Advanced MPI Programming

Tutorial at SC14, November 2014

Latest slides and code examples are available at

www.mcs.anl.gov/~thakur/sc14-mpi-tutorial

Pavan Balaji

Argonne National Laboratory

Email: balaji@mcs.anl.gov

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

William Gropp

University of Illinois, Urbana-Champaign

Email: wgropp@illinois.edu

Web: www.cs.illinois.edu/~wgropp

Torsten Hoefler

ETH Zurich

Email: htor@inf.ethz.ch

Web: http://htor.inf.ethz.ch/

Rajeev Thakur

Argonne National Laboratory

Email: thakur@mcs.anl.gov

Web: www.mcs.anl.gov/~thakur

About the Speakers

- **Pavan Balaji**: Computer Scientist, Mathematics and Computer Science Division, Argonne National Laboratory
- **William Gropp**: Professor, University of Illinois, Urbana-Champaign
- **Torsten Hoefler**: Assistant Professor, ETH Zurich
- **Rajeev Thakur**: Deputy Director, Mathematics and Computer Science Division, Argonne National Laboratory

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

Outline

Morning

- Introduction
  - MPI-1, MPI-2, MPI-3
- Running example: 2D stencil code
  - Simple point-to-point version
- Derived datatypes
  - Use in 2D stencil code
- One-sided communication
  - Basics and new features in MPI-3
  - Use in 2D stencil code
  - Advanced topics
    - Global address space communication

Afternoon

- MPI and Threads
  - Thread safety specification in MPI
  - How it enables hybrid programming
  - Hybrid (MPI + shared memory) version of 2D stencil code
- Nonblocking collectives
  - Parallel FFT example
- Process topologies
  - 2D stencil example
- Neighborhood collectives
  - 2D stencil example
- Recent efforts of the MPI Forum
- Conclusions

MPI-1

- MPI is a message-passing library interface standard.
  - Specification, not implementation
  - Library, not a language
- 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)

MPI-2

- Same process of definition by MPI Forum
- MPI-2 is an extension of MPI
  - Extends the message-passing model.
    - Parallel I/O
    - Remote memory operations (one-sided)
    - Dynamic process management
  - Adds other functionality
    - C++ and Fortran 90 bindings
      - similar to original C and Fortran-77 bindings
    - External interfaces
    - Language interoperability
    - MPI interaction with threads

Timeline of the MPI Standard

- MPI-1 (1994), presented at SC'93
  - Basic point-to-point communication, collectives, datatypes, etc
- MPI-2 (1997)
  - Added parallel I/O, Remote Memory Access (one-sided operations), dynamic processes, thread support, C++ bindings, …
- ---- Stable for 10 years ----
- MPI-2.1 (2008)
  - Minor clarifications and bug fixes to MPI-2
- MPI-2.2 (2009)
  - Small updates and additions to MPI 2.1
- MPI-3 (2012)
  - Major new features and additions to MPI

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

Status of MPI-3 Implementations (*)

	MPICH	MVAPICH	Open MPI	Cray MPI	Tianhe MPI	Intel MPI	IBM BG/Q MPI [1]	IBM PE MPICH [2]	IBM Platform	SGI MPI	Fujitsu MPI	MS MPI
NB collectives	✔	✔	✔	✔	✔	✔	✔	Q4 ‘14	✔	✔	✔	*
Neighborhood collectives	✔	✔	✔	✔	✔	✔	✔	Q4 ‘14	Q3 ‘15	✔	Q2 ‘15	
RMA	✔	✔	✔	✔	✔	✔	✔	Q4 ‘14	Q3 ‘15	✔	Q2 ‘15	
Shared memory	✔	✔	✔	✔	✔	✔	✔	Q4 ‘14	Q3 ‘15	✔	Q2 ‘15	✔
Tools Interface	✔	✔	✔	✔	✔	✔	✔ [3]	Q4 ‘14	Q3 ‘15	✔	Q2 ‘15	*
Non-collective comm. create	✔	✔	✔	✔	✔	✔	✔	Q4 ‘14	Q3 ‘15	✔	Q2 ‘15	
F08 Bindings	✔	✔	✔	Q4 ‘14	✔	Q4 ‘14	✔	Q4 ‘14	Q3 ‘15	✔	Q2 ‘15	
New Datatypes	✔	✔	✔	✔	✔	✔	✔	Q4 ‘14	Q3 ‘15	✔	Q2 ‘15	*
Large Counts	✔	✔	✔	✔	✔	✔	✔	Q4 ‘14	Q3 ‘15	✔	Q2 ‘15	*
Matched Probe	✔	✔	✔	✔	✔	✔	✔	Q4 ‘14	Q3 ‘15	✔	✔	*

Release dates are estimates and are subject to change at any time.
Empty cells indicate no *publicly announced* plan to implement/support that feature.

[1] Open source, but unsupported
[2] Beta release
[3] No MPI_T variables exposed
* Under development

(*) Platform-specific restrictions might apply for all supported features

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

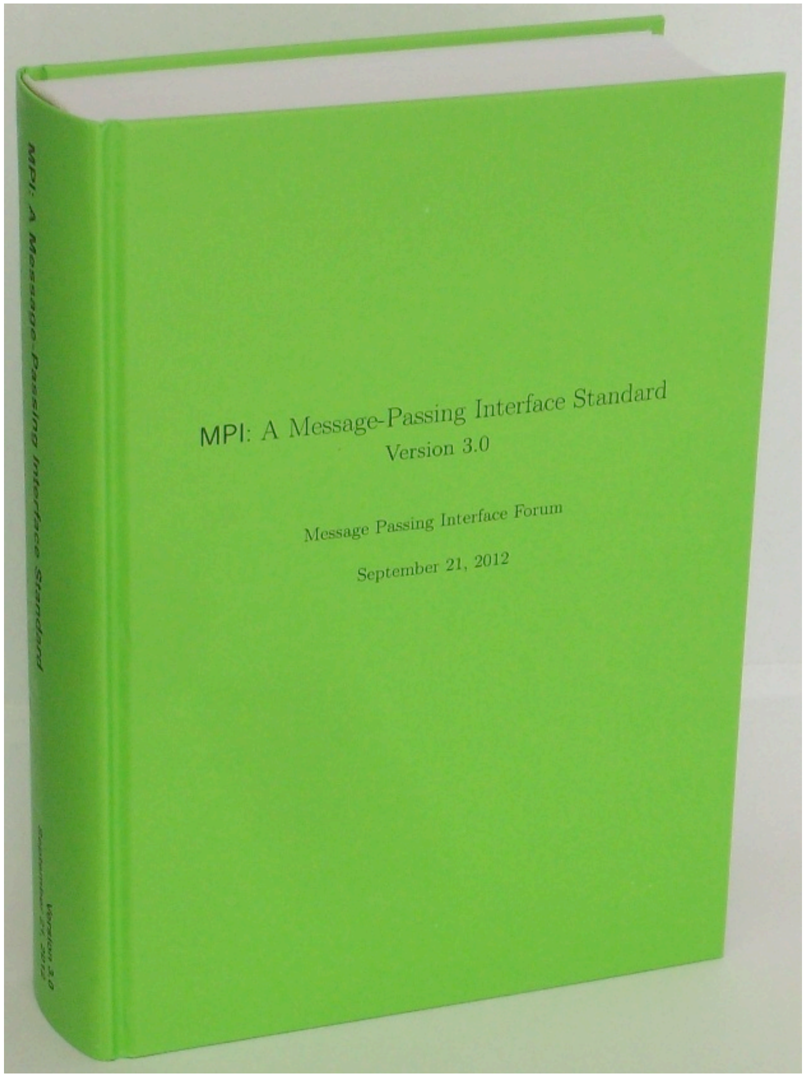
- MPI standard : http://www.mpi-forum.org/docs/docs.html
- MPI Forum : http://www.mpi-forum.org/

- 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: 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

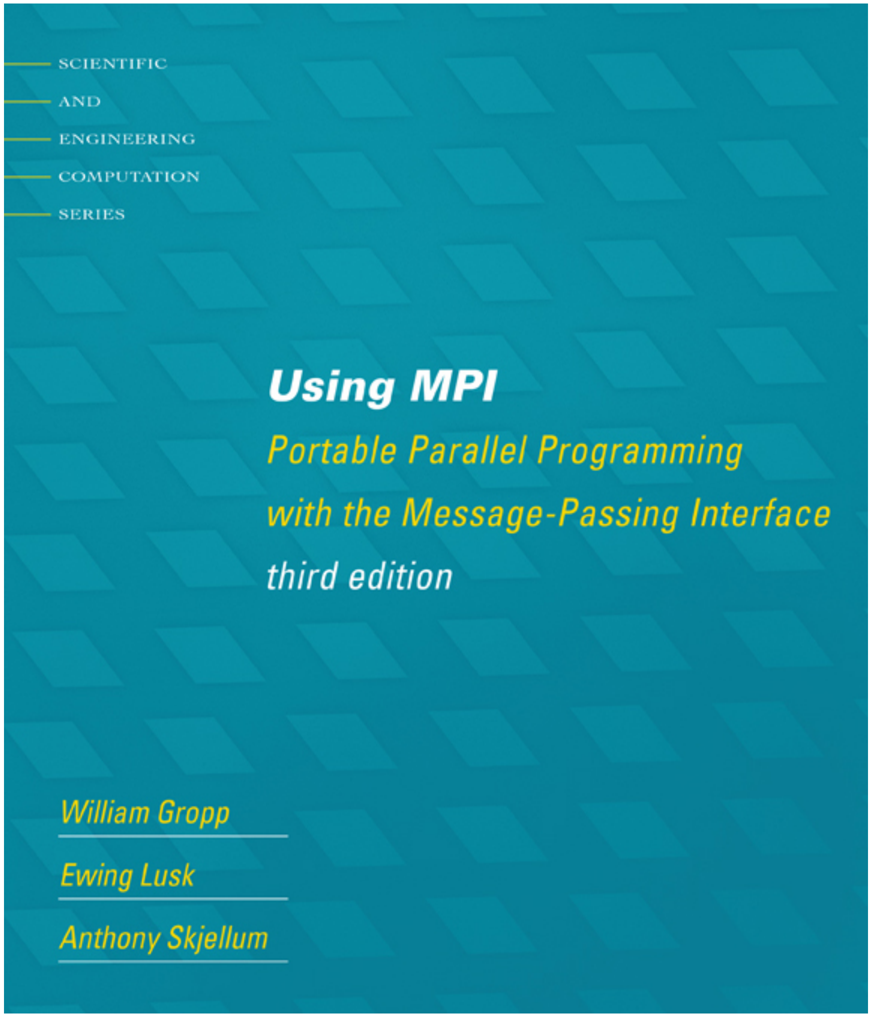
Latest MPI 3.0 Standard in Book Form

Available from amazon.com

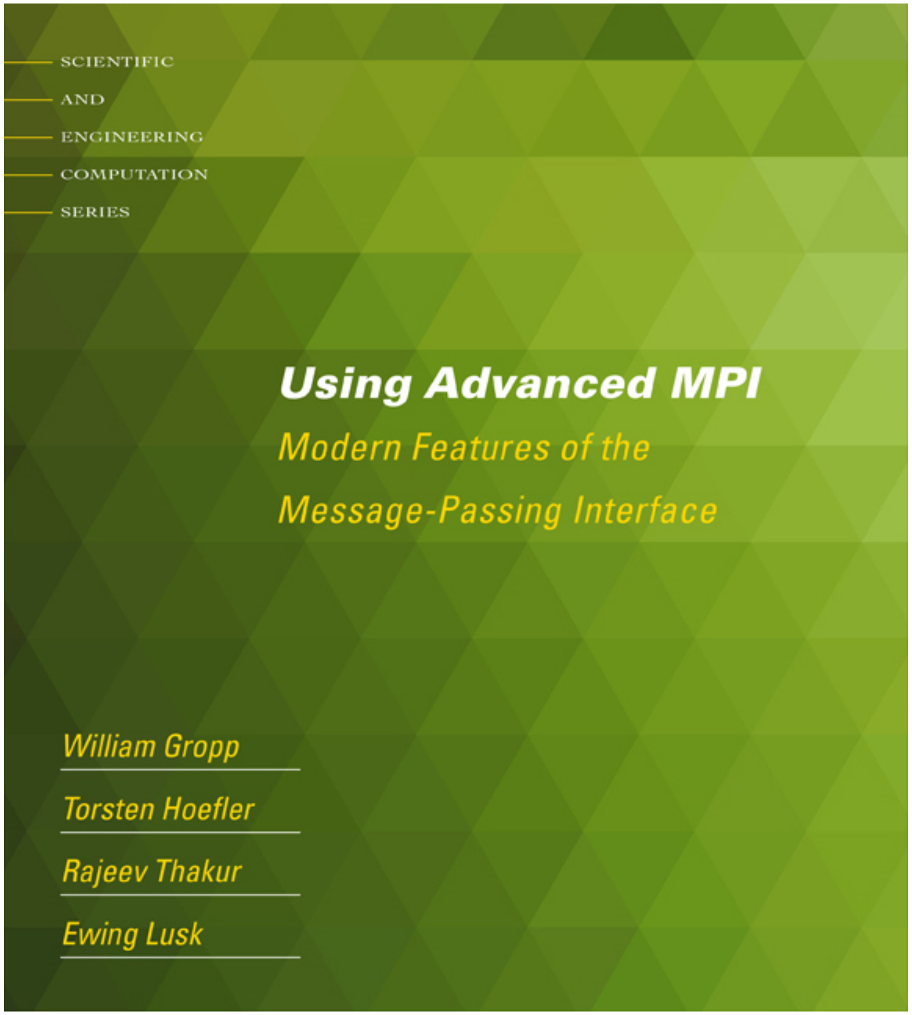
http://www.amazon.com/dp/B002TM5BQK/



New Tutorial Books on MPI



Basic MPI



Advanced MPI, including MPI-3

Our Approach in this Tutorial

- Example driven
  - 2D stencil code used as a running example throughout the tutorial
  - Other examples used to illustrate specific features
- We will walk through actual code
- We assume familiarity with basic concepts of MPI-1

Regular Mesh Algorithms

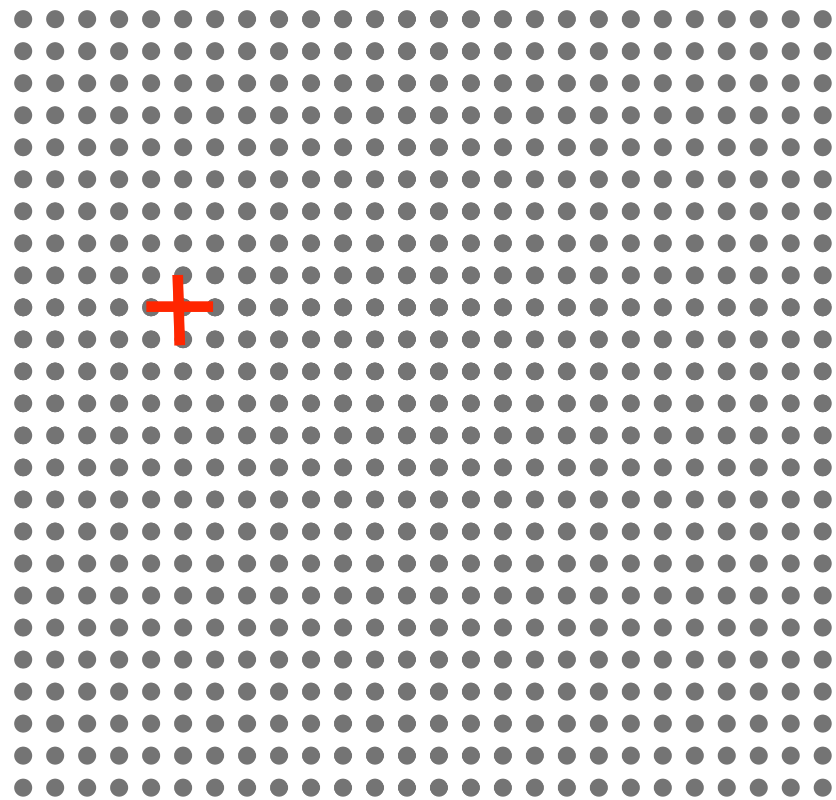
- Many scientific applications involve the solution of partial differential equations (PDEs)
- Many algorithms for approximating the solution of PDEs rely on forming a set of difference equations
  - Finite difference, finite elements, finite volume
- The exact form of the difference equations depends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

Poisson Problem

- To approximate the solution of the Poisson Problem $\nabla^2 u = f$ on the unit square, with u defined on the boundaries of the domain (Dirichlet boundary conditions), this simple 2nd order difference scheme is often used:
  - $(U(x+h,y) - 2U(x,y) + U(x-h,y)) / h^2 +$
    $(U(x,y+h) - 2U(x,y) + U(x,y-h)) / h^2 = f(x,y)$
    - Where the solution U is approximated on a discrete grid of points x=0, h, 2h, 3h, … , (1/h)h=1, y=0, h, 2h, 3h, … 1.
    - To simplify the notation, U(ih,jh) is denoted U_{ij}
- This is defined on a discrete mesh of points (x,y) = (ih,jh), for a mesh spacing “h”

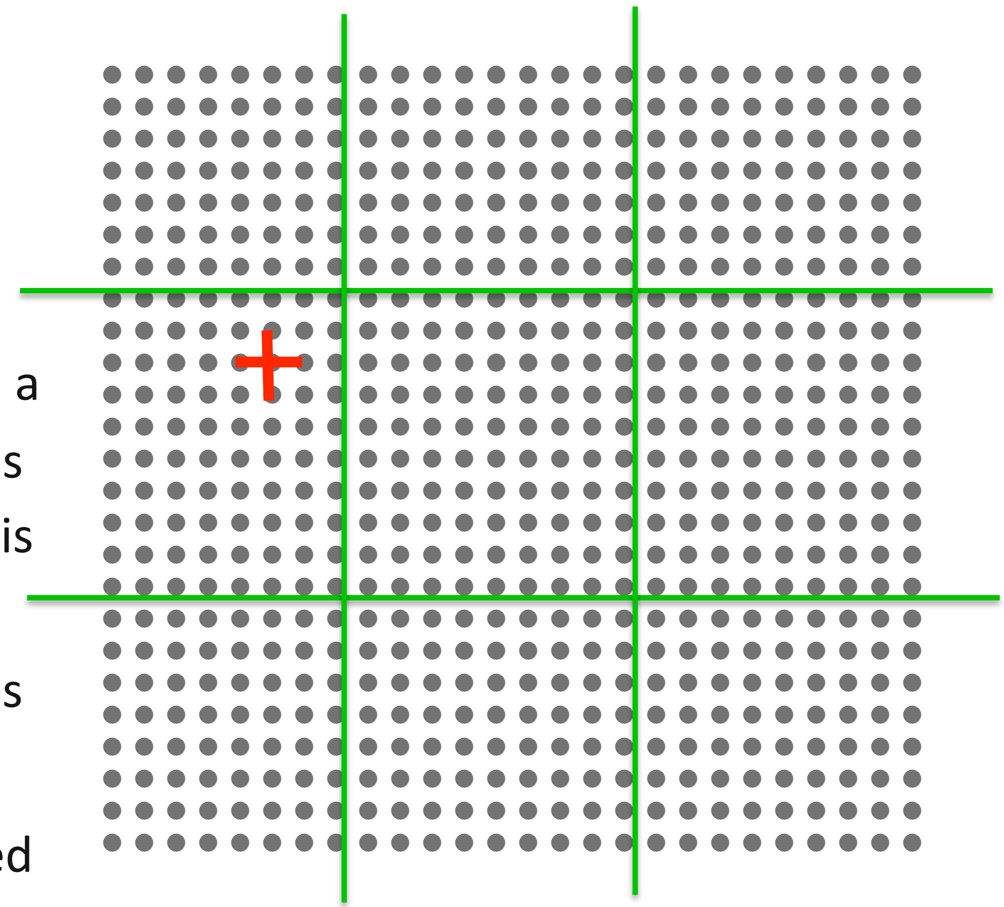
The Global Data Structure

- 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.

