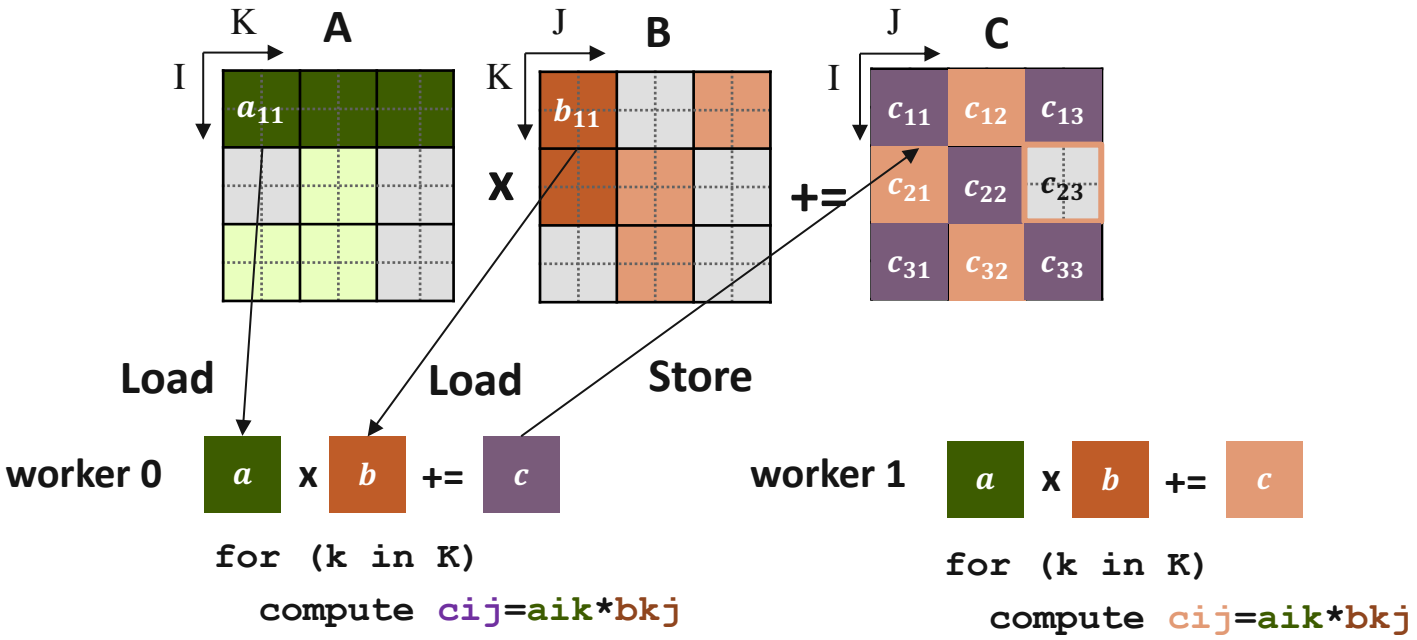
rank 0	rank 1
c11+=a11*b11	Skip a11*(b12)
c11+=a12*b21	c12+=a12*b22
Skip a13*(b31)	c12+=a13*b32
c13+=a11*b13	Skip (a21)
Skip a12*(b23)	c21+=a22*b21
Skip a13*(b33)	Skip (a23)
...	...



C-based parallelism
Set α=1, β=0 for simplicity, compute $C = AB$

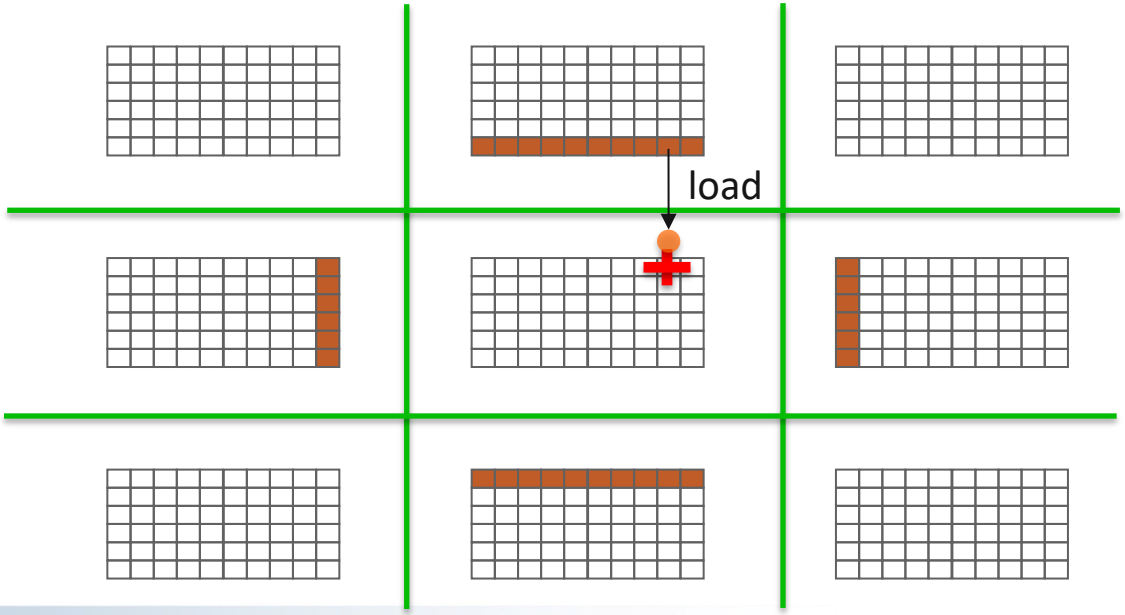
Example: BSPMM with Shared Memory (2/2)

- Replace RMA Get and Accumulate with direct load and store operations from and to the global matrices



Exercise: Stencil with Shared Memory

- Message passing model requires ghost-cells to be explicitly communicated to neighbor processes
- In the shared-memory model, there is no communication. Neighbors directly access your data.
- *Start from rma/stencil_lock_put.c*
- *Solution available in shared_mem/stencil.c*



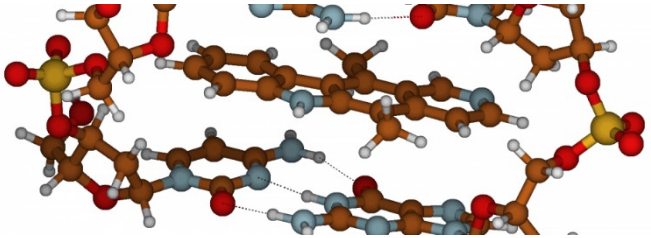
Threads vs. Process Shared Memory

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

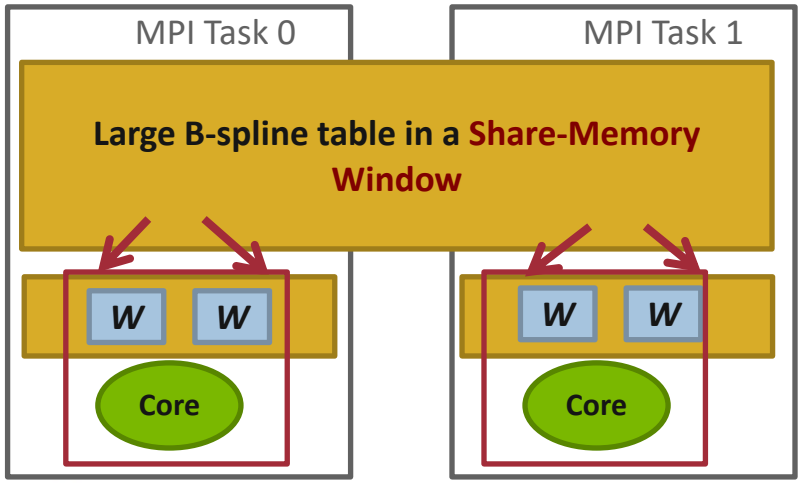
Example: Quantum Monte Carlo

- Memory capacity bound with MPI-only
- Hybrid approaches
  - MPI + threads (e.g. X = OpenMP, Pthreads)
  - MPI + shared-memory (X = MPI)
- Can use direct load/store operations instead of message passing

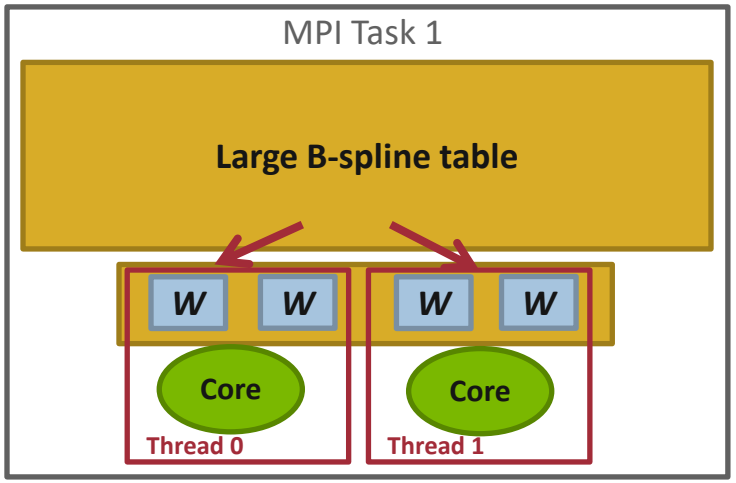
QMCPACK



MPI + Shared-Memory (MPI 3.0)
- **Everything private by default**
- **Expose shared data explicitly**



MPI + Threads
- **Share everything by default**
- **Privatize data when necessary**



Shortcomings: Compiler Interference (1/2)

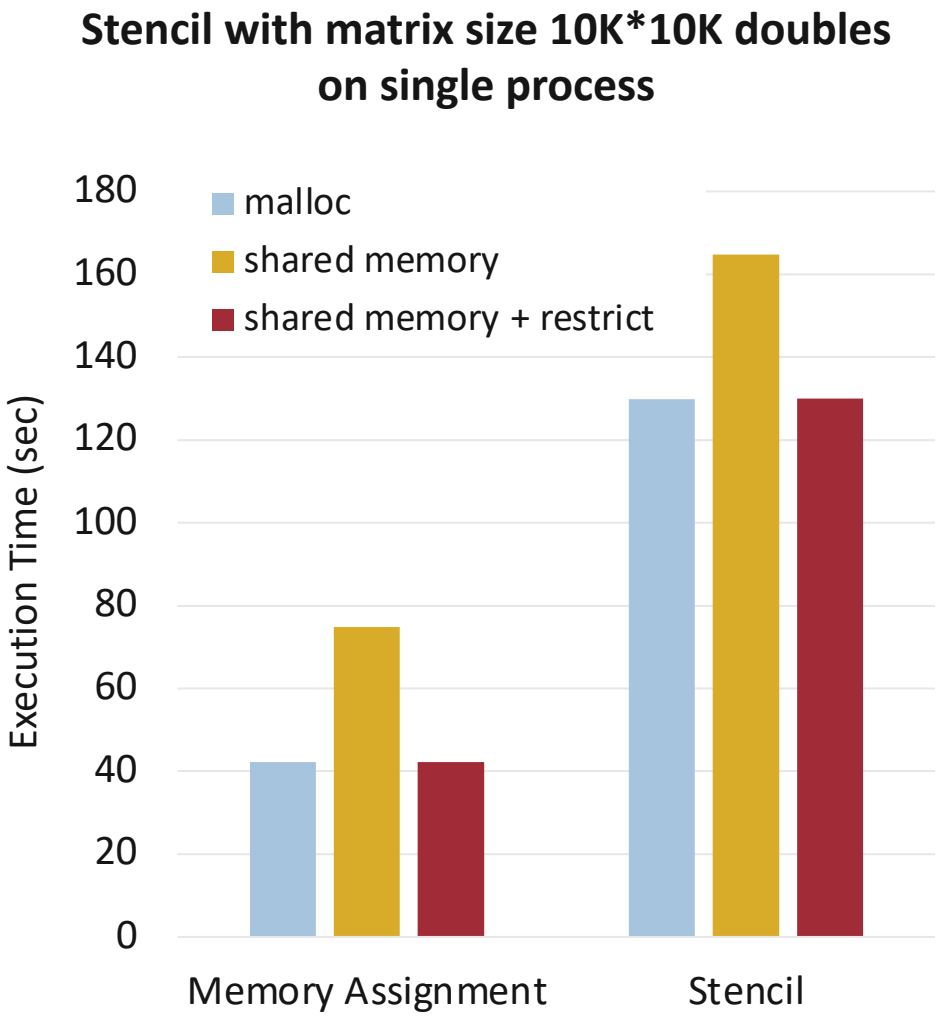
- "malloc" is special
  - Glibc special attribute decoration

```
__attribute__((__malloc__))
```

  - Tells compiler that the allocated buffer **does not contain pointers to other memory regions**
    - By modifying this buffer or anything derived from this buffer, I cannot update other memory locations in the code (no aliasing semantics)
    - Possible benefits: compiler can rearrange the code more freely, shorter assembly code

- MPI_Win_allocate_shared
  - Returns the memory allocated through the baseptr argument instead of the return value, and hence does not have this attribute
  - If MPI_Win_allocate_shared was modified to not return an error code, but to return the buffer, we could have used the above trick!
  - Solution: use **restrict** attribute

Shortcomings: Compiler Interference (2/2)




```
void stencil(int size)
{
    double *mem;
    ...
    mem =(double*)malloc(
        sizeof(double) * size);

    /* stencil computation on mem */

    free(mem);
}
```

```
void stencil(int size)
{
    double restrict *mem;
    MPI_Win win;
    ...
    MPI_Win_allocate_shared(
        size * sizeof(double),
        sizeof(double),..., &mem, &win);

    /* stencil computation on mem */

    MPI_Win_free(&win);
    ...
}
```

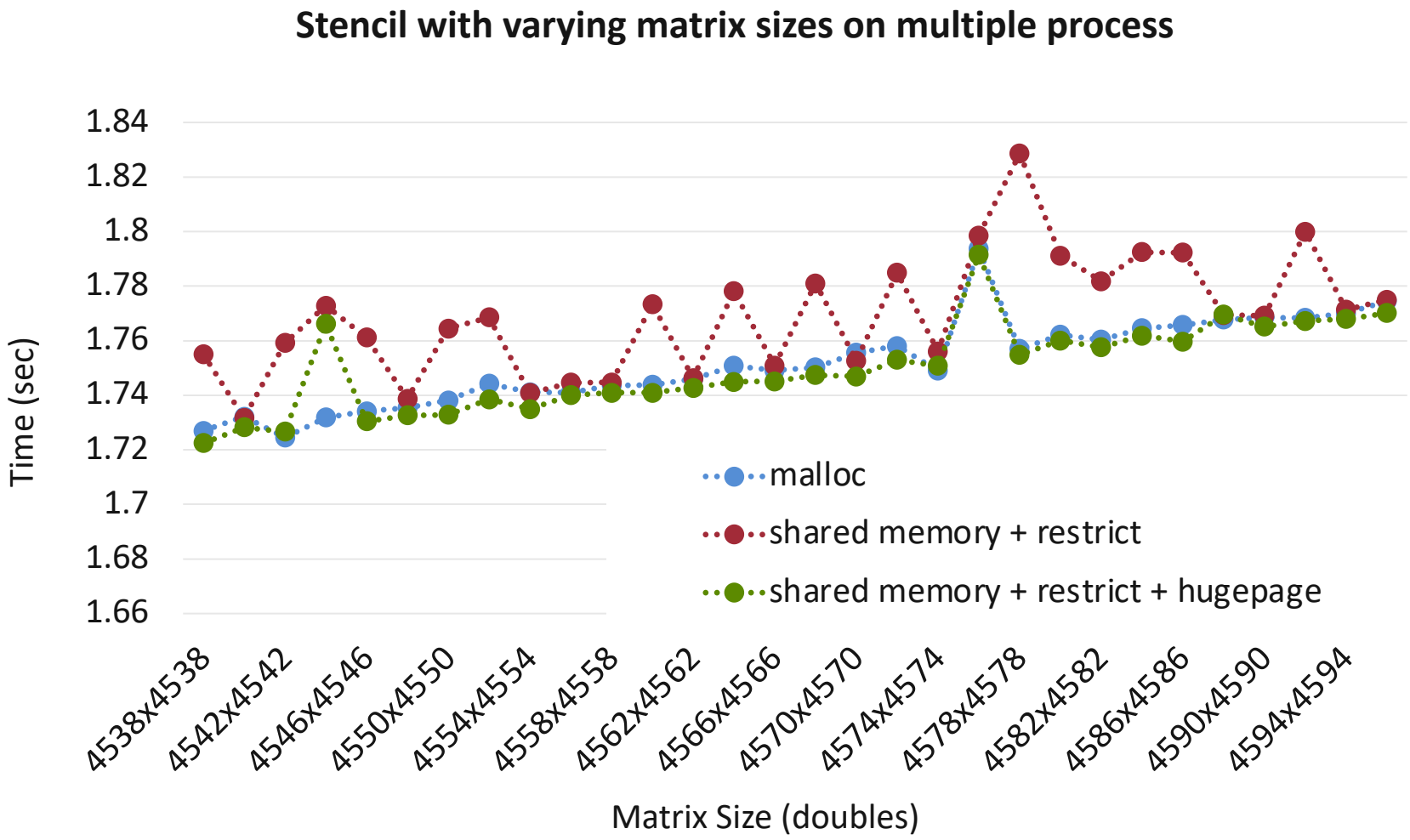
Shortcomings: OS Interference (1/3)

- OS treats regular memory allocation and shared memory allocation differently
  - Single process memory allocation internally does an anonymous mmap
    - The OS "likes" this type of memory allocations and **assigns large pages (2MB)** to back such memory
  - Multi-process memory allocation (shared memory) internally does a file-backed mmap
    - The OS assigns **regular sized pages (4KB)** to back such memory

Shortcomings: OS Interference (2/3)

- Solutions:
  - Idea is to modify the OS to give us **huge pages for shared memory**
  - System settings (enable hugepage-backed ramfs):
    - the hugepage number in /proc/sys/vm/nr_hugepages
    - the group id of huge page in /proc/sys/vm/hugetlb_shm_group
    - These two can also be done by the sysctl command by the root, or they can be set in the configuration file /etc/sysctl.conf.
  - Within the MPI implementation, allocate the file backing the shared memory on “hugetlbfs”

Shortcomings: OS Interference (3/3)



Shortcomings: Restricted Allocation Methods

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

Section Summary

- Hybrid programming with MPI-3 shared memory
  - MPI_Comm_split_type + MPI_Win_allocate_shared
  - Can use MPI functions (e.g., RMA operations) or direct load/store to access the shared memory
- MPI + Process shared memory vs. MPI + Threads
  - **Shared memory**: only share memory and few allocated objects
  - **Threads**: more resource sharing, more objects sharing, computation structure fits high-level thread-based parallelism (e.g., OpenMP loops)
- Some performance optimizations:
  - Use **restrict** for your shared memory pointer
  - Enable **hugepage**

Argonne NATIONAL LABORATORY

MPI Hybrid Programming: Accelerators

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

Slides Available at https://anl.box.com/v/yguo-isc-tutorial-2021

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 private memory
  - Small shared memory
  - Shallow pipelines
  - Large off-chip global High-Bandwidth Memory (HBM)
  - High FLOPs/W and FLOPs/$

Top500 Accelerators Based Systems (Nov 2018)

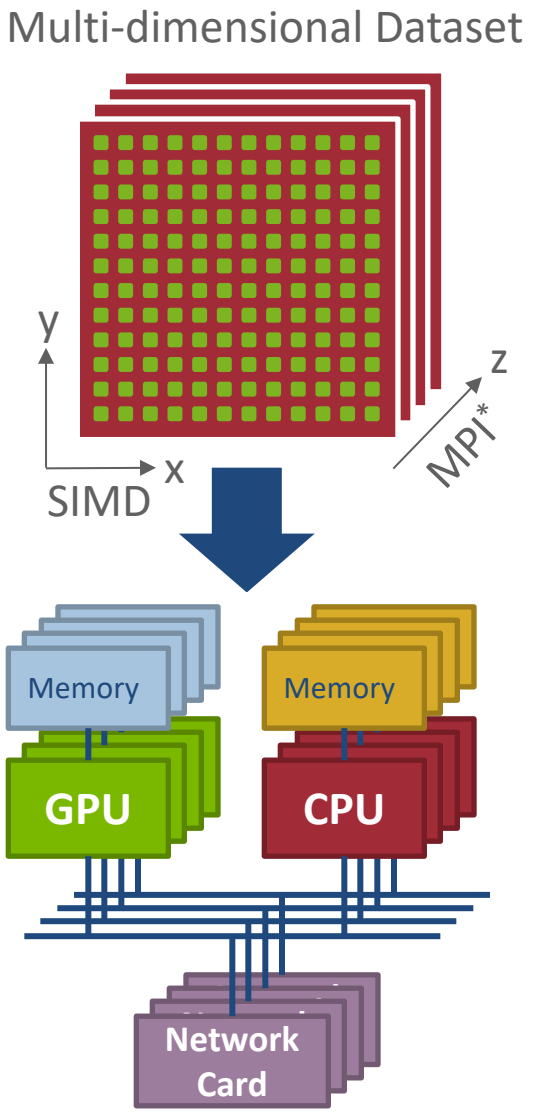
- #1 - Summit (ORNL USA)
  - NVIDIA Volta GV100
- #2 - Sierra (LLNL USA)
  - NVIDIA Volta GV100
- #5 - Piz Daint (CSCS Switzerland)
  - NVIDIA Tesla P100
- #7 - AI Bridging Cloud Infrastructure (AIST Japan)
  - NVIDIA Tesla V100
- #9 - Titan (ORNL USA)
  - NVIDIA K20X

