Argonne NATIONAL LABORATORY

MPI for Scalable Computing

Tutorial at ATPESC, July 2026

Links to latest slides and code examples are posted in Slack

Hui Zhou and Rajeev Thakur

Argonne National Laboratory

U.S. DEPARTMENT OF ENERGY

About the Speakers

- **Hui Zhou:** Principal Research Software Engineer, Argonne National Laboratory
- **Rajeev Thakur:** Deputy Division Director and Argonne Distinguished Fellow, Argonne National Laboratory

- We have been involved in MPI standardization (in the MPI Forum) and in MPI implementation (MPICH) for a long time

The MPI Part of ATPESC

- We assume everyone already has some MPI experience
- We will focus on understanding MPI concepts and use hands-on code examples to illustrate them
- Emphasis will be on issues affecting scalability and performance
- There will be code walkthroughs and hands-on exercises

Outline

- (15 min) Introduction
- (20 min) Avoiding unnecessary synchronization
- (25 min) Stencil example (hands-on)
- (90 min) The MPI Programming Model
  1. Collective agreement
     - Collectives • RMA • Dynamic Processes • Sessions

3:00-3:30 break

  2. Progressive abstraction
     - Datatypes • Nonblocking Collectives • Persistent Operations
  3. Interoperability
     - Threads • GPUs
- (30 min) MPI Benchmarking (hands-on)

What is MPI?

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

MPI-1

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

Following MPI Standards

- MPI-2 was released in 1997
  - Several new features including MPI + threads, MPI-I/O, remote memory access functionality, C++ bindings, and many others
- MPI-2.1 (2008) and MPI-2.2 (2009) were released with some corrections to the standard and small features
- MPI-3 (2012) several new features
- MPI-3.1 (2015) minor corrections and features
- MPI-4 (June 2021) several new features
- MPI-4.1 (November 2023) minor corrections and features,
- MPI-5.0 (June 2025) standardization of application binary interface (ABI)
- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both PDF and HTML

Overview of MPI-3

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

Overview of MPI-4

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

Overview of MPI-5

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

Important considerations while using MPI

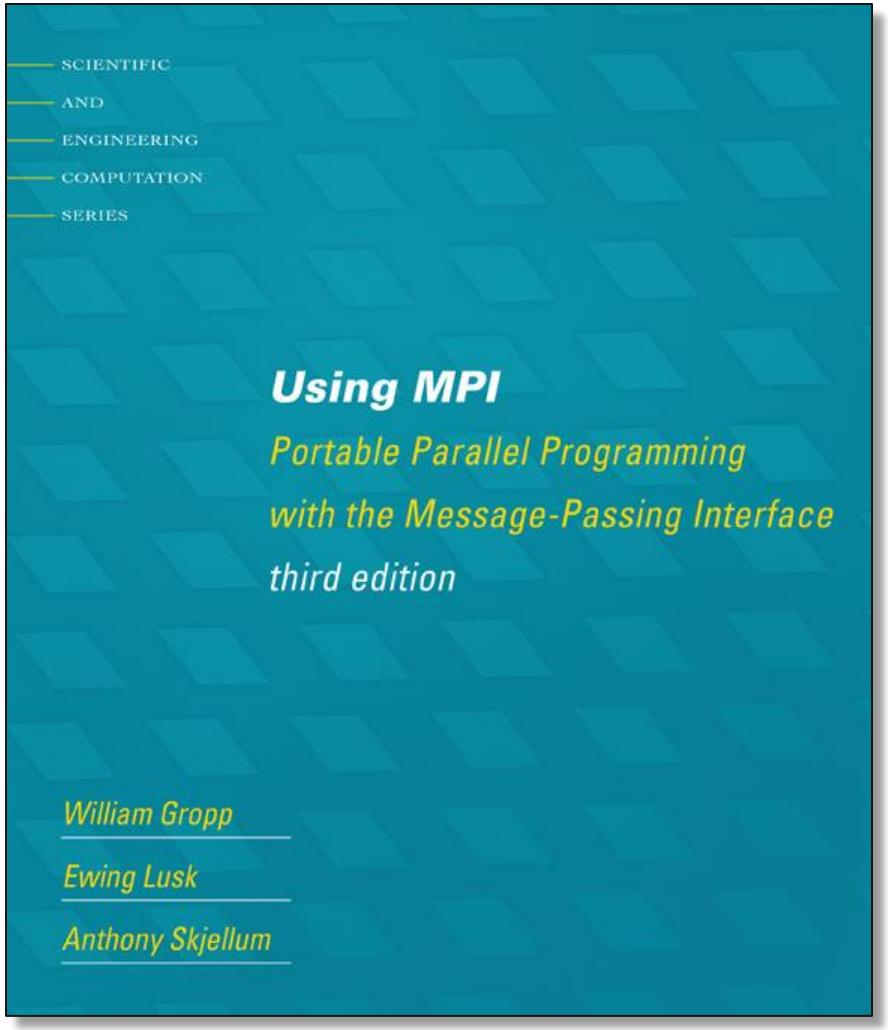
- All parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs

Web Pointers

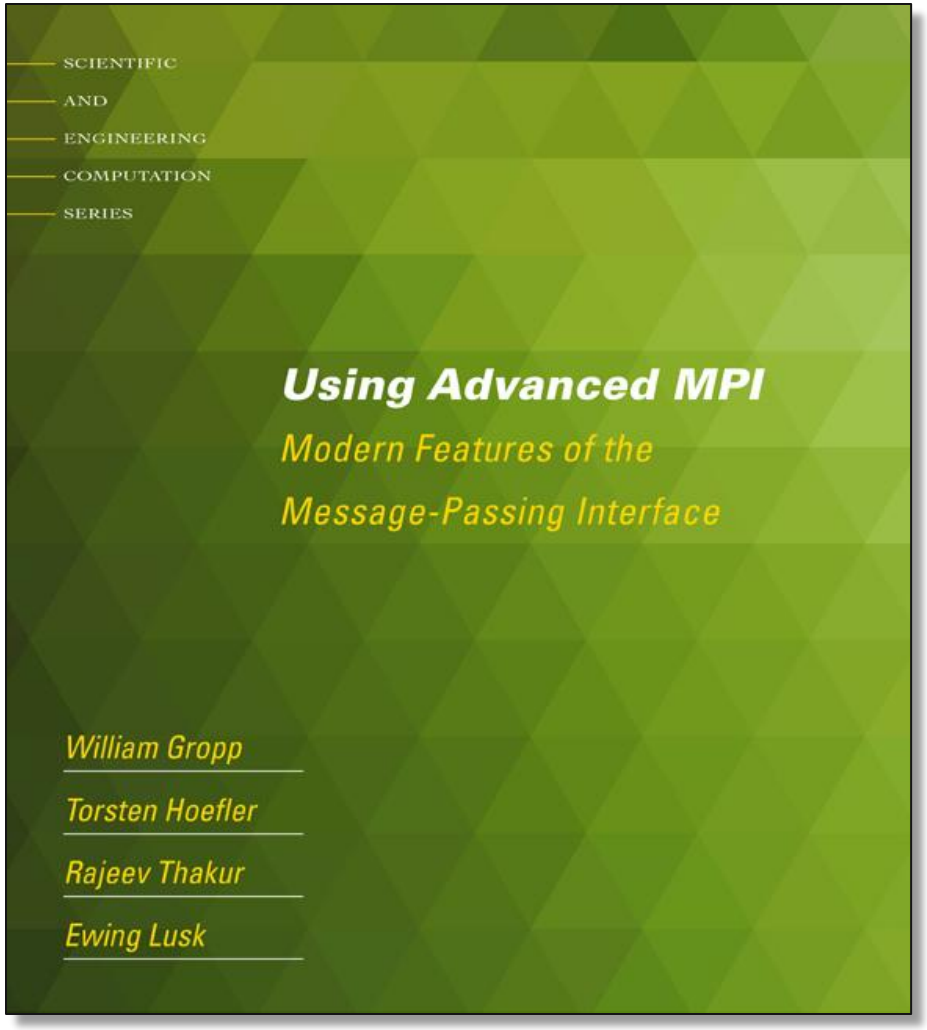
- MPI Standard : http://www.mpi-forum.org/docs/
- MPI Forum : http://www.mpi-forum.org/

- MPI implementations:
  - MPICH : http://www.mpich.org/
  - MVAPICH : http://mvapich.cse.ohio-state.edu/
  - Intel MPI: http://software.intel.com/en-us/intel-mpi-library/
  - Microsoft MPI: https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi
  - Open MPI : http://www.open-mpi.org/
  - IBM Spectrum MPI, Cray MPICH, ParaStation MPI, …
- General MPI Education
  - https://rookiehpc.org/mpi/
  - https://mpitutorial.com/tutorials/

Tutorial Books on MPI



Basic MPI



Advanced MPI, including MPI-3

MPI Fundamentals Refresher

Blocking vs Nonblocking Communication

Blocking vs. Nonblocking Communication

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

Blocking vs. Nonblocking Completion Semantics

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

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

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

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

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

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

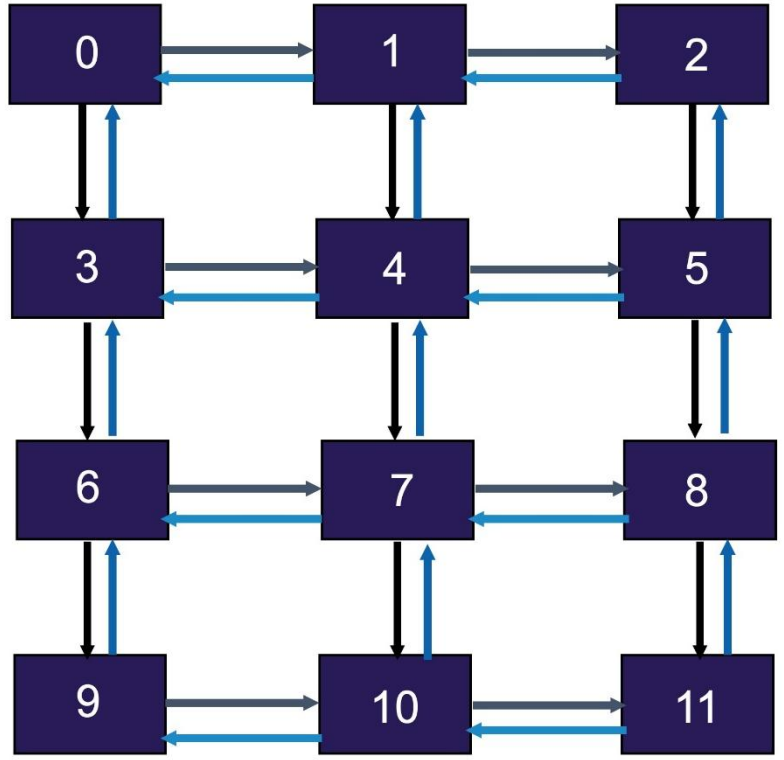
Nonblocking Communication Overlap

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

Costs of Unintended Synchronization

Unintended Synchronization

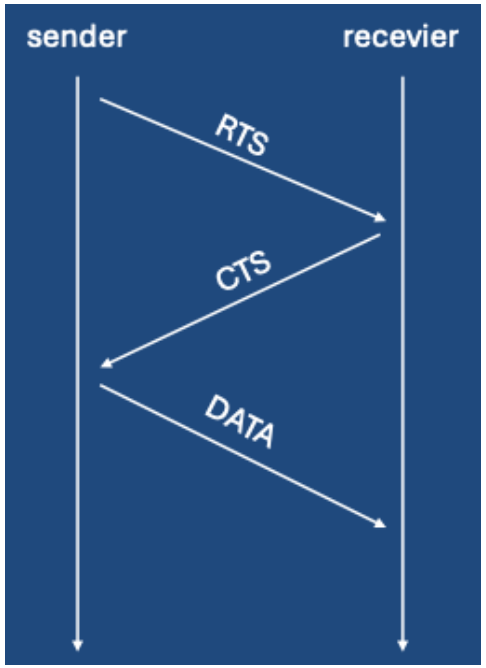
- MPI send and receive messaging combines data movement and synchronization
  - In many typical data exchange patterns, unnecessary synchronization can lead to poor performance
  - An example of unnecessary synchronization is sending and receiving messages one-at-a-time using blocking communication primitives.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



Neighbor Exchange Example Code

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

rendezvous



2D exchange (blocking)

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

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


- Unsafe use of blocking send. Risk of deadlock!

2D exchange (blocking + ordering)

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


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

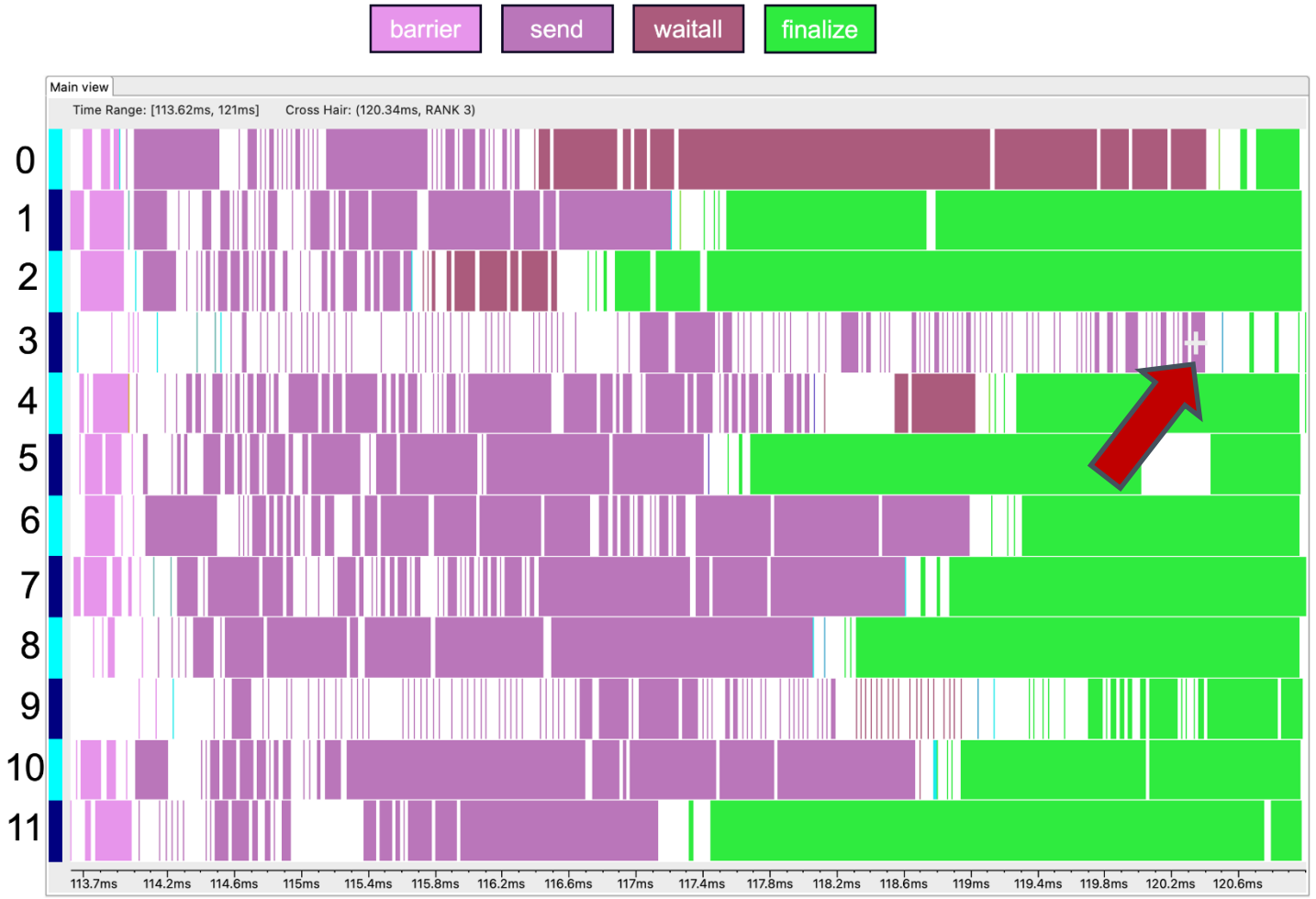
2D exchange (nonblocking receives)

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

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

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


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

Timeline of nonblocking receive exchange

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



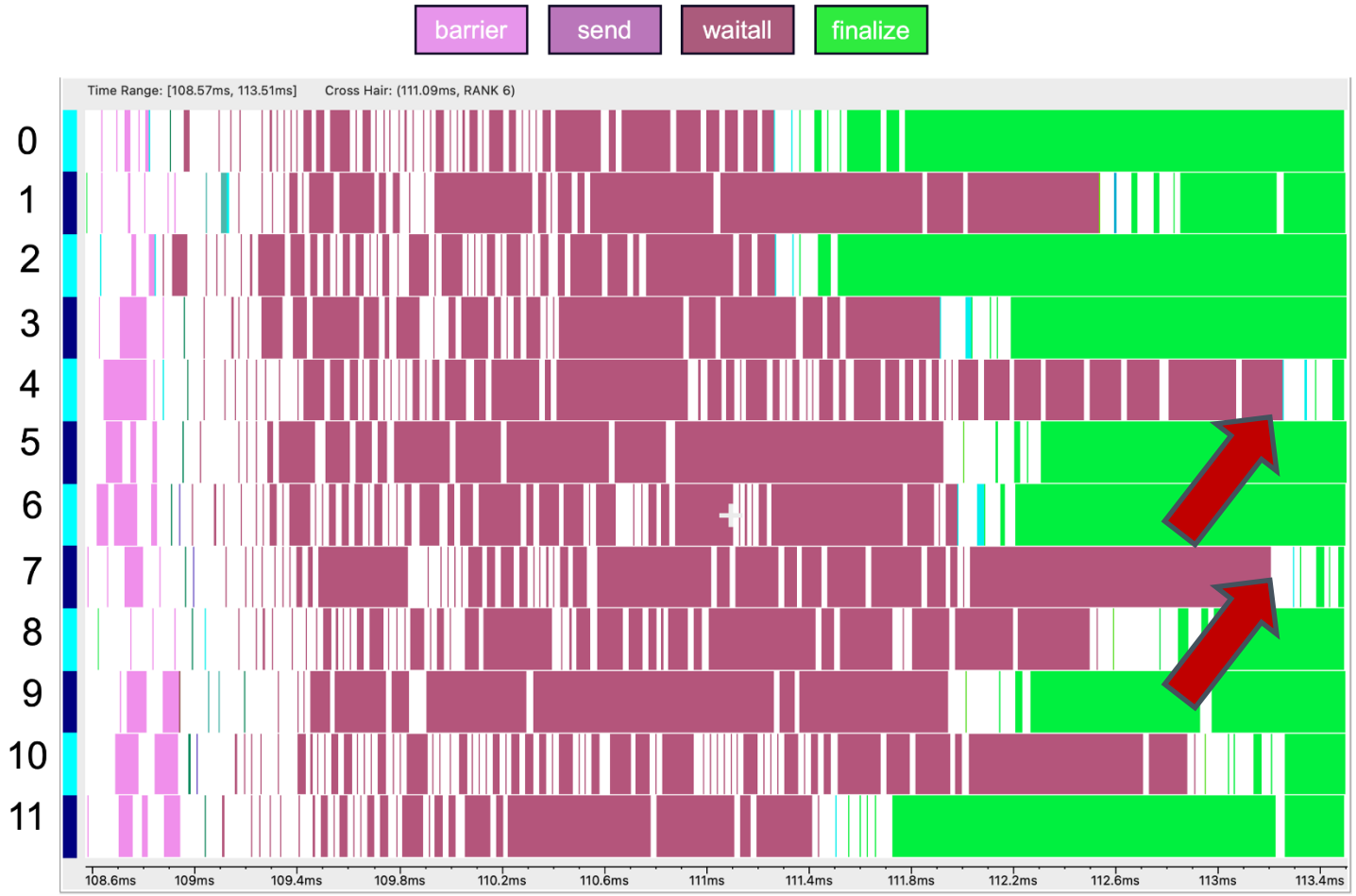
2d exchange (fully nonblocking)

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

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

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


- Best utilization. No arbitrary or unintended synchronization.