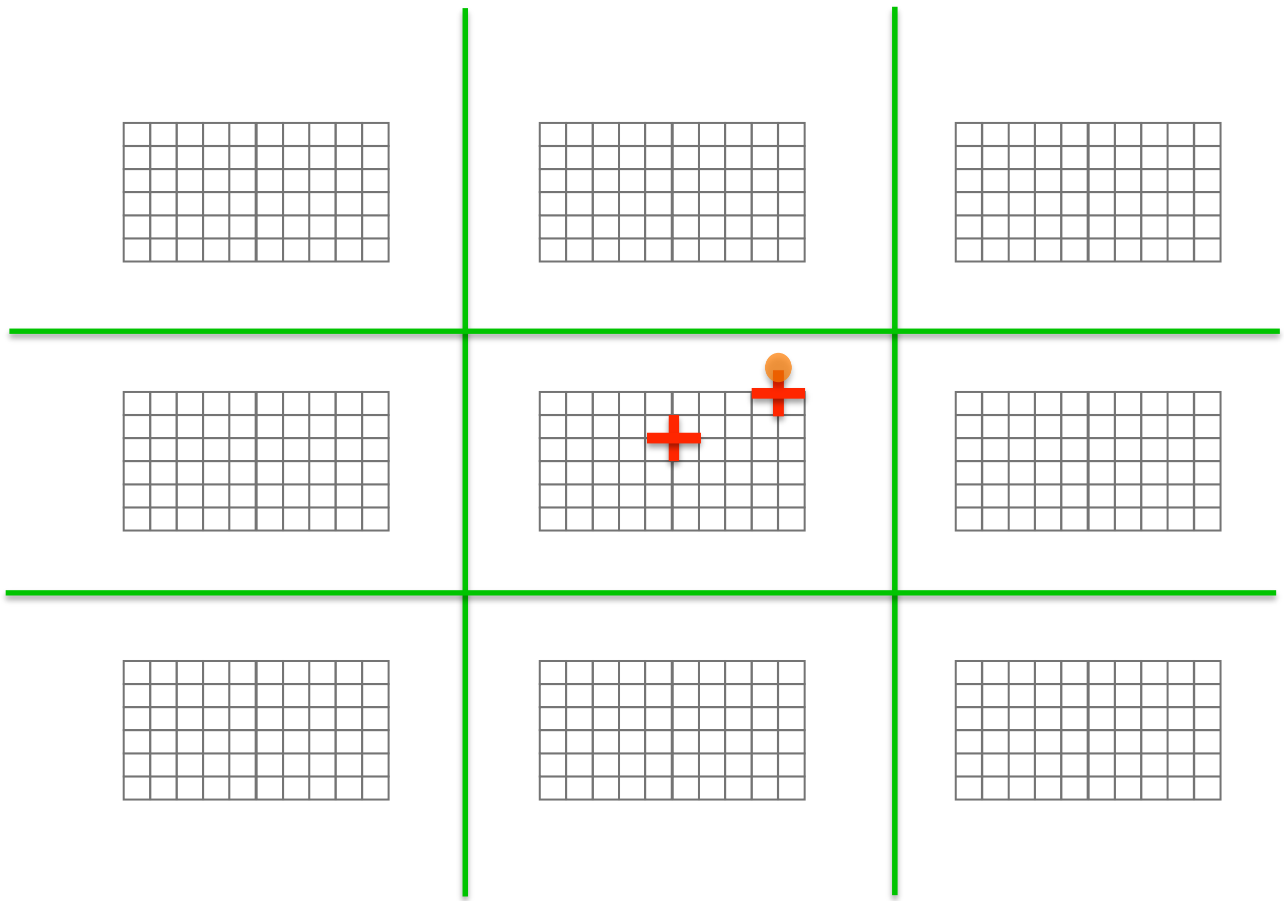


The Global Data Structure

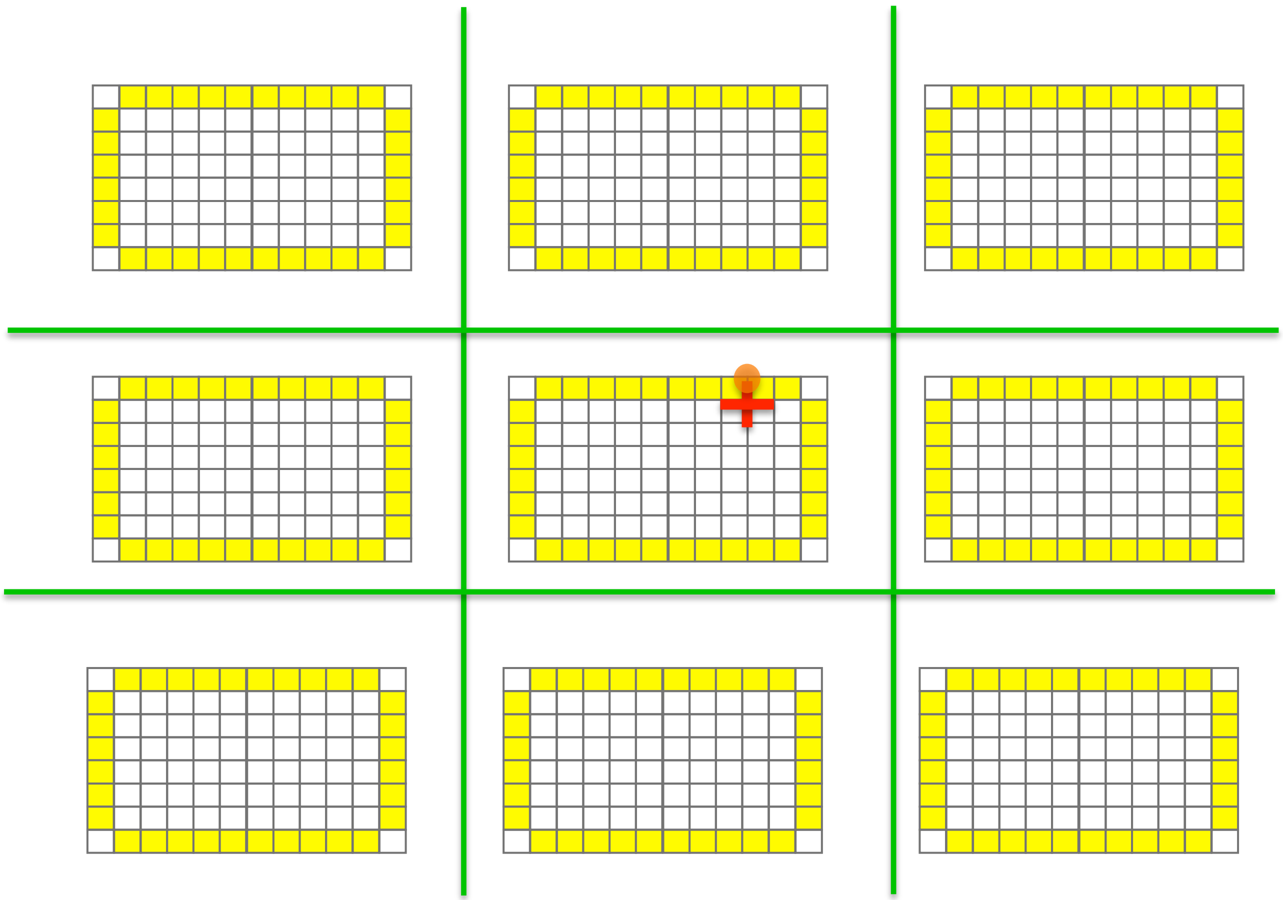
- Each circle is a mesh point
- Difference equation evaluated at each point involves the four neighbors
- The red “plus” is called the method’s stencil
- Good numerical algorithms form a matrix equation Au=f; solving this requires computing Bv, where B is a matrix derived from A. These evaluations involve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



Necessary Data Transfers

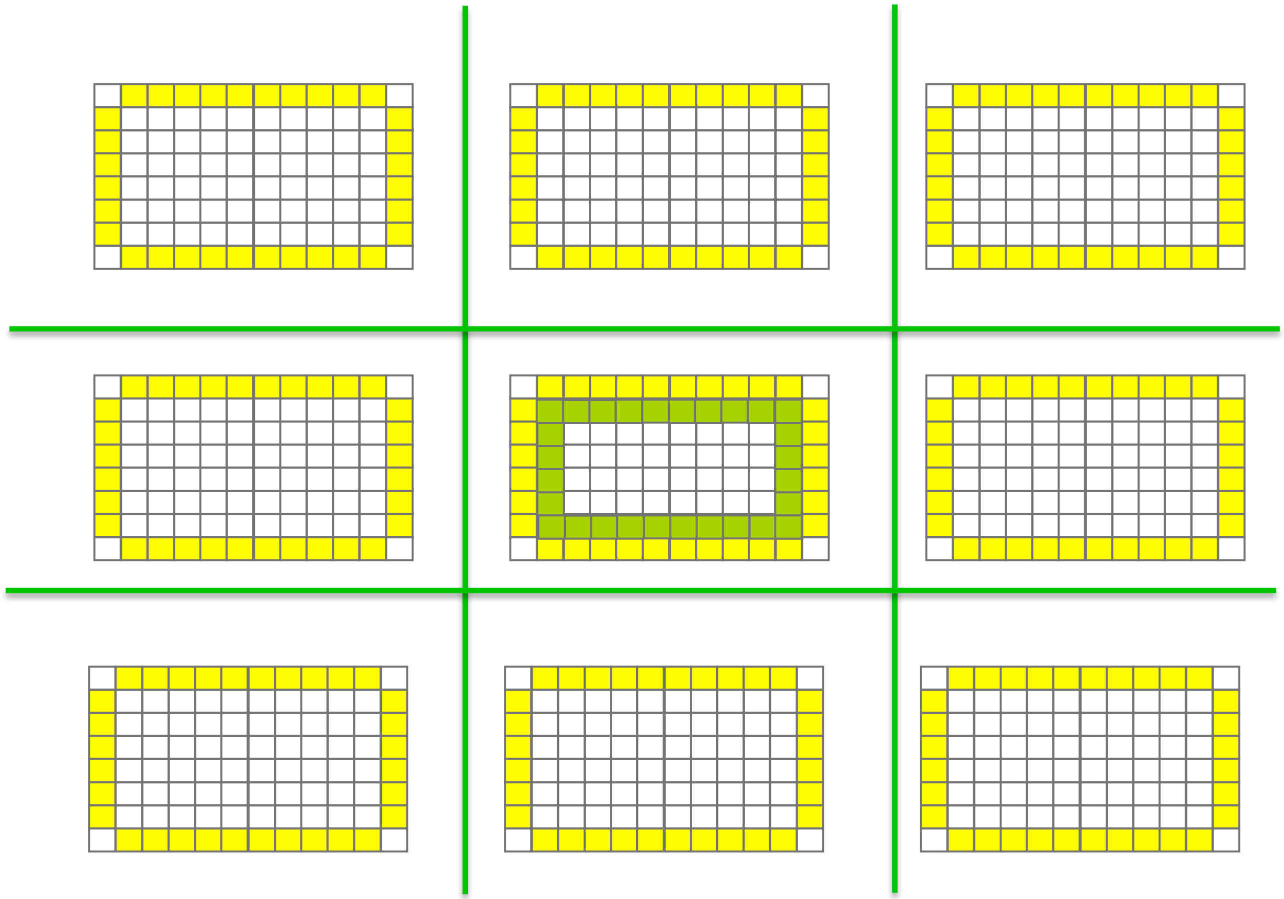


Necessary Data Transfers



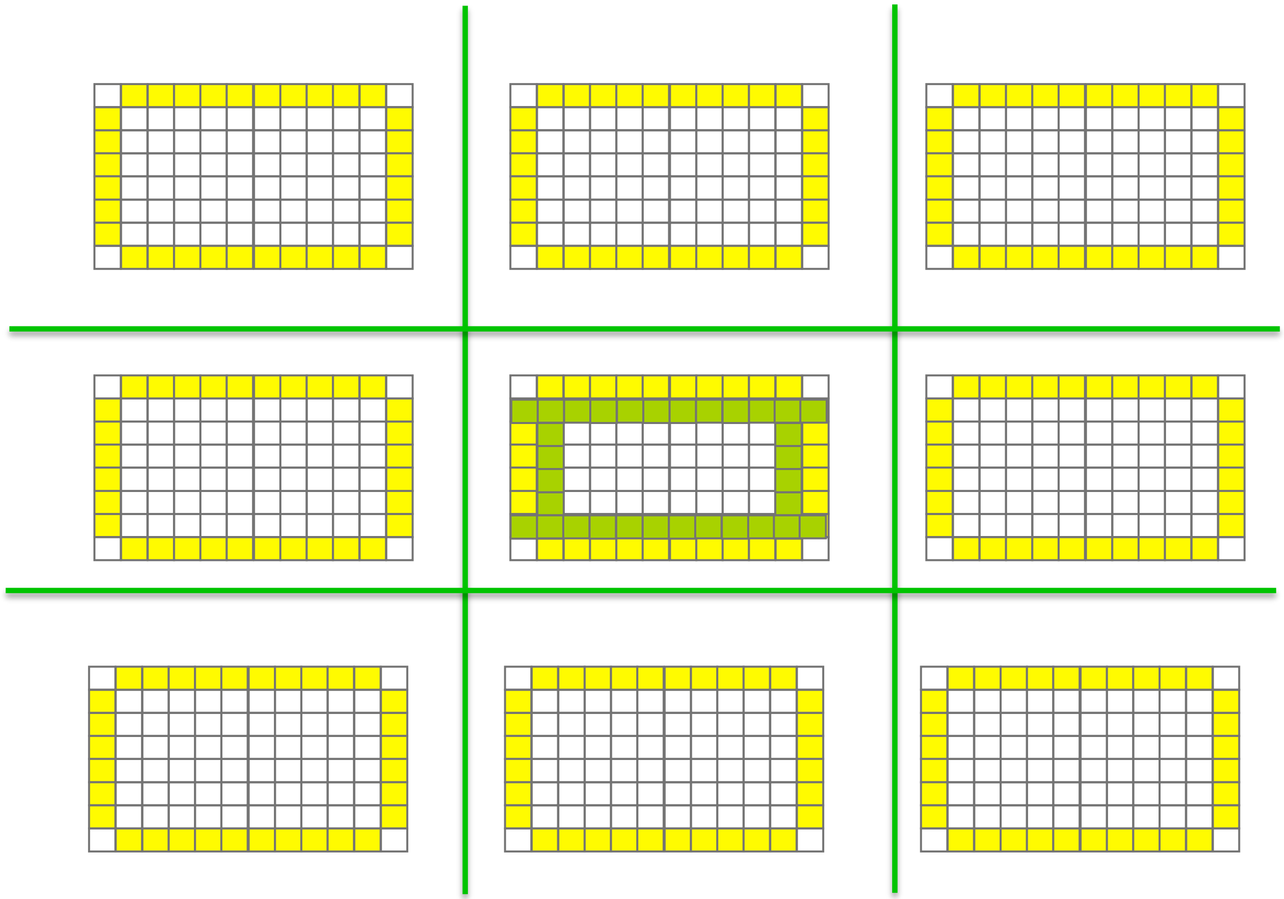
Necessary Data Transfers

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



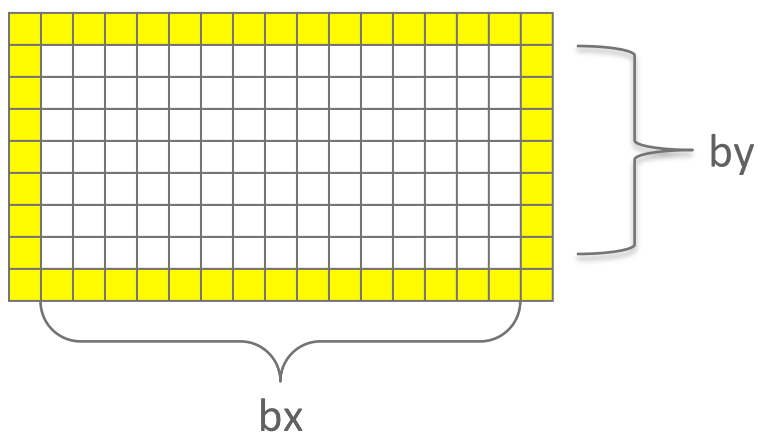
Necessary Data Transfers

- Provide access to remote data through a *halo* exchange (9 point with trick)



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)



2D Stencil Code Walkthrough

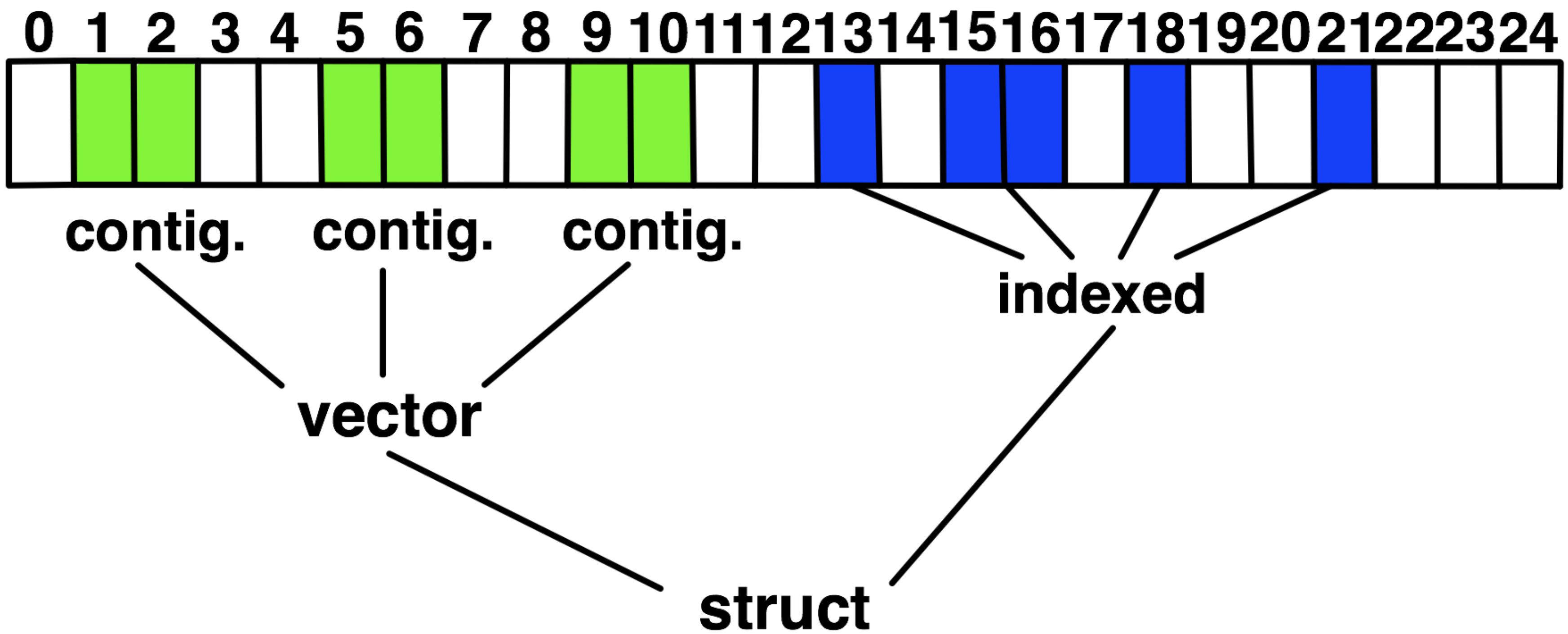
- Code can be downloaded from `www.mcs.anl.gov/~thakur/sc14-mpi-tutorial`

Datatypes

Introduction to Datatypes in MPI

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

Derived Datatype Example



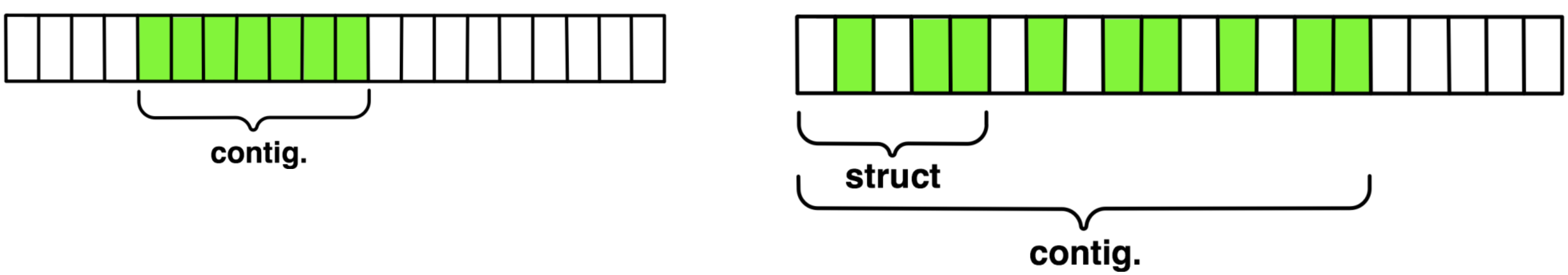
MPI's Intrinsic Datatypes

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

MPI_Type_contiguous

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

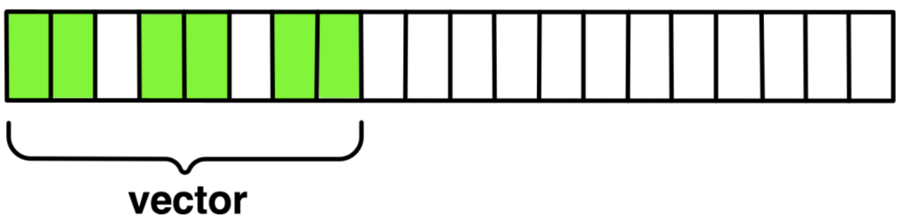
- Contiguous array of oldtype
- Should not be used as last type (can be replaced by count)

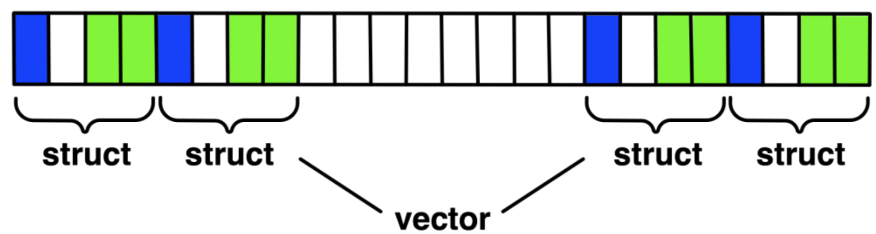


MPI_Type_vector

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

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





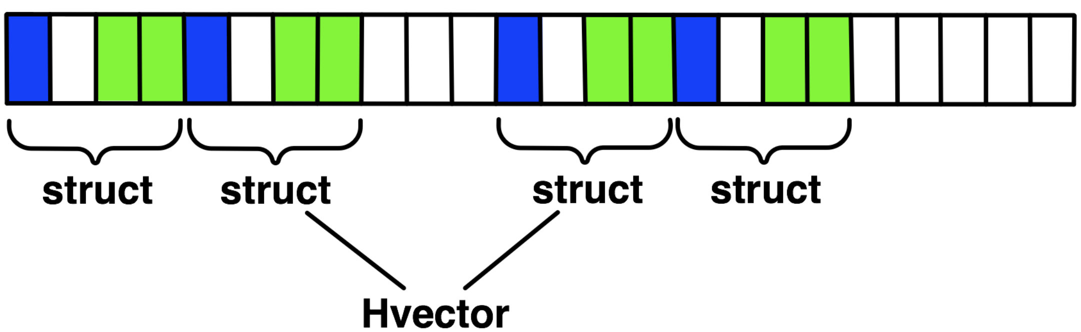
2D Stencil Code with Datatypes Walkthrough

- Code can be downloaded from
  www.mcs.anl.gov/~thakur/sc14-mpi-tutorial

MPI_Type_create_hvector

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

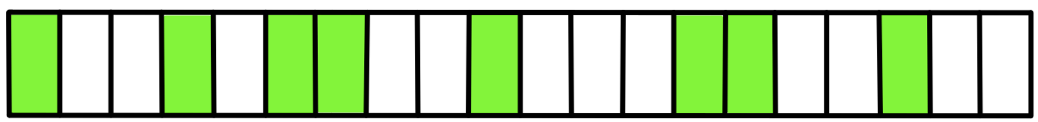
- Stride is specified in bytes, not in units of size of oldtype
- Useful for composition, e.g., vector of structs



MPI_Type_indexed

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

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

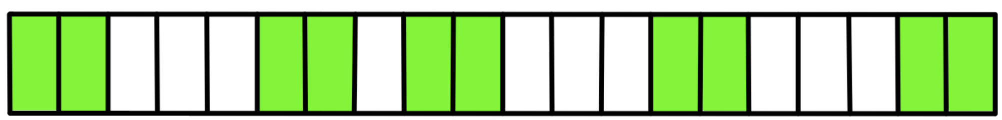


  - blen={1,1,2,1,2,1}
  - displs={0,3,5,9,13,17}

MPI_Type_create_indexed_block

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

- Like Create_indexed but blocklength is the same

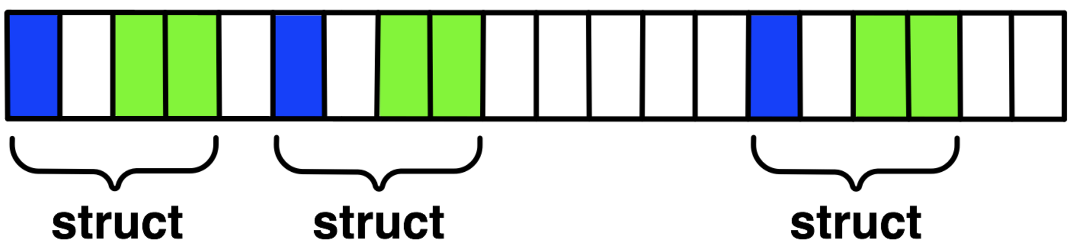


  - blen=2
  - displs={0,5,9,13,18}

MPI_Type_create_hindexed

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

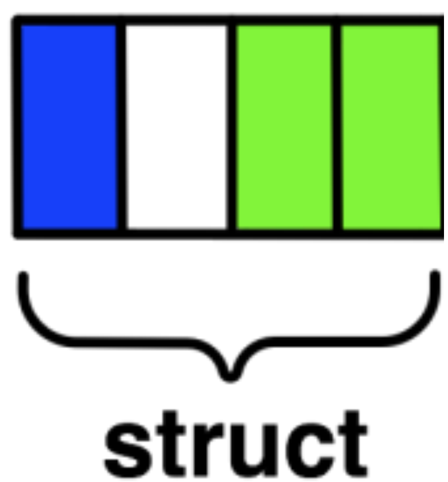
- Indexed with non-unit-sized displacements, e.g., pulling types out of different arrays