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
- Frontier (ORNL USA)
  - AMD based GPU technology
    - https://www.ornl.gov/news/us-department-energy-and-cray-deliver-record-setting-frontier-supercomputer-ornl
- Tianhe-3 (NUDT China)
  - Custom

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 (our focus will be on their memory model and interaction with MPI)
  - NVIDIA CUDA (NVIDIA platform only)
  - AMD ROCm & HIP
  - OpenCL & SYCL
  - OpenMP
  - …

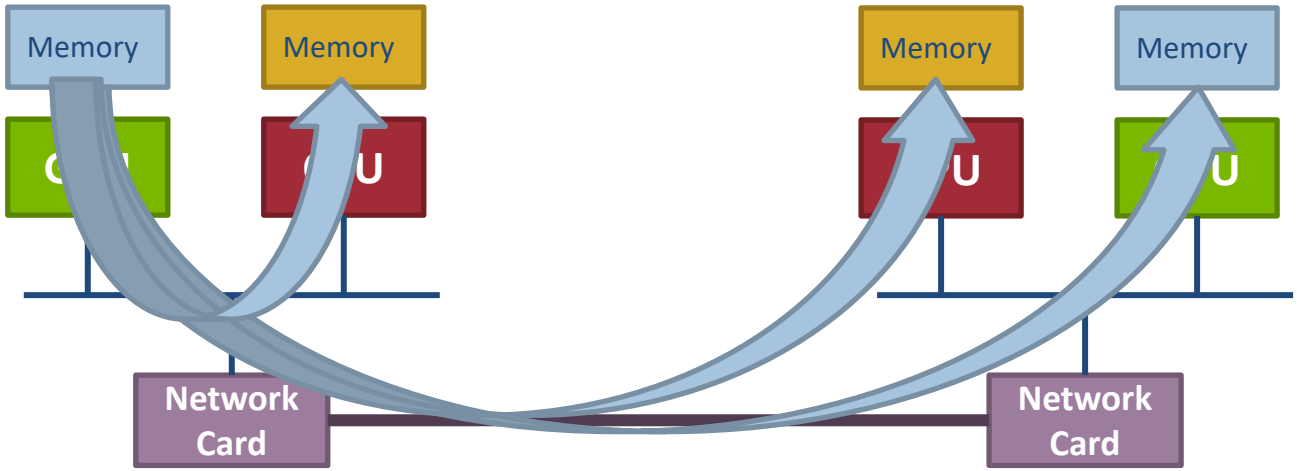


(*)Single Program Multiple Data

Interoperability with MPI

GPUs have separate physical memory subsystem

How to move data between GPUs with MPI?

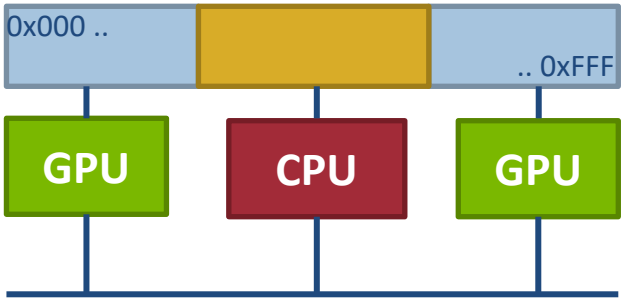


Real answer: It depends on what GPU library, what hardware and what MPI implementation you are using

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

Unified Virtual Addressing (UVA)

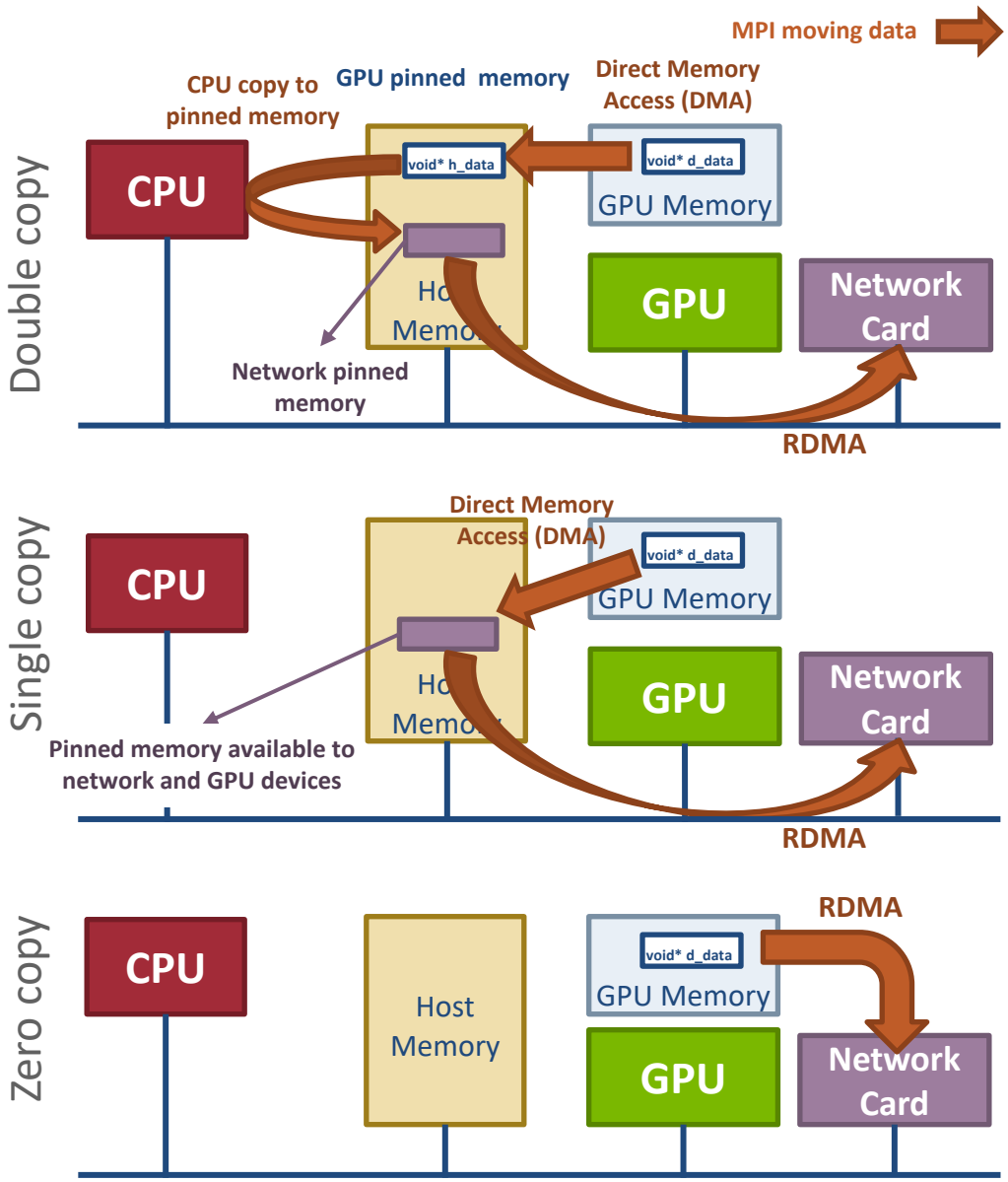
- UVA is a memory address management system supported in modern 64-bit architectures
  - Requires device driver support
- The same virtual address space is used for all processors, host or devices
- No distinction between host and device pointers
- The user can query the location of the data allocation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

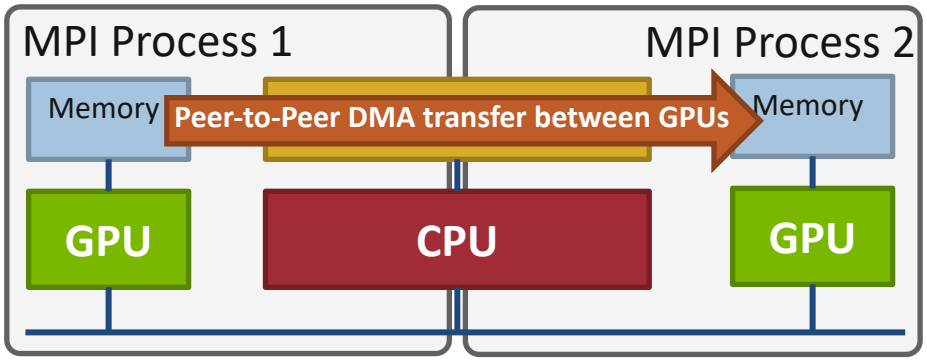
Remote Direct Memory Access with UVA

- 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



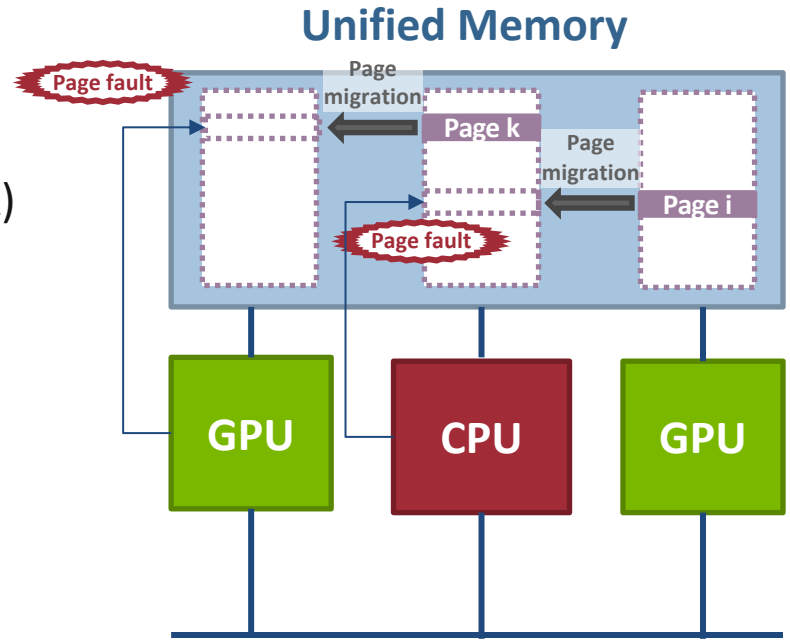
Intranode Communication with UVA

- Intranode Optimization
  - GPU peer-to-peer data transfers are possible
  - MPI can directly move data between GPU devices
- Limitation
  - PCIe devices doing peer-to-peer transfers have to share the same upstream root complex



Heterogeneous Memory Management (HMM)

- Supported in the Linux Kernel since version 4.14 through helper functions to be used by device drivers
  - Support for Shared Virtual Addresses (SVA) allowing devices to work with host virtual addresses directly
  - Support paging in device for migrating memory between host and device
- Automatic data management between host and GPU memories (called Unified Memory in CUDA)
  - Data is automatically migrated between the host and devices on page faults
  - Moving pages to device and back to host is similar to swap-out and swap-in of pages to and from disk



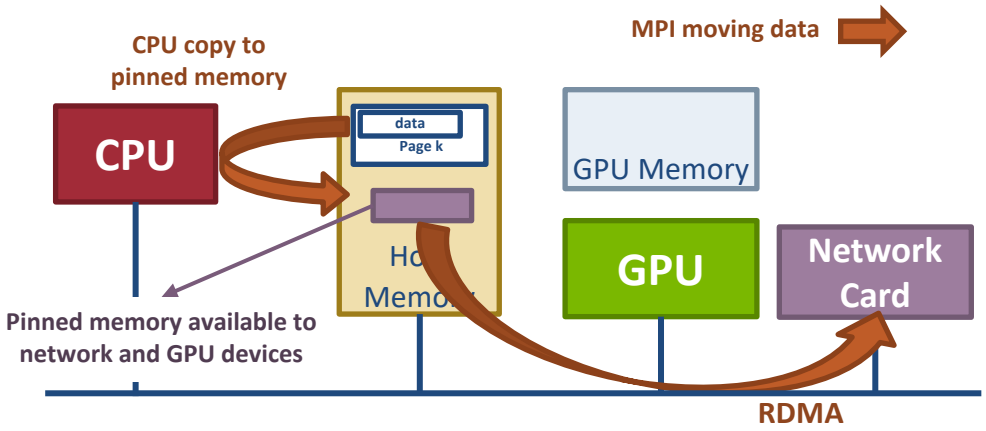
Single memory space accessible to all devices and host. Transparently managed heterogeneous memory.

MPI + HMM in a Nutshell

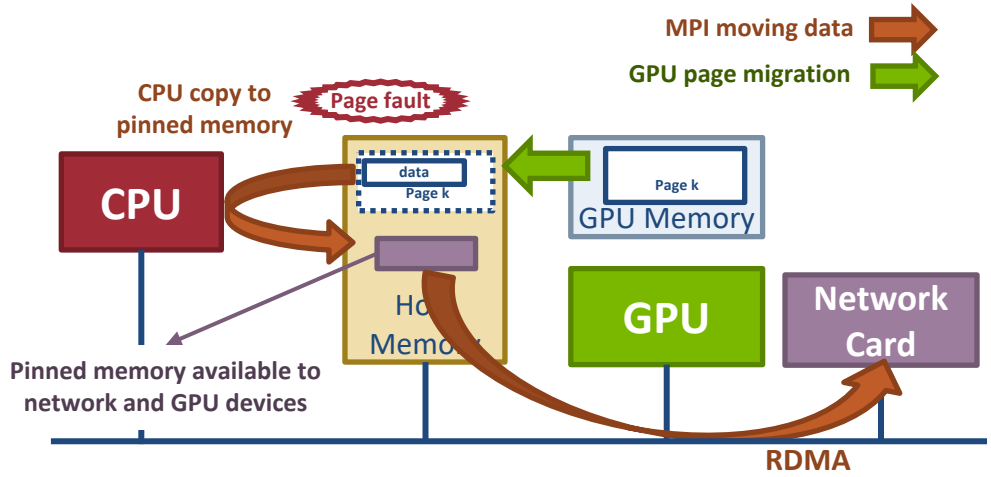
- In theory, any MPI implementation can transparently work with HMM
  - The MPI implementation can simply assume that the data is on the host
  - GPUs take care of moving data between device and host memories
  - *(In theory, the network memory registration should simply fail for HMM allocations, but there have been reports of silent failures in this regard for CUDA, so you might need to be careful)*
- *But performance can be bad:*
  - Issue #1: MPI does not know where the data is located
    - Data management is completely handled by GPU
  - Issue #2: managed heterogeneous memory cannot be directly accessed by the network
    - Virtual address cannot be pinned to a fixed physical memory region since GPU might need to migrate the data (bounce buffer is needed)

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



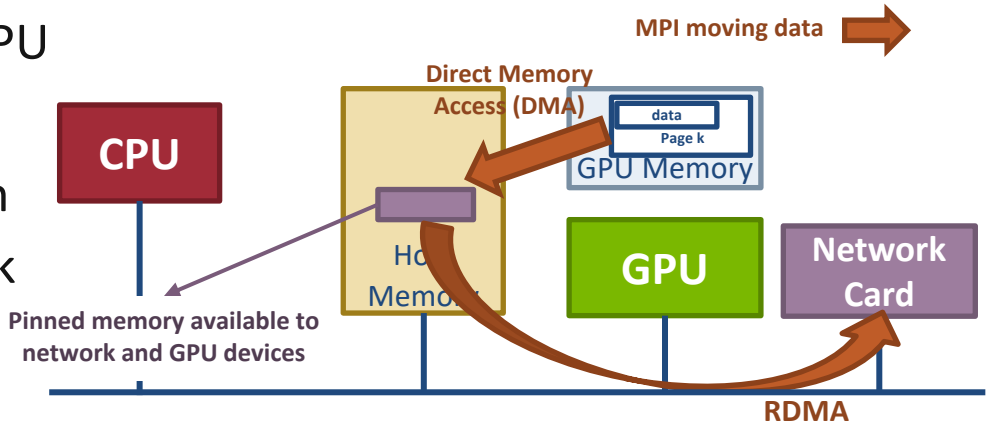
Correct guess: data is on host memory



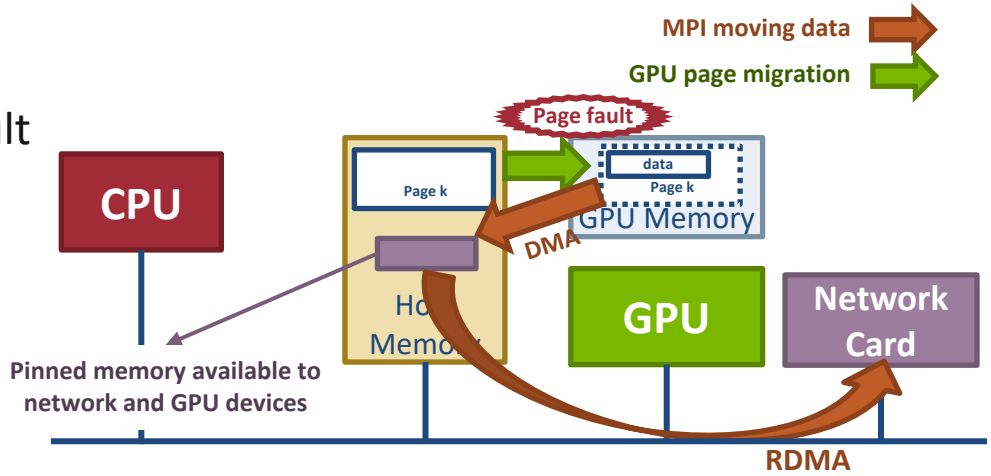
MPI moving data from host memory on a wrong guess

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



Correct guess: data is on the target GPU



MPI moving data from host memory on a wrong guess

Argonne NATIONAL LABORATORY

Compute Unified Device Architecture (CUDA)

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

Slides Available at https://anl.box.com/v/balaji-tutorials-2019/

U.S. DEPARTMENT OF ENERGY

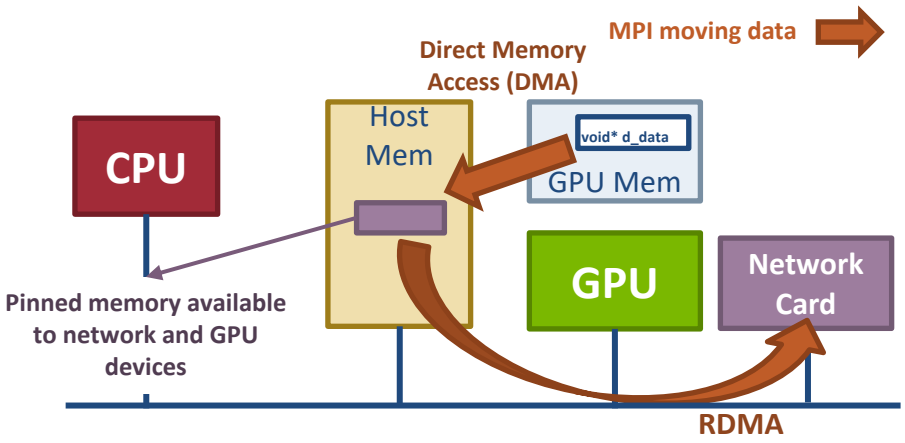
Overview

- 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
  - 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,
                           int size,
                           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, size, out_d);

  cudaMemcpy(out_h, out_d, size,
             cudaMemcpyDeviceToHost);

  [...snip...]

  return 0;
}
```

MPI + CUDA-4 with GPUDirect

- 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




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

if(my_rank == sender) {
    gpu_kernel<<<..>>>(dev_buf);
    cudaDeviceSynchonize();
    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);
}
```

- Register memory to ensure CUDA memory operations complete before returning control:
  - cuPointerSetAttribute(…, CU_POINTER_ATTRIBUTE_SYNC_MEMOPS, …)

