UNIVERSITY of HOUSTON
CENTER FOR ADVANCED COMPUTING & DATA SYSTEMS

Introduction to Parallel Programming with MPI Part II

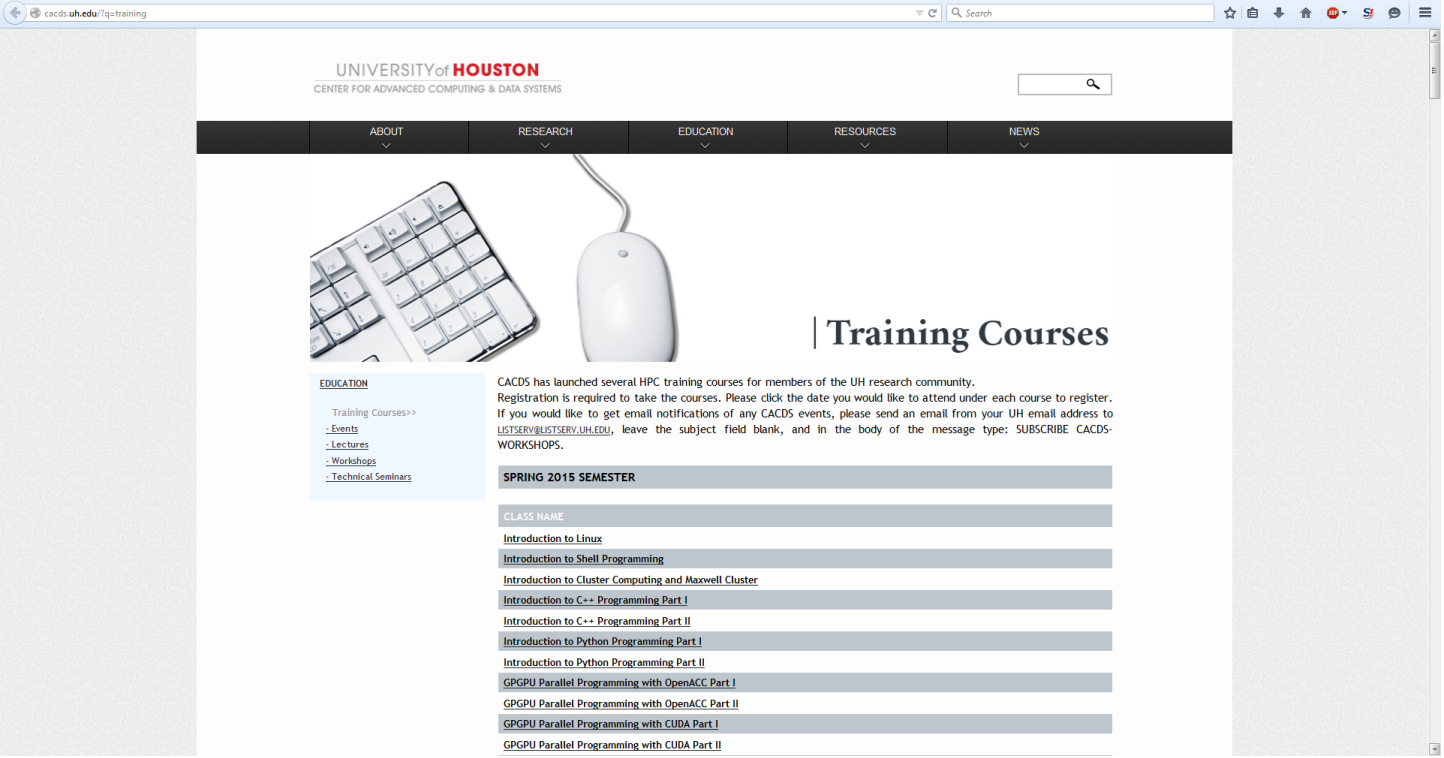
Amit Amritkar
HPC Specialist, CACDS
cacds.uh.edu

About CACDS

Mission Statement:

CACDS provides High Performance Computing (HPC) resources and services to advance Tier One research and education goals at the University of Houston (UH).