MPI_Type_create_struct

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

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



MPI_Type_create_subarray

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

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

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

MPI_Type_create_darray

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

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

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

MPI_BOTTOM and MPI_Get_address

- MPI_BOTTOM is the absolute zero address
  - Portability (e.g., may be non-zero in globally shared memory)
- MPI_Get_address
  - Returns address relative to MPI_BOTTOM
  - Portability (do not use "&" operator in C!)
- Very important to
  - build struct datatypes
  - If data spans multiple arrays

Commit, Free, and Dup

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

Other Datatype Functions

- Pack/Unpack
  - Mainly for compatibility to legacy libraries
  - Avoid using it yourself
- Get_envelope/contents
  - Only for expert library developers
  - Libraries like MPITypes[1] make this easier
- MPI_Type_create_resized
  - Change extent and size (dangerous but useful)

http://www.mcs.anl.gov/mpitypes/

Datatype Selection Order

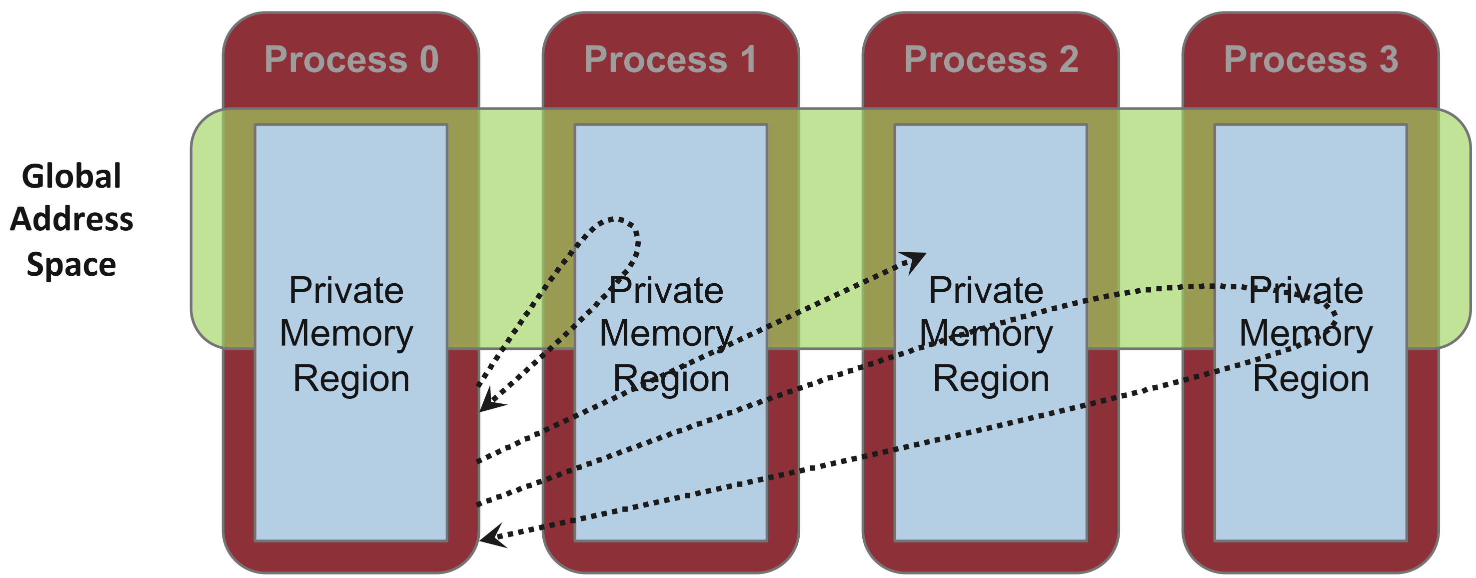
- Simple and effective performance model:
  - More parameters == slower
- **predefined < contig < vector < index_block < index < struct**
- Some (most) MPIs are inconsistent
  - But this rule is portable

W. Gropp et al.: Performance Expectations and Guidelines for MPI Derived Datatypes

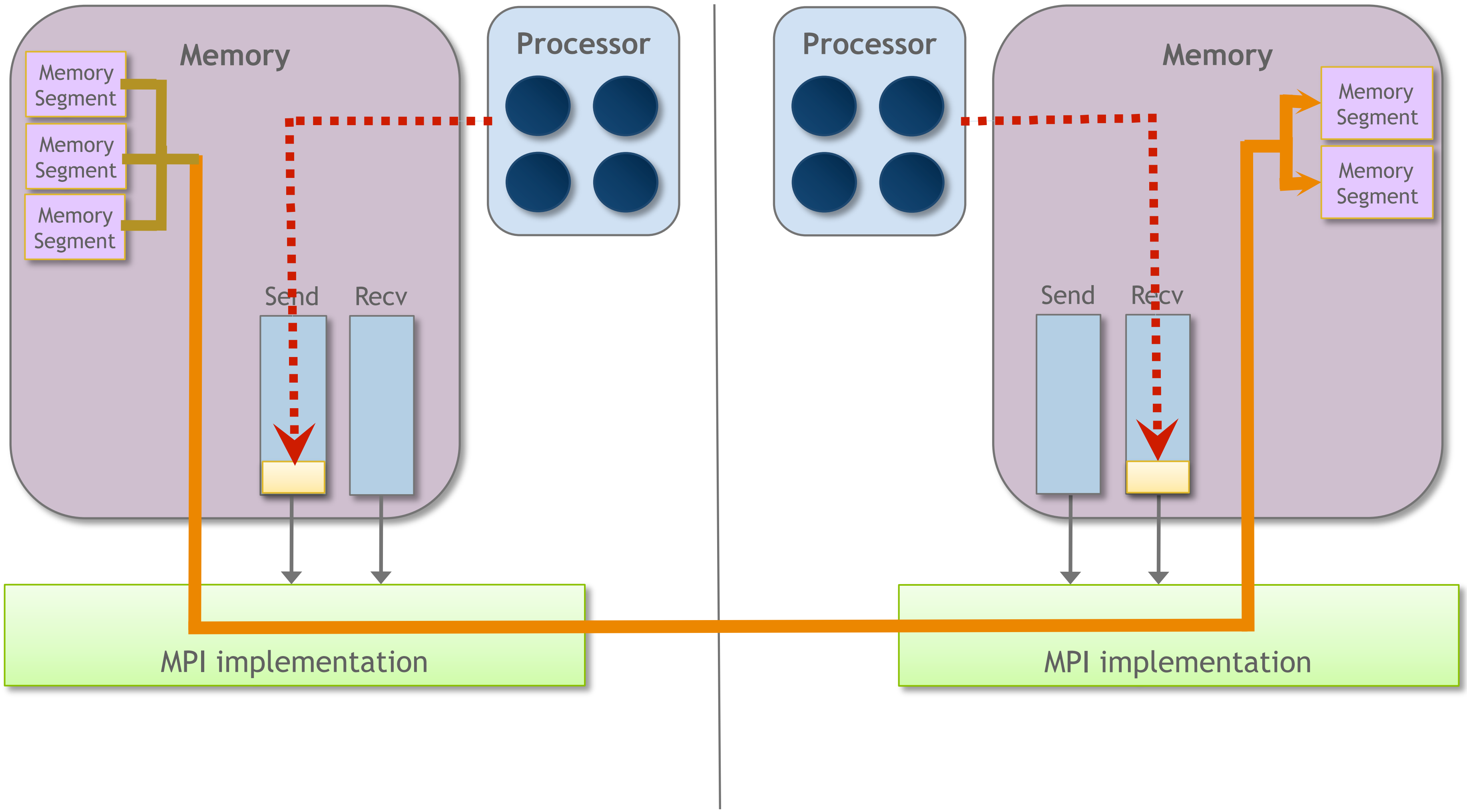
Advanced Topics: One-sided Communication

One-sided Communication

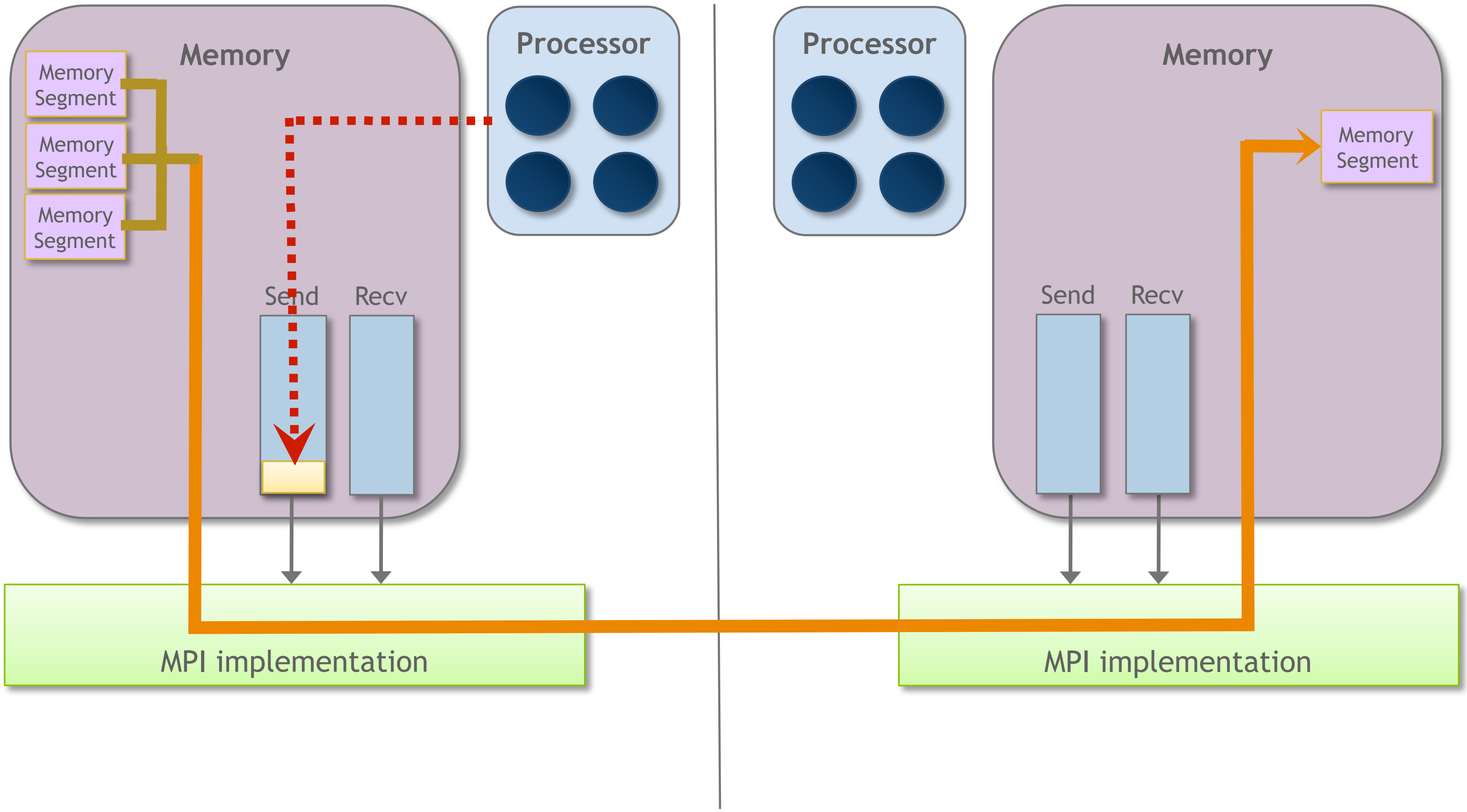
- The basic idea of one-sided communication models is to decouple data movement with process synchronization
  - Should be able move data without requiring that the remote process synchronize
  - Each process exposes a part of its memory to other processes
  - Other processes can directly read from or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



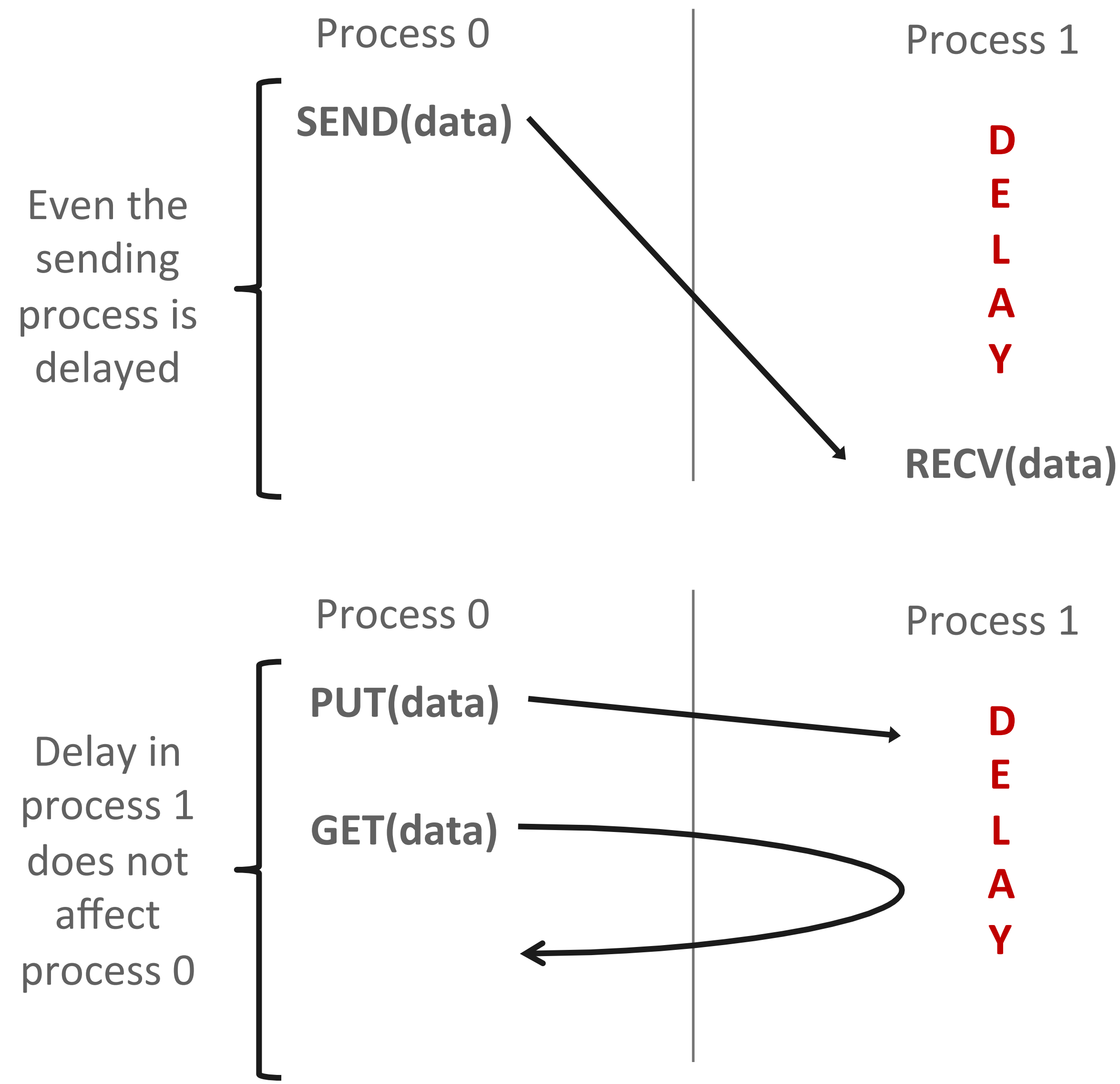
Two-sided Communication Example



One-sided Communication Example



Comparing One-sided and Two-sided Programming



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_CREATE
    - You already have an allocated buffer that you would like to make remotely accessible
  - MPI_WIN_ALLOCATE
    - You want to create a buffer and directly make it 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_CREATE

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

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

Example with MPI_WIN_CREATE

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

    MPI_Init(&argc, &argv);

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

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

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

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

MPI_WIN_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_DYNAMIC

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

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

Example with MPI_WIN_CREATE_DYNAMIC

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

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

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

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

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

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

    MPI_Finalize(); return 0;
}
```

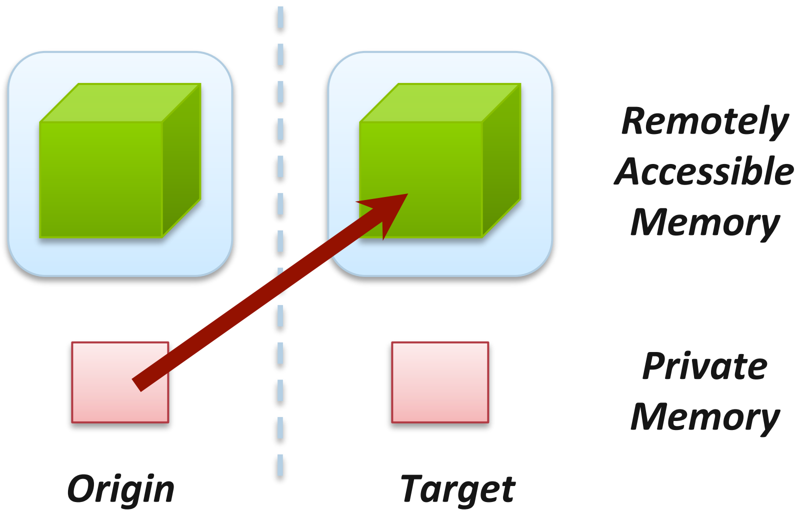
Data movement

- MPI provides ability to read, write and atomically modify data in remotely accessible memory regions
  - MPI_PUT
  - MPI_GET
  - MPI_ACCUMULATE
  - MPI_GET_ACCUMULATE
  - MPI_COMPARE_AND_SWAP
  - MPI_FETCH_AND_OP

Data movement: *Put*

```
MPI_Put(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)
```

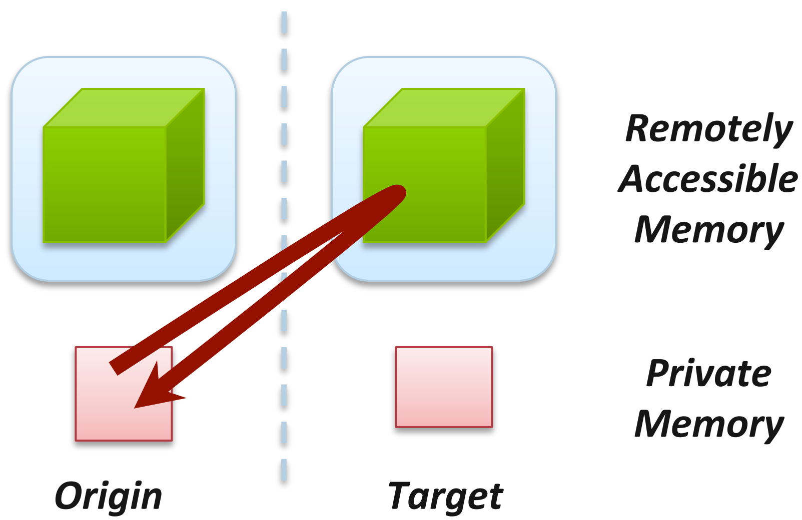
- Move data <u>from</u> origin, <u>to</u> target
- Separate data description triples for origin and target



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)
```

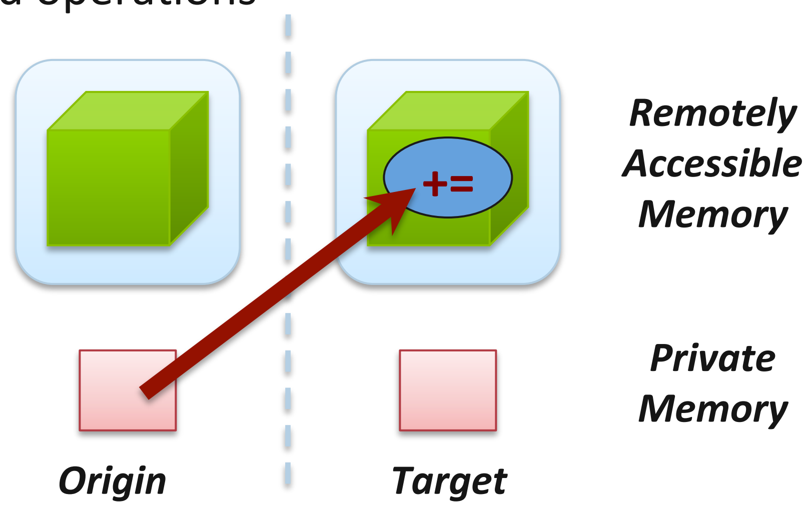
- Move data to origin, from target



Atomic Data Aggregation: *Accumulate*

```
MPI_Accumulate(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)
```

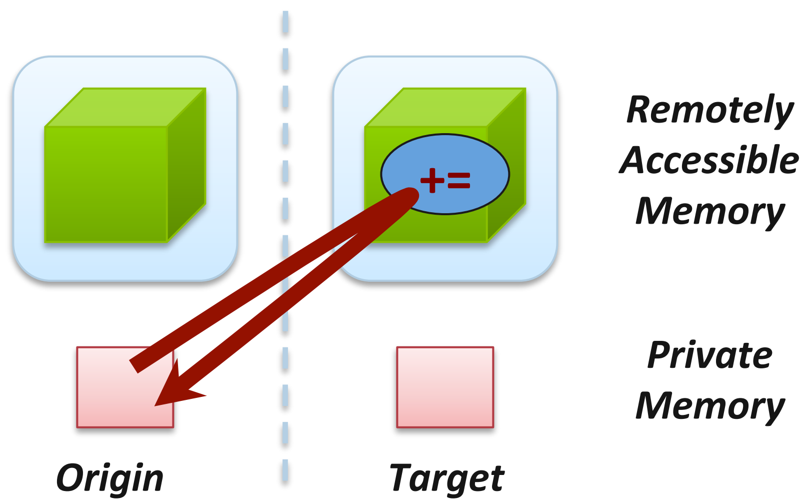
- Atomic update operation, similar to a put
  - Reduces origin and target data into target buffer using op argument as combiner
  - Predefined ops only, no user-defined operations
- Different data layouts between target/origin OK
  - Basic type elements must match
- Op = MPI_REPLACE
  - Implements *f(a,b)=b*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



Atomic Data Aggregation: *Get Accumulate*

```
MPI_Get_accumulate(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: *CAS and FOP*

```
MPI_Fetch_and_op(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(void *origin_addr, 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

