Argonne NATIONAL LABORATORY

MPI for Scalable Computing

https://anl.box.com/v/PETASCALE-MPI-2019/

Yanfei Guo Ken Raffenetti Giuseppe Congiu

Argonne National Laboratory

U.S. DEPARTMENT OF ENERGY

Outline

- Monday (1.5hrs)
  - Introduction to MPI
    - Communicators
    - Point-to-point communication
  - MPI Datatypes
  - Hands-on exercise
- Tuesday (2hrs)
  - Collective Communication
  - Remote Memory Access (RMA)
  - Hybrid programming
    - MPI + accelerators
  - Hands-on exercises

What is MPI?

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

Timeline of the MPI Standard

- MPI-1 (1994), presented at SC'93
  - Basic point-to-point communication, collectives, datatypes, etc
- MPI-2 (1997)
  - Added parallel I/O, Remote Memory Access (one-sided operations), dynamic processes, thread support, C++ bindings, …
- ---- Unchanged for 10 years ----
- MPI-2.1 (2008)
  - Minor clarifications and bug fixes to MPI-2
- MPI-2.2 (2009)
  - Small updates and additions to MPI 2.1
- MPI-3.0 (2012)
  - Major new features and additions to MPI (nonblocking collectives, neighborhood collectives, improved RMA, tools interface, Fortran 2008 bindings, etc.)
- MPI-3.1 (2015)
  - Small updates to MPI 3.0

Status of MPI-3.1 Implementations

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

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

Platform-specific restrictions might apply to the supported features

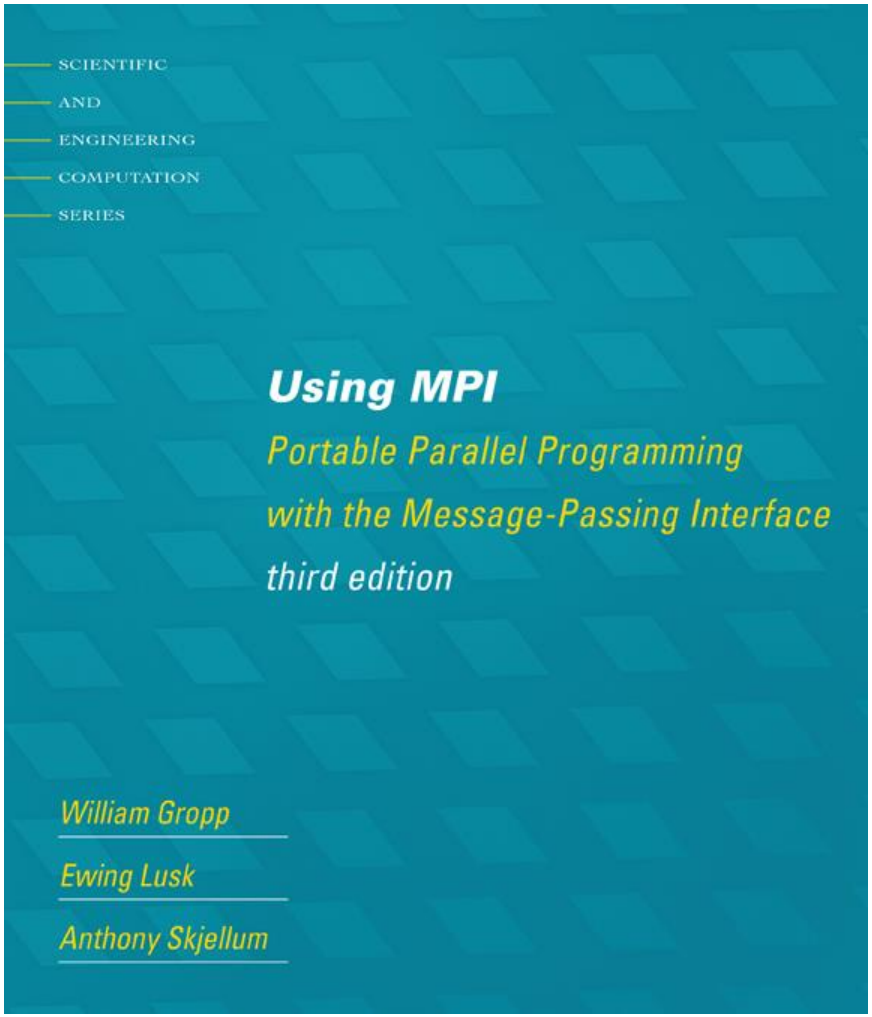
[1] Open Source but unsupported [2] No MPI_T variables exposed * Under development (*) Partly done

Web Pointers

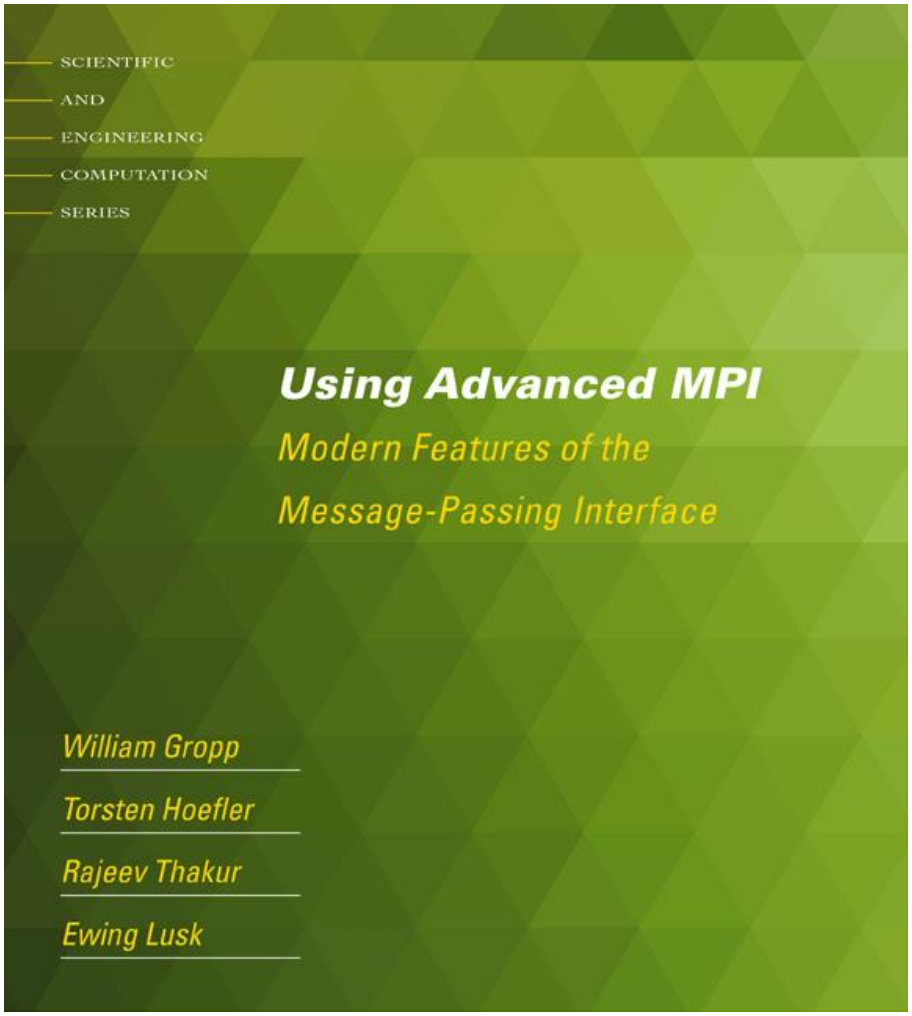
- MPI Standard : http://www.mpi-forum.org/docs/docs.html
- MPI Forum : http://www.mpi-forum.org/

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

Tutorial Books on MPI (November 2014)



Basic MPI



Advanced MPI, including MPI-2 and MPI-3

Important considerations while using MPI

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

Compiling MPI Programs

- Compilation Wrappers
  - For C programs:
  - For C++ programs:
  - For Fortran programs:

```
% mpicc test.c -o test
```

```
% mpicxx test.cpp -o test
```

```
% mpifort test.f90 -o test
```

- You can link other libraries are required too
  - To link to a math library:

```
% mpicc test.c -o test -lm
```

- You can just assume that “mpicc” and friends have replaced your regular compilers (gcc, gfortran, etc.)

Compiling MPI Programs on Cray Systems

- Compilation Wrappers
  - For C programs:
  - For C++ programs:
  - For Fortran programs:

```
% cc test.c -o test
```

```
% c++ test.cpp -o test
```

```
% ftn test.f90 -o test
```

- You can link other libraries
  - To link to a math library:

```
% cc test.c -o test -lm
```

- Cray MPI libraries are automatically linked

Running MPI Programs

- Launch 16 processes on the local node:

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

- Launch 16 processes on 4 nodes (each has 4 cores)

```
% mpiexec -hosts h1:4,h2:4,h3:4,h4:4 -n 16 ./test
```

  - Runs the first four processes on h1, the next four on h2, etc.

```
% mpiexec -hosts h1,h2,h3,h4 -n 16 ./test
```

  - Runs the first process on h1, the second on h2, etc., and wraps around
  - So, h1 will have the 1st, 5th, 9th and 13th processes

- If there are many nodes, it might be easier to create a host file

```
% cat hf
      h1:4
      h2:2
% mpiexec -hostfile hf -n 16 ./test
```

Interaction with Resource Managers

- Resource managers such as SGE, PBS, SLURM or Cray ALPS are common in many managed clusters
  - MPI automatically detects them and interoperates with them
- For example with PBS, you can create a script such as:

```
#! /bin/bash
% cd $PBS_O_WORKDIR
# No need to provide -np or -hostfile options
% mpiexec ./test
```

- Job can be submitted as: `% qsub -l nodes=2:ppn=2 test.sub`
  - "mpiexec" will automatically know that the system has PBS, and ask PBS for the number of cores allocated (4 in this case), and which nodes have been allocated
- The usage is similar for other resource managers

Running MPI Programs on Cray Systems

- Launch 16 processes within a job allocation:

```
% aprun –n 16 ./test
```

Example: Hello World!

- *communicators/hello.c*
- Basic program where each process prints its rank

Introduction to MPI Communicators

MPI Communicators

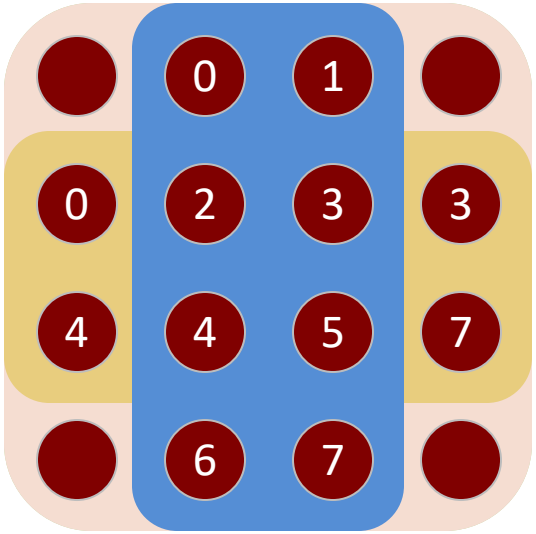
- MPI processes are collected into groups
  - Each group can have multiple colors (some times called context)
  - *Group + color == communicator (it is like a name for the group)*
  - When an MPI application starts, the group of all processes is initially given a predefined name called **`MPI_COMM_WORLD`**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

Communicators

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

Communicators do not need to contain all processes in the system

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



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

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

The same process might have different ranks in different communicators

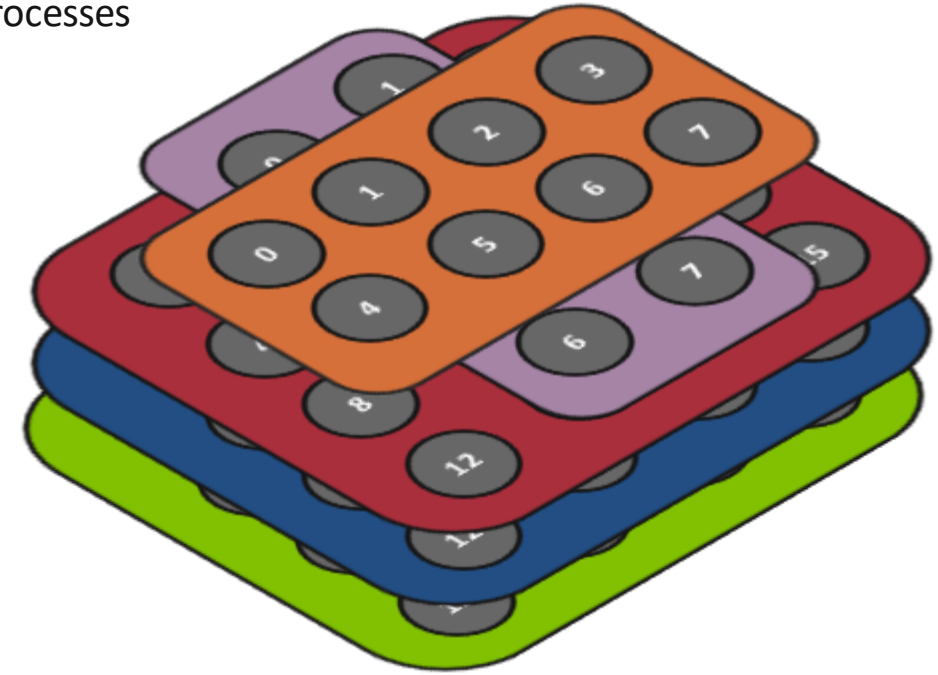
Communicators can be created “by hand” or using tools provided by MPI (not discussed in this tutorial)

Simple programs typically only use the predefined communicator **MPI_COMM_WORLD**

Communicators

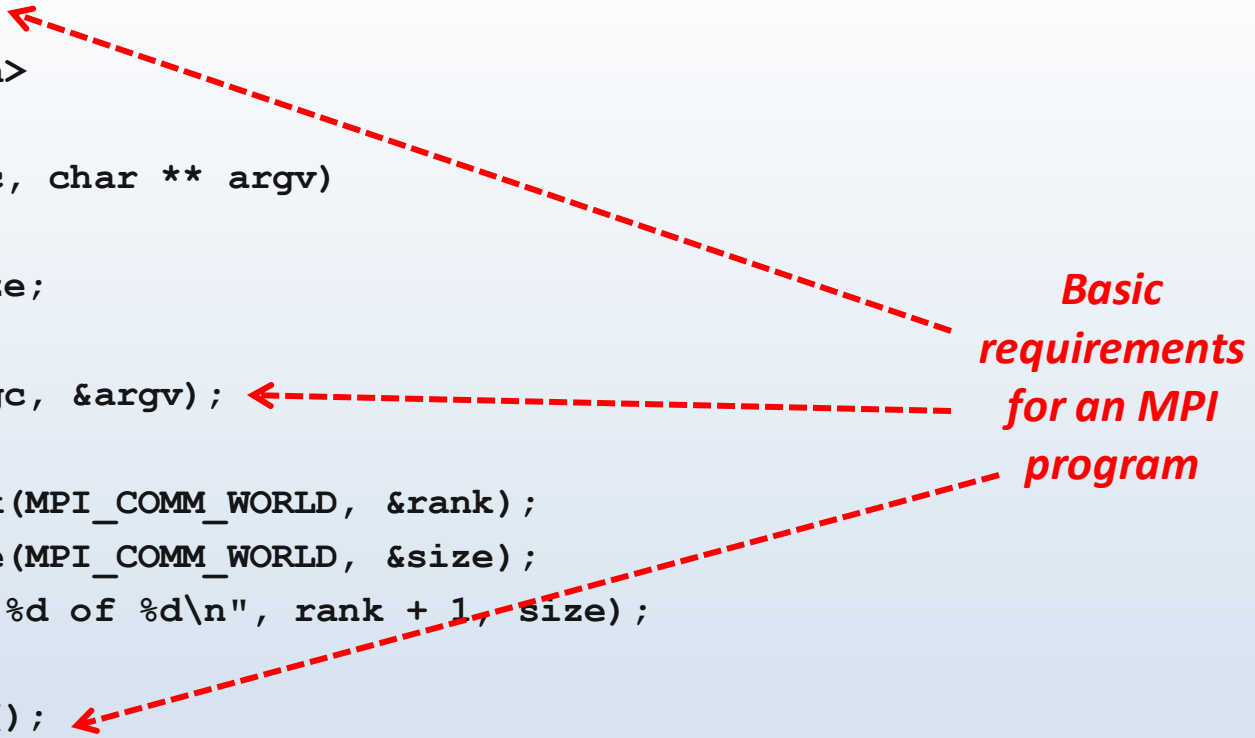
Can be thought of as independent communication layers over a group of processes

Messages in one layer will not affect messages in another



Simple MPI Program Identifying Processes

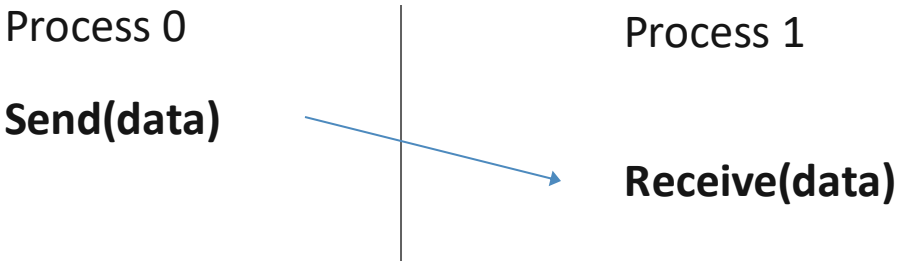
```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char ** argv)
{
    int rank, size;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("I am %d of %d\n", rank + 1, size);

    MPI_Finalize();
    return 0;
}
```



Blocking Point-to-Point Operations

MPI Basic Send/Receive

- Simple communication model



- Application needs to specify to the MPI implementation:
  1. How do you compile and run an MPI application?
  2. How will processes be identified?
  3. How will “data” be described?

Point-to-point Communication

- Point-to-point communication in MPI is like email exchange
  - One process sends a copy of the data to another process (or a group of processes), and the other process receives it
- Communication requires the following information:
  - Sender has to know:
    - Whom to send the data to (receiver's process rank)
    - What kind of data to send (100 integers or 200 characters, etc)
    - A user-defined "tag" for the message (think of it as an email subject; allows the receiver to understand what type of data is being received)
  - Receiver "might" have to know:
    - Who is sending the data (OK if the receiver does not know; in this case sender rank will be **`MPI_ANY_SOURCE`**, meaning anyone can send)
    - What kind of data is being received (partial information is OK: I might receive *up to* 1000 integers)
    - What the user-defined "tag" of the message is (OK if the receiver does not know; in this case tag will be **`MPI_ANY_TAG`**)

More Details on Describing Data for Communication

- MPI Datatype is very similar to a C or Fortran datatype
  - **`int`** → **`MPI_INT`**
  - **`double`** → **`MPI_DOUBLE`**
  - **`char`** → **`MPI_CHAR`**
- More complex datatypes are also possible:
  - E.g., you can create a structure datatype that comprises of other datatypes → a char, an int and a double.
  - Or, a vector datatype for the columns of a matrix
- The "count" in **`MPI_SEND`** and **`MPI_RECV`** refers to how many datatype elements should be communicated

MPI Basic (Blocking) Send

```
MPI_Send(const void *buf, int count,
         MPI_Datatype datatype, int dest, int tag,
         MPI_Comm comm)
```

- The message buffer is described by (**`buf, count, datatype`**).
- The target process is specified by **`dest`** and **`comm`**.
  - **`dest`** is the rank of the target process in the communicator specified by **`comm`**.
- **`tag`** is a user-defined "type" for the message
- When this function returns, the data has been delivered to the system and the buffer can be reused.
  - The message may not have been received by the target process.