CACDS Training courses: Register at www.cacds.uh.edu/training



CACDS brings
Talk on Big data by XSEDE, 4/7

- This workshop will focus on topics such as Hadoop and Spark.

Register here (4/7/2015)

- https://portal.xsede.org/course-calendar/-/training-user/class/378/session/633

First Access Your Account

- Log into your accounts
  - Username or login = hpc_user**X**
  - Where X = serial number 1 – 47 from the sign-in sheet
  - Password = **cacds2014**

Accessing Tutorial Materials

First login into the cluster:

TYPE AND EXECUTE COMMANDS IN Green!!!

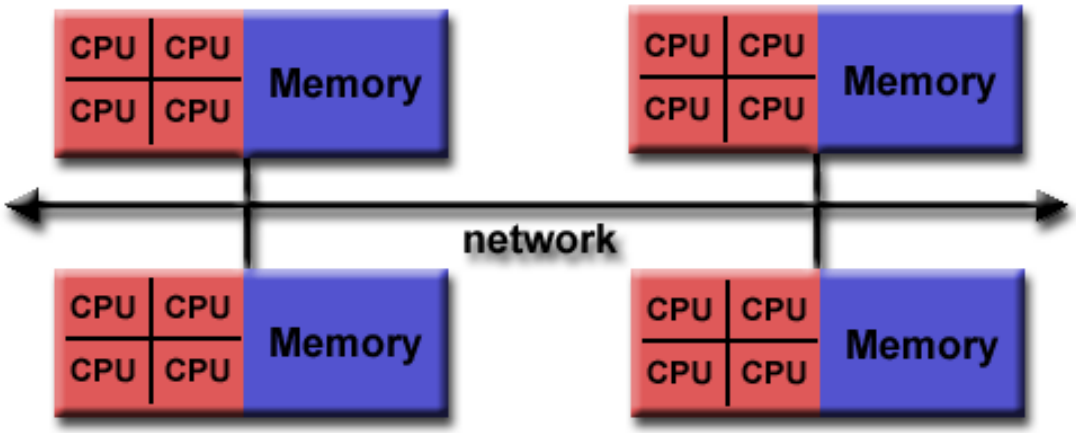
```
cd
cp /share/apps/tutorials/mpi_2.pdf  ~
cp /share/apps/tutorials/mpi_2.zip  ~
unzip mpi_2.zip
cd mpi_2
module add openmpi
```

Overview

- Recap
  - MPI
  - Basics
  - Communication
    - Point to point
    - Collective
- Advanced MPI
  - Derived Types
  - Process Groups and Communicators
  - Virtual Topology
  - One sided communication
  - Heterogeneous memory systems
- Process bindings
- Road Ahead
  - ExaScale computing