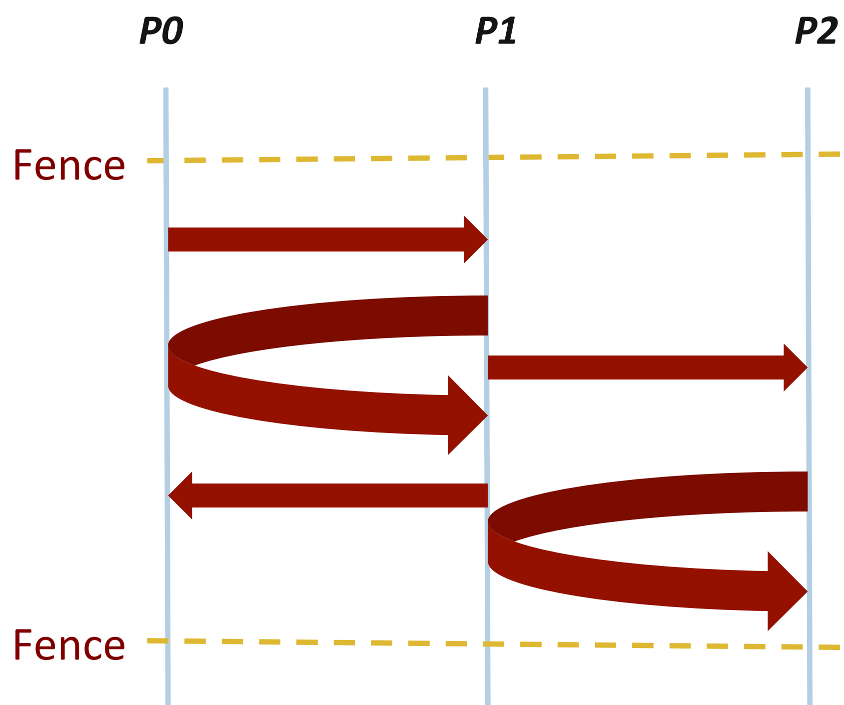
RMA Synchronization Models

- 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)
```

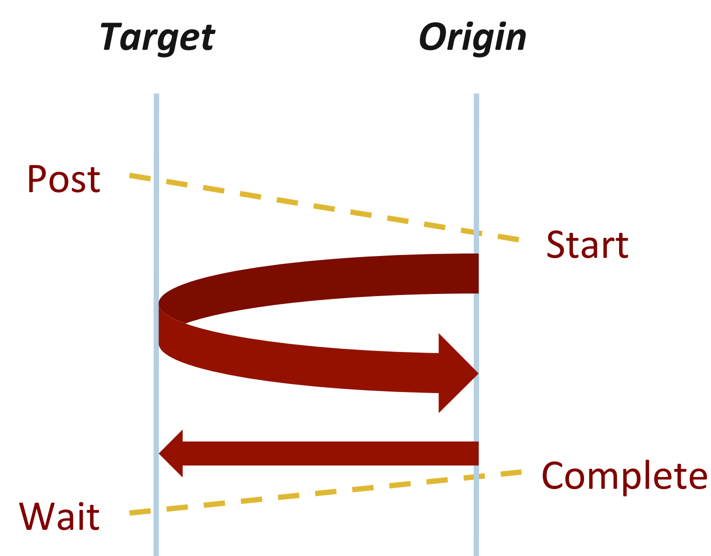
- Collective synchronization model
- Starts *and* ends access and exposure epochs on all processes in the window
- All processes in group of "win" do an MPI_WIN_FENCE to open an epoch
- Everyone can issue PUT/GET operations to read/write data
- Everyone does an MPI_WIN_FENCE to close the epoch
- All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



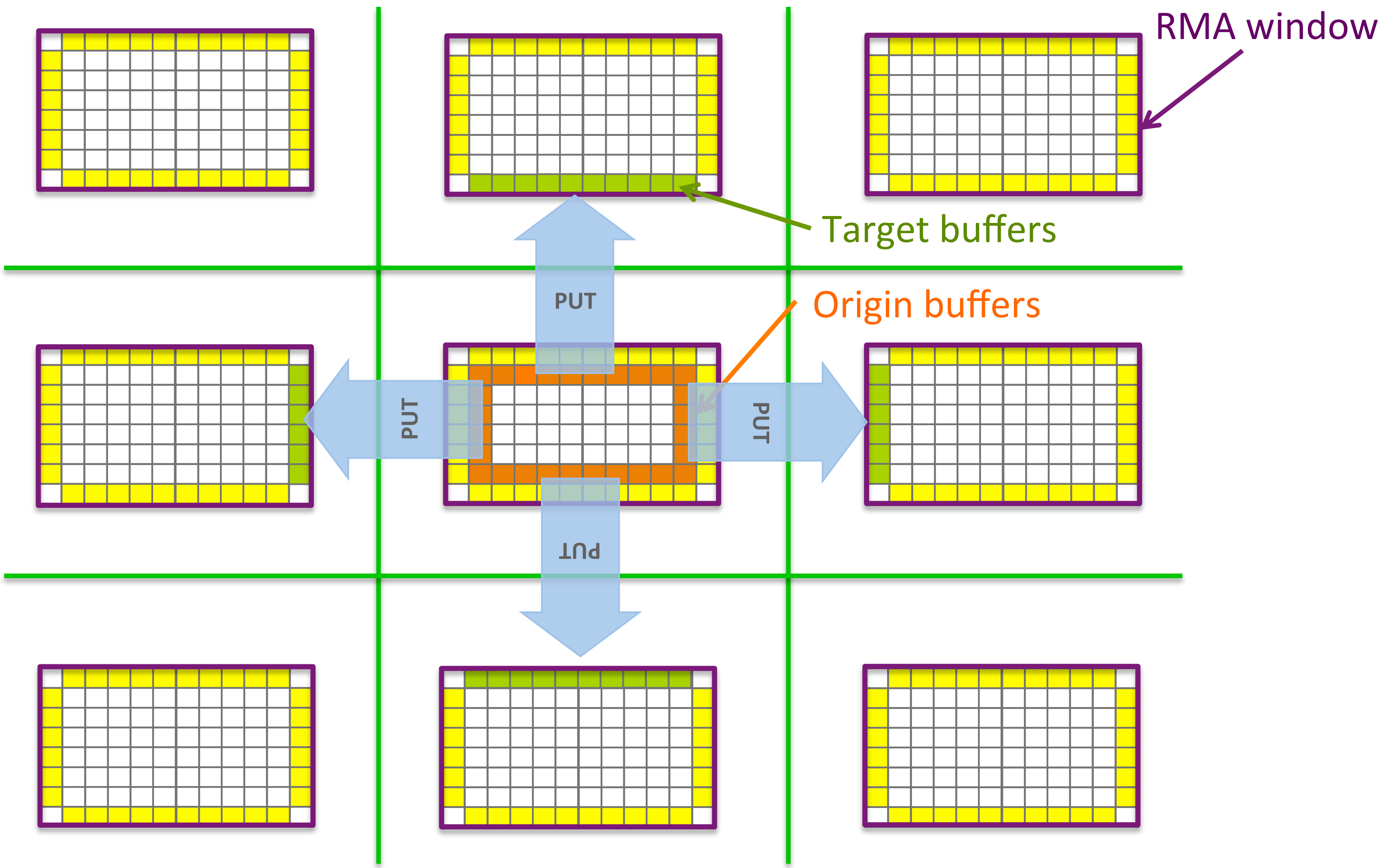
PSCW: Generalized Active Target Synchronization

```
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



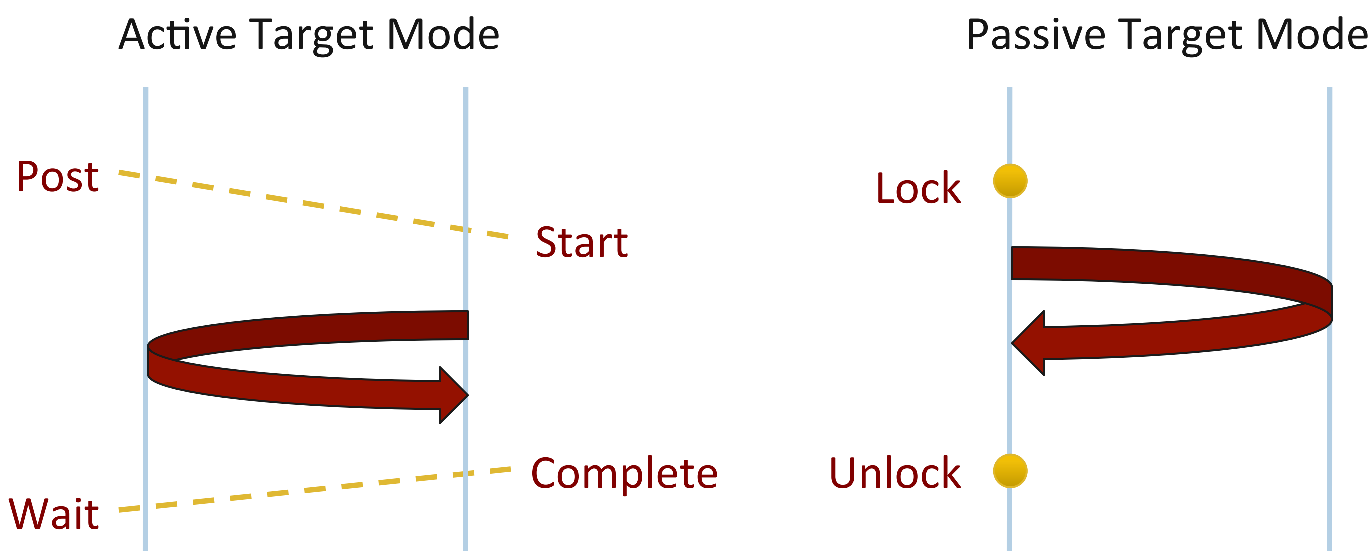
Implementing Stencil Computation with RMA Fence



Walkthrough of 2D Stencil Code with RMA

- Code can be downloaded from
  www.mcs.anl.gov/~thakur/sc14-mpi-tutorial

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

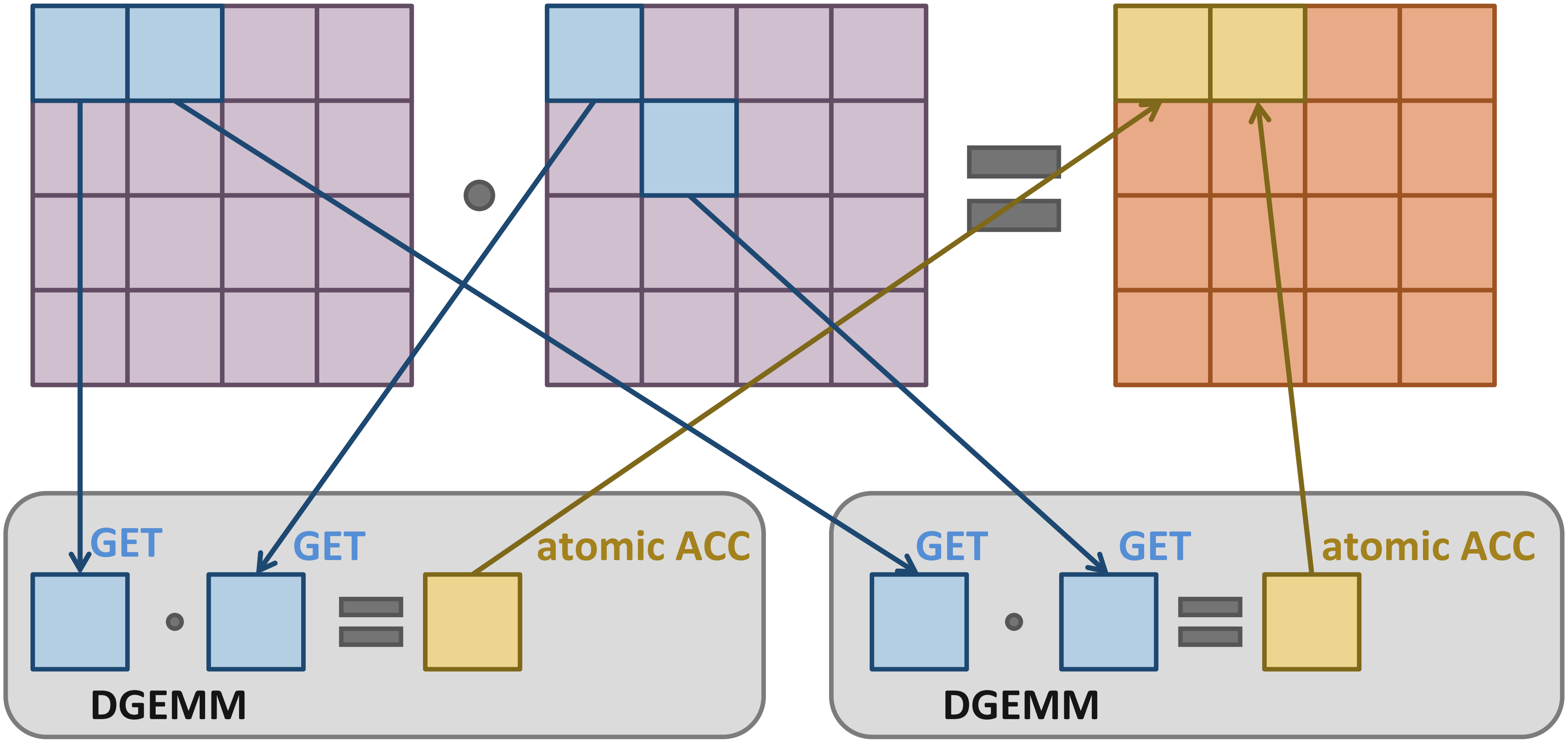
```
MPI_Win_lock_all(int assert, MPI_Win win)
```

```
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

Implementing GA-like Computation by RMA Lock/Unlock



local buffer on P0

local buffer on P1

Code Example

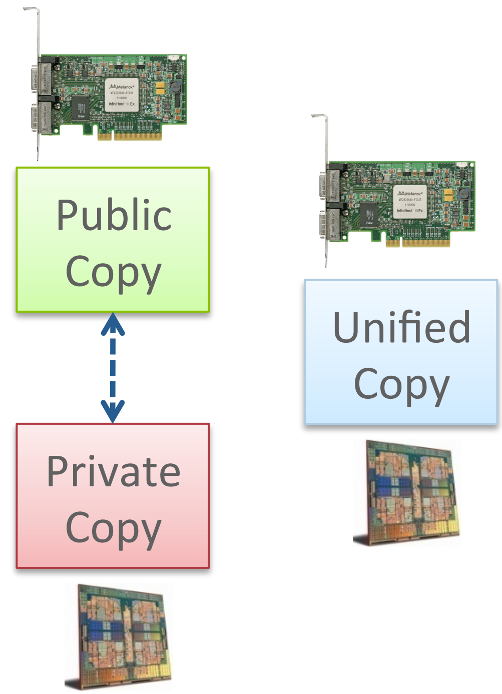
- ga_mpi_ddt_rma.c
- Only synchronization from origin processes, no synchronization from target processes

Which synchronization mode should I use, when?

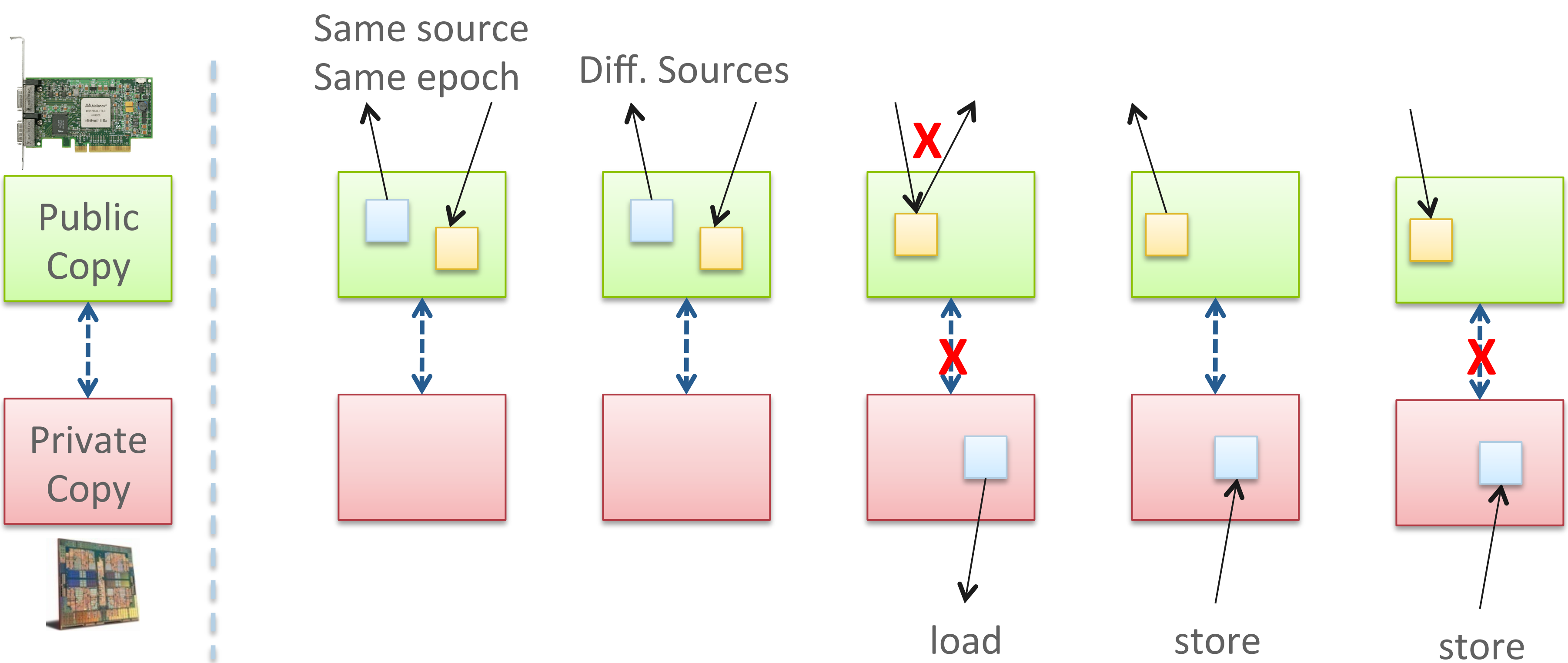
- RMA communication has low overheads versus send/recv
  - Two-sided: Matching, queuing, buffering, unexpected receives, etc…
  - One-sided: No matching, no buffering, always ready to receive
  - Utilize RDMA provided by high-speed interconnects (e.g. InfiniBand)
- Active mode: bulk synchronization
  - 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
- MPI-2: Separate Model
  - Logical public and private copies
  - MPI provides software coherence between window copies
  - Extremely portable, to systems that don't provide hardware coherence
- MPI-3: New Unified Model
  - Single copy of the window
  - System must provide coherence
  - Superset of separate semantics
    - E.g. allows concurrent local/remote access
  - Provides access to full performance potential of hardware

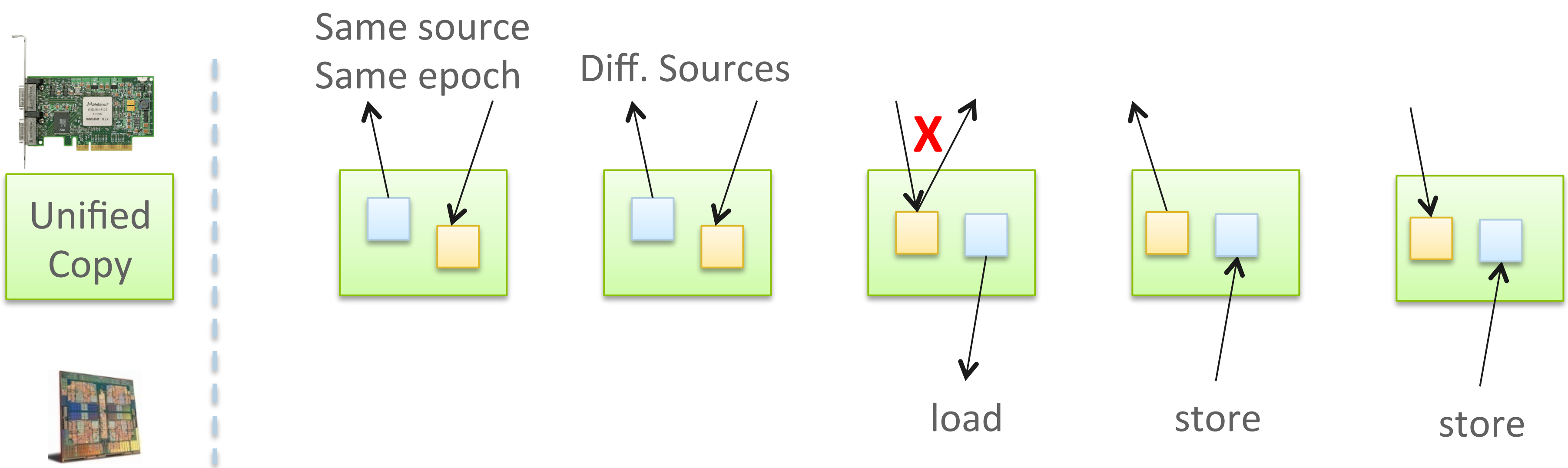


MPI RMA Memory Model (separate windows)



- 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

Hybrid Programming with Threads, Shared Memory, and GPUs

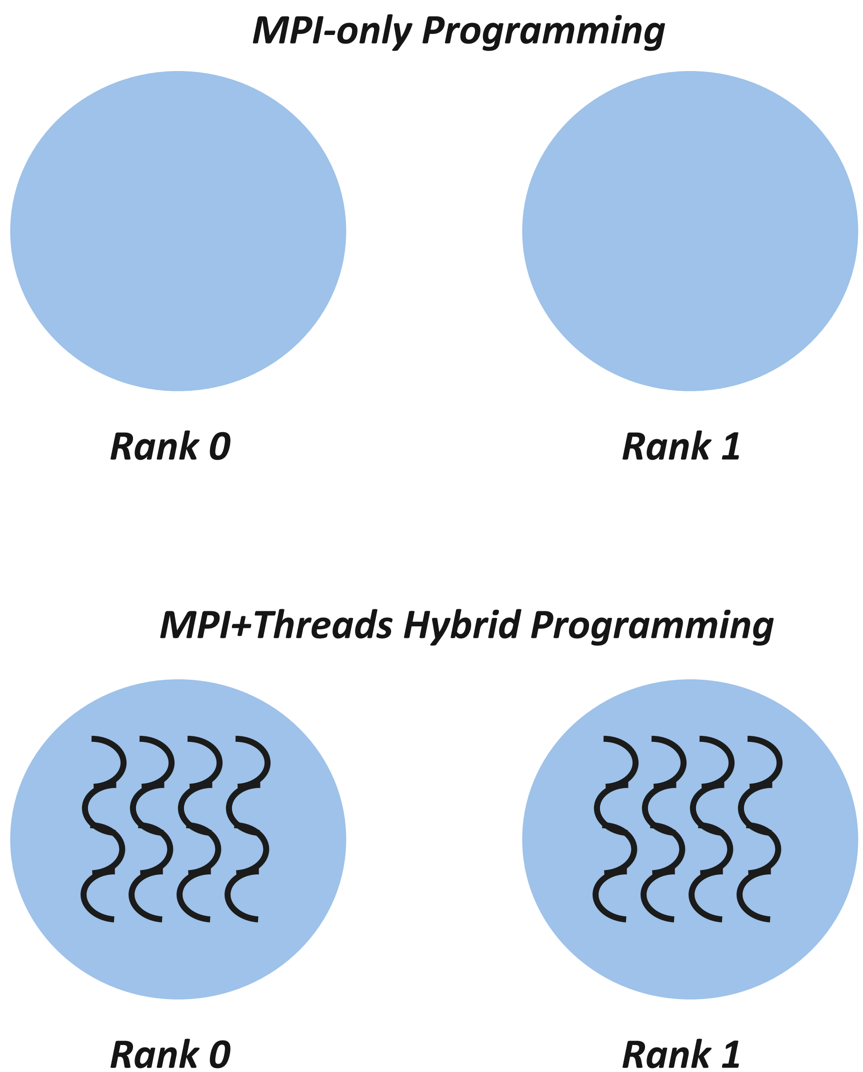
MPI and Threads

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

Programming for Multicore

- Common options for programming multicore clusters
  - All MPI
    - MPI between processes both within a node and across nodes
    - MPI internally uses shared memory to communicate within a node
  - MPI + OpenMP
    - Use OpenMP within a node and MPI across nodes
  - MPI + Pthreads
    - Use Pthreads within a node and MPI across nodes
- The latter two approaches are known as “hybrid programming”

Hybrid Programming with MPI+Threads



- In MPI-only programming, each MPI process has a single program counter
- In MPI+threads hybrid programming, there can be multiple threads executing simultaneously
  - All threads share all MPI objects (communicators, requests)
  - The MPI implementation might need to take precautions to make sure the state of the MPI stack is consistent

MPI's Four Levels of Thread Safety

- MPI defines four levels of thread safety -- these are commitments the application makes to the MPI
  - MPI_THREAD_SINGLE: only one thread exists in the application
  - MPI_THREAD_FUNNELED: multithreaded, but only the main thread makes MPI calls (the one that called MPI_Init_thread)
  - MPI_THREAD_SERIALIZED: multithreaded, but only one thread *at a time* makes MPI calls
  - MPI_THREAD_MULTIPLE: multithreaded and any thread can make MPI calls at any time (with some restrictions to avoid races – see next slide)
- 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 threads in the system
  - E.g., there are no OpenMP parallel regions

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

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

    MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

#pragma omp parallel for
    for (i = 0; i < 100; i++)
        compute(buf[i]);

    /* Do MPI stuff */

    MPI_Finalize();

    return 0;
}
```