MPI Basic (Blocking) Receive

```
MPI_Recv(void *buf, int count, MPI_Datatype datatype,
         int source, int tag, MPI_Comm comm, MPI_Status *status)
```

- Waits until a matching (on **source**, **tag**, **comm**) message is received from the system, and the buffer can be used.
- **source** is rank in communicator **comm**, or **MPI_ANY_SOURCE**.
- Receiving fewer than **count** occurrences of **datatype** is OK, but receiving more is an error.
- **status** contains further information:
  - Who sent the message (can be used if you used **MPI_ANY_SOURCE**)
  - How much data was actually received
  - What tag was used with the message (can be used if you used **MPI_ANY_TAG**)
  - **MPI_STATUS_IGNORE** can be used if we don't need any additional information

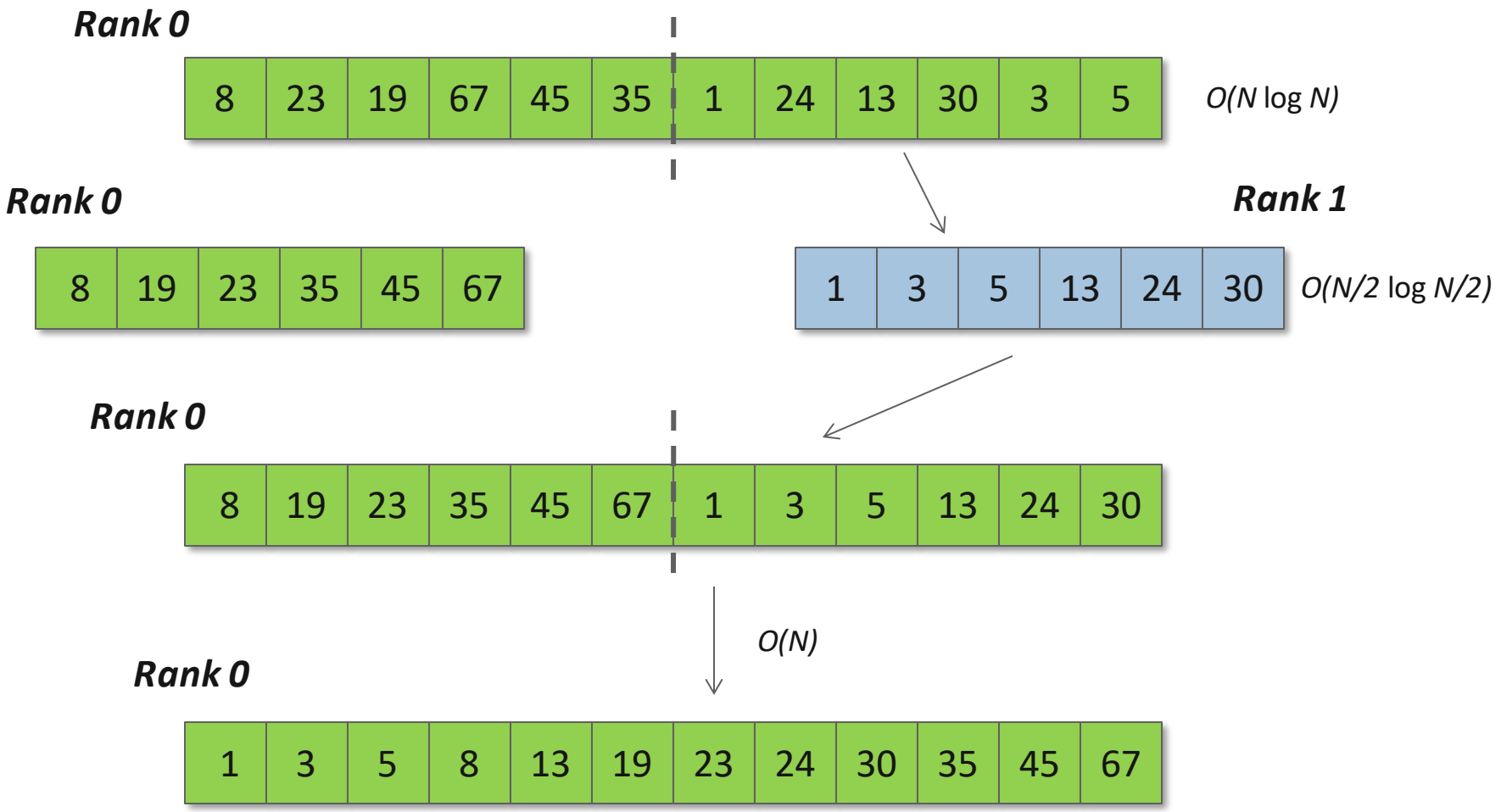
Simple Communication in MPI

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char ** argv)
{
    int rank, data[100];

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0)
        MPI_Send(data, 100, MPI_INT, 1, 0, MPI_COMM_WORLD);
    else if (rank == 1)
        MPI_Recv(data, 100, MPI_INT, 0, 0, MPI_COMM_WORLD,  MPI_STATUS_IGNORE);

    MPI_Finalize();
    return 0;
}
```

Example: Basic Send/Receive

- *blocking_p2p/sendrecv.c*
- Simple send/receive program to show basic data transfer

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;
}
```

Example: Sorting with Two Processes

- *blocking_p2p/sort_2_procs.c*
- Sorting using two processes

Status Object

- The status object is used after completion of a receive to find the actual length, source, and tag of a message
- Status object is MPI-defined type and provides information about:
  - The source process for the message (`status.MPI_SOURCE`)
  - The message tag (`status.MPI_TAG`)
  - Error status (`status.MPI_ERROR`)
- The number of elements received is given by:

```
MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)
```

`status`	return status of receive operation (status)
`datatype`	datatype of each receive buffer element (handle)
`count`	number of received elements (integer)(OUT)

Using the “status” field (contd.)

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char ** argv)
{
    [..snip..]
    if (rank < world_size - 1) {  /* worker process */
        MPI_Send(data, rand() % 100, MPI_INT, world_size - 1, rank / 2,
                 MPI_COMM_WORLD);
    }
    else {  /* master process */
        for (i = 0; i < world_size - 1; i++) {
            MPI_Recv(data, 100, MPI_INT, MPI_ANY_SOURCE,
                     MPI_ANY_TAG, MPI_COMM_WORLD, &status);
            MPI_Get_count(&status, MPI_INT, &count);
            printf("worker ID: %d; task ID: %d; count: %d\n",
                   status.MPI_SOURCE, status.MPI_TAG, count);
        }
    }
    [..snip..]
}
```

Example: Master-worker Computation

- *blocking_p2p/master_worker_simple.c*
- Master-worker example
  - Each task is computed by three workers
  - Every worker knows statically what task it needs to compute
  - Results accumulated by the master
  - The amount of data sent by each worker is arbitrary, but less than 100 integers

Section Summary

- MPI is simple
- Many (if not all) parallel programs can be written using just these six functions, only two of which are non-trivial:
  - **`MPI_INIT`** – initialize the MPI library (must be the first routine called)
  - **`MPI_COMM_SIZE`** – get the size of a communicator
  - **`MPI_COMM_RANK`** – get the rank of the calling process in the communicator
  - **`MPI_SEND`** – send a message to another process
  - **`MPI_RECV`** – send a message to another process
  - **`MPI_FINALIZE`** – clean up all MPI state (must be the last MPI function called by a process)
- For performance, however, you need to use other MPI features

Nonblocking Point-to-Point Operations

Blocking vs. Nonblocking Communication

- **`MPI_SEND/MPI_RECV`** are blocking communication calls
  - Return of the routine implies completion
  - When these calls return the memory locations used in the message transfer can be safely accessed for reuse
  - For “send” completion implies variable sent can be reused/modified
  - Modifications will not affect data intended for the receiver
  - For “receive” variable received can be read
- **`MPI_ISEND/MPI_IRECV`** are nonblocking variants
  - Routine returns immediately – completion has to be separately tested for
  - These are primarily used to overlap computation and communication to improve performance

Blocking Communication

- In blocking communication
  - **MPI_SEND** does not return until buffer is empty (available for reuse)
  - **MPI_RECV** does not return until buffer is full (available for use)
- A process sending data will be blocked until data in the send buffer is emptied
- A process receiving data will be blocked until the receive buffer is filled
- Exact completion semantics of communication generally depends on the message size and the system buffer size
- Blocking communication is simple to use but can be prone to deadlocks

```
if (rank == 0) {
    MPI_SEND(..to rank 1..)
    MPI_RECV(..from rank 1..)
} else if (rank == 1) {
    MPI_SEND(..to rank 0..)
    MPI_RECV(..from rank 0..)
}
```

Usually deadlocks → ← reverse send/recv

Nonblocking Communication

- Nonblocking operations return (immediately) "request handles" that can be waited on and queried

```
MPI_ISEND(buf, count, datatype, dest, tag, comm, request)
MPI_IRECV(buf, count, datatype, src, tag, comm, request)
MPI_WAIT(request, status)
```

- Nonblocking operations allow overlapping computation and communication
- One can also test without waiting using **MPI_Test**

```
MPI_Test(request, flag, status)
```

- Anywhere you use **MPI_Send** or **MPI_Recv**, you can use the pair of **MPI_Isend/MPI_Wait** or **MPI_Irecv/MPI_Wait**

Multiple Completions

- It is sometimes desirable to wait on multiple requests:

```
MPI_Waitall(count, array_of_requests, array_of_statuses)
MPI_Waitany(count, array_of_requests, &index, &status)
MPI_Waitsome(incount, array_of_requests, outcount,
                array_of_indices, array_of_statuses)
```

- There are corresponding versions of **`TEST`** for each of these

Message Completion and Buffering

- For a communication to succeed:
  - Sender must specify a valid destination rank
  - Receiver must specify a valid source rank (including **MPI_ANY_SOURCE**)
  - The communicator must be the same
  - Tags must match
  - Receiver's buffer must be large enough
- A send has completed when the user supplied buffer can be reused

```
*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; /* Receiver may get 3, 4, or
             anything else */
MPI_Wait(…);
```

- 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

A Nonblocking 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...]
}
```

Section Summary

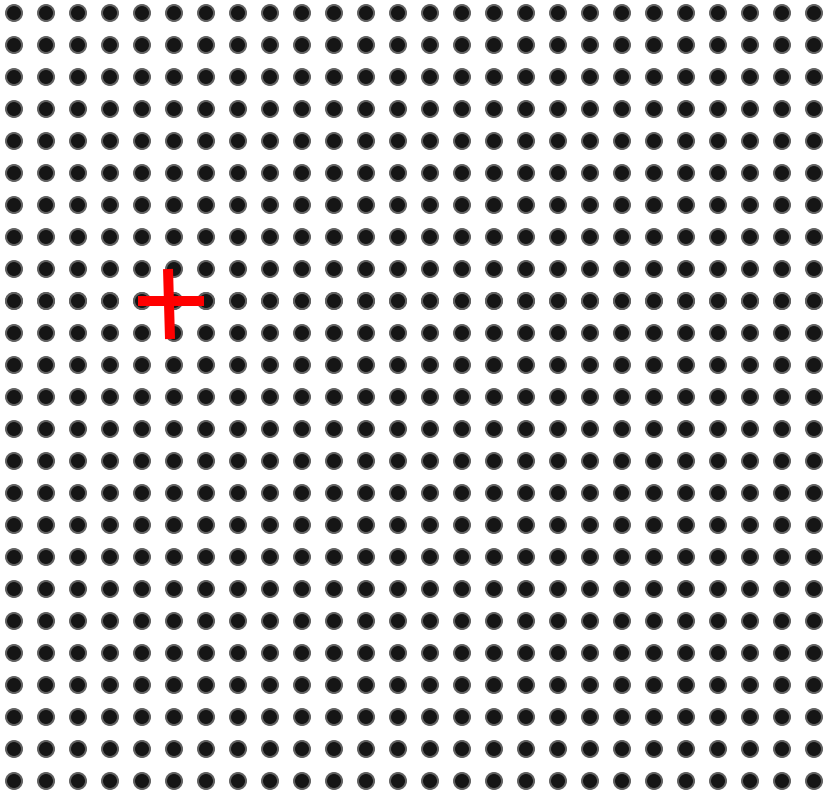
- Nonblocking communication is an enhancement over blocking communication
- Allows for computation and communication to be potentially overlapped
  - MPI implementation might, but is not guaranteed to overlap
  - Depends on what capabilities the network provides
  - Depends on how the MPI library is implemented (e.g., some libraries might tradeoff between better overlap and better basic performance)
- Critical for event-driven programming
  - Multiple outstanding operations, and the application performs a corresponding task depending on what completes next

Example Code: 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
  - *nonblocking_p2p/stencil.c*