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<<<..>>>(dev_buf);
    cudaDeviceSynchronize();
    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 + CUDA Managed Memory

- Unified Memory 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
- Managed Memory is not guaranteed to work with MPI
- As mentioned before, performance can also 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<<<..>>>(buf);
    cudaDeviceSynchronize();
    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
  - MLNX_OFED v2.1-x.x.x or later
  - Plugin Module to enable GPUDirect RDMA
- NVIDIA GPUs:
  - NVIDIA Tesla, Quadro K-Series or Tesla/Quadro P-Series
  - NVIDIA Driver
  - NVIDIA Runtime and Toolkit

Argonne NATIONAL LABORATORY

Radeon Open Compute Platform & Heterogeneous-compute Interface for Portability (ROCm & HIP)

(*Thanks to Brad Benton from AMD for review and comments*)

Slides Available at https://anl.box.com/v/balaji-tutorials-2019/

U.S. DEPARTMENT OF ENERGY

Overview

- General-purpose parallel computing platform and programming model released by AMD in 2016 as alternative to CUDA
- Provides runtime library (*ROCr*), device drivers (*ROCk*) and C/C++ compiler (HCC)
- HCC is invoked through hipcc wrapper script
- HIP extensions for C/C++ language
  - 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 or the **hipLaunchKernelGGL** API
  - Each thread executing the kernel is given a unique thread ID accessible from inside the kernel using the built-in **threadIdx** variable

```
/* Kernel definition */
__global__ void gpu_kernel(double *in,
                           int size,
                           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);
  hipMalloc(&in_d, size);
  hipMalloc(&out_d, size);

  hipMemcpy(in_d, in_h, size,
            hipMemcpyHostToDevice);

  /* kernel invocation with N threads */
  hipLaunchKernelGGL(gpu_kernel, dim3(2),
                     dim3(N), 0, 0, in_d,
                     size, out_d);

  hipMemcpy(out_h, out_d, size,
            hipMemcpyDeviceToHost);

  [...snip...]

  return 0;
}
```

MPI + ROCmRDMA

- Since ROCm 1.9.2 (Nov 2018) there is Peer-to-peer and RDMA MPI support for the Vega GPUs (GCN architecture)
  - Including support for ROCmRDMA on Mellanox InfiniBand (PeerDirect) and support for GPU's data interaction in different nodes via MPI with HIP and OpenCL applications

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

if(my_rank == sender) {
    gpu_kernel<<<..>>>(dev_buf);
    hipDeviceSynchronize();
    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);
}
```

ROCm Supported Hardware

- Supported AMD GFX8 GPUs:
  - Fiji chips: AMD Radeon R9 Fury X and Radeon Instinct MI8
  - Polaris 10 chips: AMD Radeon RX 580 and Radeon Instinct MI6
  - Polaris 11 chips: AMD Radeon RX 570 and Radeon Pro WX 4100
  - Polaris 12 chips: AMD Radeon RX 550 and Radeon RX 540
- Supported AMD GFX9 GPUs:
  - Vega 10 chips: AMD Radeon RX Vega 64 and Radeon Instinct MI25
  - Vega 7nm chips: AMD Radeon Instinct MI50, Radeon Instinct MI60 or Radeon VII
- For the full list of supported hardware and software consult: https://github.com/RadeonOpenCompute/ROCm#new-features-and-enhancements-in-rocm-24

Argonne NATIONAL LABORATORY

OpenCL & SYCL

(*Thanks to Rod Burns from Codeplay for review and comments*)

Slides Available at https://anl.box.com/v/balaji-tutorials-2019/

Overview

- OpenCL is a low-level C/C++ based programming language for heterogeneous computing (CPUs, GPUs, FPGAs, DSPs):
  - **Platform Layer**: discover devices with their capabilities and create contexts for managing and using them
  - **Runtime**: manipulate contexts (create command queues, submit work, manage memory and kernels dependencies)
  - **Compiler**: supports OpenCL C/C++ and other representations
- SYCL is a single source high-level abstraction layer for OpenCL
  - Hides a lot of the OpenCL platform layer and runtime details (e.g., platform discovery and explicit data movement)
  - Takes advantage of C++ features like templatization, lambda functions and parallel constructs introduced by C++11 (e.g., **parallel_for**)

Memory Management in OpenCL

- OpenCL ≤ 1.2 host and device memory managed separately
  - Device memory allocated using **clCreateBuffer** returns a **cl_mem** object that can be used by OpenCL (only) for data movement
- OpenCL ≥ 2.0 host and device can share virtual memory (SVM)
  - **Coarse-grained sharing**: used for memory and virtual pointer sharing between multiple devices as well as host and one or more devices (required)
    - Device memory allocated through OpenCL SVM allocator (see next slide)
    - Multiple kernels on the same device can safely access the same memory location
    - Memory consistency is guaranteed at synchronization points
  - **Fine-grained sharing** (optional in OpenCL, requires hardware support):
    - Fine-grained *buffer* sharing: device memory, allocated through OpenCL SVM allocator, can be accessed by host directly → similar to **cudaMallocManaged**
    - Fine-grained *system* sharing: similar but system allocated and managed
    - Memory consistency for concurrent access to same area is guaranteed only if device supports atomics, undefined otherwise

Shared Virtual Memory in OpenCL

- **Buffer** shared memory allocated using **clSVMAlloc**
  - Returned SVM pointer has to be passed to device kernels through **clSetKernelArgSVMPointer**
  - Coarse-grained sharing: host can access device memory by mapping it to its address space through **clEnqueueSVMMap** and once done unmap it through **clEnqueueSVMUnmap**
  - Fine-grained sharing: host can directly access device memory at the lowest granularity (only accessed pages are moved to host)
- **System** shared memory allocated through **malloc** or **mmap**
  - Host pointer has to be passed to device kernels through **clSetKernelArgSVMPointer**
  - Can also pass the kernel the head of a linked list without passing every single node pointer explicitly (requires HMM support from HW & OS)

OpenCL SVM Summary

	Coarse-grained	Fine-grained
Buffer Allocation	Sharing happens at the SVM buffer granularity	Sharing happens at the load/store granularity on the SVM buffer
System Allocation	Not applicable	Sharing happens at the load/store granularity on any host allocated memory - provided that the device has an explicit entry pointer (e.g., head of a linked list)
Memory Consistency	At synchronization points and **clEnqueueSVMMap /clEnqueueSVMUnmap**)	If device does not support atomics concurrent accesses to same area result in undefined behavior

MPI + Coarse-Grained SVM in OpenCL (1/2)

```
/* Coarse-grained sharing */
__kernel void gpu_kernel(__global double *buf)
{
    [...snip...]
}

/* Create context, program, kernel, and command queue */
double *dev_buf;
dev_buf = clSVMAlloc(ctx, CL_MEM_READ_WRITE, size, 0);

clSetKernelArgSVMPointer(kernel, 0, dev_buf);

if(my_rank == sender) {
    clEnqueueNDRangeKernel(queue, kernel, …);
    clEnqueueSVMMap(queue, CL_TRUE, CL_MAP_READ, dev_buf, …);
    MPI_Isend(dev_buf, size, MPI_DOUBLE, receiver, 0, comm, req);
    clEnqueueSVMUnmap(queue, dev_buf, …);
} else {
    clEnqueueSVMMap(queue, CL_TRUE, CL_MAP_READ, dev_buf, …);
    MPI_Recv(dev_buf, size, MPI_DOUBLE, sender, 0, comm, &status);
    clEnqueueSVMUnmap(queue, dev_buf, …, &event);
    clEnqueueNDRangeKernel(queue, kernel, …, 1, &event, …);
}
```

Block until mapping is done

Dependency/Synchronization

MPI + Coarse-Grained SVM in OpenCL (2/2)

- Host and device can share virtual memory, however
  - The specification does not define the behavior if host tries accessing shared memory without first mapping it (the behavior is implementation specific)
    - Since coarse-grained sharing support is mandatory, implementations do not have to rely on any specific hardware memory management feature
    - One implementation can chose to allocate a host buffer and a hidden device buffer and use the same virtual address for both, moving data on demand (at Map/Unmap and/or synchronization points)
  - The specification does not define any query API that third party libraries can use to discover SVM pointers (no **clPointerGetAttribute** type of interface)
    - MPI implementations cannot make any assumption on the nature of the pointers given by the user

Memory Management in SYCL

- Like OpenCL, SYCL kernels are ran asynchronously
- Unlike OpenCL, SYCL does not require the user to explicitly manage data dependencies and synchronization
- The SYCL runtime provides memory consistency by automatically handling data dependencies
- In order to do so SYCL defines:
  - **Buffers** represent memory allocated using default or user defined allocators
  - **Accessors** define access modes (read, write, read_write, …) to buffers providing SYCL with all the information it needs to manage memory access dependencies

Shared Virtual Memory in SYCL

- Introduced in SYCL 2.0
- Specification is not yet finalized and is therefore still evolving
- Eventually will provide same SVM functionalities as OpenCL ≥ 2.0
  - Coarse-grained Buffer
  - Fine-grained
    - Buffer
    - System

OpenCL ≥ 2.0 Vendor Support

- AMD ROCm 2.0:
  - https://github.com/RadeonOpenCompute/ROCm-OpenCL-Runtime
- NVIDIA driver 378.66 (beta for evaluation):
  - https://www.nvidia.com/download/driverResults.aspx/115492/en-us
- Intel SDK for OpenCL Applications:
  - https://software.intel.com/en-us/intel-opencl
- POCL:
  - http://portablecl.org/index.html
- Currently no vendor supports GPUDirect RDMA like features using SVM

SYCL Implementations

- Intel contribution to the LLVM Clang compiler for SYCL
  - https://github.com/intel/llvm/tree/sycl
- triSYCL (v1.2.1)
  - https://github.com/triSYCL/triSYCL
- hipSYCL (v1.2.1)
  - https://github.com/illuhad/hipSYCL
- sycl-gtx (v1.2.1)
  - https://github.com/proGTX/sycl-gtx
- ComputeCpp (v1.2.1)
  - https://www.codeplay.com/products/computesuite/computecpp

Argonne NATIONAL LABORATORY

OpenMP

(*Thanks to Shintaro Iwasaki from Argonne National Laboratory for review and comments*)

Slides Available at https://anl.box.com/v/yguo-isc-tutorial-2021

Overview

- OpenMP is a multi-platform shared memory and multi-threaded programming model for C/C++ and Fortran programs providing:
  - Compiler directives for SPMD, tasking, device offload, worksharing and synchronization
  - OpenMP API runtime library routines to control execution environment, synchronization, timing, …
- Support for additional devices (a.k.a. targets) is also available starting with V4.0

```
double A[N], B[N], C[N];

for (int i = 0; i < N; i++)
  /* Initialize A and B */

/* Do vector addition on host */
#pragma omp parallel for
for (int i = 0; i < N; i++)
  C[i] = A[i] + B[i];
```

```
double A[N], B[N], C[N];

for (int i = 0; i < N; i++)
  /* Initialize A and B */

/* Do vector addition on device */
#pragma omp target
#pragma omp teams num_teams(M)
#pragma omp distribute
for (int i = 0; i < N; i += N/M)
  #pragma omp parallel for
  for (int j = i; j < i + N/M; j++)
    C[j] = A[j] + B[j];
```

Memory Management in OpenMP (1/2)

- Host and Target memories are separate and managed by the host using *Data Environments* (no SVM exposed in OpenMP)
- A Data Environment encompasses all the variables referenced in a *task* or *worksharing* construct and their access attributes relatively to the thread (these includes the usual suspects like **private**, **shared**, **firstprivate**, …)
- OpenMP defines clauses for moving data across the *Data Environment* of the host device and the target devices (accelerators) identified by the **target** construct:
  - **map(**map-type**:** list**)**: copies data from/to the data environment of the task running on the host to/from a device target data environment
  - **map-type**: can be **to** | **from** | **tofrom** | **alloc**

Memory Management in OpenMP (2/2)

- When **target** code has to run multiple times on the same data, data movement between host device and target can be minimized using the **target data** directive covering a structured block
  - similar construct is **target** (**enter** | **exit**) **data** defining arbitrary scope

```
#pragma omp target data map(to: X[0:N]) /* Get X to device and keep it there */
{
#pragma omp target map(from: Y[0:N])    /* Put Y to host data environment */
#pragma omp teams
#pragma omp distribute
#pragma omp parallel for
for (int i = 0; i < N; i++)
  /* Compute Y[i] = f(X,i) */

/* Use Y in host */

#pragma omp target map(from: Y[0:N])    /* Put Y to host data environment */
#pragma omp teams
#pragma omp distribute
#pragma omp parallel for
for (int i = 0; i < N; i++)
  /* Compute Y[i] = g(X,i);
}
```

MPI + OpenMP

- OpenMP makes no assumption on how data movement is done (e.g., HMM capable systems do not require copies)

```
double *X = malloc(sizeof(double) * N);
double *Y = malloc(sizeof(double) * N);
double *Z = malloc(sizeof(double) * N);

#pragma omp target data map(to: X[0:N]) /* Get X to device and keep it there */
{
#pragma omp target map(from: Y[0:N])    /* Put Y to host data environment */
#pragma omp teams
#pragma omp distribute
#pragma omp parallel for
for (int i = 0; i < N; i++)
  /* Compute Y[i] = f(X,i) */