MPI_THREAD_SERIALIZED

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

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

    MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &provided);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

#pragma omp parallel for
    for (i = 0; i < 100; i++) {
        compute(buf[i]);
#pragma omp critical
        /* Do MPI stuff */
    }

    MPI_Finalize();

    return 0;
}
```

MPI_THREAD_MULTIPLE

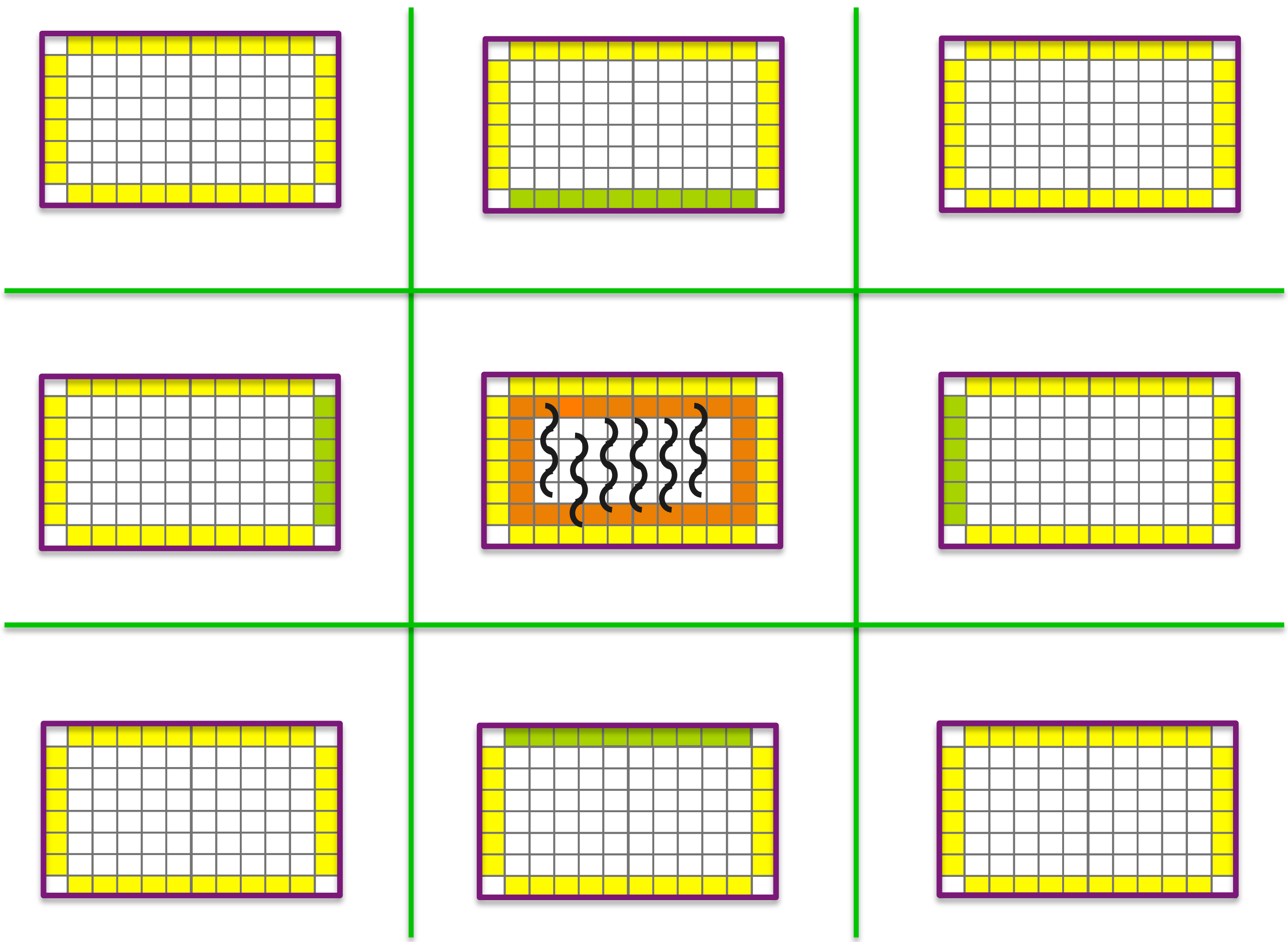
- Any thread can make MPI calls any time (restrictions apply)

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

    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

#pragma omp parallel for
    for (i = 0; i < 100; i++) {
        compute(buf[i]);
        /* Do MPI stuff */
    }

    MPI_Finalize();

    return 0;
}
```

Threads and MPI

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

Implementing Stencil Computation using MPI_THREAD_FUNNELED



Code Examples

- *stencil_mpi_ddt_funneled.c*
- Parallelize computation (OpenMP parallel for)
- Main thread does all communication

Specification of 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
- ***Blocking:*** 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 1	MPI_Bcast(comm)	MPI_Bcast(comm)
Thread 2	MPI_Barrier(comm)	MPI_Barrier(comm)

- P0 and P1 can have different orderings of Bcast and Barrier
- Here the user must use some kind of synchronization to ensure that either thread 1 or thread 2 gets scheduled first on both processes
- Otherwise a broadcast may get matched with a barrier on the same communicator, which is not allowed in MPI

Ordering in MPI_THREAD_MULTIPLE: Incorrect Example with 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

Ordering in MPI_THREAD_MULTIPLE: Incorrect Example with Object Management

	Process 0	Process 1
Thread 1	MPI_Bcast(comm)	MPI_Bcast(comm)
Thread 2	MPI_Comm_free(comm)	MPI_Comm_free(comm)

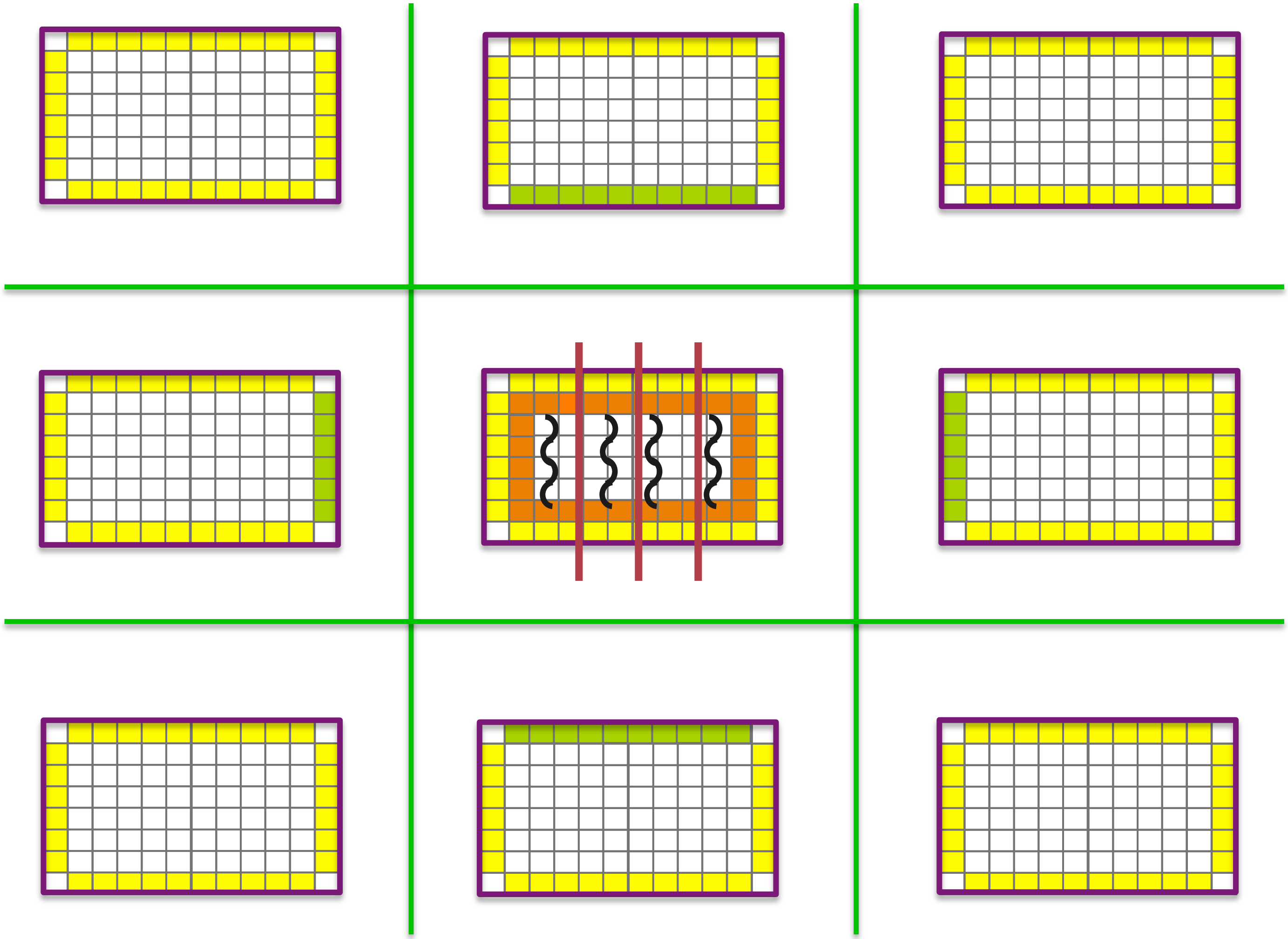
- The user has to make sure that one thread is not using an object while another thread is freeing it
  - This is essentially an ordering issue; the object might get freed before it is used

Blocking Calls in MPI_THREAD_MULTIPLE: Correct Example

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

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

Implementing Stencil Computation using MPI_THREAD_MULTIPLE



Code Examples

- *stencil_mpi_ddt_multiple.c*
- Divide the process memory among OpenMP threads
- Each thread responsible for communication and computation

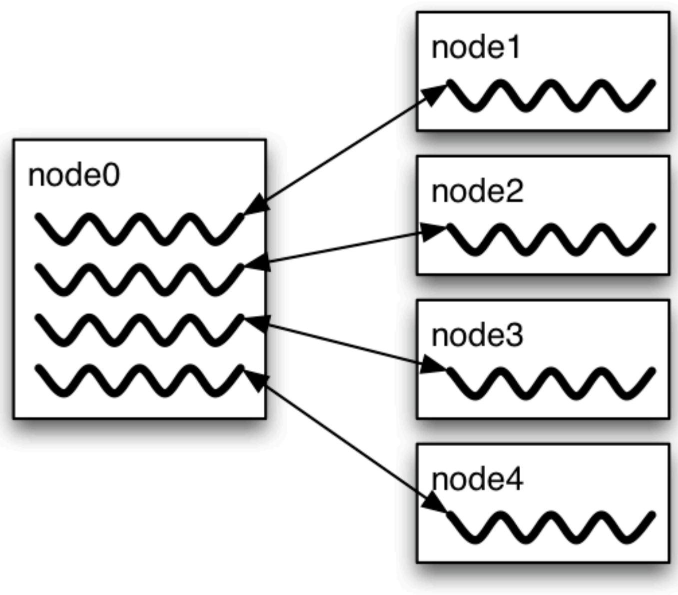
The Current Situation

- All MPI implementations support MPI_THREAD_SINGLE (duh).
- They probably support MPI_THREAD_FUNNELED even if they don't admit it.
  - Does require thread-safe malloc
  - Probably OK in OpenMP programs
- Many (but not all) implementations support THREAD_MULTIPLE
  - Hard to implement efficiently though (lock granularity issue)
- "Easy" OpenMP programs (loops parallelized with OpenMP, communication in between loops) only need FUNNELED
  - So don't need "thread-safe" MPI for many hybrid programs
  - But watch out for Amdahl's Law!

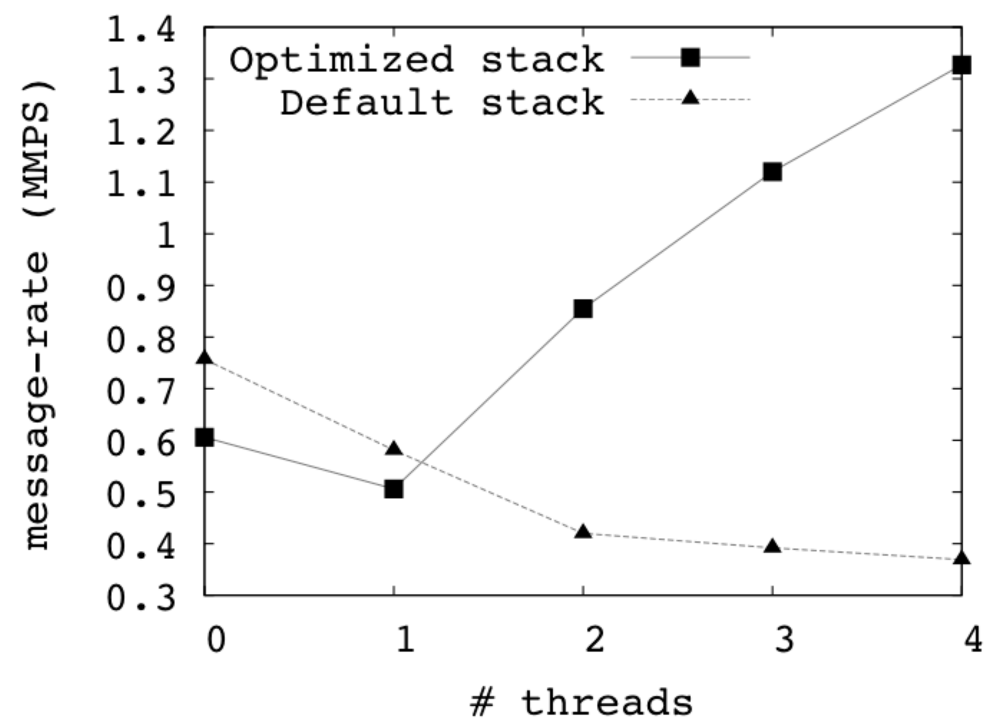
Performance with MPI_THREAD_MULTIPLE

- Thread safety does not come for free
- The implementation must protect certain data structures or parts of code with mutexes or critical sections
- To measure the performance impact, we ran tests to measure communication performance when using multiple threads versus multiple processes
  - For results, see Thakur/Gropp paper: "Test Suite for Evaluating Performance of Multithreaded MPI Communication," *Parallel Computing*, 2009

Message Rate Results on BG/P



Message Rate Benchmark



"Enabling Concurrent Multithreaded MPI Communication on Multicore Petascale Systems" EuroMPI 2010

Why is it hard to optimize MPI_THREAD_MULTIPLE

- MPI internally maintains several resources
- Because of MPI semantics, it is required that all threads have access to some of the data structures
  - E.g., thread 1 can post an Irecv, and thread 2 can wait for its completion – thus the request queue has to be shared between both threads
  - Since multiple threads are accessing this shared queue, it needs to be locked – adds a lot of overhead

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)

Example of Problem with Threads

- Ptolemy is a framework for modeling, simulation, and design of concurrent, real-time, embedded systems
- Developed at UC Berkeley (PI: Ed Lee)
- It is a rigorously tested, widely used piece of software
- Ptolemy II was first released in 2000
- Yet, on April 26, 2004, four years after it was first released, the code deadlocked!
- The bug was lurking for 4 years of widespread use and testing!
- A faster machine or something that changed the timing caught the bug
- See "The Problem with Threads" by Ed Lee, IEEE Computer, 2006

An Example we encountered recently

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

2 Proceses, 2 Threads, Each Thread Executes this Code

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

Intended Ordering of Operations

Rank 0		*Rank 1*	
2 recvs (T1)		2 sends (T1)	
2 sends (T1)		2 recvs (T1)	
2 recvs (T1)		2 sends (T1)	
2 sends (T1)		2 recvs (T1)	
	2 recvs (T2)		2 sends (T2)
	2 sends (T2)		2 recvs (T2)
	2 recvs (T2)		2 sends (T2)
	2 sends (T2)		2 recvs (T2)

- Every send matches a receive on the other rank

Possible Ordering of Operations in Practice

Rank 0		***Rank 1***	
2 recvs (T1)		2 sends (T1)	
2 sends (T1)		1 recv (T1)	
1 recv (T1)			2 sends (T2)
	1 recv (T2)		1 recv (T2)

---

1 recv (T1)	1 recv (T2)	1 recv (T1)	1 recv (T2)

---

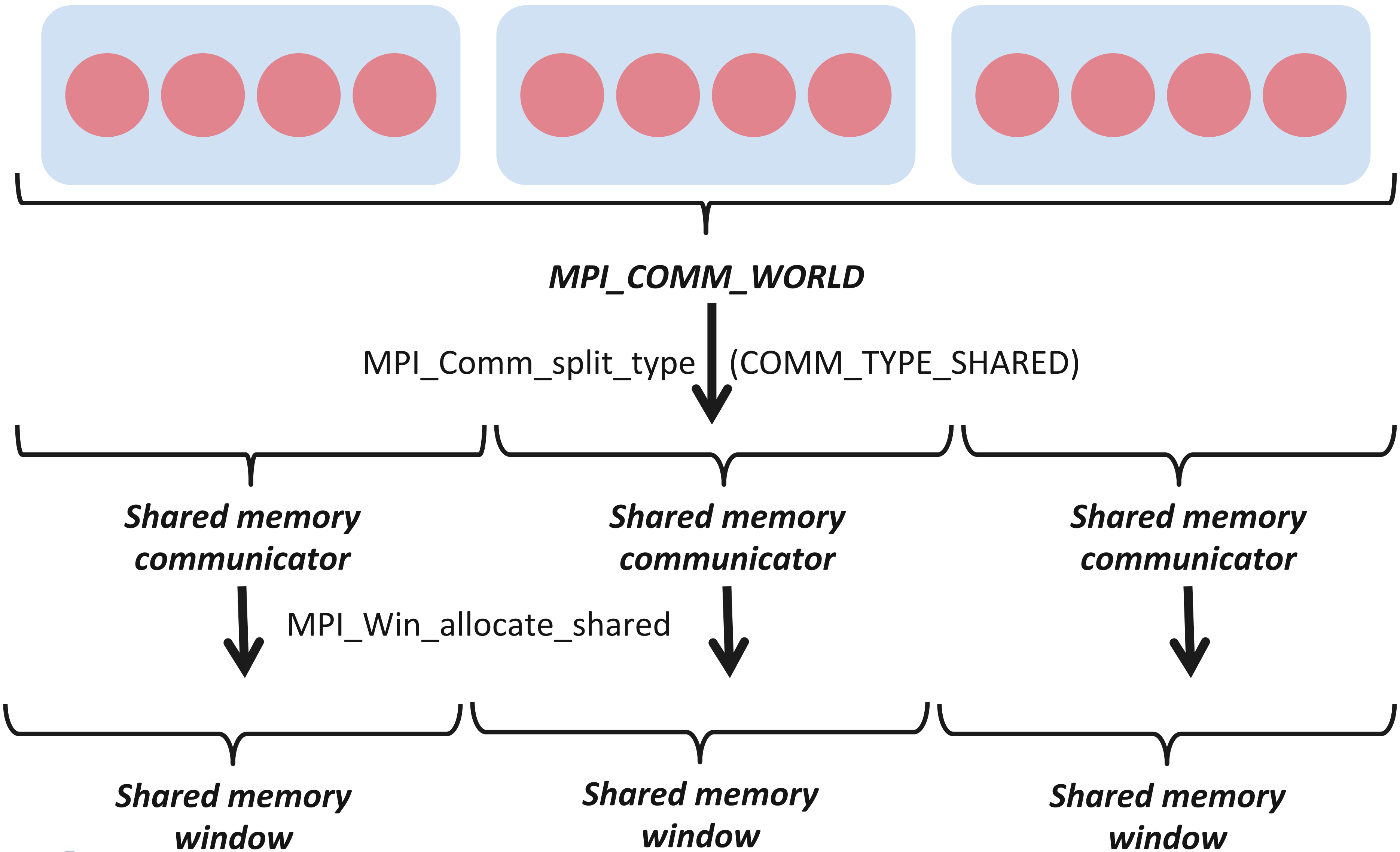
2 sends (T1)	2 sends (T2)	2 sends (T1)	2 sends (T2)
	2 recvs (T2)	2 recvs (T1)	2 recvs (T2)
	2 sends (T2)		