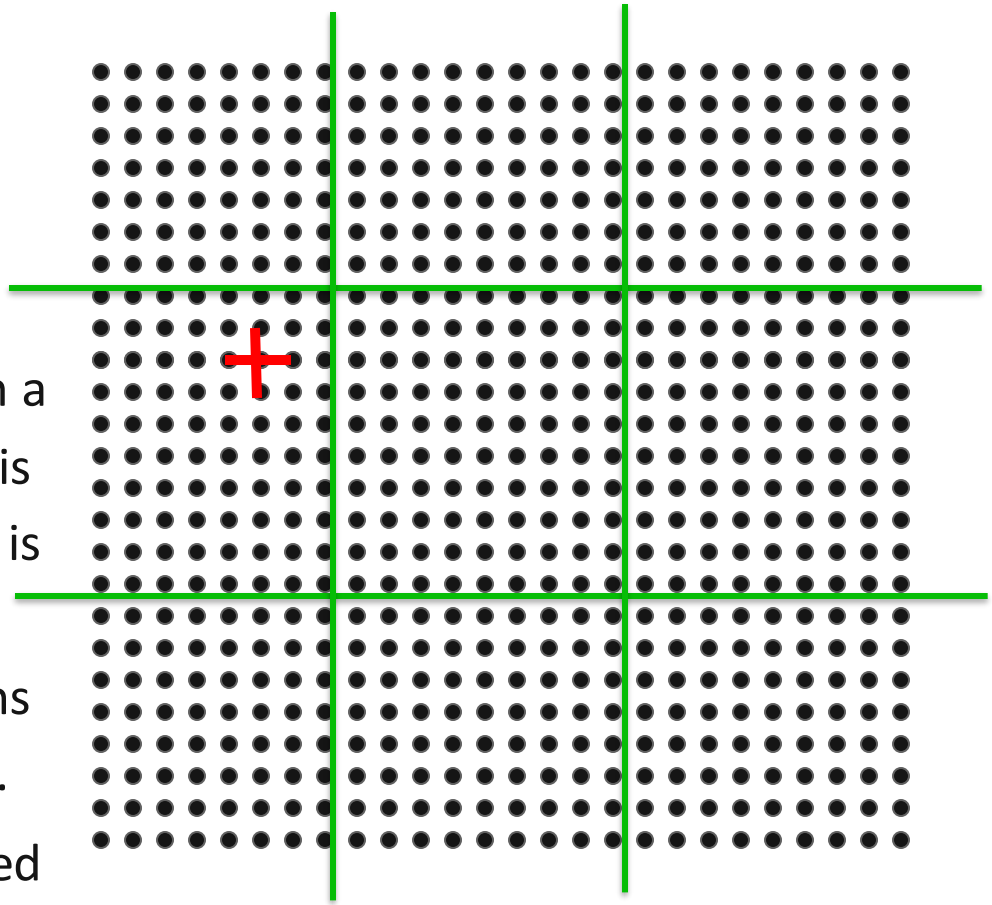
The Global Data Structure

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



The Global Data Structure

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



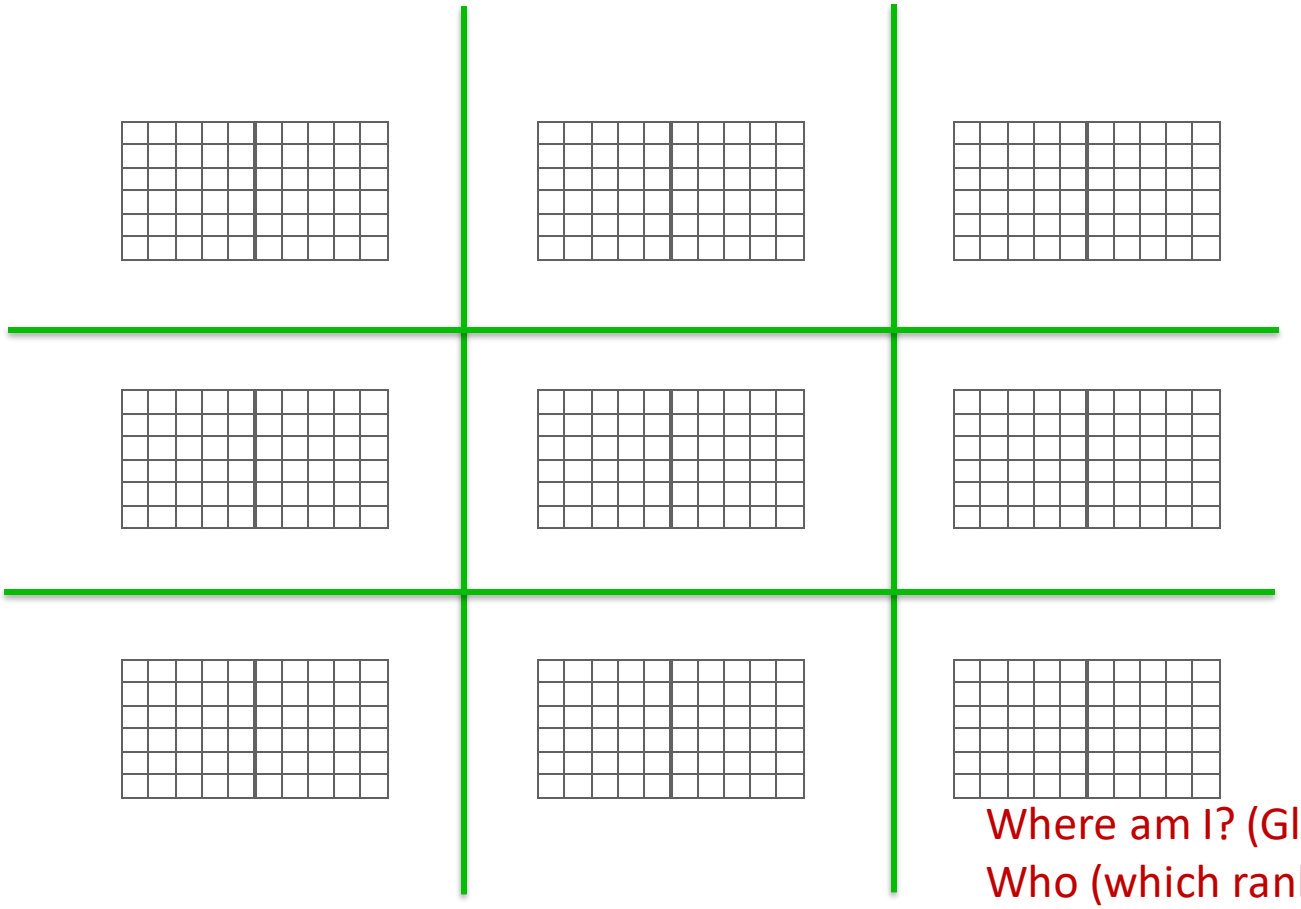
Step 1: Domain Decomposition

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

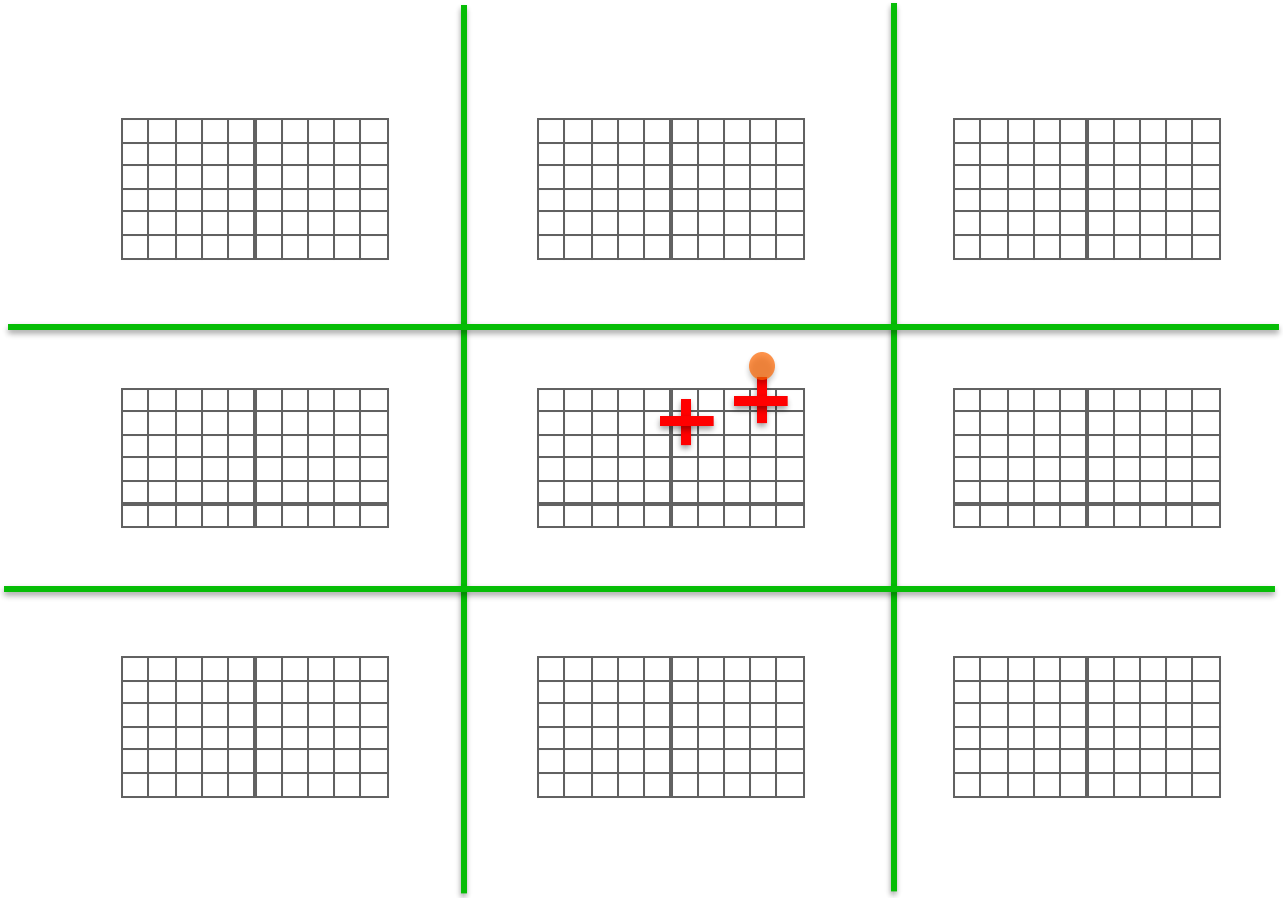


Where am I? (Global offset)

Who (which ranks) are my neighbors?

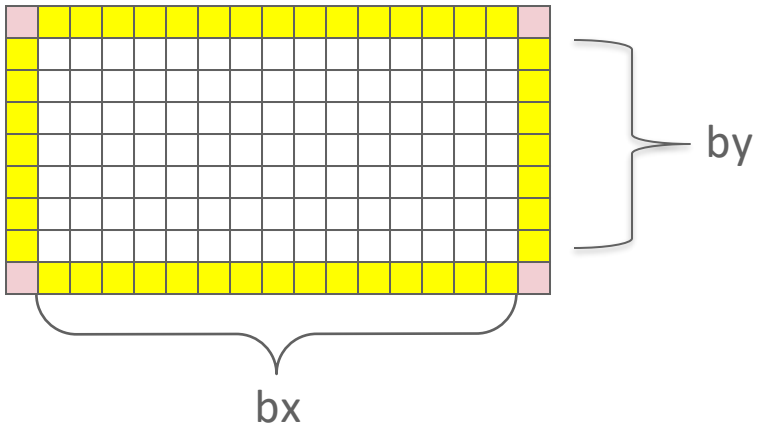
Use MPI_PROC_NULL for boundary

Necessary Data Transfers



Step 2: The Local Data Structure

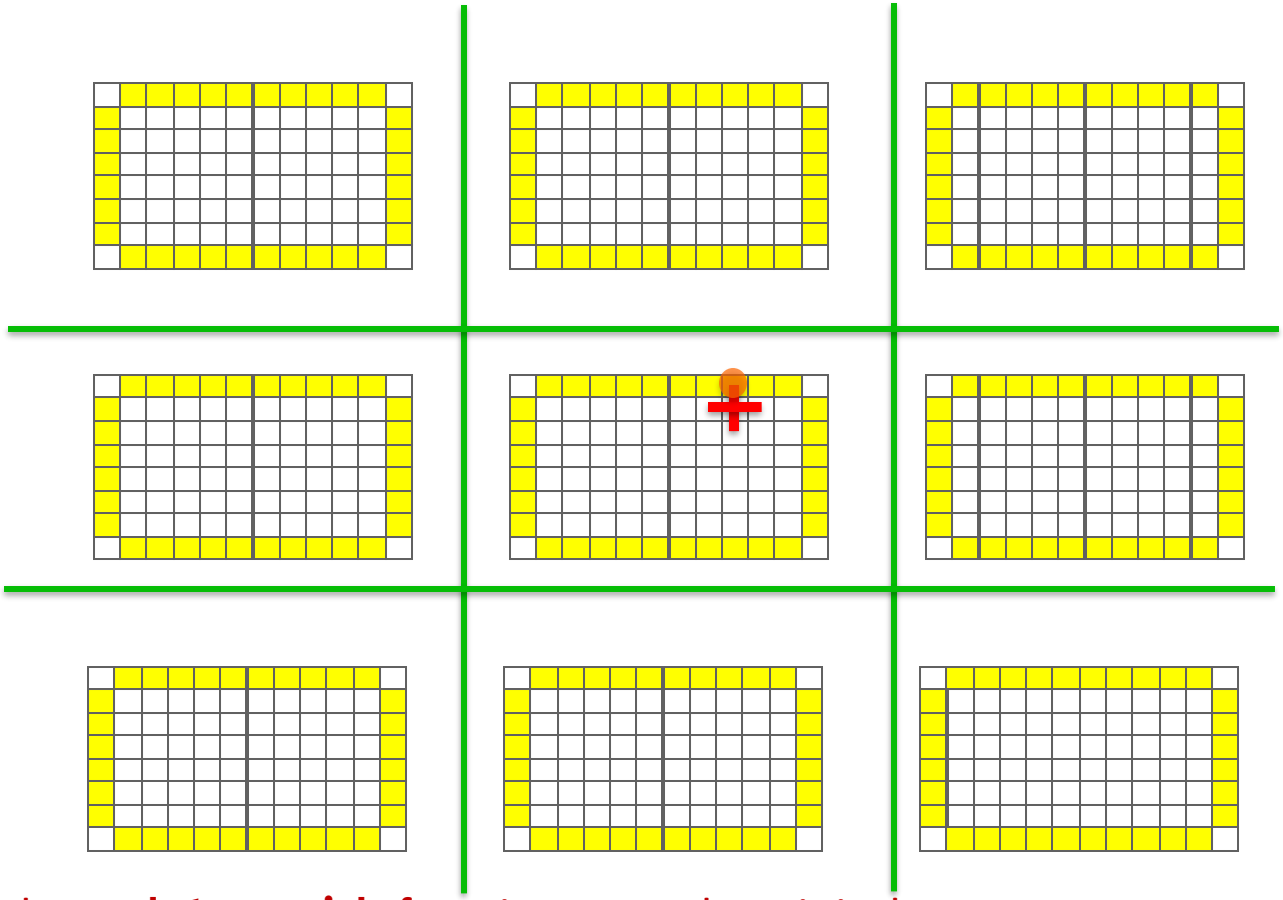
- Each process has its local “patch” of the global array
  - “bx” and “by” are the sizes of the local array
  - Always allocate a halo around the patch
  - Array allocated of size (bx+2)x(by+2)
- Each process also have send/recv buffers for each neighbor



Check the **`alloc_bufs`** function to see how buffers are allocated

Calculation

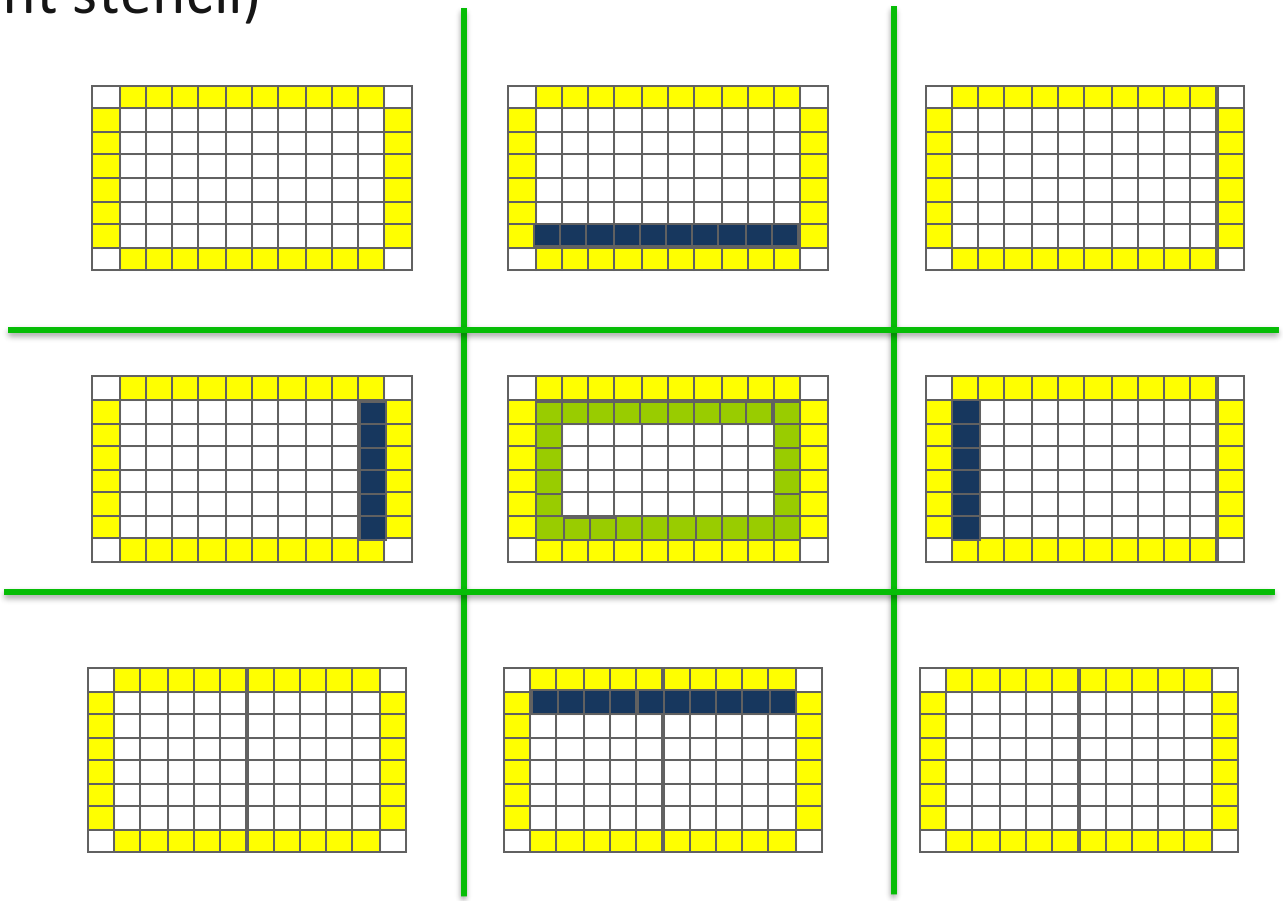
- Two buffers alternating
  - aold for current value
  - anew for newly value in this iteration (will become aold in next iter)



Check the **`update_grid`** function to see how it is done

Step 3: Data Transfers with MPI_Isend/MPI_Irecv

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



Note the differences in send/recv buffers, the requirement of data packing.

Step 3: Data Transfers with MPI_Isend/MPI_Irecv (cont'd)

- Data exchange with neighbors using corresponding send/recv buffers
- How to complete the communication? (MPI_Wait? MPI_Waitall?)
- Does order matters?

Step 4: Calculating Total Heat

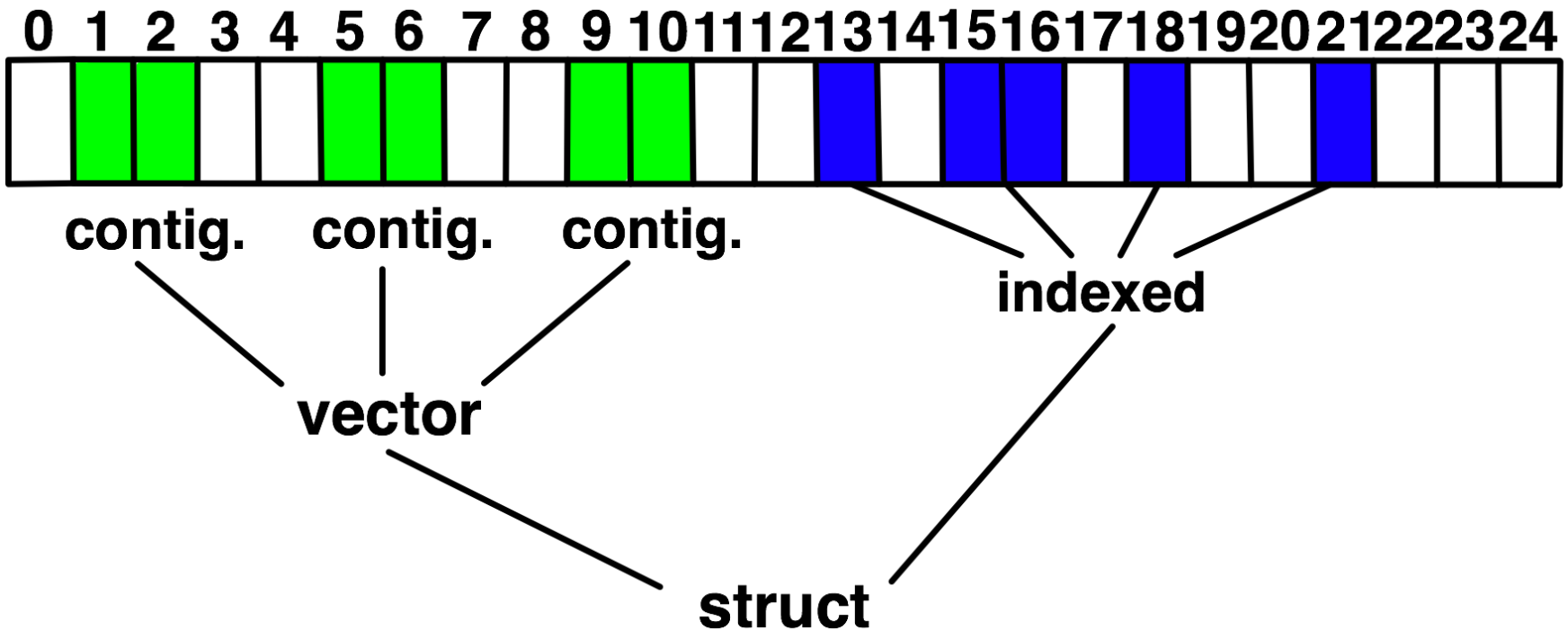
- Using MPI_Allreduce to calculate total heat

Datatypes

Introduction to Datatypes in MPI

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

Derived Datatype Example



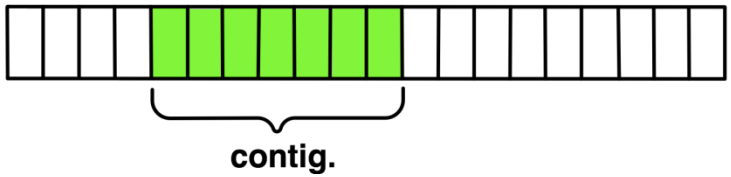
MPI's Intrinsic Datatypes

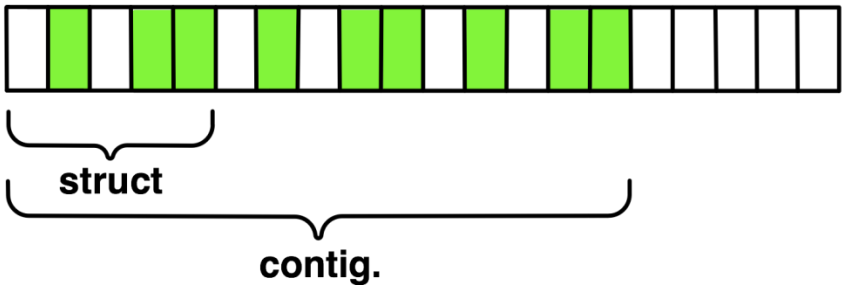
- Why intrinsic types?
  - Heterogeneity, nice to send a Boolean from C to Fortran
  - Conversion rules are complex, not discussed here
  - Length matches to language types
    - No sizeof(int) mess
- Users should generally use intrinsic types as basic types for communication and type construction!
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