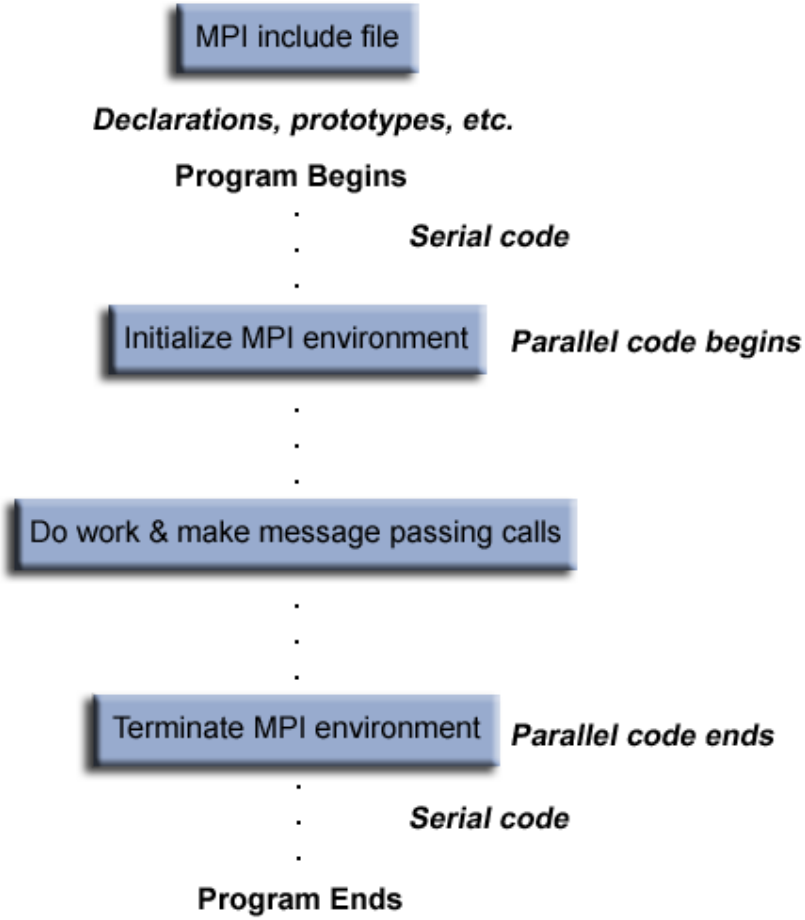
What is MPI?

- MPI: Message Passing Interface
  - The MPI Forum has broad participation
    - So
  - Incor
    - Ea
    - Ea
      - what the
      - mantics match



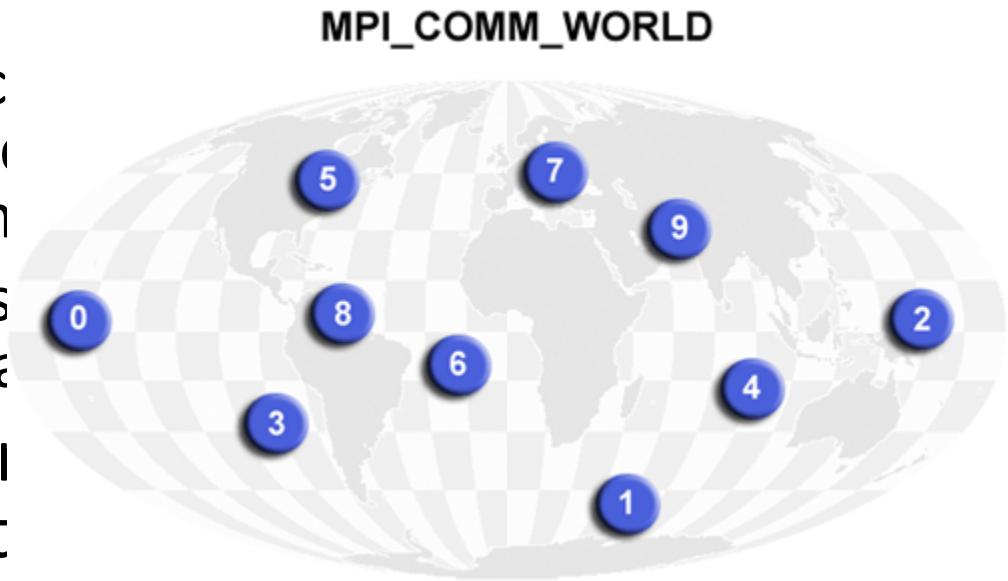
- MPI is not…
  - a language or compiler specification
  - a specific implementation or product

MPI Program Structure



Rank and Communicator

- Communicator:
  - MPI uses objects c
    define which colle
    communicate with
  - Most MPI routines
    communicator as a
- Rank - Within a co
  its own unique, int
  system when the process initializes. A rank is sometimes also called a "task ID". Ranks are contiguous and begin at zero.