Timeline of fully nonblocking exchange

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



Takeaway: Defer synchronization!

- Send-receive accomplishes two things:
  - Data transfer
  - Synchronization
- In many cases, there is more synchronization than required
- Consider the use of nonblocking operations and MPI_Waitall to defer synchronization
- Gives MPI the best chance to utilize communication resources effectively and results in best performance
  - High amounts of memory and network bandwidth in modern hardware. Do not waste it!

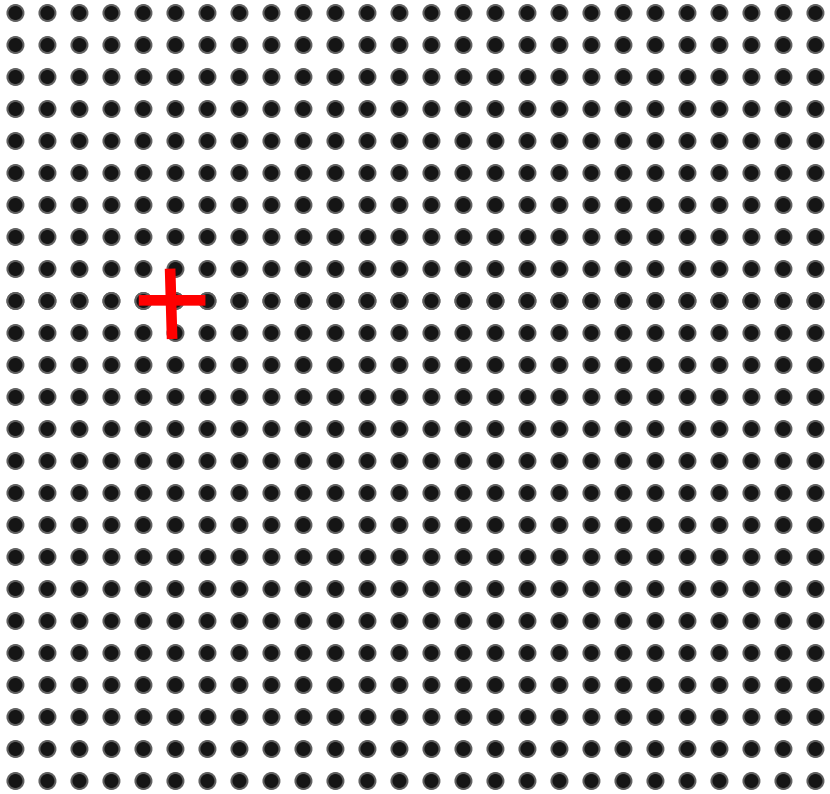
Running Example: Stencil

Running Example: Regular Mesh Algorithms

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

The Global Data Structure

- Each circle is a mesh point
- Difference equation evaluated at each point involves the four neighbors
- The red “plus” is called the method’s stencil
- Good numerical algorithms form a matrix equation Au=f; solving this requires computing Bv, where B is a matrix derived from A. These evaluations involve computations with the neighbors on the mesh.
- Example uses “fixed boundary conditions”. Heat is not lost through the boundary, but it is no longer measured by the system.



MPI Examples

- https://github.com/pmodels/mpi-tutorial-examples/
- Clone repo in your home or project directory
- Allocate or submit job using reservation on Aurora
  - qsub -q ATPESC -l select=1,walltime=1:00:00,filesystems=home -A ATPESC2026

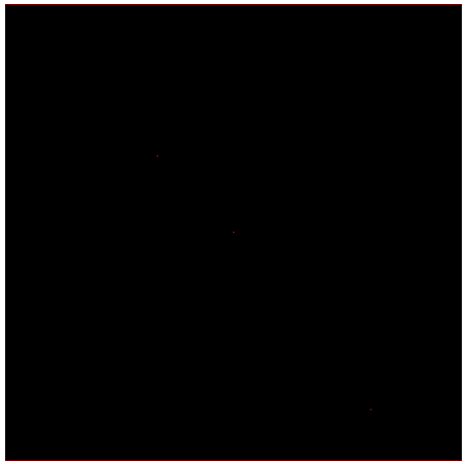
Example: Stencil

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

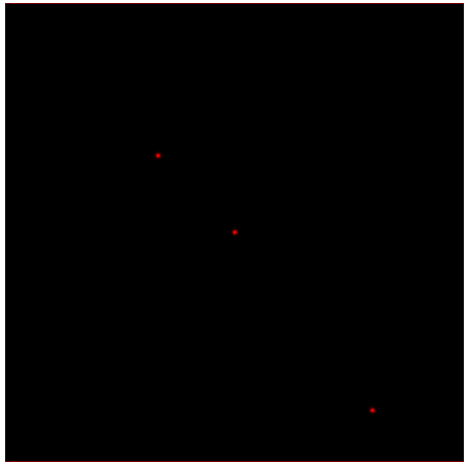
```
# mpicc –o stencil_serial stencil_serial.c
# qsub -q ATPESC –l select=1,walltime=1:00:00,filesystems=home –A
ATPESC2025 -I # Aurora
```

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

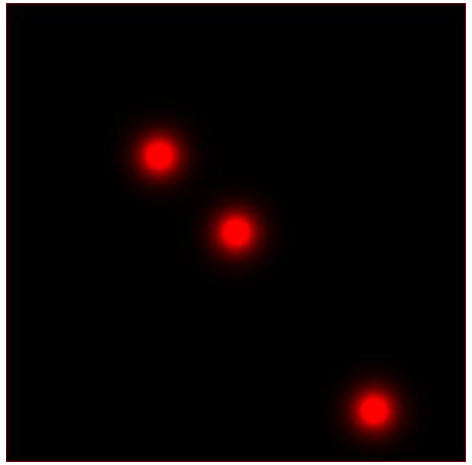
```
# ./stencil_serial 1000 10
  last heat: 30.000000
```



iter=10



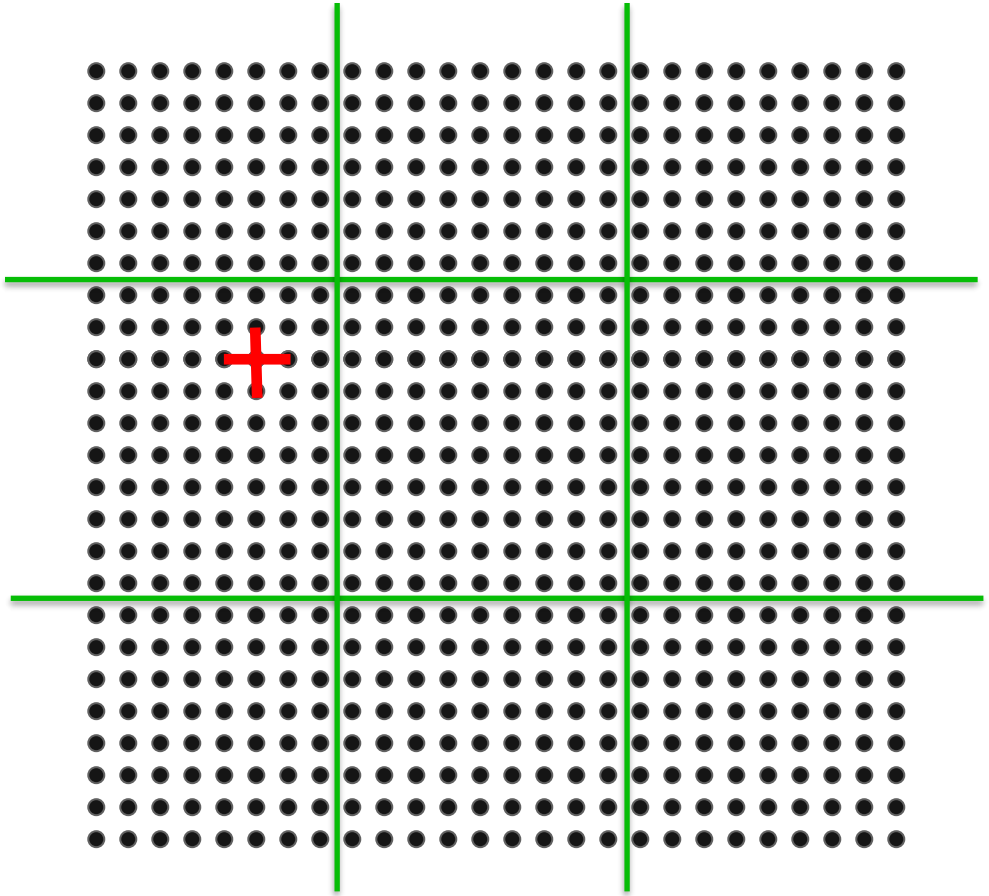
iter=100



iter=10000

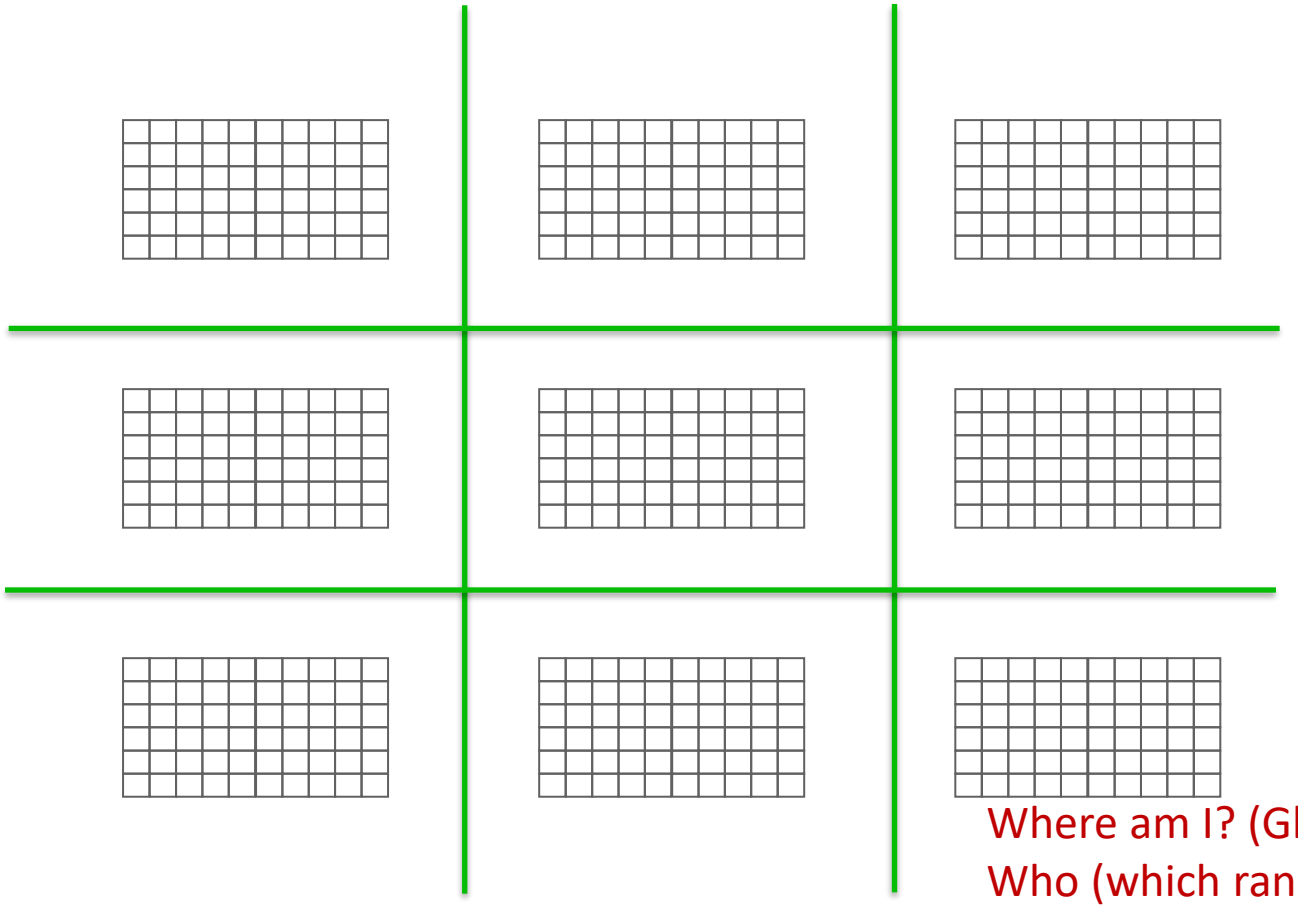
The Global Data Structure

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



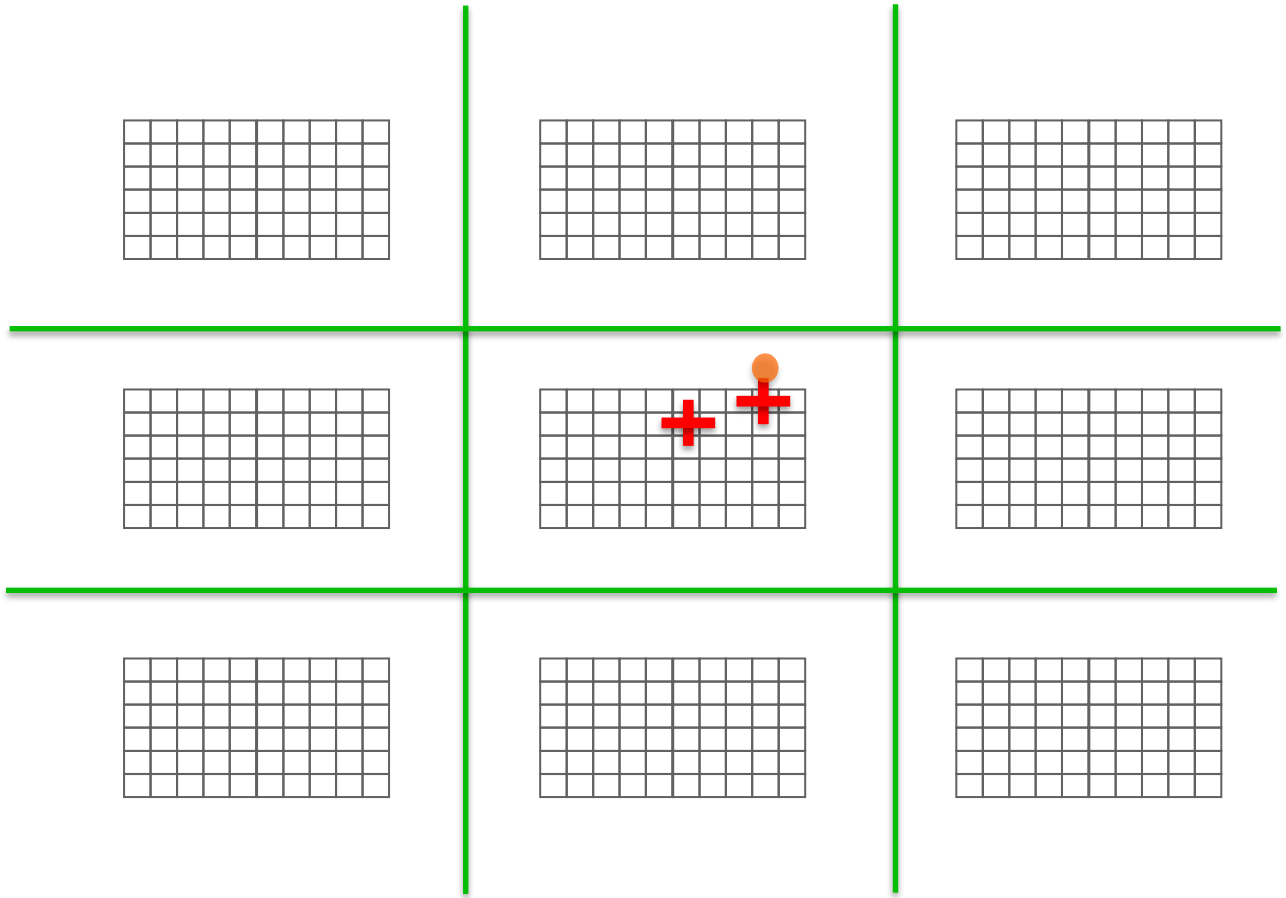
Domain Decompositioin

Parameters for domain decomposition:
N = Size of the edge of the global problem domain (assuming square)
PX, PY = Number of processes in X and Y dimension
N % PX == 0, N % PY == 0



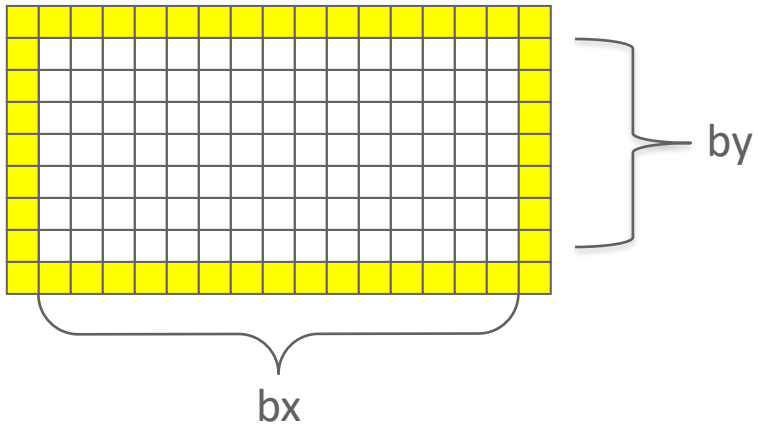
Where am I? (Global offset)
Who (which ranks) are my neigbhors?
Use MPI_PROC_NULL for exterior boundaries