MPI_Type_contiguous

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

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

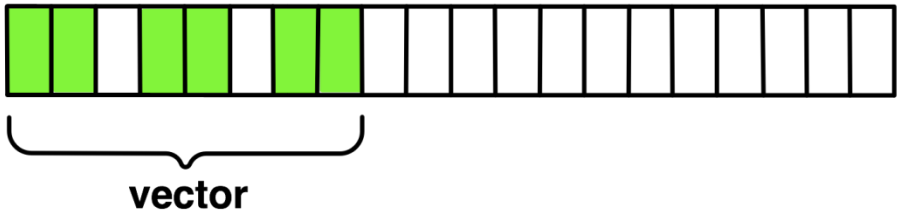


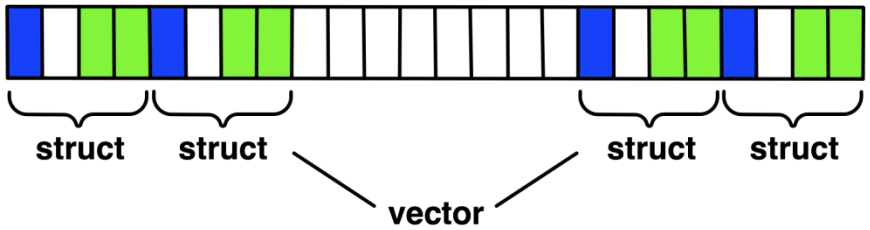


MPI_Type_vector

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

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

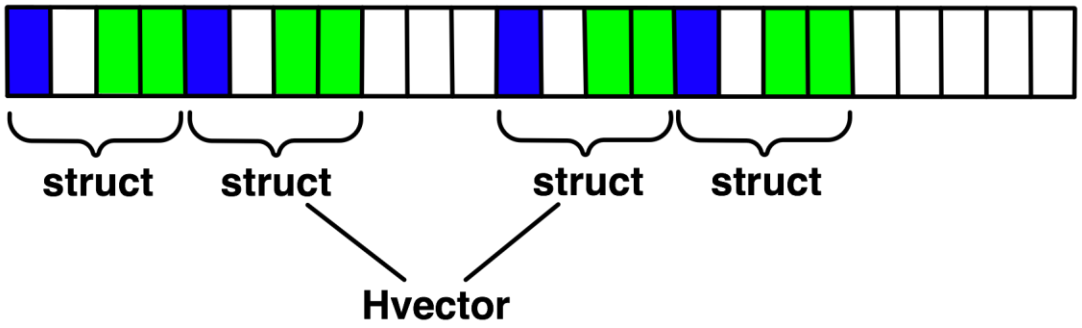




MPI_Type_create_hvector

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

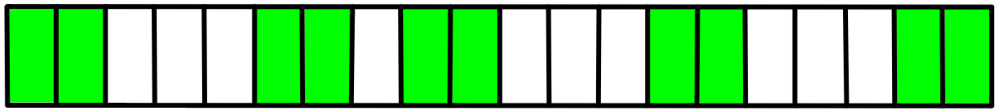
- Create non-unit strided vectors
- Useful for composition, e.g., vector of structs



MPI_Type_create_indexed_block

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

- Like MPI_Type_indexed but blocklength is the same

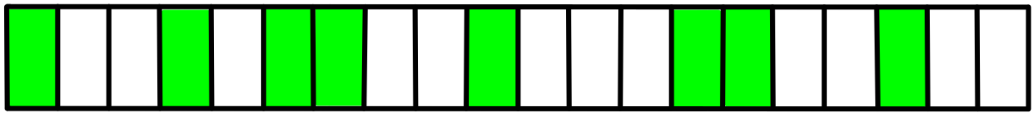


  - blen=2
  - displs={0,5,9,13,18}

MPI_Type_indexed

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

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

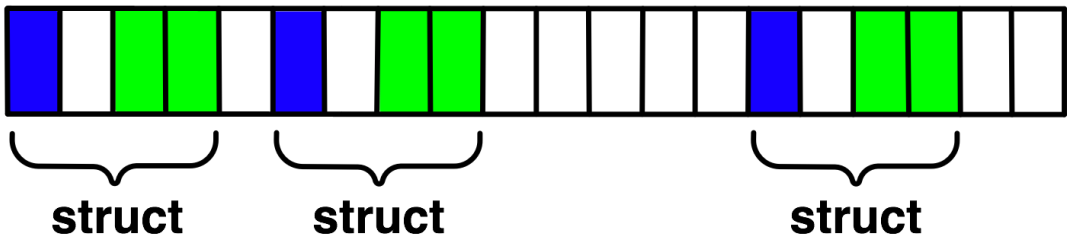


  - blen={1,1,2,1,2,1}
  - displs={0,3,5,9,13,17}

MPI_Type_create_hindexed

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

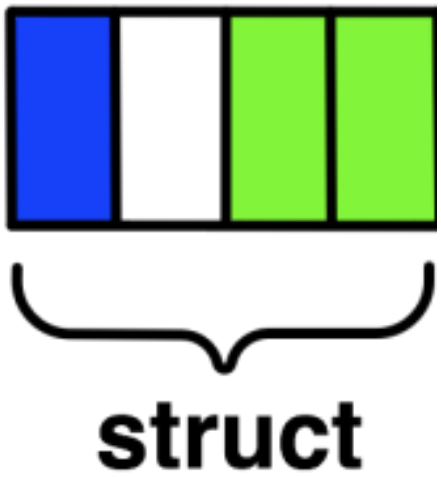
- Indexed with non-unit displacements, e.g., pulling types out of different arrays



MPI_Type_create_struct

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

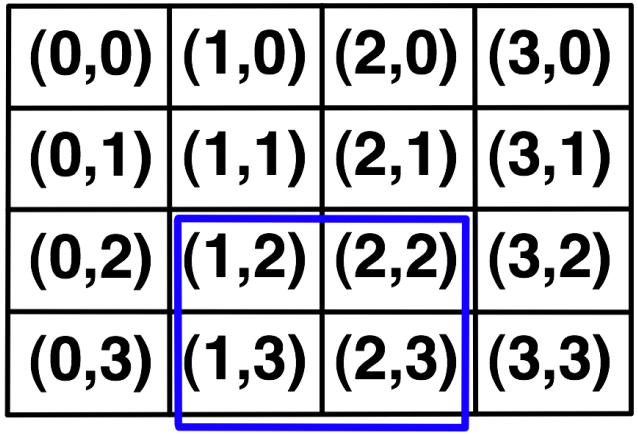
- Most general constructor, allows different types and arbitrary arrays (also most costly)



MPI_Type_create_subarray

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

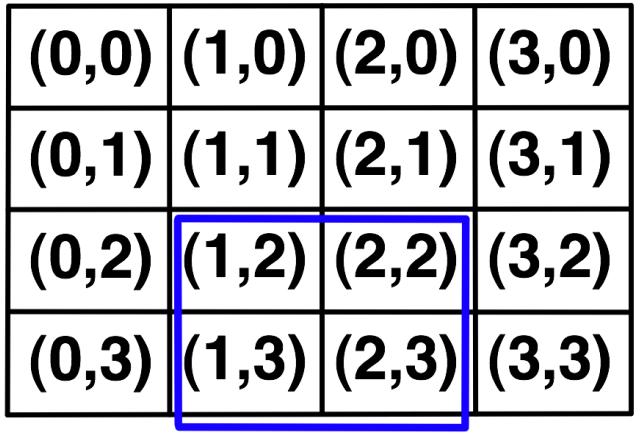
- Specify subarray of n-dimensional array (sizes) by start (starts) and size (subsize)



MPI_Type_create_darray

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

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



Commit, Free, and Dup

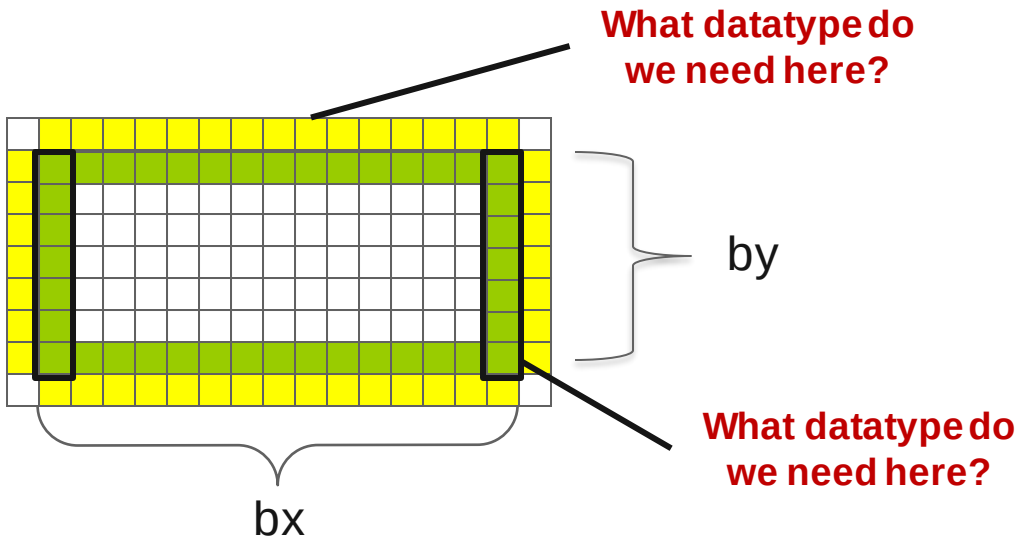
- Types must be committed before use
  - Only the ones that are used explicitly in a call!
  - MPI_Type_commit may perform time-consuming optimizations (but few implementations currently exploit this feature)
- MPI_Type_free
  - Free MPI resources of datatypes
  - Does not affect types built from it
- MPI_Type_dup
  - Duplicates a type
  - Library abstraction (composability)

Datatype Performance in Practice

- Datatypes *can* provide performance benefits, particularly for certain regular patterns
  - However, many implementations do not optimize datatype operations
  - If performance is critical, you will need to test
    - Even manual packing/unpacking can be slow if not properly optimized by the compiler – make sure to check optimization reports or if the compiler doesn't provide good reports, inspect the assembly code
- For parallel I/O, datatypes *do* provide large performance benefits in many cases

Exercise: Stencil with Derived Datatypes (1)

- 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



Exercise: Stencil with Derived Datatypes (2)

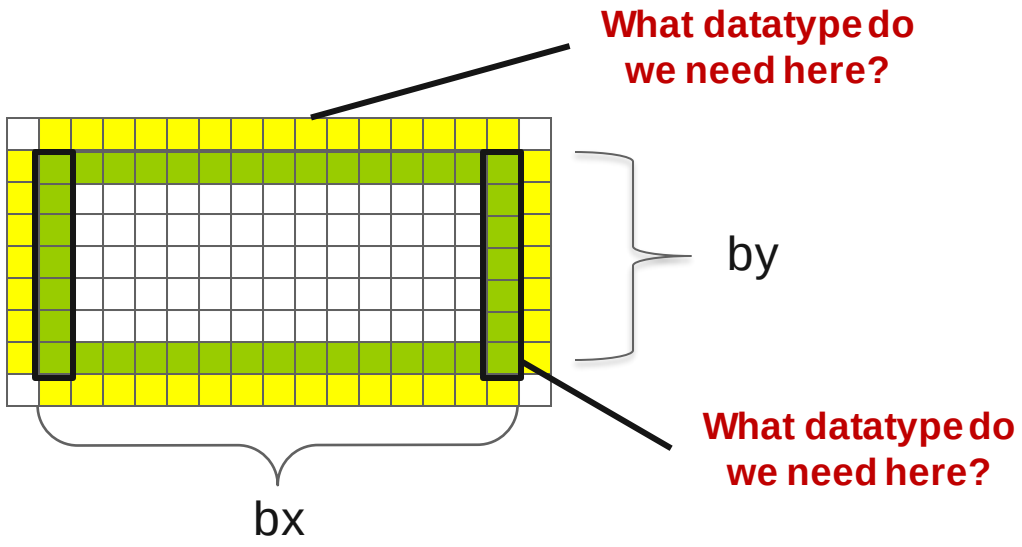
- Nonblocking sends and receives
- Data location specified by MPI datatypes
- Manual packing of data no longer required
- *Start from nonblocking_p2p/stencil.c*
- *Solution in derived_datatype/stencil.c*

Example: Basic Send/Receive

- *blocking_p2p/sendrecv.c*
- Simple send/receive program to show basic data transfer

Exercise: Stencil with Derived Datatypes (1)

- 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



Exercise: Stencil with Derived Datatypes (2)

- Nonblocking sends and receives
- Data location specified by MPI datatypes
- Manual packing of data no longer required
- *Start from nonblocking_p2p/stencil.c*
- *Solution in derived_datatype/stencil.c*

Collectives and Nonblocking Collectives

Introduction to Collective Operations in MPI

- Called by all processes in a communicator
- Share similar syntax and semantics with pt2pt operations but have no tags and have stricter type matching rules
- May or may not have synchronization effects on the calling processes
- Three classes of collective operations: *synchronization, data movement, collective computation*
- Communication and computation is coordinated among a group of processes in a communicator
- Non-blocking collective operations have been introduced in MPI-3

MPI Collective Communication Semantics

```
MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
          int root, MPI_Comm comm)
```

- Distributes data from one process (the root) to all others in a communicator

```
MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
           MPI_Datatype datatype, MPI_Op op, int root,
           MPI_Comm comm)
```

- Combines data from all processes in the communicator and returns it to one process
- *In many numerical algorithms, SEND/RECV can be replaced by BCAST/REDUCE, improving both simplicity and efficiency*

Synchronization

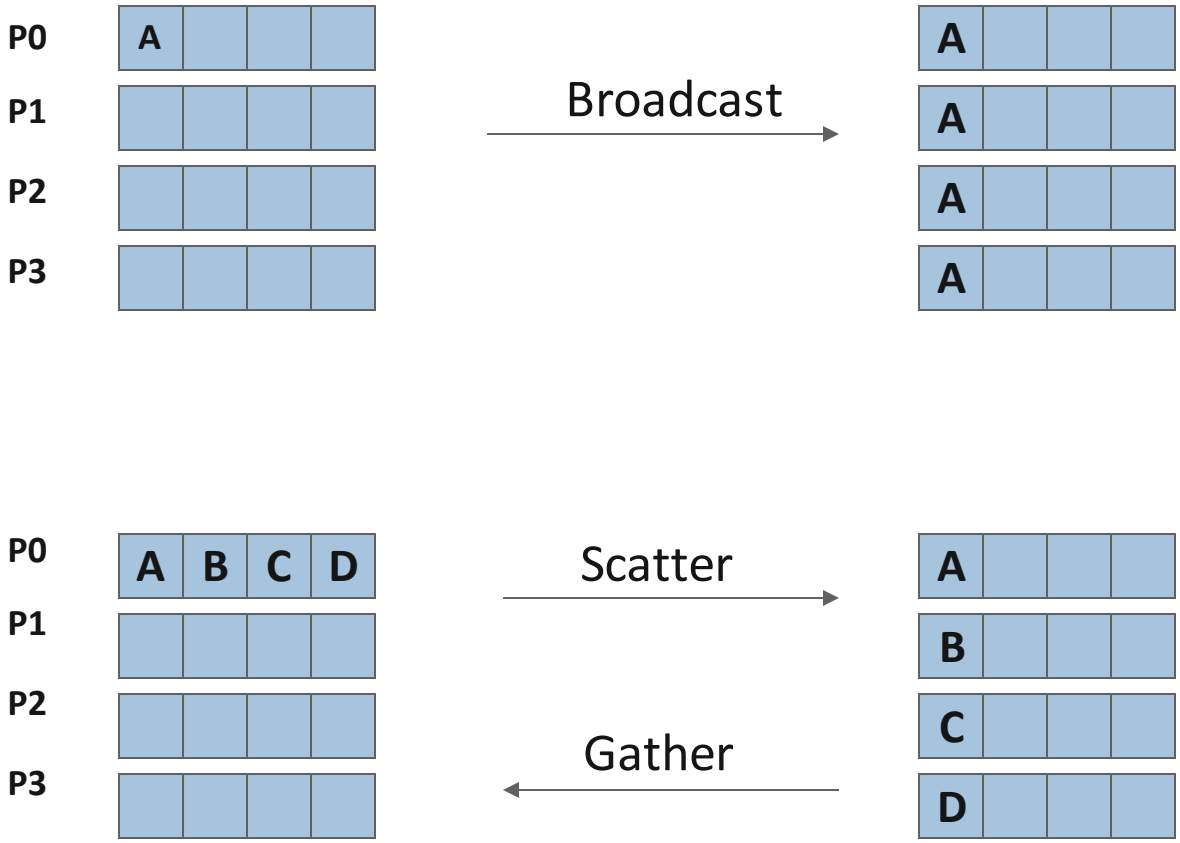
```
MPI_Barrier(MPI_Comm comm)
```

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

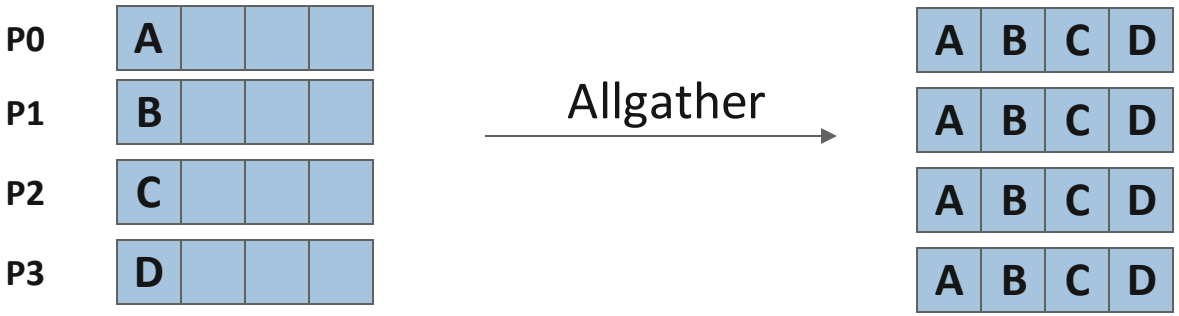
- Note that a barrier is rarely, if ever, necessary in an MPI program
- Adding barriers "just to be sure" is a bad practice and causes unnecessary synchronization. Remove unnecessary barriers from your code

- One legitimate use of a barrier is before the first call to MPI_Wtime to start a timing measurement. This causes each process to start at *approximately* the same time
- Avoid using barriers other than for this

