Advanced MPI

Slides are available at

https://anl.box.com/v/balaji-tutorials-2016

Pavan Balaji

Argonne National Laboratory

Email: balaji@mcs.anl.gov

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

Torsten Hoefler

ETH, Zurich

Email: htor@inf.ethz.ch

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

About the Speakers

- **Pavan Balaji**: Computer Scientist, MCS, Argonne
  - Group Lead: programming models and runtime systems
  - Leads the MPICH implementation of MPI
  - Chairs the Hybrid working group for MPI-3 and MPI-4
  - Member of various other working groups including RMA, contexts and communicators, etc., for MPI-3 and MPI-4
- **Torsten Hoefler**: Assistant Professor, ETH, Zurich
  - Chairs the Collectives working group for MPI-3 and MPI-4
  - Member of various other working groups including RMA, hybrid programming, etc., for MPI-3 and MPI-4
- 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
- Nonblocking Collective Communication
  - MPI-3
- Hybrid Programming with Threads and Shared Memory
  - MPI-2 and MPI-3
- Topology-aware Communication
  - MPI-1 and MPI-2.2

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

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 recently released with some corrections to the standard and small features
- MPI-3 (2012) added several new features to MPI
- MPI-3.1 (2015) added 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

Status of MPI-3.1 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	MPC	NEC MPI
NBC	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	(*)	✓	✓
Nbrhood collectives	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓
RMA	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	Q2'17	✓
Shared memory	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	*	✓
Tools Interface	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	*	Q4'16	✓
Comm-creat group	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	*	✓
F08 Bindings	✓	✓	✓	✓	✓	✗	✓	✗	✗	✓	✗	✗	Q2'16	✓
New Datatypes	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓
Large Counts	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	Q2'16	✓
Matched Probe	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	Q2'16	✓
NBC I/O	✓	Q3'16	✓	✓	✗	✗	✗	✗	✗	✓	✗	✗	Q4'16	✓

Release dates are estimates and are subject to change at any time.

"✗" indicates no publicly announced plan to implement/support that feature.

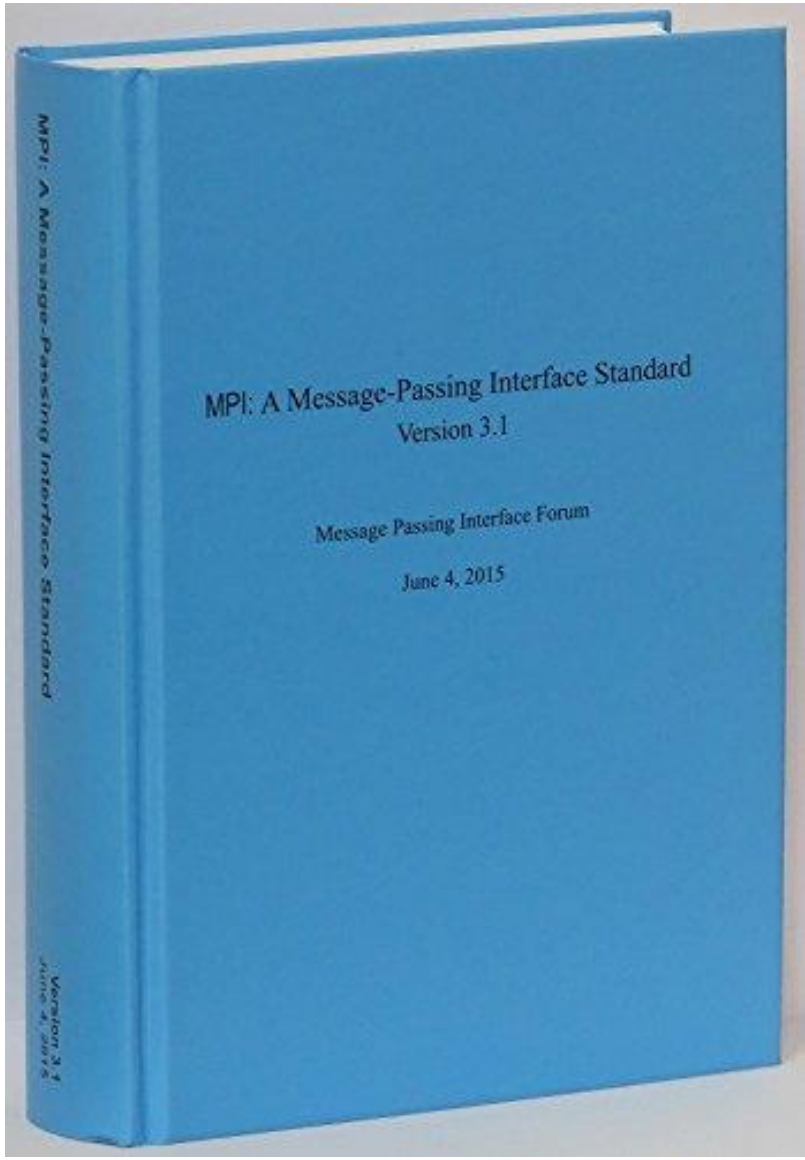
Platform-specific restrictions might apply to the supported features

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

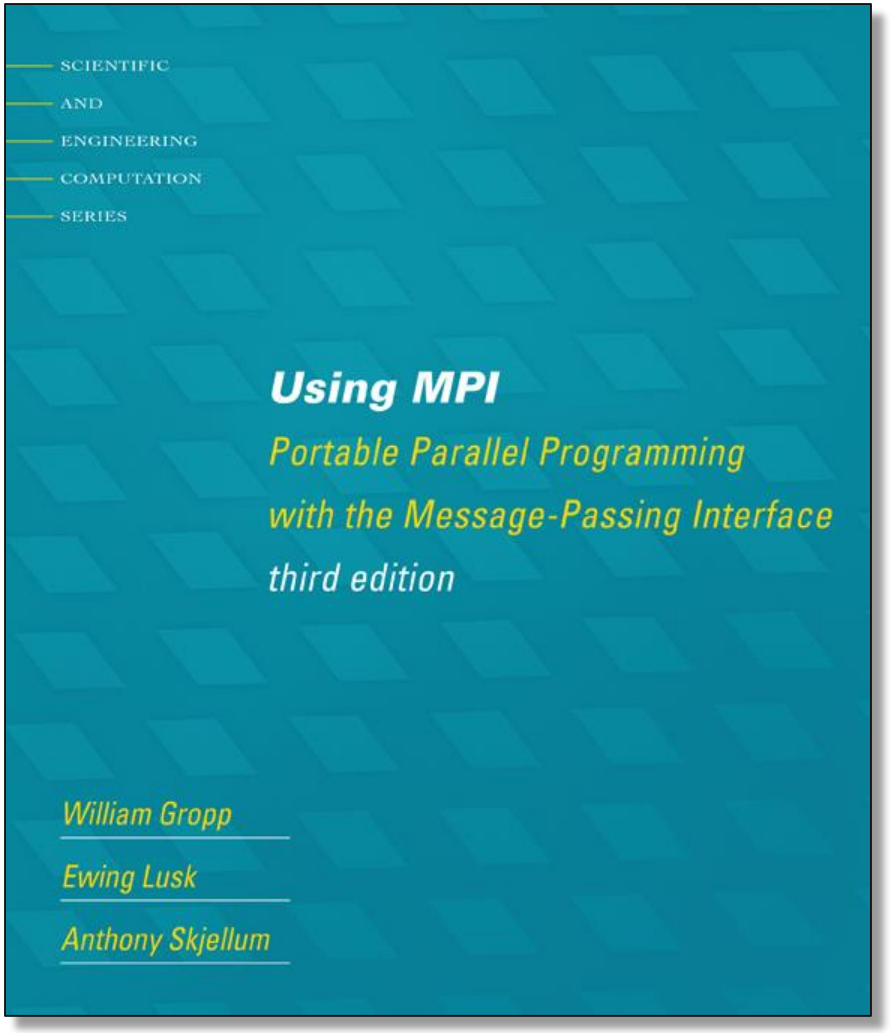
Latest MPI 3.1 Standard in Book Form

Available from amazon.com

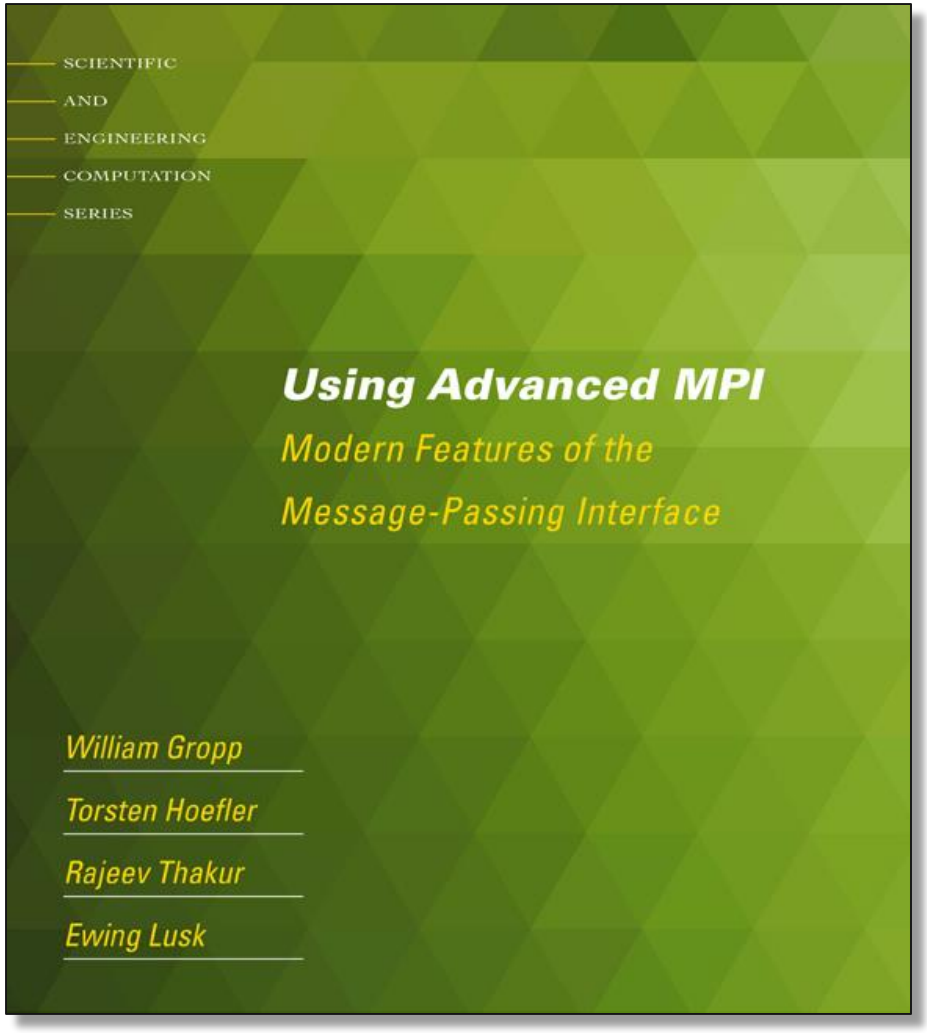
http://www.amazon.com/dp/B015CJ42CU/



New Tutorial Books on MPI



Basic MPI

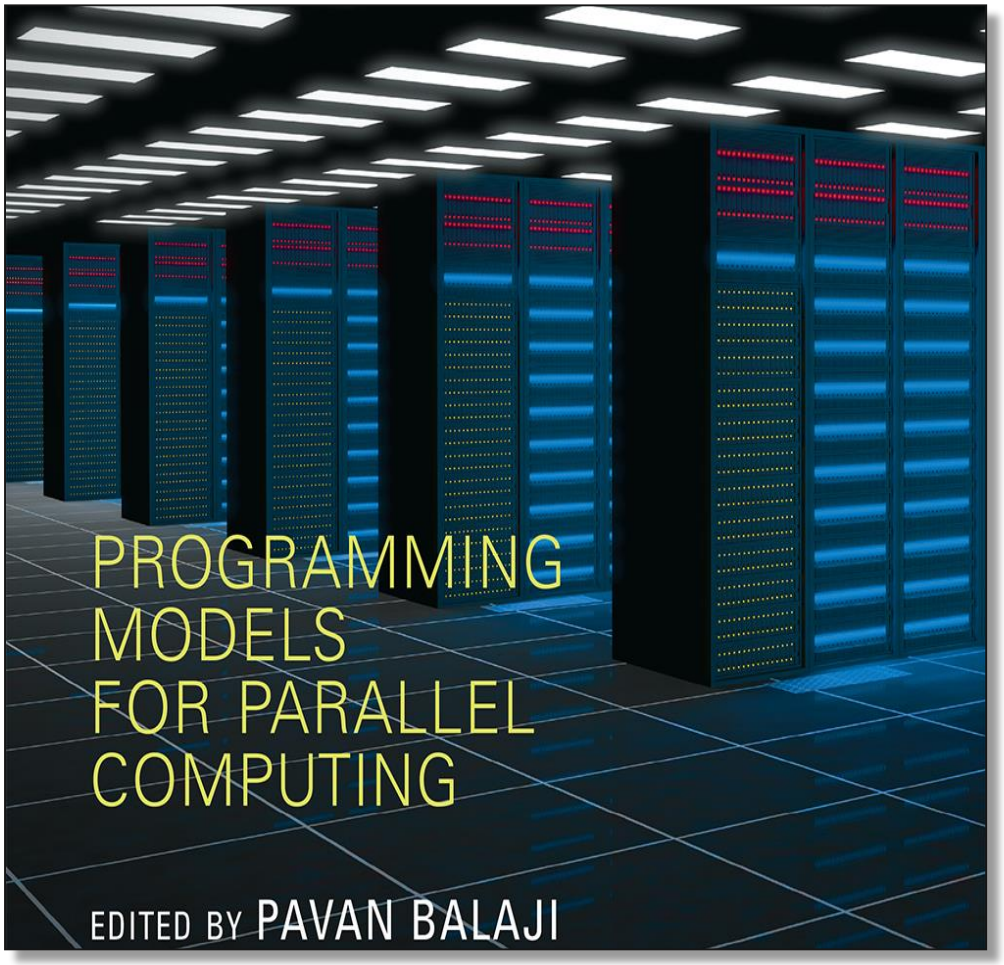


Advanced MPI, including MPI-3

New Book on Parallel Programming Models

Edited by Pavan Balaji

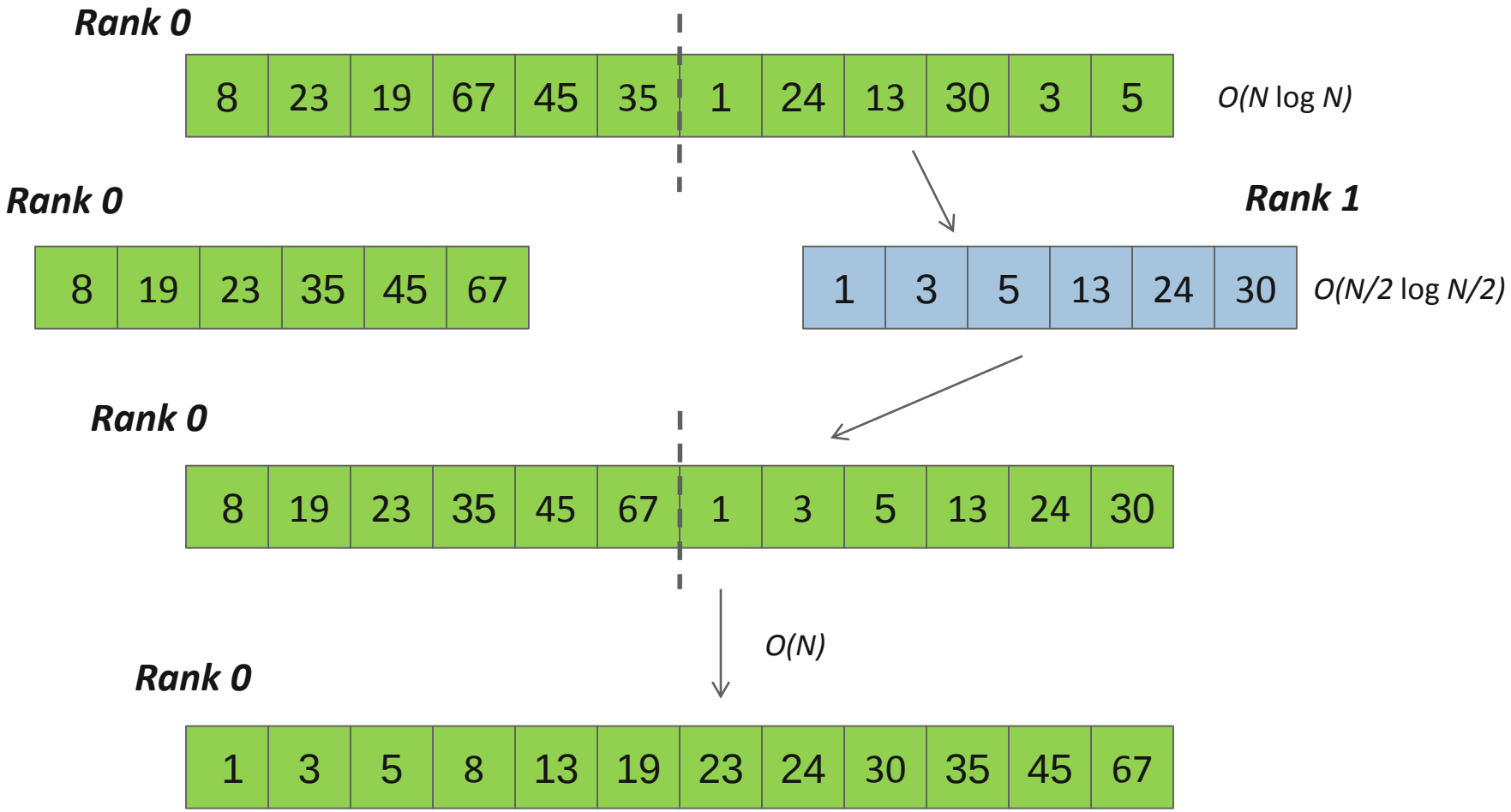
- ***MPI:*** W. Gropp and R. Thakur
- ***GASNet:*** P. Hargrove
- ***OpenSHMEM:*** J. Kuehn and S. Poole
- ***UPC:*** K. Yelick and Y. Zheng
- ***Global Arrays:*** S. Krishnamoorthy, J. Daily, A. Vishnu, and B. Palmer
- ***Chapel:*** B. Chamberlain
- ***Charm++:*** L. Kale, N. Jain, and J. Lifflander
- ***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



Important considerations while using MPI

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

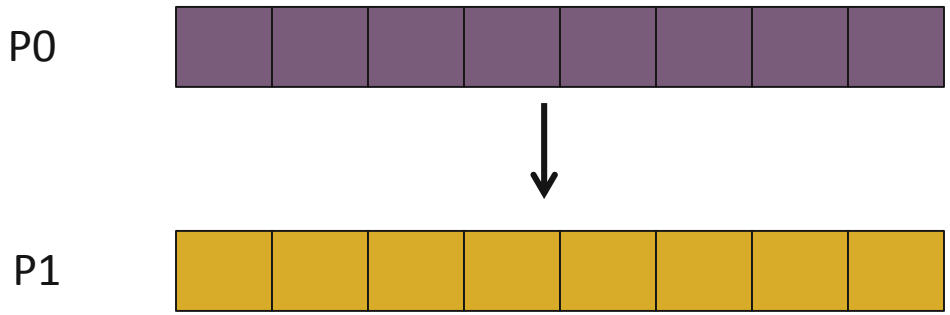
Parallel Sort using MPI Send/Recv



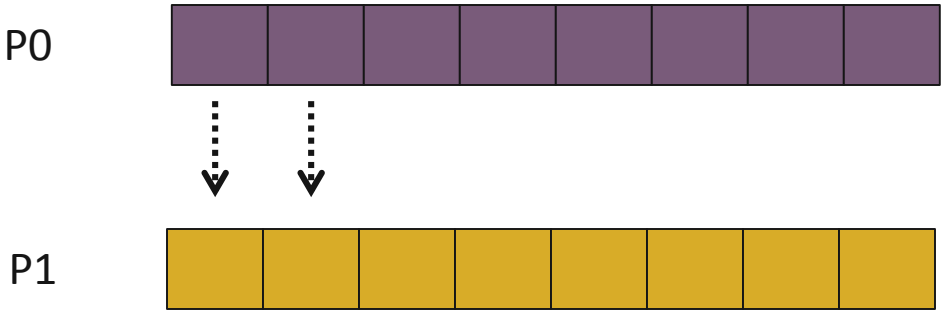
Parallel Sort using MPI Send/Recv (contd.)

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char ** argv)
{
    int rank, a[1000], b[500];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        MPI_Send(&a[500], 500, MPI_INT, 1, 0, MPI_COMM_WORLD);
        sort(a, 500);
        MPI_Recv(b, 500, MPI_INT, 1, 0, MPI_COMM_WORLD,
                                MPI_STATUS_IGNORE);

        /* Serial: Merge array b and sorted part of array a */
    }
    else if (rank == 1) {
        MPI_Recv(b, 500, MPI_INT, 0, 0, MPI_COMM_WORLD,
                                MPI_STATUS_IGNORE);
        sort(b, 500);
        MPI_Send(b, 500, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    MPI_Finalize(); return 0;
}
```

A Non-Blocking communication example



Blocking Communication



Non-blocking Communication

A Non-Blocking communication example