Necessary Data Transfers



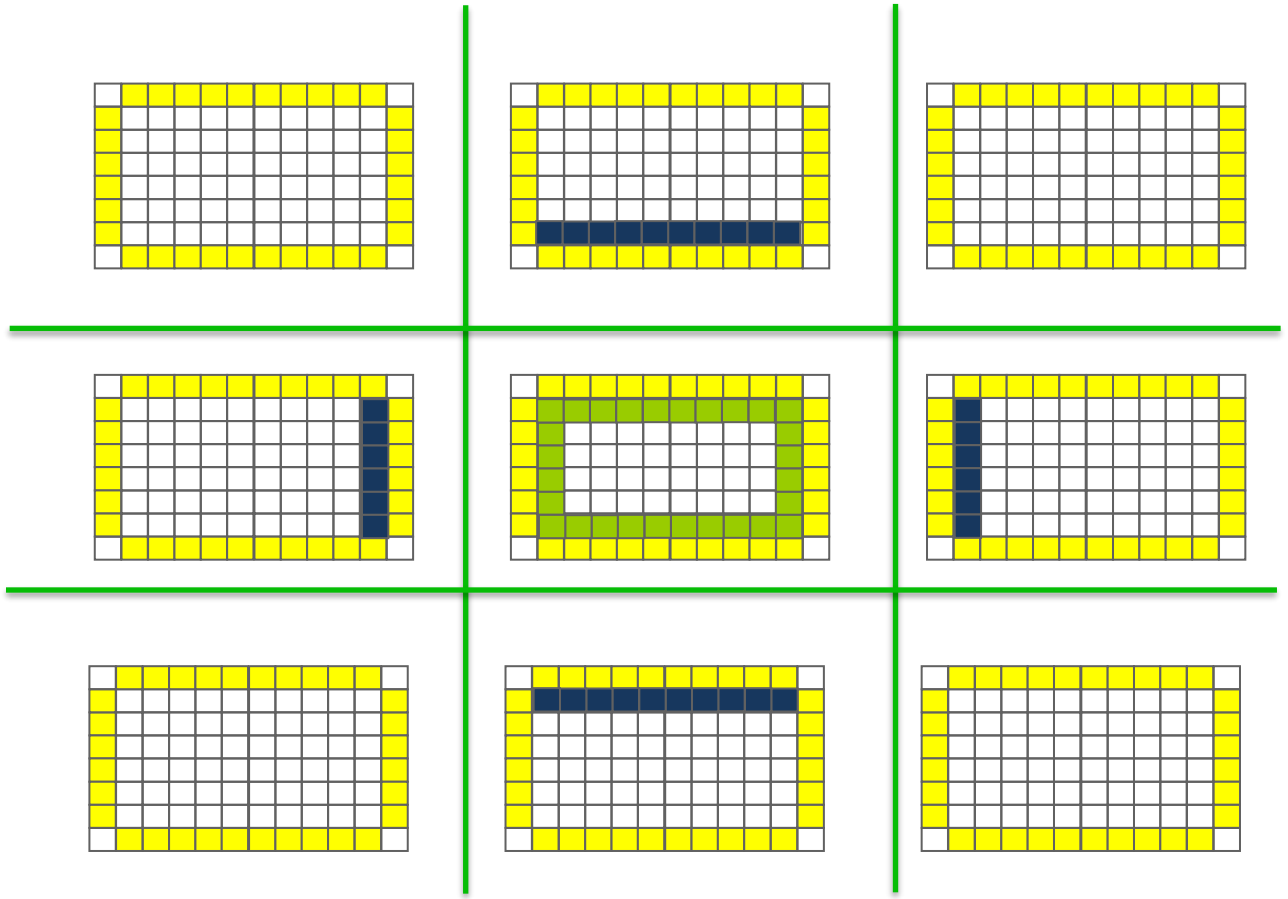
The Local Data Structure

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



Necessary Data Transfers

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



Example: Stencil with Nonblocking Send/recv

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

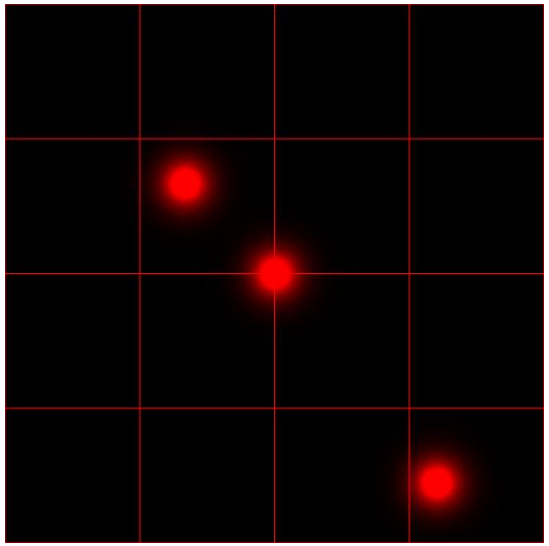
```
# qsub -q ATPESC -l select=1,walltime=1:00:00,filesystems=home –A ATPESC2025 -I # Aurora
```

For most systems (or local machine)

```
# mpiexec -n <nproc> ./stencil <domain size> <iterations> <px> <py>
```

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

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



iter=10000

The MPI Programming Model

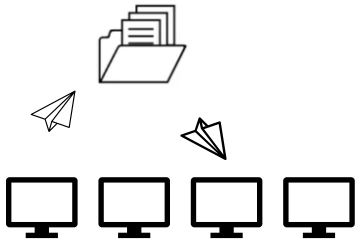
Core MPI - Concept #1

Assumed Collective Agreement

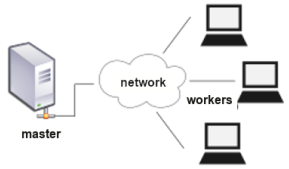
Parallel Programming Models

client / server

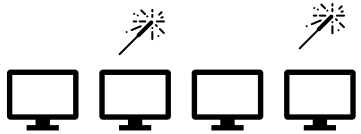
task dispatching



master / worker



event driven



Most parallel programming models are modeled after how human works in a society.

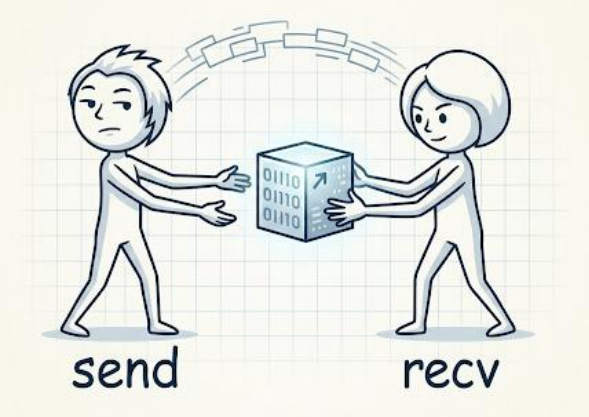
Commonly require:

- Dynamic discovery
- Protocol handshake
- Overhead of infrastructure management

41

The MPI Model

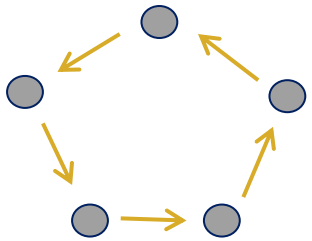
Point-to-point



Notably:

- Assumed coordination
- Lack of dynamic discovery and negotiation
- Lack of infra structure overhead

Collective



Concept #1 - Assumed Collective Agreement

- MPI programs are designed and written collectively
- Operations are collective with an assumed execution context
  - Assuming the processes
  - Assuming the task parameters
  - Assuming the synchronizations
- These assumptions are built into MPI specification
  - Send / Receives are matched and paired statically in code
  - Collectives are written as a single operation
  - Synchronizations are explicit
- MPI programs are distinctly
  - Simple: no extra negotiations or scheduling
  - High performance: no overhead

MPI Is A Hive-Mind Society Model

Point-to-Point Semantics Is The Physics Layer

SEMANTICS:

- Matching by (rank, tag, comm)
- Ordered within the process
- Concurrent between processes
- Provides explicit one-way synchronization

NOTES:

- Point-to-point messaging is dynamic in nature.
- It can be used to implement most dynamic designs.
- But it's key role in MPI is to support collective.

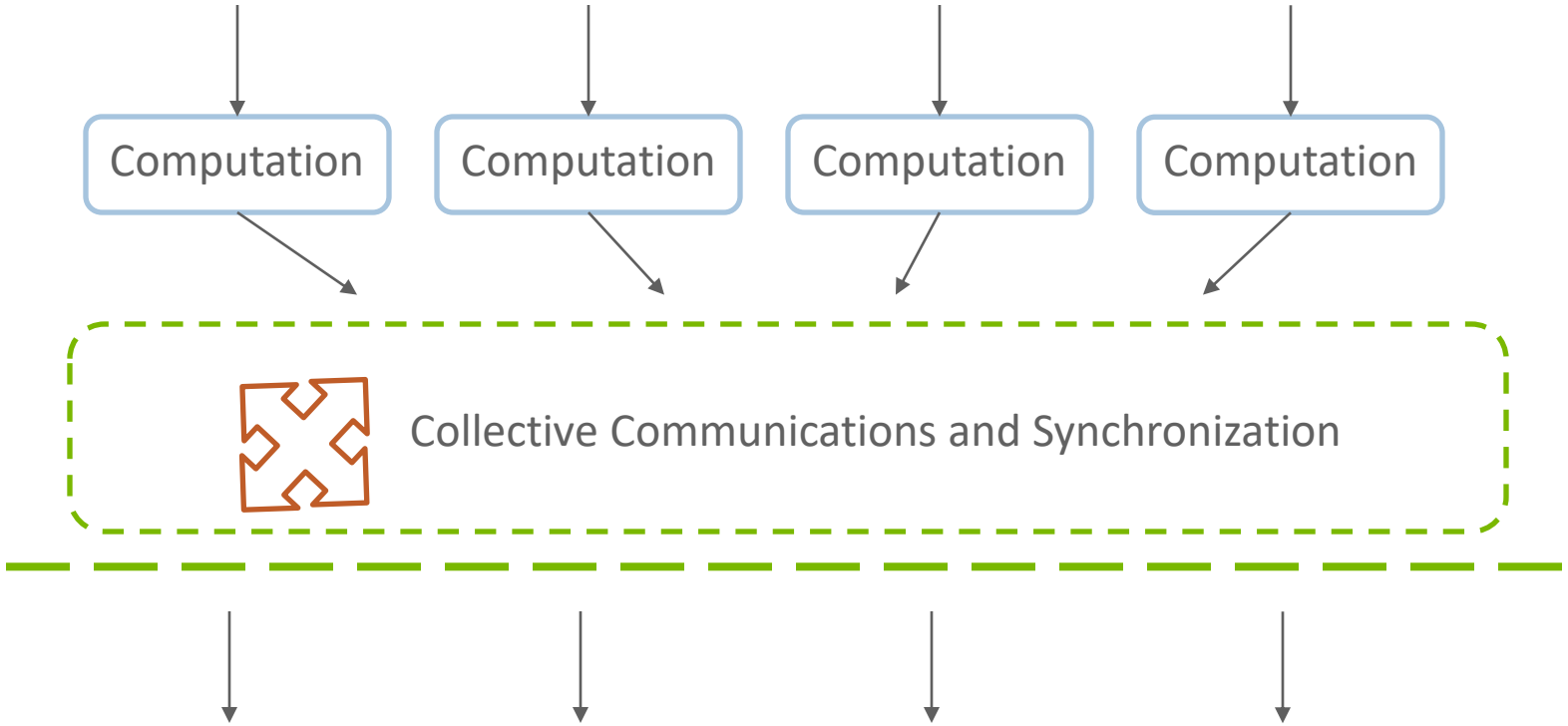
```c
if (rank == 0) {
    int value = 42;

    printf("Rank %d: Sending value %d to rank 1\n", rank, value);

    MPI_Send(&value,          /* buffer */
             1,               /* number of elements */
             MPI_INT,         /* datatype */
             1,               /* destination rank */
             tag,             /* message tag */
             MPI_COMM_WORLD); /* communicator */
} else if (rank == 1) {
    int value;
    MPI_Status status;

    MPI_Recv(&value,          /* receive buffer */
             1,               /* number of elements */
             MPI_INT,         /* datatype */
             0,               /* source rank */
             tag,             /* expected tag */
             MPI_COMM_WORLD,  /* communicator */
             &status);        /* receive status */

    printf("Rank %d: Received value %d from rank %d\n",
           rank, value, status.MPI_SOURCE);
}
```

Collectives Are Natural MPI Expressions

- Collective operations are called by all processes in a communicator.
- **`MPI_BCAST`** distributes data from one process (the root) to all others in a communicator.
- **`MPI_REDUCE`** combines data from all processes in the communicator and returns it to one process.
- Collective expression passes parallel contexts down the software stack, providing **optimization opportunity** in addition to **programming simplicity**

```c
/* Rank 0 initializes a value */
int value;
if (rank == 0) {
    value = 100;
    printf("Rank %d: Broadcasting value %d\n", rank, value);
}

/* Broadcast the value from rank 0 to all processes */
MPI_Bcast(&value,           /* buffer */
          1,                /* number of elements */
          MPI_INT,          /* datatype */
          0,                /* root process */
          MPI_COMM_WORLD);  /* communicator */

printf("Rank %d: Received broadcast value %d\n", rank, value);

/* Each process contributes its rank */
int local = rank;
int sum = 0;

/* Sum all ranks on the root process */
MPI_Reduce(&local,          /* send buffer */
           &sum,            /* receive buffer (root only) */
           1,               /* number of elements */
           MPI_INT,         /* datatype */
           MPI_SUM,         /* reduction operation */
           0,               /* root process */
           MPI_COMM_WORLD); /* communicator */

if (rank == 0) {
    printf("Sum of all ranks = %d\n", sum);
}
```

Block Synchronous Parallel (BSP) Are Natural MPI Patterns

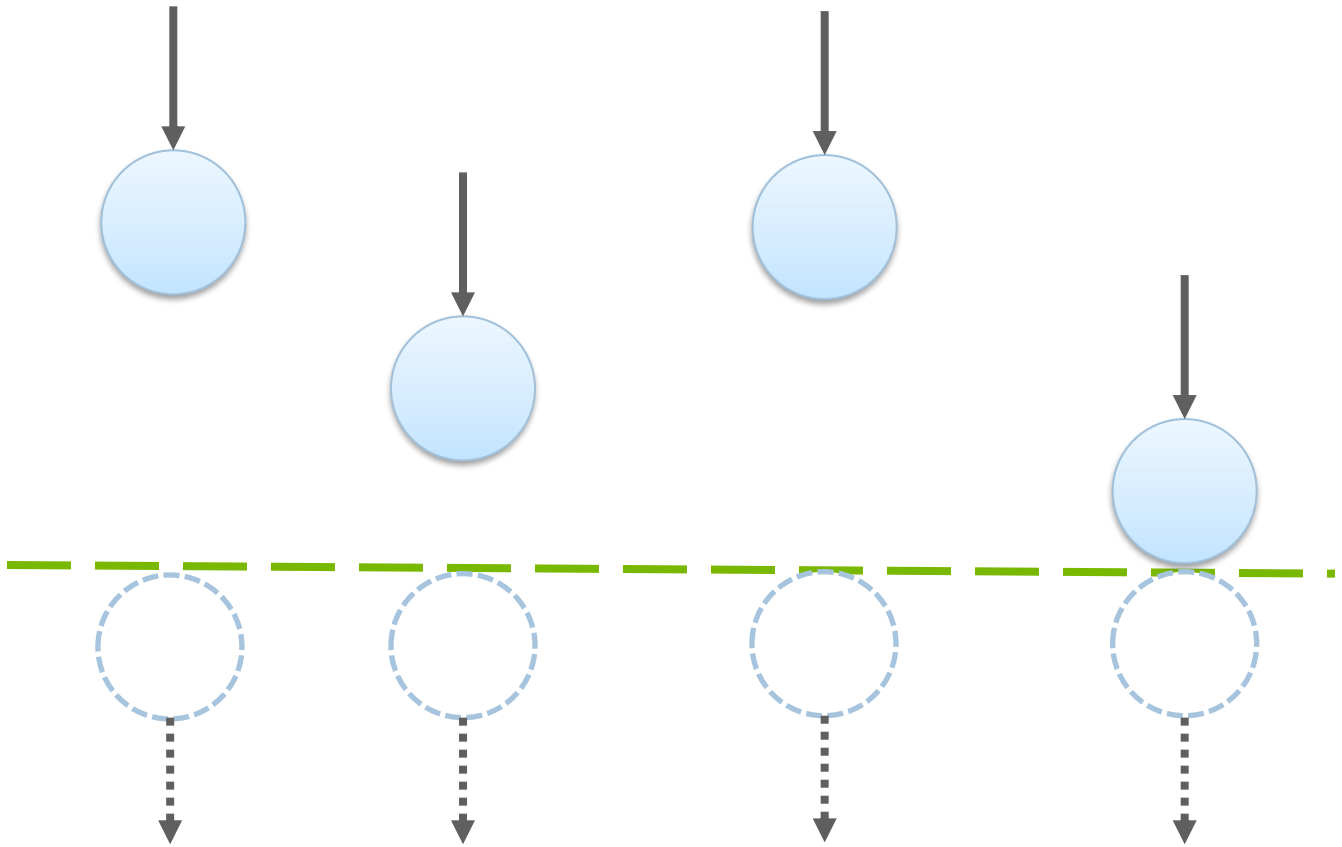


- Phases of individual actions and collective operations.
- Synchronized super steps
- A step-up from “embarrassingly parallel” by adding synchronization.
- MPI specializes in efficient synchronization.