Call Format

- Header File

C include file	Fortran include file
#include "mpi.h"	include 'mpif.h'

- Format of MPI Calls

C Binding	
Format:	rc = MPI_Xxxxx(parameter, ...)
Example:	rc = MPI_Bsend(&buf,count,type,dest,tag,comm)
Error code:	Returned as "rc". MPI_SUCCESS if successful
Fortran Binding	
Format:	CALL MPI_XXXXX(parameter,..., ierr) call mpi_xxxxx(parameter,..., ierr)
Example:	CALL MPI_BSEND(buf,count,type,dest,tag,comm,ierr)
Error code:	Returned as "ierr" parameter. MPI_SUCCESS if successful

Point-to-point communication

- Divide work between available tasks which communicate data via point-to-point message passing calls
- MPI_send, MPI_recv, MPI_sendrecv
- Blocking vs. Non-blocking
- MPI_isend, MPI_irecv, MPI_wait

Synchronization routines

- MPI_Barrier
  - Blocks until all processes in the communicator have reached this routine.
  - C syntax: MPI_Barrier (comm)
  - Fortran syntax: MPI_BARRIER (comm, ierr)
  - Seldom use (avoid if possible)

- MPI_wait
  - MPI_Wait blocks until a specified non-blocking send or receive operation has completed
  - C syntax: int MPI_Wait(MPI_Request *request, MPI_Status *status)
  - Fortran syntax: MPI_WAIT(request, status, ierr)

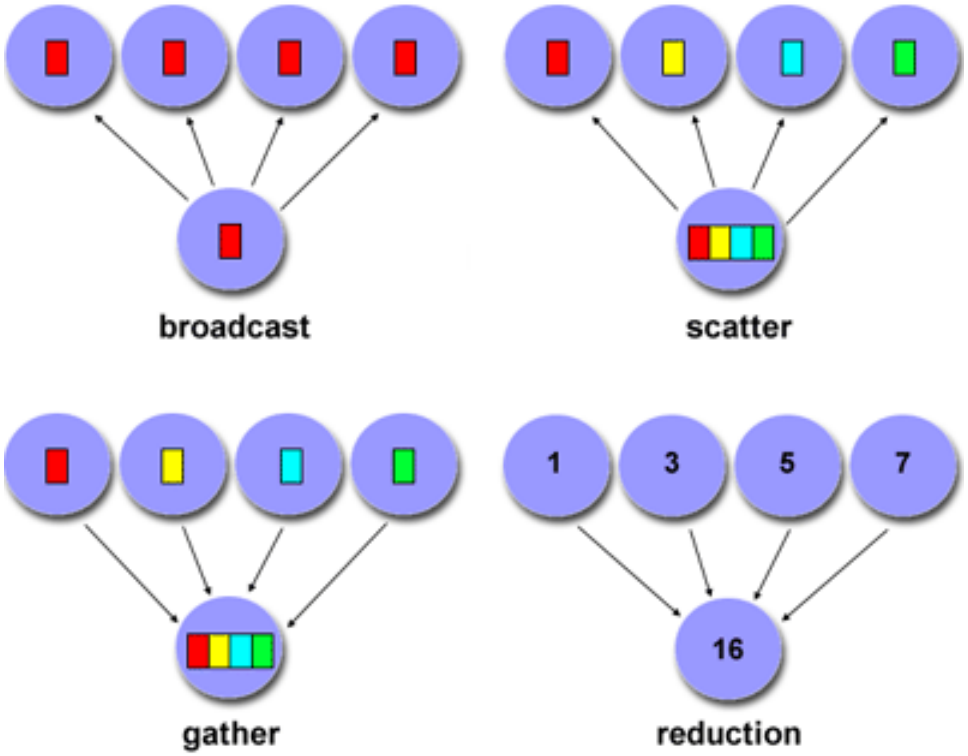
Exercise 0

- Print the value of $(mpi\_rank)^2$ on 4 processors while non-blocking communication is being performed
  - Use exercise0.c or exercise0.f