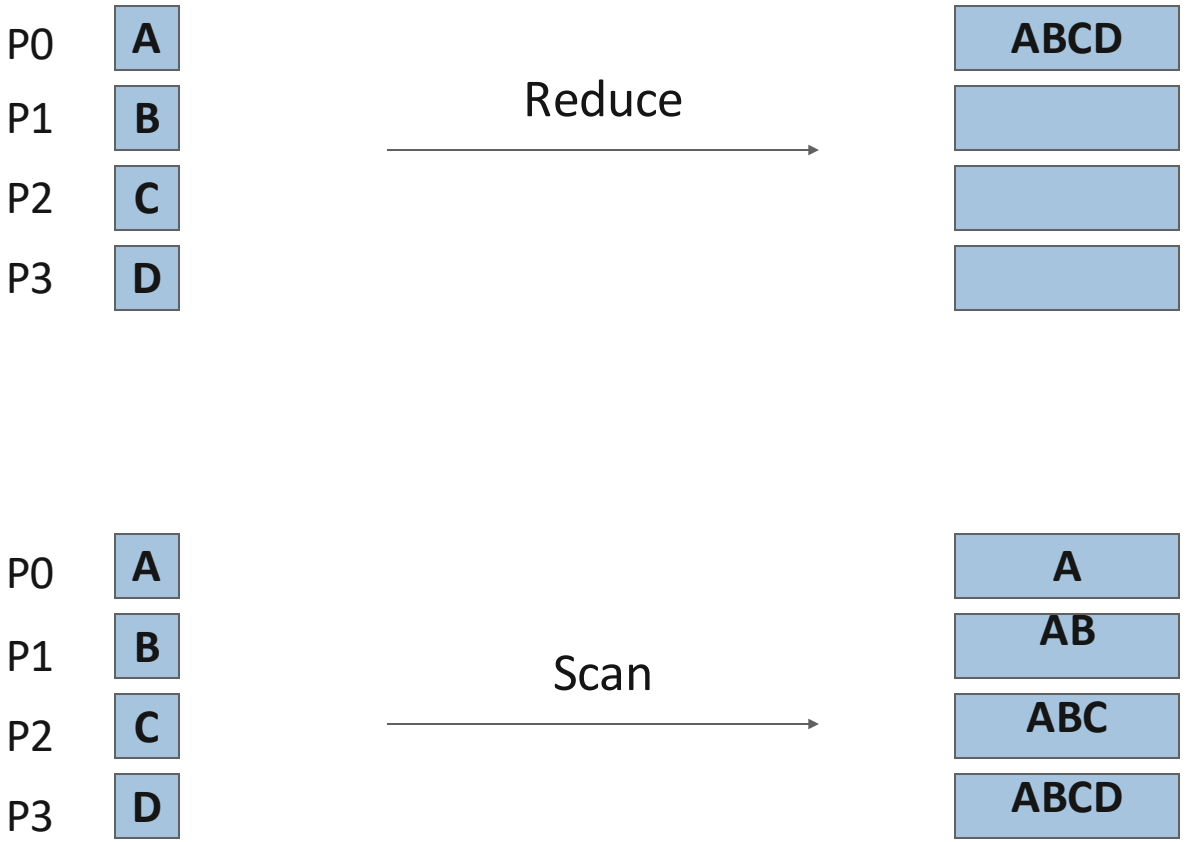
Collective Data Movement



More Collective Data Movement



Collective Computation



MPI Collective Routines

- Many Routines, including: **`MPI_ALLGATHER`**, **`MPI_ALLGATHERV`**, **`MPI_ALLREDUCE`**, **`MPI_ALLTOALL`**, **`MPI_ALLTOALLV`**, **`MPI_BCAST`**, **`MPI_EXSCAN`**, **`MPI_GATHER`**, **`MPI_GATHERV`**, **`MPI_REDUCE`**, **`MPI_REDUCE_SCATTER`**, **`MPI_SCAN`**, **`MPI_SCATTER`**, **`MPI_SCATTERV`**
- "**`All`**" versions deliver results to all participating processes
- "**`V`**" versions (stands for vector) allow the chunks to have different sizes
- "**`W`**" versions for ALLTOALL allow the chunks to have different sizes in bytes, rather than units of datatypes
- **`MPI_ALLREDUCE`**, **`MPI_REDUCE`**, **`MPI_REDUCE_SCATTER`**, **`MPI_REDUCE_SCATTER_BLOCK`**, **`MPI_EXSCAN`**, 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
MPI_REPLACE, MPI_NO_OP	Replace and no operation (RMA)

Defining your own Collective Operations

- Create your own collective computations with:

```
MPI_OP_CREATE(user_fn, commutes, &op);
MPI_OP_FREE(&op);

user_fn(invec, inoutvec, len, datatype);
```

- The user function should perform:

```
inoutvec[i] = invec[i]  op  inoutvec[i];
```

for i from 0 to len-1

- The user function can be non-commutative, but must be associative

Nonblocking Collectives

Nonblocking Collective Communication

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

Nonblocking Communication

- Semantics are simple:
  - Function returns no matter what
  - No progress guarantee!
- E.g., MPI_Isend(<send-args>, MPI_Request *req);
- Nonblocking tests:
  - Test, Testany, Testall, Testsome
- Blocking wait:
  - Wait, Waitany, Waitall, Waitsome

Nonblocking Collective Communication

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

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

Nonblocking Collective Communication

- Semantic advantages:
  - Enable asynchronous progression (and manual)
    - Software 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

A Non-Blocking Barrier?

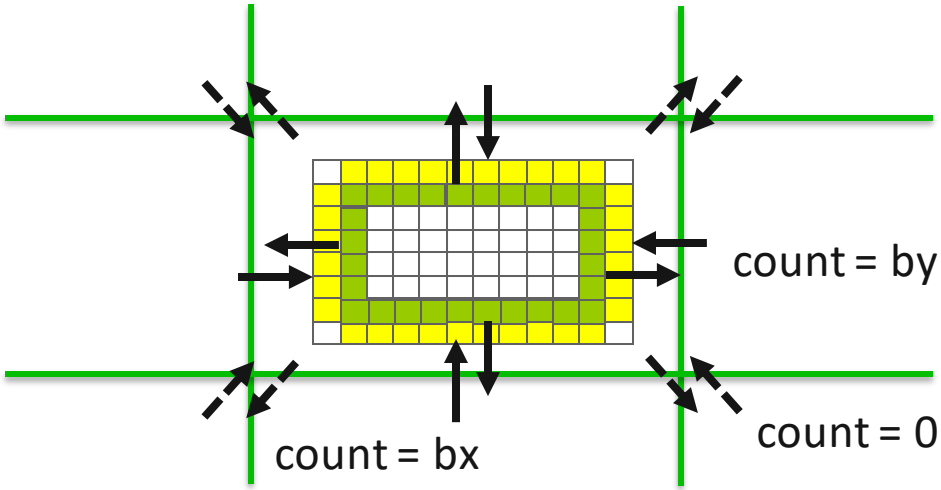
- What can that be good for? Well, quite a bit!
- Semantics:
  - MPI_Ibarrier() – calling process entered the barrier, **no** synchronization happens
  - Synchronization **may** happen asynchronously
  - MPI_Test/Wait() – synchronization happens **if** necessary
- Uses:
  - Overlap barrier latency (small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

Nonblocking And Collective Summary

- Nonblocking communication
  - Overlap and relax synchronization
- Collective communication
  - Specialized pre-optimized routines
  - Performance portability
  - Hopefully transparent performance
- They can be composed
  - E.g., software pipelining

Exercise: Stencil using Alltoallv

- In the basic version of the stencil code
  - Used nonblocking send/receive for each direction
- Let's try to use single alltoallv collective call
- *Start from nonblocking_p2p/stencil.c*
- *Solution available in blocking_coll/stencil_alltoallv.c*

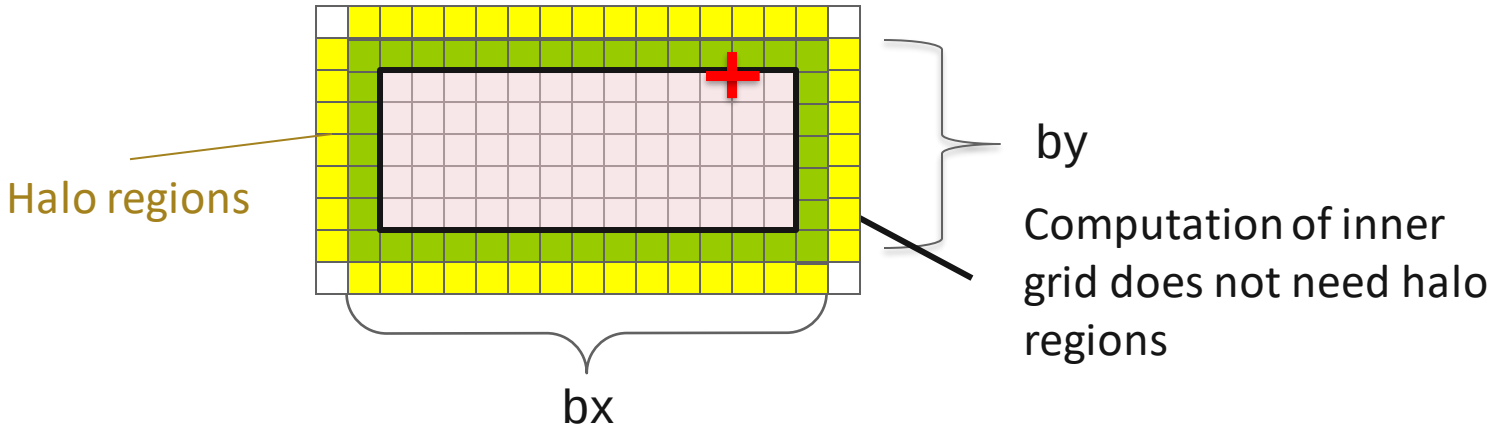


Exercise: Stencil with Derived Datatypes and Collectives

- Simplify collective version of stencil
  - Alltoallv: defines a set of counts and displacements with the same datatype (see *blocking_coll/stencil_alltoallv.c*)
  - Alltoallw: defines a set of counts, displacements, and datatypes
- Data location specified by MPI datatypes
- Manual packing of data no longer required
- *Start from blocking_coll/stencil_alltoallv.c*
- *Solution in derived_datatype/stencil_alltoallw.c*

Exercise: Stencil with Nonblocking Collectives

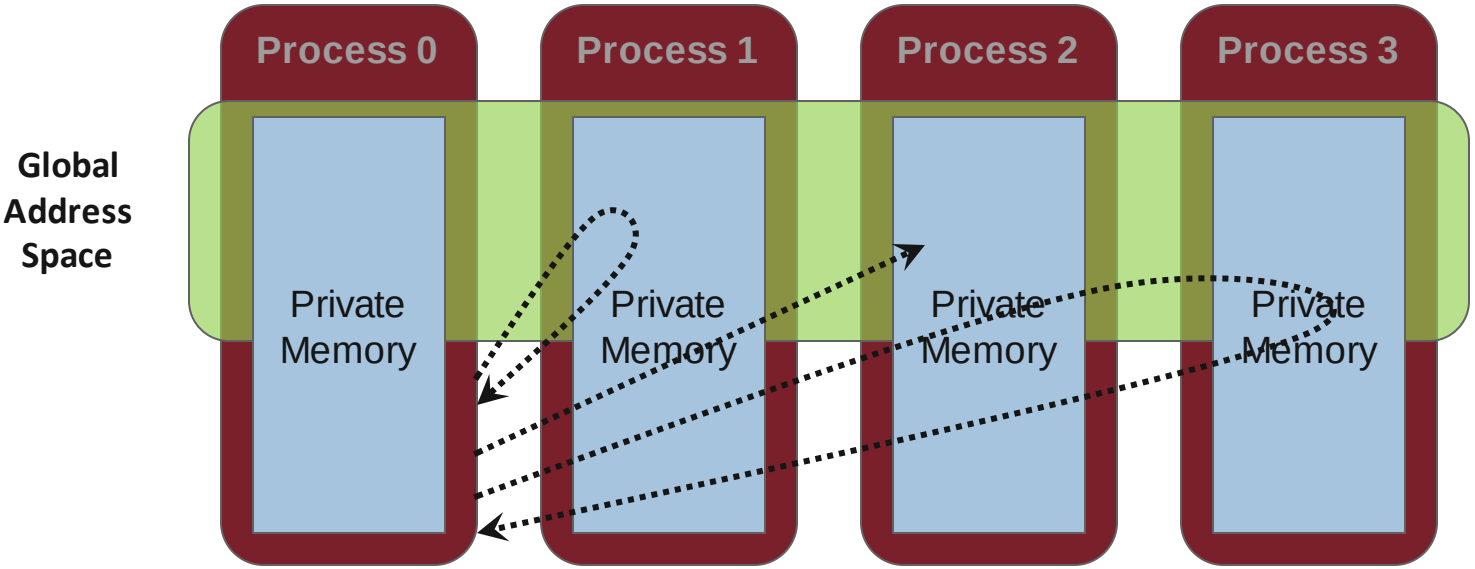
- Use nonblocking collective to overlap computation and communication
  - Compute inner grid while waiting for completion of data movement
  - Compute outer grid with updated halo regions
- *Start from derived_datatype/stencil_alltoallw.c*
- *Solution available in nonblocking_coll/stencil_alltoallw.c*



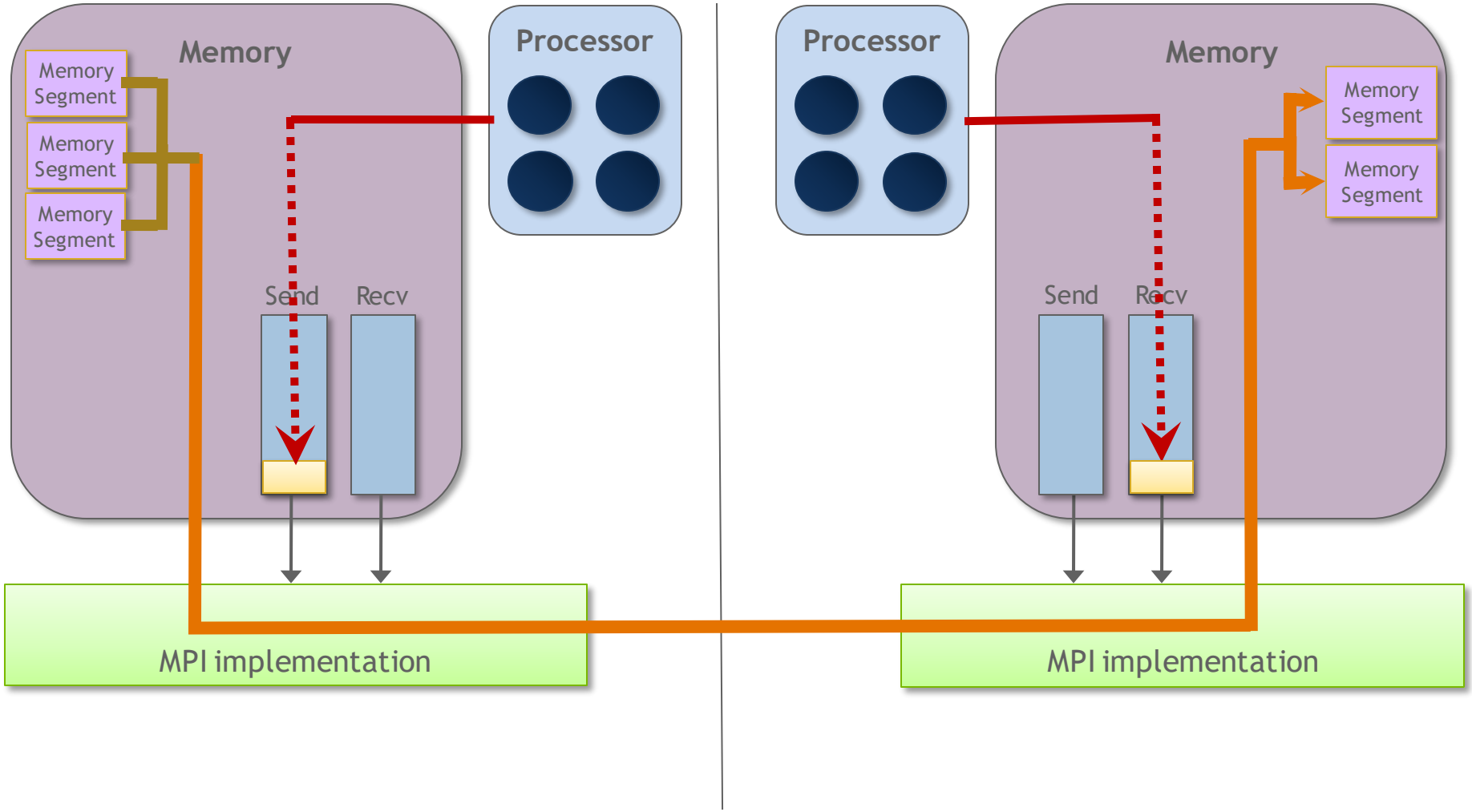
Advanced Topics: One-sided Communication

One-sided Communication

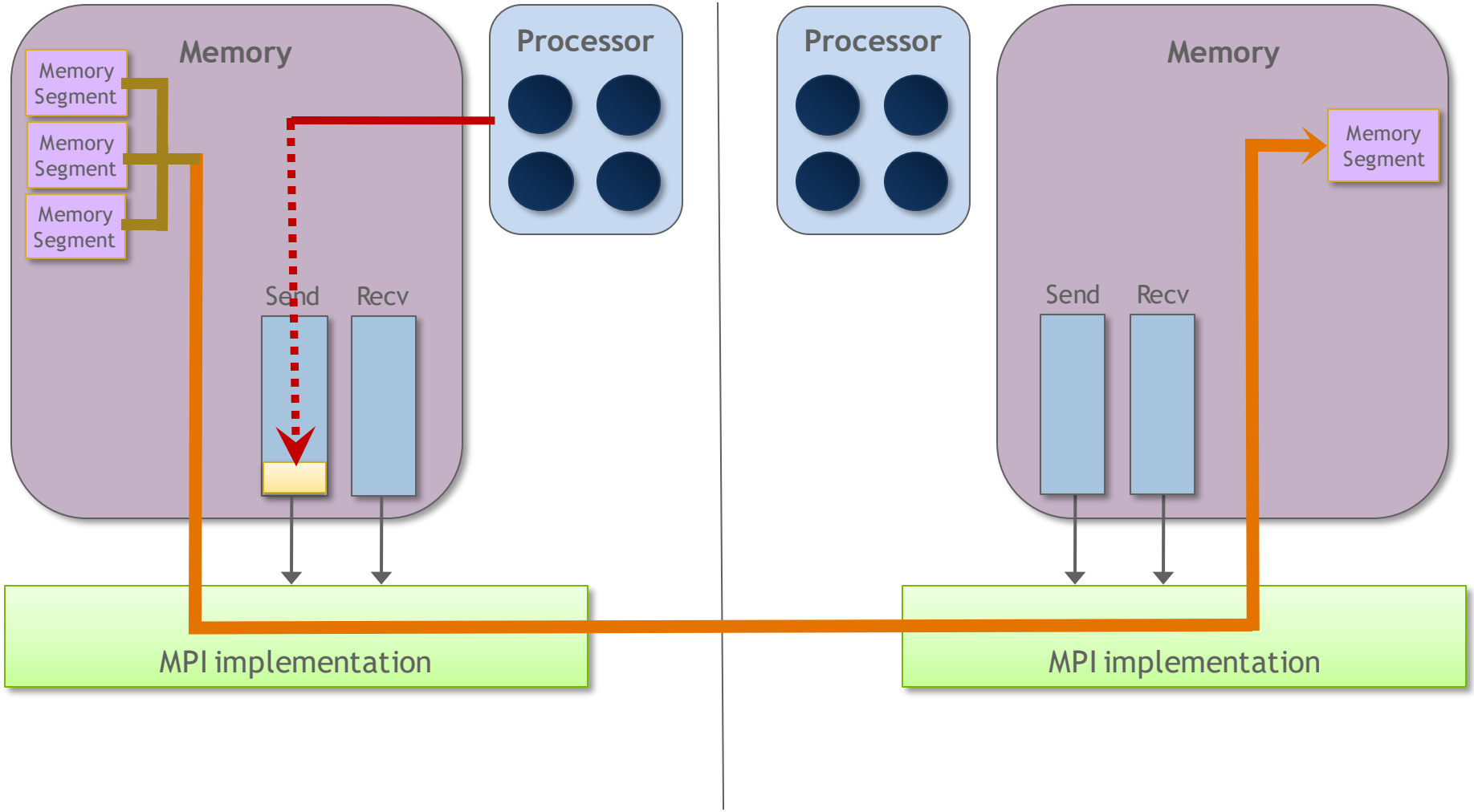
- The basic idea of one-sided communication models is to decouple data movement with process synchronization
  - Should be able to move data without requiring that the remote process synchronize
  - Each process exposes a part of its memory to other processes
  - Other processes can directly read from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