```
int main(int argc, char ** argv)
{
    [...snip...]
    if (rank == 0) {
        for (i=0; i< 100; i++) {
            /* Compute each data element and send it out */
            data[i] = compute(i);
            MPI_Isend(&data[i], 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
                      &request[i]);
        }
        MPI_Waitall(100, request, MPI_STATUSES_IGNORE)
    }
    else if (rank == 1){
        for (i = 0; i < 100; i++)
            MPI_Recv(&data[i], 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
                     MPI_STATUS_IGNORE);
    }
    [...snip...]
}
```

MPI Collective Routines

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

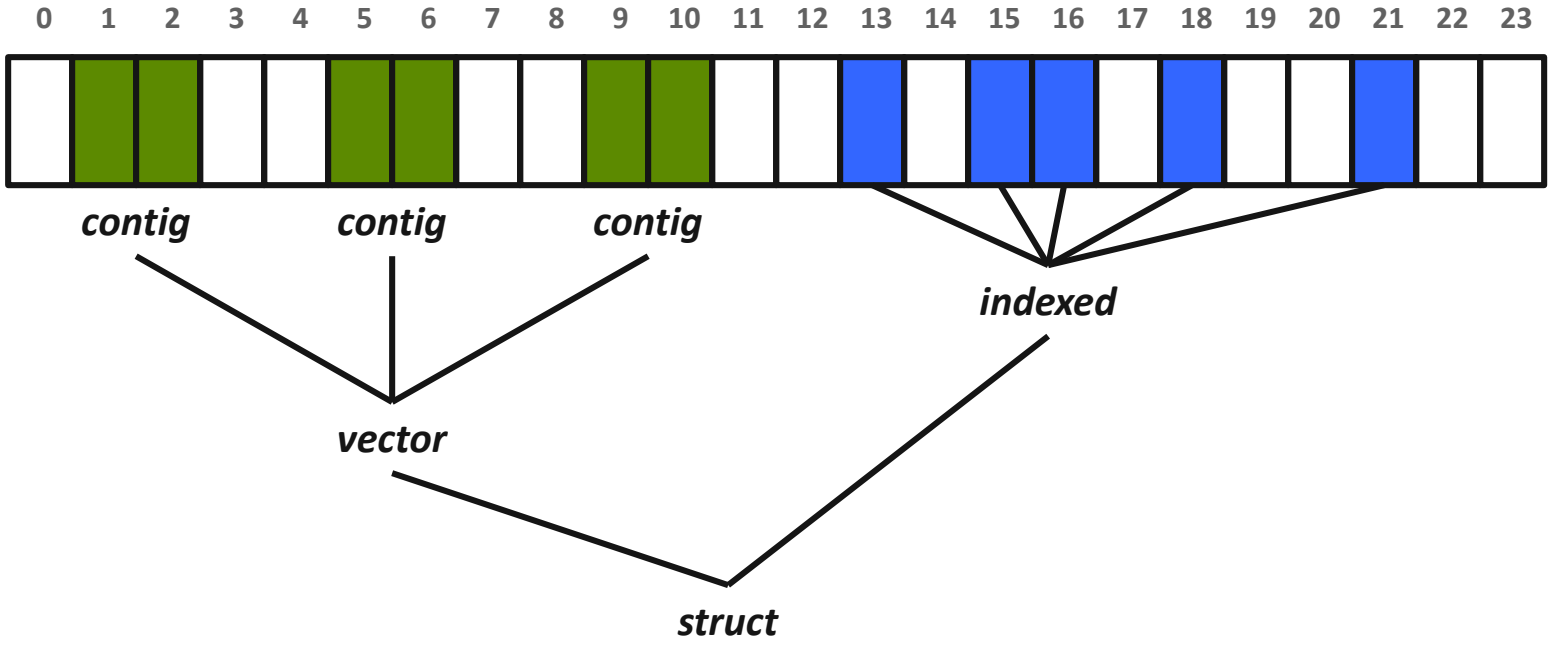
MPI Built-in Collective Computation Operations

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

Introduction to Datatypes in MPI

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

Derived Datatype Example



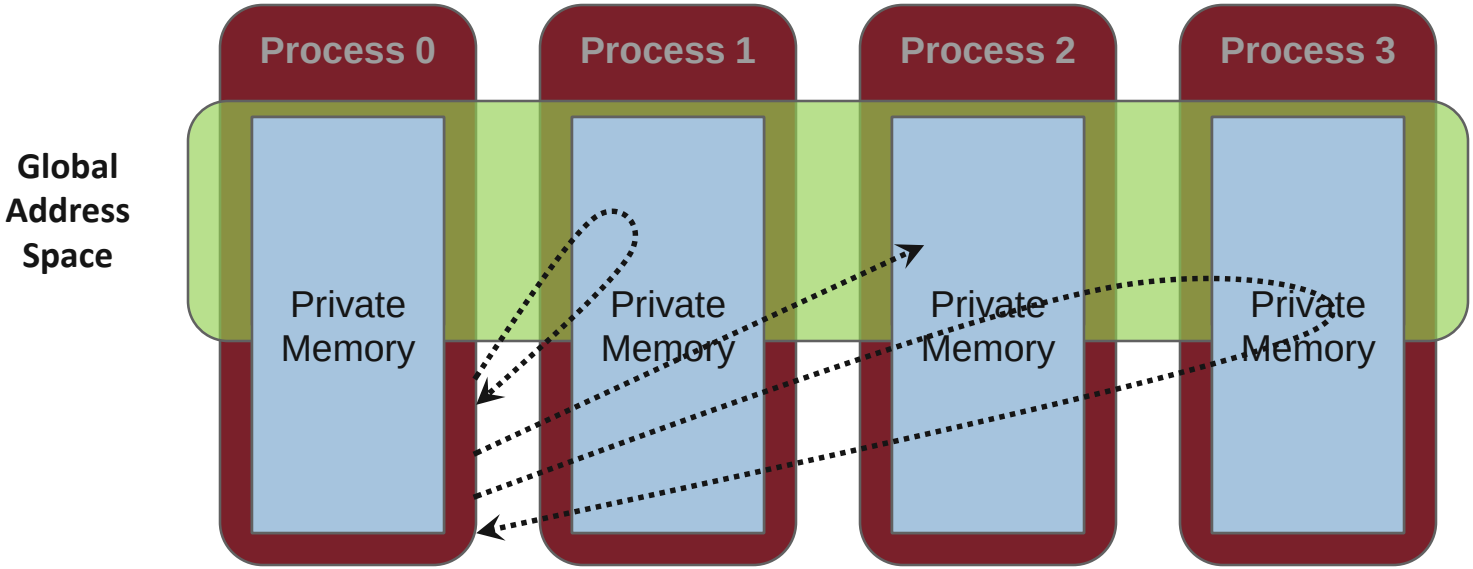
Argonne NATIONAL LABORATORY

Advanced Topics: One-sided Communication

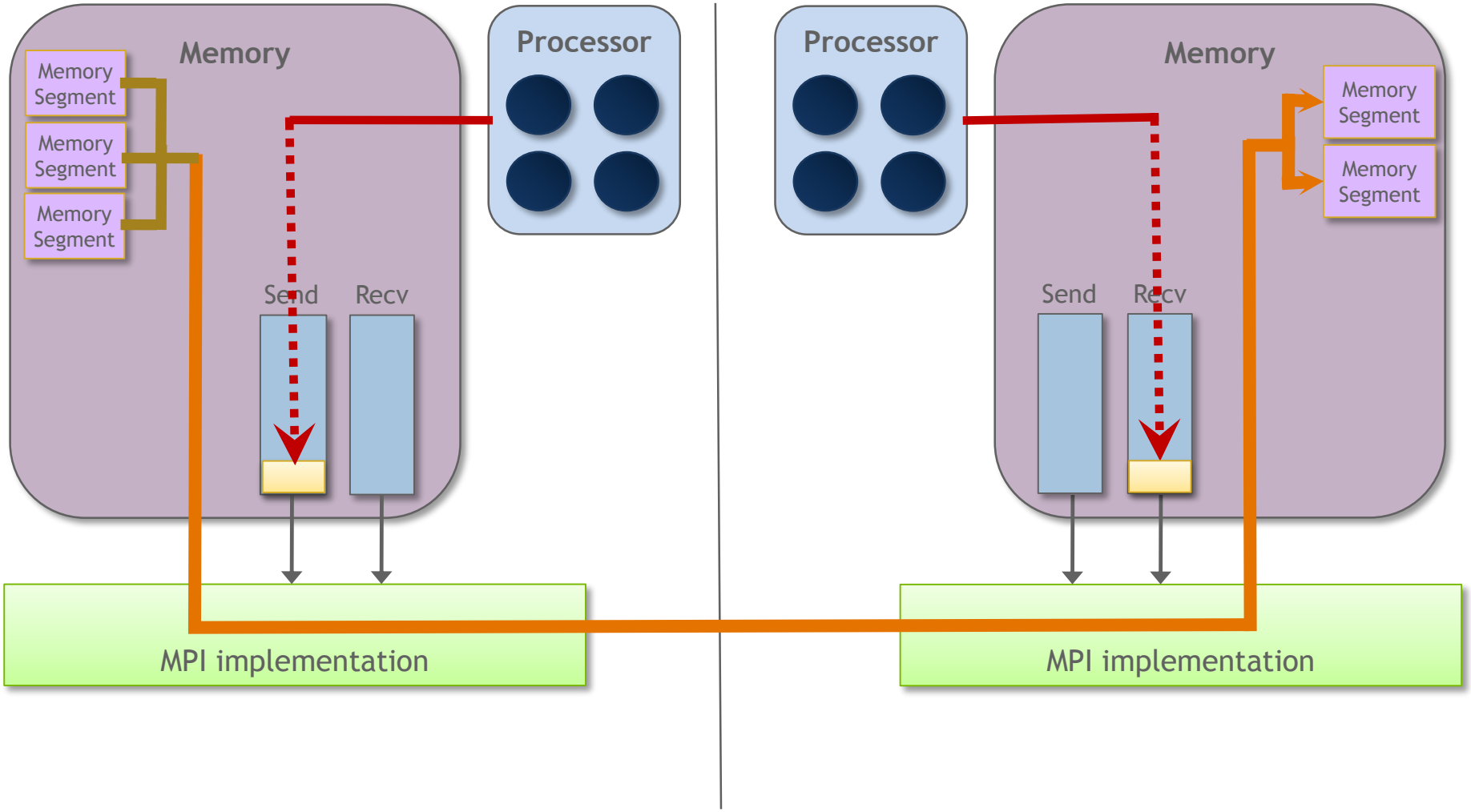
ETH Zürich

One-sided Communication

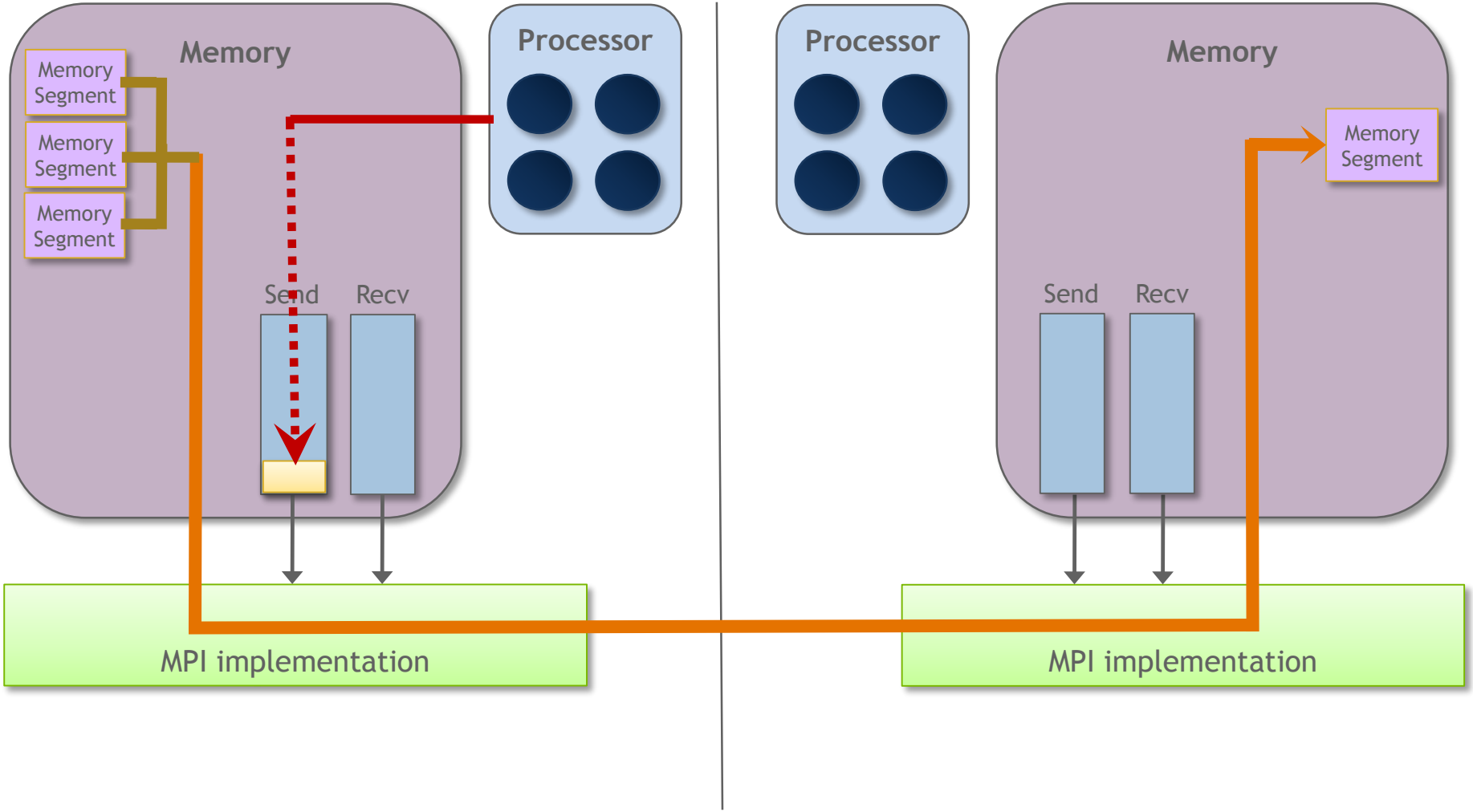
- The basic idea of one-sided communication models is to decouple data movement with process synchronization
  - Should be able to move data without requiring that the remote process synchronize
  - Each process exposes a part of its memory to other processes
  - Other processes can directly read from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



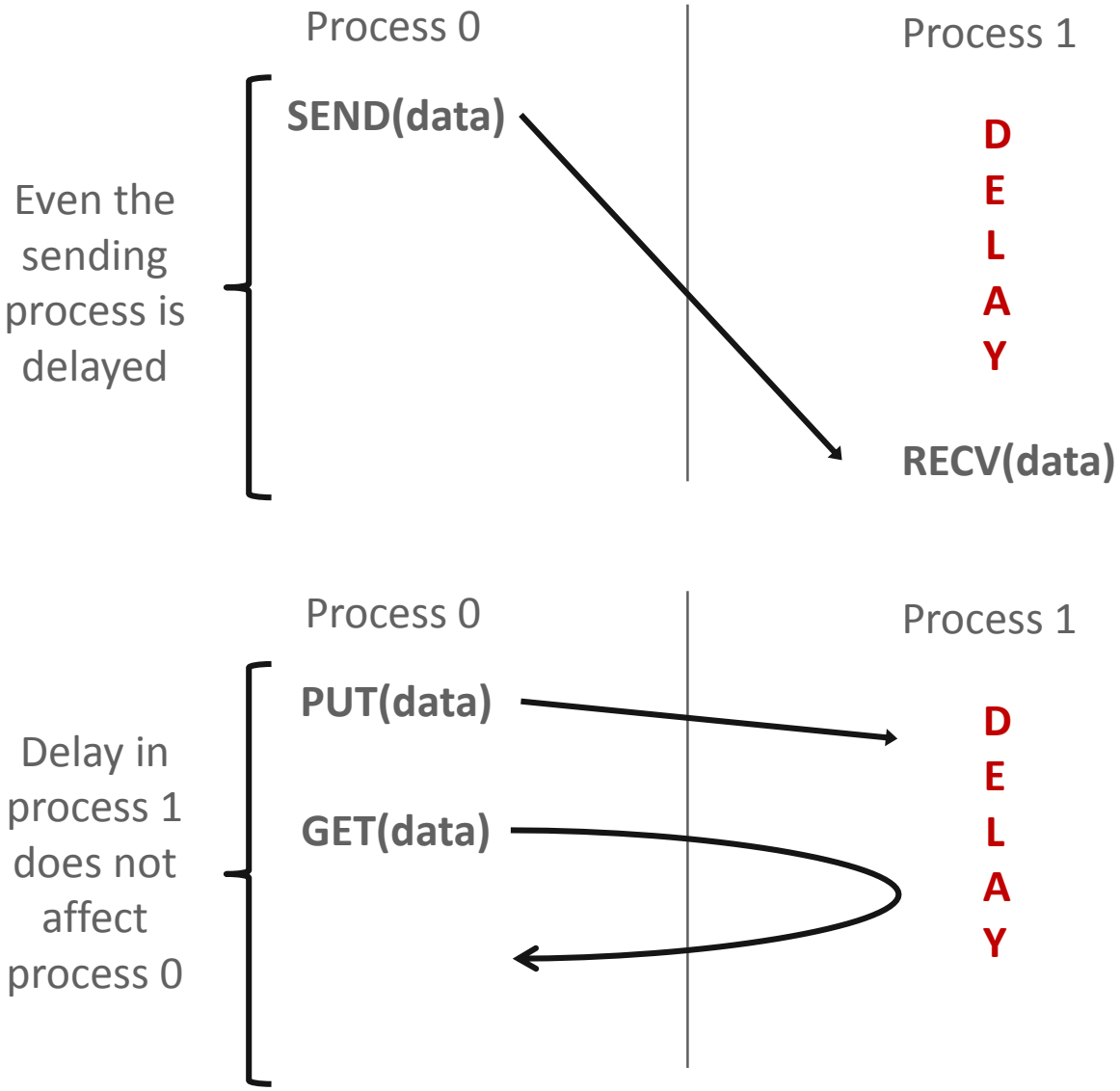
Two-sided Communication Example



One-sided Communication Example



Comparing One-sided and Two-sided Programming

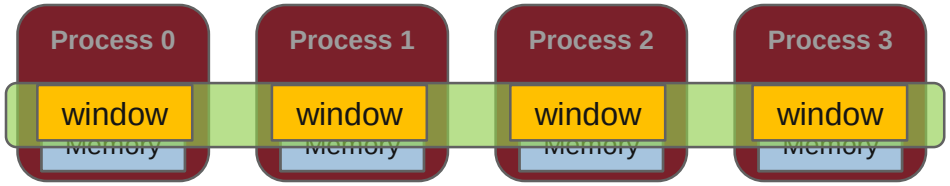


What we need to know in MPI RMA

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

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

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

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

MPI_WIN_CREATE_DYNAMIC

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

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

Example with MPI_WIN_CREATE_DYNAMIC

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

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

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

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

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

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

   MPI_Finalize(); return 0;
}
```

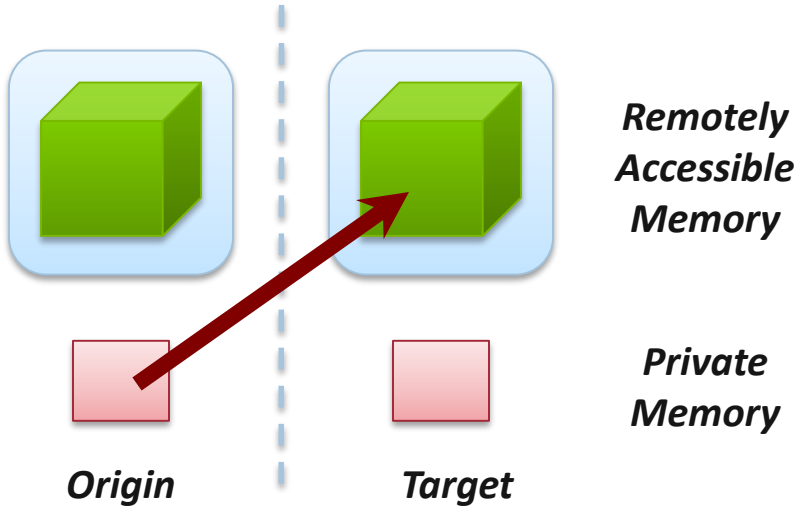
Data movement

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

Data movement: *Put*

```
MPI_Put(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)
```

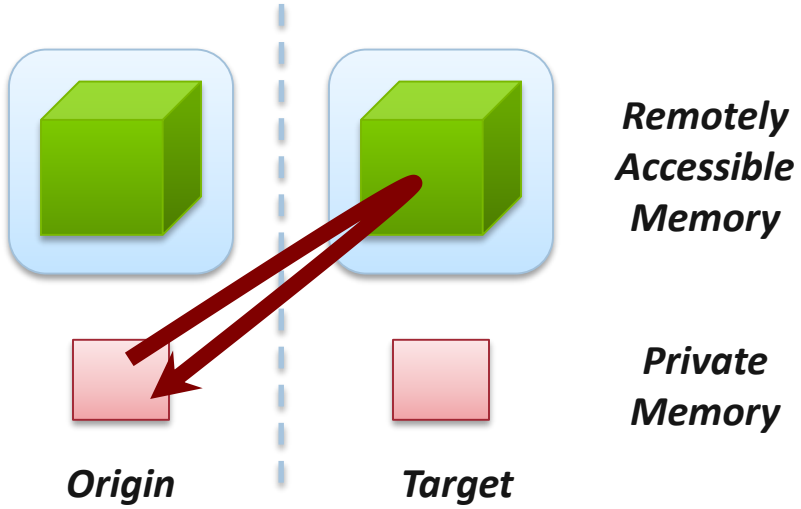
- Move data from origin, to target
- Separate data description triples for **origin** and **target**



Data movement: *Get*

```
MPI_Get(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)
```

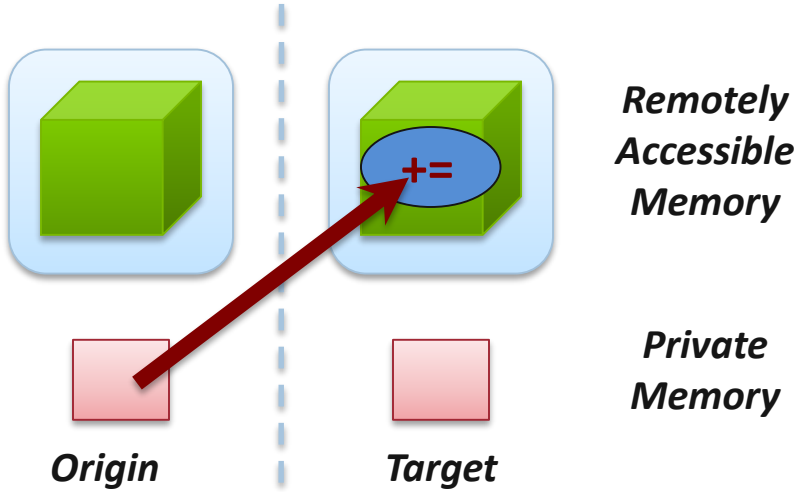
- Move data to origin, from target
- Separate data description triples for **origin** and **target**



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)
```

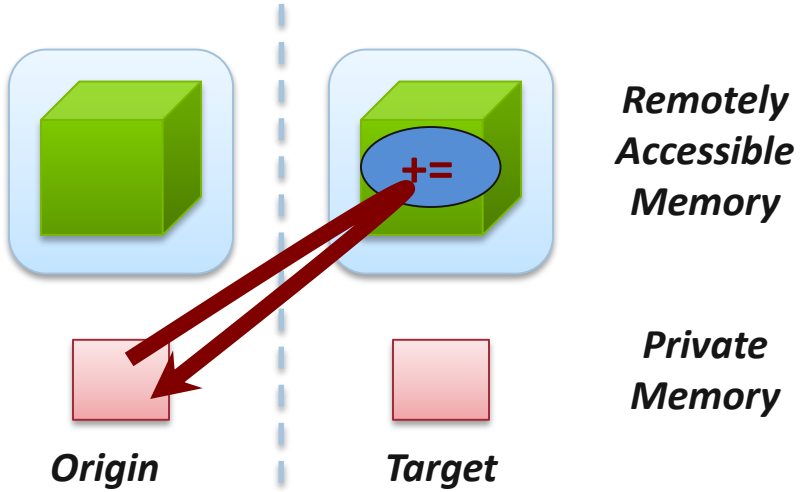
- Atomic update operation, similar to a put
  - Reduces origin and target data into target buffer using op argument as combiner
  - Op = MPI_SUM, MPI_PROD, MPI_OR, MPI_REPLACE, MPI_NO_OP, …
  - Predefined ops only, no user-defined operations
- Different data layouts between target/origin OK
  - Basic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

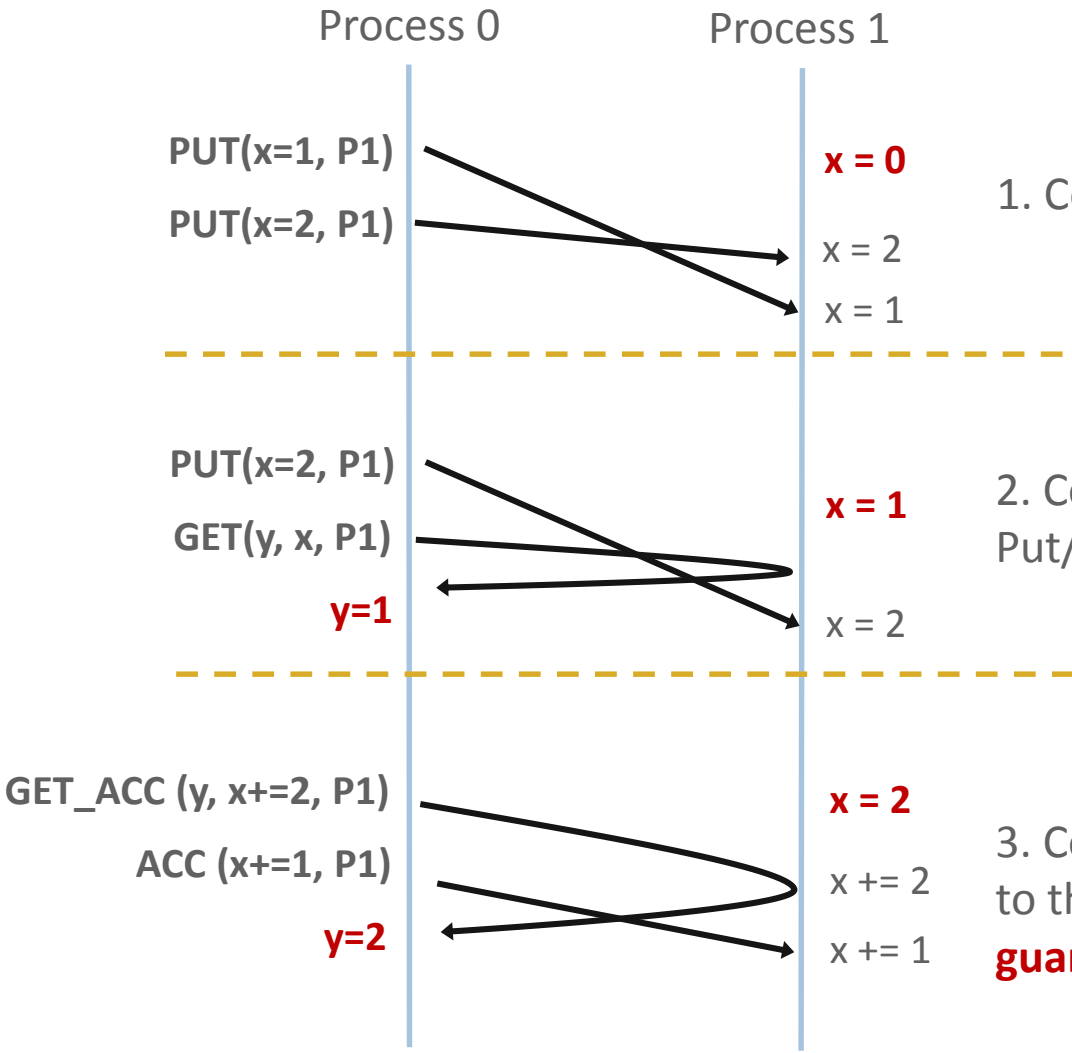
```
MPI_Compare_and_swap(void *origin_addr, void *compare_addr,
        void *result_addr, MPI_Datatype dtype, int target_rank,
        MPI_Aint target_disp, MPI_Win win)
```

- 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

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**

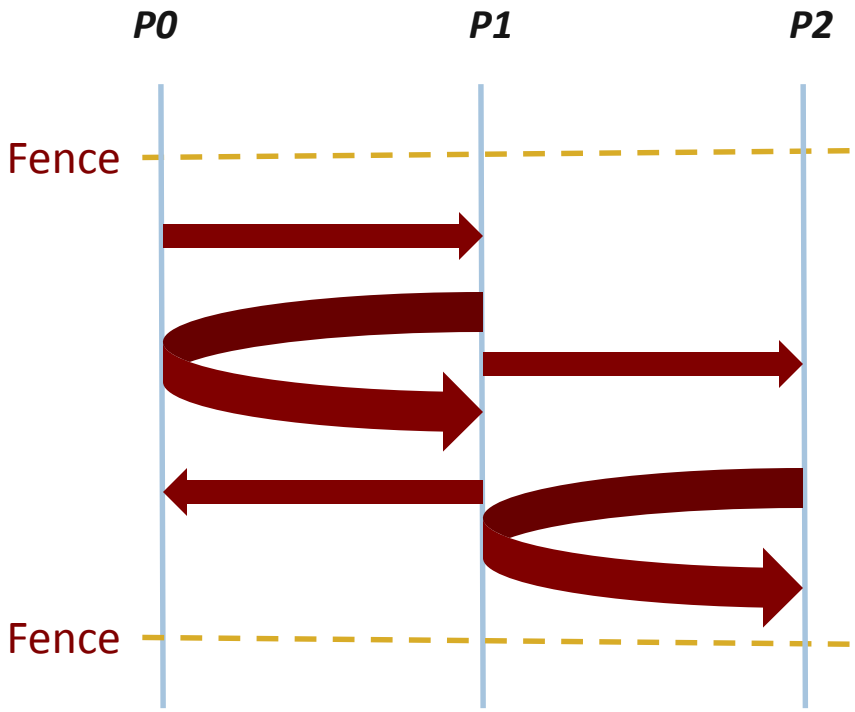
RMA Synchronization Models

- 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

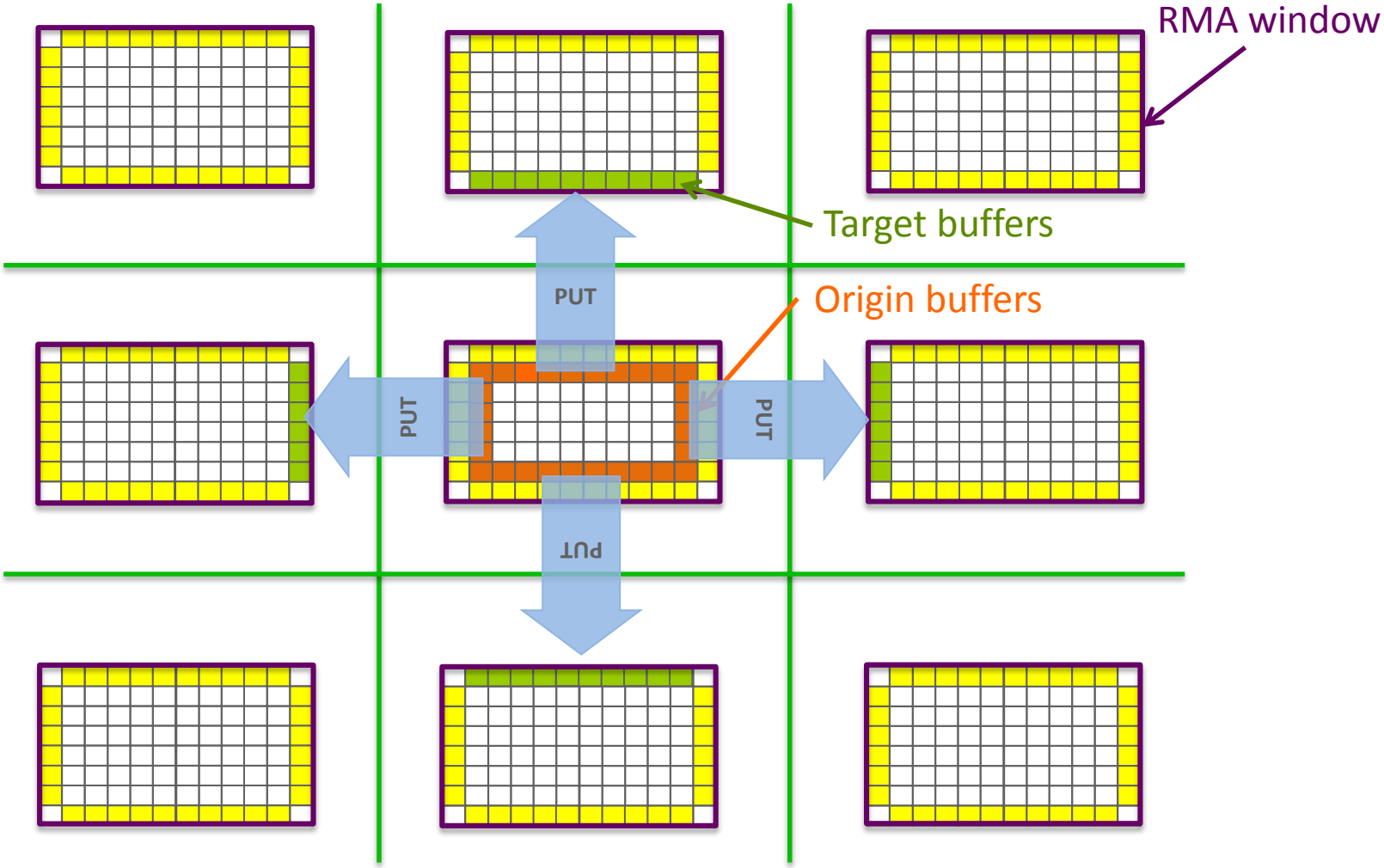
Fence: Active Target Synchronization