/* Use Y in host */

#pragma omp target map(from: Y[0:N])    /* Put Y to host data environment */
#pragma omp teams
#pragma omp distribute
#pragma omp parallel for
for (int i = 0; i < N; i++)
  /* Compute Y[i] = g(X,i) */
}
MPI_Allreduce(Y, Z, N, MPI_DOUBLE, MPI_SUM, comm);
```

Example: stencil_omp

- *accelerators/stencil_omp.c*
- Solves the same stencil problem presented for point-to-point moving computation to accelerators using OpenMP targets

Summary

- Accelerators are becoming increasingly important
- MPI is playing its role in enabling the usage of accelerators across distributed memory nodes
- The situation with MPI + GPU support is improving in both MPI implementations and in GPU hardware/software capabilities
  - 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 data movement between host and device memory is still required
  - For OpenMP/OpenACC data movement is managed through directives

Argonne NATIONAL LABORATORY

Nonblocking Collectives

Slides Available at https://anl.box.com/v/yguo-isc-tutorial-2021

U.S. DEPARTMENT OF ENERGY

Nonblocking Collective Communication

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

Nonblocking Collective Communication

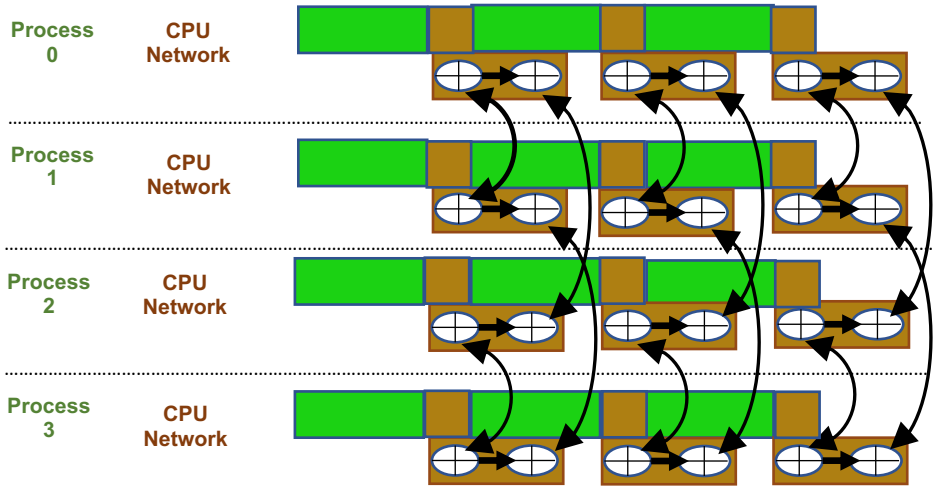
- Nonblocking variants of all collectives
  - MPI_Ibcast(<bcast args>, MPI_Request *req);
- Semantics
  - Function returns no matter what
  - No guaranteed progress (quality of implementation)
  - Usual completion calls (wait, test) + mixing
  - Out-of order completion
- Restrictions
  - Send and vector 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

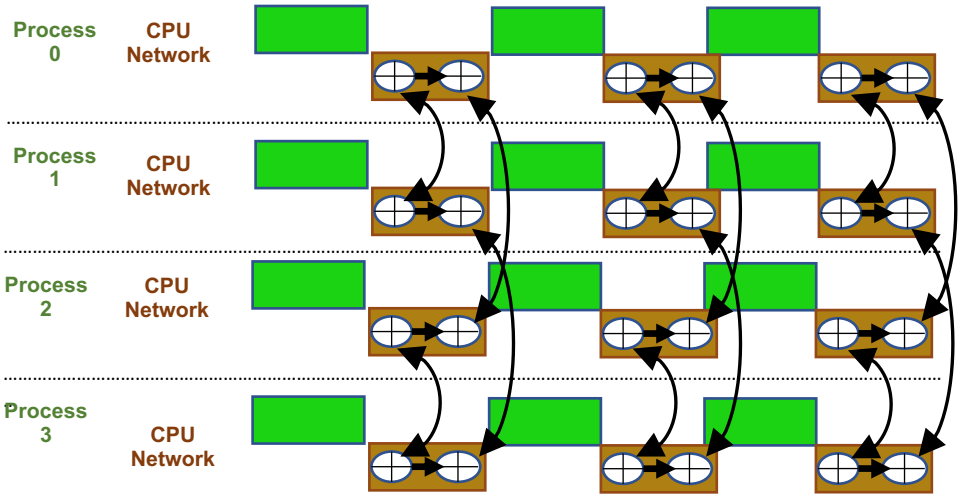
Nonblocking Collective Communication

- 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

Nonblocking Collectives Overlap

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





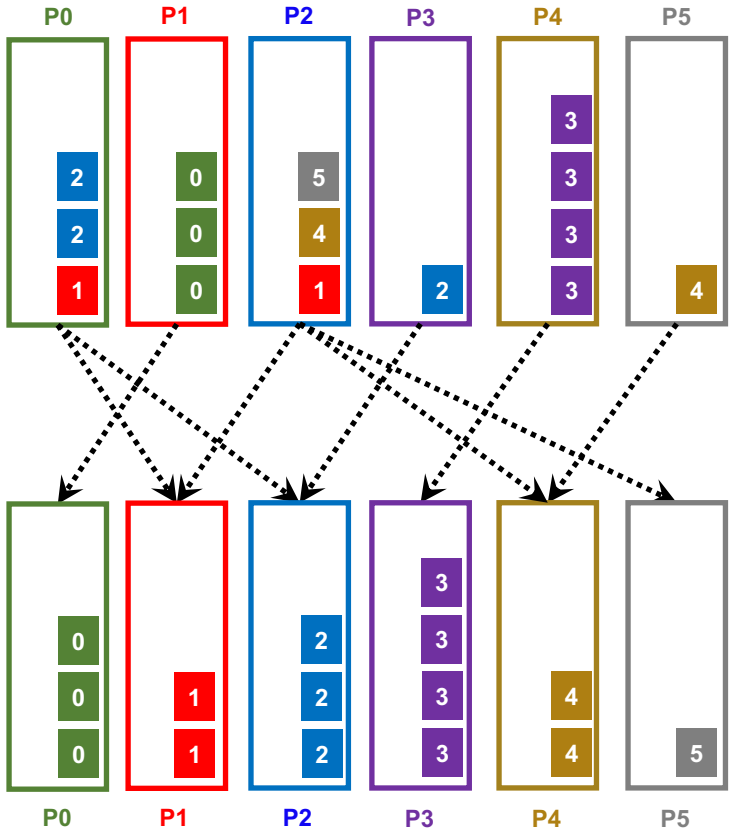
A Nonblocking Barrier?

- 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!

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)

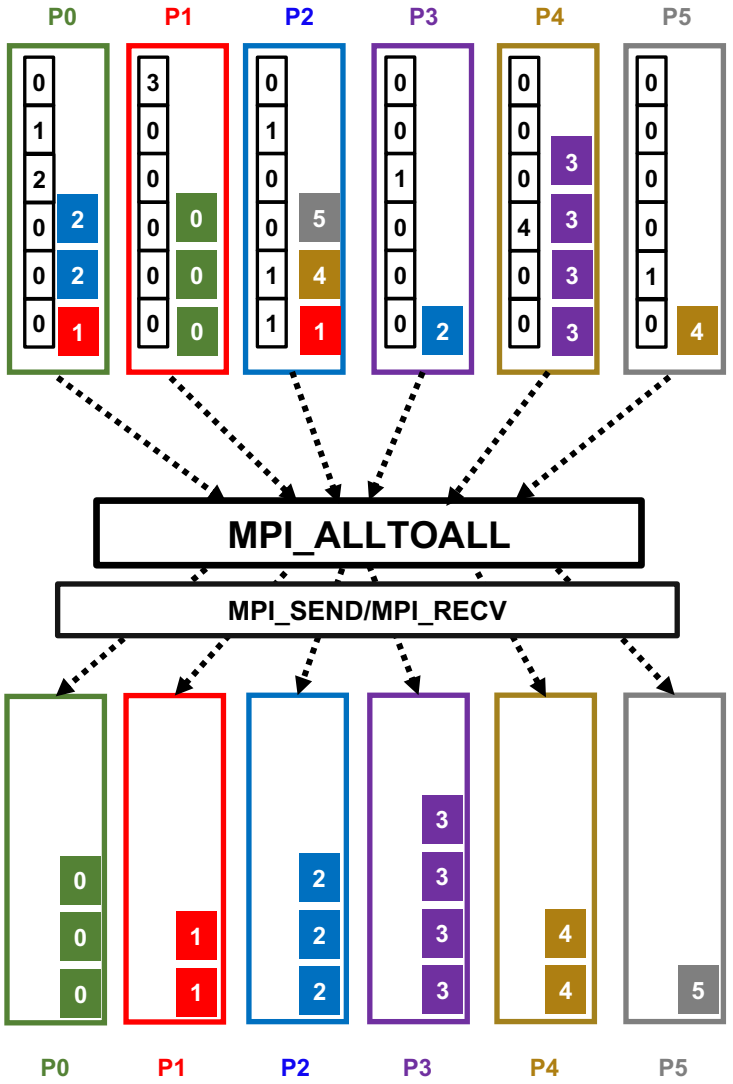
  OR:
  - Use MPI semantics:
    - Unknown sender (MPI_ANY_SOURCE)
    - Unknown message size (MPI_PROBE)
    - Reduces problem to counting the number of neighbors
    - Allow faster implementation!



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

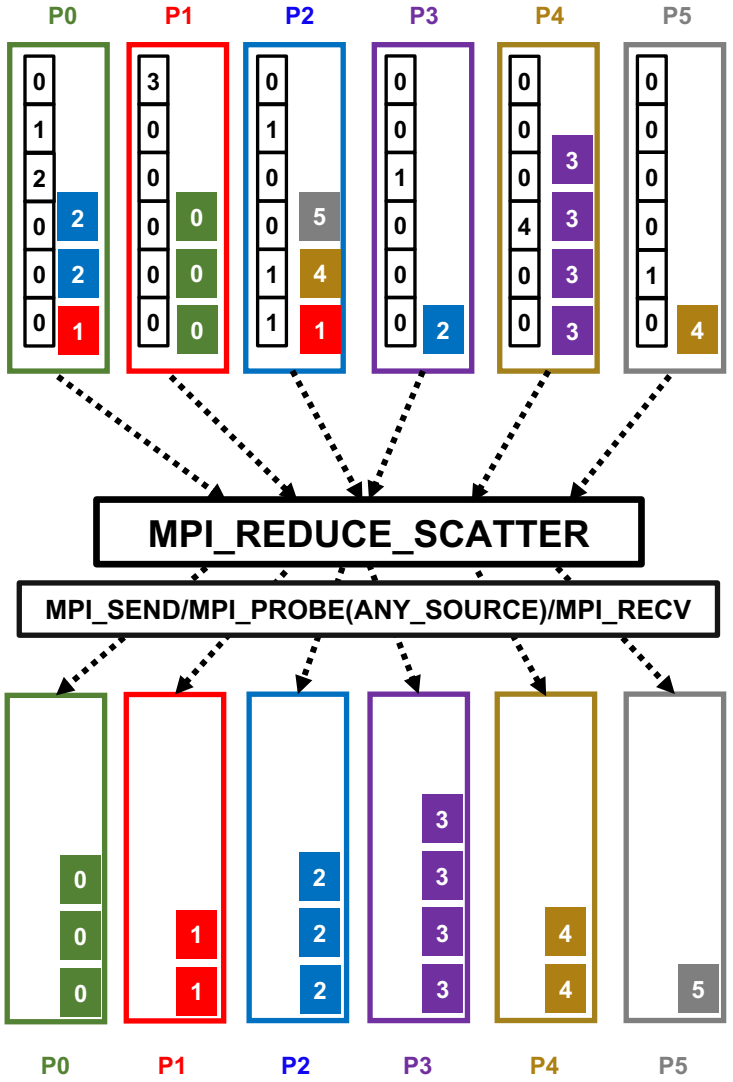
Using Alltoall (PEX)