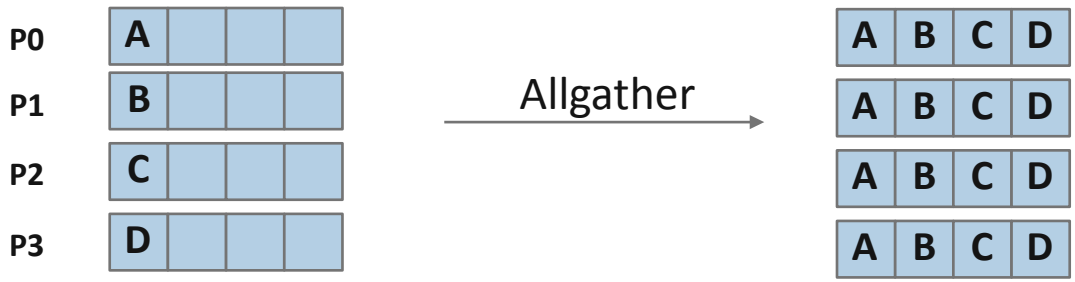
- Solution hint
  - print *, 'rank2= ',rank**2

Collective communication

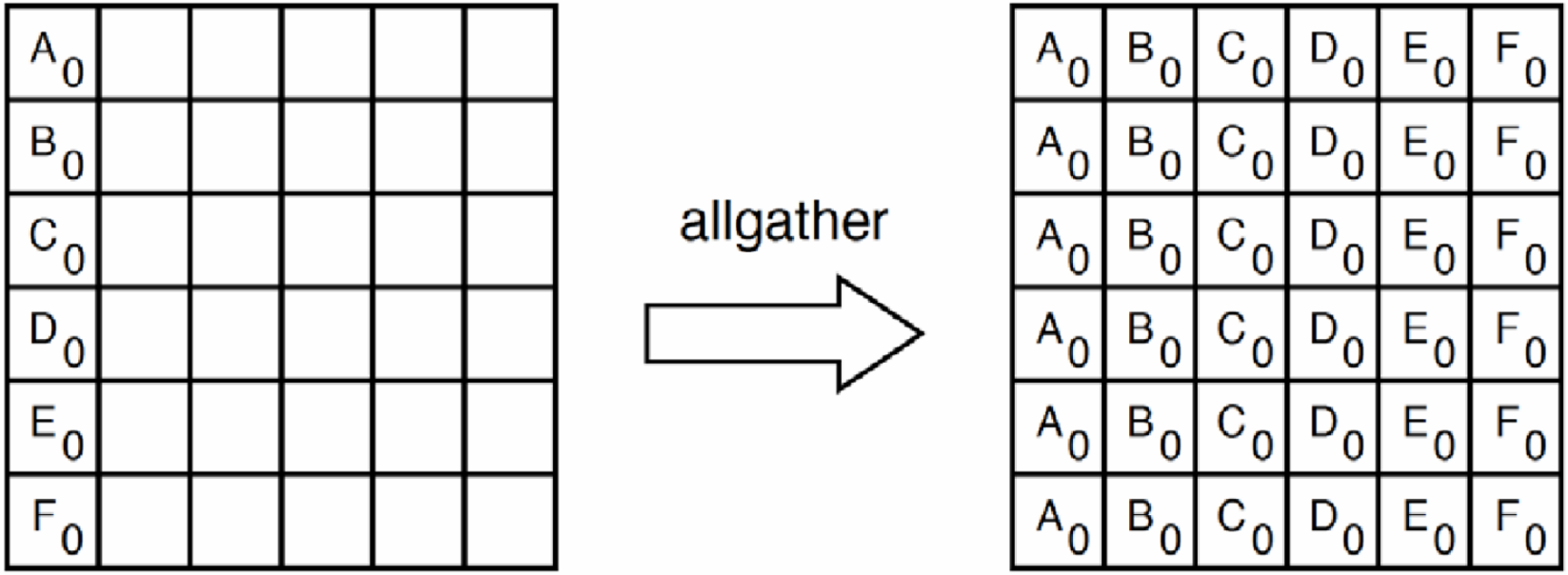
- Broadcast, Scatter, Gather and Reduction