```
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



Implementing Stencil Computation with RMA Fence



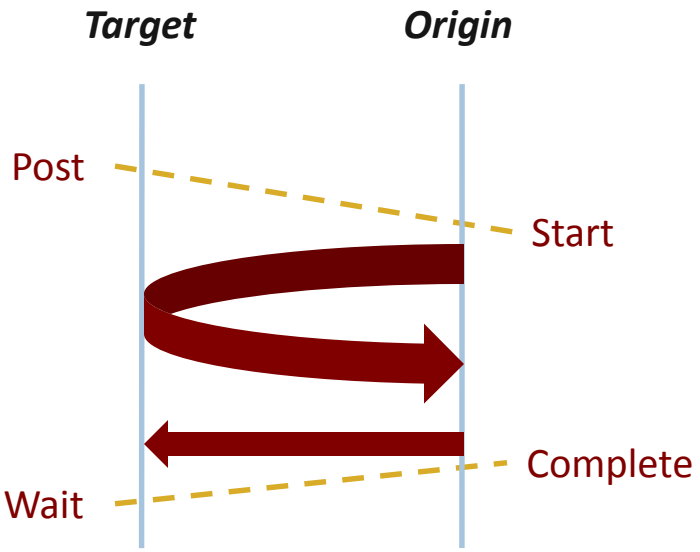
Code Example

- *stencil_mpi_ddt_rma.c*
- Use MPI_PUTs to move data, explicit receives are not needed
- Data location specified by MPI datatypes
- Manual packing of data no longer required

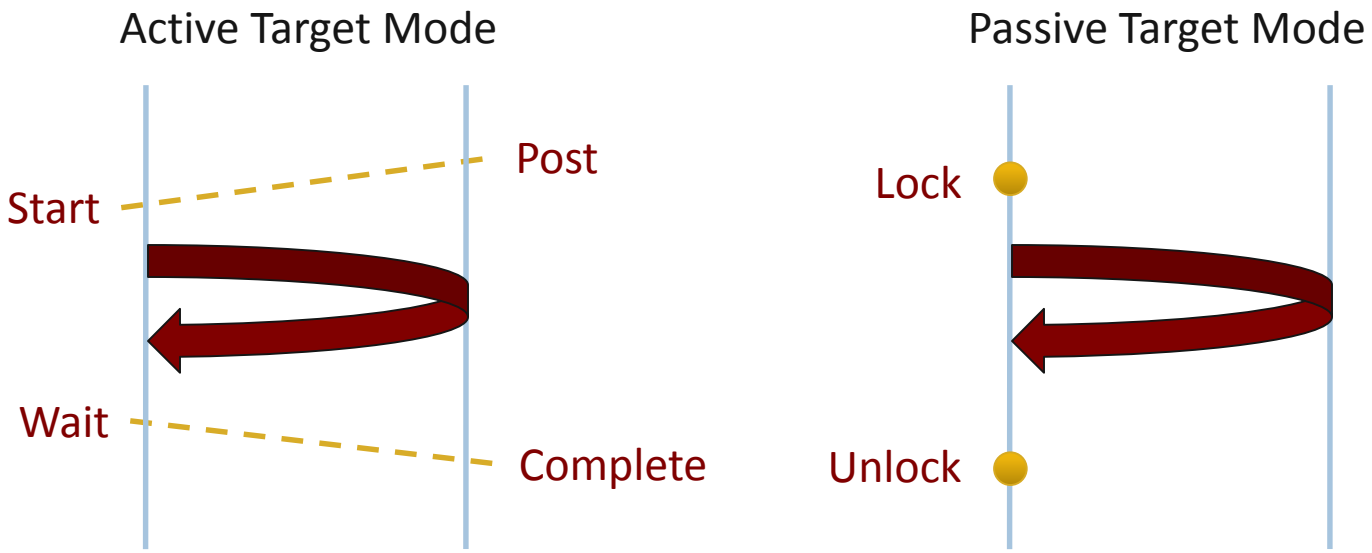
PSCW: Generalized Active Target Synchronization

```
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



Lock/Unlock: Passive Target Synchronization



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

Passive Target Synchronization

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

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

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

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

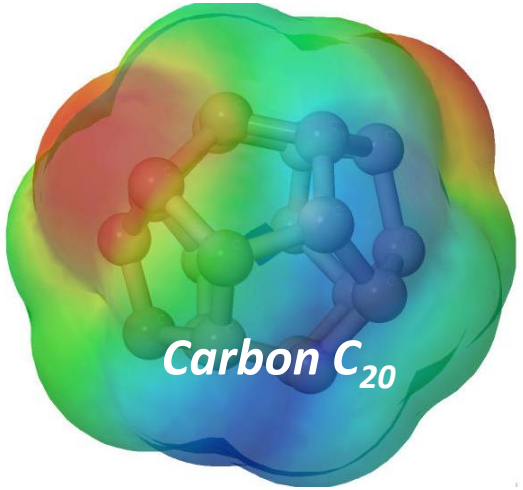
Advanced Passive Target Synchronization

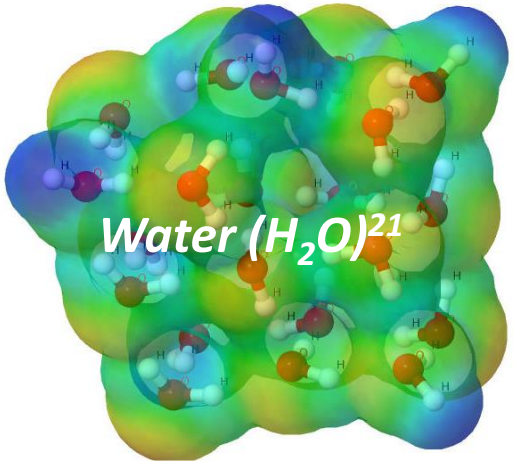
```
MPI_Win_lock_all(int assert, MPI_Win win)
MPI_Win_unlock_all(MPI_Win win)
MPI_Win_flush_all/flush_local_all(MPI_Win win)
```

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

NWChem [1]

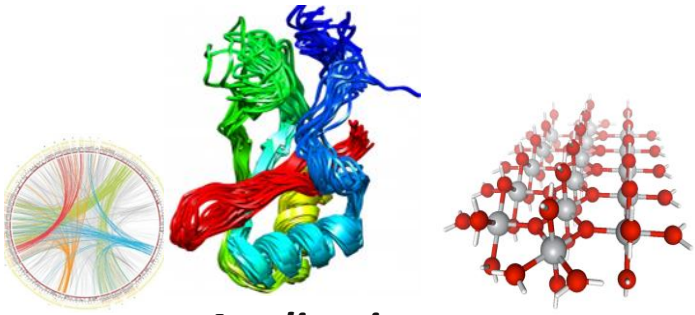
- High performance computational chemistry application suite
- Quantum level simulation of molecular systems
  - Very expensive in computation and data movement, so is used for small systems
  - Larger systems use molecular level simulations
- Composed of many simulation capabilities
  - Molecular Electronic Structure
  - Quantum Mechanics/Molecular Mechanics
  - Pseudo potential Plane-Wave Electronic Structure
  - Molecular Dynamics
- Very large code base
  - 4M LOC; Total investment of ~1B $ 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



Applications

Global Arrays [2]

ARMCI : Communication interface for RMA[3]

ARMCI native ports

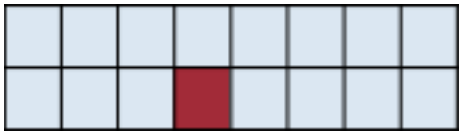
IB | DMMAP | ...

ARMCI-MPI

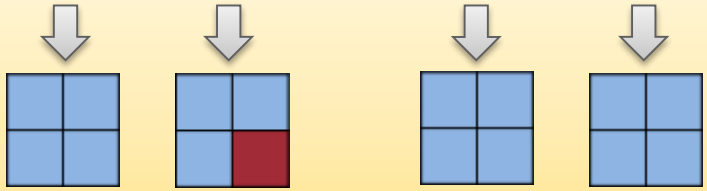
MPI RMA

Abstractions for distributed arrays

Global Address Space



Physically distributed to different processes



Hidden from user

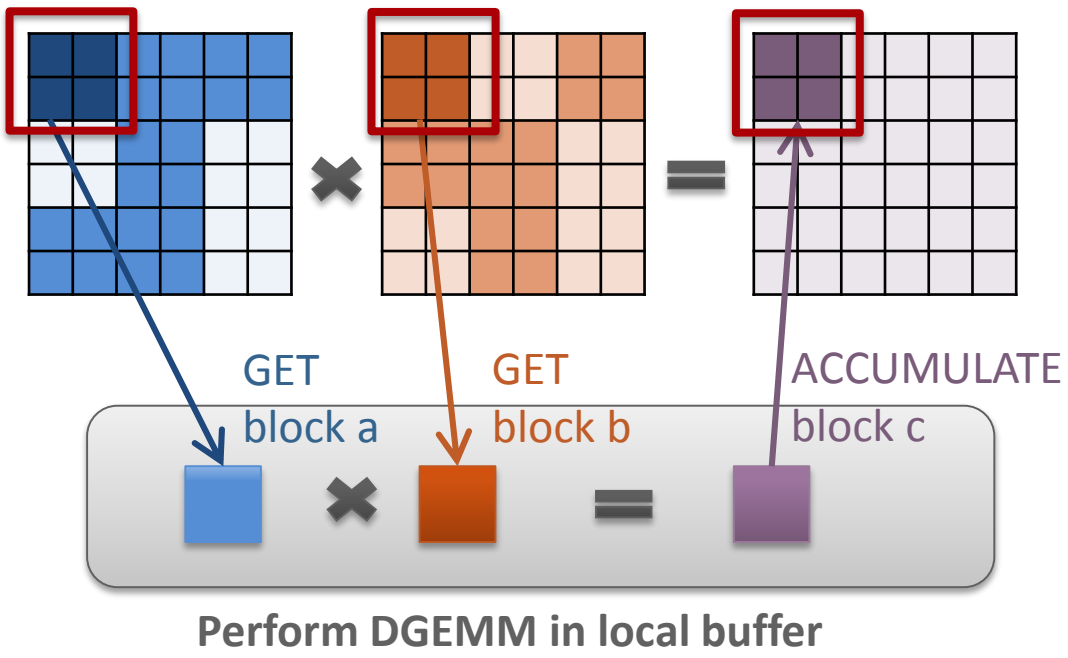
Irregularly access large amount of remote memory regions

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

Get-Compute-Update

- Typical Get-Compute-Update mode in GA programming

All of the blocks are non-contiguous data



Pseudo code

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

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

Code Example

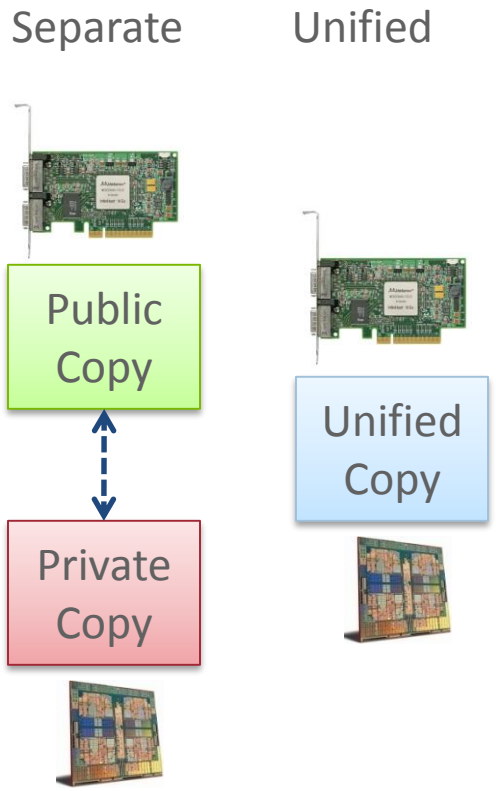
- ga_mpi_ddt_rma.c
- Only synchronization from origin processes, no synchronization from target processes

Which synchronization mode should I use, when?

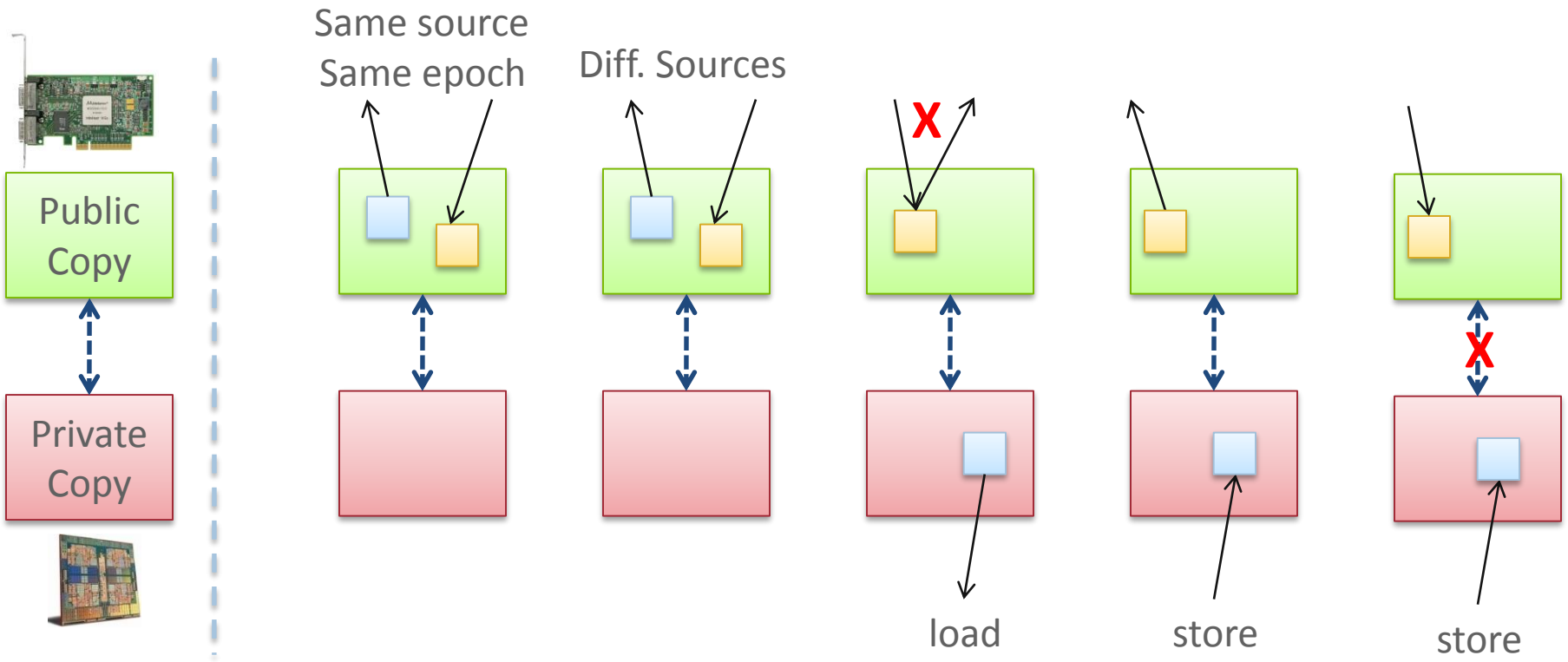
- RMA communication has low overheads versus send/recv
  - Two-sided: Matching, queuing, buffering, unexpected receives, etc…
  - One-sided: No matching, no buffering, always ready to receive
  - Utilize RDMA provided by high-speed interconnects (e.g. InfiniBand)