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



Reduce_scatter (PCX)

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

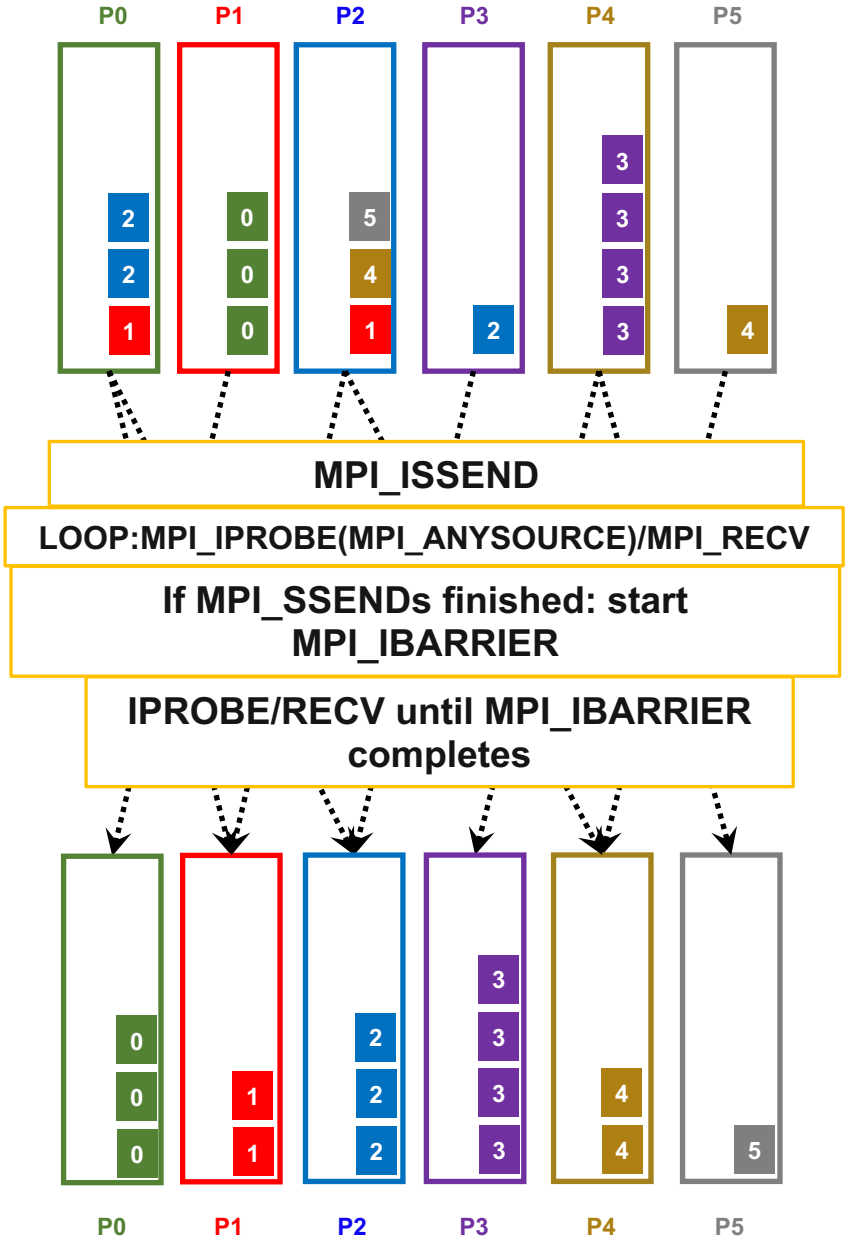


MPI_IBARRIER (NBX)

- Complexity - census (barrier):

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

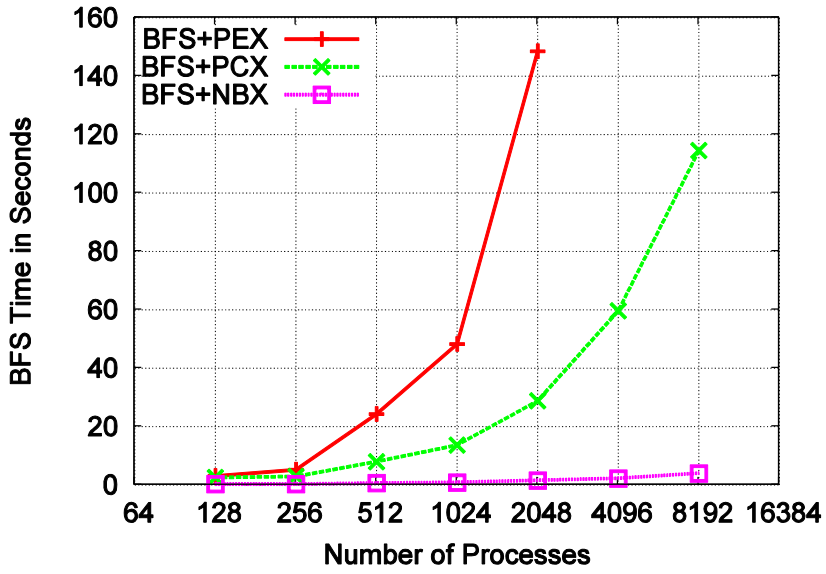
  - Combines metadata with actual transmission
  - Point-to-point synchronization
  - Continue receiving until barrier completes
  - Processes start collective synchronization (ibarrier) when p2p phase ended
    - barrier = distributed marker!
  - Better than Alltoall, reduce-scatter!



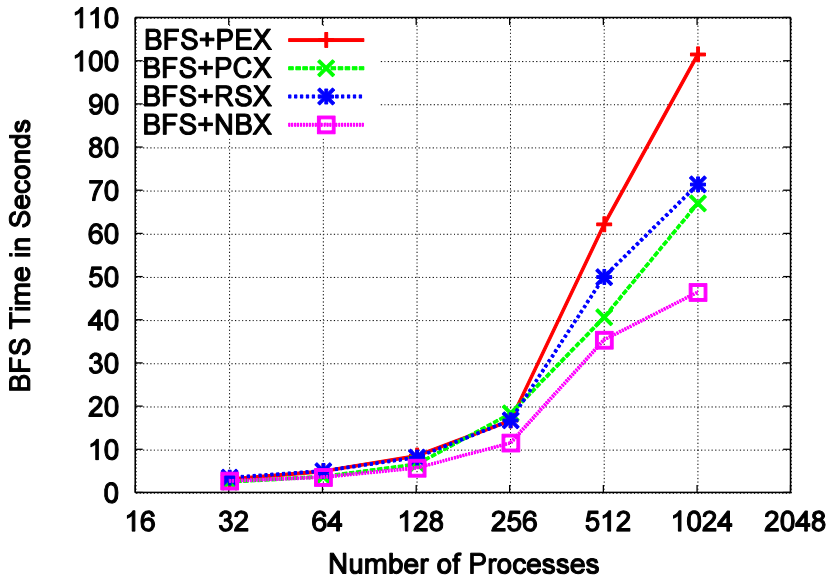
Parallel Breadth First Search

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

BlueGene/P – with HW barrier!



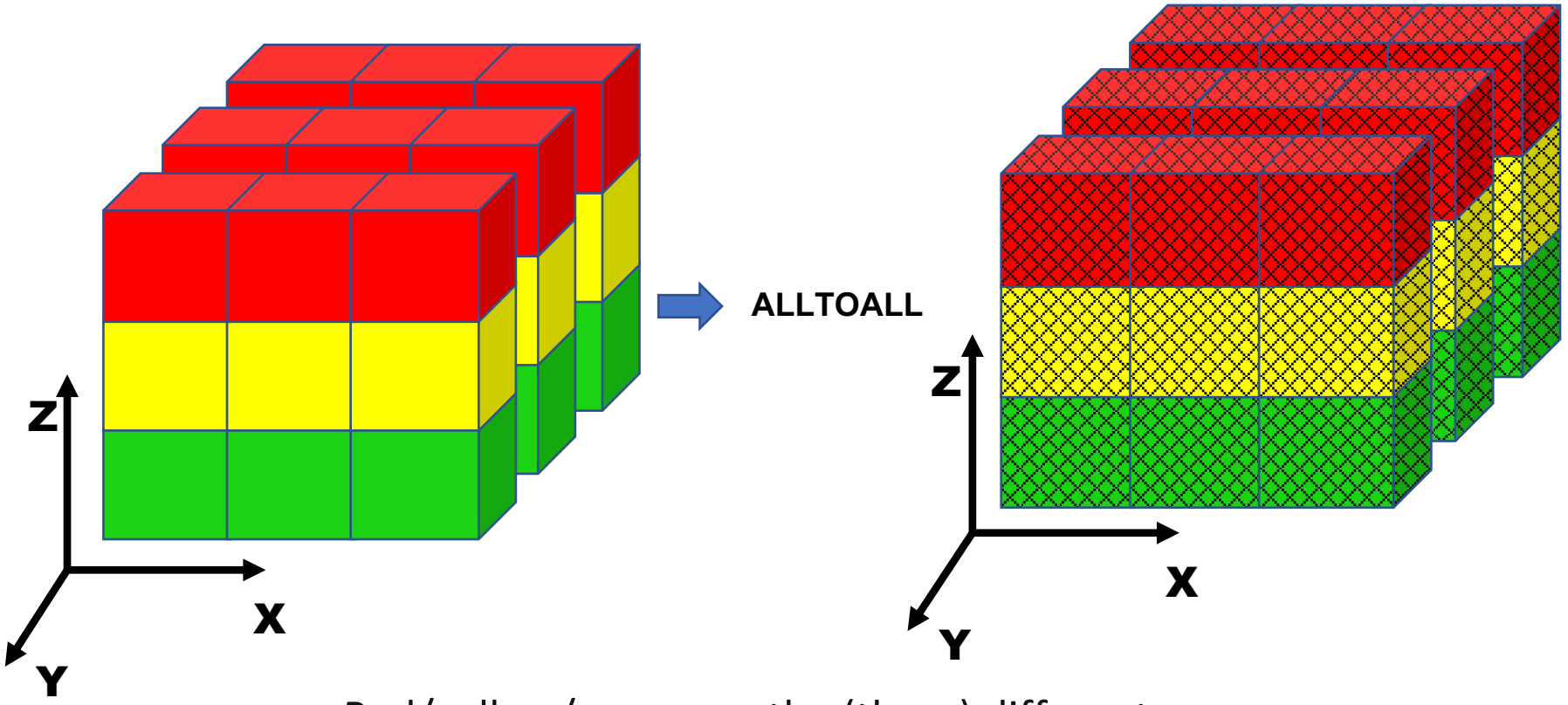
Myrinet 2000 with LibNBC



Impact of HW barrier is significant at large scale!

Parallel Fast Fourier Transform

- 1D FFTs in all three dimensions
  - Assume 1D decomposition (each process holds a set of planes)
  - Best way: call optimized 1D FFTs in parallel → alltoall



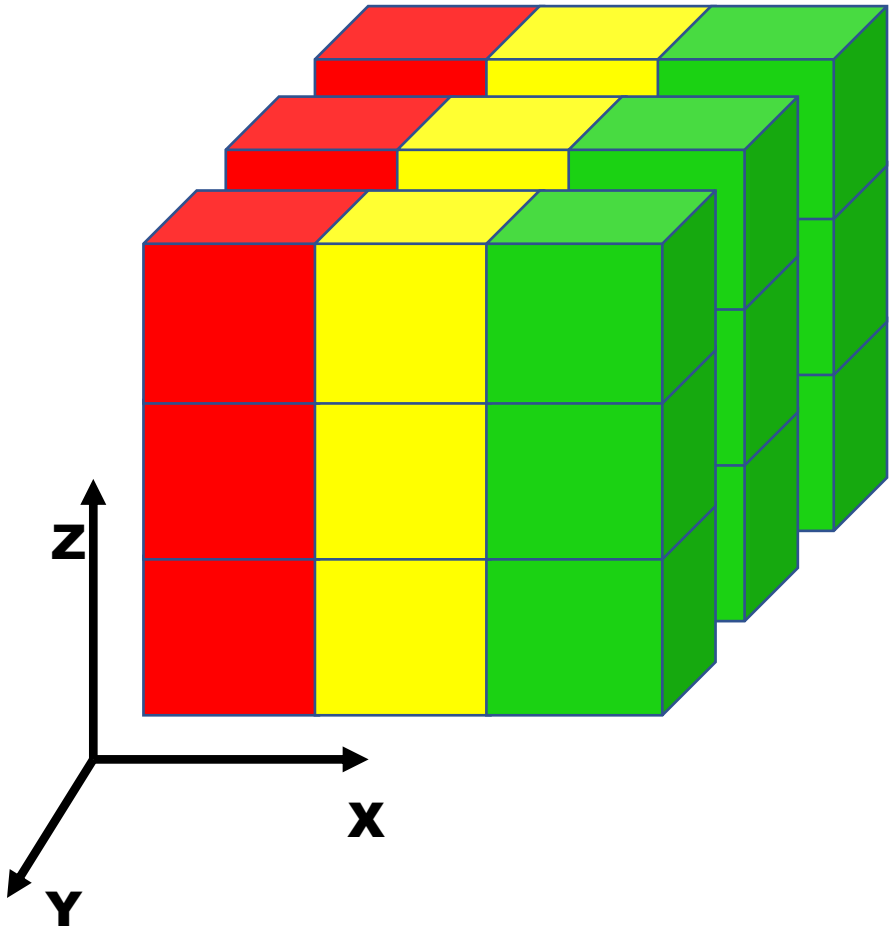
Red/yellow/green are the (three) different processes

A Complex Example: FFT

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

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


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

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

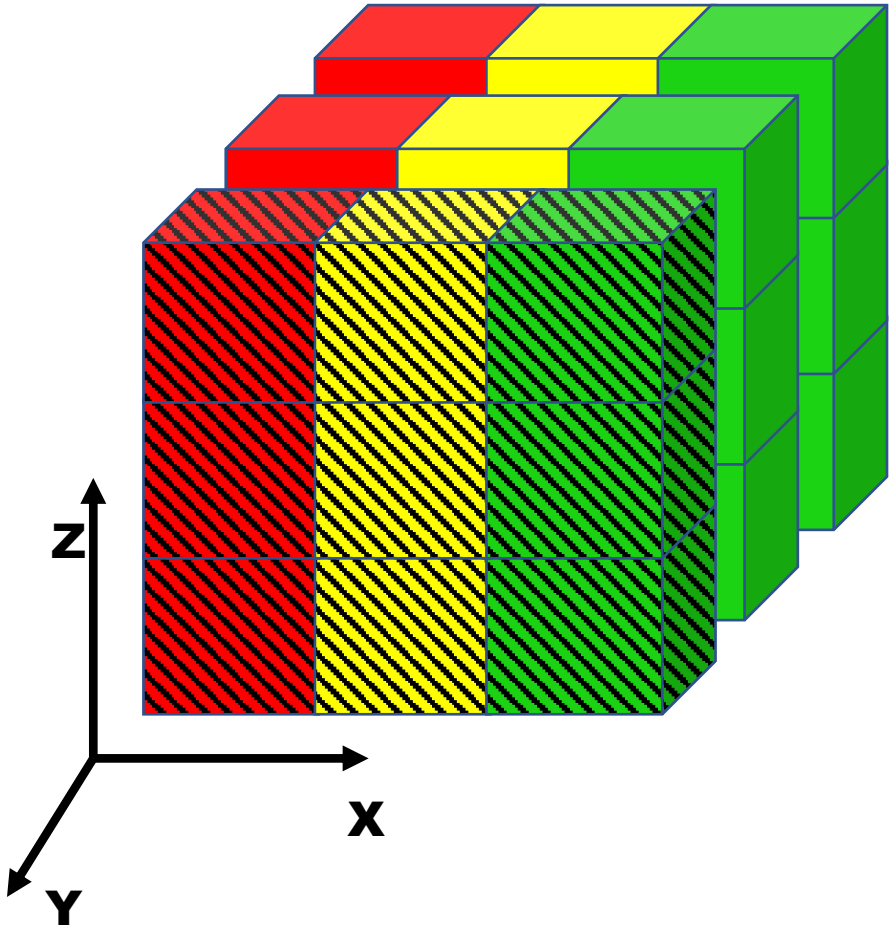
Parallel Fast Fourier Transform

- Data already transformed in y-direction



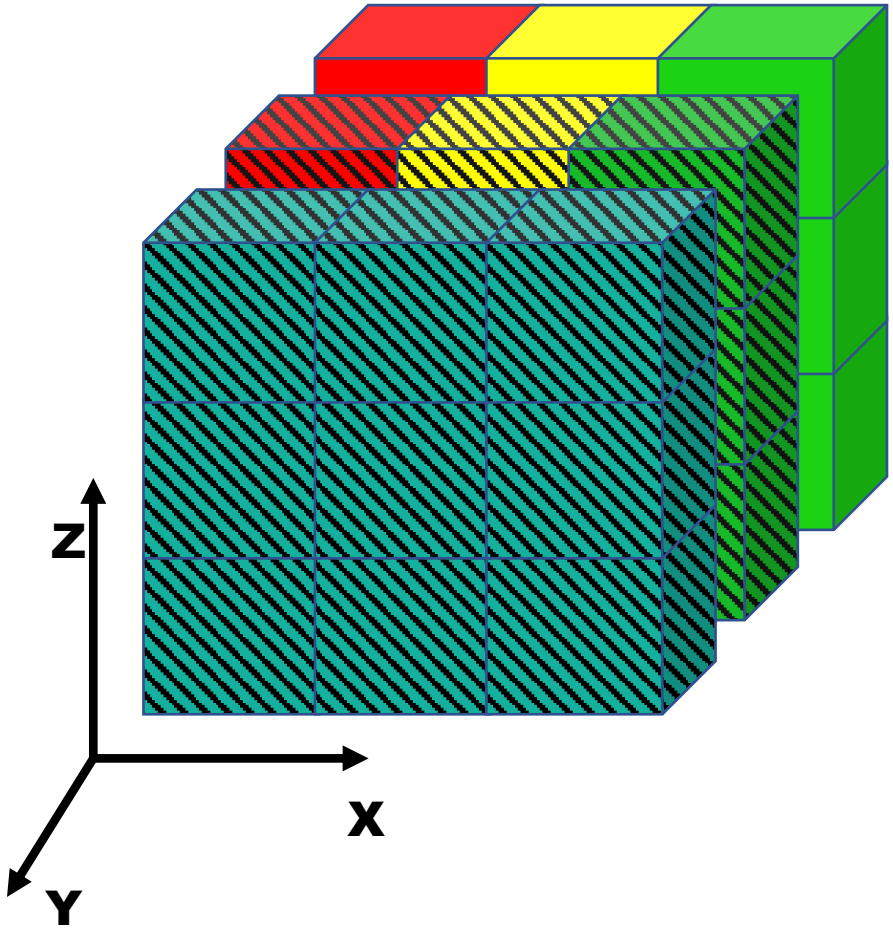
Parallel Fast Fourier Transform

- Transform first y plane in z



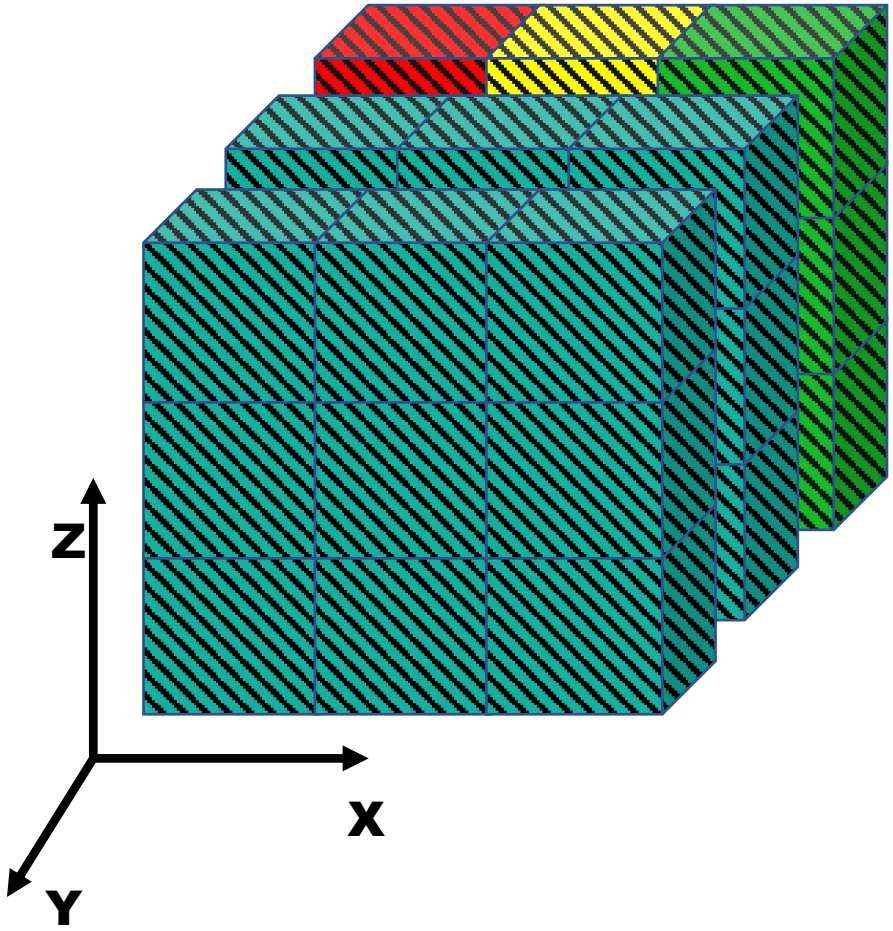
Parallel Fast Fourier Transform

- Start ialltoall and transform second plane



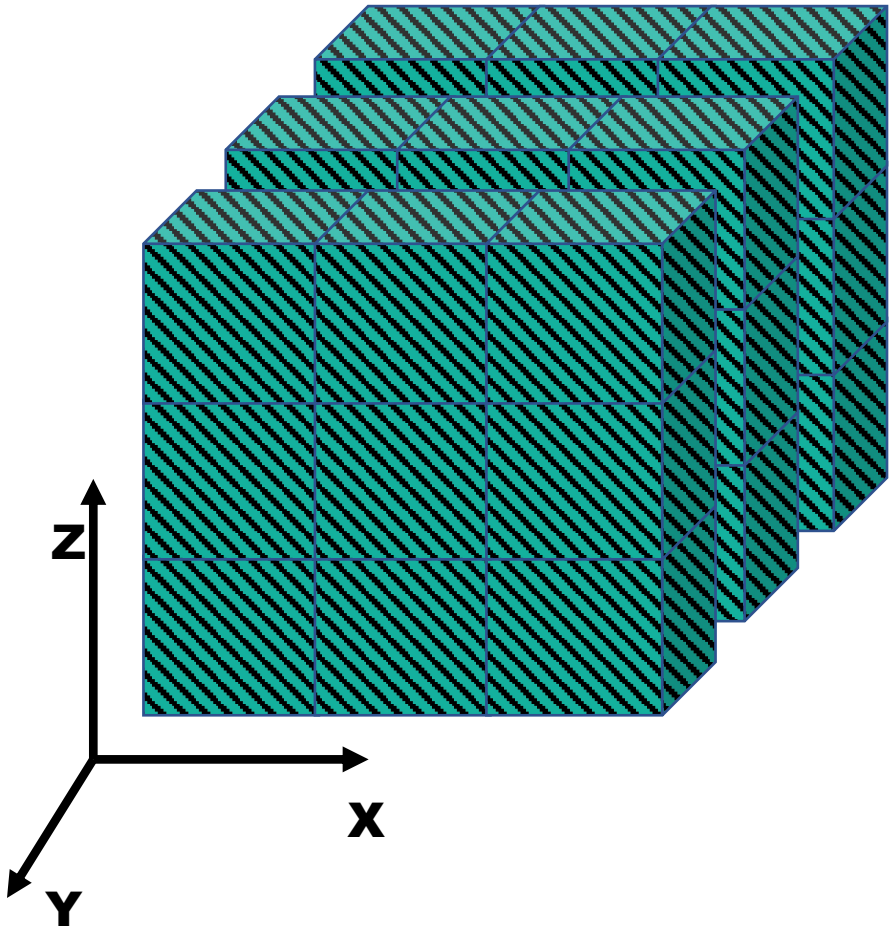
Parallel Fast Fourier Transform

- Start ialltoall (second plane) and transform third



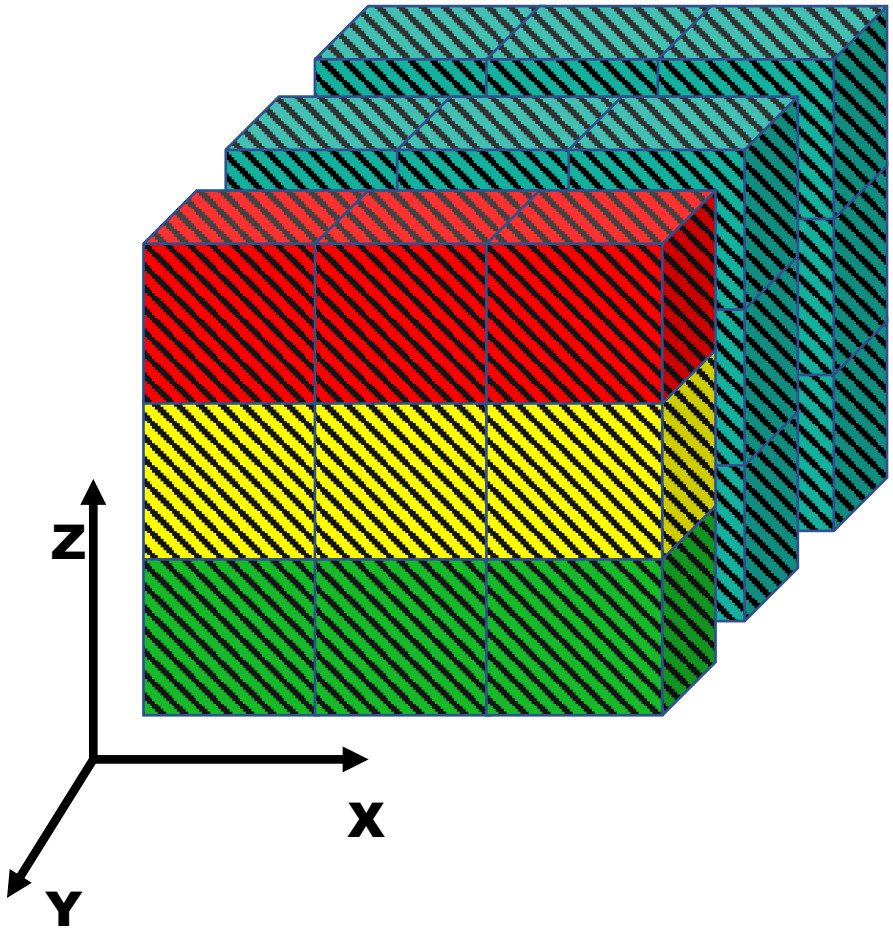
Parallel Fast Fourier Transform

- Start ialltoall of third plane and …



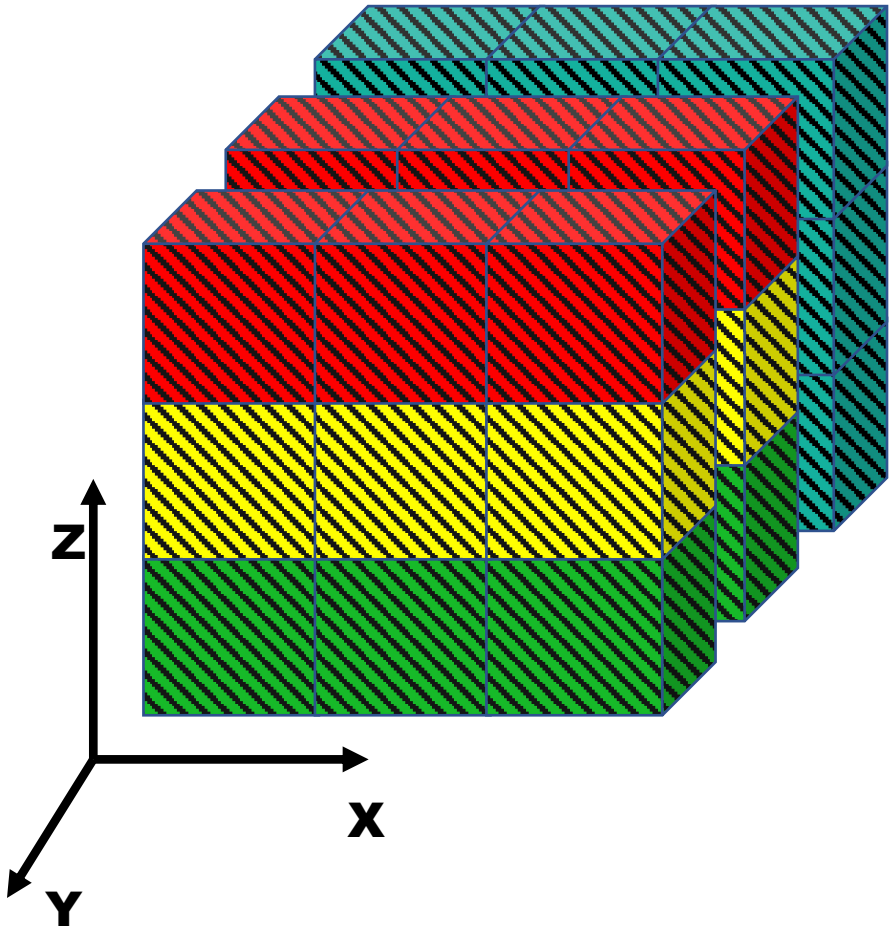
Parallel Fast Fourier Transform

- Finish ialltoall of first plane, start x transform



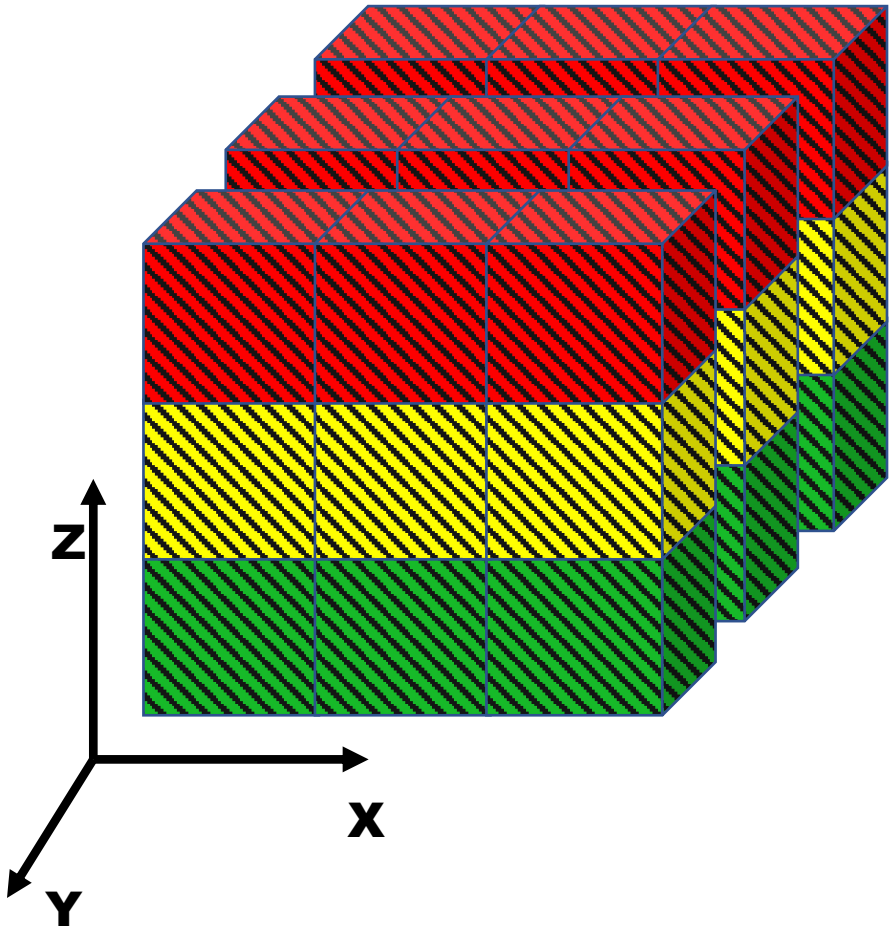
Parallel Fast Fourier Transform

- Finish second ialltoall, transform second plane



Parallel Fast Fourier Transform

- Transform last plane → done

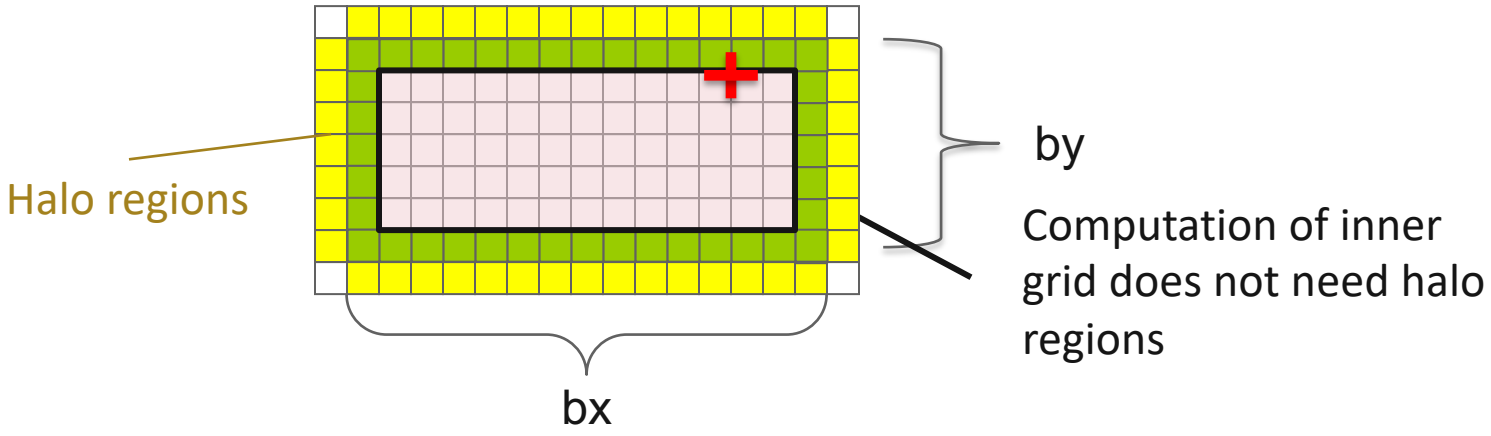


FFT Software Pipelining