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

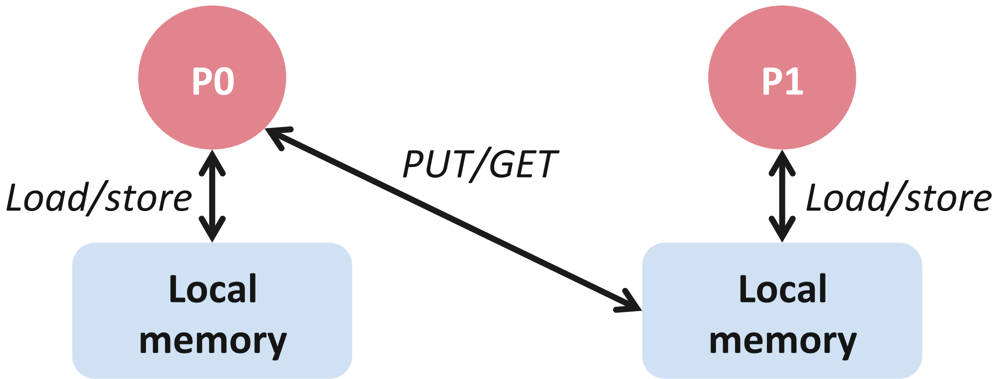
Hybrid Programming with Shared Memory

- 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

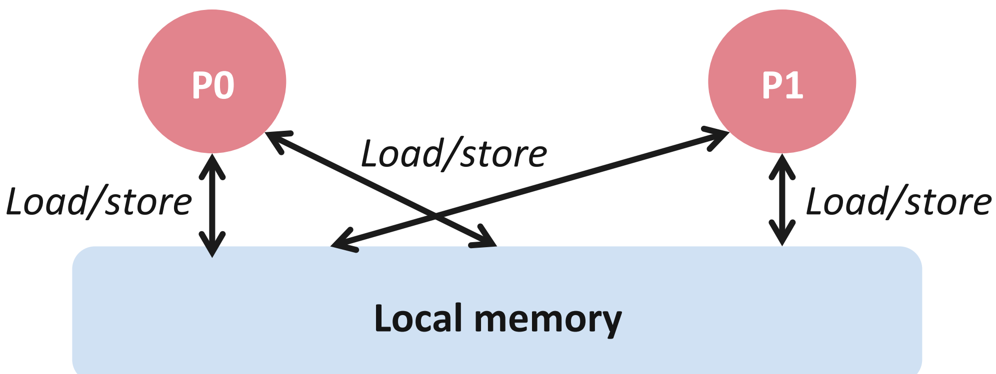
Creating Shared Memory Regions in MPI



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

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

Shared Arrays with Shared memory windows

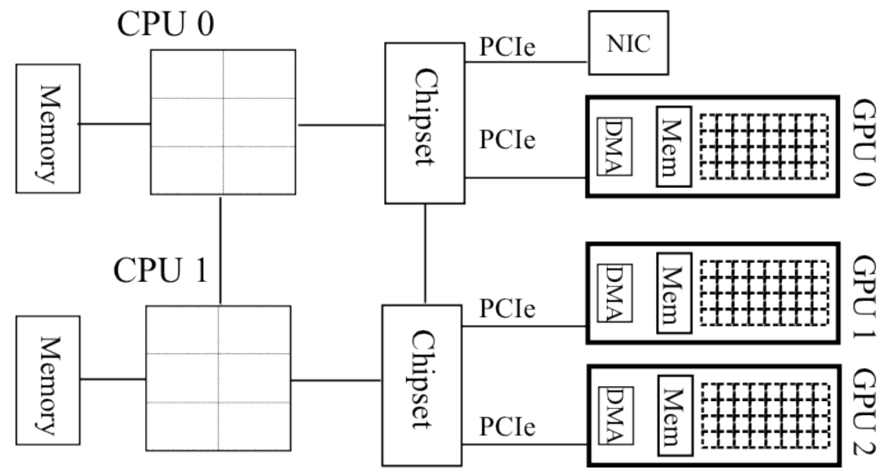
```
int main(int argc, char ** argv)
{
    int buf[100];

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

    MPI_Comm_rank(comm, &rank);

    MPI_Win_lockall(win);

    /* copy data to local part of shared memory */
    MPI_Barrier(comm);

    /* use shared memory */

    MPI_Win_unlock_all(win);

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

Walkthrough of 2D Stencil Code with Shared Memory Windows

- Code can be downloaded from
  www.mcs.anl.gov/~thakur/sc14-mpi-tutorial

Accelerators in Parallel Computing

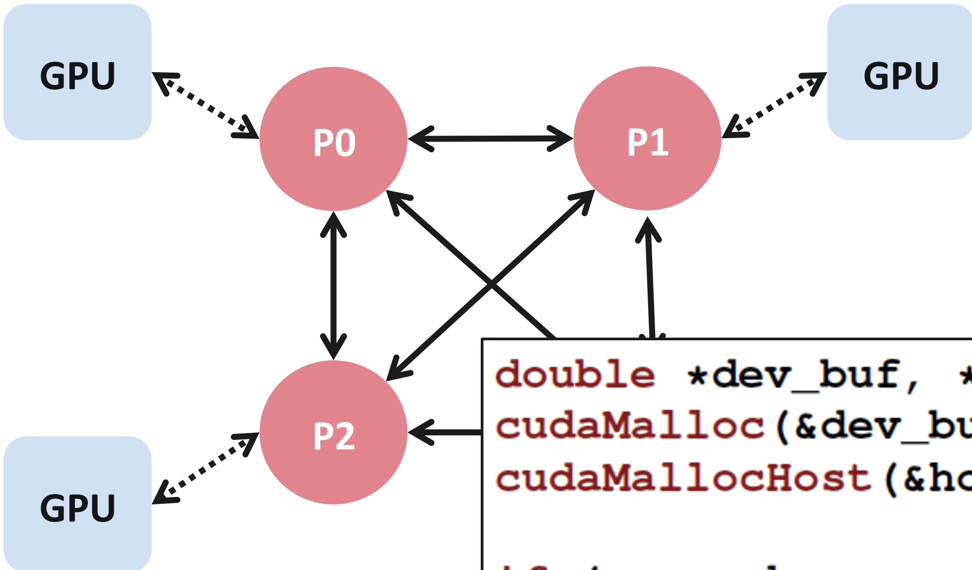
- General purpose, highly parallel processors
  - High FLOPs/Watt and FLOPs/$
  - Unit of execution *Kernel*
  - Separate memory subsystem
  - Prog. Models: CUDA, OpenCL, …
- Clusters with accelerators are becoming common
- New programmability and performance challenges for programming models and runtime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



Hybrid Programming with Accelerators

- Many users are looking to use accelerators within their MPI applications
- The MPI standard does not provide any special semantics to interact with accelerators
  - Current MPI threading semantics are considered sufficient by most users
  - There are some research efforts for making accelerator memory directly accessibly by MPI, but those are not a part of the MPI standard

Current Model for MPI+Accelerator Applications




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

if (my_rank == sender) { /* sender */
  computation_on_GPU(dev_buf);
  cudaMemcpy(host_buf, dev_buf, size, ...);
  MPI_Send(host_buf, size, ...);
} else {                 /* receiver */
  MPI_Recv(host_buf, size, ...);
  cudaMemcpy(dev_buf, host_buf, size, ...);
  computation_on_GPU(dev_buf);
}
```

Alternate MPI+Accelerator models being studied

- Some MPI implementations (MPICH, Open MPI, MVAPICH) are investigating how the MPI implementation can directly send/receive data from accelerators
  - Unified virtual address (UVA) space techniques where all memory (including accelerator memory) is represented with a "void *"
  - Communicator and datatype attribute models where users can inform the MPI implementation of where the data resides
- Clear performance advantages demonstrated in research papers, but these features are not yet a part of the MPI standard (as of MPI-3)
  - Could be incorporated in a future version of the standard

Advanced Topics: Nonblocking Collectives, Topologies, and Neighborhood Collectives

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 (more than the sum of the parts ☺)
  - System noise/imbalance resiliency
  - Semantic advantages
  - Examples

Nonblocking Collective Communication

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

Hoefler et al.: Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI

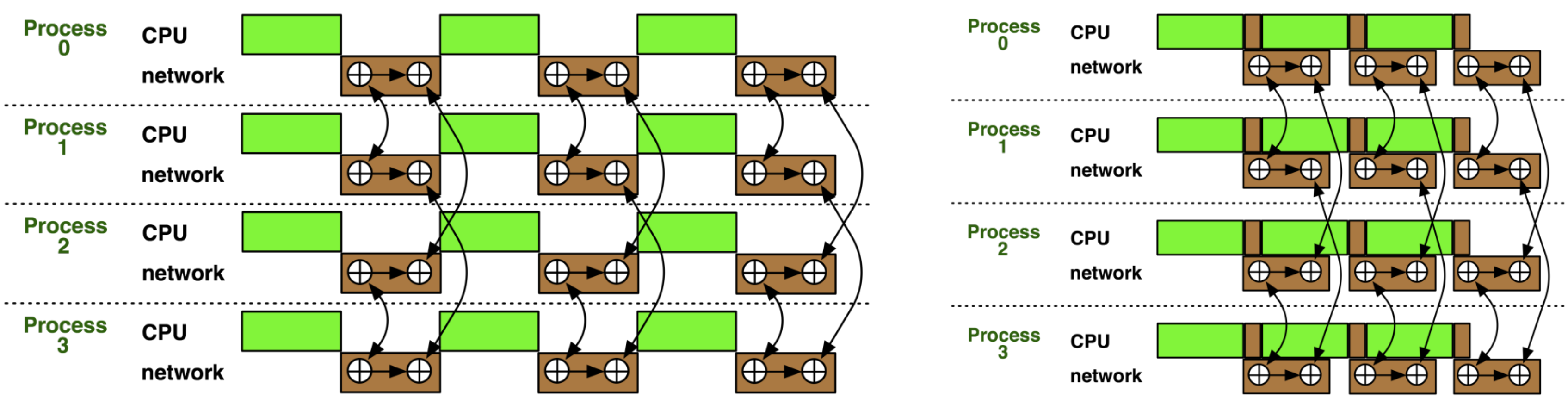
Nonblocking Collective Communication

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

Hoefler et al.: Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI

Nonblocking Collectives Overlap

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



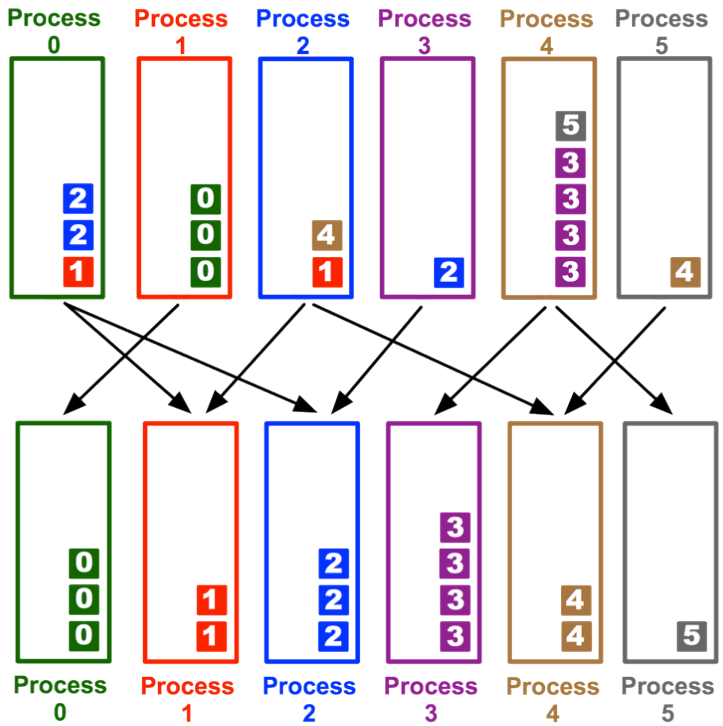
Hoefler: Leveraging Non-blocking Collective Communication in High-performance Applications

A Non-Blocking Barrier?

- What can that be good for? Well, quite a bit!
- Semantics:
  - MPI_Ibarrier() – calling process entered the barrier, **no** synchronization happens
  - Synchronization **may** happen asynchronously
  - MPI_Test/Wait() – synchronization happens **if** necessary
- Uses:
  - Overlap barrier latency (small benefit)
  - Use the split semantics! Processes **notify** non-collectively but **synchronize** collectively!

A Semantics Example: DSDE

- Dynamic Sparse Data Exchange
  - Dynamic: comm. pattern varies across iterations
  - Sparse: number of neighbors is limited ($\mathcal{O}(\log P)$)
  - Data exchange: only senders know neighbors



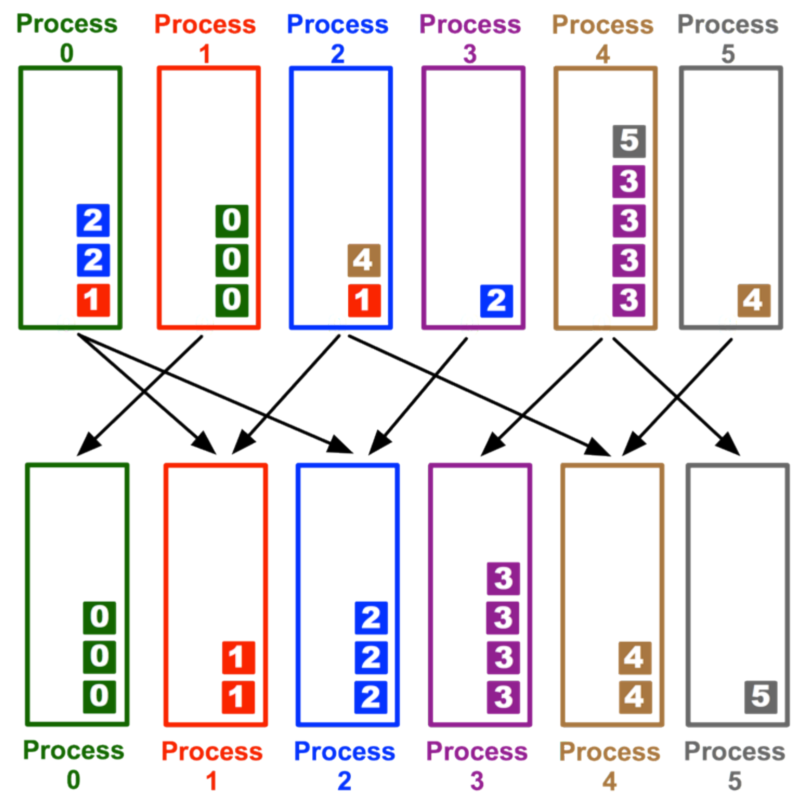
Hoefler et al.:Scalable Communication Protocols for Dynamic Sparse Data Exchange

Dynamic Sparse Data Exchange (DSDE)

- Main Problem: metadata
  - Determine who wants to send how much data to me (I must post receive and reserve memory)

OR:

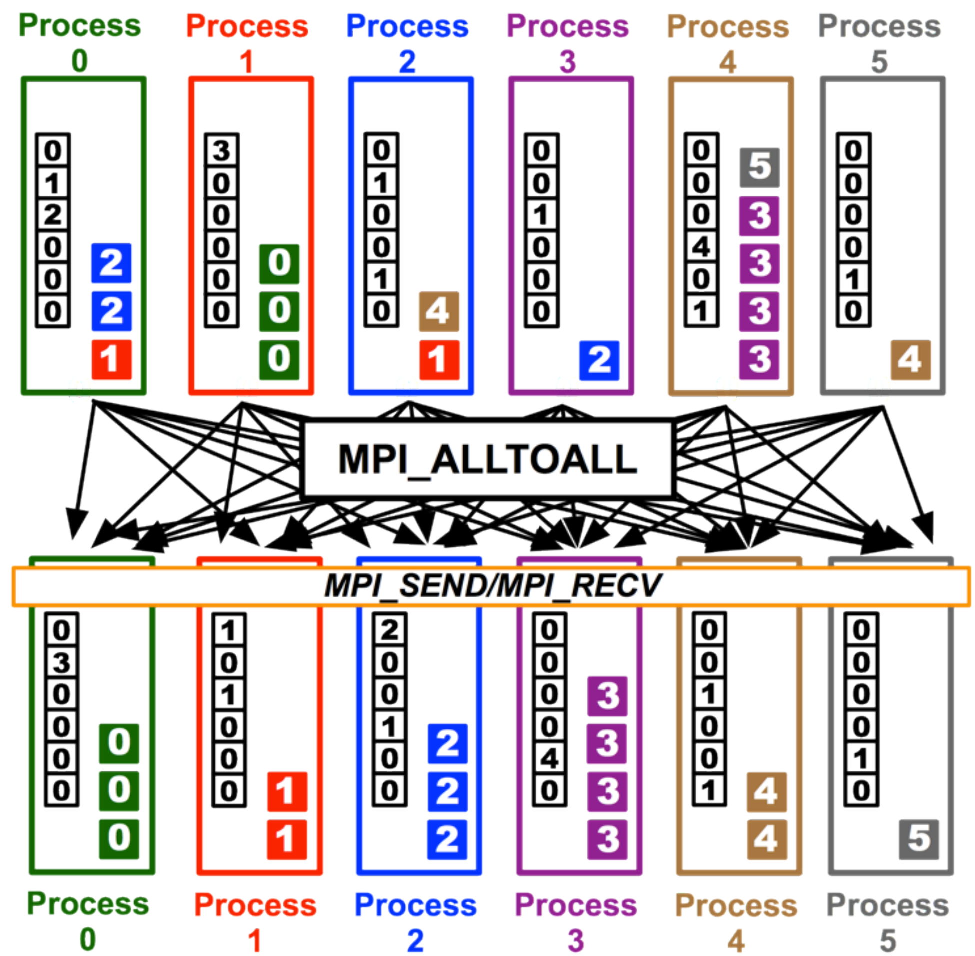
  - Use MPI semantics:
    - Unknown sender
      - MPI_ANY_SOURCE
    - Unknown message size
      - MPI_PROBE
    - Reduces problem to counting the number of neighbors
    - Allow faster implementation!



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

Using Alltoall (PEX)

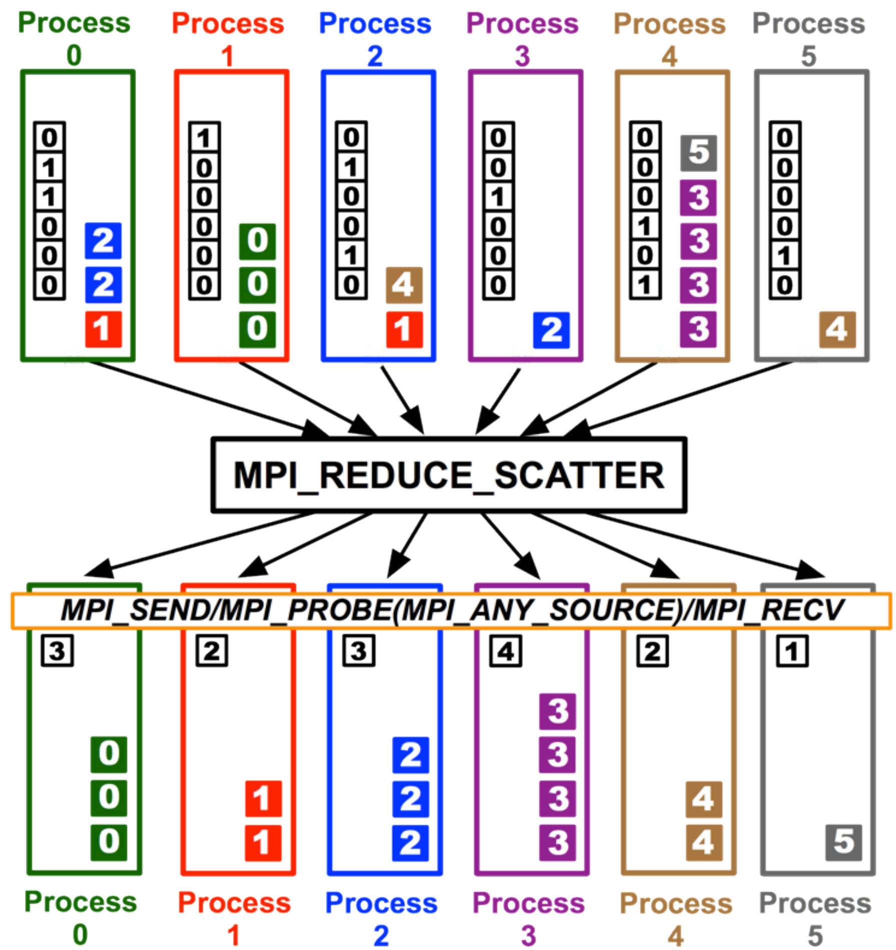
- Based on Personalized Exchange ($\Theta(P)$)
  - Processes exchange metadata (sizes) about neighborhoods with all-to-all
  - Processes post receives afterwards
  - Most intuitive but least performance and scalability!



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

Reduce_scatter (PCX)

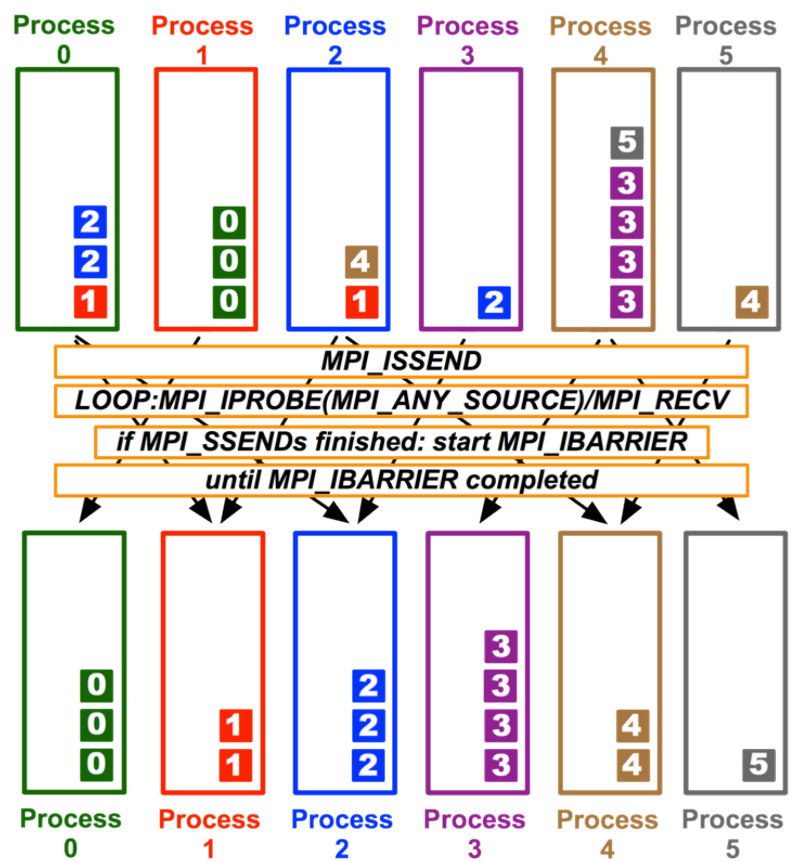
- Bases on Personalized Census ($\Theta(P)$)
  - Processes exchange metadata (counts) about neighborhoods with reduce_scatter
  - Receivers checks with wildcard MPI_IPROBE and receives messages
  - Better than PEX but non-deterministic!



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

MPI_Ibarrier (NBX)

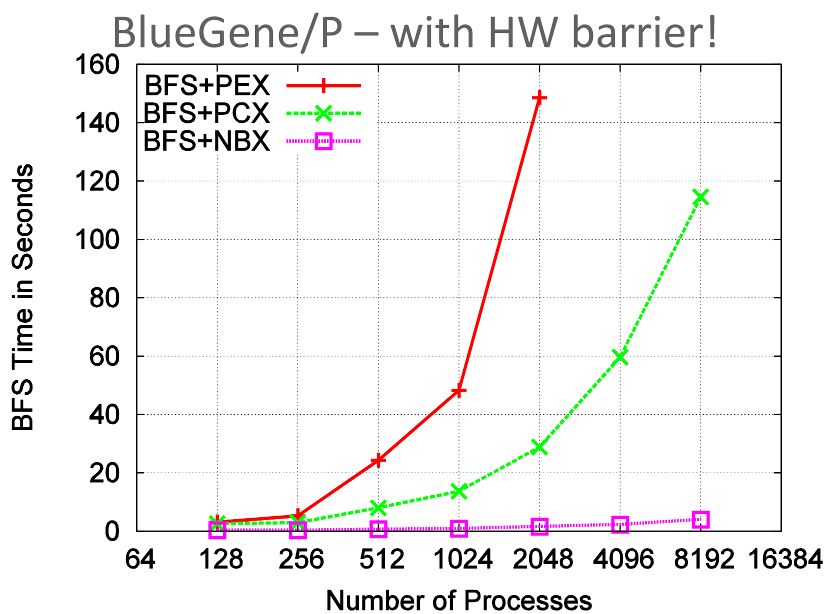
- Complexity - census (barrier): ($\Theta(\log(P))$)
  - Combines metadata with actual transmission
  - Point-to-point synchronization
  - Continue receiving until barrier completes
  - Processes start coll. synch. (barrier) when p2p phase ended
    - barrier = distributed marker!
  - Better than PEX, PCX, RSX!