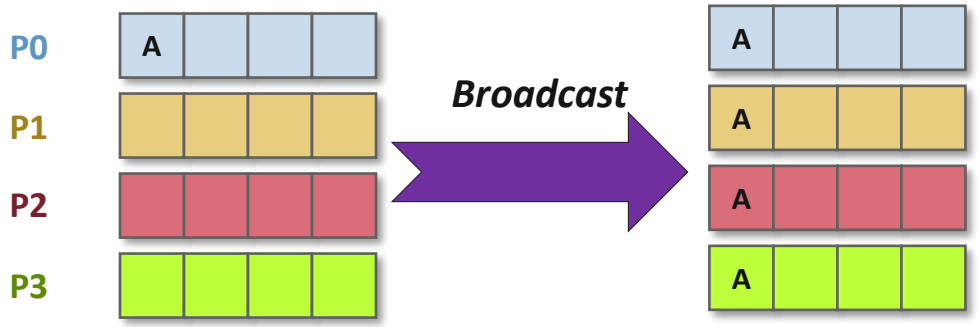
MPI Collective Communication

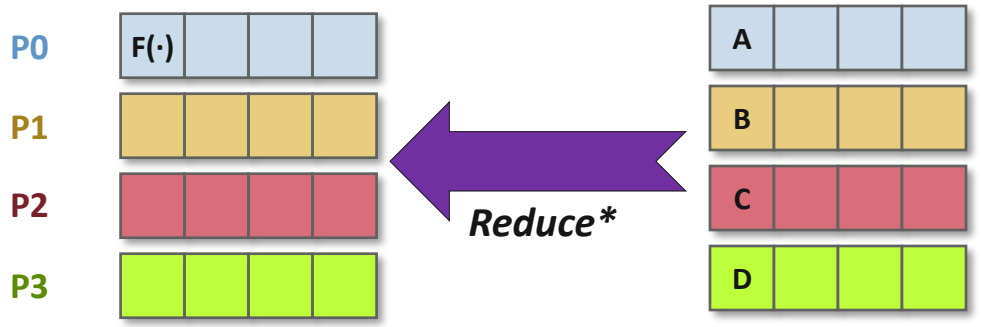
- Communication and computation is coordinated among a group of processes in a communicator
- Tags are not used; different communicators deliver similar functionality
- Nonblocking collective operations in MPI-3
- Three classes of operations: synchronization, data movement, collective computation

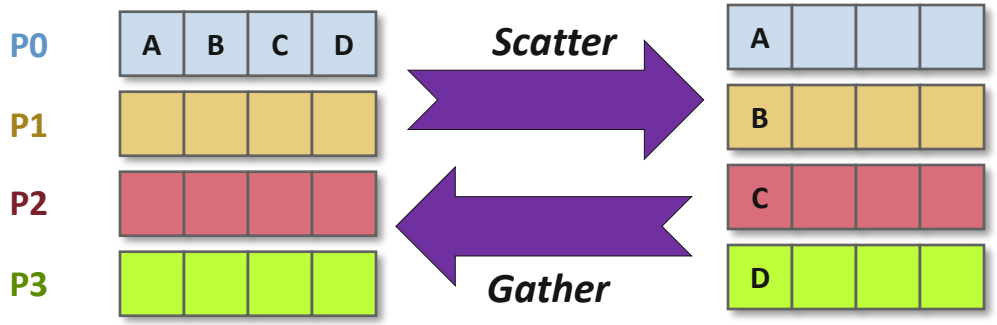
Collective Synchronization - MPI_Barrier



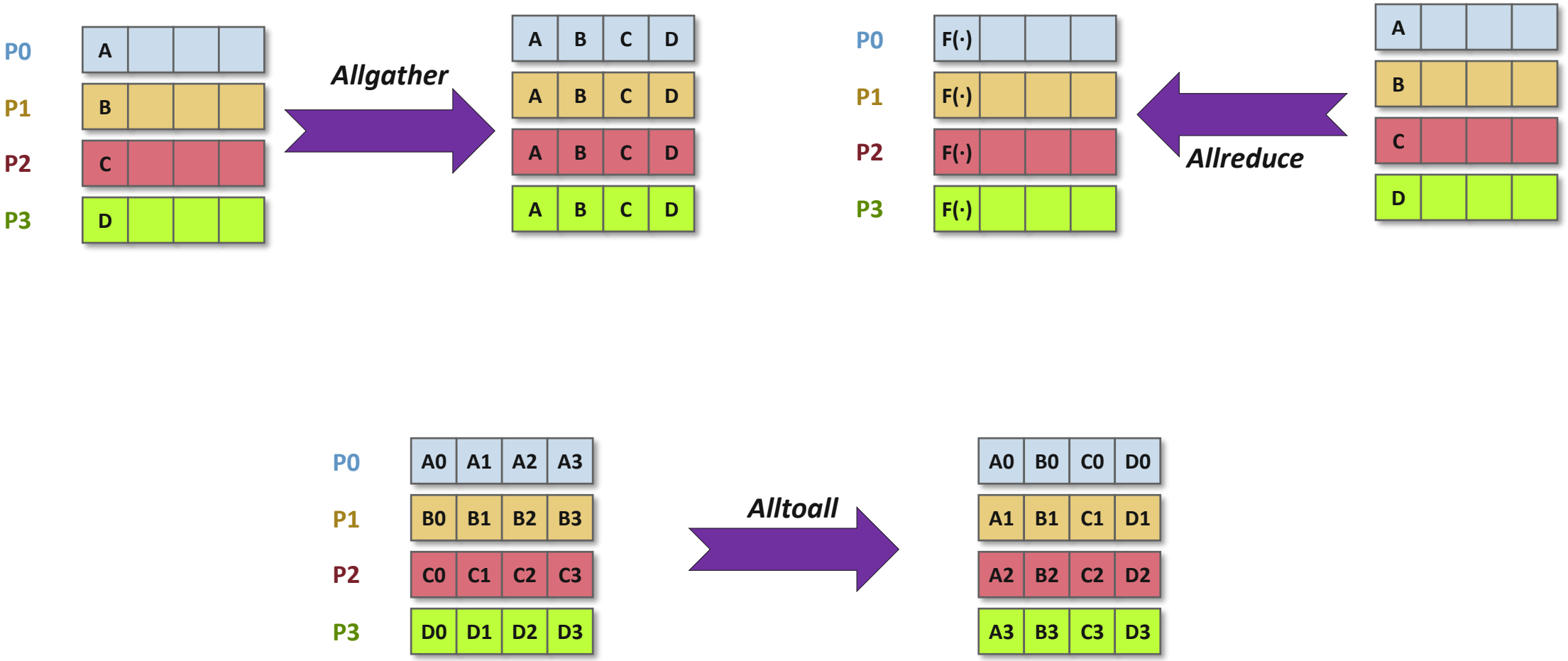
Collective Data Movement







More Collective Data Movement



Collective Algorithms

```
/* Naïve Bcast */

if (rank == root) {
    for (dest = 1 to P-1) {
        Send(buffer, dest);
    }
} else {
    Recv(buffer, root)
}
```

O(P) efficiency

```
/* Binomial Bcast */

mask = 1;
while (mask < P) {
    if (rank < mask) {
        dest = rank + mask
        if (dest < P) {
            Send(buffer, dest)'
        } else if (rank < 2*mask) {
            src = rank - mask;
            Recv(buffer, src);
        }
    }
    mask = mask << 1;
}
```

O(lg(P)) efficiency

Beyond Native Collectives

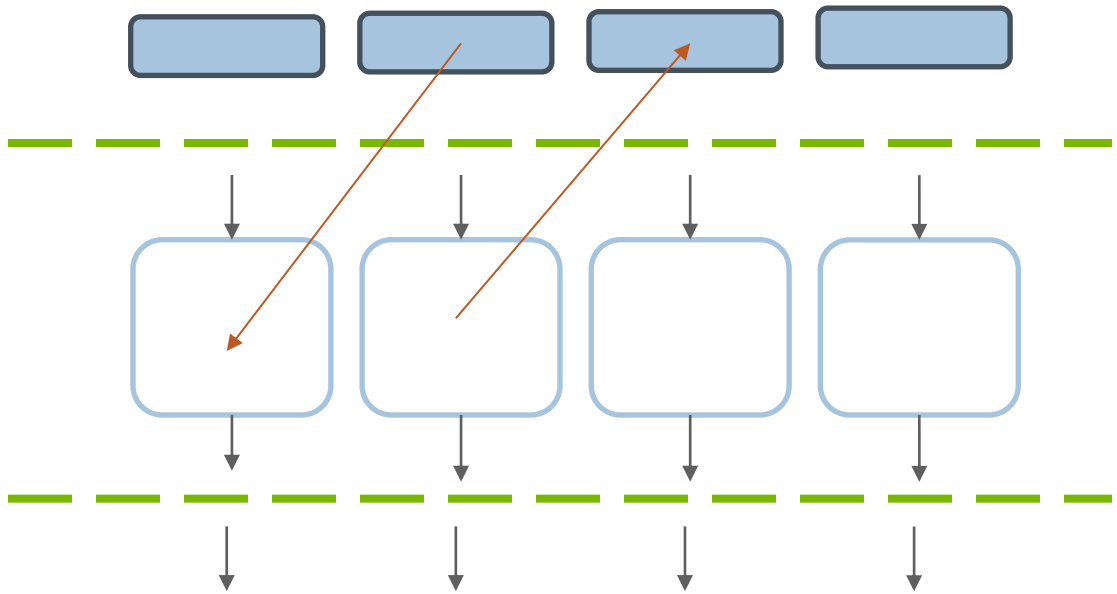
- Refactor your code to provide high-level abstraction of collective operations.
- Separately implement and optimize your collective operations.
- Advantages
  - Simplicity at high level
  - Orthogonal optimizations
  - Potential for code-reuse

Exercise:

- write the stencil example using MPI_Alltoallv

Perturbation to The Collective Agreement Principle

- The *Collective Agreement* requires synchronization. What if my pattern is unbalanced?
  - One-sided data movement
    - Get without another process to send
    - Put without another process to receive
  - Solution:
    1. Expose memory
    2. Provide synchronized epoch
    3. Allow one-sided put/get



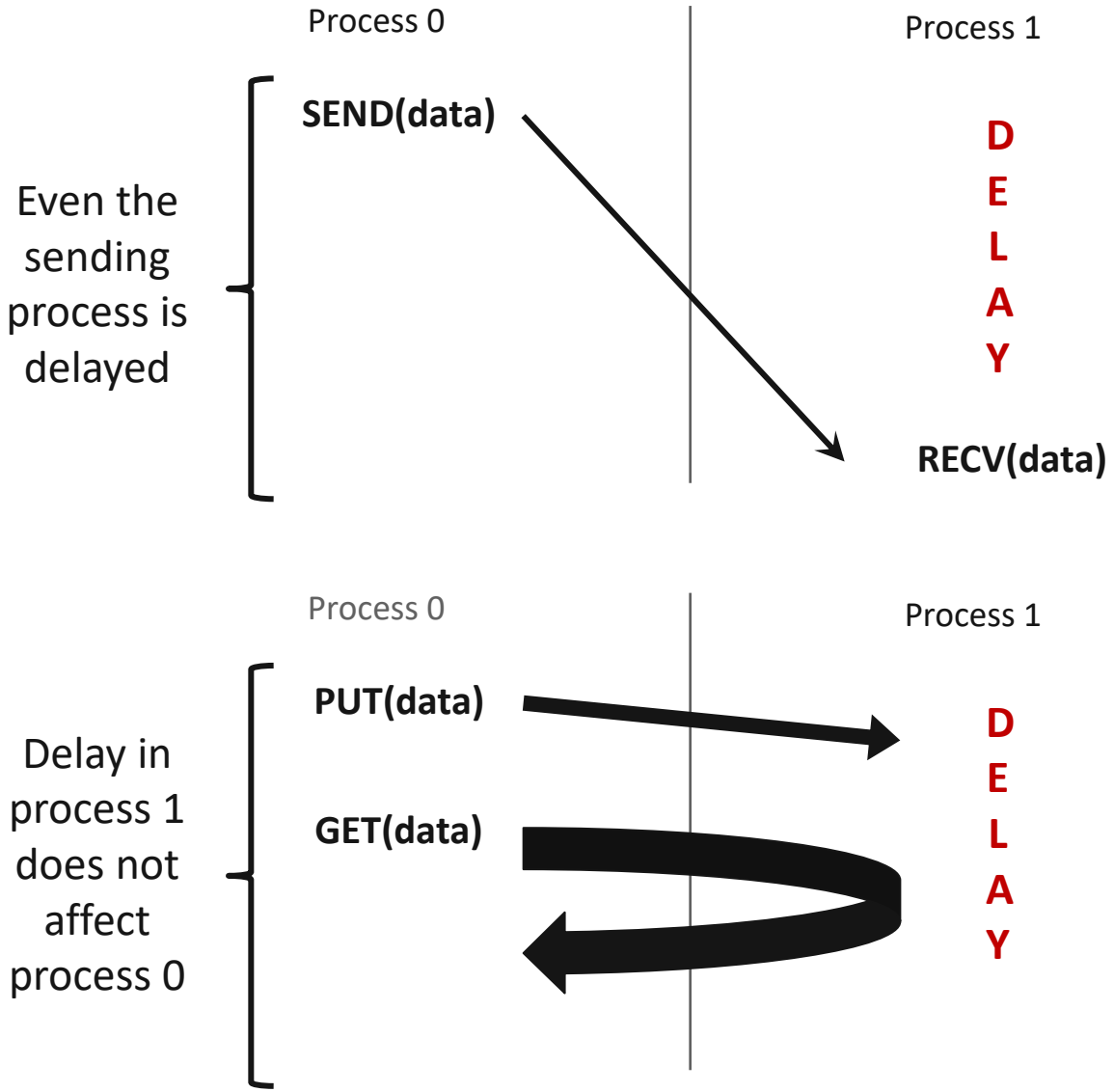
A perturbation to the BSP pattern.

MPI One-sided Communication (RMA)

- Maintain the BSP pattern via synchronized epochs
- Add asynchronized one-sided data movement
- Advantages:
  - Remove the matching overhead
  - Remove the synchronization overhead
  - Improve computation/communication overlap
- Disadvantages:
  - Lack of synchronizations
  - Mix synchronizations are costly
  - Implementation performance are uneven

```
int data[N];
MPI_Win win;

/* Initialize the array */
for (int i = 0; i < N; i++)
    data[i] = 0;

/* Expose the array as an RMA window */
MPI_Win_create(data, N * sizeof(int), sizeof(int),
               MPI_INFO_NULL, MPI_COMM_WORLD, &win);

/* Begin RMA epoch */
MPI_Win_fence(0, win);

if (rank == 0) {
    int buffer[N];

    /* Fill the local buffer */
    for (int i = 0; i < N; i++)
        buffer[i] = i + 1;

    /* Copy the entire array into rank 1's memory */
    MPI_Put(buffer, N, MPI_INT,
            1,          /* target rank */
            0,          /* target displacement */
            N, MPI_INT,
            win);
}

/* Complete RMA epoch */
MPI_Win_fence(0, win);

/* Print local array */
printf("Rank %d: ", rank);
for (int i = 0; i < N; i++)
    printf("%d ", data[i]);
printf("\n");
```

Comparing One-sided and Two-sided Programming



- MPI_Put and MPI_Get are nonblocking
- One-sided pattern is cleaner for BSP patterns that require mixed data movement and computations.
- One-sided pattern can unlock hardware efficiency when supported.
- However, performance portability is more challenging.

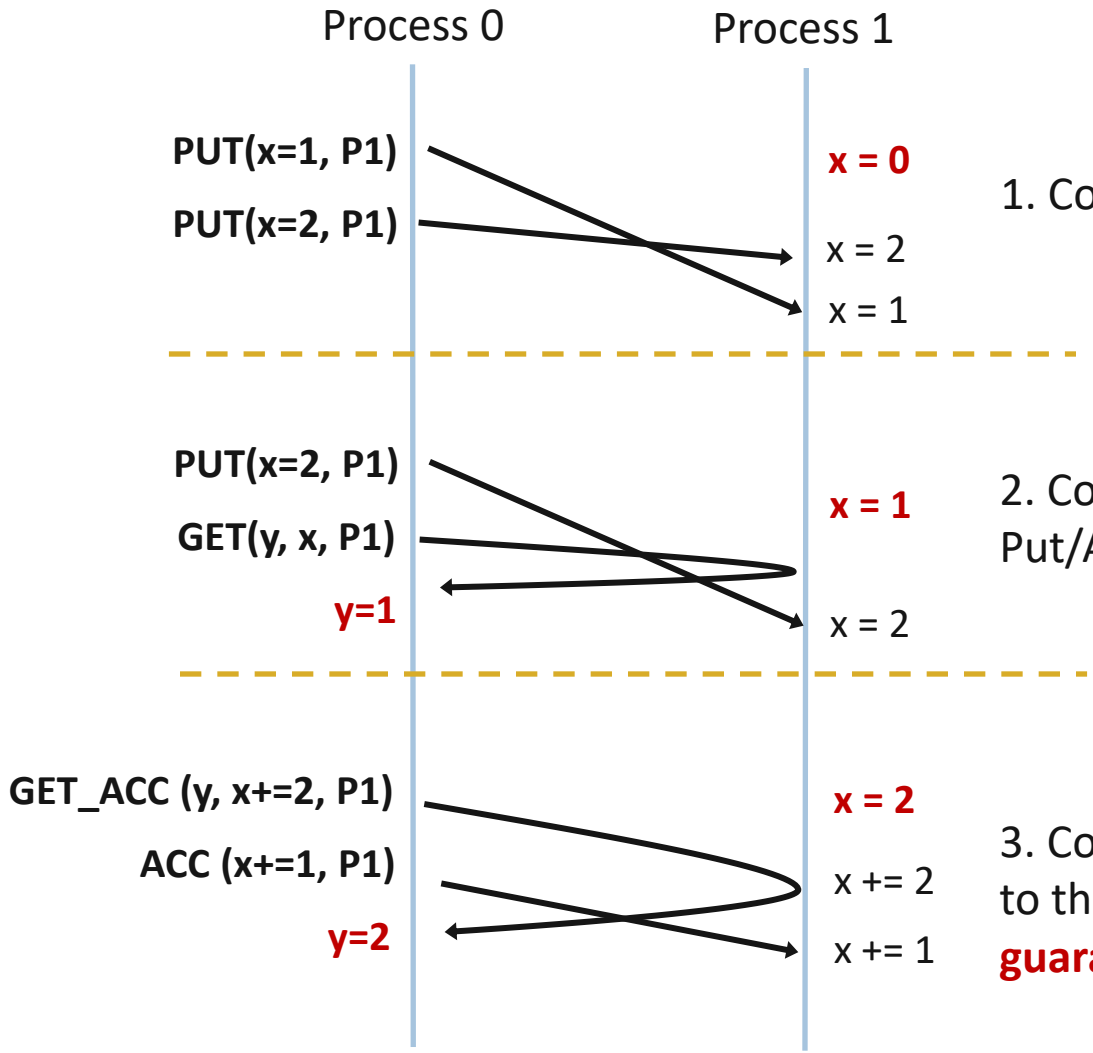
Window creation models

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

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**

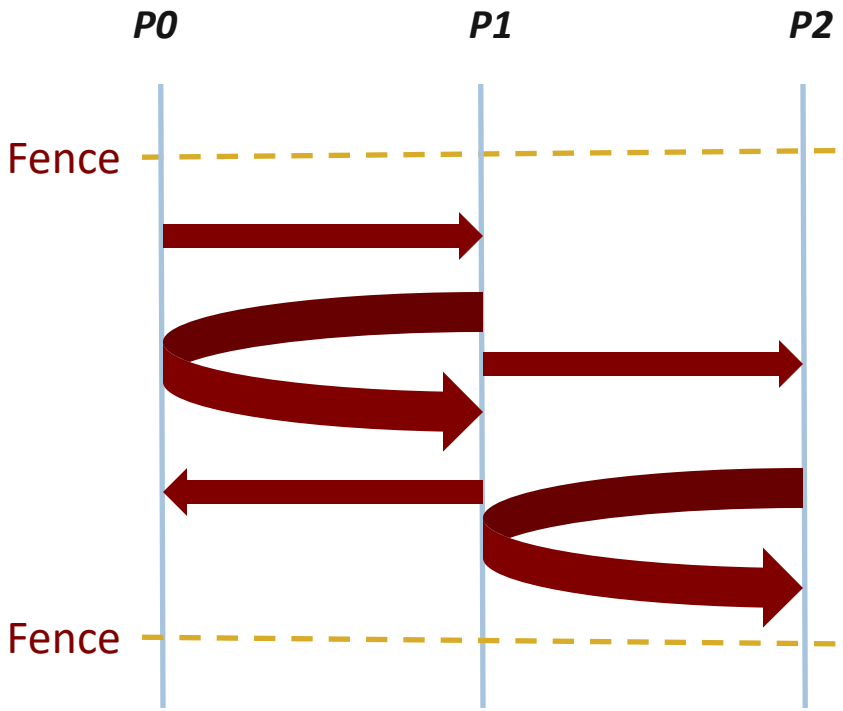
2. Concurrent Get and Put/Accumulates: **undefined**