- Active mode: bulk synchronization
  - 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

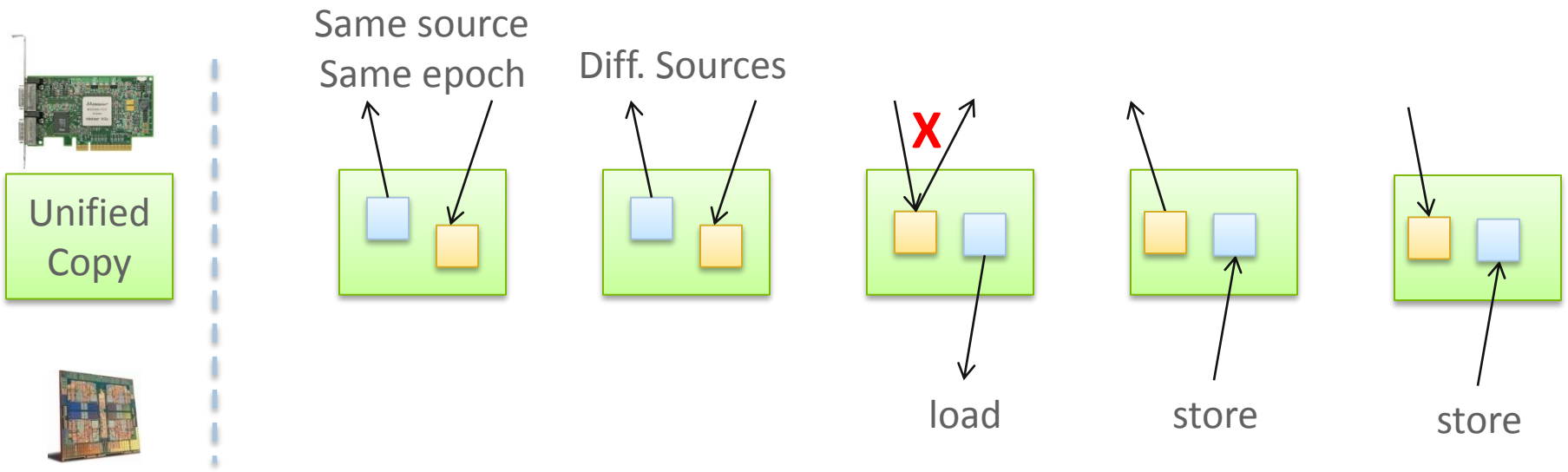


MPI RMA Memory Model (separate windows)



- 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

Argonne NATIONAL LABORATORY

Advanced Topics: Nonblocking Collectives

ETH Zürich

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

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 updated 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

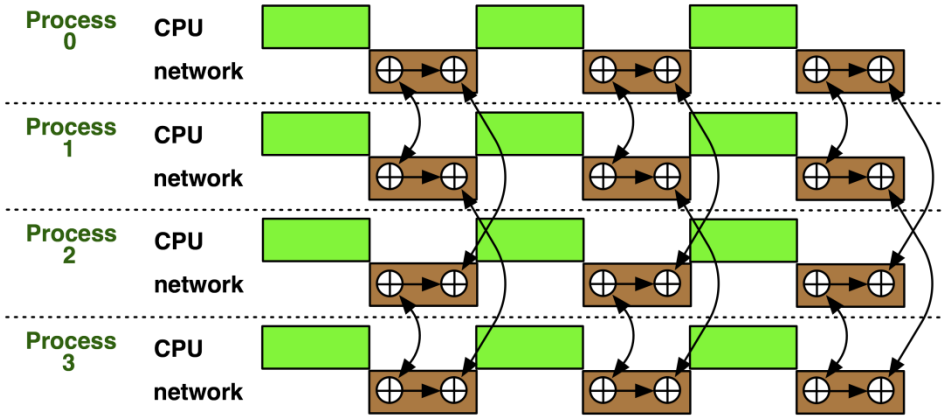
Nonblocking Collective Communication

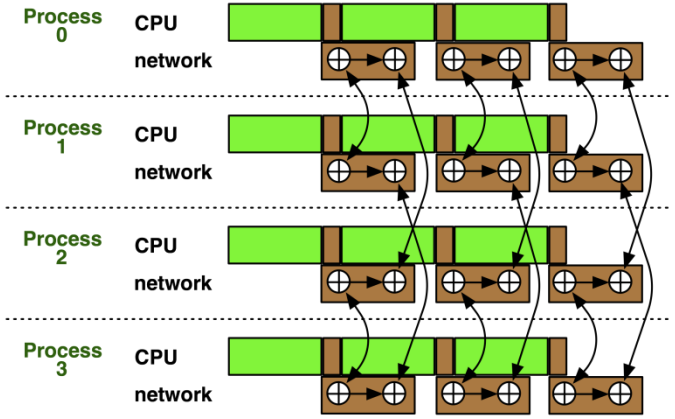
- Semantic advantages
  - Enable asynchronous progression (and manual)
    - Software pipelining
  - Decouple data transfer and synchronization
    - Noise resiliency!
  - Allow overlapping communicators
    - See also neighborhood collectives
  - Multiple outstanding operations at any time
    - Enables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





A Non-Blocking Barrier?

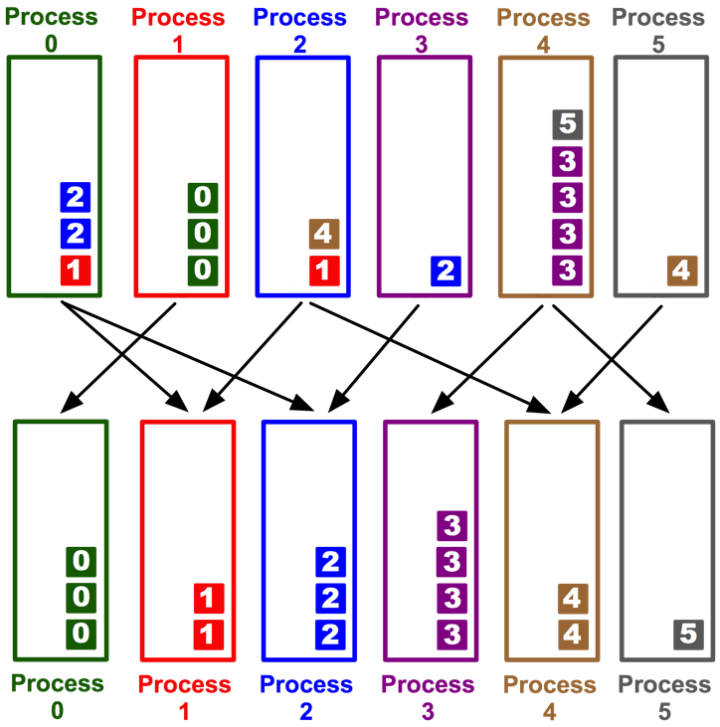
- What can that be good for? Well, quite a bit!
- Semantics:
  - MPI_Ibarrier() – calling process entered the barrier, **no** synchronization happens
  - Synchronization **may** happen asynchronously
  - MPI_Test/Wait() – synchronization happens **if** necessary
- Uses:
  - Overlap barrier latency (small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

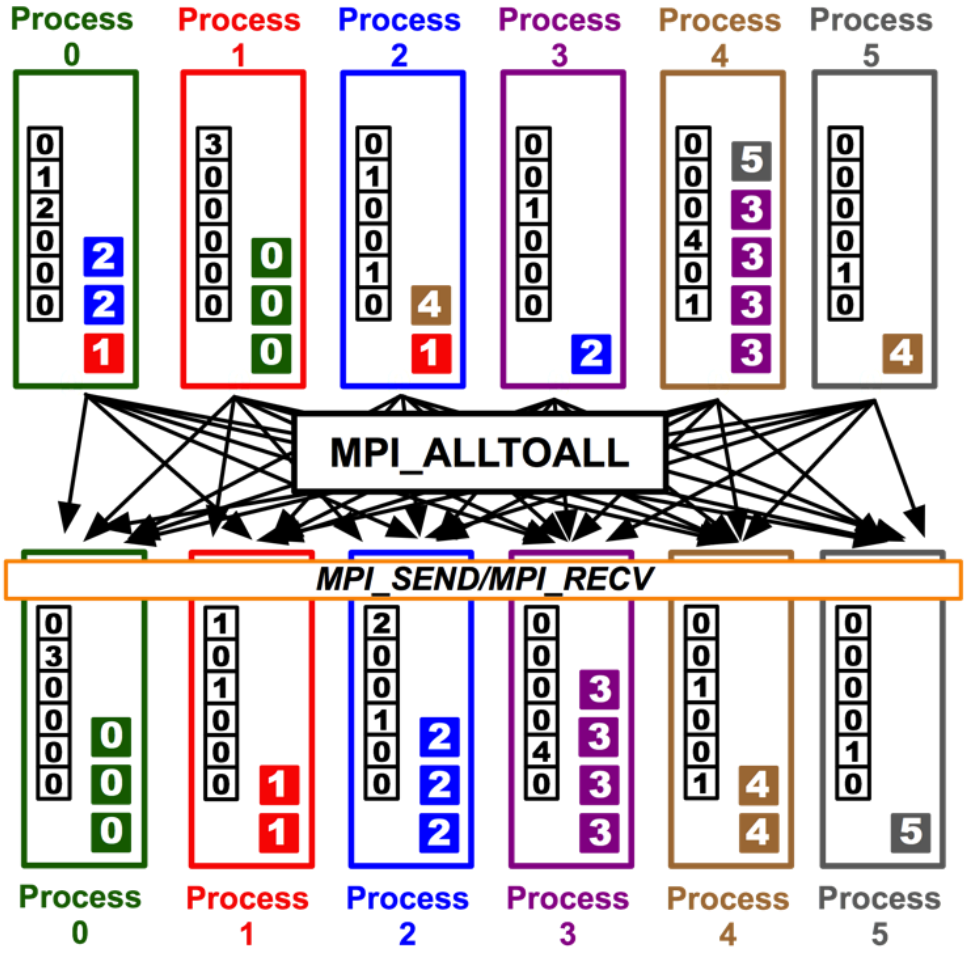
  OR:
  - Use MPI semantics:
    - Unknown sender (MPI_ANY_SOURCE)
    - Unknown message size (MPI_PROBE)
    - Reduces problem to counting the number of neighbors
    - Allow faster implementation!



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

Using Alltoall (PEX)

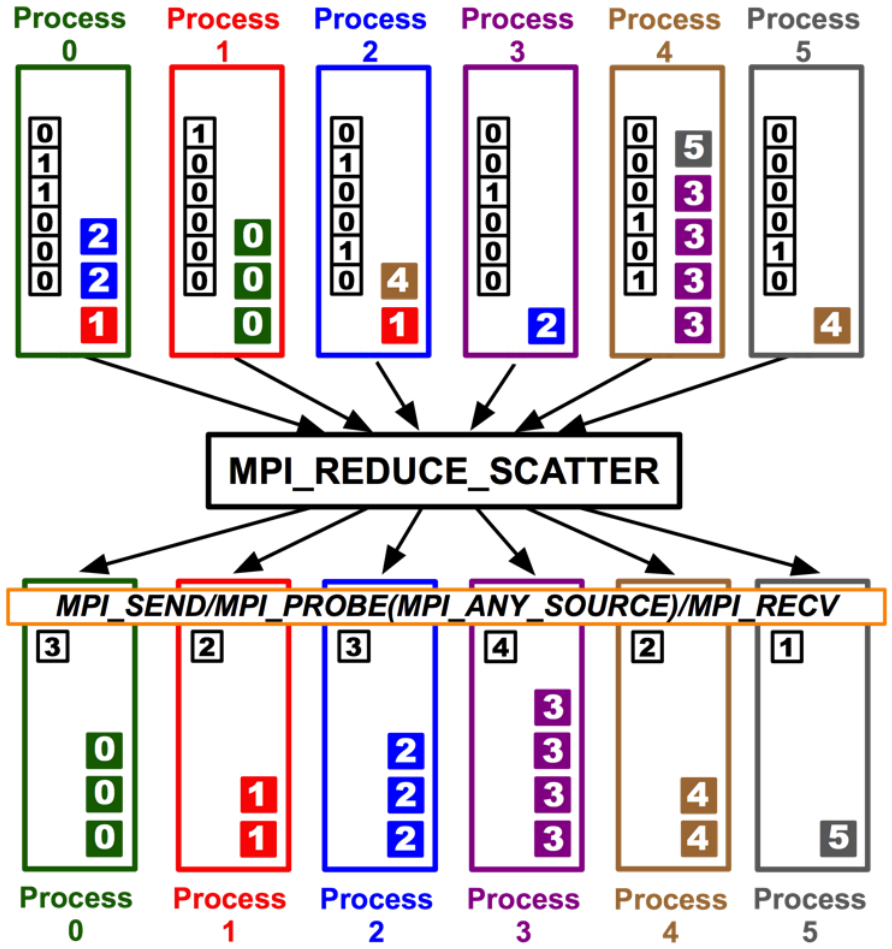
- Based on Personalized Exchange ($\Theta(P)$)
  - Processes exchange metadata (sizes) about neighborhoods with all-to-all
  - Processes post receives afterwards
  - Most intuitive but least performance and scalability!



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

Reduce_scatter (PCX)

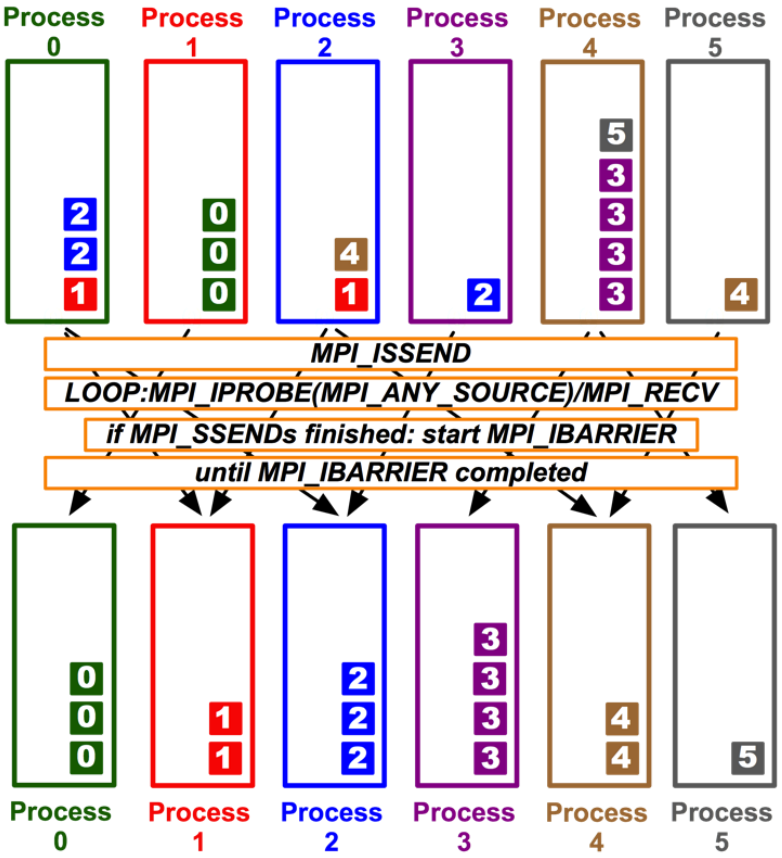
- Bases on Personalized Census ($\Theta(P)$)
  - Processes exchange metadata (counts) about neighborhoods with reduce_scatter
  - Receivers checks with wildcard MPI_IPROBE and receives messages
  - Better than PEX but non-deterministic!



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 Alltoall, reduce-scatter!

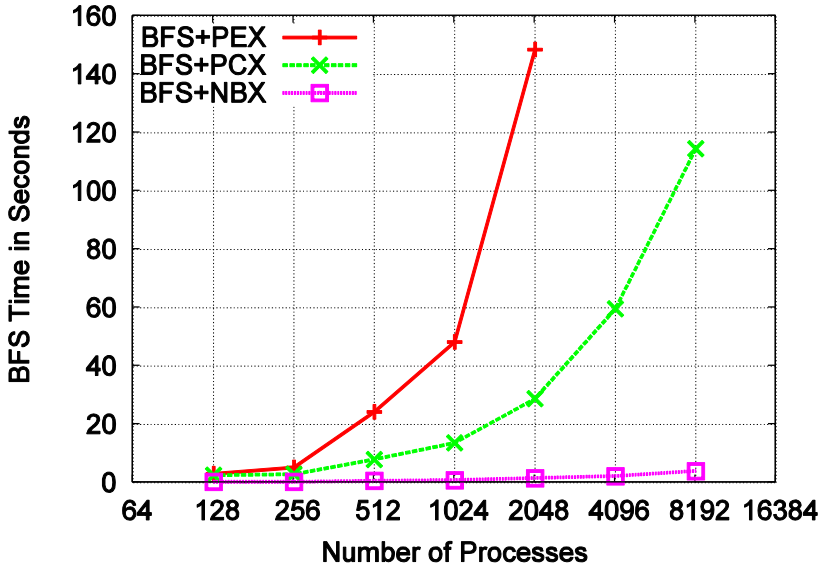


T. Hoefler et al.: Scalable Communication Protocols for Dynamic Sparse Data Exchange

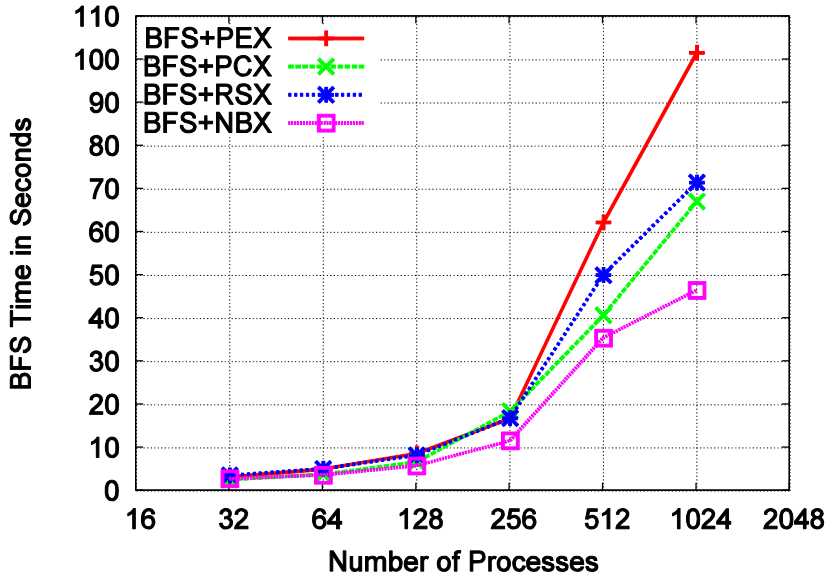
Parallel Breadth First Search

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

BlueGene/P – with HW barrier!



Myrinet 2000 with LibNBC

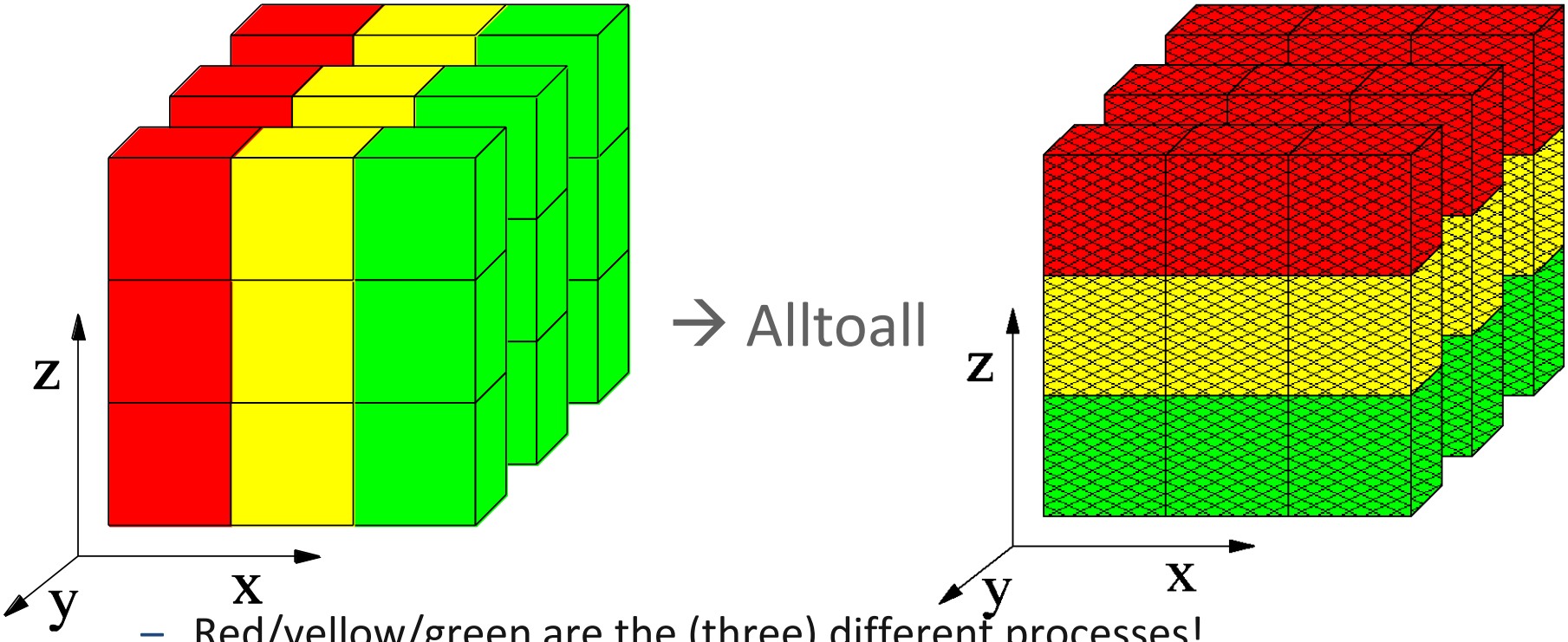


- HW barrier support is significant at large scale!

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



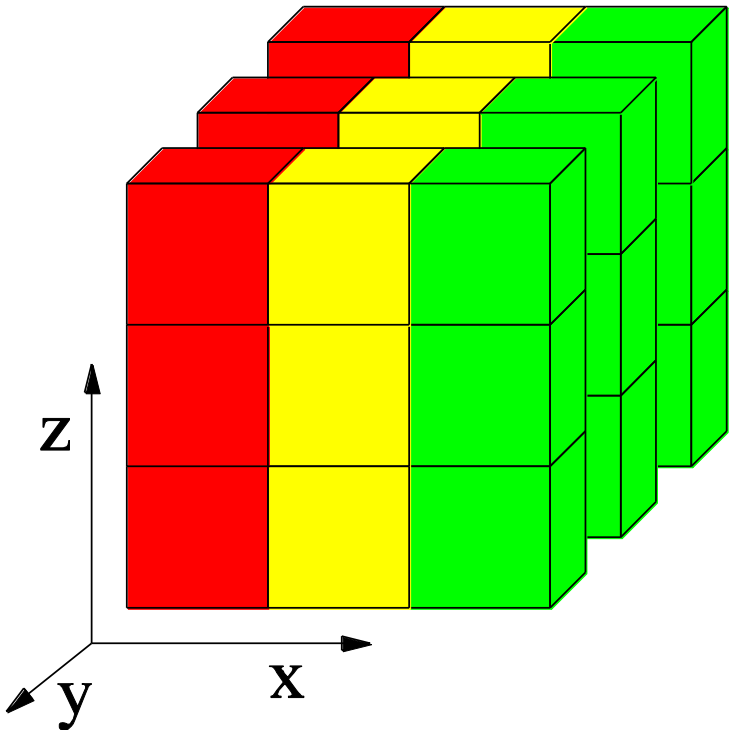
  - Red/yellow/green are the (three) different processes!

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
```