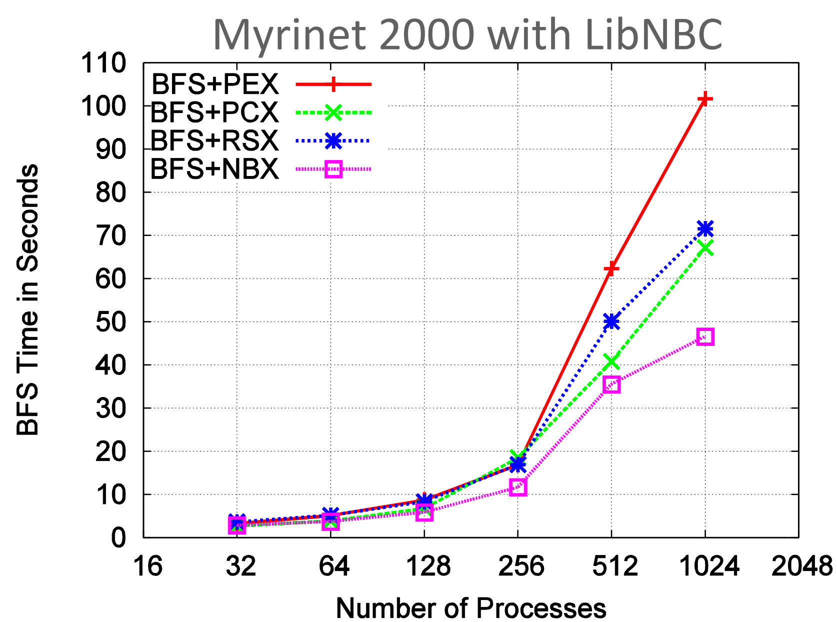


T. Hoefler et al.:Scalable Communication Protocols for Dynamic Sparse Data Exchange

Parallel Breadth First Search

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



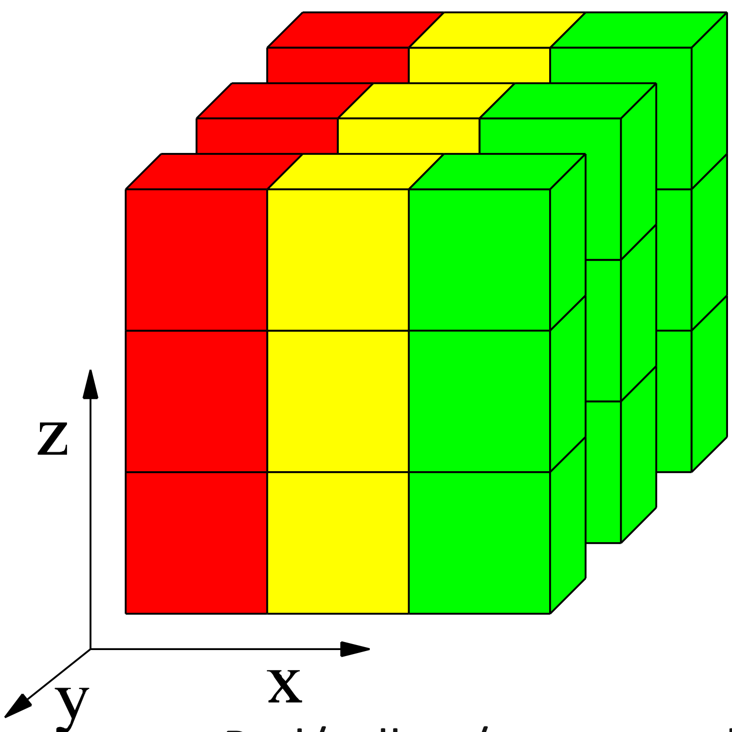


- HW barrier support is significant at large scale!

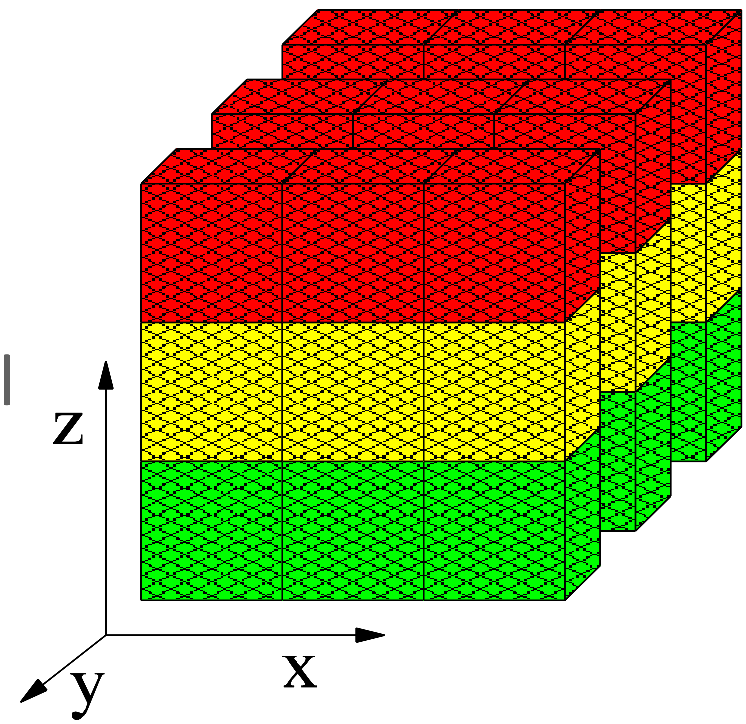
T. Hoefler et al.: Scalable Communication Protocols for Dynamic Sparse Data Exchange

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



→ Alltoall



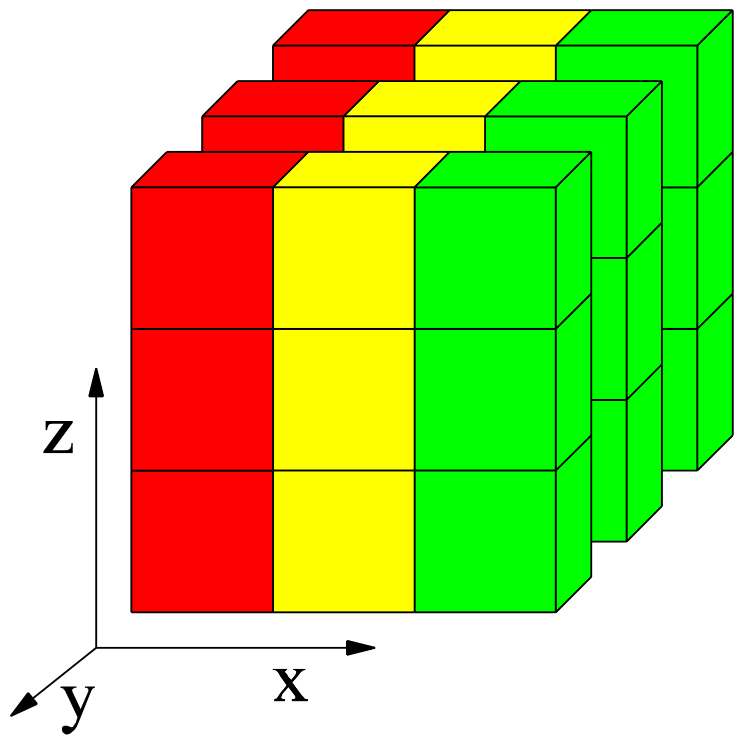
  - Red/yellow/green are the (three) different processes!

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
```

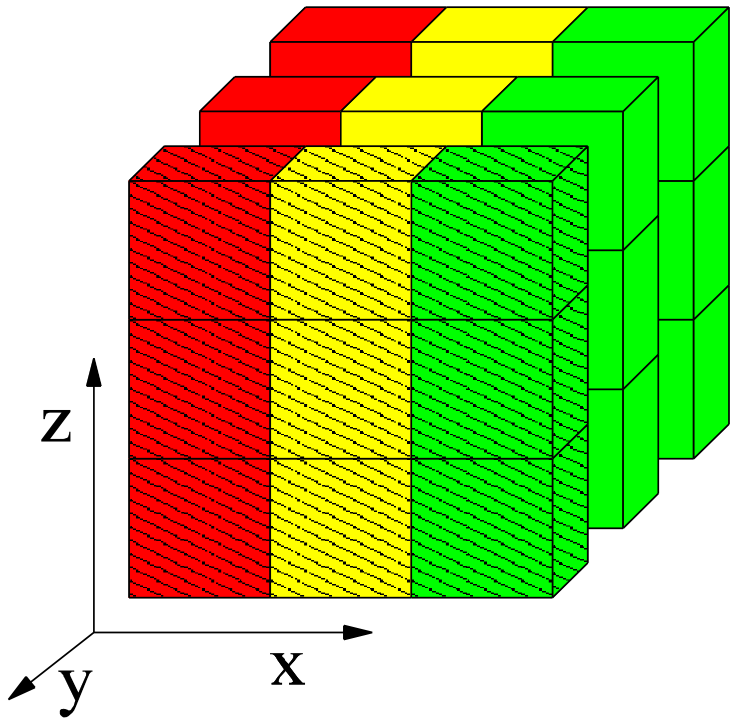
Parallel Fast Fourier Transform

- Data already transformed in y-direction



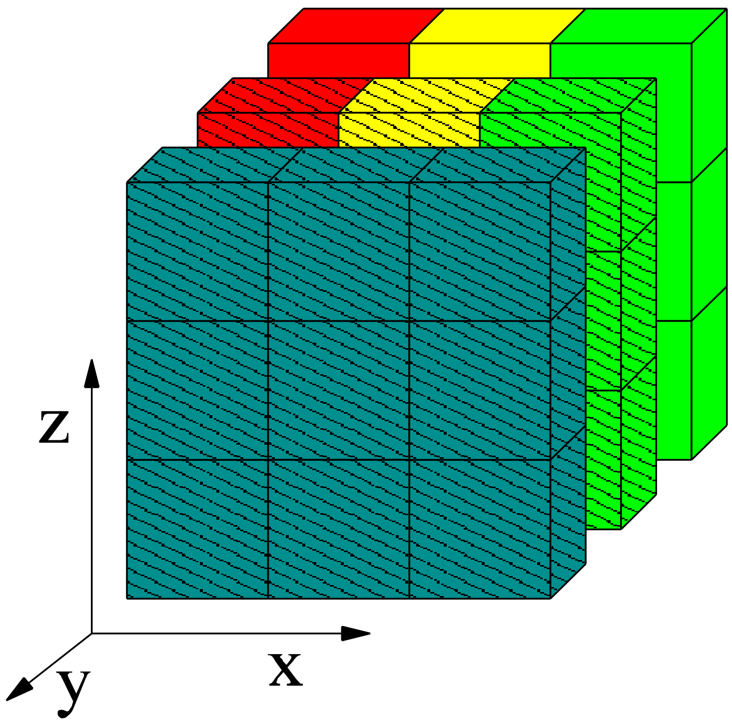
Parallel Fast Fourier Transform

- Transform first y plane in z



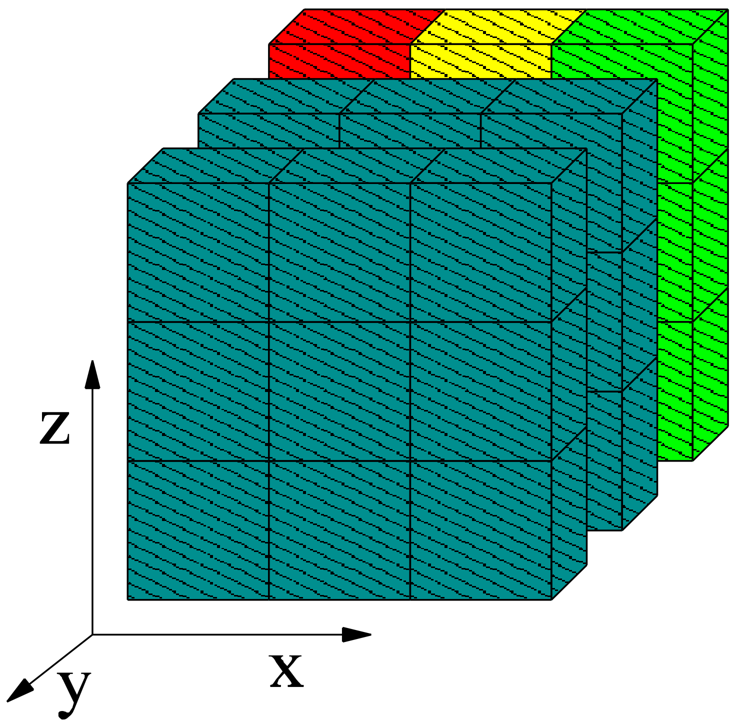
Parallel Fast Fourier Transform

- Start ialltoall and transform second plane



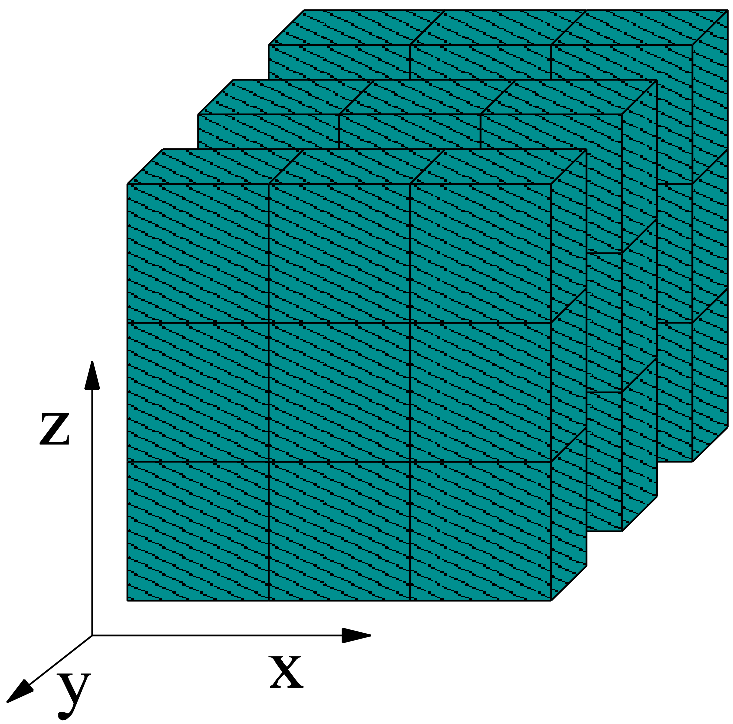
Parallel Fast Fourier Transform

- Start ialltoall (second plane) and transform third



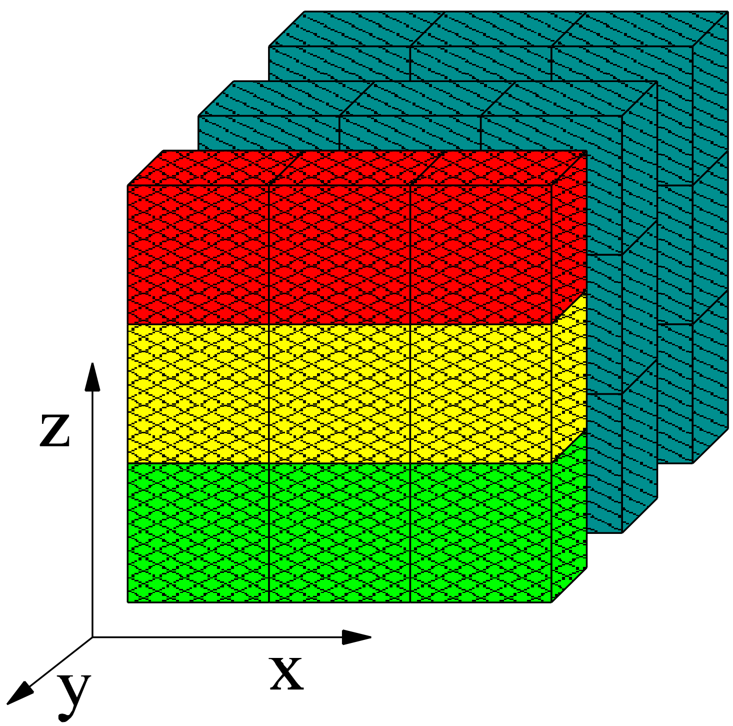
Parallel Fast Fourier Transform

- Start ialltoall of third plane and …



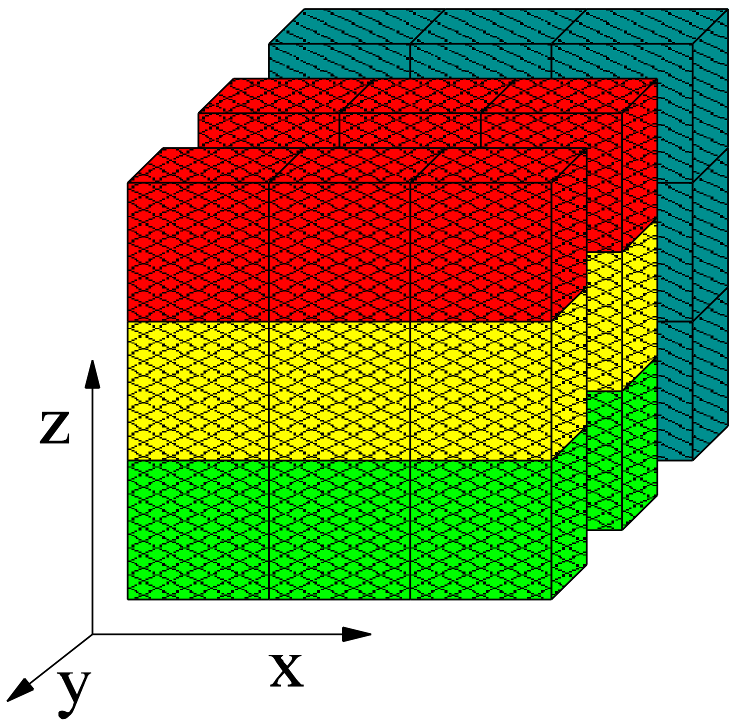
Parallel Fast Fourier Transform

- Finish ialltoall of first plane, start x transform



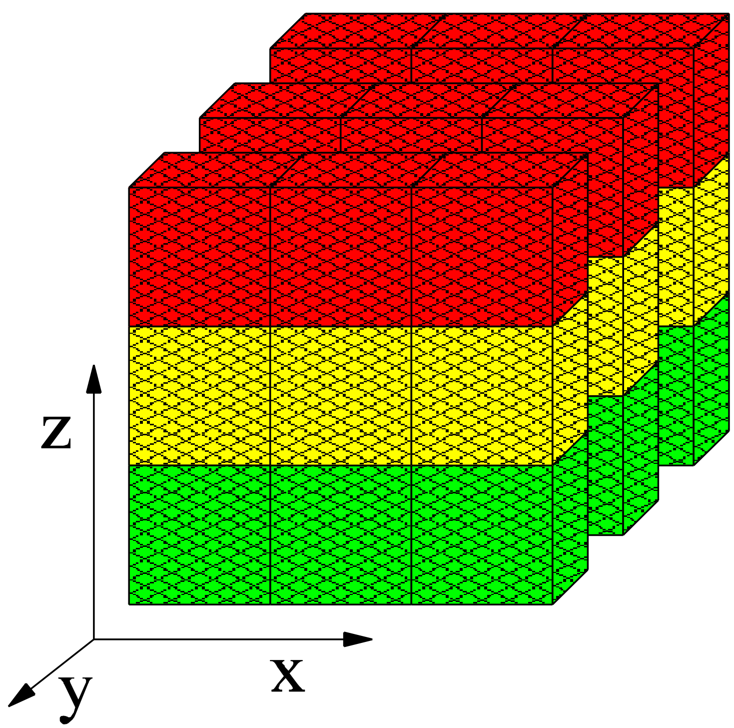
Parallel Fast Fourier Transform

- 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
```

Hoefler: Leveraging Non-blocking Collective Communication in High-performance Applications

Nonblocking And Collective Summary

- Nonblocking comm does two things:
  - Overlap and relax synchronization
- Collective comm does one thing
  - Specialized pre-optimized routines
  - Performance portability
  - Hopefully transparent performance
- They can be composed
  - E.g., software pipelining

Topologies and Topology Mapping

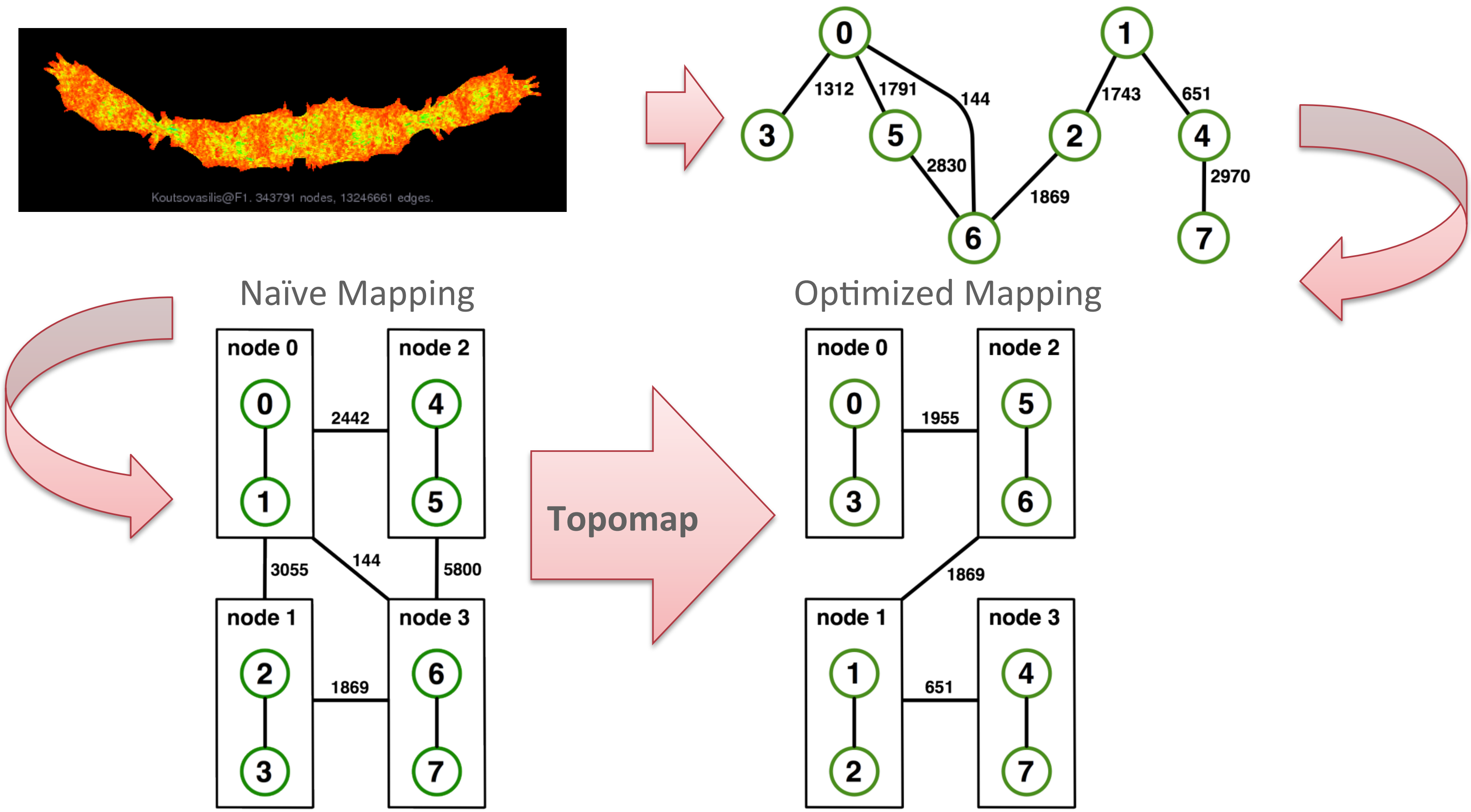
Topology Mapping and Neighborhood Collectives