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

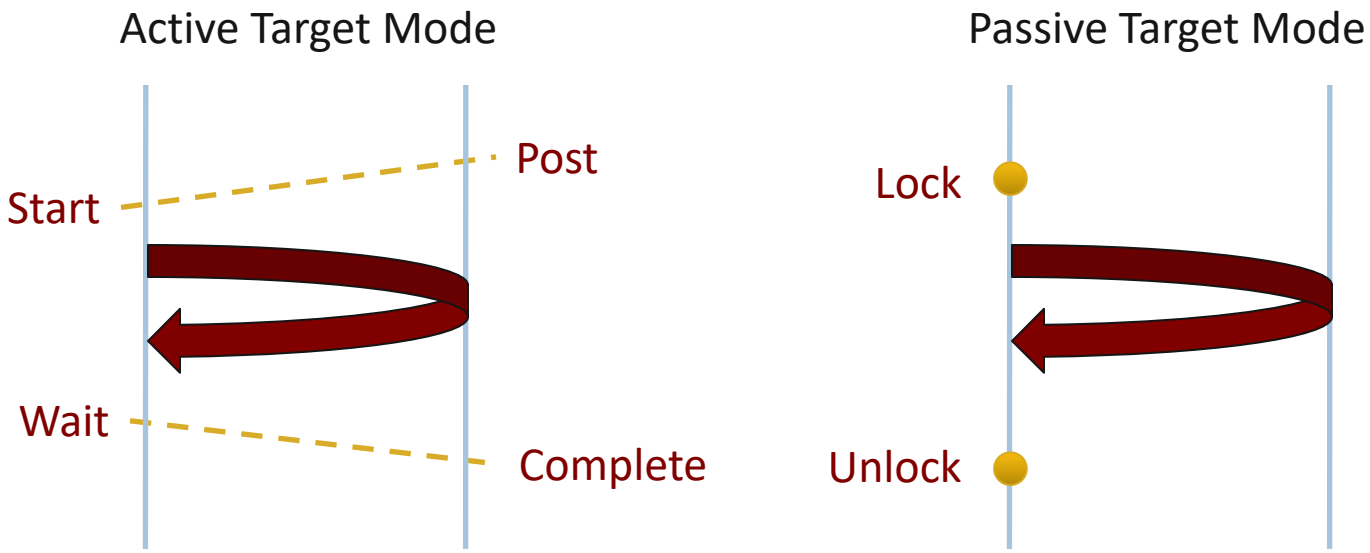
Fence: Active Target Synchronization

```
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

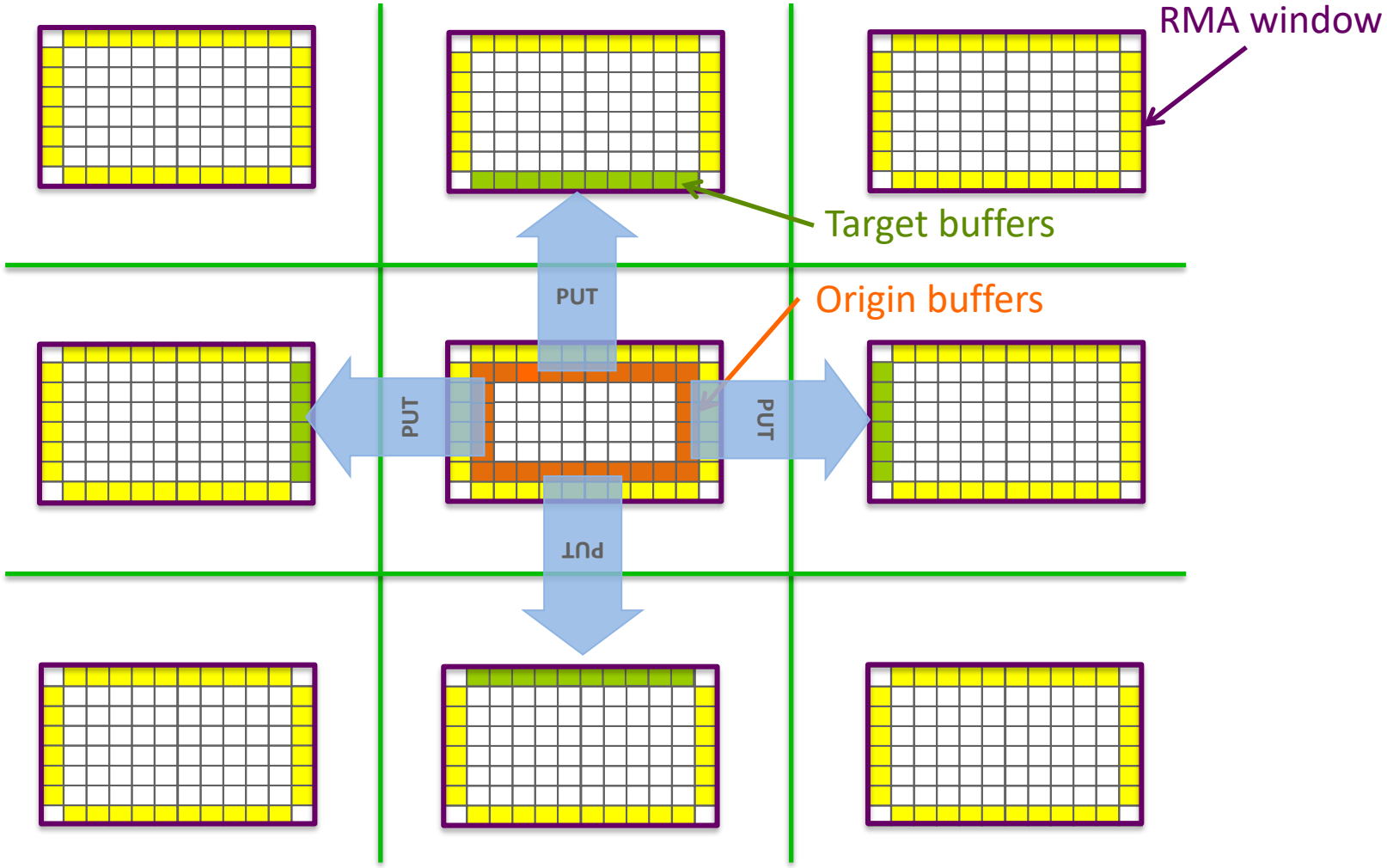


Lock/Unlock: Passive Target Synchronization



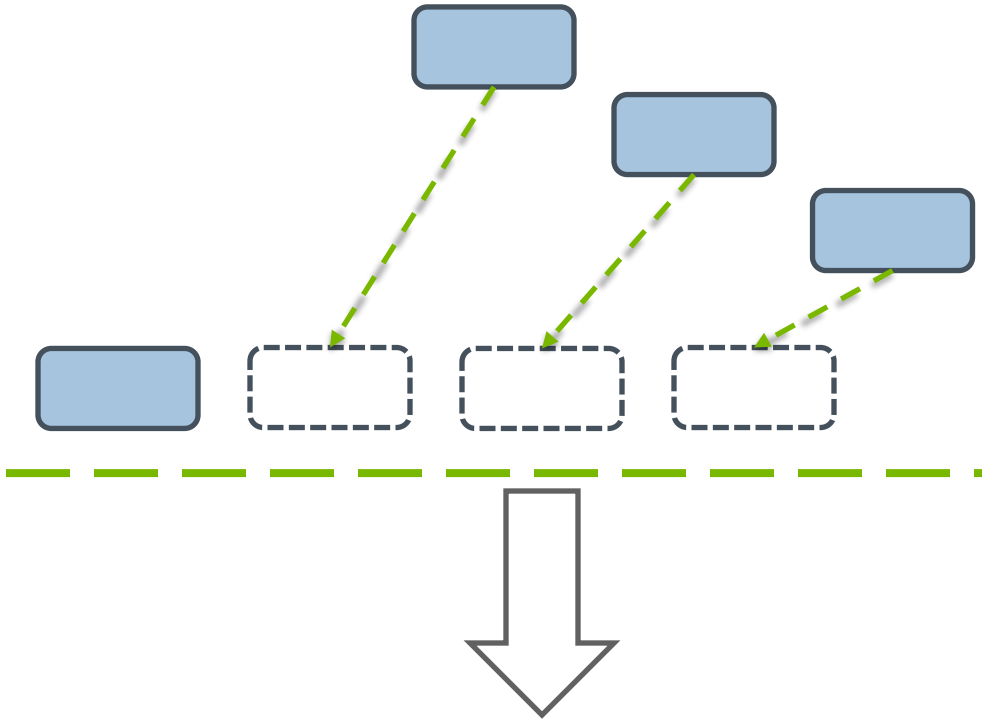
- Passive mode: One-sided, *asynchronous* communication
  - Target does **not** participate in communication operation
- Shared memory-like model

Exercise: Stencil with RMA Fence



Perturbation to The Collective Agreement Principle

- The *Collective Agreement* is rather strict. What if my pattern is -
  - One-sided data movement
  - Dynamically arranged environment
    - Solution:
      - MPI Dynamic Processes
      - MPI Sessions



MPI Dynamic Processes

parent.c

```c
MPI_Comm intercomm, comm;
int value = 42;

/* Spawn 4 child processes */
MPI_Comm_spawn("./child",
               MPI_ARGV_NULL,
               4,
               MPI_INFO_NULL,
               0,
               MPI_COMM_SELF,
               &intercomm,
               MPI_ERRCODES_IGNORE);

/* Merge parent and child groups into one communicator */
MPI_Intercomm_merge(intercomm, 0, &comm);

/* Intercommunicator no longer needed */
MPI_Comm_disconnect(&intercomm);

/* Parent becomes rank 0 in the merged communicator */
MPI_Bcast(&value, 1, MPI_INT, 0, comm);
```

child.c

```c
MPI_Comm intercomm, comm;
int rank, value = 0;

MPI_Comm_get_parent(&intercomm);

/* Merge with the parent */
MPI_Intercomm_merge(intercomm, 1, &comm);

/* Intercommunicator no longer needed */
MPI_Comm_disconnect(&intercomm);

MPI_Comm_rank(comm, &rank);

/* Standard intracommunicator broadcast */
MPI_Bcast(&value, 1, MPI_INT, 0, comm);

printf("Merged rank %d received %d\n", rank, value);
```

Singleton Init: run ./parent without using mpiexec

Note: may not work depending on process manager support.

MPI Sessions

```c
/* Initialize an MPI Session */
MPI_Session_init(MPI_INFO_NULL, MPI_ERRORS_RETURN, &session);

/* Get the default process set */
MPI_Group_from_session_pset(session, "mpi://WORLD", &group);

/* Create a communicator from the group */
MPI_Comm_create_from_group(group, "my_comm", MPI_INFO_NULL,
                           MPI_ERRORS_RETURN, &comm);

MPI_Group_free(&group);

/* Use the communicator */
MPI_Comm_rank(comm, &rank);
MPI_Comm_size(comm, &size);

printf("Hello from rank %d of %d\n", rank, size);