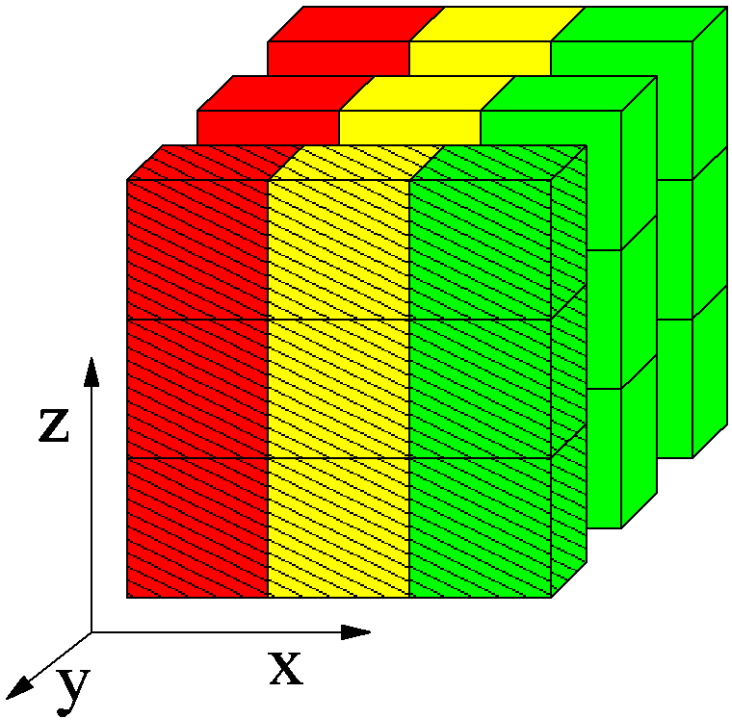
Parallel Fast Fourier Transform

- Data already transformed in y-direction



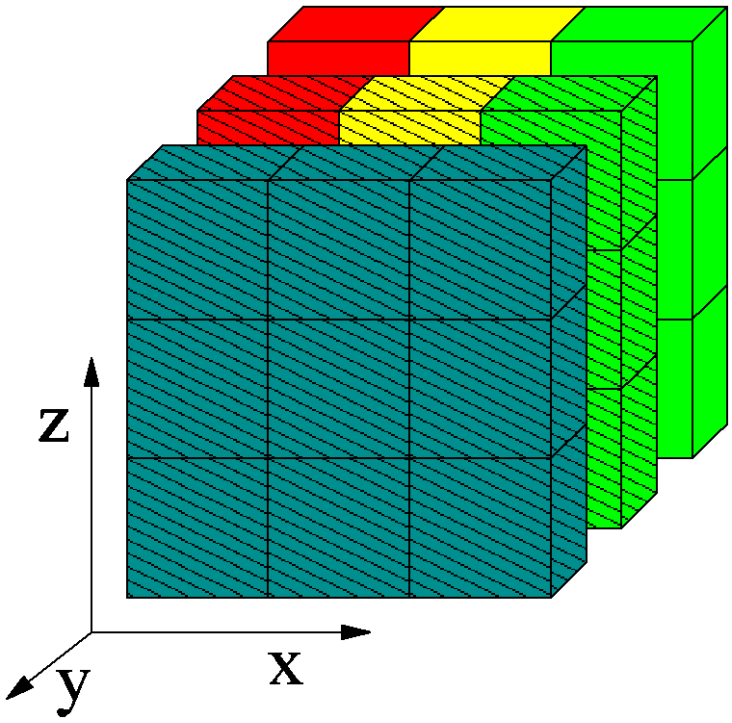
Parallel Fast Fourier Transform

- Transform first y plane in z



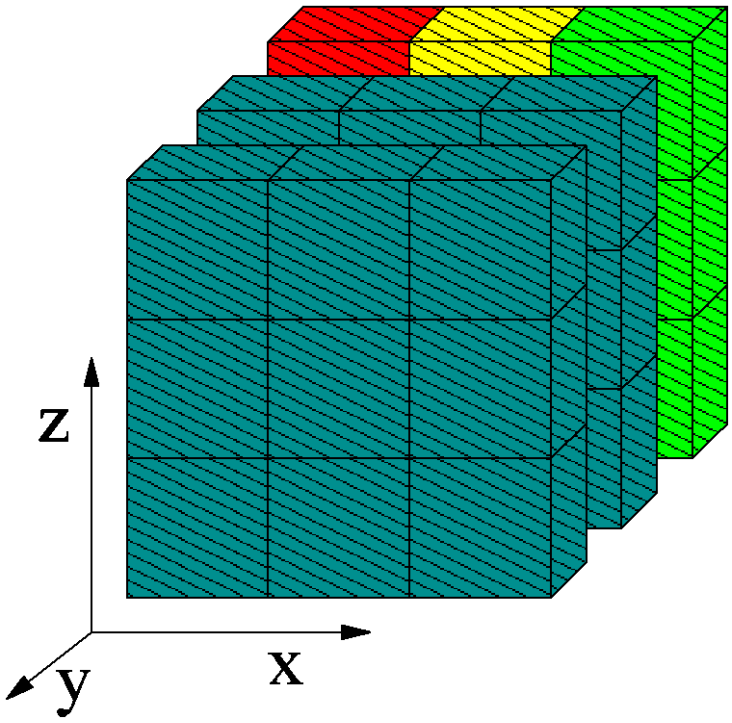
Parallel Fast Fourier Transform

- Start ialltoall and transform second plane



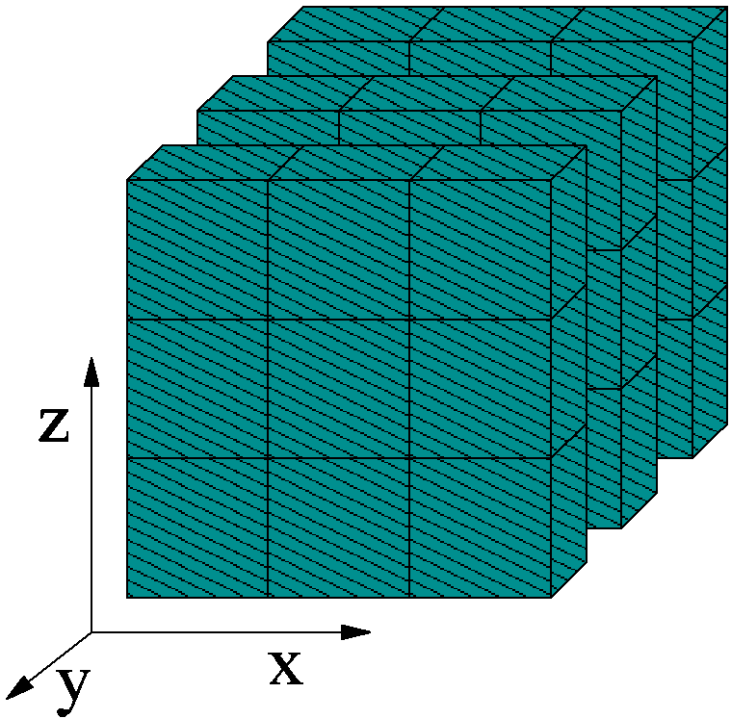
Parallel Fast Fourier Transform

- Start ialltoall (second plane) and transform third



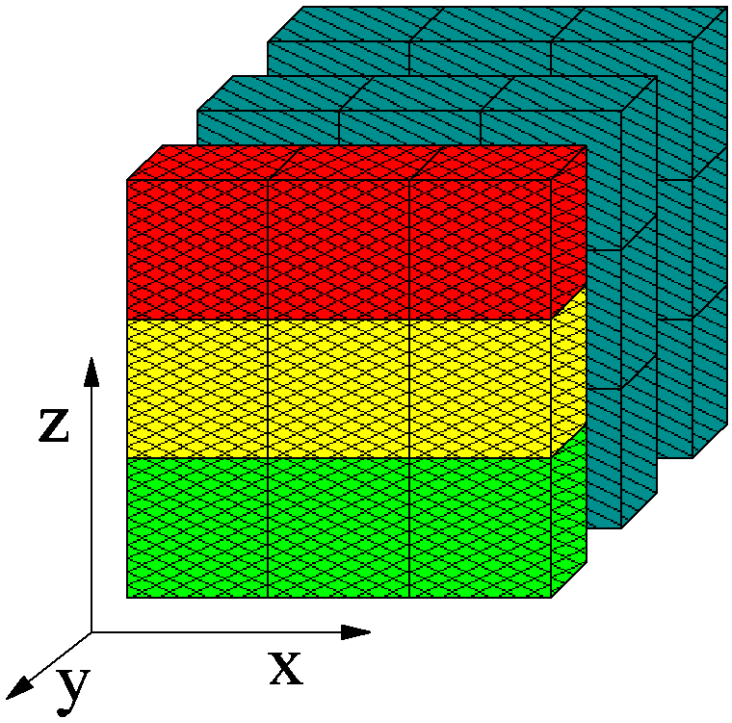
Parallel Fast Fourier Transform

- Start ialltoall of third plane and …



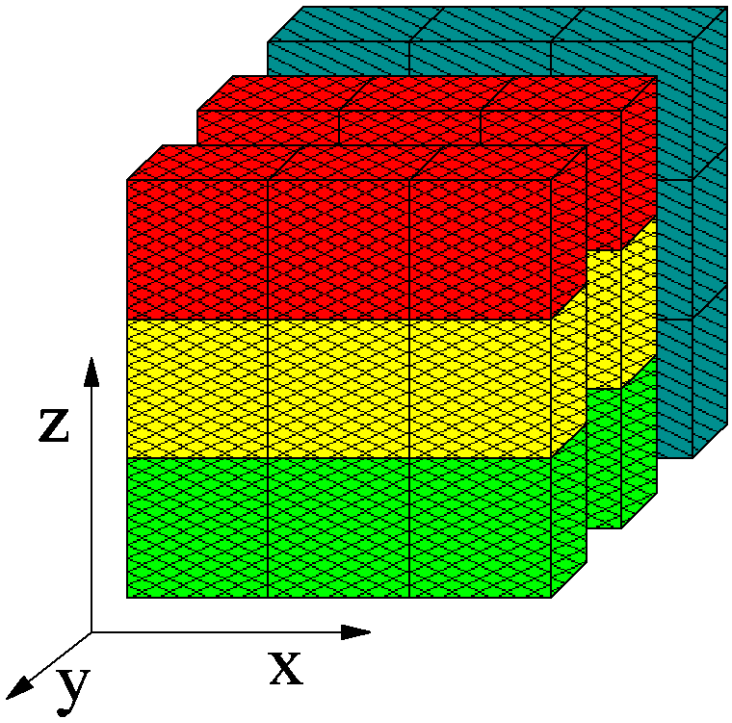
Parallel Fast Fourier Transform

- Finish ialltoall of first plane, start x transform



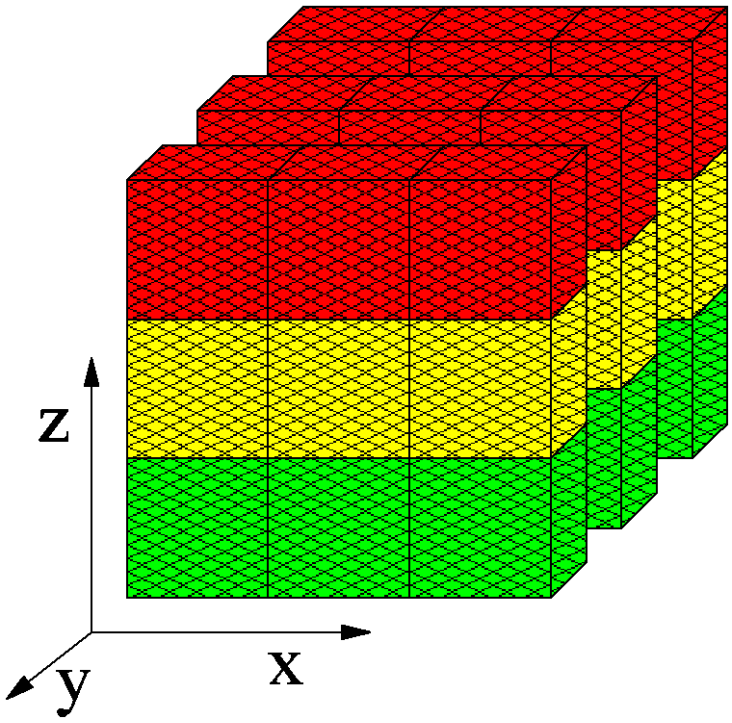
Parallel Fast Fourier Transform

- 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
```

Nonblocking And Collective Summary

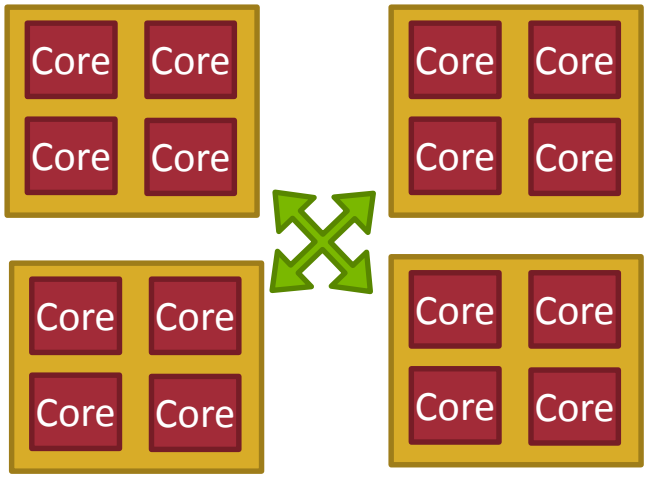
- Nonblocking comm does two things:
  - Overlap and relax synchronization
- Collective comm does one thing
  - Specialized pre-optimized routines
  - Performance portability
  - Hopefully transparent performance
- They can be composed
  - E.g., software pipelining

Argonne NATIONAL LABORATORY

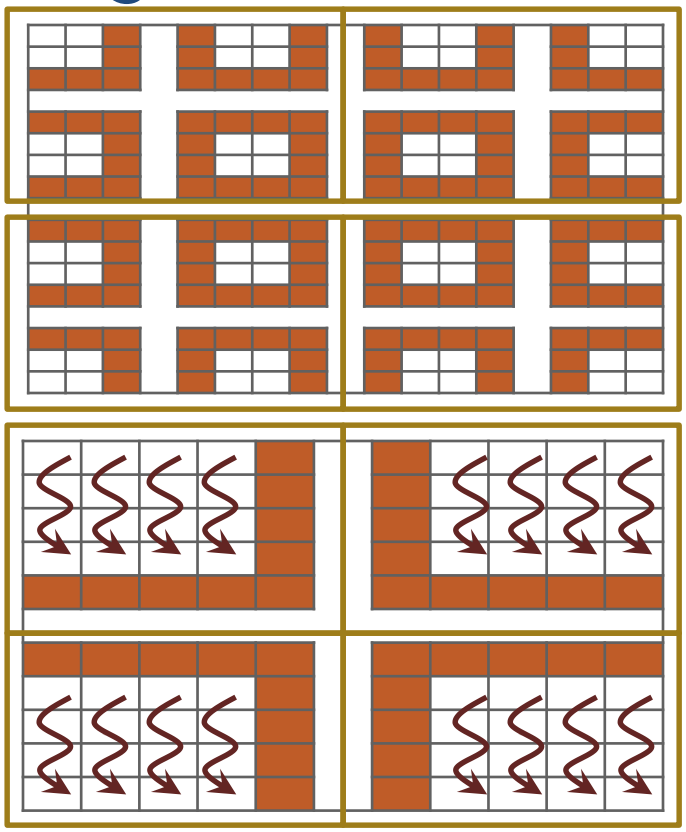
Advanced Topics: Hybrid Programming with Threads, Shared Memory, and Accelerators

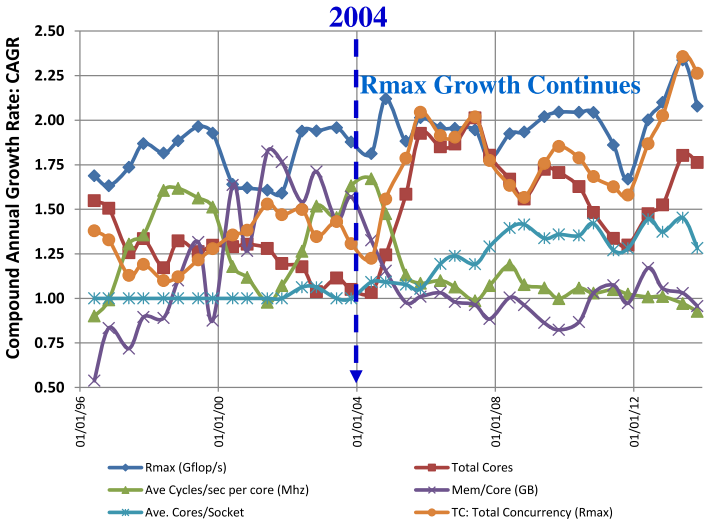
ETH Zürich

Why Going Hybrid MPI + X Programming?



Domain Decomposition





Flops/cycle increased even faster

Total Cores increased even faster

Cores/socket increased

Memory/core went flat

Clock rates went flat

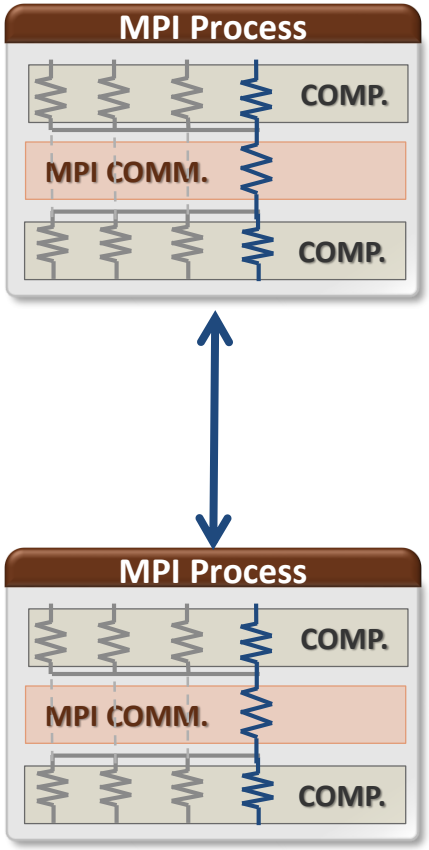
Growth of node resources in the Top500 systems. Peter Kogge: "Reading the Tea-Leaves: How Architecture Has Evolved at the High End". IPDPS 2014 Keynote

- Sharing promotes cooperation
  - Reduced memory consumption
  - Efficient use of shared resources: caches, TLB entries, network endpoints, etc.

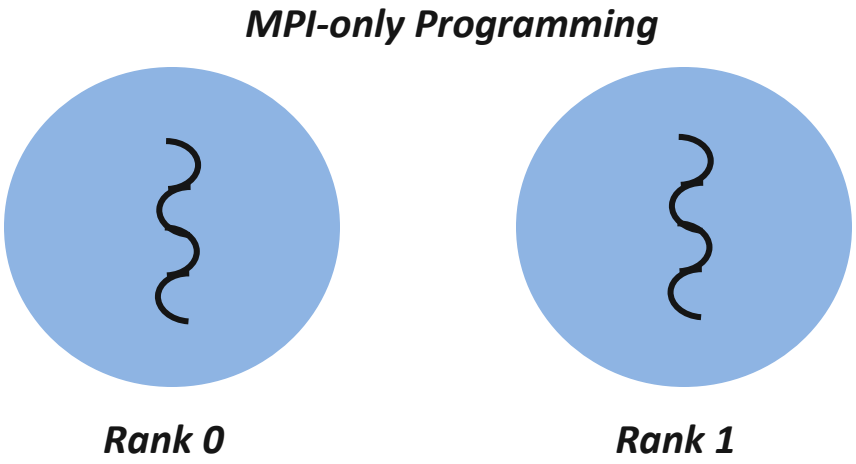
MPI + Threads

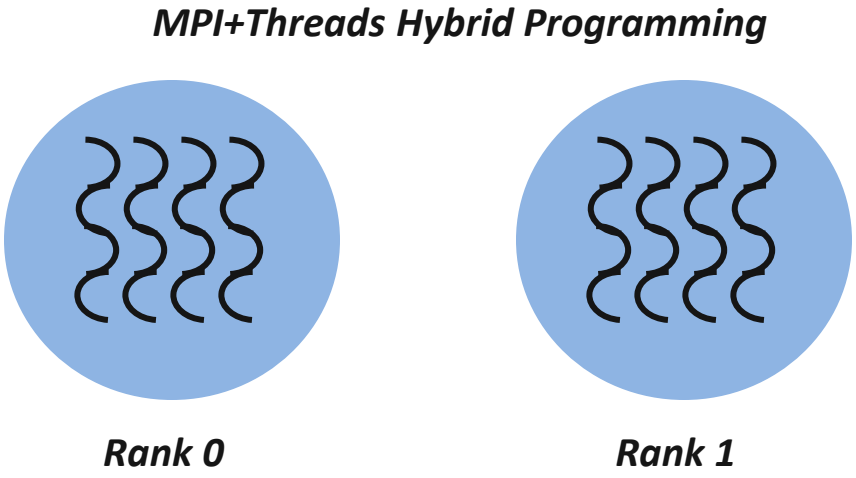
MPI and Threads

- 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.



Hybrid Programming with MPI+Threads





- In MPI-only programming, each MPI process has a single thread of execution
- 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

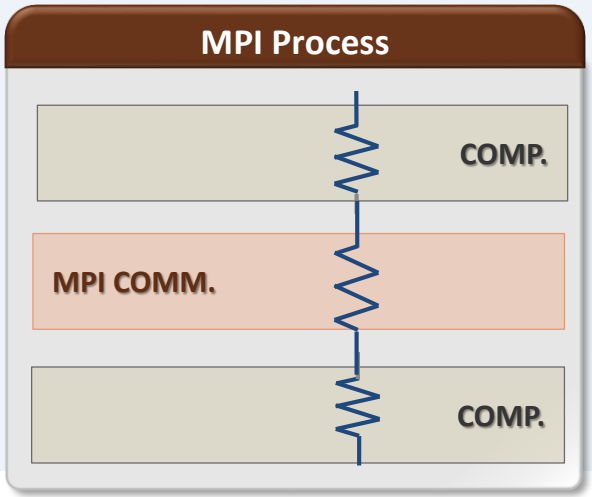
MPI's Four Levels of Thread Safety

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

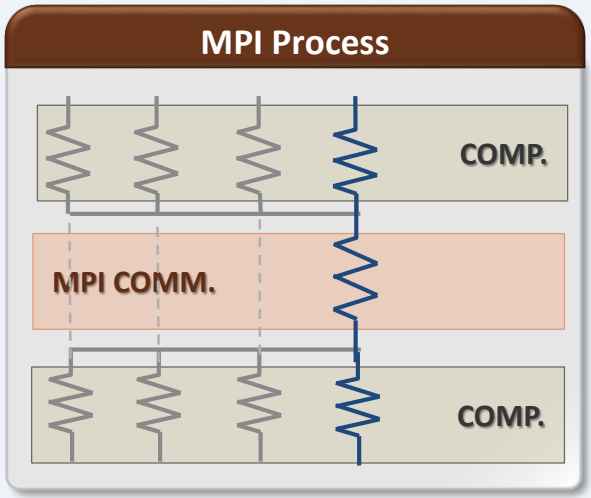
```
int main(int argc, char ** argv)
{
    int buf[100];

    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

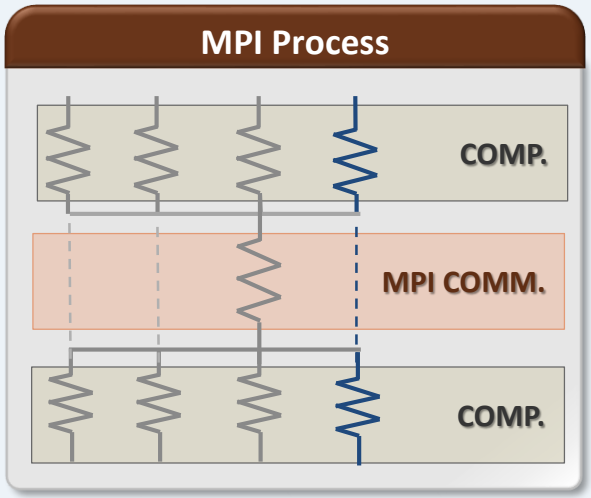
```
int main(int argc, char ** argv)
{
   int buf[100], provided;

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

#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

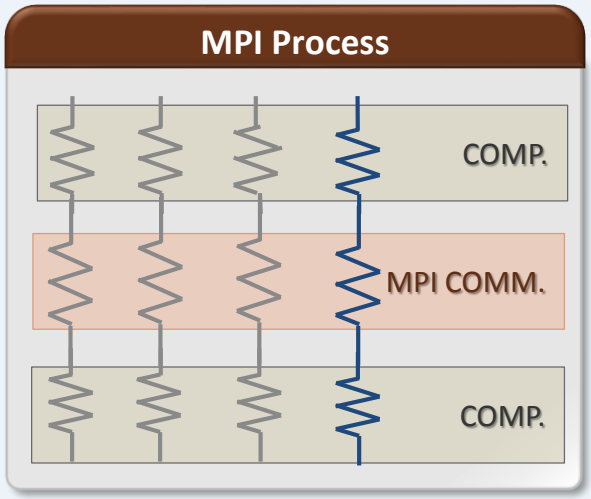
```
int main(int argc, char ** argv)
{
    int buf[100], provided;

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

#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)