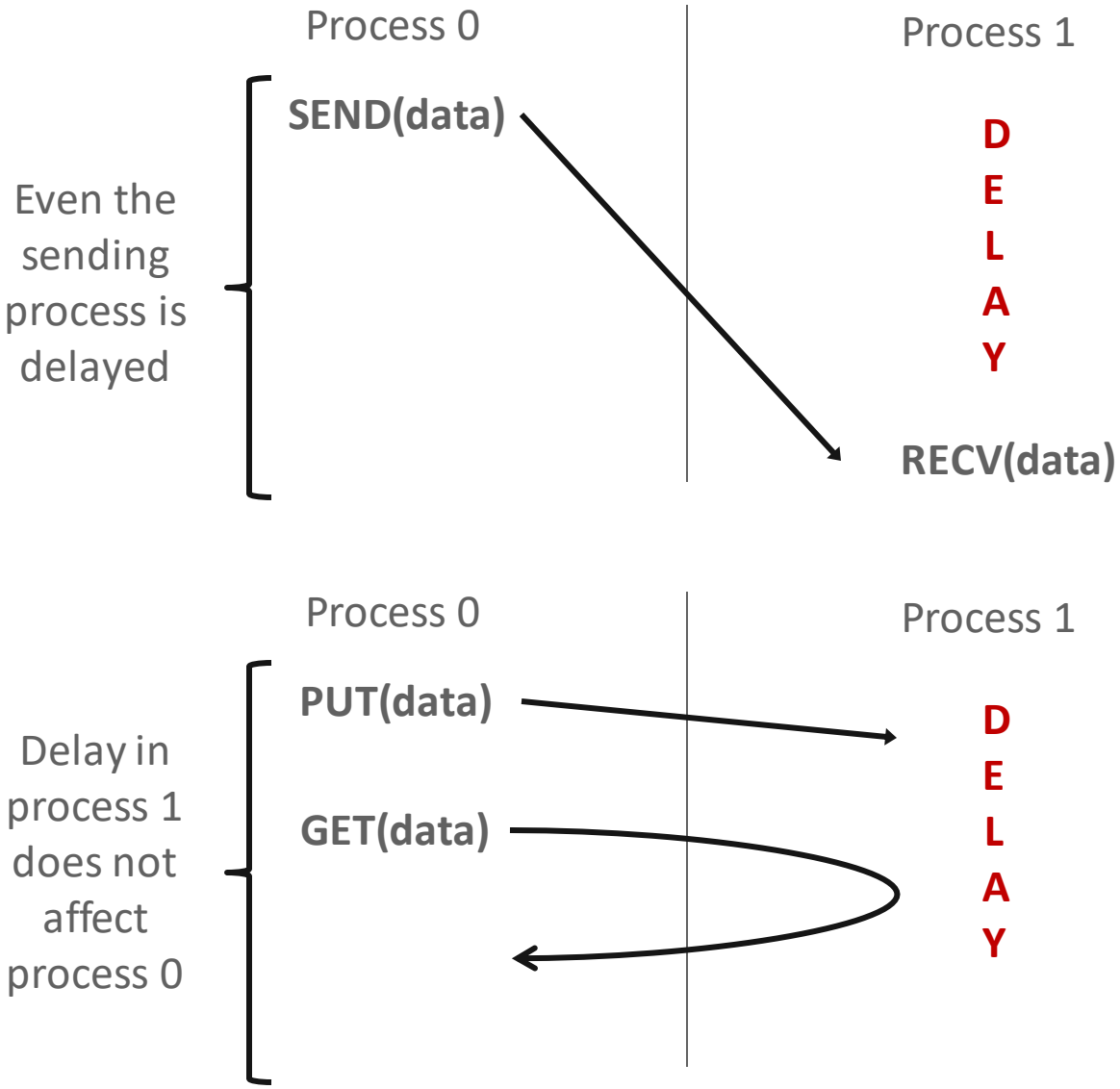
Two-sided Communication Example



One-sided Communication Example



Comparing One-sided and Two-sided Programming

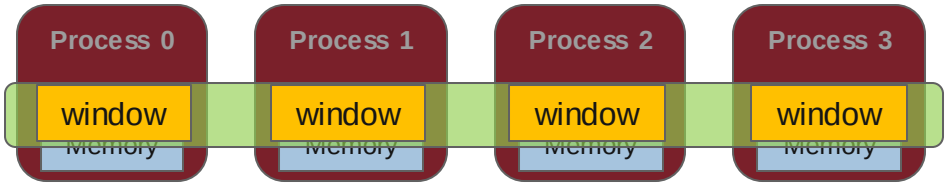


What we need to know in MPI RMA

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

Creating Public Memory

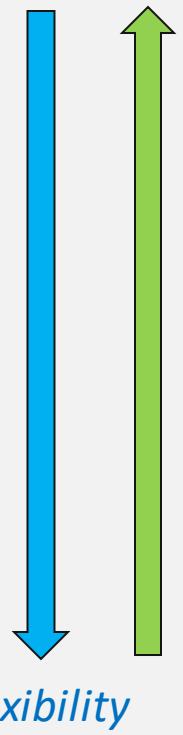
- Any memory used by a process is, by default, only locally accessible
  - X = malloc(100);



- 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;
}
```

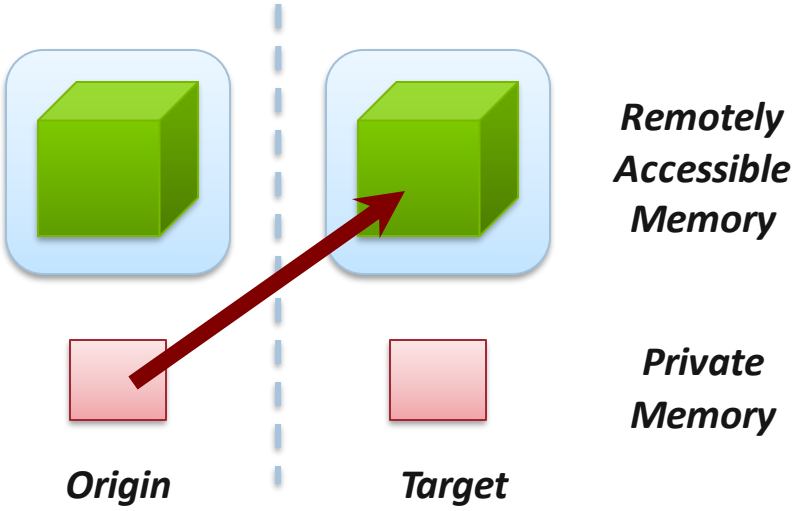
Data movement

- 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(const void *origin_addr, int origin_count,
        MPI_Datatype origin_dtype, int target_rank,
        MPI_Aint target_disp, int target_count,
        MPI_Datatype target_dtype, MPI_Win win)
```

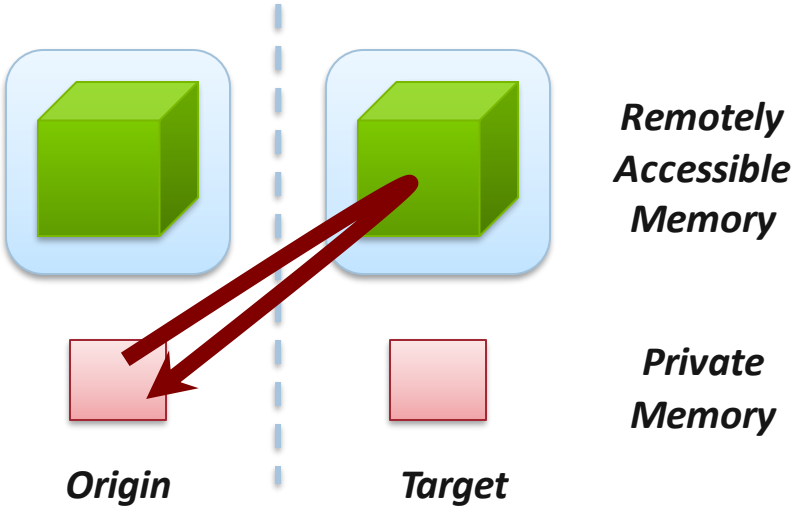
- Move data from origin, to target
- Separate data description triples for **origin** and **target**



Data movement: *Get*

```
MPI_Get(void *origin_addr, int origin_count,
        MPI_Datatype origin_dtype, int target_rank,
        MPI_Aint target_disp, int target_count,
        MPI_Datatype target_dtype, MPI_Win win)
```

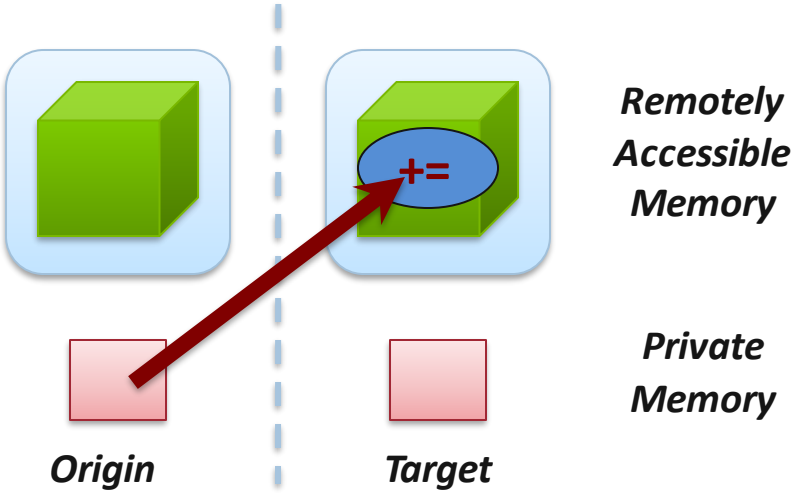
- Move data to origin, from target
- Separate data description triples for **origin** and **target**



Atomic Data Aggregation: *Accumulate*

```
MPI_Accumulate(const void *origin_addr, int origin_count,
        MPI_Datatype origin_dtype, int target_rank,
        MPI_Aint target_disp, int target_count,
        MPI_Datatype target_dtype, MPI_Op op, MPI_Win win)
```

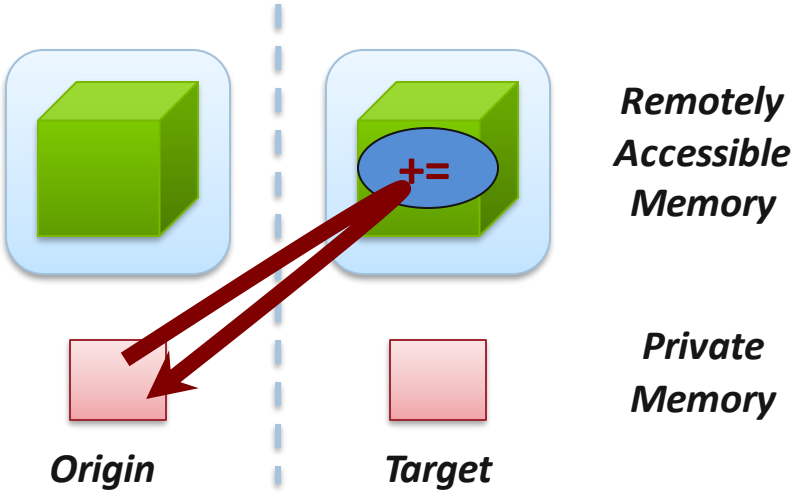
- Atomic update operation, similar to a put
  - Reduces origin and target data into target buffer using op argument as combiner
  - Op = MPI_SUM, MPI_PROD, MPI_OR, MPI_REPLACE, MPI_NO_OP, …
  - Predefined ops only, no user-defined operations
- Different data layouts between target/origin OK
  - Basic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



Atomic Data Aggregation: *Get Accumulate*

```
MPI_Get_accumulate(const void *origin_addr,
        int origin_count, MPI_Datatype origin_dtype,
        void *result_addr,int result_count,
        MPI_Datatype result_dtype, int target_rank,
        MPI_Aint target_disp,int target_count,
        MPI_Datatype target_dype, MPI_Op op, MPI_Win win)
```

- Atomic read-modify-write
  - Op = MPI_SUM, MPI_PROD, MPI_OR, MPI_REPLACE, MPI_NO_OP, …
  - Predefined ops only
- Result stored in target buffer
- Original data stored in result buf
- Different data layouts between target/origin OK
  - Basic type elements must match
- Atomic get with MPI_NO_OP
- Atomic swap with MPI_REPLACE



Atomic Data Aggregation: *CAS and FOP*

```
MPI_Fetch_and_op(const void *origin_addr, void *result_addr,
        MPI_Datatype dtype, int target_rank,
        MPI_Aint target_disp, MPI_Op op, MPI_Win win)
```

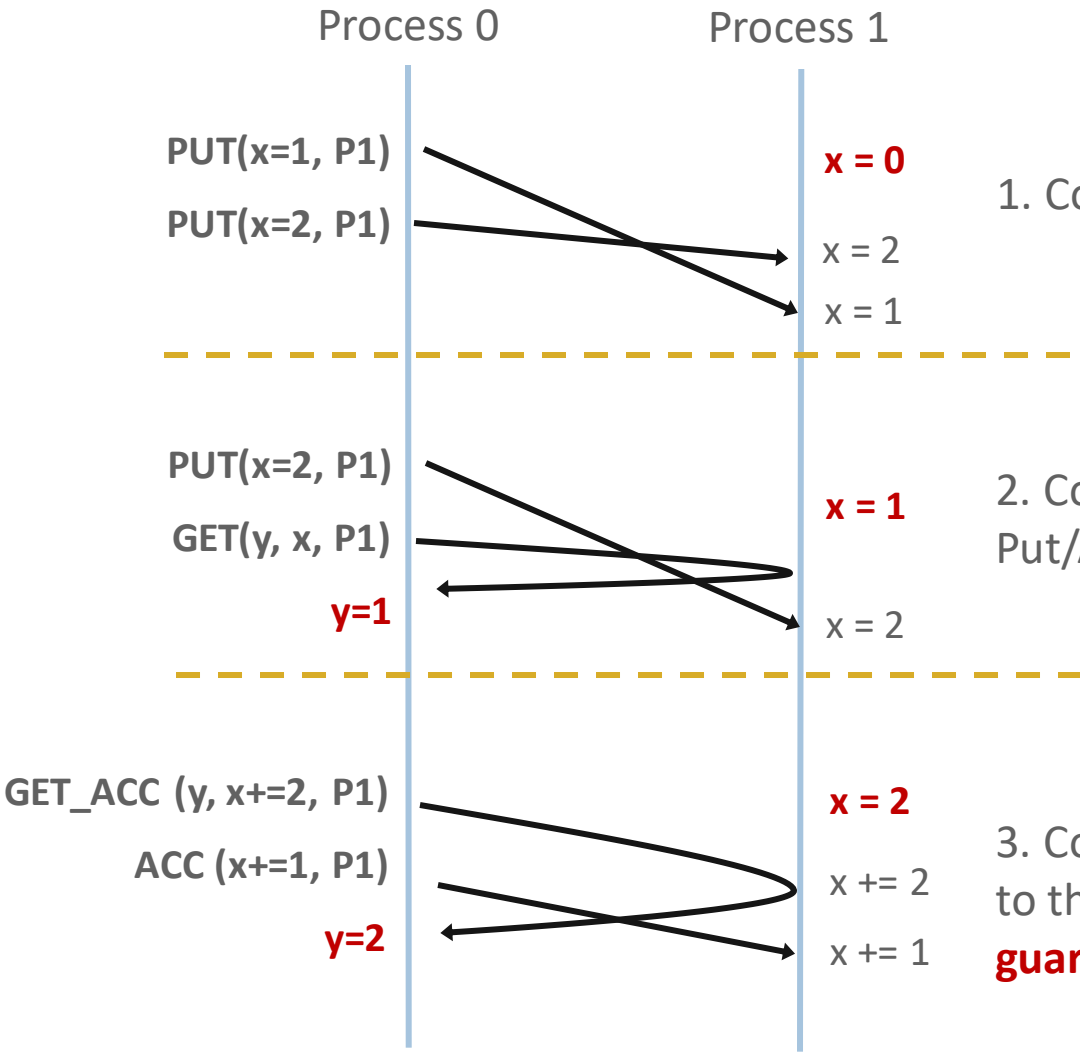
```
MPI_Compare_and_swap(const void *origin_addr,
        const void *compare_addr, void *result_addr,
        MPI_Datatype dtype, int target_rank,
        MPI_Aint target_disp, MPI_Win win)
```

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

Ordering of Operations in MPI RMA

- No guaranteed ordering for Put/Get operations
- Result of concurrent Puts to the same location undefined
- Result of Get concurrent Put/Accumulate undefined
  - Can be garbage in both cases
- Result of concurrent accumulate operations to the same location are defined according to the order in which the occurred
  - Atomic put: Accumulate with op = MPI_REPLACE
  - Atomic get: Get_accumulate with op = MPI_NO_OP
- Accumulate operations from a given process are ordered by default
  - User can tell the MPI implementation that (s)he does not require ordering as optimization hint
  - You can ask for only the needed orderings: RAW (read-after-write), WAR, RAR, or WAW