MPI_Barrier(comm);
```

- Non-interfering sessions
  - Nice if you are writing a library
- Non-collectively decide participants
- MPI-Business as usual

Exercise:

- Write the stencil example using sessions

Summary of MPI Core Concept #1

- Assumed collective is the core pattern of the MPI programing model
  - MPI extensions supports adjacent collective patterns
    - RMA is a perturbation inside the collective setup
    - Dynamic processes and Sessions are perturbations outside the collective setup
  - Other perturbations that we didn't cover:
    - MPI_ANY_SOURCE / MPI_ANY_TAG
    - MPI_Probe
  - The perturbations bring additional flexibility and performance
    - They are likely to be supported once they are adopted in the MPI standard
    - Their performances are more likely depend on implementations

Argonne NATIONAL LABORATORY

Core MPI - Concept #2

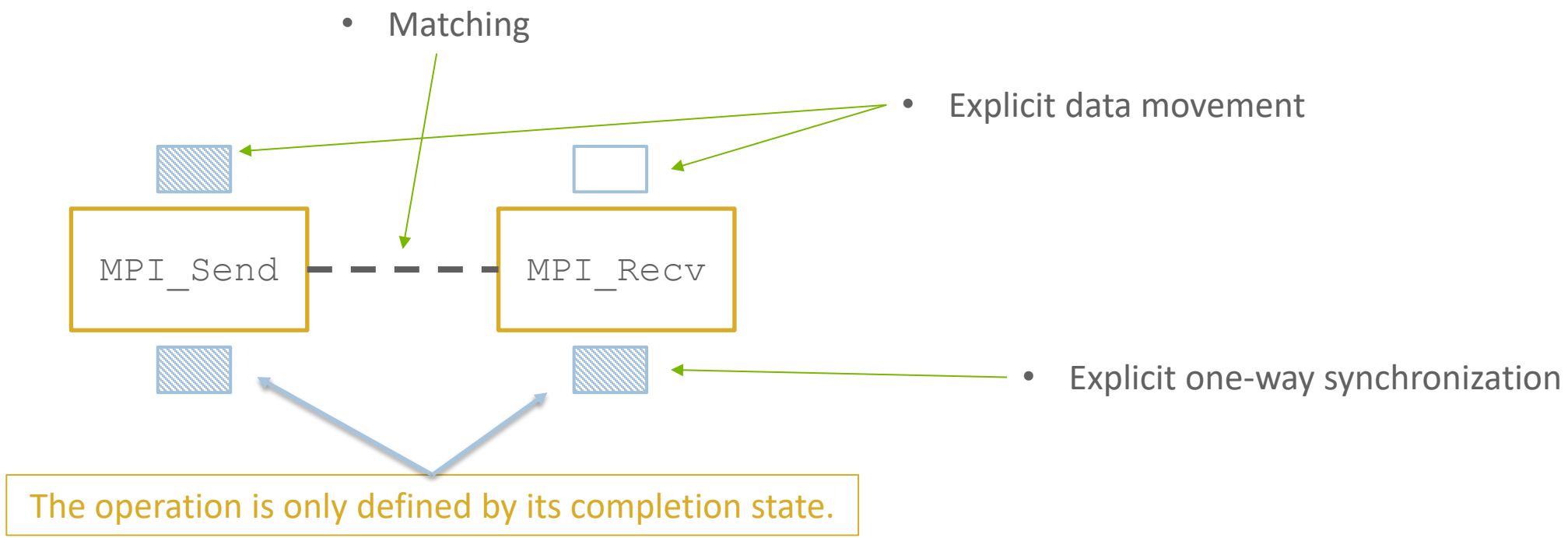
Progressive Abstractions

U.S. DEPARTMENT OF ENERGY

Concept #2 - Progressive Abstractions

- MPI programs are designed to be portable
- MPI semantics are sufficiently abstract so
  - It does not bound its implementation to specific hardware or platforms
  - It does not prevent specific optimizations for specific systems
- MPI semantics are sufficiently high level
  - It supports programming simplicity
  - It includes necessary elements to avoid traps
- Progressively exposing internal states to allow performance optimizations

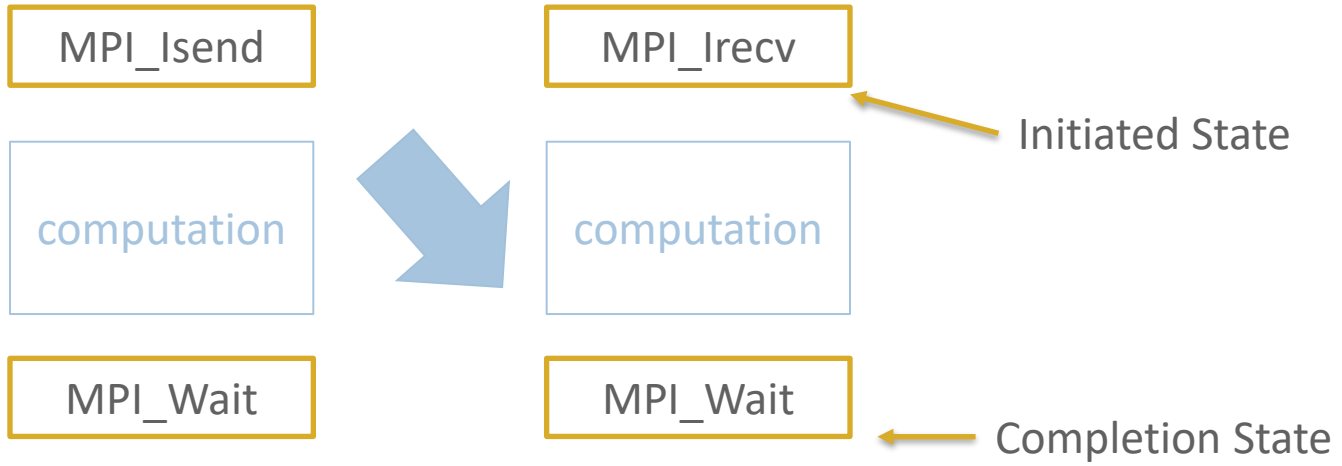
A Good MPI Abstraction – Blocking Send / Recv



Progressive Exposure of Internal States

Extending MPI by Progressive Exposure of Internal States

- Blocking operation cannot express concurrency
- Nonblocking extension enables the expression of concurrency
- Communication can happen concurrently to processors and overlapping is critical for performance



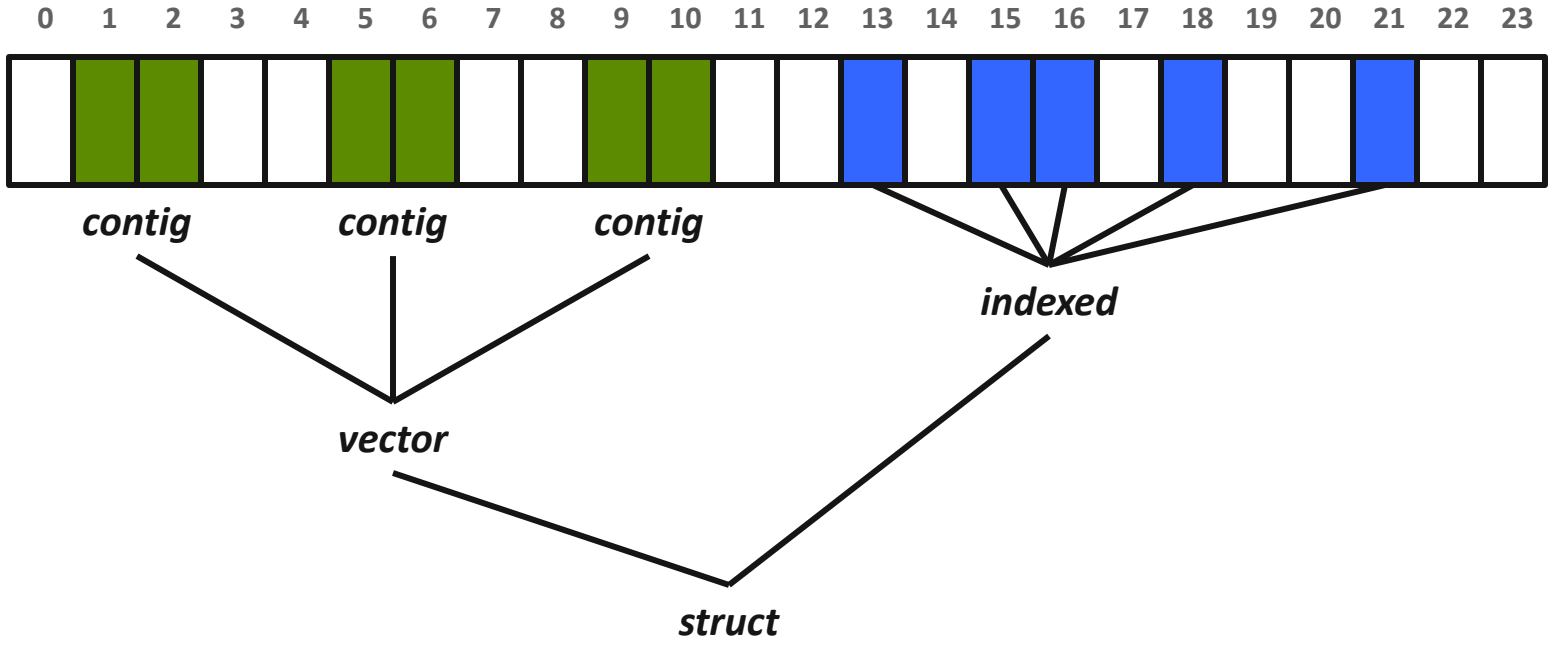
MPI Datatypes

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

Simple/Predefined Datatypes

- Equivalents exist for all C, C++ and Fortran native datatypes
  - C int → MPI_INT
  - C float → MPI_FLOAT
  - C double → MPI_DOUBLE
  - C uint32_t → MPI_UINT32_T
  - Fortran integer → MPI_INTEGER
- Derived datatypes to abstract complex data layout into a single data entity
  - Contiguous
  - Vector/Hvector
  - Indexed/Indexed_block/Hindexed/Hindexed_block
  - Struct
  - Some convenience types (e.g., subarray)

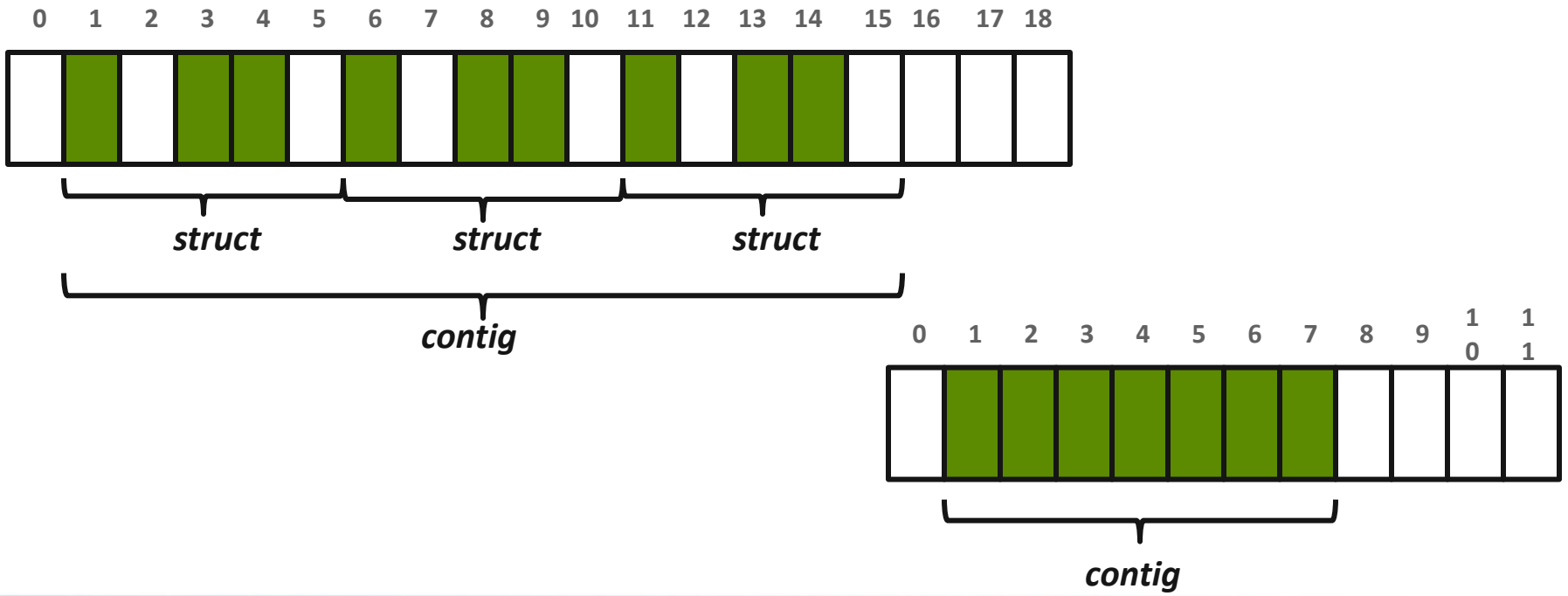
Derived Datatype Example



MPI_Type_contiguous

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

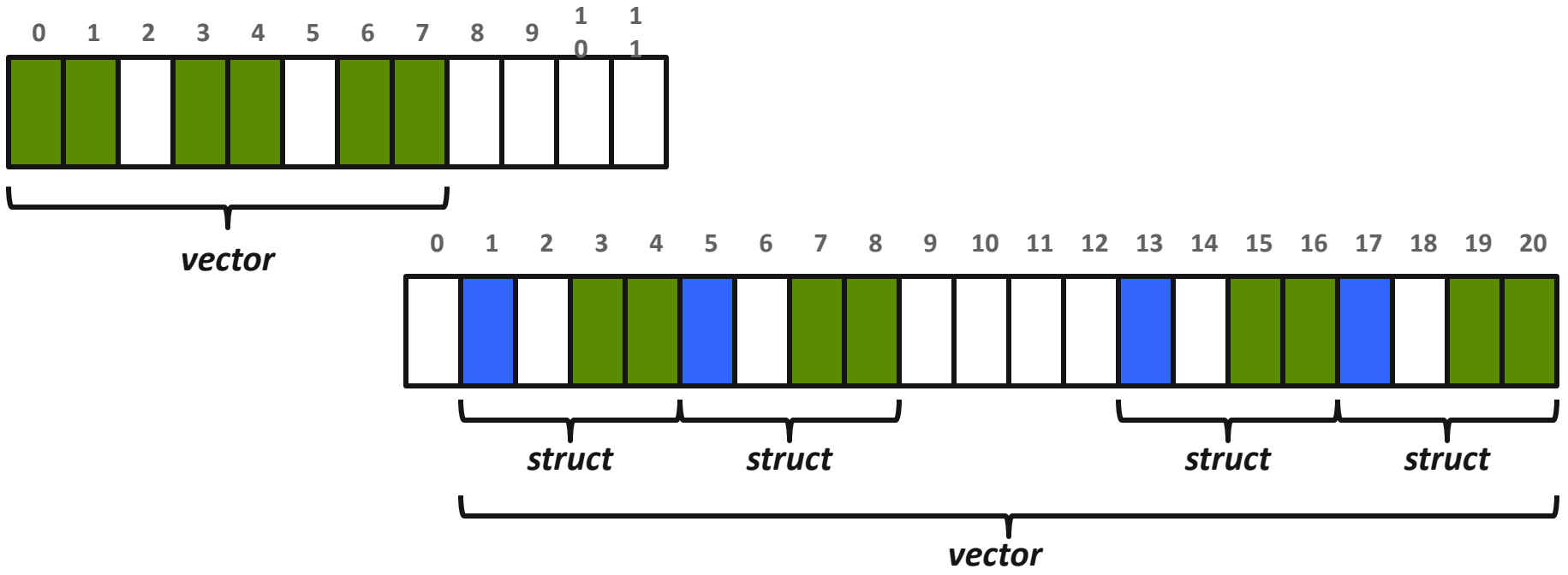
- Contiguous array of oldtype
- Should not be used as last type (can be replaced by count)



MPI_Type_vector

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

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



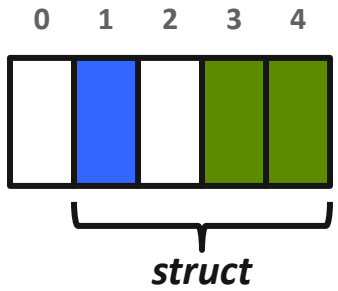
Commit, Free, and Dup

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

MPI_Type_create_struct

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

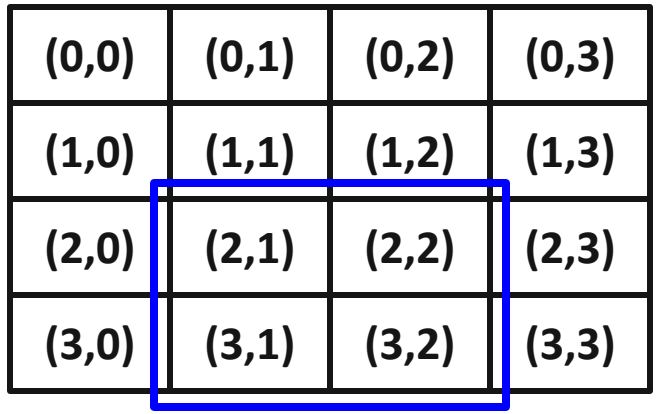
- Most general constructor, allows different types and arbitrary arrays (also most costly)



MPI_Type_create_subarray

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

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

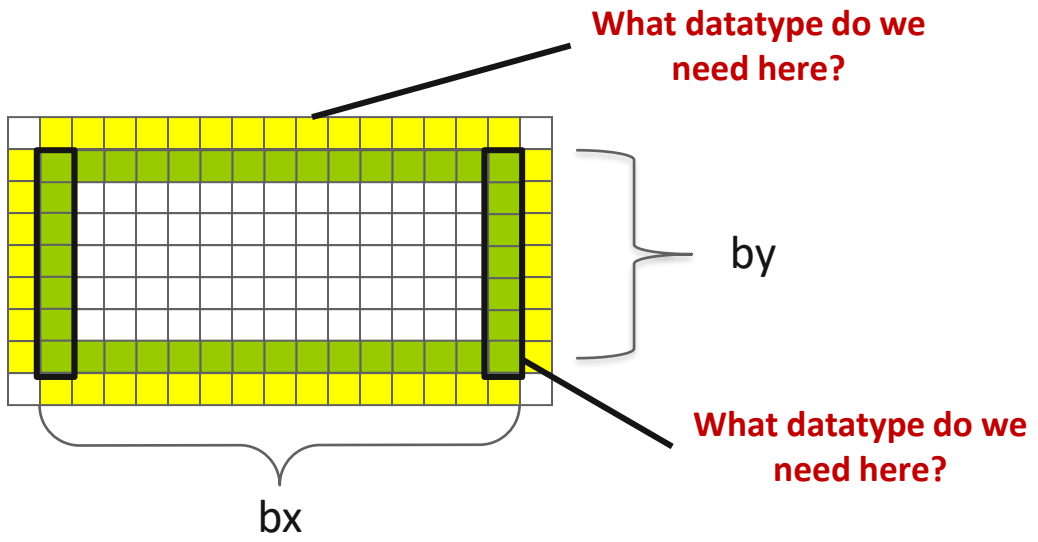


Datatype Summary

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

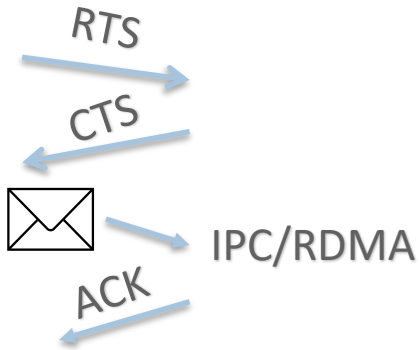
Exercise: Stencil with Derived Datatypes (1/2)

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



Message Modes

- Eager Mode
  - Send & forget
  - Good latency for small messages
  - Double copies
- Rendezvous Mode
  - Send header-only “Request-to-Send”
  - Protocol handshake & negotiations
  - Avoids extra copy and extra resource
  - Maximizes bandwidth but sacrifices latency



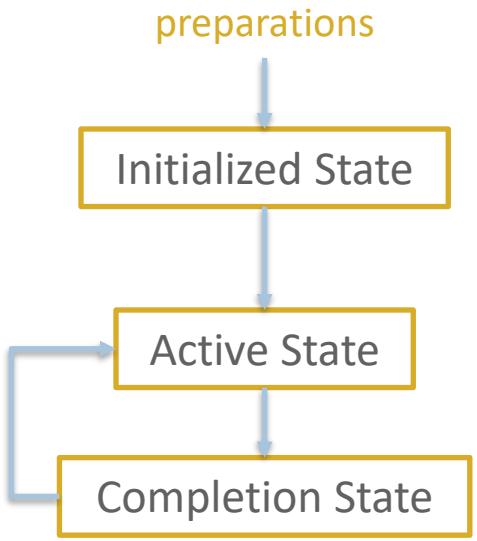
MPI Sends

- MPI_Send *(default send)*
  - Implementation decides message mode based on its own intelligence
  - May block due to the late-posting-recv
  - Or may not block if eager mode is taken
  - Recommended by default for best performance
- MPI_Ssend *(Synchronized send)*
  - Forcing send to wait for receiver to be posted, effectively the RNDV mode
- MPI_Bsend *(Buffered send)*
  - Forcing send to buffer and return
  - User need provide explicit buffer space
- MPI_Rsend *(Ready send)*
  - You know the recv is already posted
  - In practice, little advantage due to missing RTS/CTS

* Correct MPI program should still work if MPI_Send Is replaced with another send mode.

Persistent Requests

- MPI_{Send/Recv}_init -> MPI_Start -> MPI_Wait/MPI_Test -> repeat
- Nonblocking requests that are persistent
- Exposes an additional preparation state to implementations
- More complexity to juggle
- Rationale:
  - Potential persistent memory registration
  - Potential established protocols
  - Potential persistent user hints



Nonblocking Collectives

- MPI_Ibarrier, MPI_Ibcast, MPI_Iallgather, MPI_Iallreduce, …
- Similar rationale as nonblocking point-to-point
- However –
  - Collective performance model typically assumes synchrony
  - Collective schedules are more CPU/host reliant for progress
- Interesting use case – MPI_Ibarrier

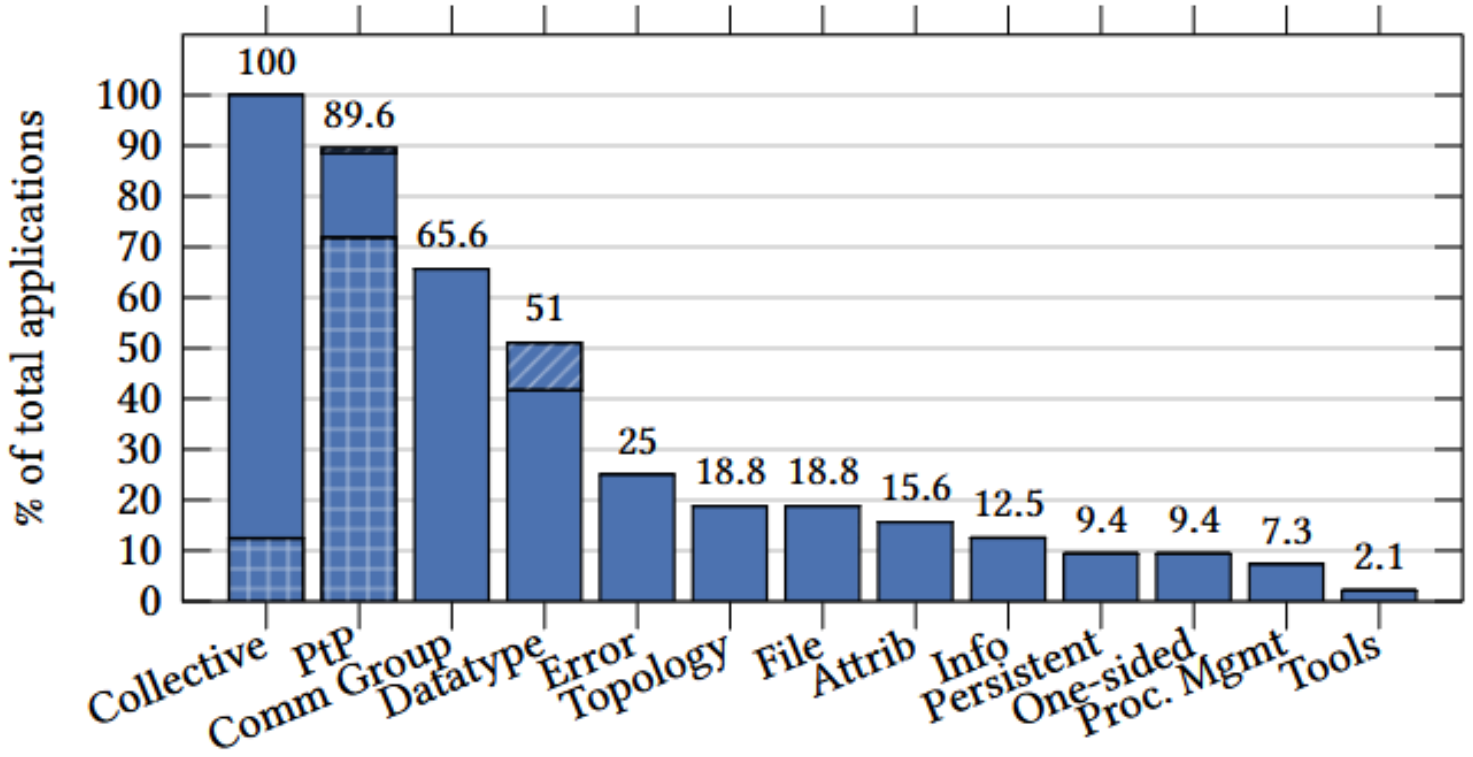
A Nonblocking Barrier?

- Semantics:
  - MPI_Ibarrier() – calling process entered the barrier, **no** synchronization happens
  - Synchronization **may** happen asynchronously
  - MPI_Test/Wait() – synchronization happens **if** necessary
- Uses:
  - It's like a hotel check-in: guests synchronize without necessarily meet!
  - Use the split semantics! Processes **notify** noncollectively but **synchronize** collectively!

87

Diminishing Returns

- Nonblocking point-to-point are much more popular than non-blocking collectives or persistent requests – WHY?
- There are good room inside implementation that can achieve portable performance without exposing unnecessary details.



Core MPI - Concept #3

Interoperability

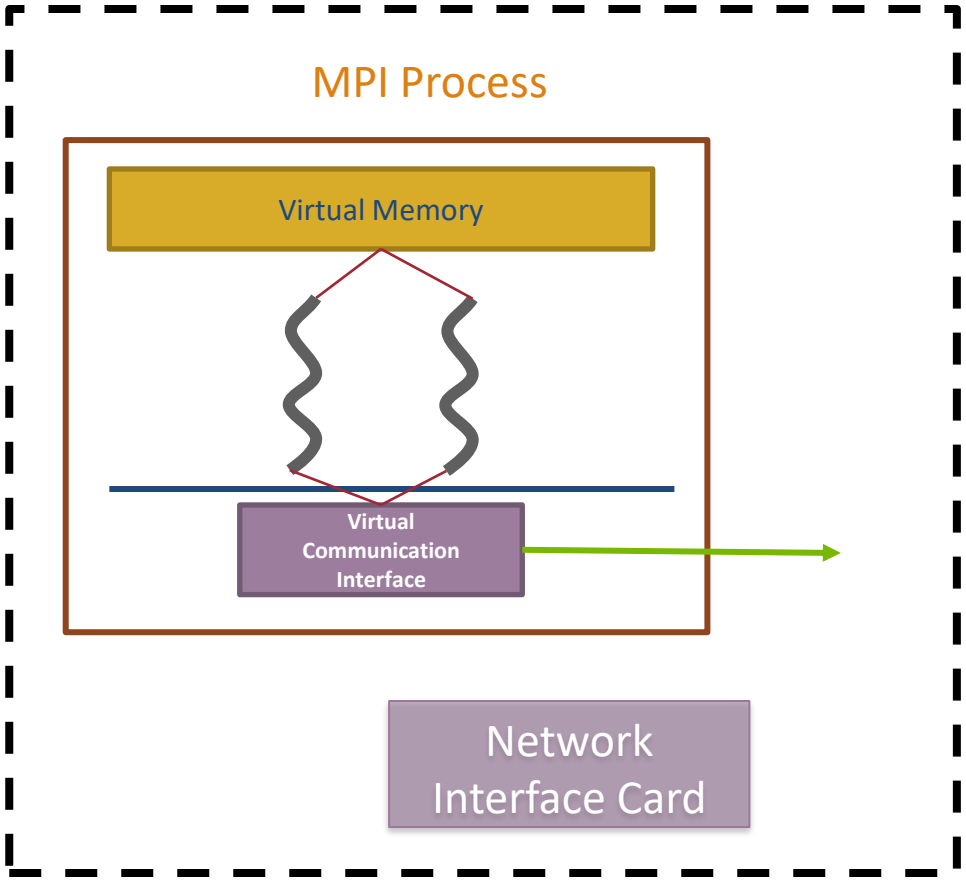
Concept #3 - Interoperability

- MPI is the *de facto* runtime for HPC systems
- The dominant programming patterns for HPC applications are BSP
- However, an HPC system should not only support BSP applications
- Other runtimes:
  - Tasks, RPC, OpenMP, CUDA, SYCL
- Options:
  - Implement alternative runtimes using MPI, predominantly MPI point-to-point APIs
  - MPI+X
    - MPI+Threads
    - MPI+GPUs

MPI+Threads

MPI+Threads

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



Argonne NATIONAL LABORATORY

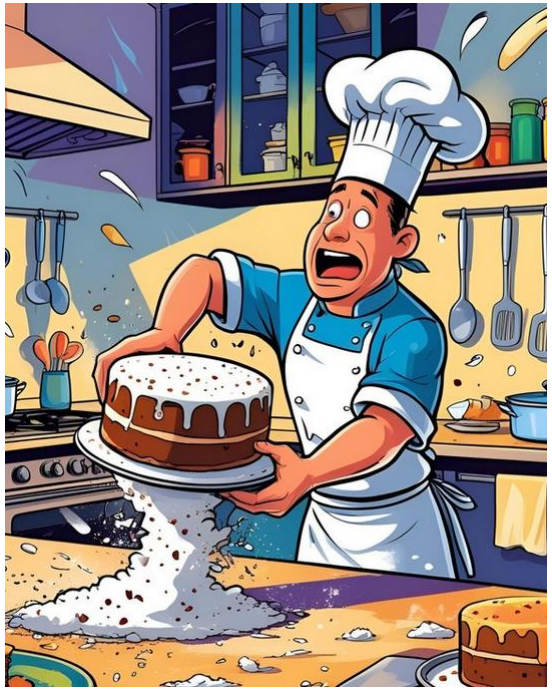
MPI+Threads – Part 1

Correctness

U.S. DEPARTMENT OF ENERGY

Exercise: MPI+Threads

- Just add an OpenMP parallel region, RIGHT?



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

What could go wrong?

MPI+Threads - What could go wrong - 1

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

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

MPI+Threads: Thread Levels

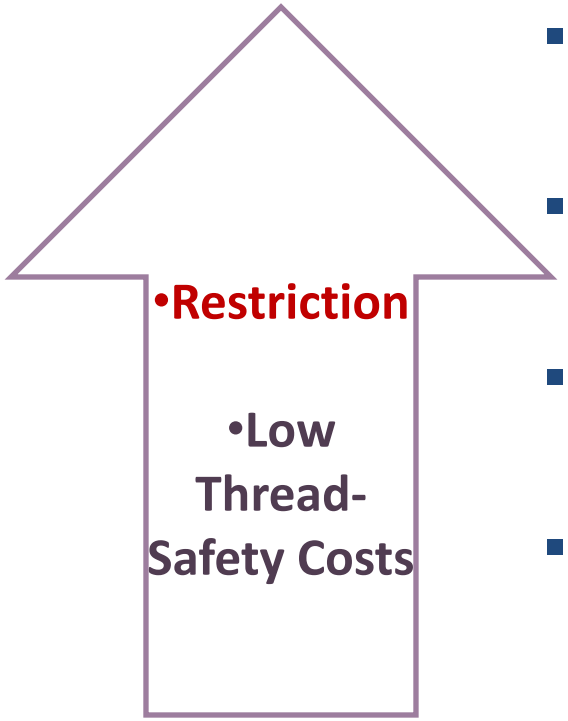
MPI_Init_thread(requested, provided)

MPI + Threads

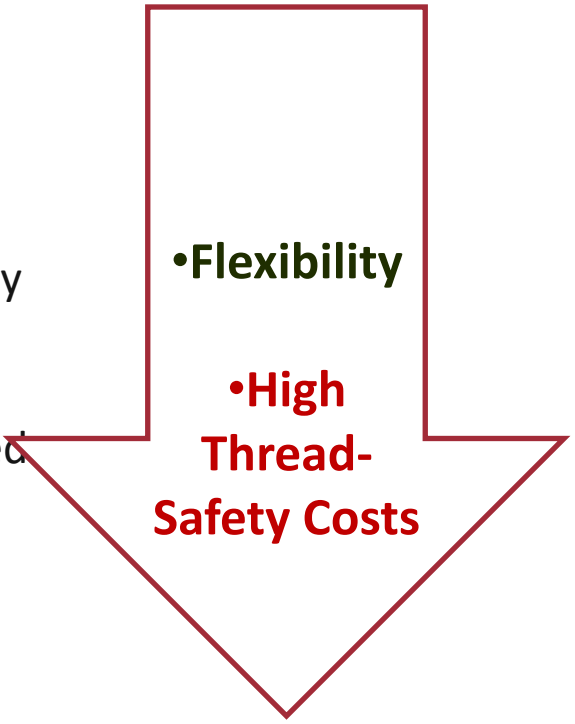
↓

Interoperability

Interoperation or thread levels:



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



Threads and MPI

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

MPI+Threads - What could go wrong - 2

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

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

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

problem:
Race conditions on message buffers!

MPI+Threads - What could go wrong - 3

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

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

problem:
Indeterministic message order!

Argonne NATIONAL LABORATORY

MPI+Threads – Part 2

Performance

U.S. DEPARTMENT OF ENERGY

(pre-assignment) Exercise: MPI+Threads performance

- Expectation: more threads drive more bandwidth

- Reality: ? (find out after we cover the benchmarking section)

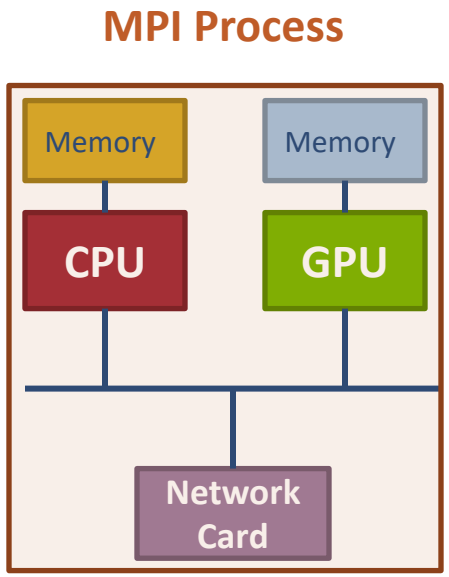
Tips for MPI+Threads

- By default, MPI need ensure global consistency between threads
  - Tip: avoid MPI communications in parallel regions
- In general, implicitly sharing context between threads are BAD!
  - Tip: use separate communicators for each thread.
- Optimized MPI implementations can route traffic from different communicators
  - MPICH internally can create multiple Virtual Communication Interfaces (VCIs)
  - Set `MPIR_CVAR_CH4_NUM_VCIS` to enable multiple VCIs in MPICH

MPI+GPU

MPI + GPU

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

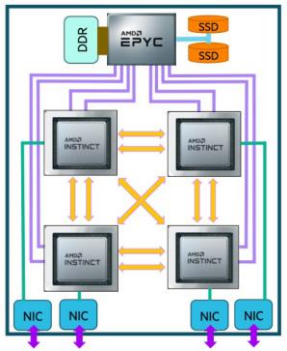


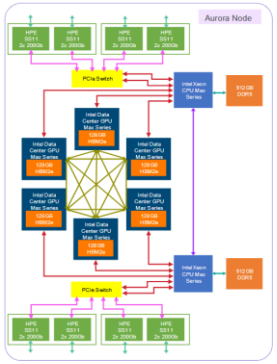
Exascale Supercomputers

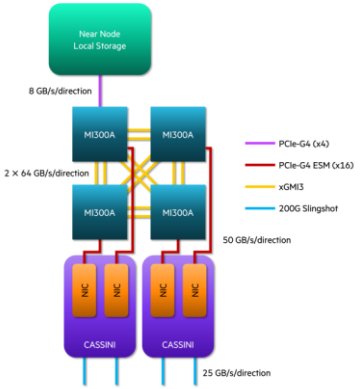
1 x AMD EPYC CPU
4 x AMD MI250X GPU

2 x Intel Xeon CPU
6 x Intel Data Center GPU

4 x AMD MI300A APU







Exercise: MPI+GPU

- *It just works!*
- But complicated for performance

Allocating memory:

```
/* openmp */

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

```
/* sycl */

void *buf_gpu = sycl::malloc_device<int>(count, q);

void *buf_gpu = sycl::malloc_device<int>(count, dev, ctx);
void *buf_host = sycl::malloc_host<int>(count, q);
void *buf_shared = sycl::malloc_shared(count, q);
```

MPI:

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

Compile:

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

```
$ mpicxx -fsycl …
```

Hybrid Memory Types

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

OpenMP routine	Type of allocation
Omp_target_alloc	Device
Omp_target_alloc_device	Device
Omp_target_alloc_host	Host
Omp_target_alloc_shared	Shared

SYCL routine	Type of allocation
sycl::malloc_device	Device
sycl::malloc_host	Host
sycl::malloc_shared	Shared

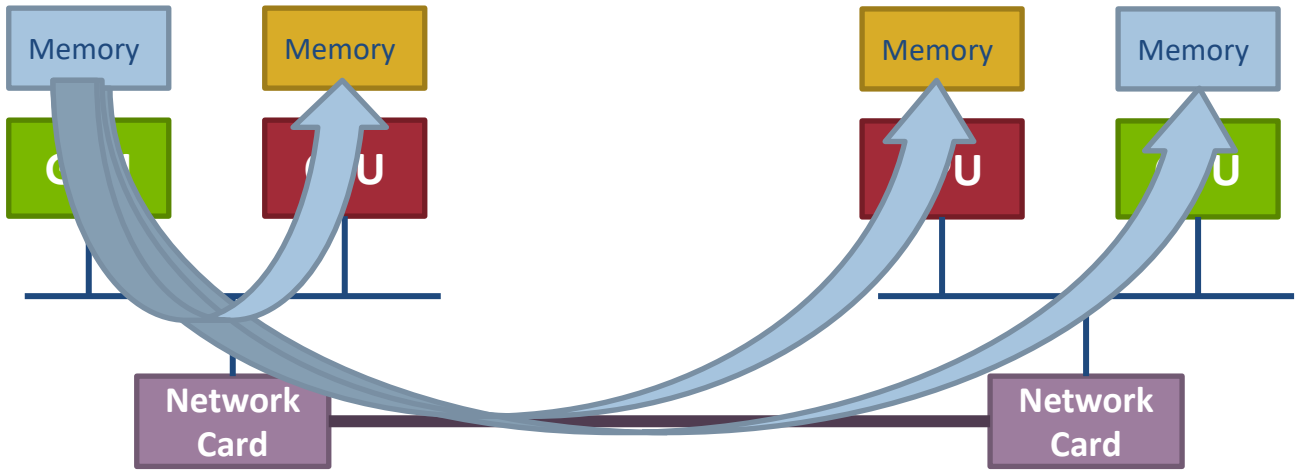
* Shared memory may limit MPI messaging performance

How MPI Moves Data Between Different Types Of Memory?

- Intranode or Internode
- Memory types
  - H-to-H
  - H-to-D
  - D-to-H
  - D-to-D

GPUs have separate physical memory subsystem

How to move data between GPUs with MPI?



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

GPU buffers with GPU-Unaware MPI implementations

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

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

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

Hands on: disable MPI GPU support: MPIR_CVAR_ENABLE_GPU=0

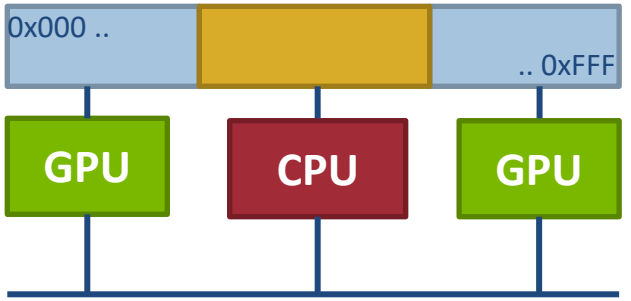
GPU buffers with GPU-Aware MPI implementations

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

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

Unified Virtual Addressing (UVA)

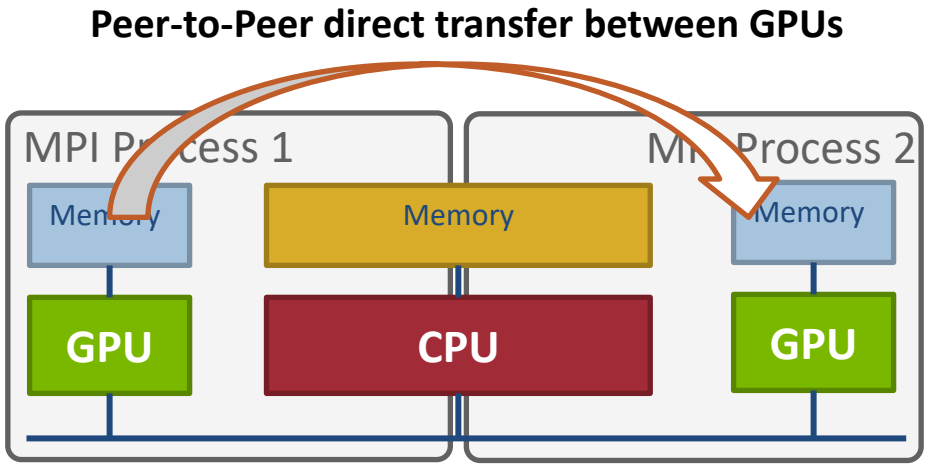
- UVA is a memory address management system supported in modern 64-bit architectures
  - Requires device driver support
- The same virtual address space is used for all processors, host or devices
- No distinction between host and device pointers
- The user can query the location of the data allocation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



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

Intranode Communication with UVA

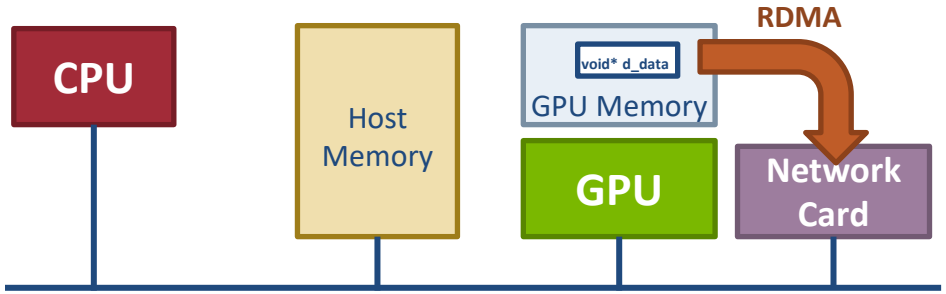
- Intranode Optimization
  - GPU peer-to-peer data transfers are possible
  - MPI can directly move data between GPU devices