- 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

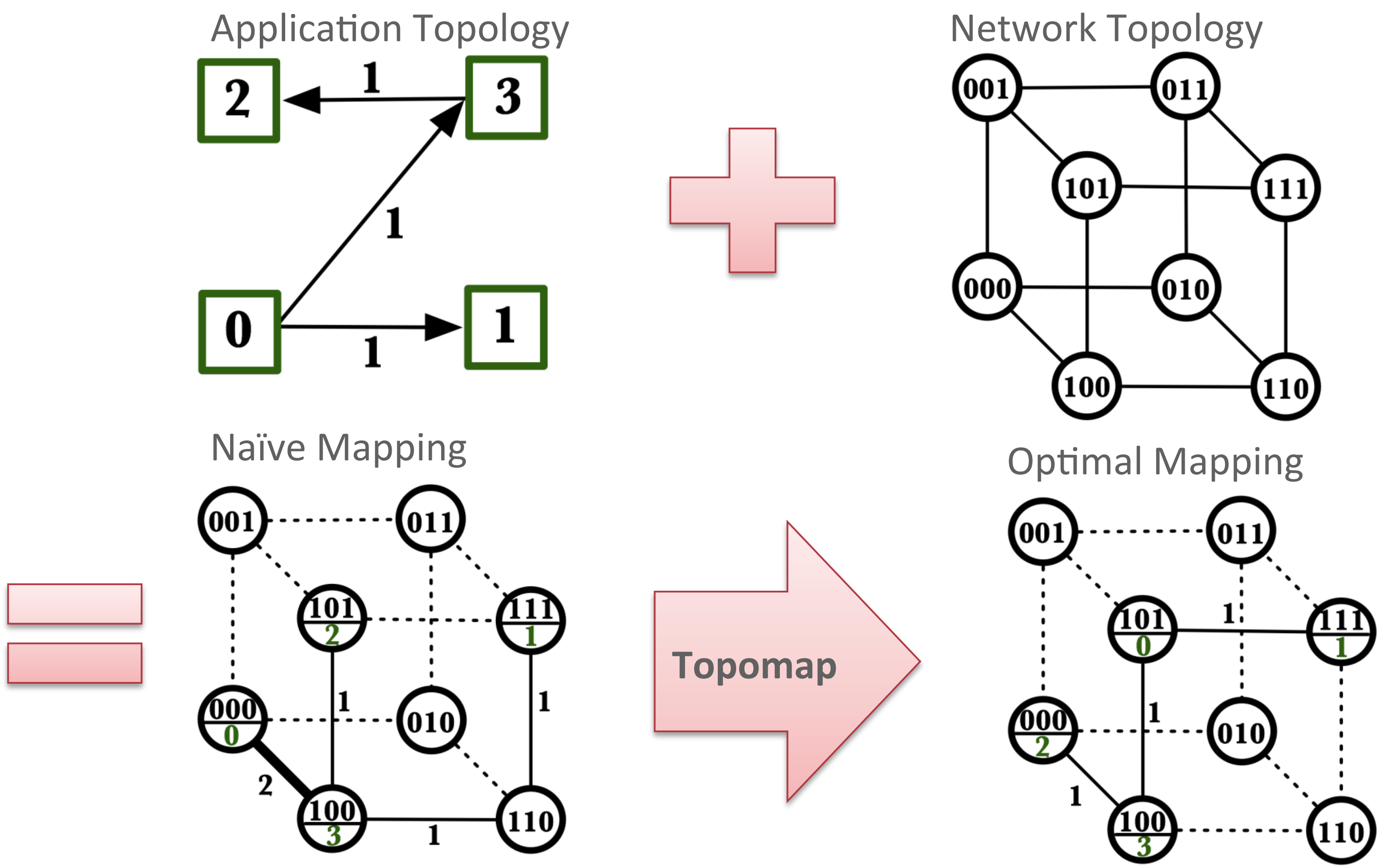
- MPI supports rank reordering
  - Change numbering in a given allocation to reduce congestion or dilation
  - Sometimes automatic (early IBM SP machines)
- Properties
  - Always possible, but effect may be limited (e.g., in a bad allocation)
  - Portable way: MPI process topologies
    - Network topology 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



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 3-d Torus of size 5x5x5
- 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

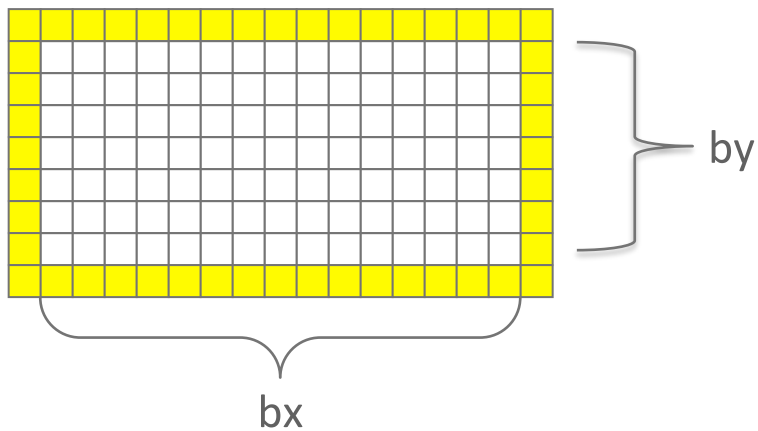
Cartesian Communication Helpers

```
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

Code Example

- *stencil-mpi-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

Hoefler et al.: The Scalable Process Topology Interface of MPI 2.2

MPI_Dist_graph_create_adjacent

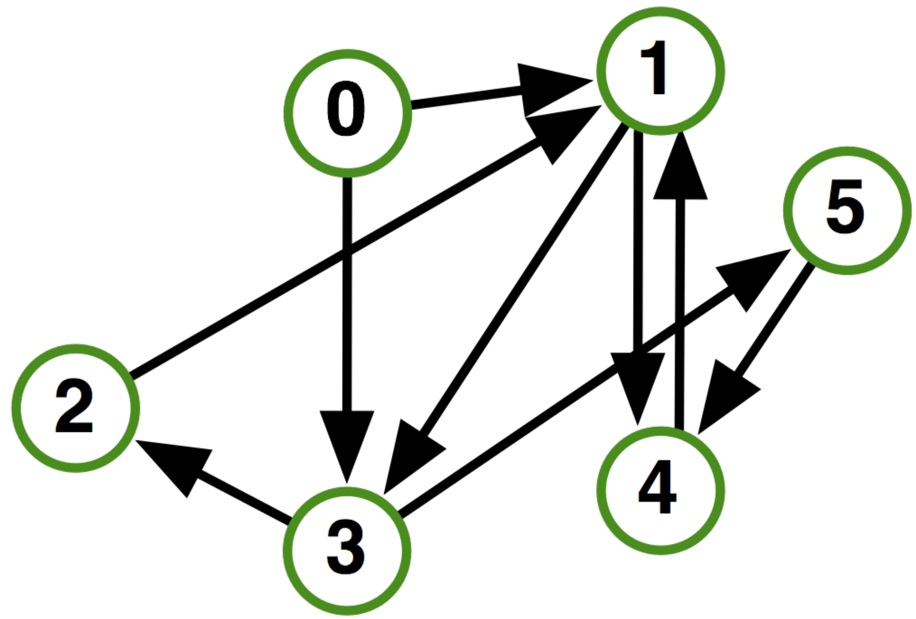
```
MPI_Dist_graph_create_adjacent(MPI_Comm comm_old,
        int indegree, const int sources[], const int sourceweights[],
        int outdegree, const int destinations[],
        const int destweights[], MPI_Info info, int reorder,
        MPI_Comm *comm_dist_graph)
```

- 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)

Hoefler et al.: The Scalable Process Topology Interface of MPI 2.2

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}
- …



Hoefler et al.: The Scalable Process Topology Interface of MPI 2.2

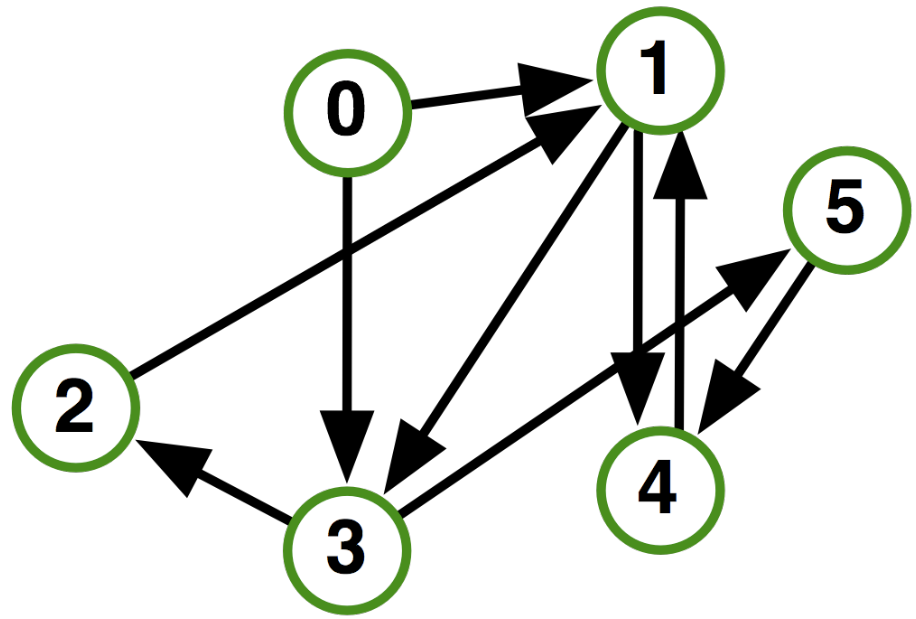
MPI_Dist_graph_create

```
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

Hoefler et al.: The Scalable Process Topology Interface of MPI 2.2

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

Hoefler et al.: The Scalable Process Topology Interface of MPI 2.2

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 to write libraries on top of MPI topologies!

Neighborhood Collectives

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!

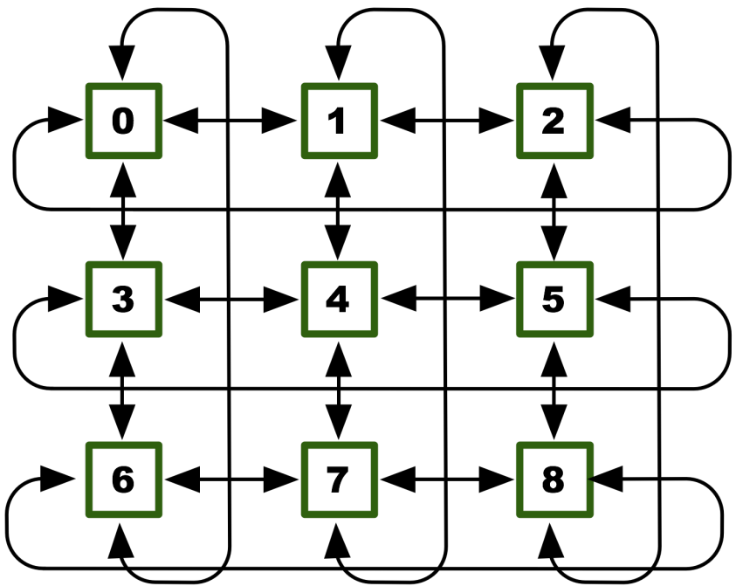
Cartesian Neighborhood Collectives

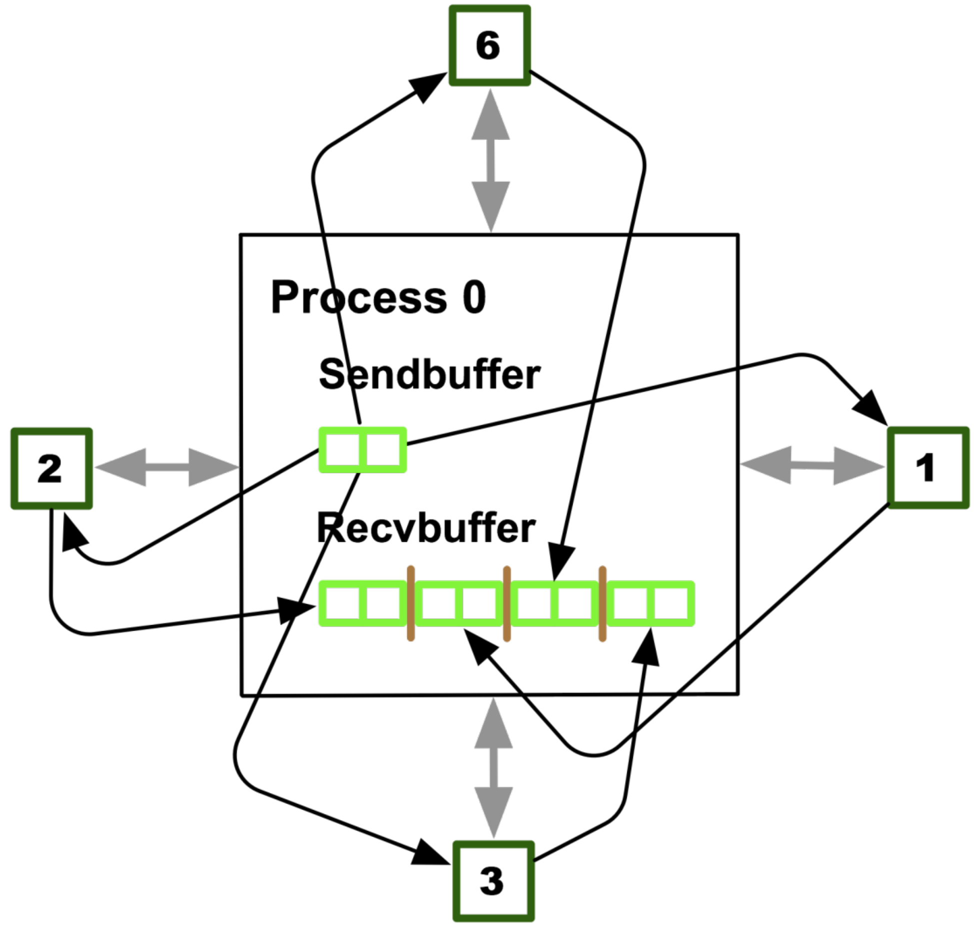
- Communicate with direct neighbors in Cartesian topology
  - Corresponds to cart_shift with disp=1
  - Collective (all processes in comm must call it, including processes without neighbors)
  - Buffers are laid out as neighbor sequence:
    - Defined by order of dimensions, first negative, then positive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

T. Hoefler and J. L. Traeff: Sparse Collective Operations for MPI

Cartesian Neighborhood Collectives

- Buffer ordering example:





T. Hoefler and J. L. Traeff: Sparse Collective Operations for MPI

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!

T. Hoefler and J. L. Traeff: Sparse Collective Operations for MPI

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!

Walkthrough of 2D Stencil Code with Neighborhood Collectives

- Code can be downloaded from
  www.mcs.anl.gov/~thakur/sc14-mpi-tutorial

Why is Neighborhood Reduce Missing?

```
MPI_Ineighbor_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

T. Hoefler and J. L. Traeff: Sparse Collective Operations for MPI

Topology Summary

- Topology functions allow to specify application communication patterns/topology
  – Convenience functions (e.g., Cartesian)
  – Storing neighborhood relations (Graph)
- Enables topology mapping (reorder=1)
  – Not widely implemented yet
  – May requires manual data re-distribution (according to new rank order)
- MPI does not expose information about the network topology (would be very complex)

Neighborhood Collectives Summary

- Neighborhood collectives add communication functions to process topologies
  - Collective optimization potential!
- Allgather
  - One item to all neighbors
- Alltoall
  - Personalized item to each neighbor
- High optimization potential (similar to collective operations)
  - Interface encourages use of topology mapping!

Section Summary

- Process topologies enable:
  - High-abstraction to specify communication pattern
  - Has to be relatively static (temporal locality)
    - Creation is expensive (collective)
  - Offers basic communication functions
- Library can optimize:
  - Communication schedule for neighborhood colls
  - Topology mapping

Recent Efforts of the MPI Forum for MPI-3.1, MPI-4, and Future MPI Standards

Introduction

- The MPI Forum continues to meet once every 3 months to define future versions of the MPI Standard
  - The next Forum meeting is December 8-11, 2014, in San Jose
- We describe some of the proposals the Forum is currently considering

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
  - Lots of other details in the proposal…

Better Hybrid Programming: Extending MPI to Support Multiple Endpoints Per Process

- In MPI today, each process has a single communication endpoint (rank in MPI_COMM_WORLD)
- Multiple threads of a process communicate through that single endpoint, requiring the implementation to use locks etc., which are expensive
- MPI Forum is discussing a proposal (for MPI-4) that allows a process to have multiple endpoints
- Threads within a process can attach to different endpoints and communicate through those endpoints as if they are separate ranks
- The MPI implementation can avoid using locks if each thread communicates on a separate endpoint
- This allows the MPI standard to support "MPI + X" more efficiently without specifying what X is

Other concepts being considered

- MPI Streams interface
  - Streaming data between sender and receiver
- Nonblocking File Manipulation routines
  - Nonblocking versions of file open, close, set_view, etc.
- Active Messages
  - Initiate operations on remote processes
  - Possibly as an addition to MPI RMA
- Tools Interface
  - Scalable process acquisition interface
  - Introspection of MPI handles

Concluding Remarks

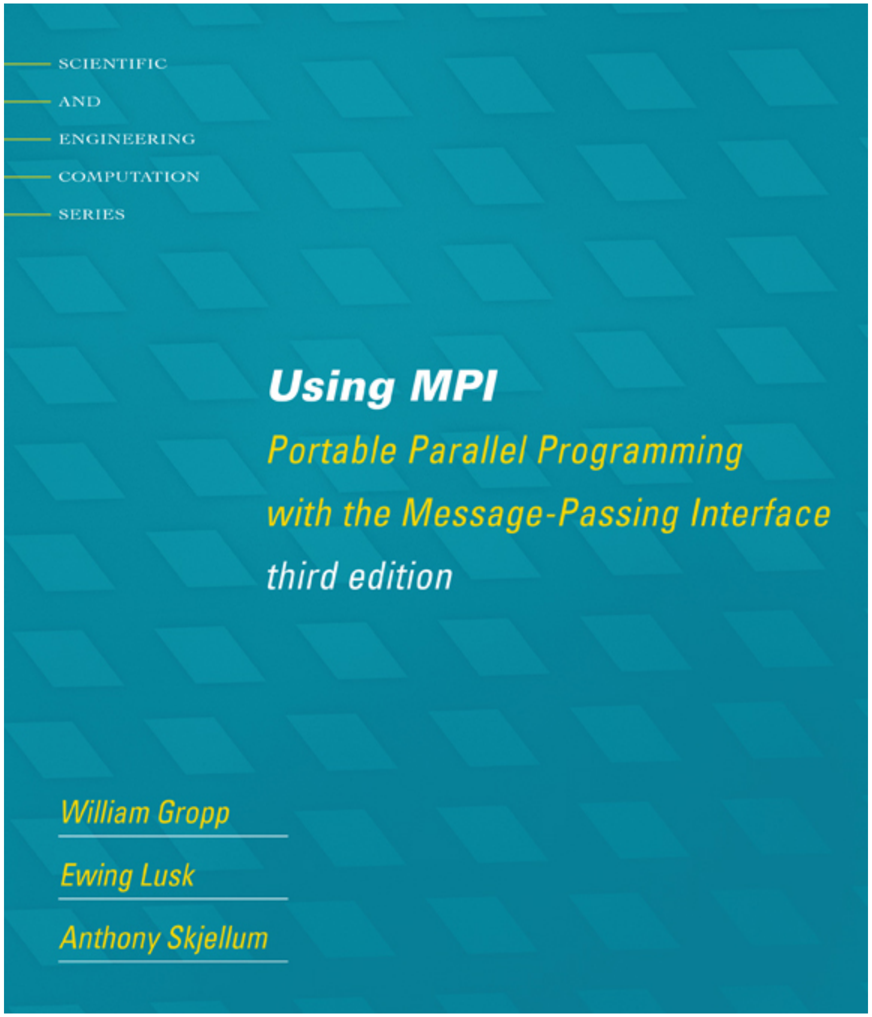
Conclusions

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

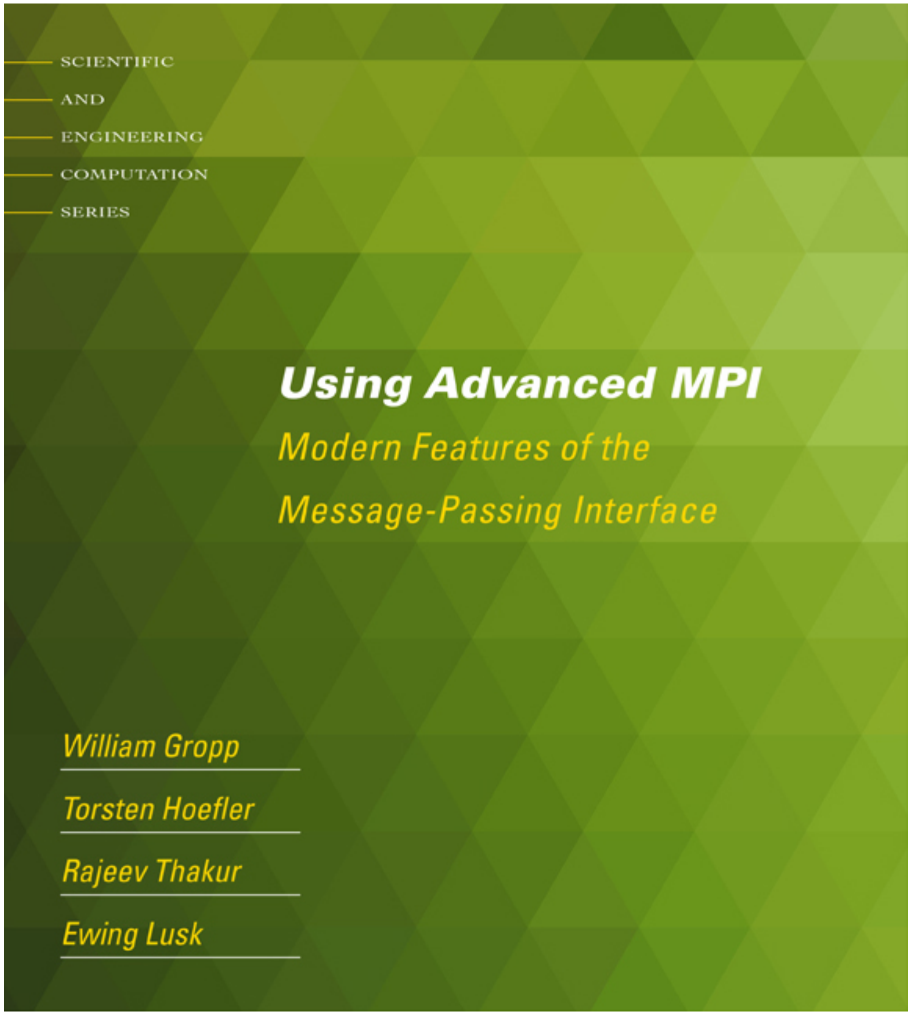
Web Pointers

- MPI standard : http://www.mpi-forum.org/docs/docs.html
- MPI Forum : http://www.mpi-forum.org/
- MPI implementations:
  - MPICH : http://www.mpich.org
  - MVAPICH : http://mvapich.cse.ohio-state.edu/
  - Intel MPI: http://software.intel.com/en-us/intel-mpi-library/
  - Microsoft MPI: 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

New Tutorial Books on MPI



Basic MPI



Advanced MPI, including MPI-3