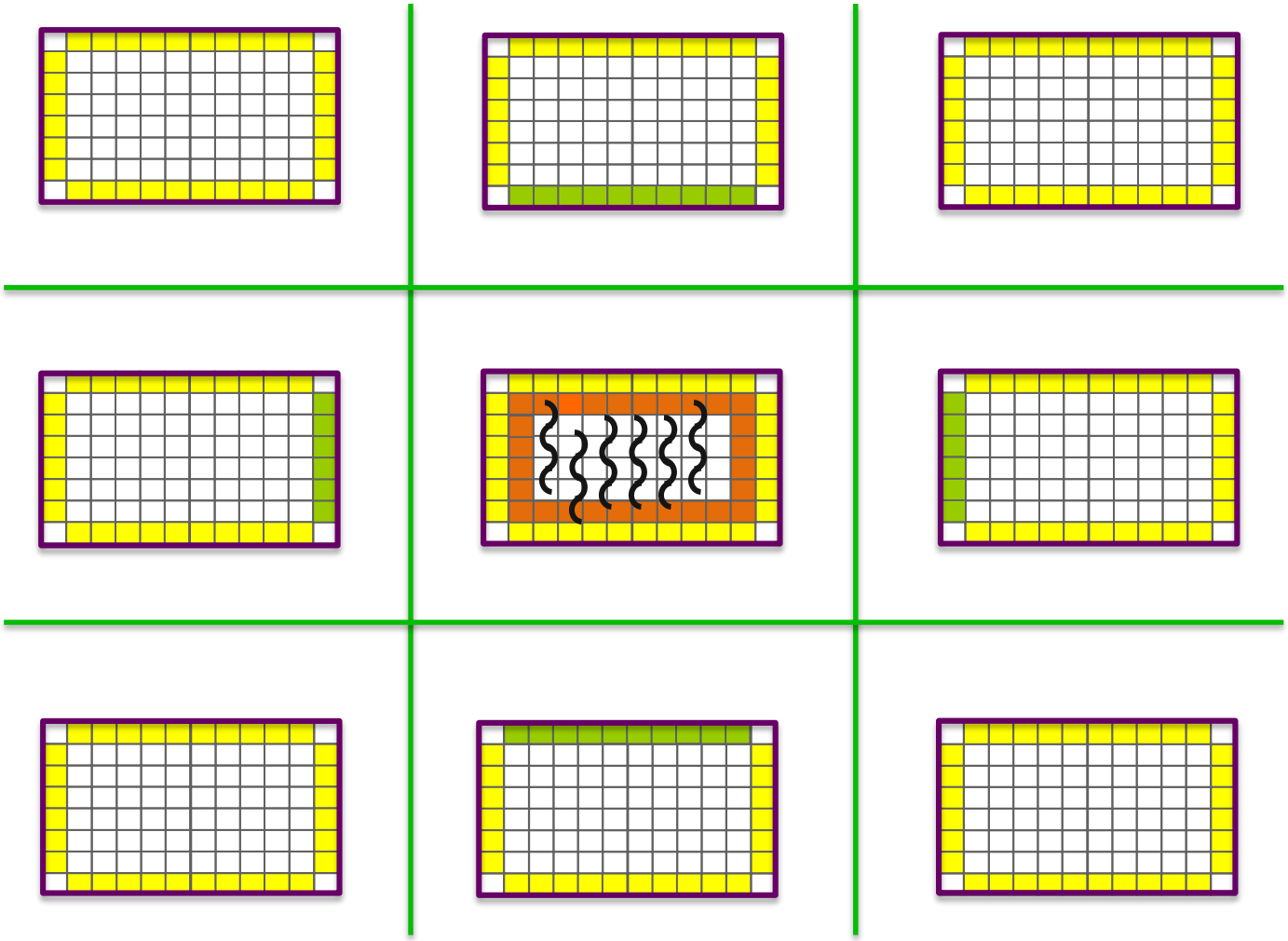
```
int main(int argc, char ** argv)
{
    int buf[100], provided;

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

#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
  - 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)*

Implementing Stencil Computation using MPI_THREAD_FUNNELED



Code Examples

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

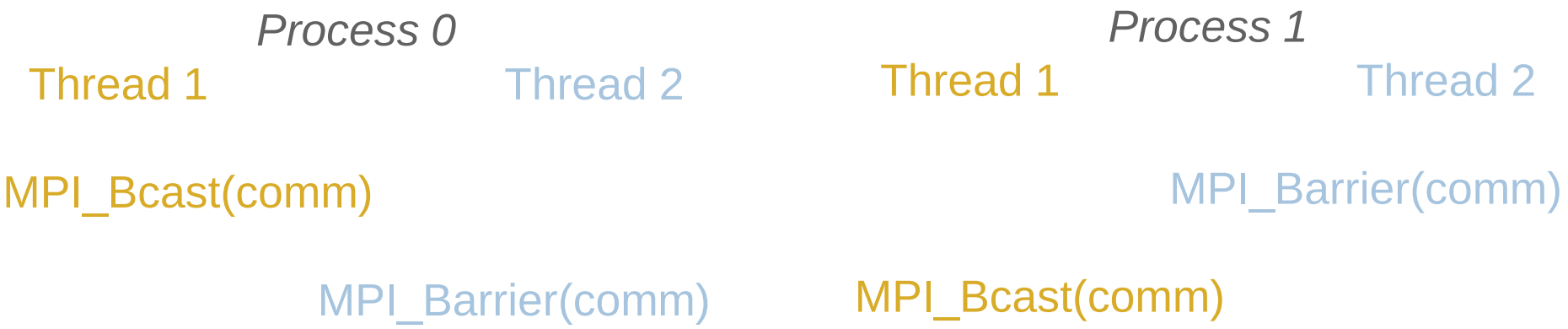
MPI Semantics and MPI_THREAD_MULTIPLE

- ***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 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

MPI_Bcast(comm)

MPI_Comm_free(comm)

Ordering in MPI_THREAD_MULTIPLE: Incorrect Example with Object Management

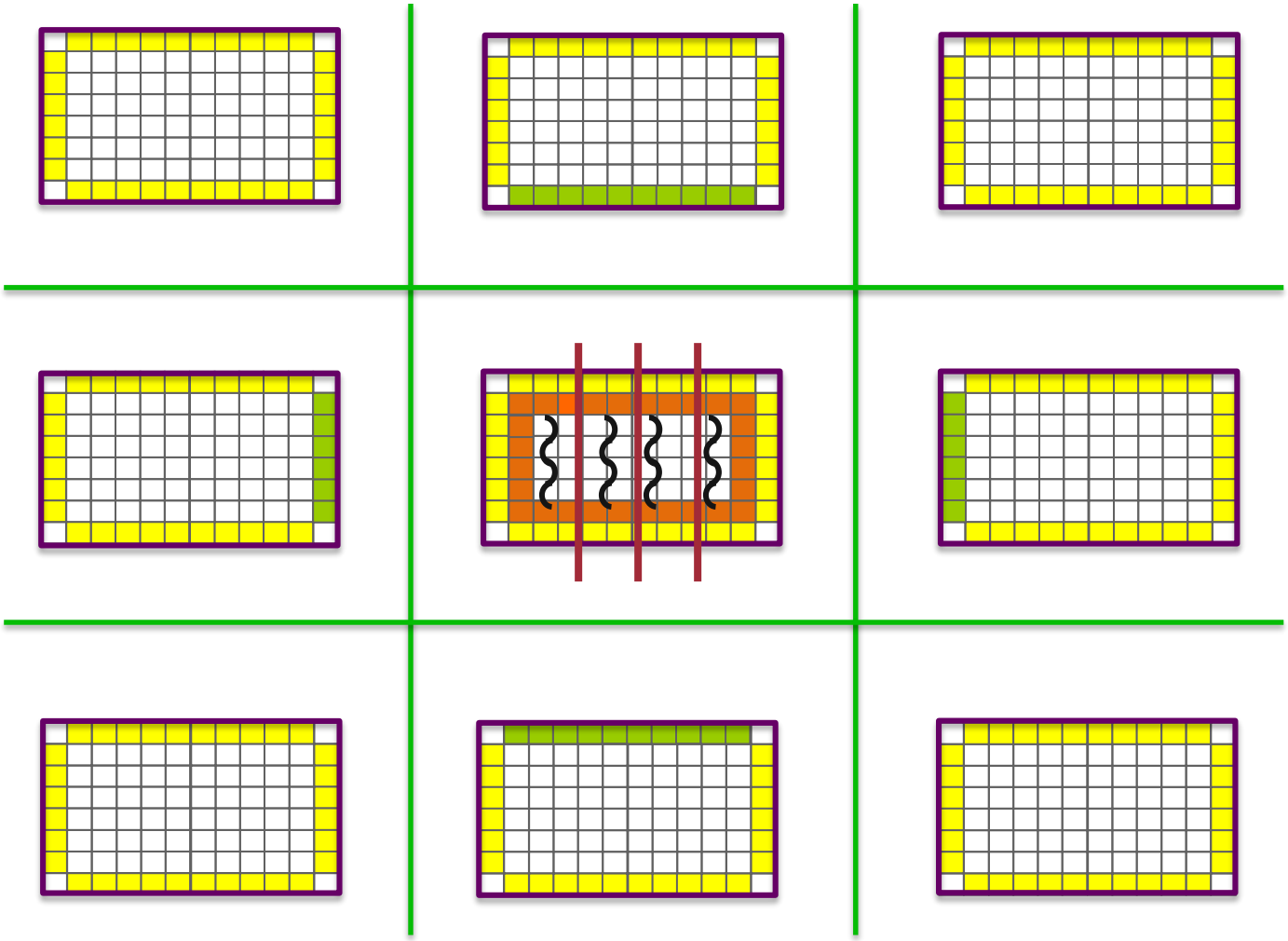
- The user has to make sure that one thread is not using an object while another thread is freeing it
  - This is essentially an ordering issue; the object might get freed before it is used

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.

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

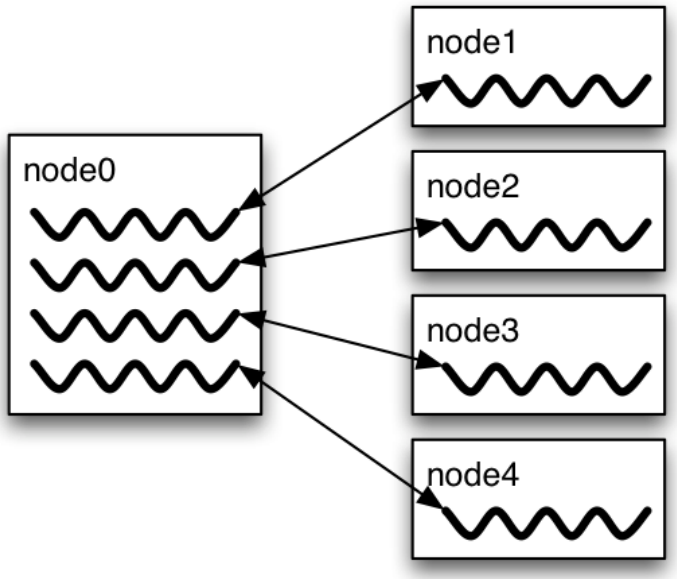
The Current Situation

- All MPI implementations support MPI_THREAD_SINGLE
- 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 in between loops) only need FUNNELED
  - So don't need "thread-safe" MPI for many hybrid programs
  - But watch out for Amdahl's Law!

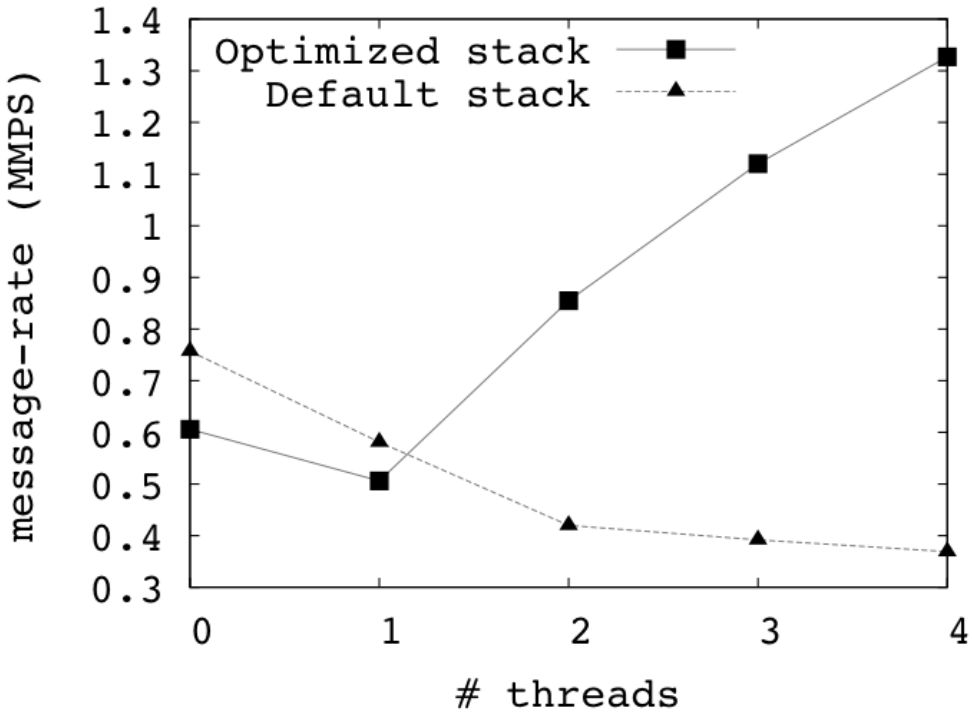
Performance with MPI_THREAD_MULTIPLE

- Thread safety does not come for free
- The implementation must access/modify several shared objects (e.g. message queues) in a consistent manner
- 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, thread-safety is required to ensure a consistent state of the queue – 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)

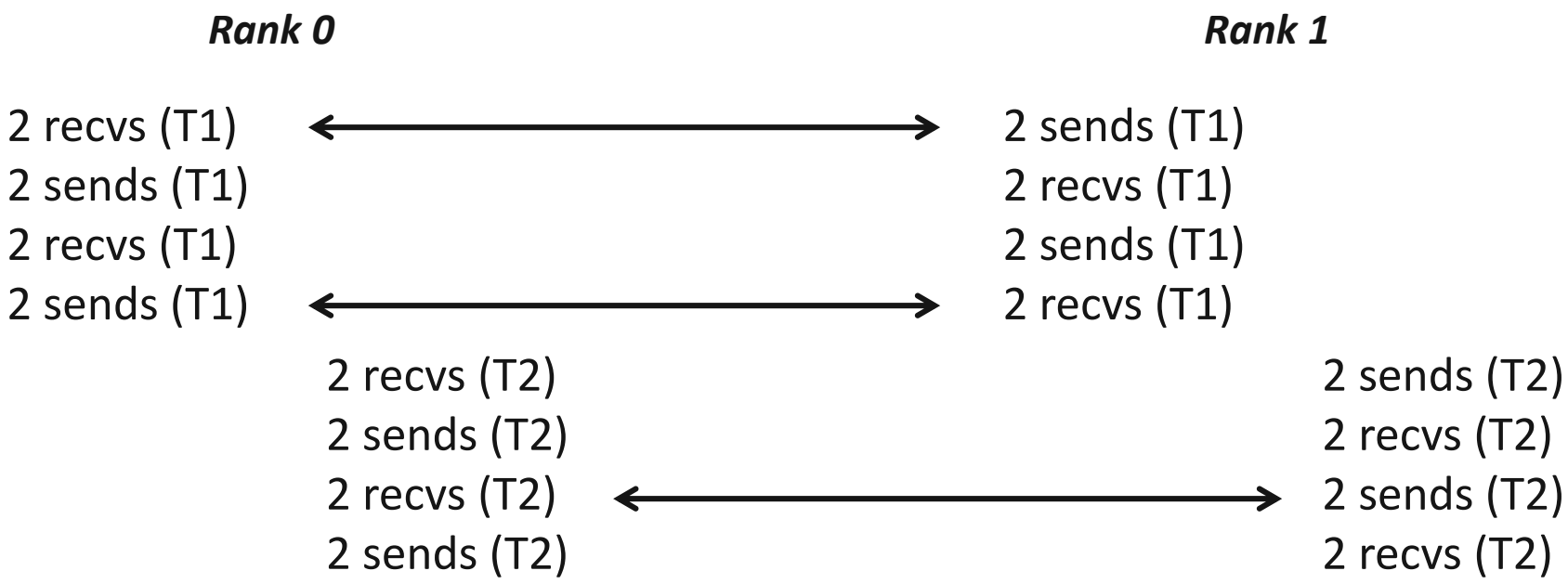
An Example we encountered

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

2 Proceses, 2 Threads, Each Thread Executes this Code

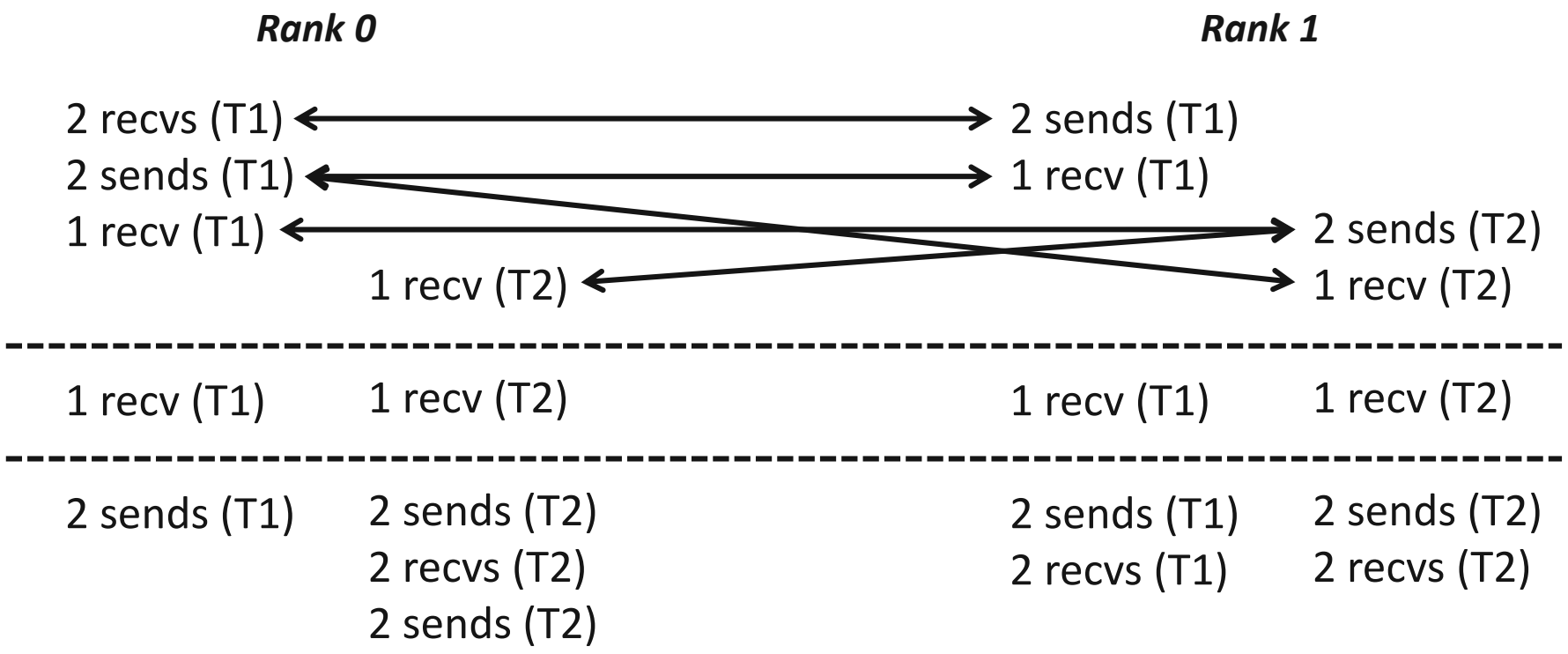
```
for (j = 0; j < 2; j++) {
    if (rank == 1) {
        for (i = 0; i < 2; i++)
            MPI_Send(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
        for (i = 0; i < 2; i++)
            MPI_Recv(NULL, 0, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &stat);
    }
    else {  /* rank == 0 */
        for (i = 0;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

Intended Ordering of Operations



- Every send matches a receive on the other rank

Possible Ordering of Operations in Practice



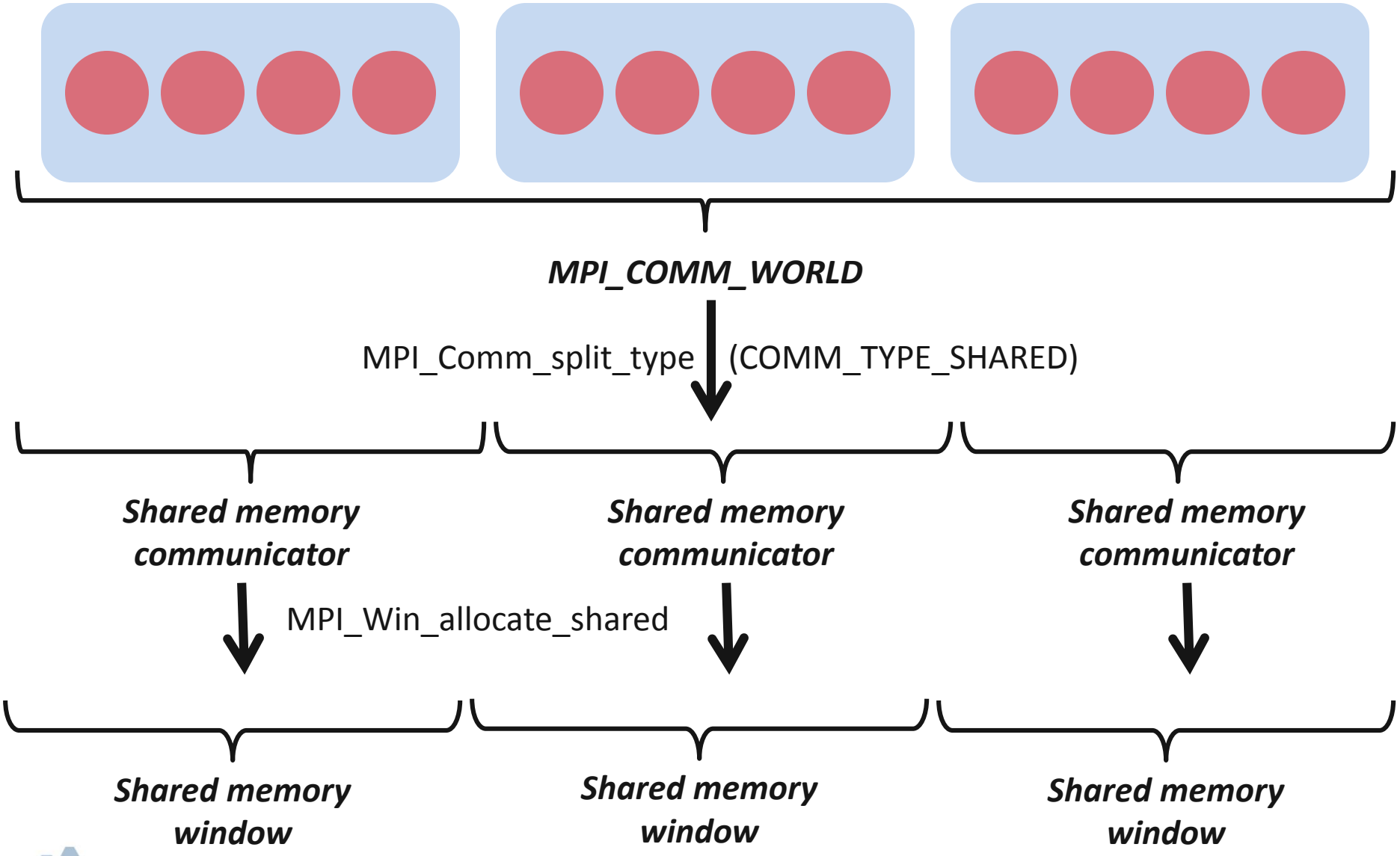
- Because the MPI operations can be issued in an arbitrary order across threads, all threads could block in a RECV call

MPI + Shared-Memory

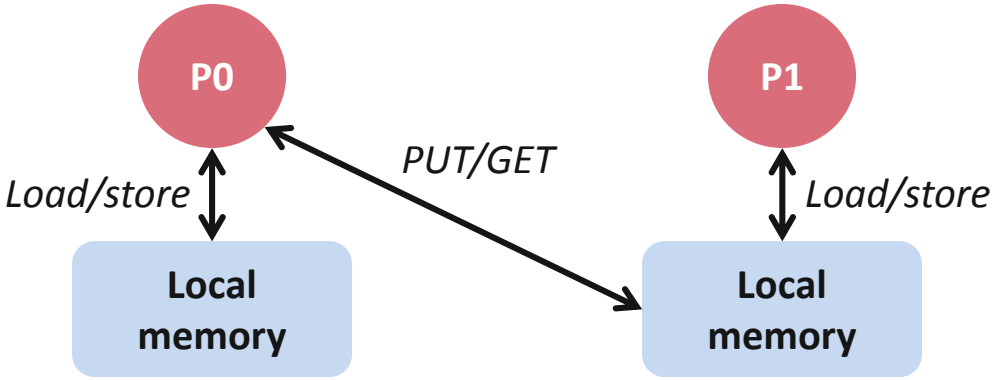
Hybrid Programming with Shared Memory

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

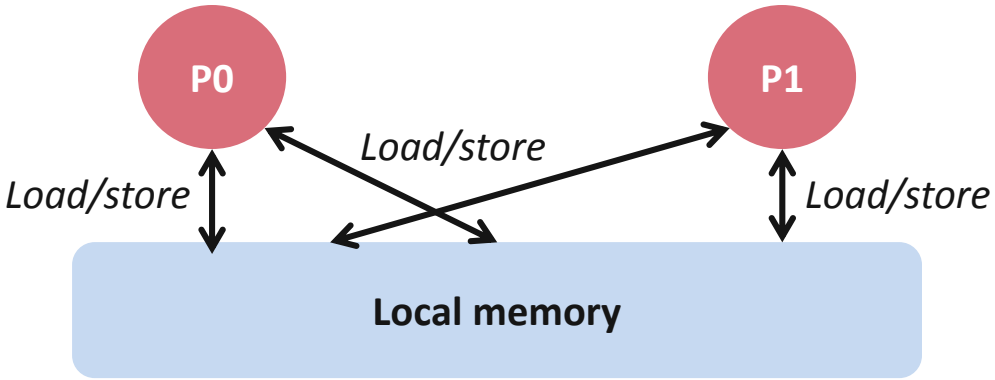
Creating Shared Memory Regions in MPI



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)