GPU Direct IPC

Internode Communication with UVA

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



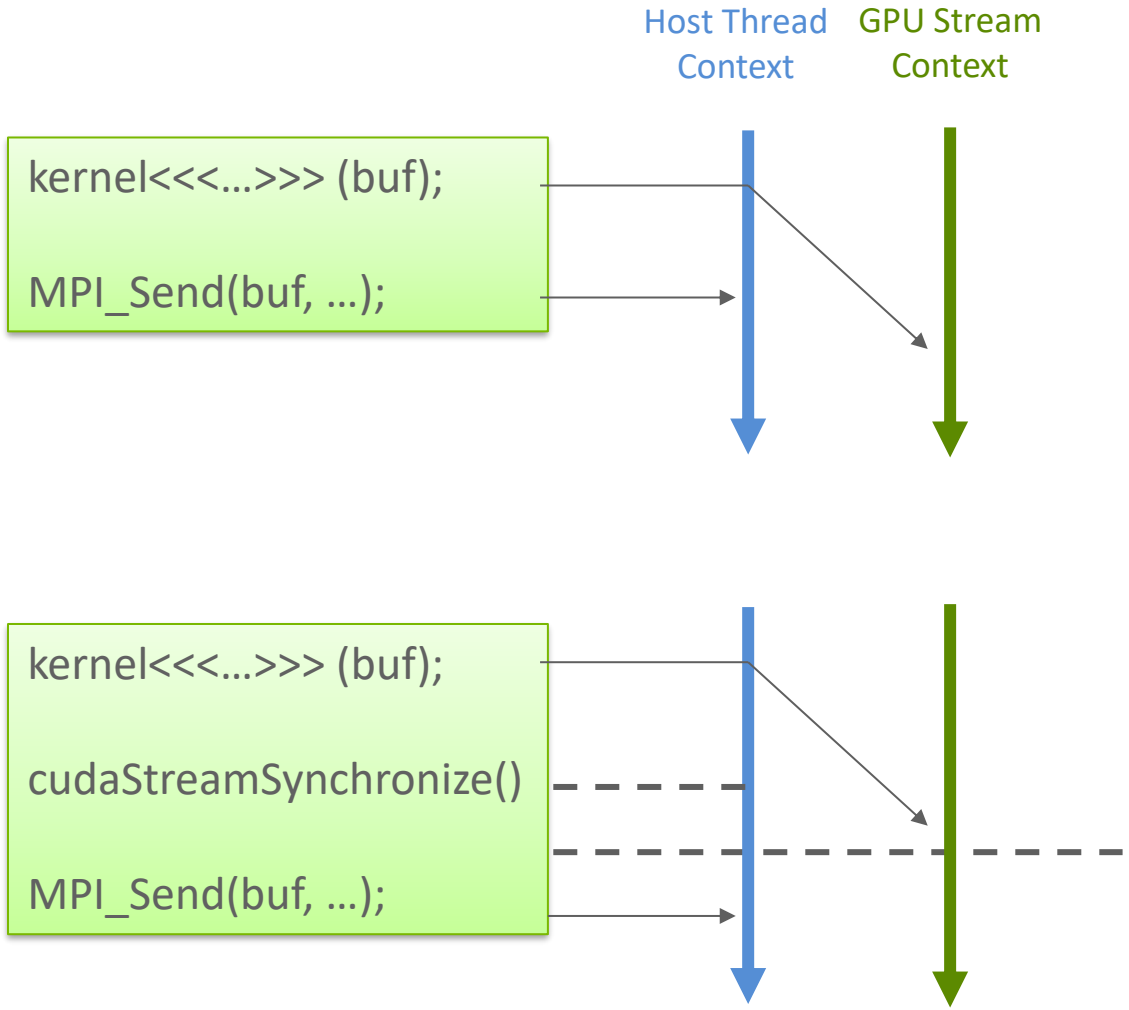
GPU Direct RDMA

Interoperability with MPI - Execution Context

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

Future MPI

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



(pre-assignement) Exercise: MPI+GPU performance

- Compare the performance of various scenarios:
  - Host – to – Host
  - Host – to – Device
  - Device – to – Device
  - Inter-tiles
  - Inter-gpus
  - Inter-nodes
- Don’t assume. Measure!

Argonne NATIONAL LABORATORY

Benchmarking MPI

U.S. DEPARTMENT OF ENERGY

Warm-up Exercise: Basic Send/Recv

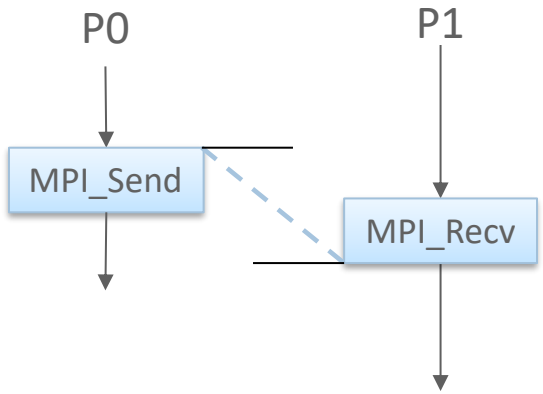
1. Start with a MPI skeleton
2. Setup variables
3. Perform Send/Recv
4. Compile and test

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

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

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

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



Optional:

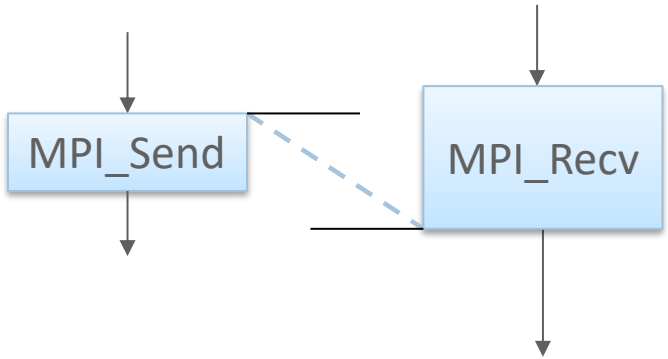
- Check MPI_Comm_size.
- Initialize send buffer and verify received data.
- Check MPI_Status.
- Use MPI_BYTE to count message size in bytes.

Reference: hybrid/basic_sendrecv.c

Semantics of MPI Messaging

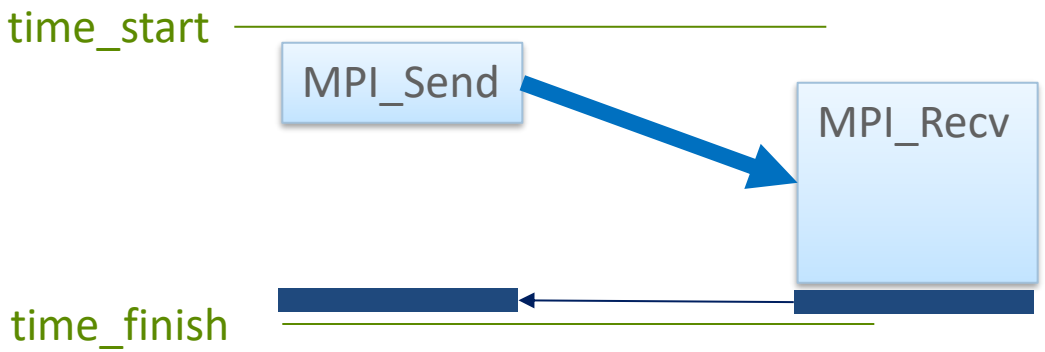
- What does an MPI message accomplish?
  - Data movement
  - Synchronization
- Synchronization
  - Performance metric
    - Latency
- Data transfer
  - Performance metric
    - Bandwidth

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



Exercise: Measure Bandwidth

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



- Use a large message size

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

- Use MPI_Wtime to measure time

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

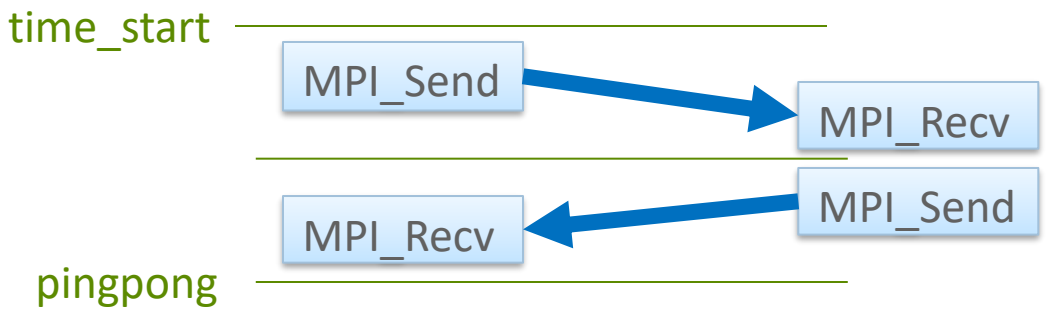
- Use a zero-sized message to synchronize completion

Reference: hybrid/bandwidth.c

Optional:
- Repeat to observe variations.
- Vary message size.
- Intranode vs. internode.
- Intra-NUMA vs. inter-NUMA.

Exercise: Measure Latency

$$Latency = Elapased\ Time$$
(µs)



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

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

Optional:
- Observe measurement variations
- Vary message size
- Intranode vs. internode
- Intra-NUMA vs. inter-NUMA

Reference: hybrid/latency.c

Example results

- Note the transition from latency-bound to bandwidth-bound

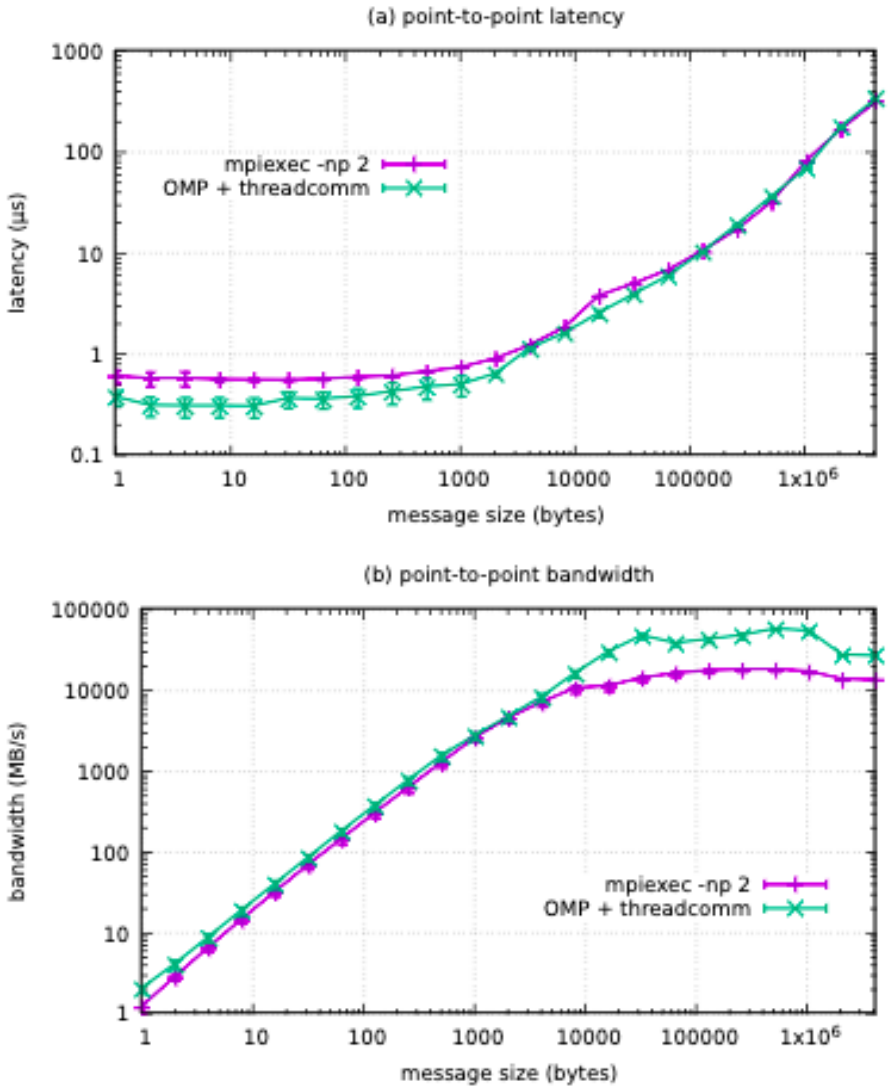
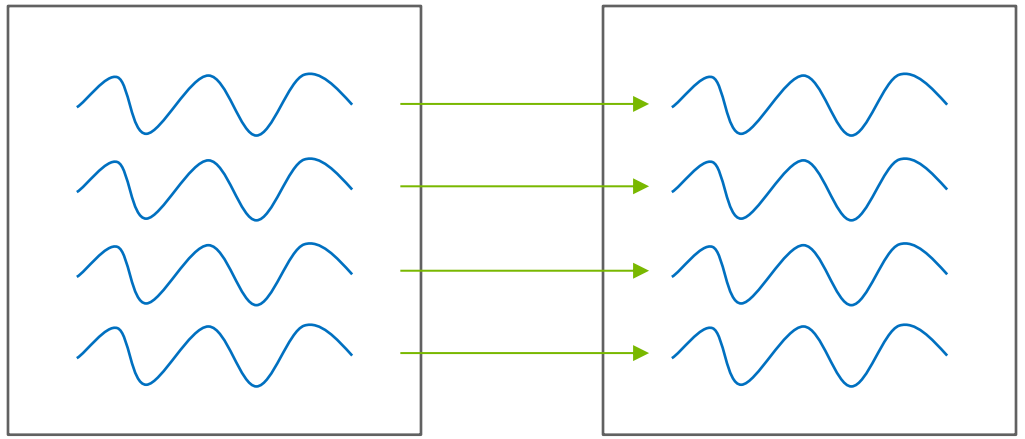


Figure 3: Point-to-point message latency and bandwidth comparison between MPI-everywhere and OpenMP+threadcomm on an Intel Xeon Gold 5317. Processes or threads are bound to cores on the same socket.

Exercise: Measure MPI+Threads Performance

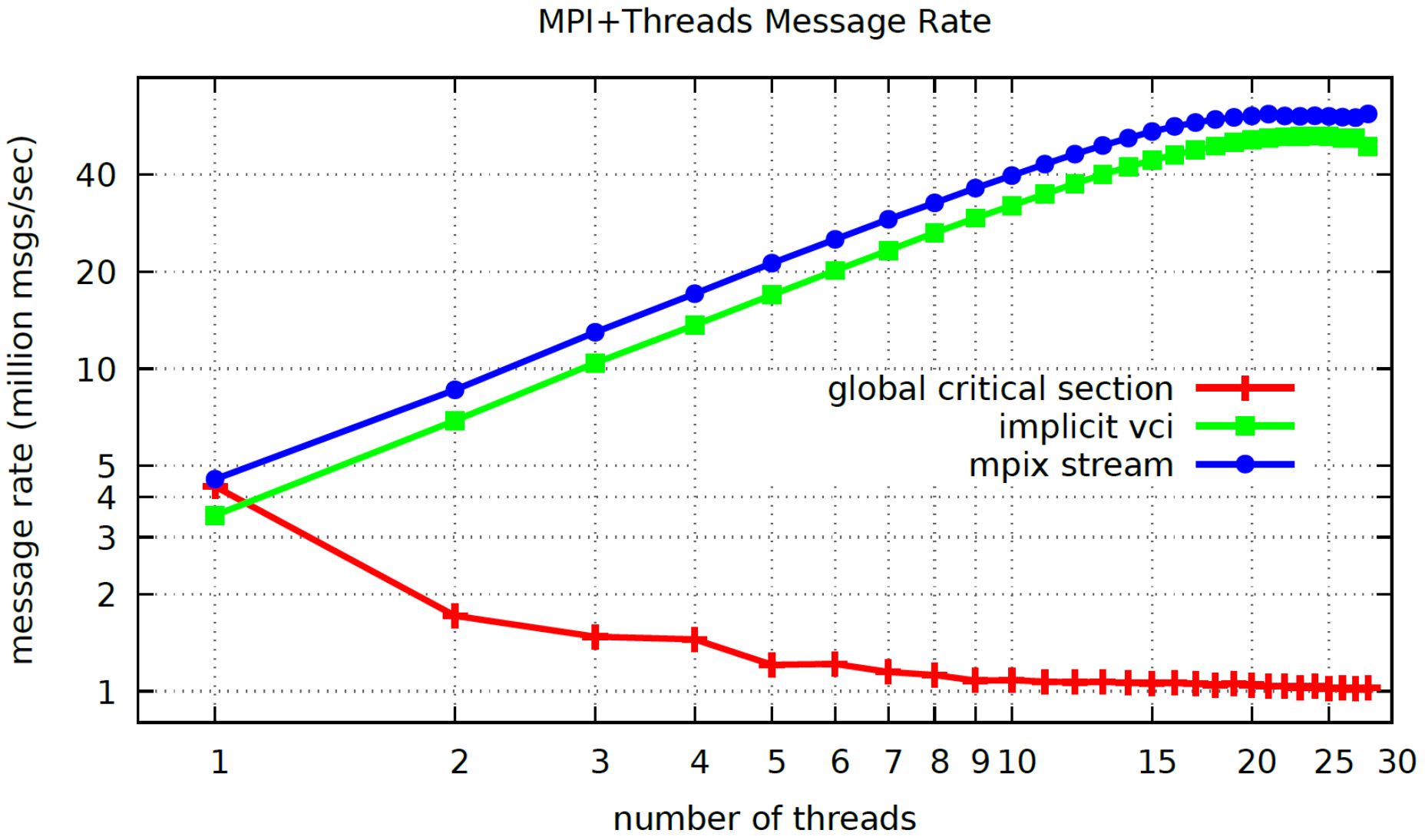
- Benchmark: send messages concurrently in multiple threads
  - Use 2 nodes to measure internode performance
  - Per-thread latency
  - Per-thread bandwidth
  - Aggregate bandwidth
  - Vary the number of threads

Pair-wise Messaging



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

Example result



Core MPI Summary

- **Concept #1 – Assumed Collective Agreement**
  - MPI programs assume a shared, collective execution context
  - RMA, Dynamic Processes, and Sessions each relax that assumption in a different way
- **Concept #2 – Progressive Abstraction**
  - MPI stays portable by keeping semantics abstract rather than tied to hardware
  - Extended APIs progressively expose more internal state for performance
- **Concept #3 – Interoperability**
  - MPI is the dominant HPC runtime and must coexist with threads and GPUs
  - Thread levels and GPU-aware data paths need explicit attention from the programmer
- *Across all three: understand your assumptions, measure before you optimize, and lean on your MPI implementation*

Exercise and Questions