More collective data movement



All Gather


A0
B0
C0
D0
E0
F0
allgather
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0
A0 B0 C0 D0 E0 F0

All Gather

- MPI_Allgather routine gathers data from all tasks and distribute the combined data to all tasks
- MPI_Allgather can be thought of as MPI_GATHER, but where all processes receive the result, instead of just the root. The block of data sent from the j-th process is received by every process and placed in the j-th block of the buffer recvbuf.

All Gather

```
MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf,
recvcount, recvtype, comm)
```

- IN sendbuf starting address of send buffer (choice)
- IN sendcount number of elements in send buffer (non-negative integer)
- IN sendtype data type of send buffer elements (handle)
- OUT recvbuf address of receive buffer (choice)
- IN recvcount number of elements received from any process (nonnegative integer)
- IN recvtype data type of receive buffer elements (handle)
- IN comm communicator (handle)

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

AllGather Example

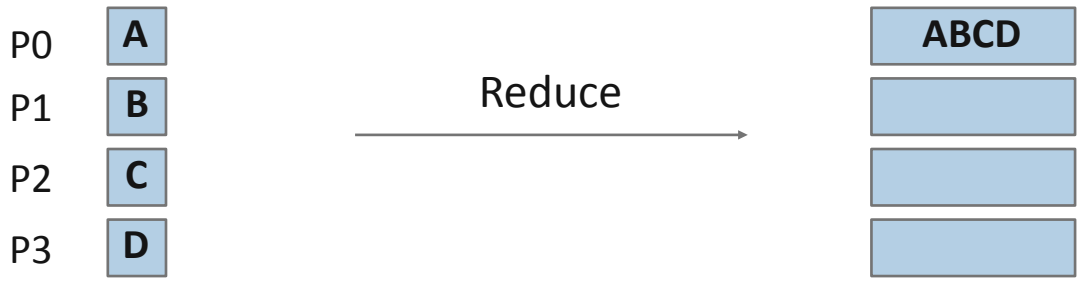
- Using MPI_ALLGATHER, we will gather 100 ints from every process in the group to every process.

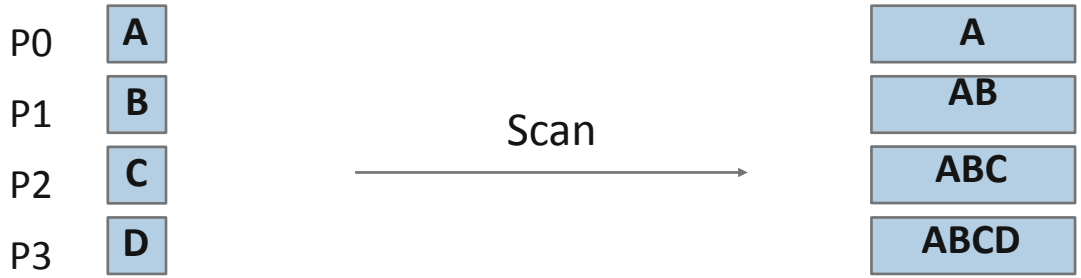
```
MPI_Comm comm;
int gsize,sendarray[100];
int *rbuf;
...
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Allgather(sendarray, 100, MPI_INT, rbuf,
100, MPI_INT, comm);
```

- After the call, every process has the group-wide concatenation of the sets of data.