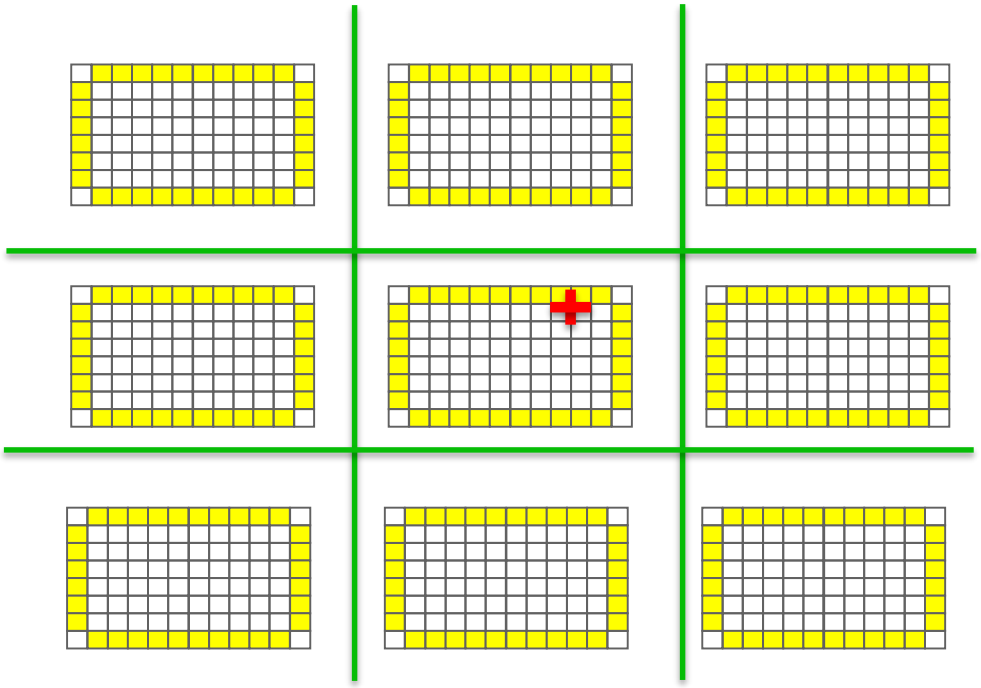
Shared Arrays with Shared memory windows

```
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

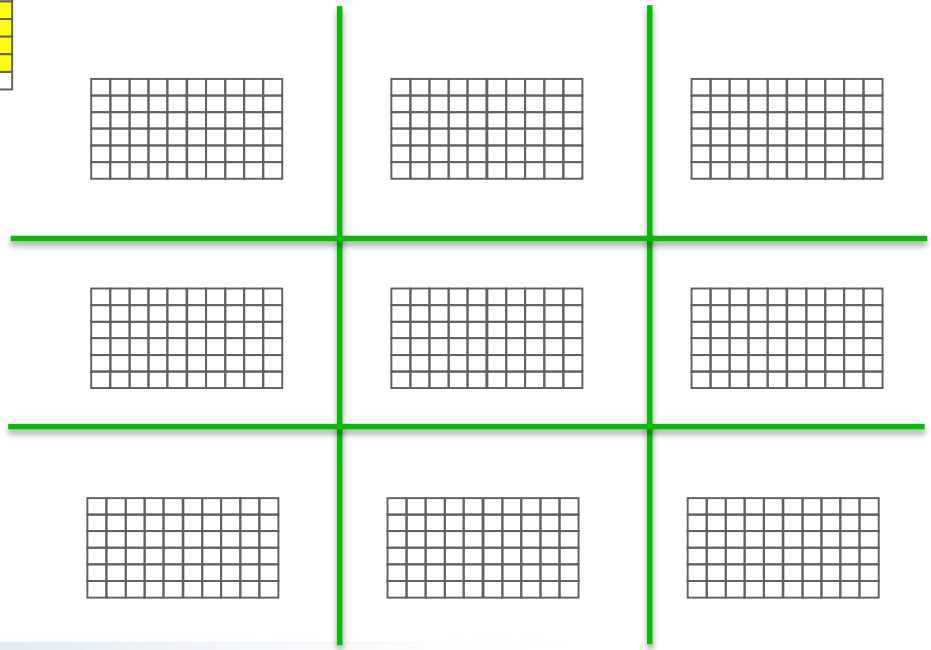
- Shared memory allocation does not need to be uniform across processes
  - Processes can allocate a different amount of memory (even zero)
- The MPI standard does not specify where the memory would be placed (e.g., which physical memory it will be pinned to)
  - Implementations can choose their own 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 Computation: Stencil



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

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



Walkthrough of 2D Stencil Code with Shared Memory Windows

- *stencil_mpi_shmem.c*

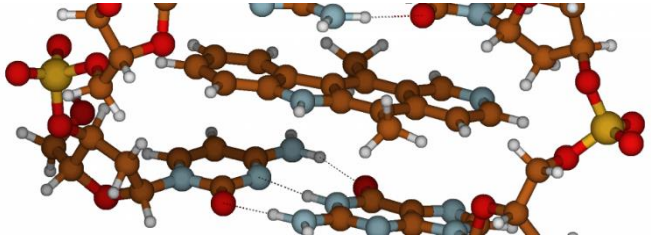
Which Hybrid Programming Method to Adopt?

- 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

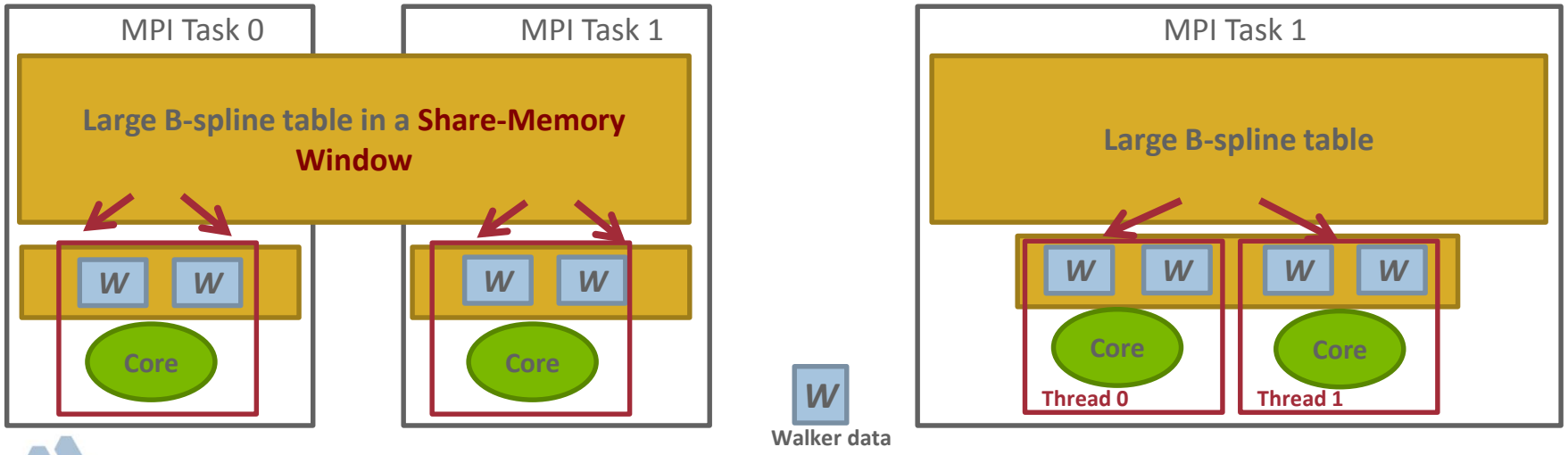
- Memory capacity bound with MPI-only
- Hybrid approaches
  - MPI + threads (e.g. X = OpenMP, Pthreads)
  - MPI + shared-memory (X = MPI)
- Can use direct load/store operations instead of message passing

QMCPACK



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

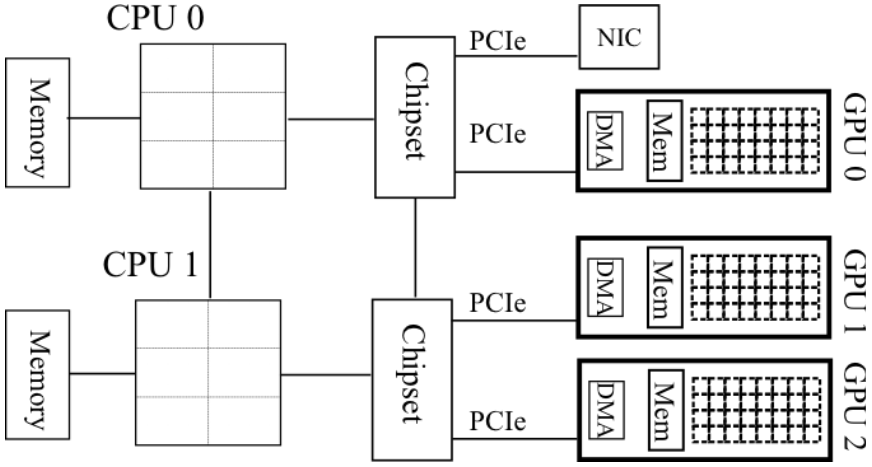
MPI + Theads
- Share everything by default
- Privatize data when necessary



MPI + Accelerators

Accelerators in Parallel Computing

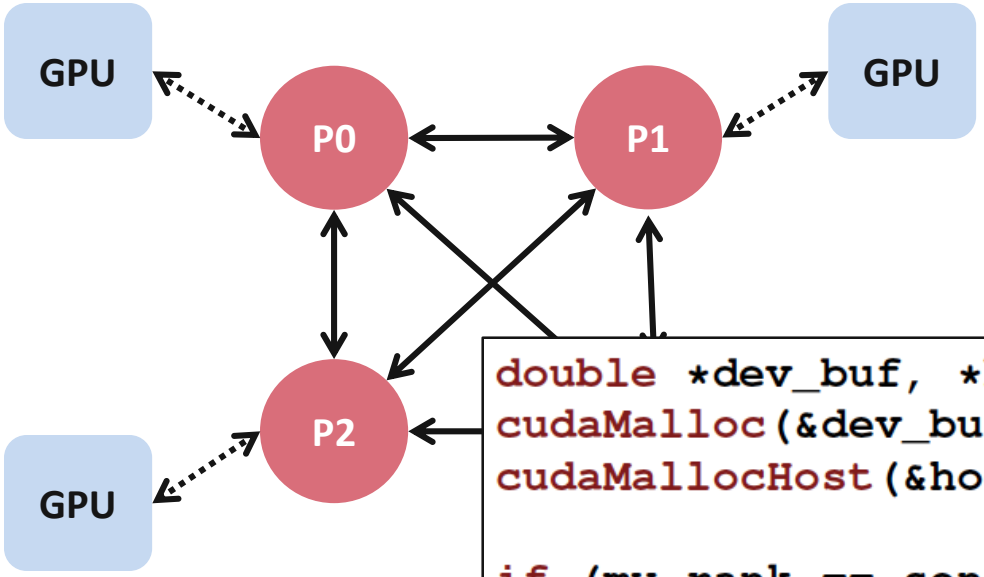
- General purpose, highly parallel processors
  - High FLOPs/Watt and FLOPs/$
  - Unit of execution *Kernel*
  - Separate memory subsystem
  - Prog. Models: CUDA, OpenCL, …
- Clusters with accelerators are becoming common
- New programmability and performance challenges for programming models and runtime 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-3.1)
  - Could be incorporated in a future version of the standard

Argonne NATIONAL LABORATORY

Advanced Topics: Network Locality and Topology Mapping

ETH Zürich

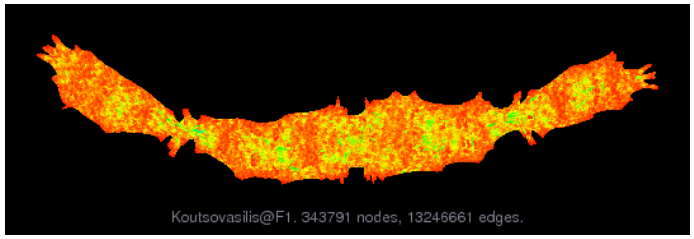
Topology Mapping and Neighborhood Collectives

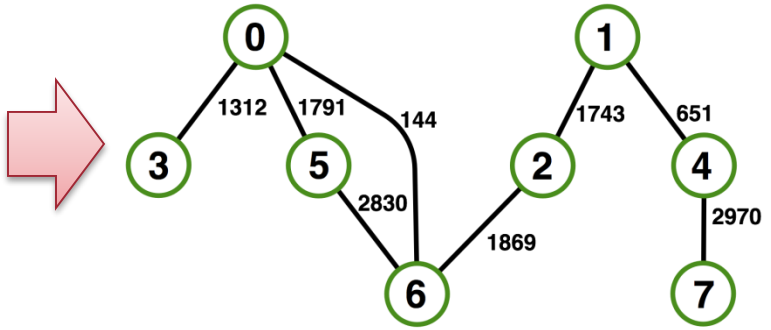
- Topology mapping basics
  - Allocation mapping vs. rank reordering
  - Ad-hoc solutions vs. portability
- MPI topologies
  - Cartesian
  - Distributed graph
- Collectives on topologies – neighborhood collectives
  - Use-cases

Topology Mapping Basics

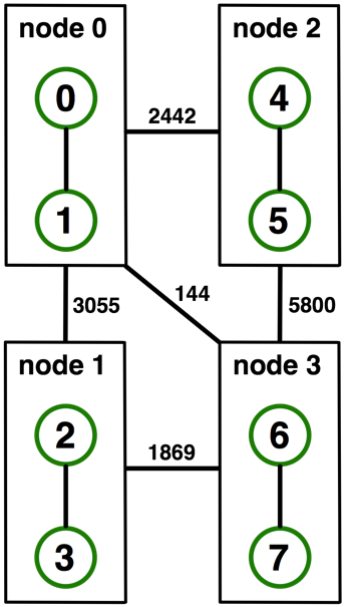
- MPI supports rank reordering
  - Change numbering in a given allocation to reduce congestion or dilation
  - Sometimes automatic (early IBM SP machines)
- Properties
  - Always possible, but effect may be limited (e.g., in a bad allocation)
  - Portable way: MPI process topologies
    - Network topology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

Example: On-Node Reordering



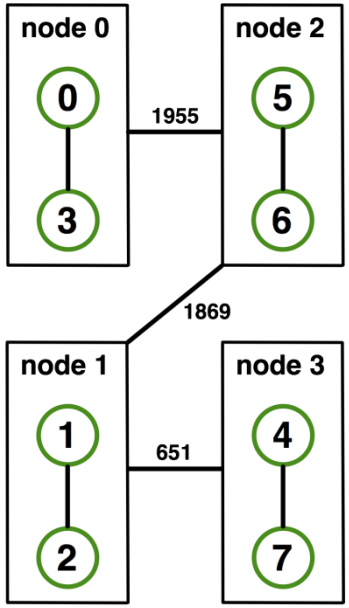


Naïve Mapping

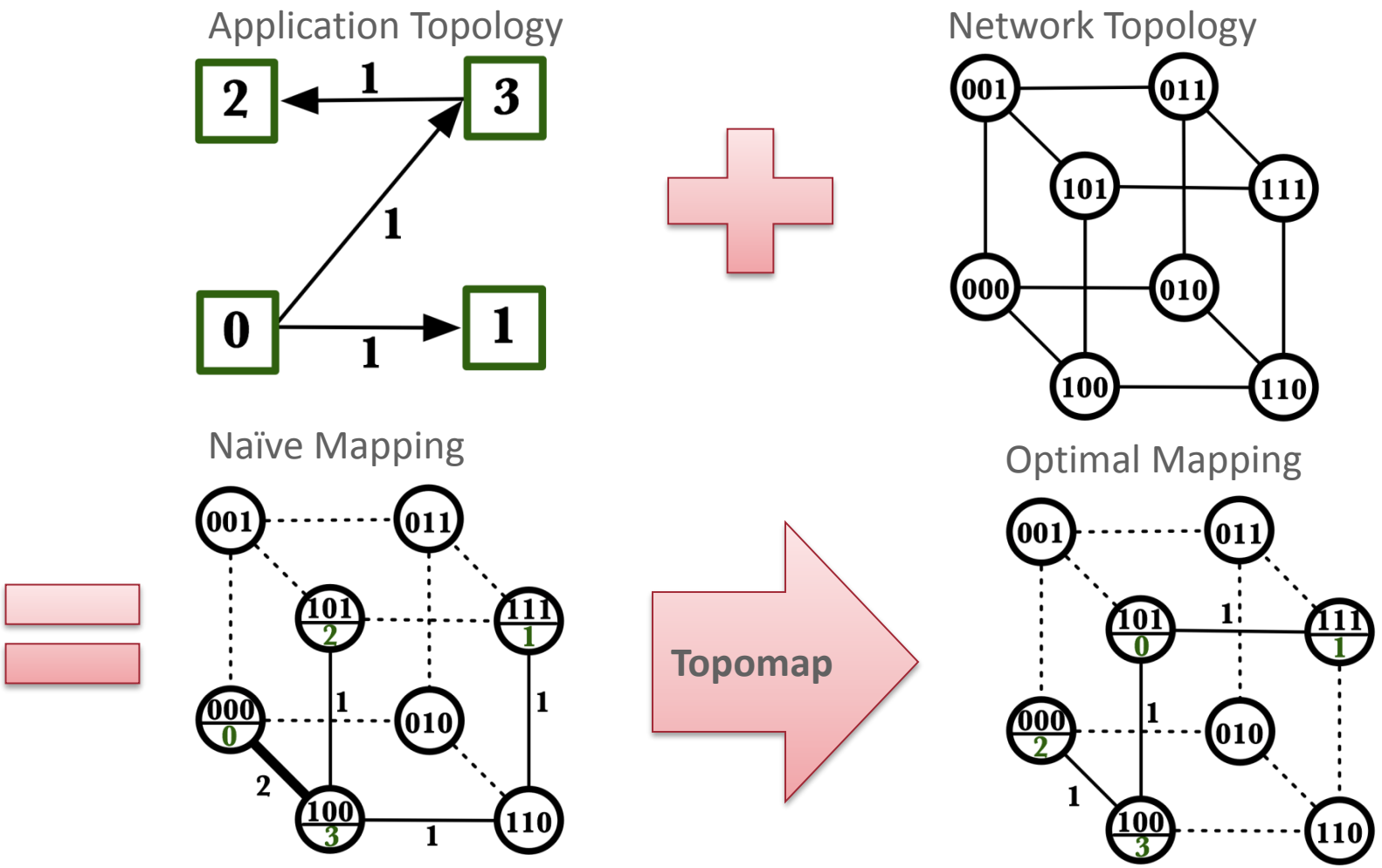


Topomap

Optimized Mapping



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

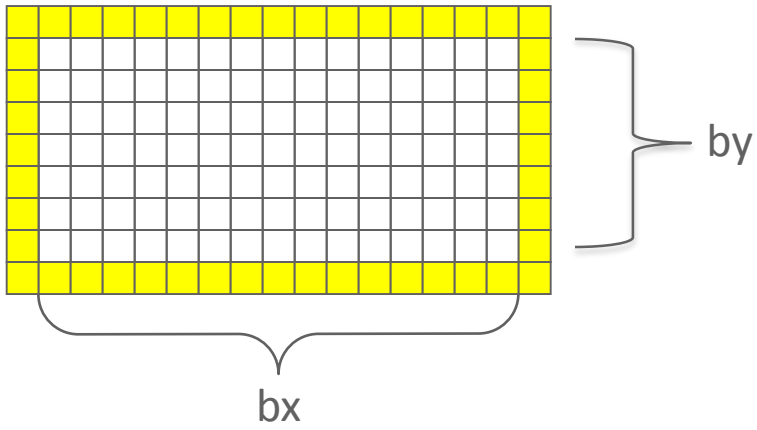
Cartesian Communication Helpers

```
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

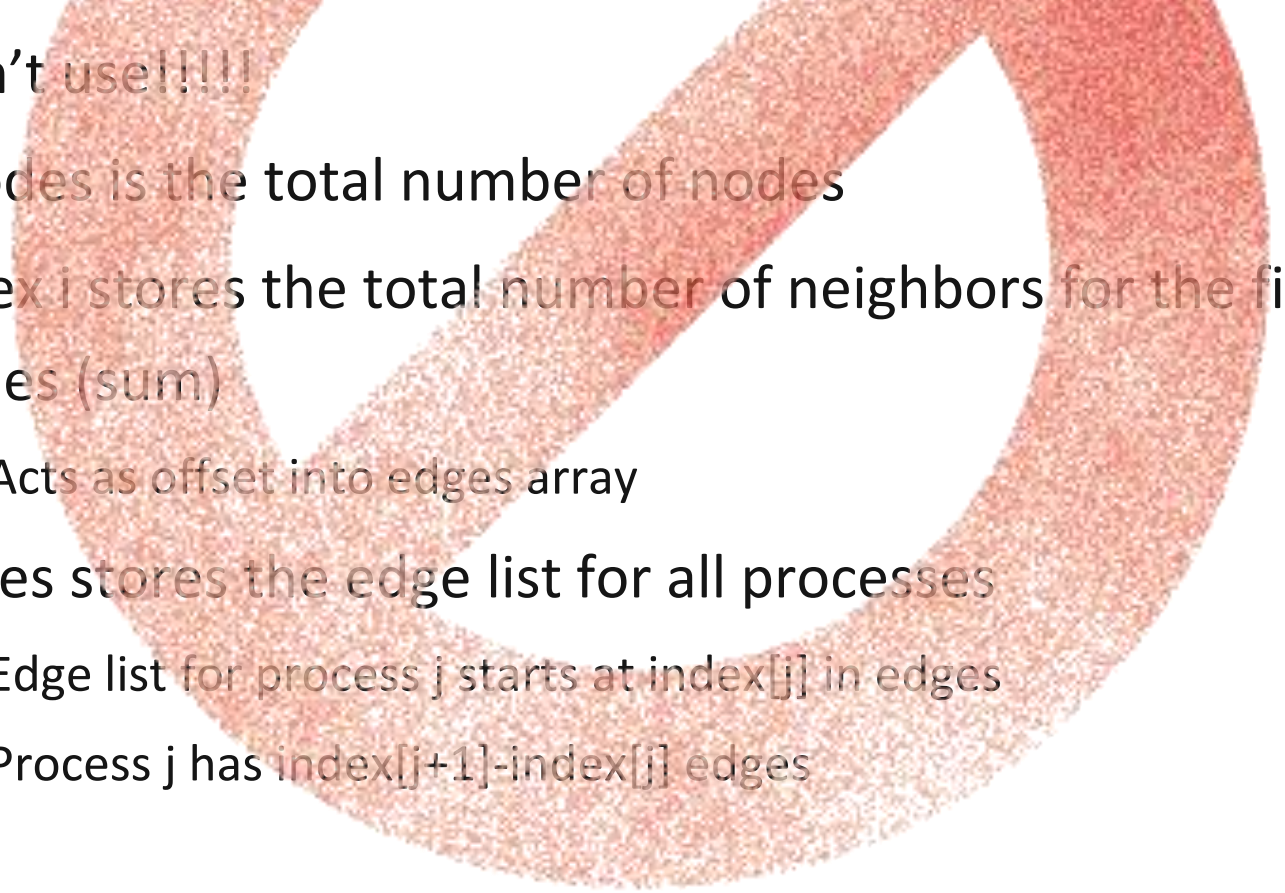
Code Example

- *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

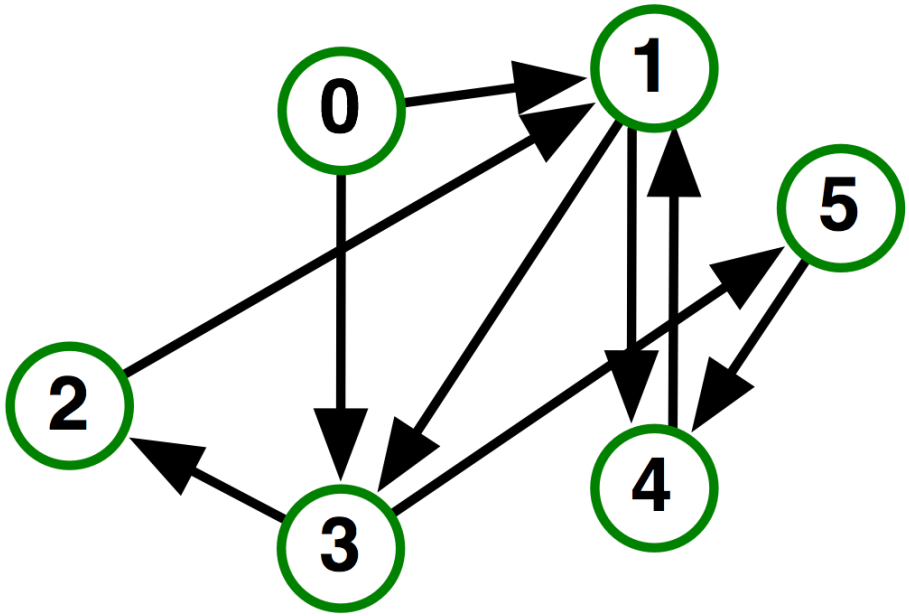
MPI_Dist_graph_create_adjacent

```
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}
- …



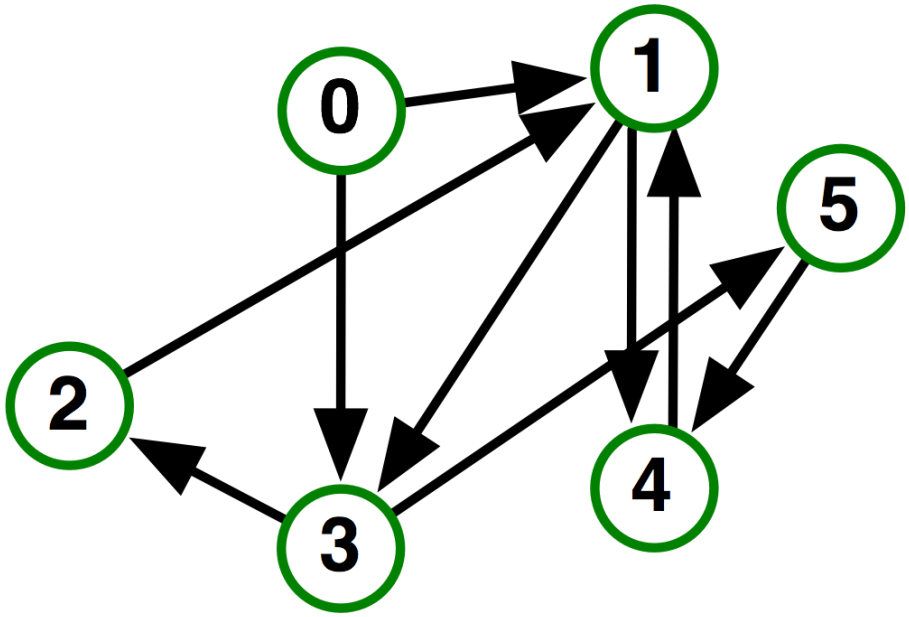
MPI_Dist_graph_create

```
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

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

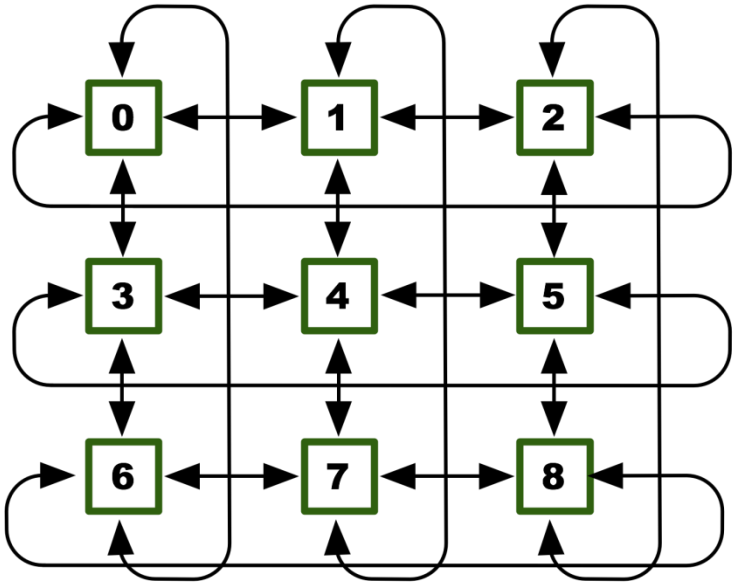
- Topologies implement no communication!
  - Just helper functions
- Collective communications only cover some patterns
  - E.g., no stencil pattern
- Several requests for “build your own collective” functionality in MPI
  - Neighborhood collectives are a simplified version
  - Cf. Datatypes for communication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

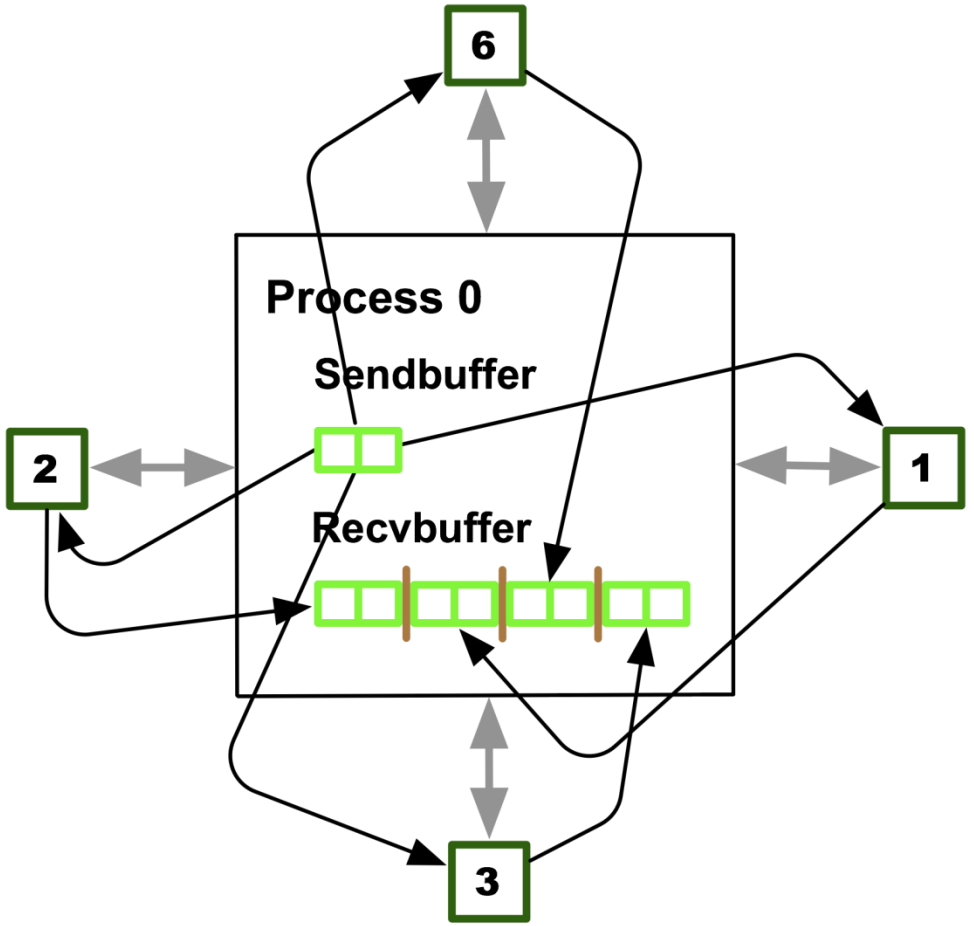
Cartesian Neighborhood Collectives

- 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!

Code Example

- *stencil_mpi_carttopo_neighcolls.c*
- Adds neighborhood collectives to the topology

Argonne NATIONAL LABORATORY

What's next
Towards MPI 4.0

Planned/Proposed Extensions

ETH Zürich

Introduction

- The MPI Forum continues to meet once 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: C11 bindings for large counts

Argonne NATIONAL LABORATORY

Point-to-Point Working Group

ETH Zürich

Current Topics

- Streaming communication
  - On hold
- Batched communication
  - Initial proposal
- Allocate receive
  - On hold
- Receive reduce/accumulate
  - On hold
- Communication relaxation hints
  - Active discussion

What is an MPI Stream?