```
MPI_Request req[nb];

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

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

MPI_Waitall(nb, req, MPI_STATUSES_IGNORE);

/* modified unpack data from alltoall and transpose */
for(int y=0; y<n/p; ++y) 1d_fft(/* y-th stencil */);
/* pack data for alltoall */

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

Exercise: Stencil with Nonblocking Collectives

- Use nonblocking collective to overlap computation and communication
  - Compute inner grid while waiting for completion of data movement
  - Compute outer grid with updated halo regions
- *Start from derived_datatype/stencil_alltoallw.c*
- *Solution available in nonblocking_coll/stencil_alltoallw.c*



Section Summary

- 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

Argonne NATIONAL LABORATORY

Virtual Topologies and Neighborhood Collectives

Slides Available at https://anl.box.com/v/yguo-isc-tutorial-2021

U.S. DEPARTMENT OF ENERGY

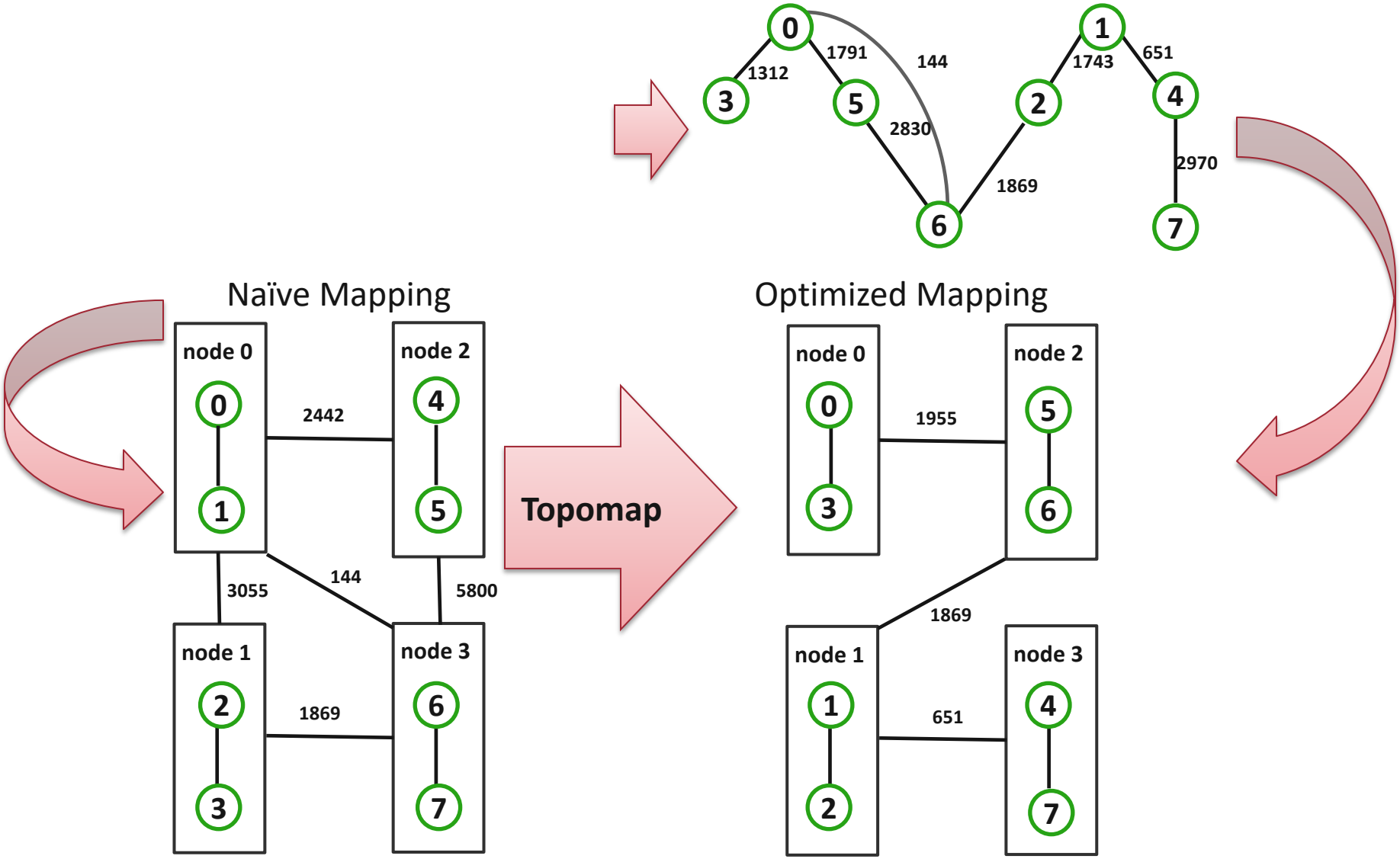
Topology Mapping and Neighborhood Collectives

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

Topology Mapping Basics

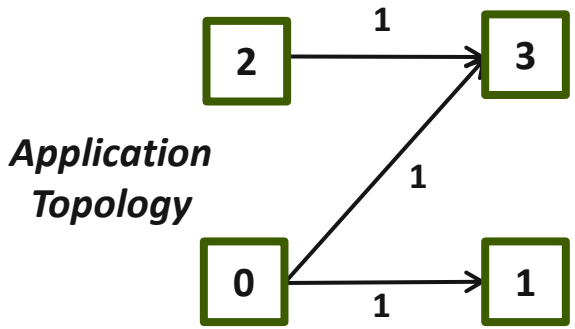
- MPI supports rank reordering
  - Change numbering in a given allocation to reduce congestion or dilation
  - Sometimes automatic (early IBM SP machines)
- Properties
  - Always possible, but effect may be limited (e.g., in a bad allocation)
  - Portable way: MPI process topologies
    - Network topology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

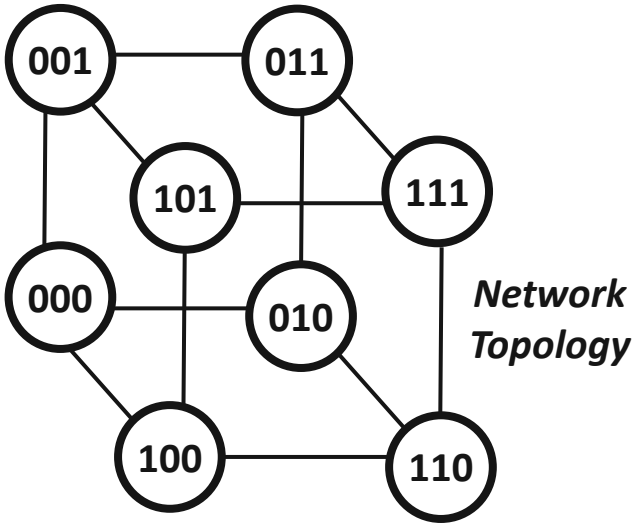
Example: On-Node Reordering

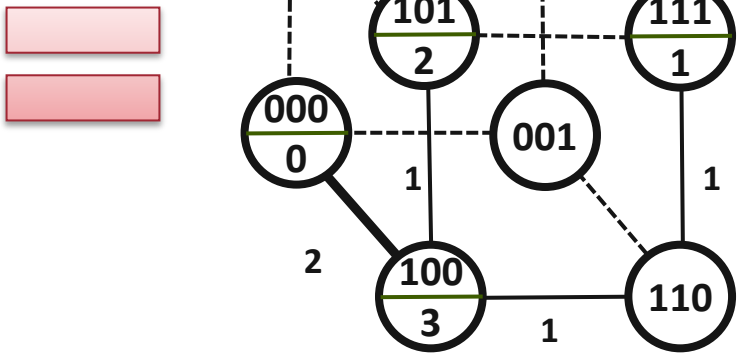


Gottschling et al.: Productive Parallel Linear Algebra Programming with Unstructured Topology Adaption

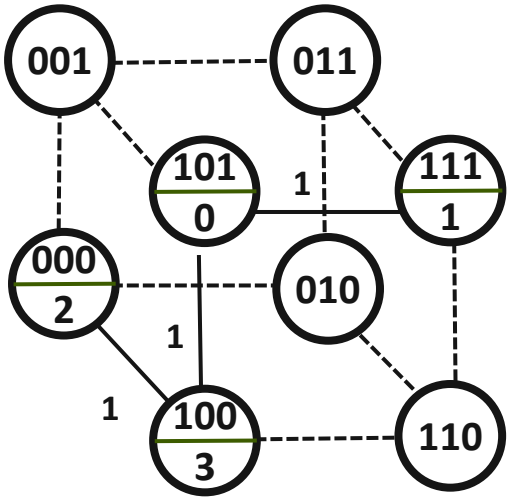
Off-Node (Network) Reordering







Topomap



MPI Topology Intro

- Convenience functions (in MPI-1)
  - Create a graph and query it, nothing else
  - Useful especially for Cartesian topologies
    - Query neighbors in n-dimensional space
  - Graph topology: each rank specifies full graph ☹
- Scalable Graph topology (MPI-2.2)
  - Graph topology: each rank specifies its neighbors **or** an arbitrary subset of the graph
- Neighborhood collectives (MPI-3.0)
  - Adding communication functions defined on graph topologies (neighborhood of distance one)

MPI_Cart_create

```
MPI_Cart_create(MPI_Comm comm_old, int ndims, const int *dims,
        const int *periods, int reorder, MPI_Comm *comm_cart)
```

- Specify ndims-dimensional topology
  - Optionally periodic in each dimension (Torus)
- Some processes may return MPI_COMM_NULL
  - Product sum of dims must be <= P
- Reorder argument allows for topology mapping
  - Each calling process may have a new rank in the created communicator
  - Data has to be remapped manually

MPI_Cart_create Example

```
int dims[3] = {5,5,5};
int periods[3] = {1,1,1};
MPI_Comm topocomm;
MPI_Cart_create(comm, 3, dims, periods, 0, &topocomm);
```

- Creates logical 3D Torus of size 5 x 5 x 5
- But we're starting MPI processes with a one-dimensional argument (-p X)
  - User has to determine size of each dimension
  - Often as "square" as possible, MPI can help!

MPI_Dims_create

```
MPI_Dims_create(int nnodes, int ndims, int *dims)
```

- Create dims array for Cart_create with nnodes and ndims
  - Dimensions are as close as possible (well, in theory)
- Non-zero entries in dims will not be changed
  - nnodes must be multiple of all non-zeroes

MPI_Dims_create Example

```
int p;
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Dims_create(p, 3, dims);

int periods[3] = {1,1,1};
MPI_Comm topocomm;
MPI_Cart_create(comm, 3, dims, periods, 0, &topocomm);
```

- Makes life a little bit easier
  - Some problems may be better with a non-square layout though

Cartesian Query Functions

- Library support and convenience!
- MPI_Cartdim_get()
  - Gets dimensions of a Cartesian communicator
- MPI_Cart_get()
  - Gets size of dimensions
- MPI_Cart_rank()
  - Translate coordinates to rank
- MPI_Cart_coords()
  - Translate rank to coordinates

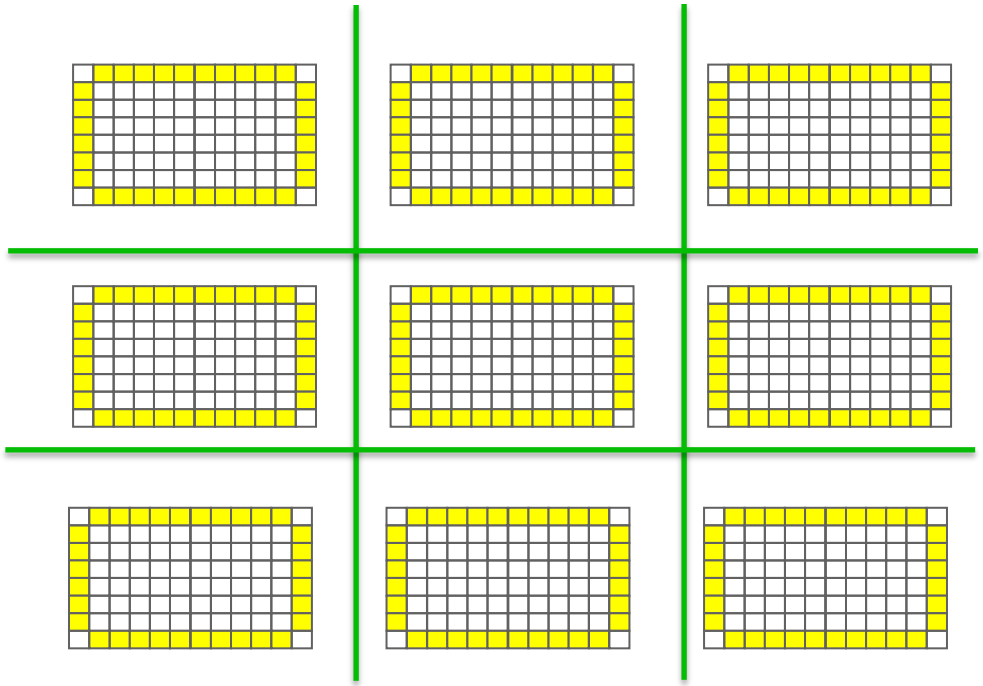
Cartesian Communication Helpers

```
MPI_Cart_shift(MPI_Comm comm, int direction, int disp,
               int *rank_source, int *rank_dest)
```

- Shift in one dimension
  - Dimensions are numbered from 0 to ndims-1
  - Displacement indicates neighbor distance (-1, 1, …)
  - May return MPI_PROC_NULL
- Very convenient, all you need for nearest neighbor communication
  - No “over the edge” though

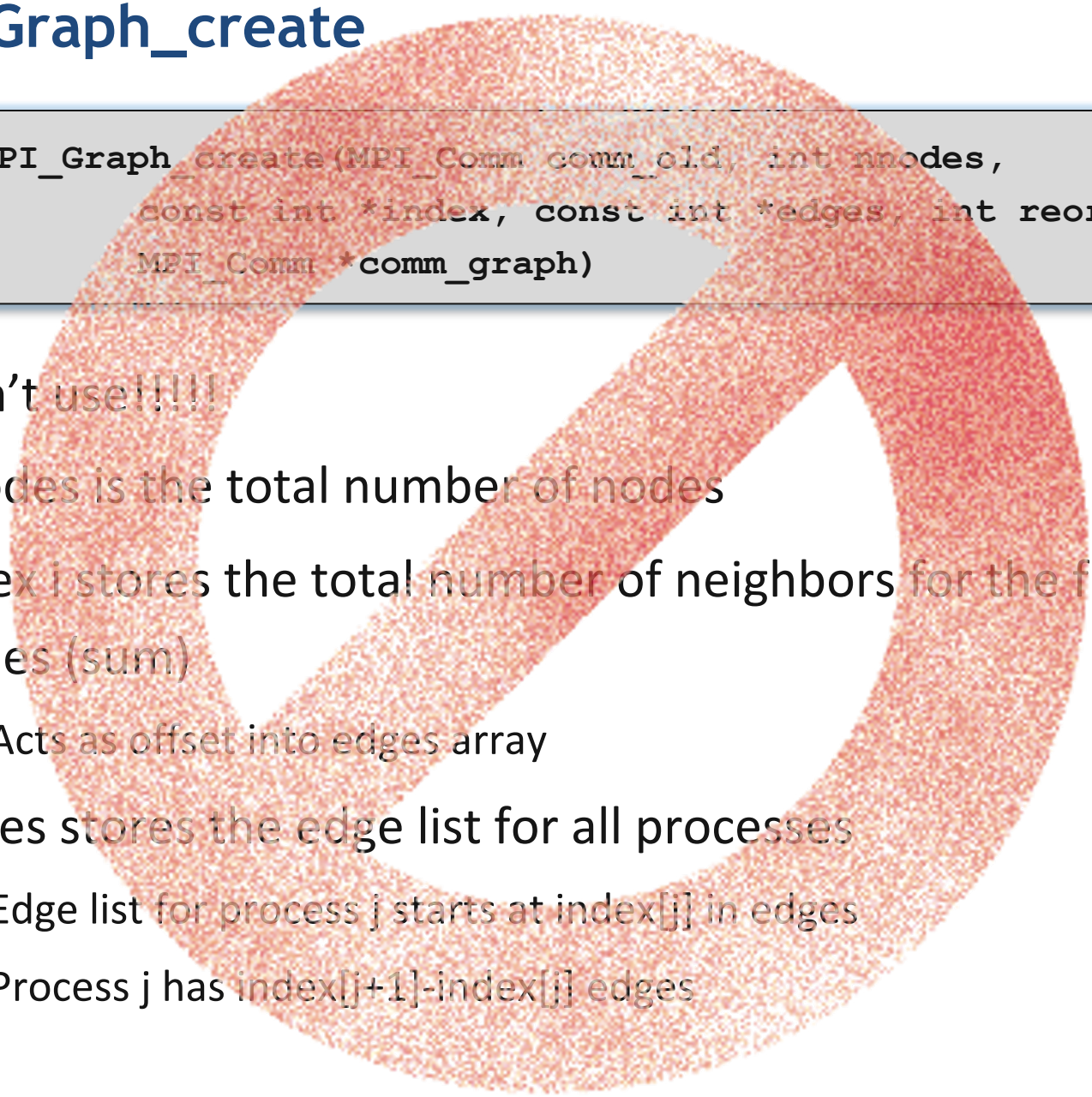
Example: Stencil with Cartesian Topology

- *topology/stencil_carttopo.c*
- Adds calculation of neighbors with topology



MPI_Graph_create

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



- Don't use!!!!!
- nnodes is the total number of nodes
- index i stores the total number of neighbors for the first i nodes (sum)
  - Acts as offset into edges array
- edges stores the edge list for all processes
  - Edge list for process j starts at index[j] in edges
  - Process j has index[j+1]-index[j] edges

Distributed graph constructor

- MPI_Graph_create is discouraged
  - Not scalable
  - Not deprecated yet but hopefully soon
- New distributed interface:
  - Scalable, allows distributed graph specification
    - Either local neighbors **or** any edge in the graph
  - Specify edge weights
    - Meaning undefined but optimization opportunity for vendors!
  - Info arguments
    - Communicate assertions of semantics to the MPI library
    - E.g., semantics of edge weights

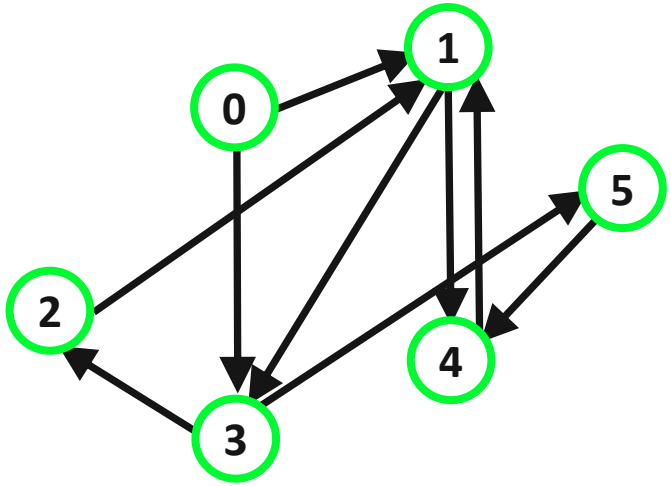
MPI_Dist_graph_create_adjacent

```
MPI_Dist_graph_create_adjacent(MPI_Comm comm_old,
        int indegree, const int sources[], const int sourceweights[],
        int outdegree, const int destinations[],
        const int destweights[], MPI_Info info, int reorder,
        MPI_Comm *comm_dist_graph)
```

- indegree, sources, ~weights – source proc. Spec.
- outdegree, destinations, ~weights – dest. proc. spec.
- info, reorder, comm_dist_graph – as usual
- directed graph
- Each edge is specified twice, once as out-edge (at the source) and once as in-edge (at the dest)

MPI_Dist_graph_create_adjacent

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



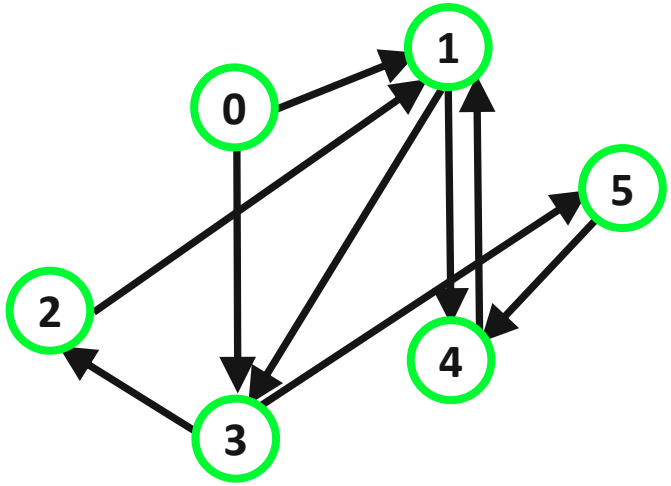
MPI_Dist_graph_create

```
MPI_Dist_graph_create(MPI_Comm comm_old, int n,
        const int sources[], const int degrees[],
        const int destinations[], const int weights[], MPI_Info info,
        int reorder, MPI_Comm *comm_dist_graph)
```

- n – number of source nodes
- sources – n source nodes
- degrees – number of edges for each source
- destinations, weights – dest. processor specification
- info, reorder – as usual
- More flexible and convenient
  - Requires global communication
  - Slightly more expensive than adjacent specification

MPI_Dist_graph_create

- Process 0:
  - N: 2
  - Sources: {0,1}
  - Degrees: {2,1} *
  - Dests: {3,1,4}
- Process 1:
  - N: 2
  - Sources: {2,3}
  - Degrees: {1,1}
  - Dests: {1,2}
- …



* Note that in this example, process 0 specifies only one of the two outgoing edges of process 1; the second outgoing edge needs to be specified by another process

Distributed Graph Neighbor Queries

```
MPI_Dist_graph_neighbors_count(MPI_Comm comm,
        int *indegree,int *outdegree, int *weighted)
```

- Query the number of neighbors of **calling process**
- Returns indegree and outdegree!
- Also info if weighted

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

- Query the neighbor list of **calling process**
- Optionally return weights

Further Graph Queries

```
MPI_Topo_test(MPI_Comm comm, int *status)
```

- Status is either:
  - MPI_GRAPH (ugs)
  - MPI_CART
  - MPI_DIST_GRAPH
  - MPI_UNDEFINED (no topology)
- Enables us to write libraries on top of MPI topologies!