Examples with operation ordering



1. Concurrent Puts: **undefined**

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

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

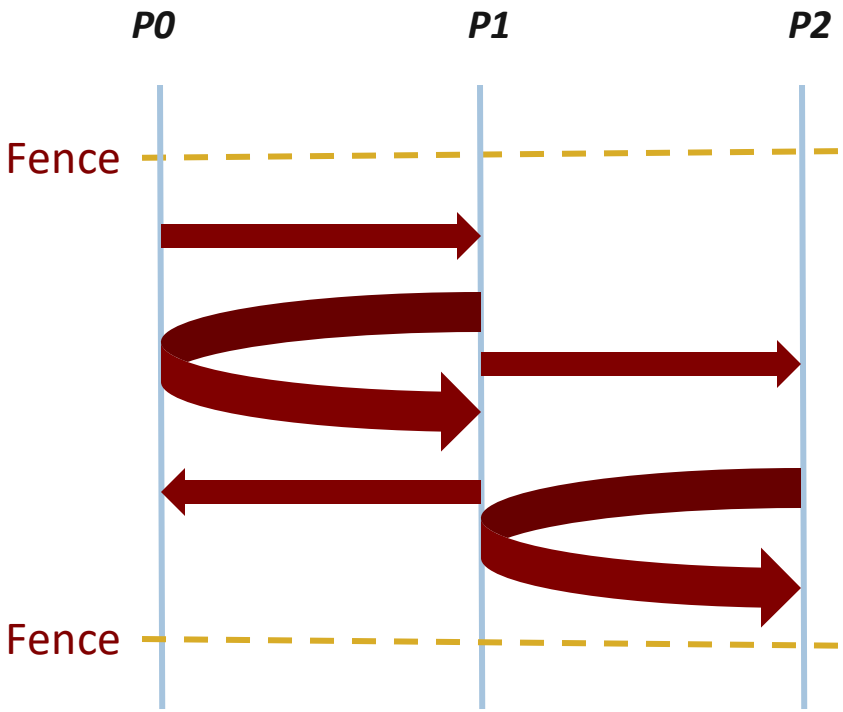
RMA Synchronization Models

- RMA data access model
  - When is a process allowed to read/write remotely accessible memory?
  - When is data written by process X is available for process Y to read?
  - RMA synchronization models define these semantics
- Three synchronization models provided by MPI:
  - Fence (active target)
  - Post-start-complete-wait (generalized active target; rarely used now)
  - 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