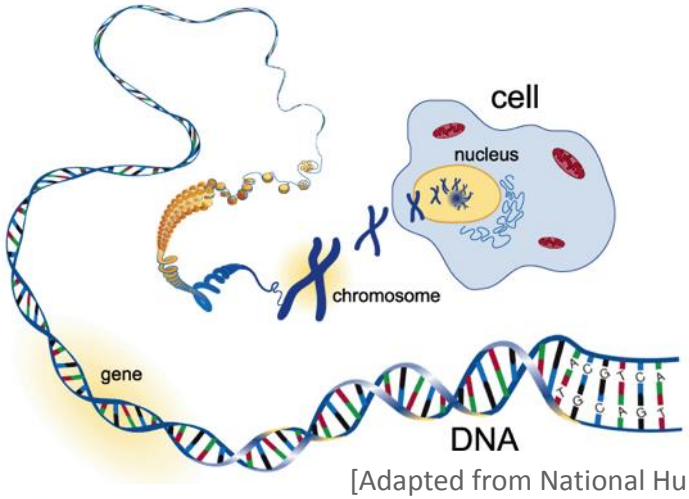
- From single sender to single receiver only
  - Joined by an existing communicator
- Ordered and reliable
- Sender can send any amount of data
- Received can receive any amount of data
  - (up to what is available)

Discussion issues with MPI streams

- Datatypes as the unit of transmission
  - Normal message boundaries would be ignored
- Flow-control/buffering
  - E.g., receiver consistently slower than sender
- Allow buffer underrun or block receiver?
  - E.g., receiver wants 33 integers, but only 16 are available
- Performance benefits discussion

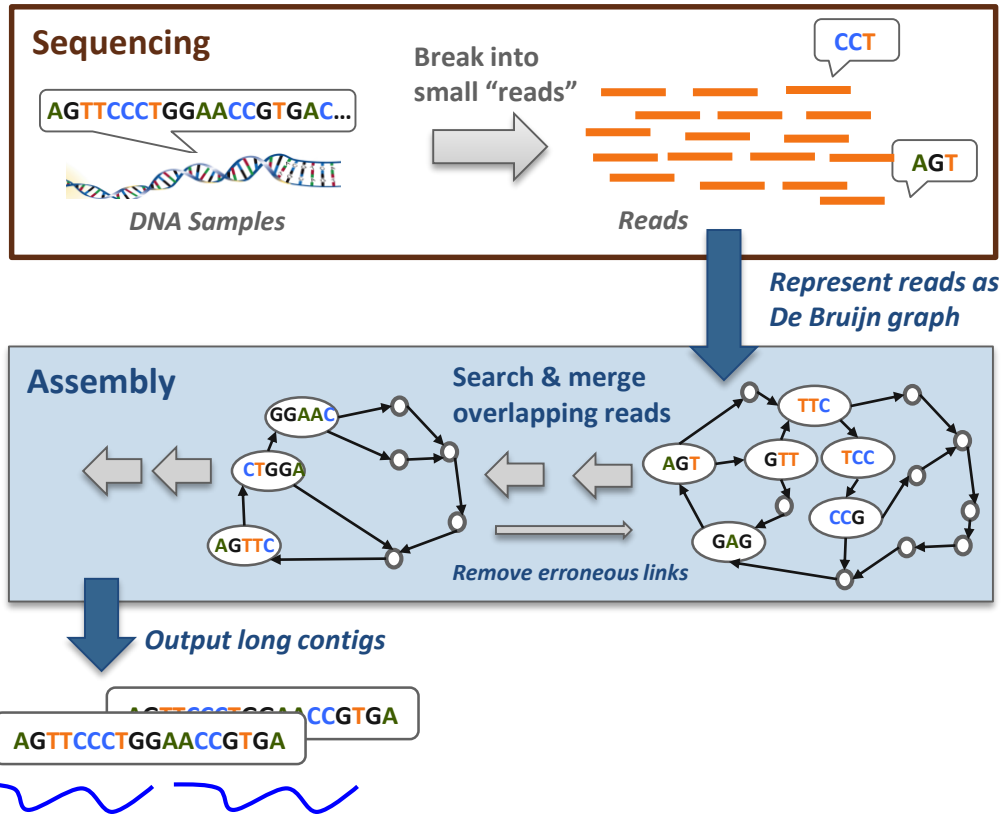
Genome Assembly

- Genome analysis
  - Sequence alignment
  - **Sequence assembly**
    - **Reconstruct** long DNA sequences by merging many small fragments
  - Gene mapping



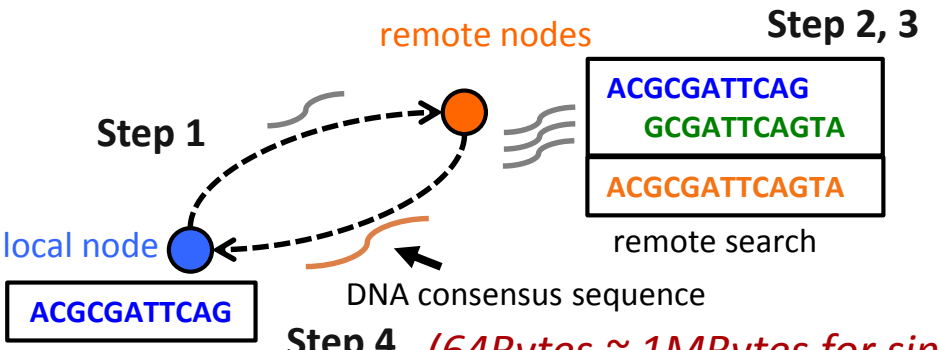
[Adapted from National Human Genome Research Institute]

Hard to read whole genomes in current sequencing technology. Instead, read many small fragments , called "**reads**".



Massive Data Movement in SWAP-Assembly

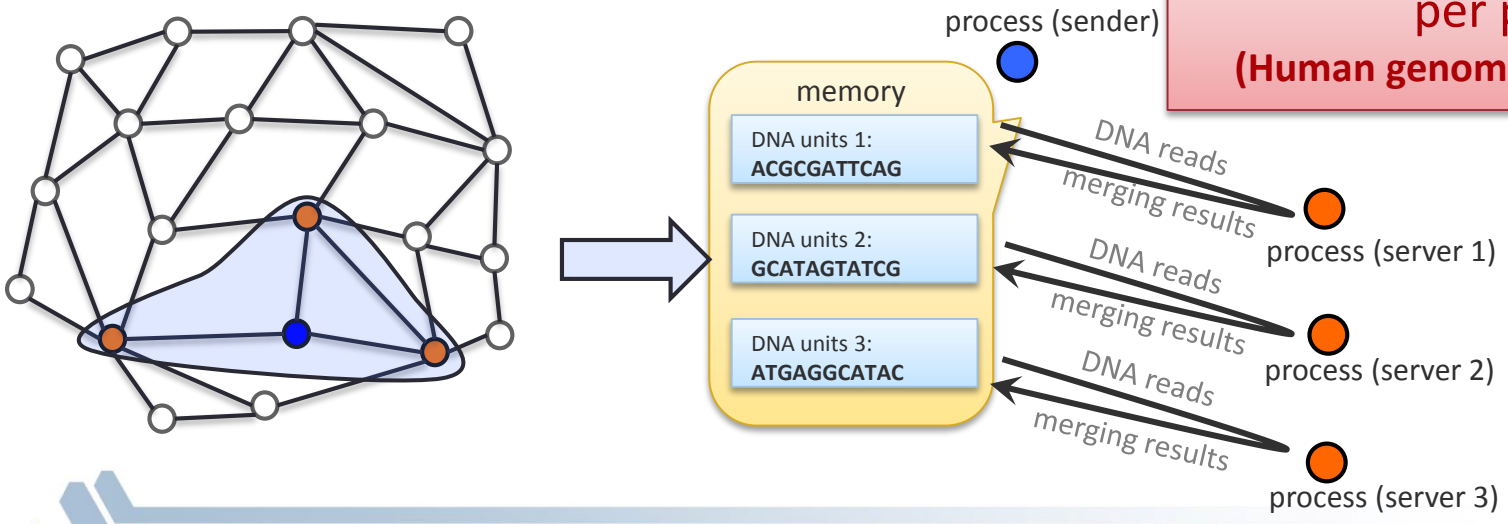
Basic edge merging algorithm



1. **Send local DNA unit to that node;**
2. **Search matching unit on that node;**
3. **Merge two units on that node;**
4. **Return merged unit.**

(64Bytes ~ 1MBytes for single message)

Large amount of outstanding data movement



10^6+ outstanding messages per process
(Human genome on Cray Edison *)

*** 64GB memory per node, 1KB memory per DNA reads, exclude runtime memory consumption.**

Issues with Traditional MPI_Isend/MPI_Irecv

- Each operation creates a new request object
- MPI library runs out of request objects after a few thousand operations
- Application cannot issue a lot of messages to fully utilize the network

Batched Communication Operations

- Ability to batch multiple operations into a single request object
  - MPI_Request_batch_init
  - MPI_Isend_batch, MPI_Irecv_batch, …
- Proportionally reduced number of requests
- Can allow applications to consolidate multiple completions into a single request

Allocate Receive

- MPI_Arecv: the receive buffer is an output argument instead of an input argument, and the implementation allocates that memory internally
- Allows implementation to allocate memory for the size of the message, eliminates buffering overhead when message size is not known a priori
- Allows copy-free implementation of unexpected messages using an eager-like protocol

Receive reduce/accumulate

- MPI_Recv_{reduce,accumulate}: the incoming data is reduced/accumulated onto the receive buffer.
- Matches a common application pattern during boundary element exchange and allows implementation to minimize buffering in this case and potentially do more efficiently.
- Useful for creating user-defined, potentially dynamic reduction trees, without graph communicators.
- May allow for more efficient implementation of some forms of active-messages.

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

Meeting Details

- Teleconference calls
  - Fortnightly on Monday at 11:00 central US
- Email list:
  - mpiwg-p2p@lists.mpi-forum.org
- Face-to-face meetings
  - http://meetings.mpi-forum.org/Meeting_details.php

Argonne NATIONAL LABORATORY

Fault Tolerance Working Group

ETH Zürich

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…

What is the working group doing?

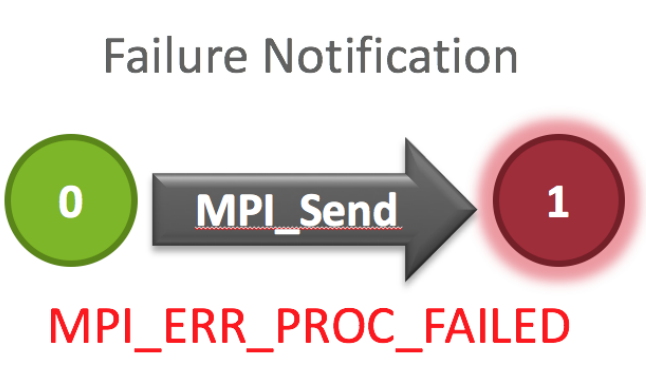
- Decide the best way forward for fault tolerance in MPI.
  - Currently looking at User Level Failure Mitigation (ULFM), but that's only part of the puzzle.
- Look at all parts of MPI and how they describe error detection and handling.
  - Error handlers probably need an overhaul
  - Allow clean error detection even without recovery
- Consider alternative proposals and how they can be integrated or live alongside existing proposals.
  - Reinit, FA-MPI, others
- Start looking at the next thing
  - Data resilience?

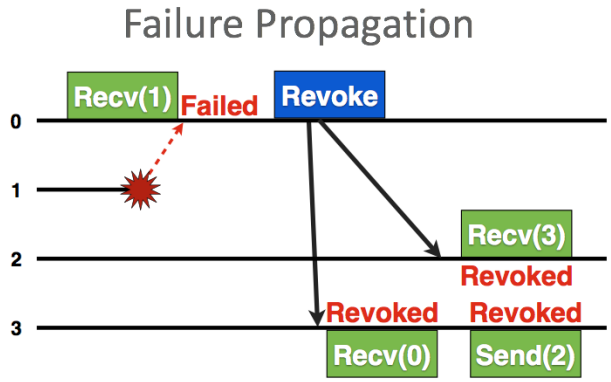
Noncatastrophic Errors

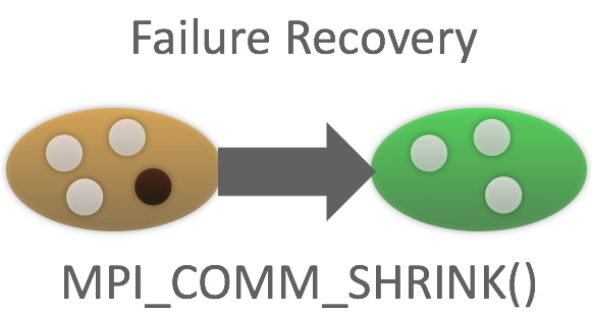
- Currently the state of MPI is undefined if any error occurs
- Even simple errors such as arguments are incorrect, can cause the state of MPI to be undefined
- Noncatastrophic errors are an opportunity for the MPI implementation to define some errors as “ignorable”
- For an error, the user can query 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

User Level Failure Mitigation Main Ideas

- 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)







Is ULFM the only way?

- No!
  - Fenix, presented at SC '14 provides more user friendly semantics on top of MPI/ULFM
- Other research discussions include
  - Reinit (LLNL) - Fail fast by causing the entire application to roll back to MPI_INIT with the original number of processes.
  - FA-MPI (Auburn/UAB) - Transactions allow the user to use parallel try/catch-like semantics to write their application.
    - Paper in the SC '15 Proceedings (ExaMPI Workshop)
- Some of these ideas fit with ULFM directly and others require some changes
  - We're working with the Tools WG to revamp PMPI to support multiple tools/libraries/etc. which would enable nice fault tolerance semantics.

How Can I Participate?

Website: http://www.github.com/mpiwg-ft

Email: mpiwg-ft@lists.mpi-forum.org

Conference Calls: Every other Tuesday at 3:00 PM Eastern US

In Person: MPI Forum Face To Face Meetings

Argonne NATIONAL LABORATORY

Hybrid Programming Working Group

ETH Zürich

MPI Forum Hybrid WG Goals

- Ensure interoperability of MPI with other programming models
  - MPI+threads (pthreads, OpenMP, user-level threads)
  - MPI+CUDA, MPI+OpenCL
  - MPI+PGAS models

MPI-3.1 Performance/Interoperability Concerns

- Resource sharing between MPI processes
  - System resources do not scale at the same rate as processing cores
    - Memory, network endpoints, TLB entries, …
    - Sharing is necessary
  - MPI+threads gives a method for such sharing of resources
- Performance Concerns
  - MPI-3.1 provides a single view of the MPI stack to all threads
    - Requires all MPI objects (requests, communicators) to be shared between all threads
    - Not scalable to large number of threads
    - Inefficient when sharing of objects is not required by the user
  - MPI-3.1 does not allow a high-level language to interchangeably use OS processes or threads
    - No notion of addressing a single or a collection of threads
    - Needs to be emulated with tags or communicators

Single view of MPI objects

- MPI-3.1 specification requirements
  - It is valid in MPI to have one thread generate a request (e.g., through MPI_IRECV) and another thread wait/test on it
  - One thread might need to make progress on another's requests
  - Requires all objects to be maintained in a shared space
  - When a thread accesses an object, it needs to be protected through locks/atomics
    - Critical sections become expensive with hundreds of threads accessing it
- Application behavior
  - Many (but not all) applications do not require such sharing
  - A thread that generates a request is responsible for completing it
    - MPI guarantees are safe, but unnecessary for such applications

P0 (Thread 1)

```
MPI_Irecv(…, comm1, &req1);
pthread_barrier();

pthread_barrier();
MPI_Wait(&req1, …);
```

P0 (Thread 2)

```
MPI_Irecv(…, comm2, &req2);
pthread_barrier();
MPI_Wait(&req2, …);
pthread_barrier();
```

P1

```
MPI_Ssend(…, comm1);
MPI_Ssend(…, comm2);
```

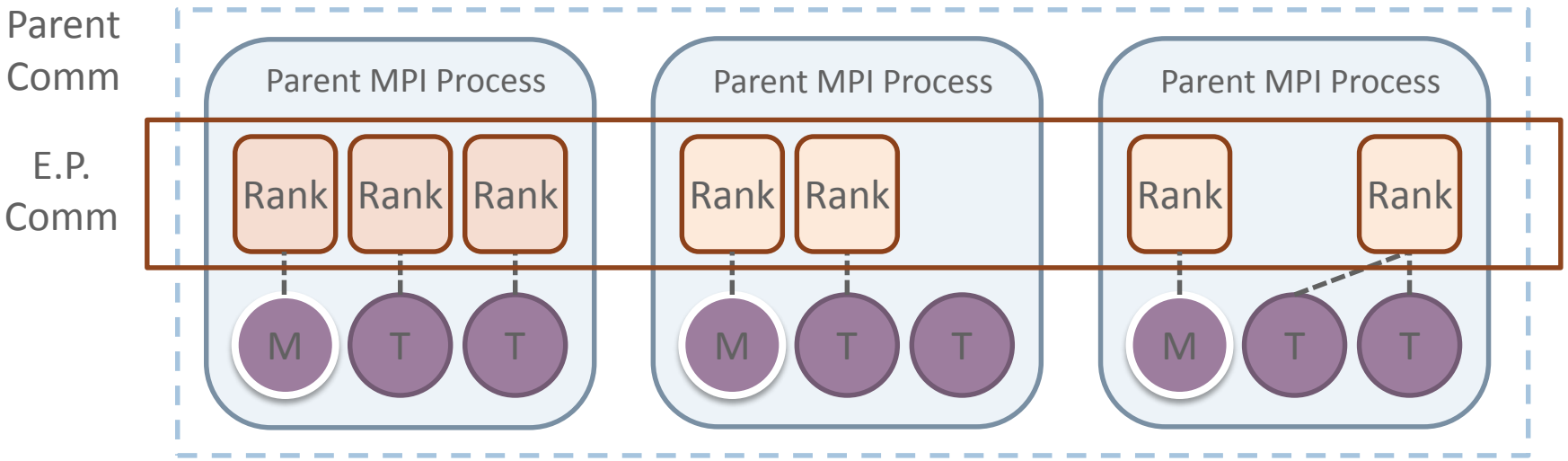
Interoperability with High-level Languages

- In MPI-3.1, there is no notion of sending a message to a thread
  - Communication is with MPI processes – threads share all resources in the MPI process
  - You can emulate such matching with tags or communicators, but some pieces (like collectives) become harder and/or inefficient
- Some high-level languages do not expose whether their processing entities are processes or threads
  - E.g., PGAS languages
- When these languages are implemented on top of MPI, the language runtime might not be able to use MPI efficiently

MPI Endpoints: Proposal for MPI-4

- Idea is to have multiple addressable communication entities within a single process
  - Instantiated in the form of multiple ranks per MPI process
- Each rank can be associated with one or more threads
- Lesser 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)

MPI Endpoints Semantics

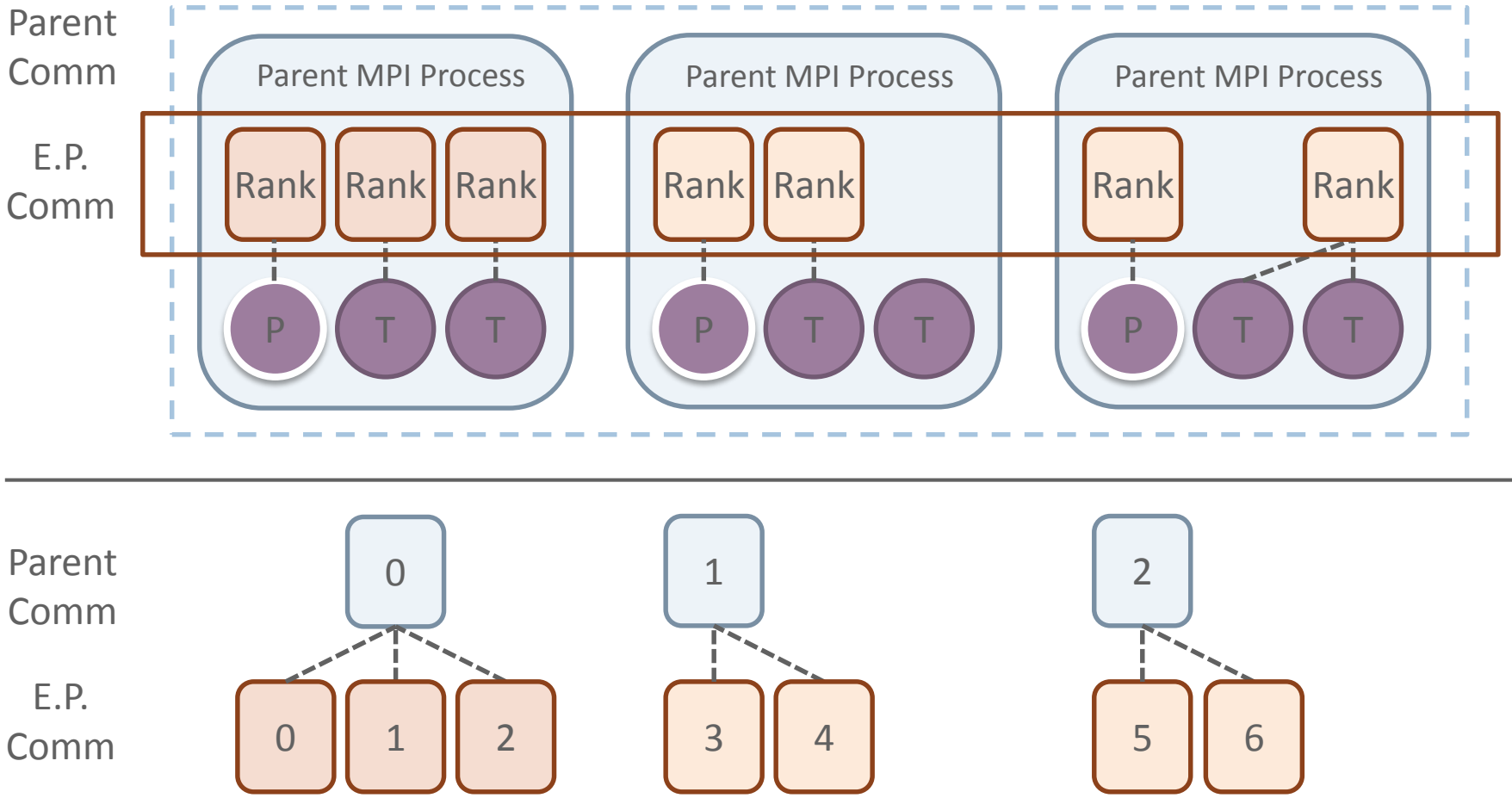



```
MPI_Comm_create_endpoints(MPI_Comm parent_comm, int my_num_ep,
        MPI_Info info, MPI_Comm out_comm_handles[])
```

- Creates new MPI ranks from existing ranks in parent communicator
  - Each process in parent comm. requests a number of endpoints
  - Array of output handles, one per local rank (i.e. endpoint) in endpoints communicator
  - Endpoints have MPI process semantics (e.g. progress, matching, collectives, …)
- Threads using endpoints behave like MPI processes
  - Provide per-thread communication state/resources
  - Allows implementation to provide process-like performance for threads

MPI Endpoints

Relax the 1-to-1 mapping of ranks to threads/processes



Hybrid MPI+OpenMP Example
With Endpoints

```
int main(int argc, char **argv) {
  int world_rank, tl;
  int max_threads = omp_get_max_threads();
  MPI_Comm ep_comm[max_threads];

  MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &tl);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

#pragma omp parallel
  {
    int nt = omp_get_num_threads();
    int tn = omp_get_thread_num();
    int ep_rank;
#pragma omp master
    {
      MPI_Comm_create_endpoints(MPI_COMM_WORLD, nt, MPI_INFO_NULL, ep_comm);
    }
#pragma omp barrier
    MPI_Comm_rank(ep_comm[tn], &ep_rank);
    ... // Do work based on 'ep_rank'
    MPI_Allreduce(..., ep_comm[tn]);

    MPI_Comm_free(&ep_comm[tn]);
  }
  MPI_Finalize();
}
```

Additional Notes

- Useful for more than just avoiding locks
  - Semantics that are "rank-specific" become more flexible
    - E.g., ordering for operations from a process
    - Ordering constraints for MPI RMA accumulate operations
- Supplementary proposal on thread-safety requirements for endpoint communicators
  - Is each rank only accessed by a single thread or multiple threads?
  - Might get integrated into the core proposal
- Implementation challenges being looked into
  - Simply having endpoint communicators might not be useful, if the MPI implementation has to make progress on other communicators too

More Info

- Endpoints:
  - https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/380
- Hybrid Working Group:
  - https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/MPI3Hybrid

Argonne NATIONAL LABORATORY

Persistence Working Group

ETH Zürich

Persistent Collective Operations

- An all-to-all transfer is done many times in an application
- The specific sends and receives represented never change (size, type, lengths, transfers)
- A nonblocking persistent collective operation can take the time to apply a heuristic and choose a faster way to move that data
- Fixed cost of making those decisions could be high (are amortized over all the times the function is used
- Static resource allocation can be done
- Choose fast(er) algorithm, take advantage of special cases
- Reduce queueing costs
- Special limited hardware can be allocated if available
- Choice of multiple transfer paths could also be performed

Basics

- Mirror 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
- All arguments “fixed” for subsequent uses
- Persistent collective operations cannot be matched with blocking or nonblocking collective calls

Init/Start

- The init function calls only perform initialization; do not start the operation
- E.g., MPI_Allreduce_init
  - Produces a persistent request (not destroyed by completion)
- Works with MPI_Start/MPI_Startall (cannot have multiple operations on the same communicator in Startall)
- Only inactive requests can be started
- MPI_Request_free can free inactive requests

Ordering of Inits and Starts

- Inits are nonblocking collective calls and must be ordered
- Persistent collective operations must be started in the same order at all processes
- Startall cannot contain multiple operations on the same communicator due to ordering ambiguity

Example

Nonblocking Collective APIs	Persistent Collective APIs

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

Argonne NATIONAL LABORATORY

Tools Working Group

ETH Zürich

Active Proposals (1/2)

- New interface to replace PMPI
  - Known, longstanding problems with the current profiling interface PMPI
    - One tool at a time can use it
    - Forces tools to be monolithic (a single shared library)
    - The interception model is OS dependent
  - New interface
    - Callback design
    - Multiple tools can potentially attach
    - Maintain all old functionality
- New feature for event notification in MPI_T
  - PERUSE
  - Tool registers for interesting event and gets callback when it happens

Active Proposals (2/2)

- Debugger support - MPIR interface
  - Fixing some bugs in the original “blessed” document
    - Missing line numbers!
  - Support non-traditional MPI implementations
    - Ranks are implemented as threads
  - Support for dynamic applications
    - Commercial applications/ Ensemble applications
    - Fault tolerance
  - Handle Introspection Interface
    - See inside MPI to get details about MPI Objects
      - Communicators, File Handles, etc.

Can I Join?

- Join the mailing list
  - http://lists.mpi-forum.org/
  - mpiwg-tools
- Join our meetings
  - https://github.com/mpiwg-tools/tools-issues/wiki/Meetings
- Look at the wiki for current topics
  - https://github.com/mpiwg-tools/tools-issues/wiki

Argonne NATIONAL LABORATORY

Large Count Working Group

ETH Zürich

Problem with Large Counts

- MPI_Send/Recv and other functions take “int” as the count for data
  - What happens for data larger than 2GB x datatype size?
  - You create a new large “contiguous” derived datatype and send that
  - Possible, but clumsy
- What about duplicating all MPI functions to change “int” to “MPI_Count” (which is a large, typically 64-bit, integer)
  - Doubles the number of MPI functions
  - Possible, but clumsy

New C11 Bindings

- Use C11 _Generic type to provide multiple function prototypes
  - Like C++ function overloading, but done with compile time macro replacement
- MPI_Send will have two function signatures
  - One for traditional “int” arguments
  - One for new “MPI_Count” arguments
- Fully backward compatible for existing applications
- New applications can promote their data lengths to 64-bit without changing functions everywhere

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

Argonne NATIONAL LABORATORY

Conclusions

ETH Zürich

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 (MPICH on InfiniBand) : http://mvapich.cse.ohio-state.edu/
  - Intel MPI (MPICH derivative): http://software.intel.com/en-us/intel-mpi-library/
  - Microsoft MPI (MPICH derivative)
  - Open MPI : http://www.open-mpi.org/
  - IBM MPI, Cray MPI, HP MPI, TH MPI, …
- Several MPI tutorials can be found on the web