Neighborhood Collectives

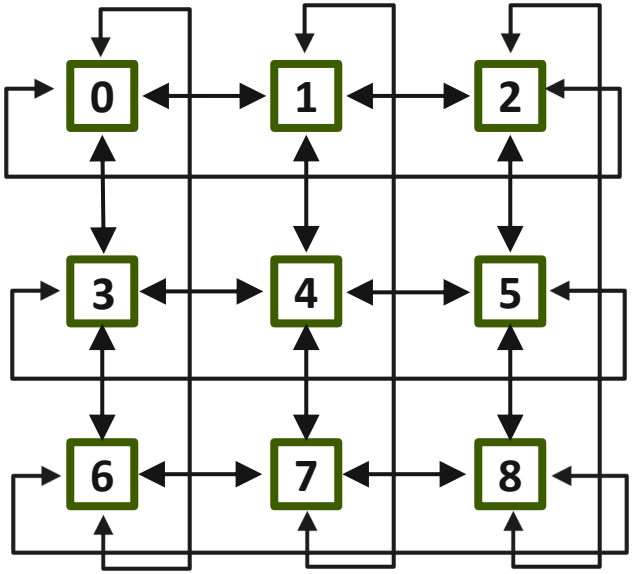
- Topologies implement no communication!
  - Just helper functions
- Collective communications only cover some patterns
  - E.g., no stencil pattern
- Several requests for “build your own collective” functionality in MPI
  - Neighborhood collectives are a simplified version
  - Cf. Datatypes for communication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

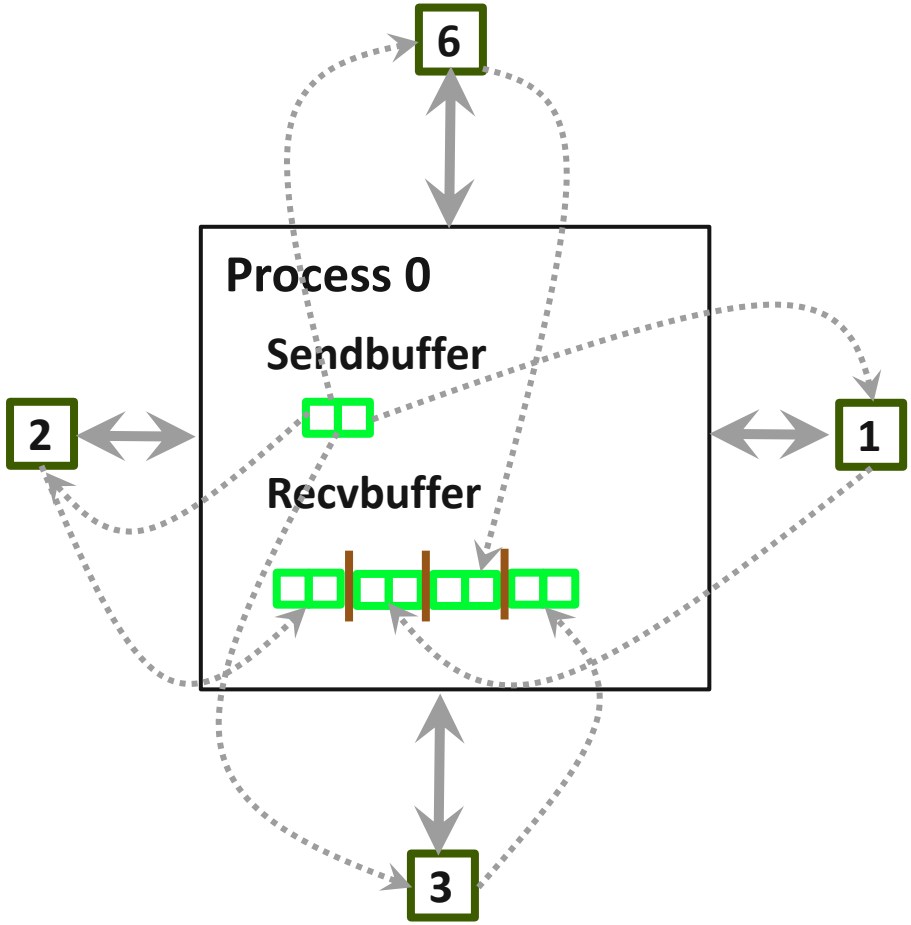
Cartesian Neighborhood Collectives

- Communicate with direct neighbors in Cartesian topology
  - Corresponds to cart_shift with disp=1
  - Collective (all processes in comm must call it, including processes without neighbors)
  - Buffers are laid out as neighbor sequence:
    - Defined by order of dimensions, first negative, then positive
    - 2*ndims sources and destinations
    - Processes at borders (MPI_PROC_NULL) leave holes in buffers (will not be updated or communicated)!

Cartesian Neighborhood Collectives

- Buffer ordering example:





Graph Neighborhood Collectives

- Collective Communication along arbitrary neighborhoods
  - Order is determined by order of neighbors as returned by (dist_)graph_neighbors.
  - Distributed graph is directed, may have different numbers of send/recv neighbors
  - Can express dense collective operations
  - Any persistent communication pattern!

MPI_Neighbor_allgather

```
MPI_Neighbor_allgather(const void* sendbuf, int sendcount,
        MPI_Datatype sendtype, void* recvbuf, int recvcount,
        MPI_Datatype recvtype, MPI_Comm comm)
```

- Sends the same message to all neighbors
- Receives indegree distinct messages
- Similar to MPI_Gather
  - The all prefix expresses that each process is a “root” of his neighborhood
- Vector version for full flexibility

MPI_Neighbor_alltoall

```
MPI_Neighbor_alltoall(const void* sendbuf, int sendcount,
        MPI_Datatype sendtype, void* recvbuf, int recvcount,
        MPI_Datatype recvtype, MPI_Comm comm)
```

- Sends outdegree distinct messages
- Received indegree distinct messages
- Similar to MPI_Alltoall
  - Neighborhood specifies full communication relationship
- Vector and w versions for full flexibility

Nonblocking Neighborhood Collectives

```
MPI_Ineighbor_allgather(…, MPI_Request *req);
MPI_Ineighbor_alltoall(…, MPI_Request *req);
```

- Very similar to nonblocking collectives
- Collective invocation
- Matching in-order (no tags)
  - No wild tricks with neighborhoods! In order matching per communicator!

Exercise: Stencil with Neighborhood Collectives

- Adds neighborhood collectives to the topology
- *Start from topology/stencil_carttopo.c and derived_datatype/stencil_alltoallw.c*
- *Solution available in topology/stencil_carttopo_neighcolls.c*

Why is Neighborhood Reduce Missing?

```
MPI_Ineighbor_allreducev(…, MPI_Request *req);
MPI_Neighbor_allreducev(…);
```

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

Section Summary

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

Argonne NATIONAL LABORATORY

MPI-4 and Future MPI Standards

Slides Available at https://anl.box.com/v/yguo-isc-tutorial-2021

U.S. DEPARTMENT OF ENERGY

Introduction

- The MPI Forum continues to meet every 3 months to define future versions of the MPI Standard
- We describe some of the proposals the Forum is currently considering
- None of these topics are guaranteed to be in MPI-4
  - These are simply proposals that are being considered

MPI Working Groups

- Point-to-point communication
- Fault tolerance
- Hybrid programming
- Persistence
- Tools interfaces
- Large counts
- Others: RMA, Collectives, Fortran, Topologies, Sessions

- https://www.mpi-forum.org/mpi-40/

Point-to-Point Working Group

Proposal 1: Batched Communication Operations

- MPI-3.1 semantics
  - Each point-to-point operation creates a new request object
  - MPI library might run out of request objects after a few thousand operations
  - Application cannot issue a lot of messages to fully utilize the network
- Batched operations
  - RMA-like semantics for MPI send/recv communication
    - Application frees request as soon as the operation is issued
    - Batch completion of all operations on a communicator
      - MPI_COMM_WAITALL
  - Proportionally reduced number of requests
  - Can allow applications to consolidate multiple completions into a single request

Proposal 2: 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

Fault Tolerance Working Group

Improved Support for Fault Tolerance

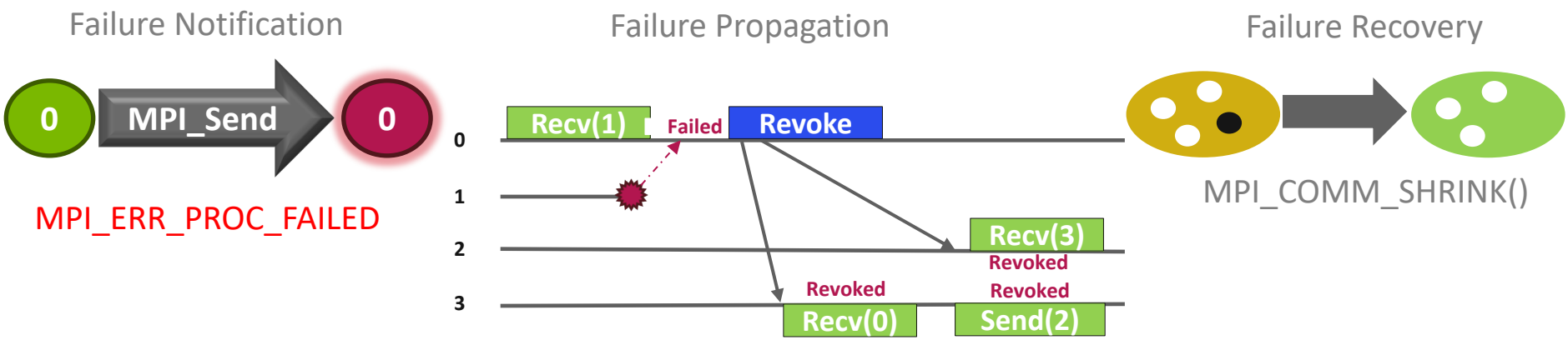
- MPI always had support for error handlers and allows implementations to return an error code and remain alive
- MPI Forum working on additional support for MPI-4
- Current proposal handles fail-stop process failures (not silent data corruption or Byzantine failures)
  - If a communication operation fails because the other process has failed, the function returns error code MPI_ERR_PROC_FAILED
  - User can call MPI_Comm_shrink to create a new communicator that excludes failed processes
  - Collective communication can be performed on the new communicator

Proposal 1: 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

Proposal 2: User Level Failure Mitigation

- Enable application-level recovery by providing minimal FT API to prevent deadlock and enable recovery
- Don’t do recovery for the application, but let the application (or a library) do what is best
- Currently focused on process failure (not data errors or protection)



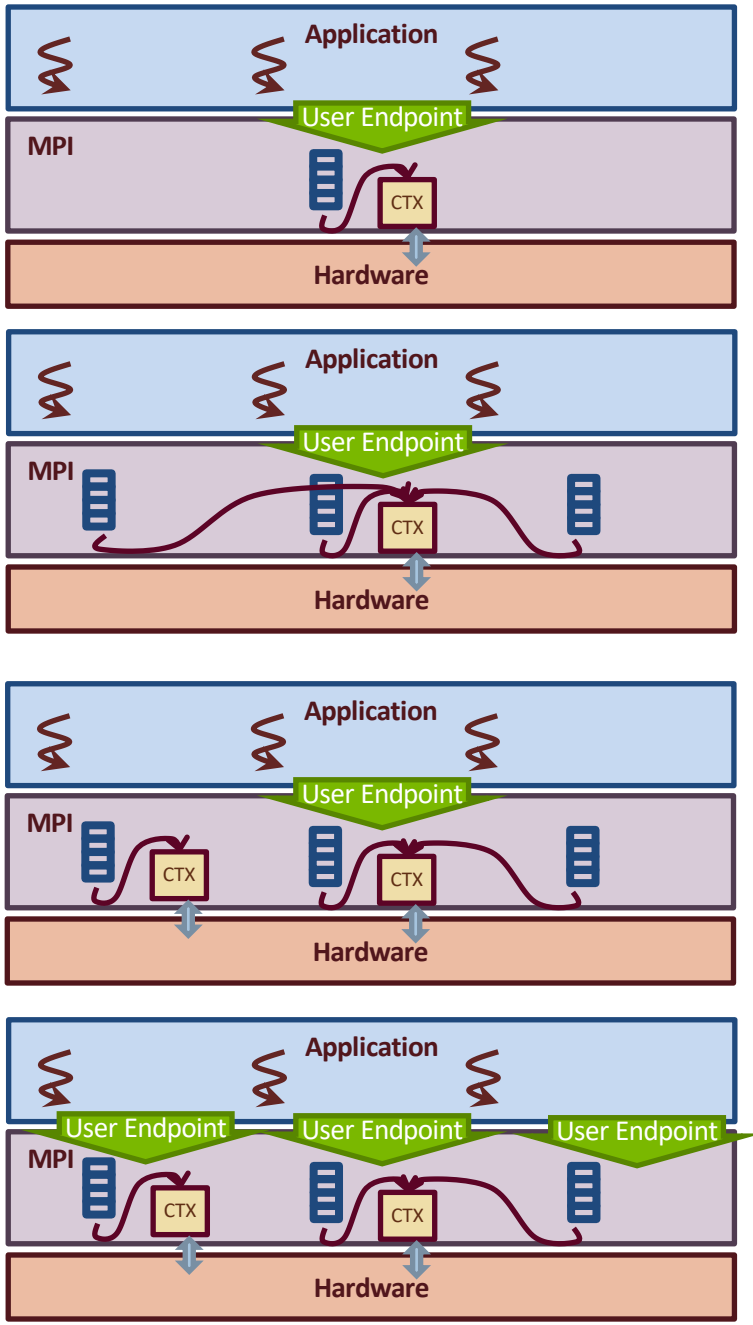
Hybrid Programming Working Group

MPI endpoints

- Idea is to have multiple addressable communication entities within a single MPI process
  - Instantiated in the form of multiple ranks per MPI process
- Each rank can be associated with one or more threads
- Reduced contention for communication on each “rank”
- In the extreme case, we could have one rank per thread (or some ranks might be used by a single thread)

Implementation phases/options

- Most common current approach
  - Single endpoint per MPI process
  - Worst case contention
- Possible optimization in MPI-3.1: multiple invisible endpoints
  - Multiple internal endpoints (BG/Q style)
  - Transparent to the user
  - E.g. one endpoint per comm, per neighbor process (regular apps)
- Endpoints proposal for MPI-4: multiple user-visible endpoints
  - Multiple endpoints managed by the user



Persistence Working Group
(Slides courtesy of Tony Skjellum)

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

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, …);
}
```

RMA Working Group

MPI Generalized Atomics

- MPI-3 atomic operations are, in some cases, restrictive and are not precisely defined
- Two proposals:
  - Clarify what operations are atomic and what are not (minor change)
  - Allow for generality of atomic operations with room for performance optimization
- Generality: Ability for different atomic operations to be issued on the same target location
- Performance: Additional info hints to restrict what the user will use (e.g., only CAS, only FOP, only basic datatypes)

https://github.com/mpi-forum/mpi-forum-historic/issues/416

Neighborhood Communication in MPI RMA

- MPI-3 defined neighborhood collectives where a process only communicates with its neighbors
- Neighborhood RMA is a generalization of that concept to allow RMA to neighboring processes
  - Allows MPI implementations to optimize state that is internally managed
  - Primarily an optimization for memory usage (e.g., MPI does not need to store information about non-neighbor processes)
    - Can also improve performance in some rare cases

Nonblocking RMA Synchronization

- RMA communication operations are nonblocking
- Some RMA synchronization operations are blocking
  - E.g., MPI_WIN_FENCE after issuing several PUT/GET operations
- Interferes with event-driven applications that want to process completion events as they occur
  - E.g., MPI_Waitany(…) followed by a handler to process whichever request completed
  - Can be done with threads where a thread blocks on call and then sends a “notification” message to unblock the MPI_Waitany
    - Cumbersome and requires a different thread for each simultaneously blocking operation
- Proposal: Nonblocking variants of synchronization operations

RMA Notification

- In passive target mode, notifying the target that data has been transmitted is currently inefficient
- Two proposals for target notification:
  - Notification on PUT/GET
  - Notification on Flush
- Idea is to notify the target when the data has been deposited into the target public memory

https://github.com/mpi-forum/mpi-issues/issues/59

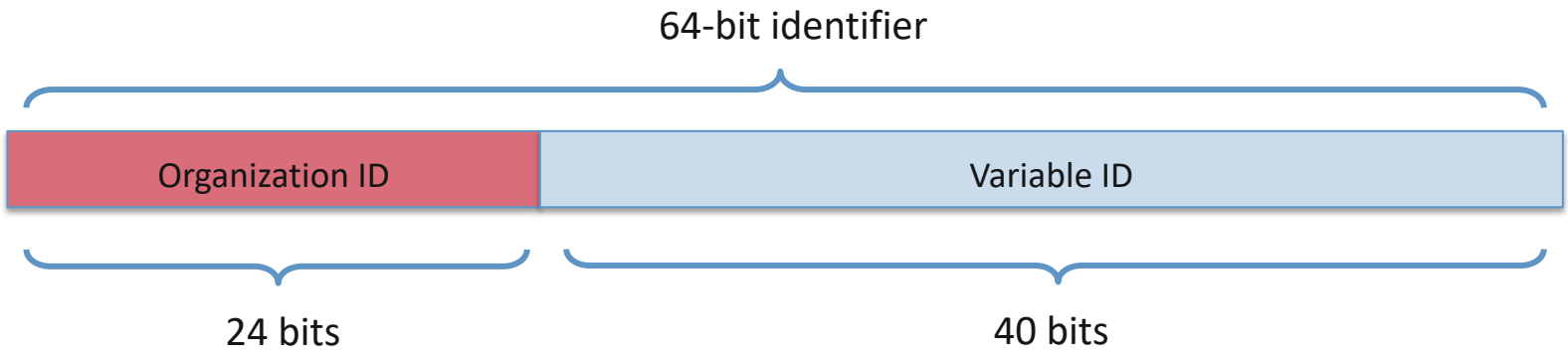
Tools Working Group
(slides courtesy Kathryn Mohror, LLNL)

What's happening now?

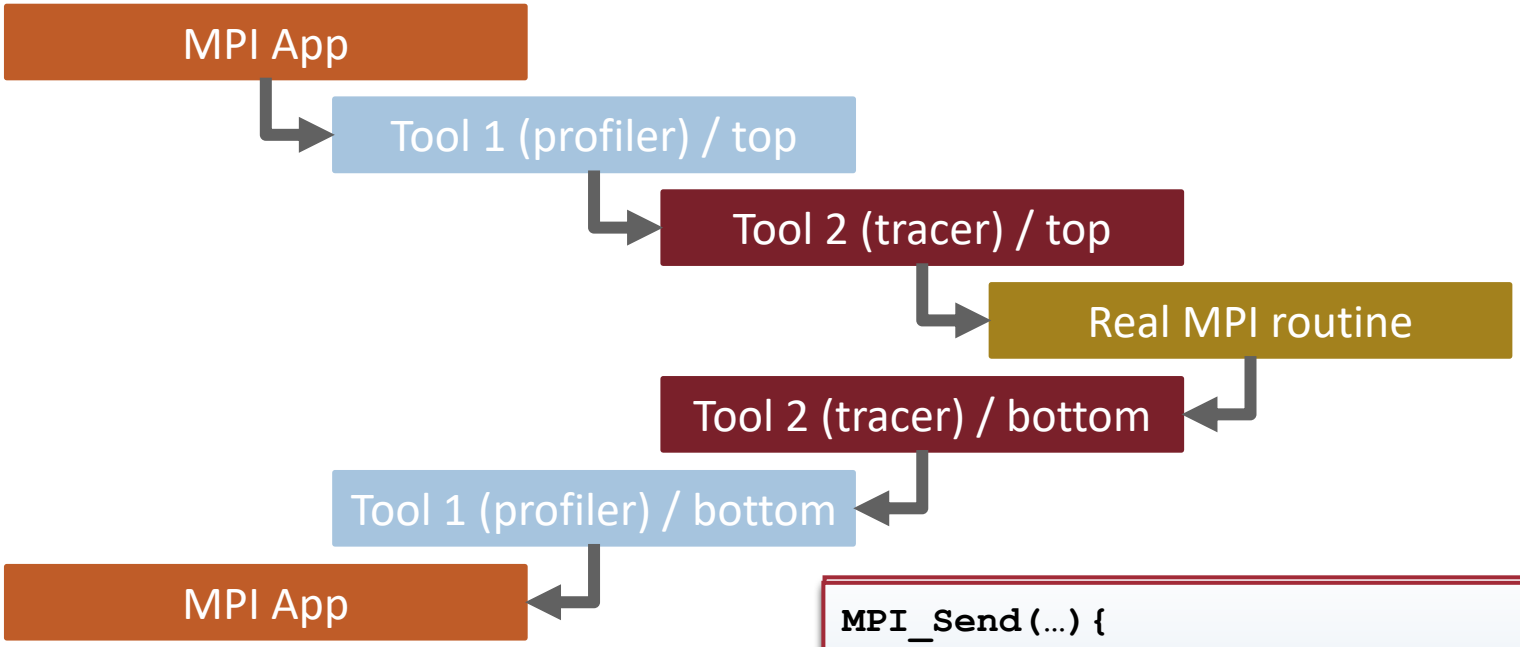
- **Variables**
  - MPI_T Variable Registration
- **QMPI**
  - A new interception interface for MPI
- **Events**
  - Get event notification from MPI

MPI_T Variable registration

- MPI implementations are free to provide whatever variables make sense for their implementation
  - Variables are allowed to change between versions of the library and across hardware
  - Want to provide some stability for tools and keep the freedom for implementations
- Organization IDs and variable identifiers registered with MPI Forum
  - API for retrieving runtime variable ID to permanent registered ID
  - Now tool has a stable ID for understanding the meaning of MPI_T variables

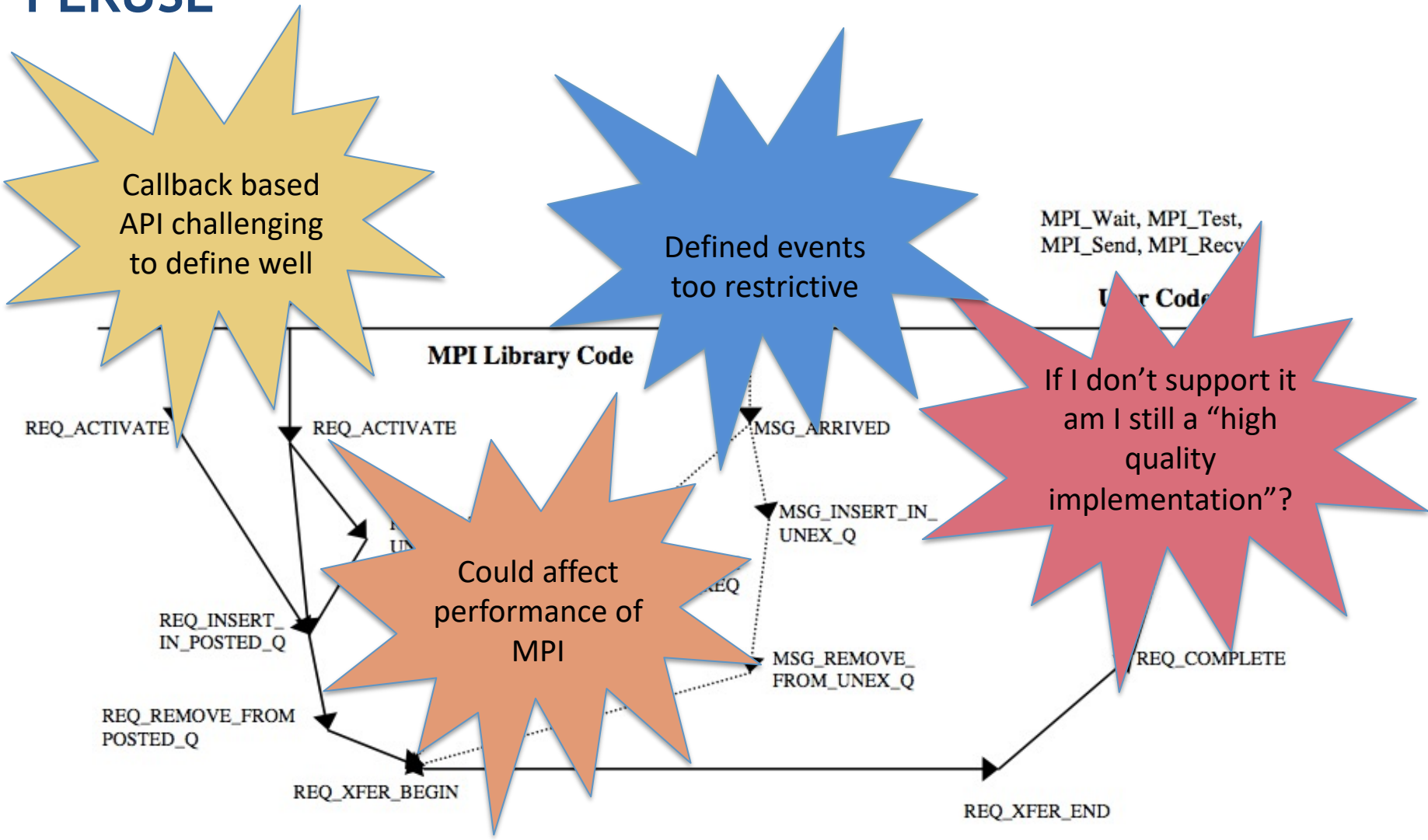


QMPI: New interface for layered tools