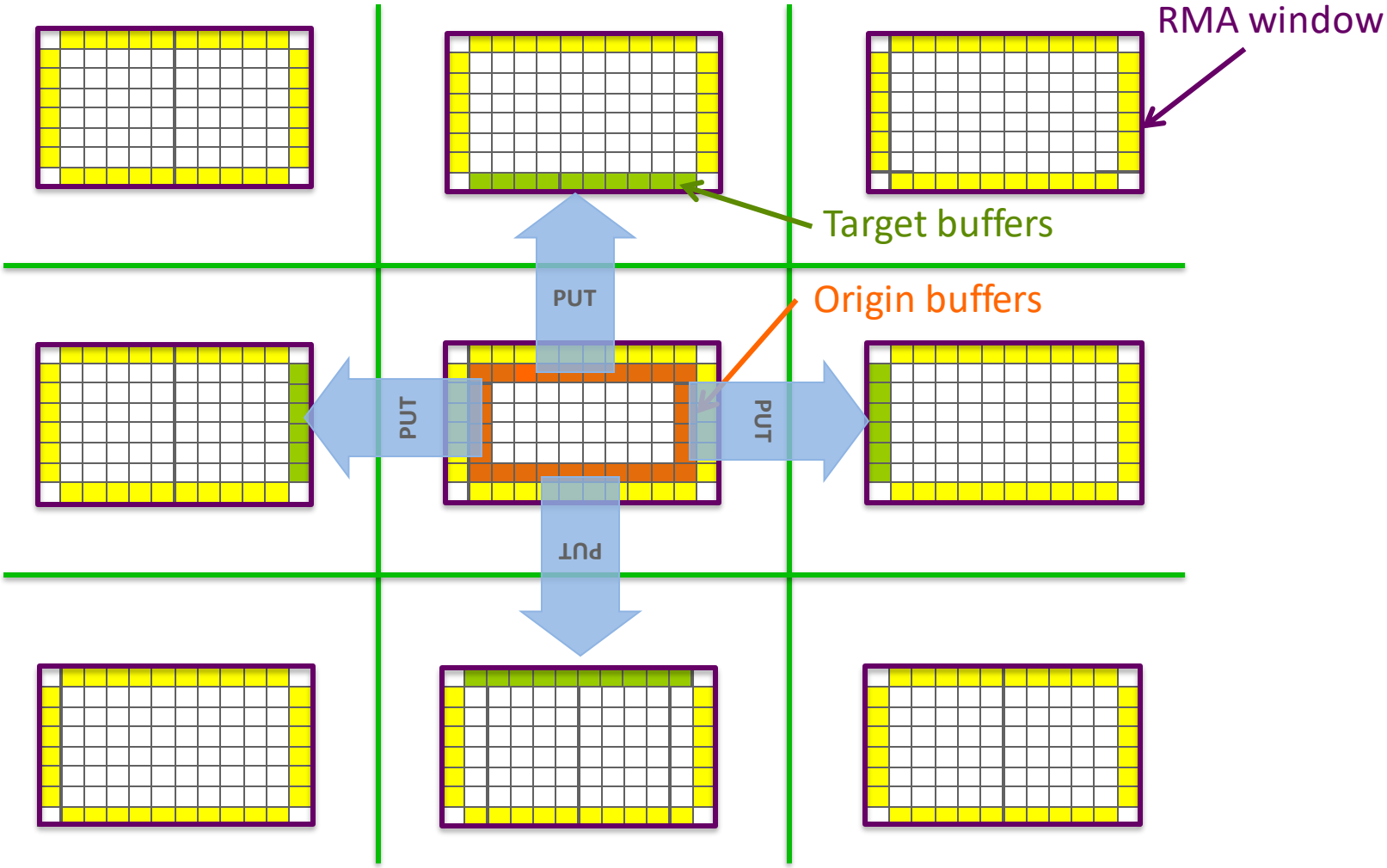
Fence: Active Target Synchronization

```
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



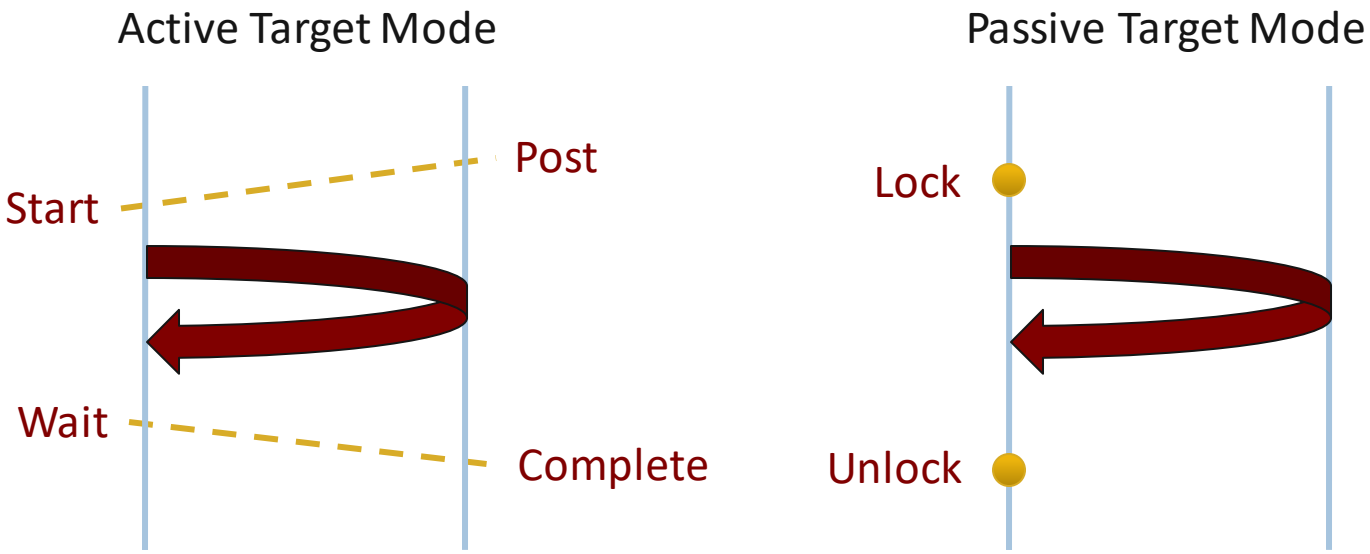
Exercise: Stencil with RMA Fence

- In the derived datatype version of the stencil code
  - Used nonblocking communication
  - Used derived datatypes
- Let's try to use RMA fence
  - Move data with PUT instead of send/recv
- *Start from derived_datatype/stencil.c*
- *Solution available in rma/stencil_fence_put.c*

Exercise: Stencil with RMA Fence (GET model)

- In the derived datatype version of the stencil code
  - Used nonblocking communication
  - Used derived datatypes
- Let's try to use RMA fence
  - Move data with GET instead of send/recv
- *Start from rma/stencil_fence_put.c*
- *Solution available in rma/stencil_fence_get.c*

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

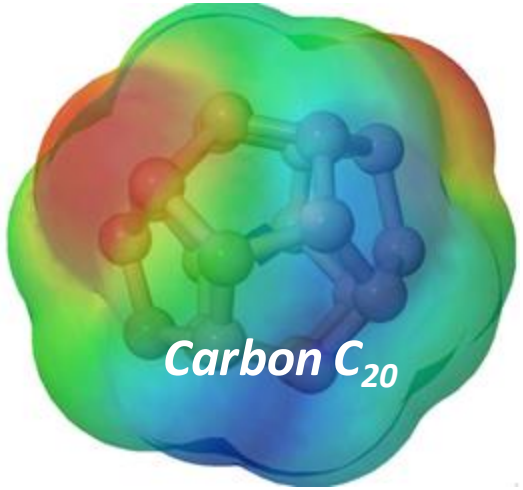
Advanced Passive Target Synchronization

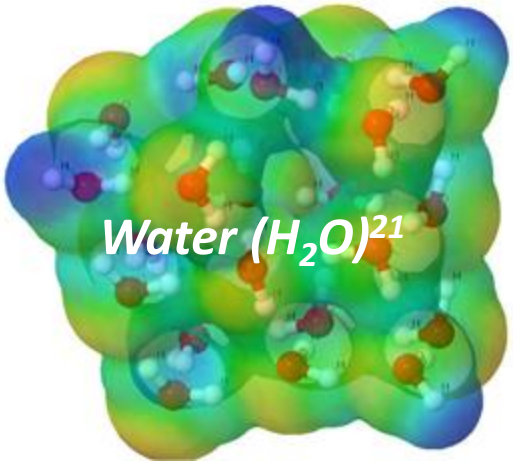
```
MPI_Win_lock_all(int assert, MPI_Win win)
MPI_Win_unlock_all(MPI_Win win)
MPI_Win_flush_all/flush_local_all(MPI_Win win)
```

- 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]

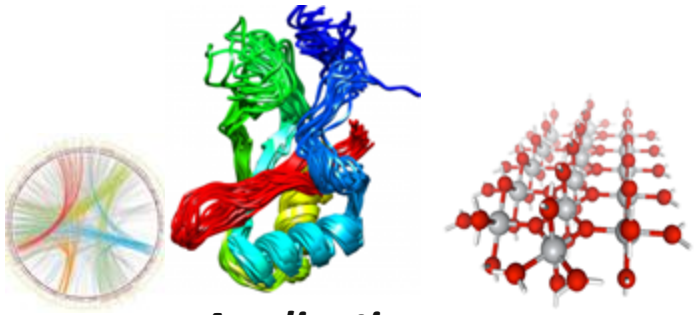
- High performance computational chemistry application suite
- Quantum level simulation of molecular systems
  - Very expensive in computation and data movement, so is used for small systems
  - Larger systems use molecular level simulations
- Composed of many simulation capabilities
  - Molecular Electronic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





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

NWChem Communication Runtime



Applications

Global Arrays [2]

ARMCI : Communication interface for RMA [3]

ARMCI native ports

IB

DMMAP

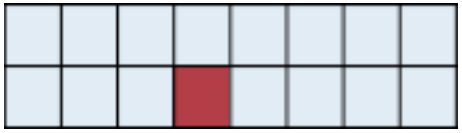
...

ARMCI-MPI

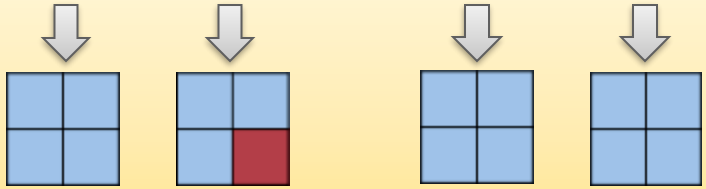
MPI RMA

Abstractions for distributed arrays

Global Address Space



Physically distributed to different processes



Hidden from user

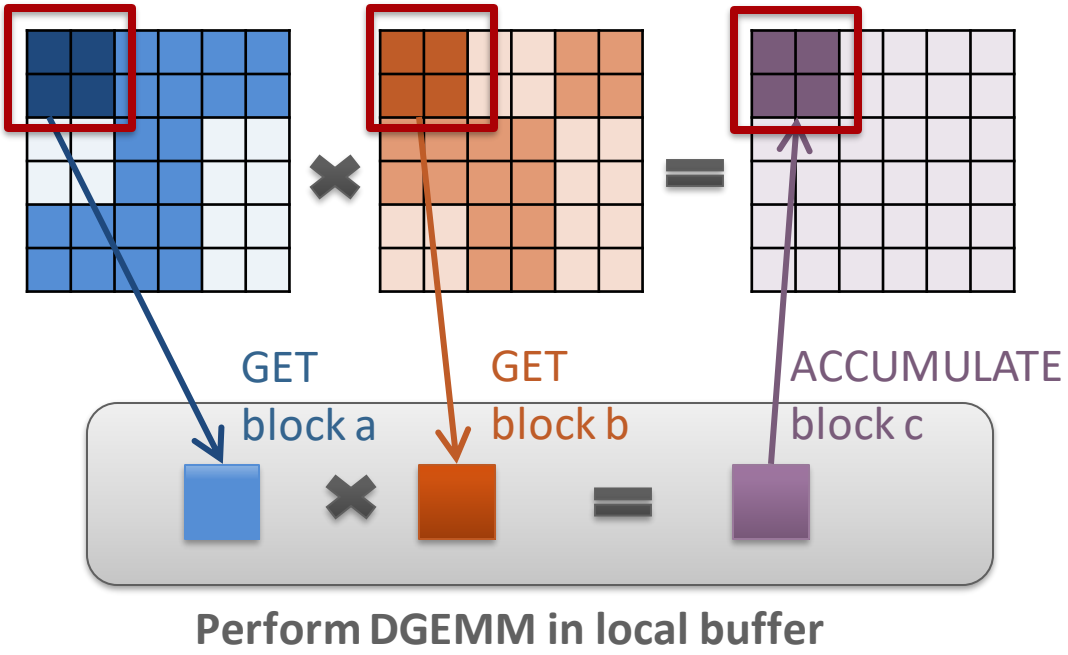
Irregularly access large amount of remote memory regions

[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.

Which synchronization mode should I use, when?

- RMA communication often has low overheads versus send/recv
  - Two-sided: Matching, queuing, buffering, unexpected receives, etc…
  - One-sided: No matching, no buffering, always ready to receive (but must separately sync the communication)
  - Direct use of RDMA provided by high-speed interconnects (e.g. InfiniBand)
    - Good two-sided implementations will also use RDMA, but must first match messages
- 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

Exercise: Stencil with RMA Fence

- In the derived datatype version of the stencil code
  - Used nonblocking communication
  - Used derived datatypes
- Let's try to use RMA fence
  - Move data with PUT instead of send/recv
- *Start from derived_datatype/stencil.c*
- *Solution available in rma/stencil_fence_put.c*

Exercise: Stencil with RMA Fence (GET model)

- In the derived datatype version of the stencil code
  - Used nonblocking communication
  - Used derived datatypes
- Let's try to use RMA fence
  - Move data with GET instead of send/recv
- *Start from rma/stencil_fence_put.c*
- *Solution available in rma/stencil_fence_get.c*

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

- In the fence and PSCW versions of the stencil code, RMA synchronization involves the target processes
- Let's try to use RMA Lock_all/Flush_all/Unlock_all
  - Only the origin processes call RMA synchronization
  - Still need **Barrier** for process synchronization (e.g., ensure neighbors have completed data update to my local window)
  - Need **Win_sync** for memory synchronization
- *Start from rma/stencil_fence_put.c*
- *Solution available in rma/stencil_lock_put.c*

MPI + Accelerators

Introduction

Accelerators are becoming increasingly popular in parallel computing

- **CPUs**
  - Task-sequential execution model (focus on latency)
  - Small # of complex compute cores (out-of-order execution)
  - Deep pipelines
  - Large caches
  - Branch prediction hardware
- **GPUs**
  - Data-parallel execution model (focus on throughput)
  - Large # of simple compute elements (in-order execution)
  - Small caches
  - Shallow pipelines
  - Large off-chip global High-Bandwidth Memory (HBM)
  - High FLOPs/W and FLOPs/$

Top500 Accelerators Based Systems (June 2019)

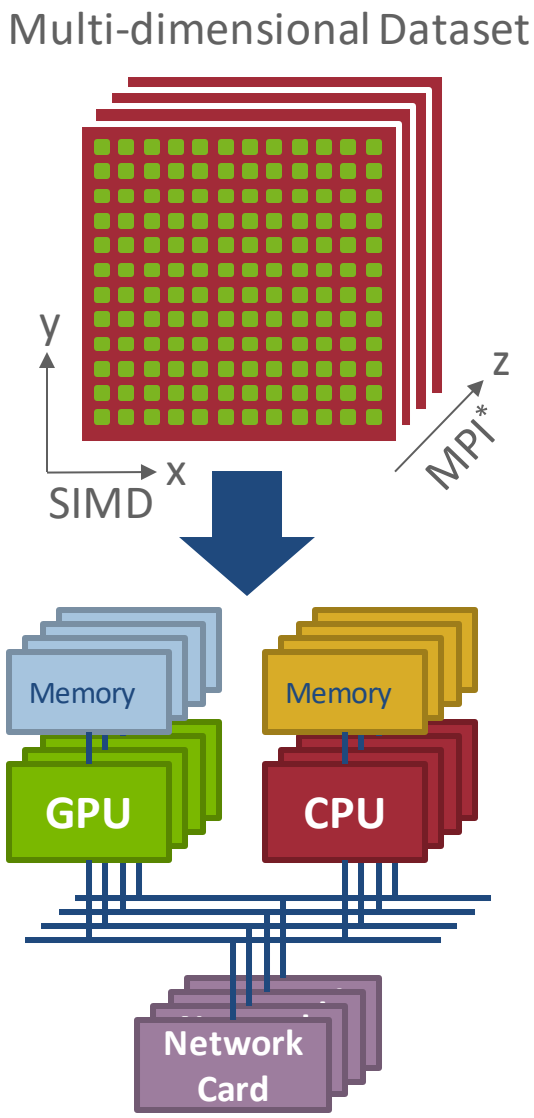
- #1 - Summit (ORNL USA)
  - NVIDIA Volta GV100
- #2 - Sierra (LLNL USA)
  - NVIDIA Volta GV100
- #6 - Piz Daint (CSCS Switzerland)
  - NVIDIA Tesla P100
- #8 - AI Bridging Cloud Infrastructure (AIST Japan)
  - NVIDIA Tesla V100
- #10 - Lassen (LLNL USA)
  - NVIDIA Volta V100

Upcoming Exascale Accelerators Based Systems

- Aurora (ANL USA)
  - Intel based technology
    - https://www.anl.gov/article/us-department-of-energy-and-intel-to-deliver-first-exascale-supercomputer
- Frontier (ORNL USA)
  - AMD based GPU technology
    - https://www.ornl.gov/news/us-department-energy-and-cray-deliver-record-setting-frontier-supercomputer-ornl
- Tianhe-3 (NUDT China)
  - Custom

Programming Model for Accelerators

- **GPUs** are well suited for fine grain data level parallelism
- Shared Memory, Single Instruction Multiple Data (SIMD) model
- Many available compute platforms and programming frameworks (focus on their memory model and interaction with MPI)
  - NVIDIA CUDA (NVIDIA platform only)
  - AMD ROCm & HIP
  - OpenMP
  - OpenACC
  - OpenCL & SYCL

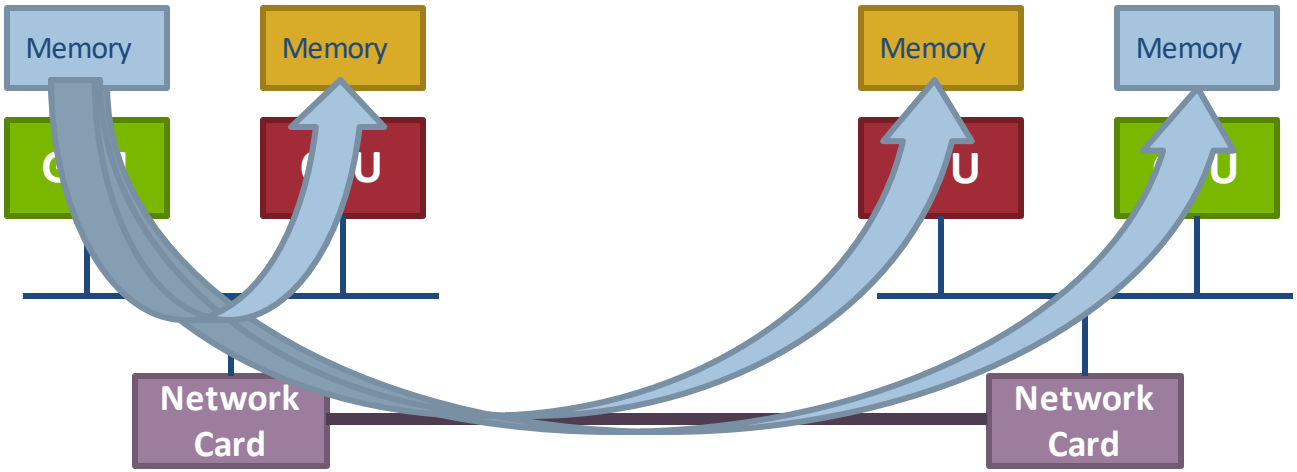


(*)Single Program Multiple Data

Interoperability with MPI

GPUs have separate physical memory subsystem

How to move data between GPUs with MPI?

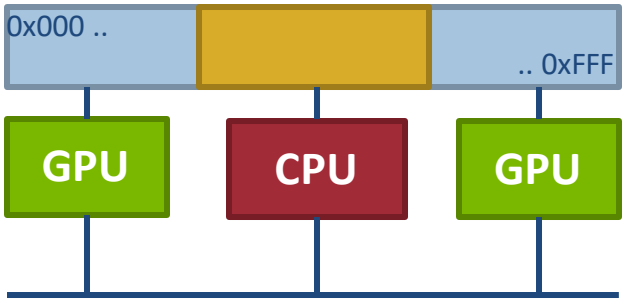


Real answer: It depends on what GPU library, what hardware and what MPI implementation you are using

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

Unified Virtual Addressing (UVA)

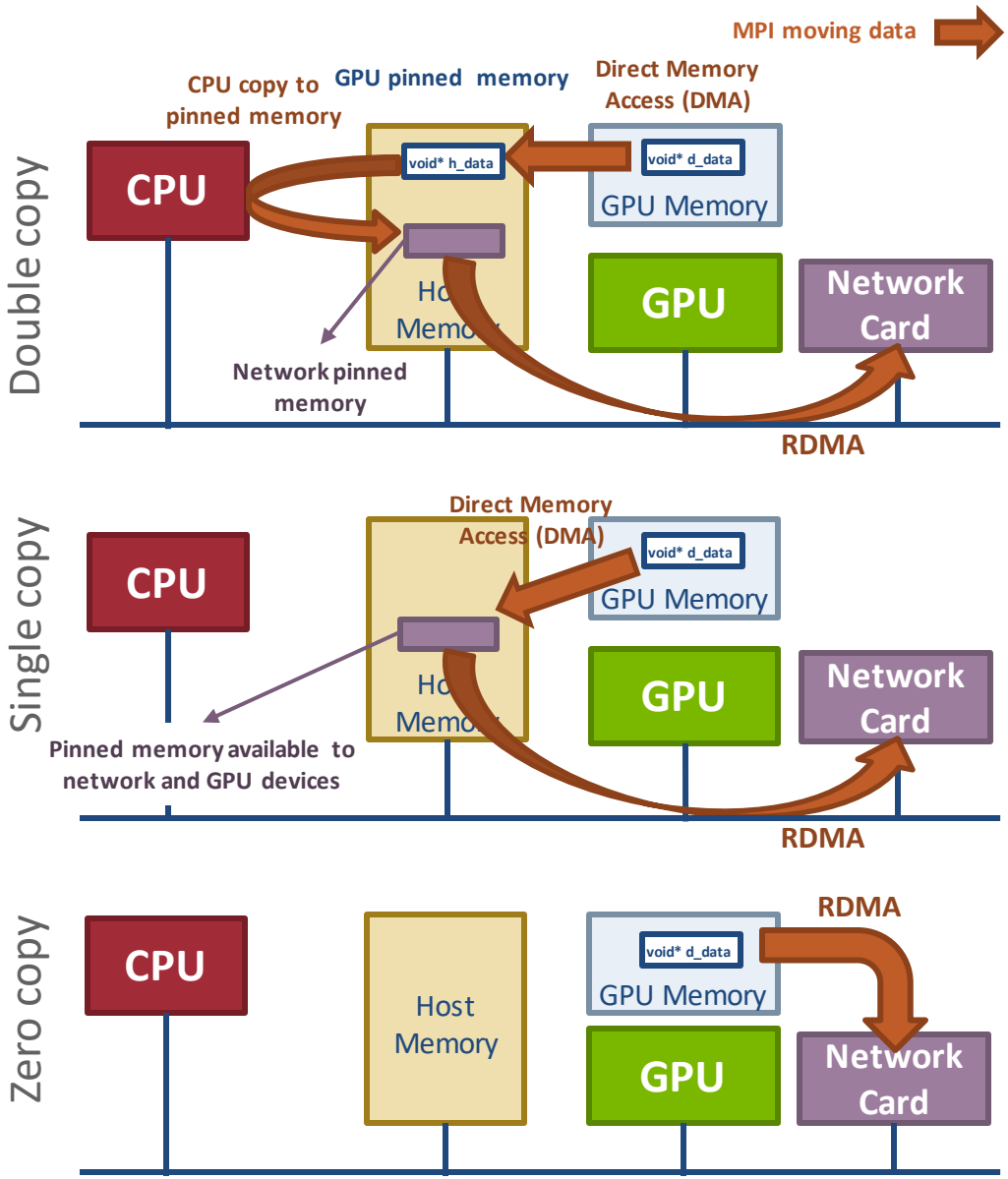
- UVA is a memory address management system supported in modern 64-bit architectures
  - Requires device driver support
- The same virtual address space is used for all processors, host or devices
- No distinction between host and device pointers
- The user can query the location of the data allocation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



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

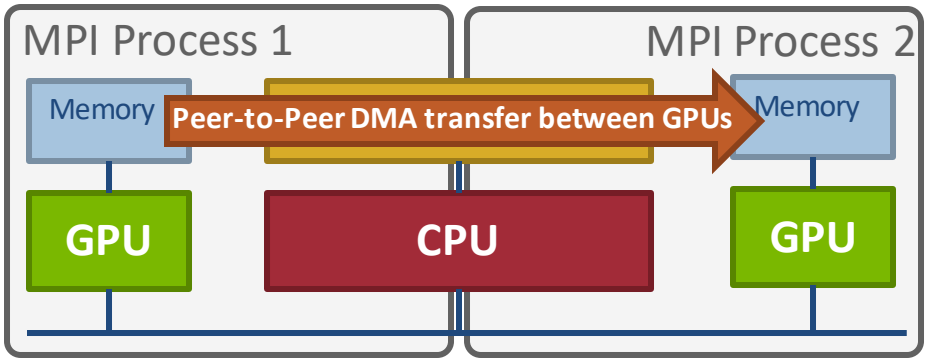
Remote Direct Memory Access with UVA

- Only GPU-enabled MPI implementations can take advantage of UVA
- User can pass device pointer to MPI
- MPI implementation can query for the owner (host or device) of the data
- If the data is on the device, **the MPI implementation can optimize** data transfers



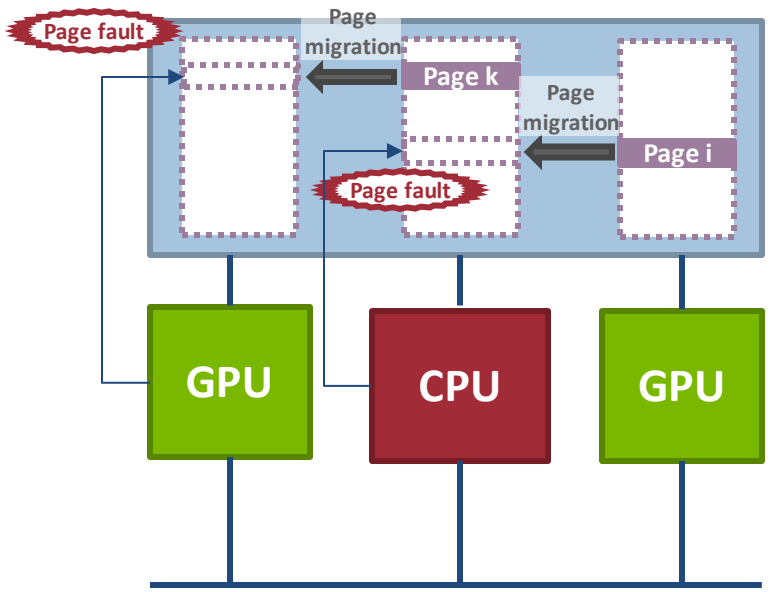
Intranode Communication with UVA

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



Heterogeneous Memory Management (HMM)

- Next step towards the unification of heterogeneous memory spaces in the Linux Kernel (not yet available)
- Support started with version 4.14 through helper functions to be used by device drivers
  - Support paging in device for migrating memory between host and GPU
- Automatic data movement between host and GPU memories (called Unified Memory in CUDA)
  - Data is automatically migrated between host and GPU on page faults
  - Moving pages to GPU and back to host is similar to swap-out and swap-in of pages to and from disk



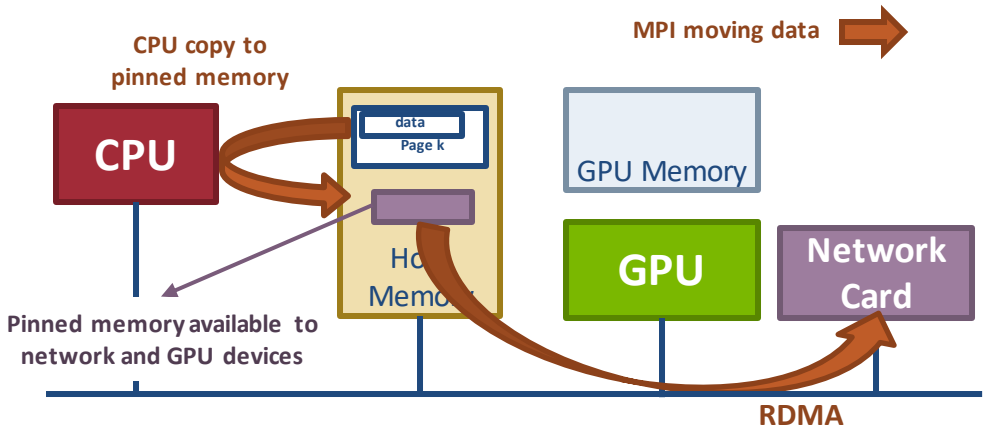
Single memory space accessible to all devices and host. Transparently managed heterogeneous memory.

MPI + HMM in a Nutshell

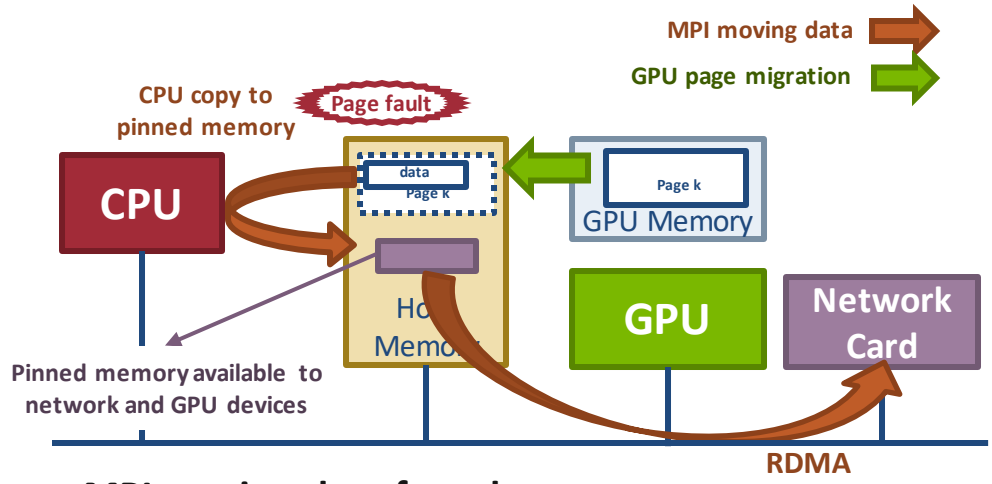
- In theory, any MPI implementation can transparently work with HMM
  - The MPI implementation can always assume that the data is on the host (or device)
  - GPUs take care of moving data between device and host memories
  - *Trying to register memory on the wrong device from the network should simply fail for HMM, but there have been reports of silent failures in this regard for CUDA, so you might need to be careful*
  - *Data in HMM can be corrupted if GPU updates pages during network transfer*
- In practice MPI implementations should never use HMM directly
  - Managed heterogeneous memory cannot be directly accessed by the network (correctness issue)
    - Virtual address cannot be pinned to a fixed physical memory region since GPU might need to migrate the data
    - Intermediate buffer is needed to copy data from HMM
- In any case MPI can never know in which device data is physically located
  - Data management is completely handled by GPU and can cause unnecessary data movement (performance issue)

MPI + HMM Assuming Data on Host

- MPI can assume data is on host memory
- MPI copies data to network pinned memory
  - Network registration will fail
- On a correct guess
  - The copy will not trigger a page fault to bring data from GPU
- On incorrect guess
  - An **expensive page fault** will occur



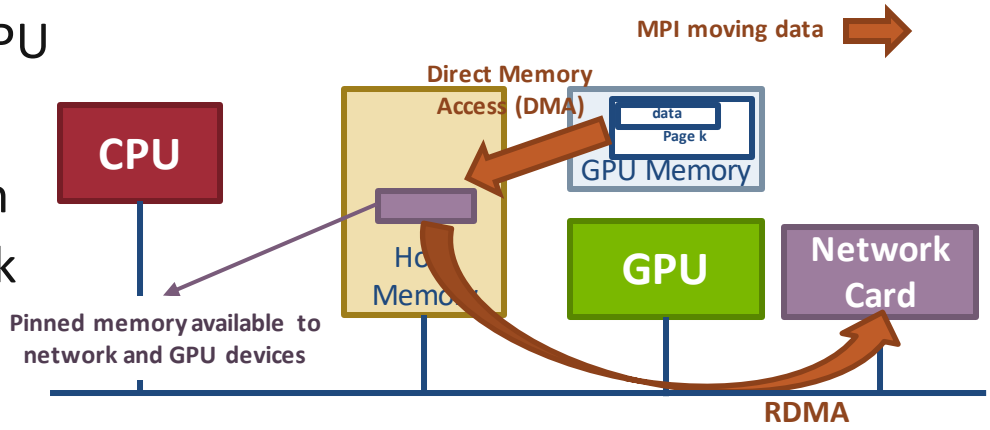
Correct guess: data is on host memory



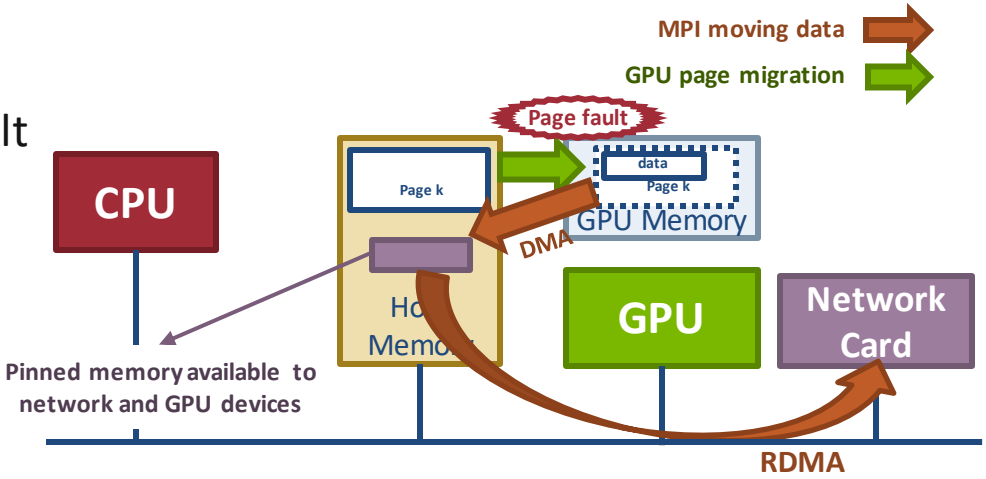
MPI moving data from host memory on a wrong guess

MPI + HMM Assuming Data on GPU

- MPI can assume data is on some GPU device memory
- MPI would need to move data from the GPU device memory to network pinned memory
  - This can be either host or GPU memory (but not unified memory)
- On a correct guess
  - The copy will not trigger a page fault
- On incorrect guess
  - An **expensive page fault** will occur when accessing data on the GPU device memory
- Most MPI implementations assume memory to reside on the GPU



Correct guess: data is on the target GPU



MPI moving data from host memory on a wrong guess