```
MPI_Send(…){
  ++sends;
  pmpi_next_send = get_fn_ptr();
  ret = pmpi_next_send(…);
  return ret;
}
```

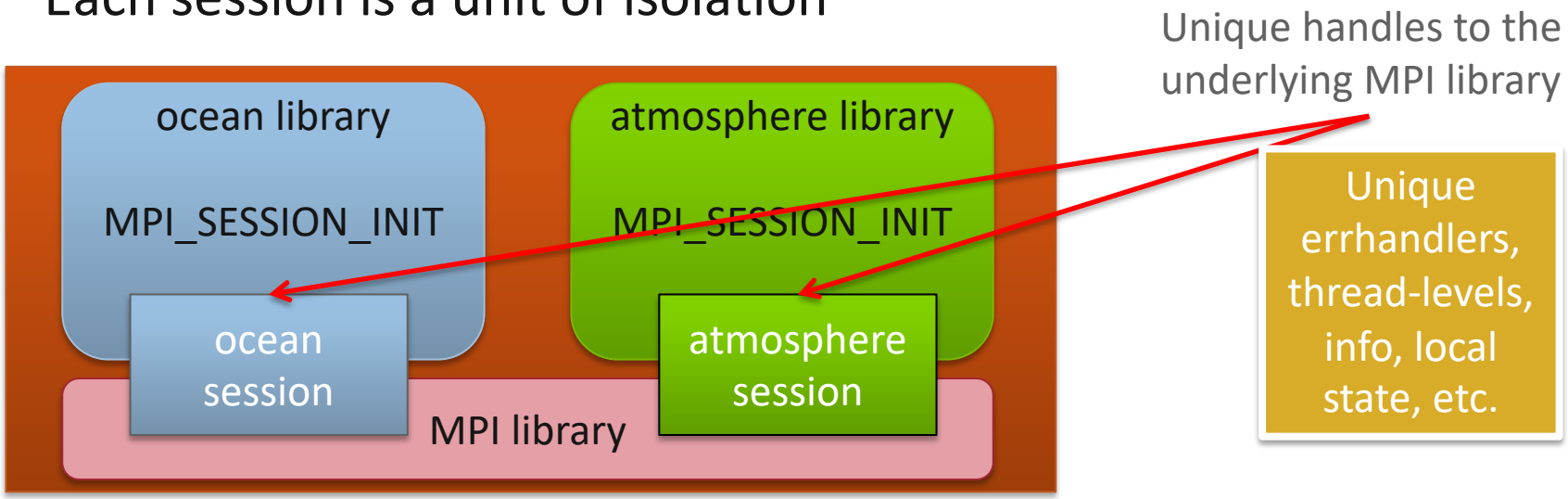
The inspiration for MPI_T_events comes from PERUSE



Sessions Working Group

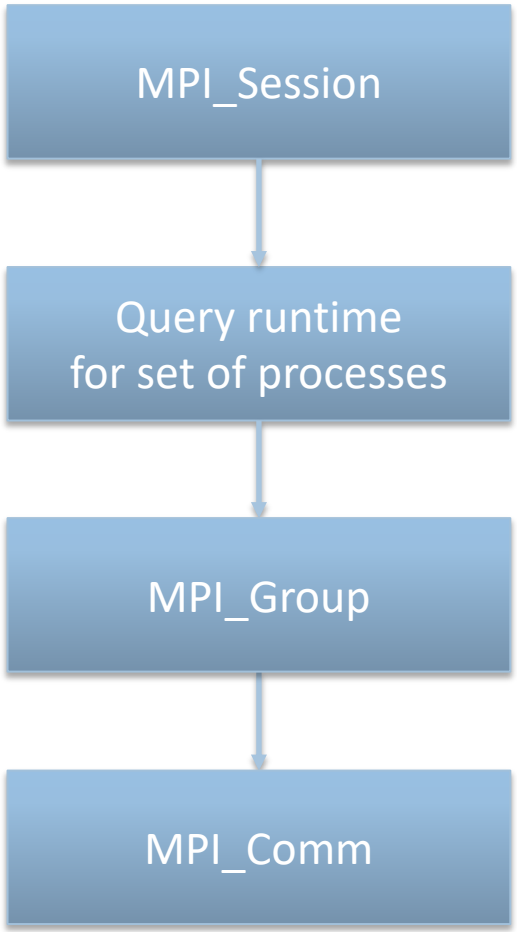
New Concept: "Session"

- 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



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



Concluding Remarks

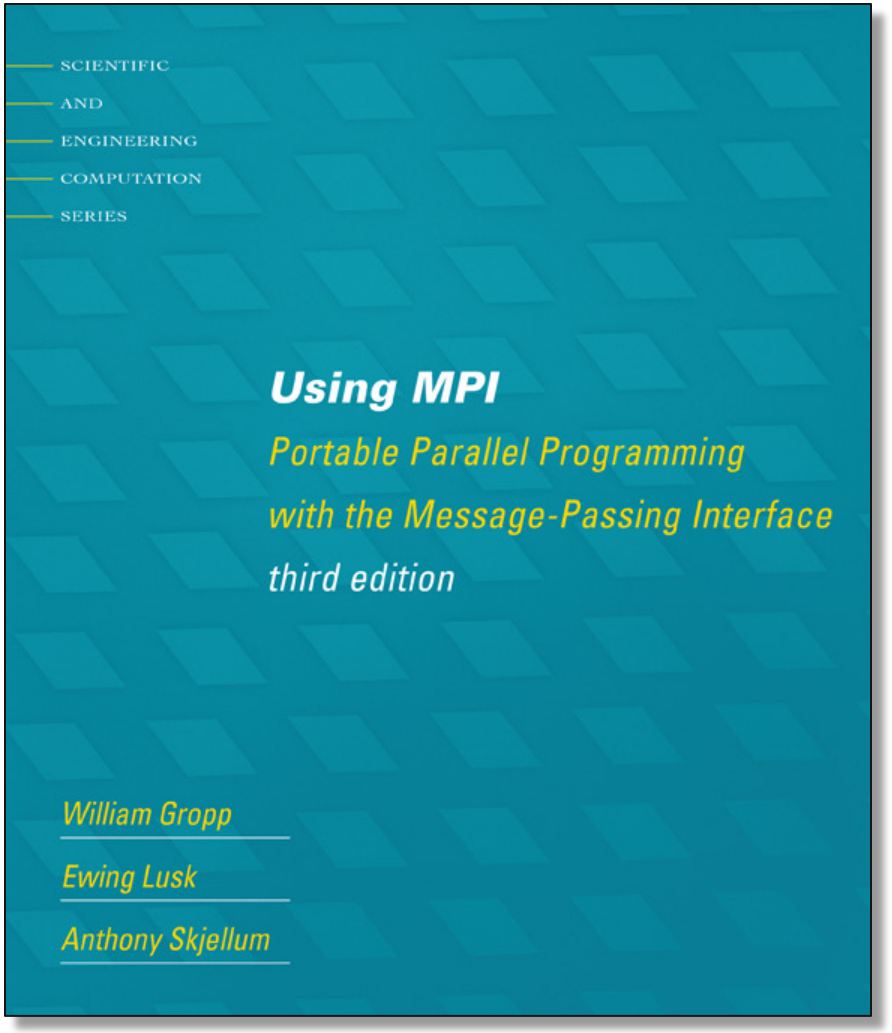
- Parallelism is critical today, given that that is the only way to achieve performance improvement with the modern hardware
- MPI is an industry standard model for parallel programming
  - A large number of implementations of MPI exist (both commercial and public domain)
  - Virtually every system in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

Web Pointers

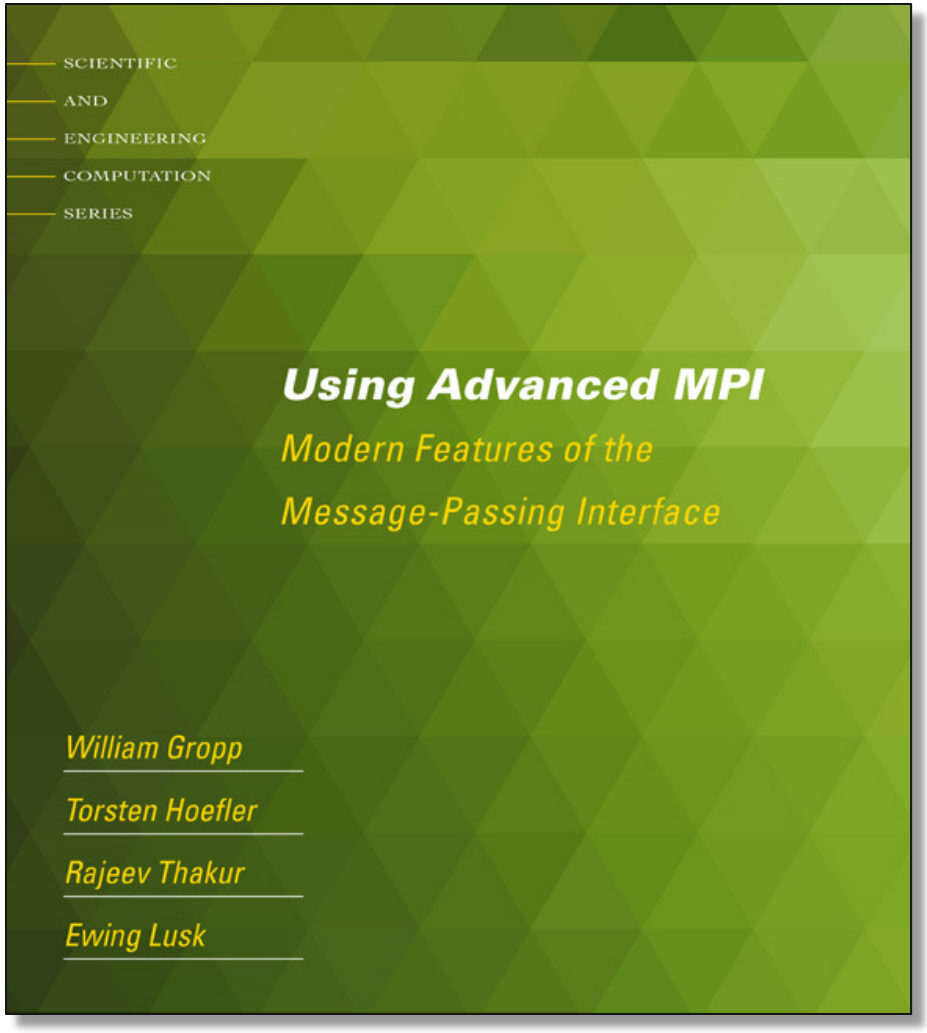
- MPI standard : http://www.mpi-forum.org/docs/docs.html
- MPI Forum : http://www.mpi-forum.org/

- MPI implementations:
  - MPICH : http://www.mpich.org
  - MVAPICH : http://mvapich.cse.ohio-state.edu/
  - Intel MPI: http://software.intel.com/en-us/intel-mpi-library/
  - Microsoft MPI: www.microsoft.com/en-us/download/details.aspx?id=39961
  - 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



Basic MPI

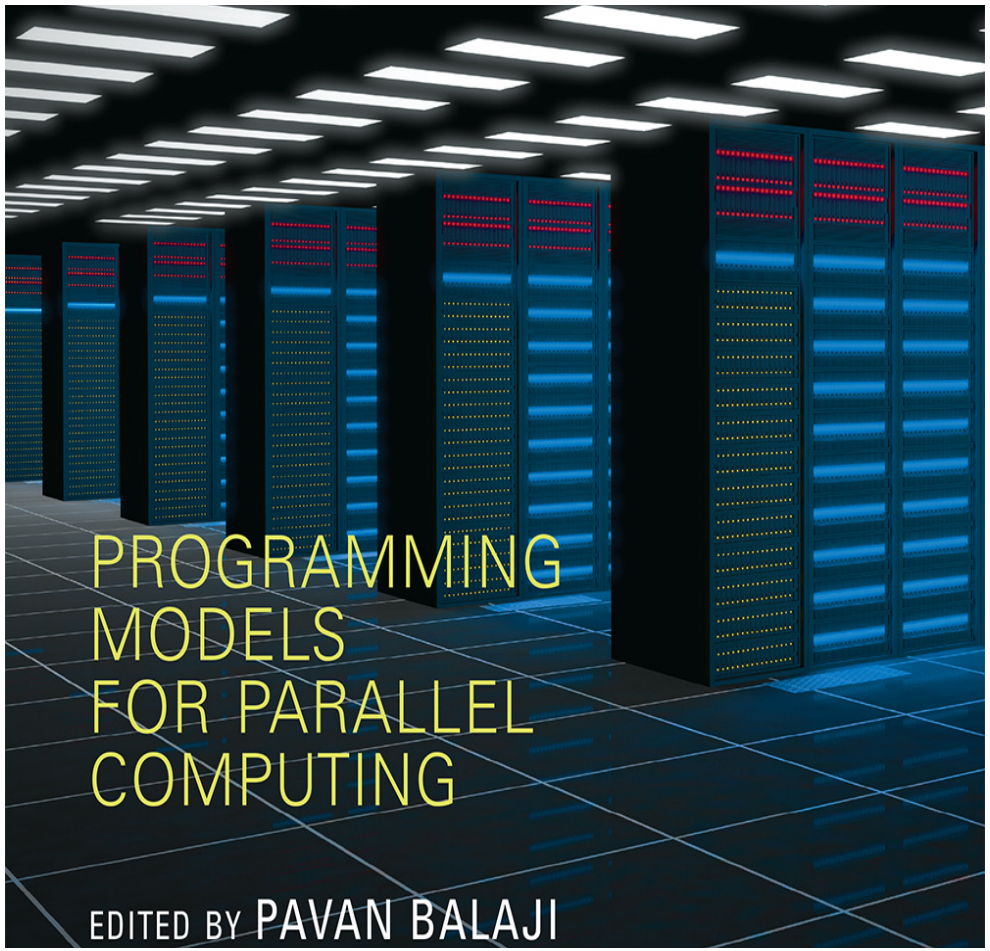


Advanced MPI, including MPI-3

Book on Parallel Programming Models

Edited by Pavan Balaji

- ***MPI:*** W. Gropp and R. Thakur
- ***GASNet:*** P. Hargrove
- ***OpenSHMEM:*** J. Kuehn and S. Poole
- ***UPC:*** K. Yelick and Y. Zheng
- ***Global Arrays:*** S. Krishnamoorthy, J. Daily, A. Vishnu, and B. Palmer
- ***Chapel:*** B. Chamberlain
- ***Charm++:*** L. Kale, N. Jain, and J. Lifflander
- ***ADLB:*** E. Lusk, R. Butler, and S. Pieper
- ***Scioto:*** J. Dinan
- ***SWIFT:*** T. Armstrong, J. M. Wozniak, M. Wilde, and I. Foster
- ***CnC:*** K. Knobe, M. Burke, and F. Schlimbach
- ***OpenMP:*** B. Chapman, D. Eachempati, and S. Chandrasekaran
- ***Cilk Plus:*** A. Robison and C. Leiserson
- ***Intel TBB:*** A. Kukanov
- ***CUDA:*** W. Hwu and D. Kirk
- ***OpenCL:*** T. Mattson



https://mitpress.mit.edu/books/programming-models-parallel-computing

Advanced MPI

Slides Available at https://anl.box.com/v/yguo-isc-tutorial-2021

Pavan Balaji
Facebook
Email: pavanbalaji.work@gmail.com
Web: https://pavanbalaji.github.io/

Torsten Hoefler
ETH Zurich
Email: htor@inf.ethz.ch
Web: http://htor.inf.ethz.ch/

Antonio Pena
Barcelona Supercomputing Center
Email: antonio.pena@bsc.es
Web: https://www.bsc.es/pena-antonio

Yanfei Guo
Argonne National Laboratory
Email: yguo@anl.gov
Web: https://www.mcs.anl.gov/~yguo/