mpi_2/example_allgather.c → add print statement

Collective Computation





MPI_Reduce

Reduces values on all processes to a single value.

```
MPI_Reduce(sendbuf,recvbuf, count,
  MPI_Datatype, MPI_Op, root, MPI_Comm)
```

sendbuf = address of send buffer (choice)
count = number of elements in send buffer (integer)
datatype = data type of elements of send buffer (handle)
op = reduce operation (handle)
root = rank of root process (integer)
comm = communicator (handle)

MPI_Reduce

Reduces values on all processes to a single value.

```
MPI_Reduce(sendbuf,
   MPI_Datatype, MPI_
```

sendbuf = address of se
count = number of ele
datatype = data type of e
op = reduce operation
root = rank of root p
comm = communicato

[MPI_MAX] maximum
[MPI_MIN] minimum
[MPI_SUM] sum
[MPI_PROD] product
[MPI_LAND] logical and
[MPI_BAND] bit-wise and
[MPI_LOR] logical or
[MPI_BOR] bit-wise or
[MPI_LXOR] logical xor
[MPI_BXOR] bit-wise xor
[MPI_MAXLOC] max value and location
[MPI_MINLOC] min value and location

Exercise 1

- Compute the average of random numbers across multiple (4 and 8) processors
- The program takes the following steps:
  - Generate a random array of numbers on the root process (process 0).
  - Scatter the numbers to all processes, giving each process an equal amount of numbers.
  - Each process computes the average of their subset of the numbers.
  - Reduce the sum across all the processors and print "sum/total_elements"
- Start from exercise1.c

Solution 1 (partial code)

```
// Sum the numbers locally
float local_sum = 0;
int i;
for (i = 0; i < num_elements_per_proc; i++) {
  local_sum += rand_nums[i];
}

// Print the random numbers on each process
printf("Local sum for process %d - %f, avg = %f\n",
      world_rank, local_sum, local_sum / num_elements_per_proc);

// Reduce all of the local sums into the global sum
float global_sum;
MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0,
         MPI_COMM_WORLD);

// Print the result
if (world_rank == 0) {
  printf("Total sum = %f, avg = %f\n", global_sum,
        global_sum / (world_size * num_elements_per_proc));
}
```

```
Local sum for process 1 - 26.951777, avg = 0.539036
Local sum for process 2 - 26.363638, avg = 0.527273
Local sum for process 3 - 21.108759, avg = 0.422175
Local sum for process 0 - 26.951777, avg = 0.539036
Total sum = 101.375946, avg = 0.506880
```

Predefined primitive data types

- For communication and type construction

C Data Types		Fortran Data Types
MPI_CHAR MPI_WCHAR	MPI_C_COMPLEX	MPI_CHARACTER
MPI_SHORT MPI_INT	MPI_C_FLOAT_COMPLEX	MPI_INTEGER
MPI_LONG	MPI_C_DOUBLE_COMPLEX	MPI_INTEGER1
MPI_LONG_LONG_INT	MPI_C_LONG_DOUBLE_COMPLEX	MPI_INTEGER2
MPI_LONG_LONG	MPI_C_BOOL MPI_LOGICAL	MPI_INTEGER4
MPI_SIGNED_CHAR	MPI_C_LONG_DOUBLE_COMPLEX	MPI_REAL
MPI_UNSIGNED_CHAR	MPI_INT8_T	MPI_REAL2
MPI_UNSIGNED_SHORT	MPI_INT16_T	MPI_REAL4
MPI_UNSIGNED_LONG	MPI_INT32_T	MPI_REAL8
MPI_UNSIGNED MPI_FLOAT	MPI_INT64_T	MPI_DOUBLE_PRECISION
MPI_DOUBLE	MPI_UINT8_T	MPI_COMPLEX
MPI_LONG_DOUBLE	MPI_UINT16_T	MPI_DOUBLE_COMPLEX
	MPI_UINT32_T	MPI_LOGICAL
	MPI_UINT64_T	~~MPI_BYTE~~
	~~MPI_BYTE~~	MPI_PACKED
	MPI_PACKED	

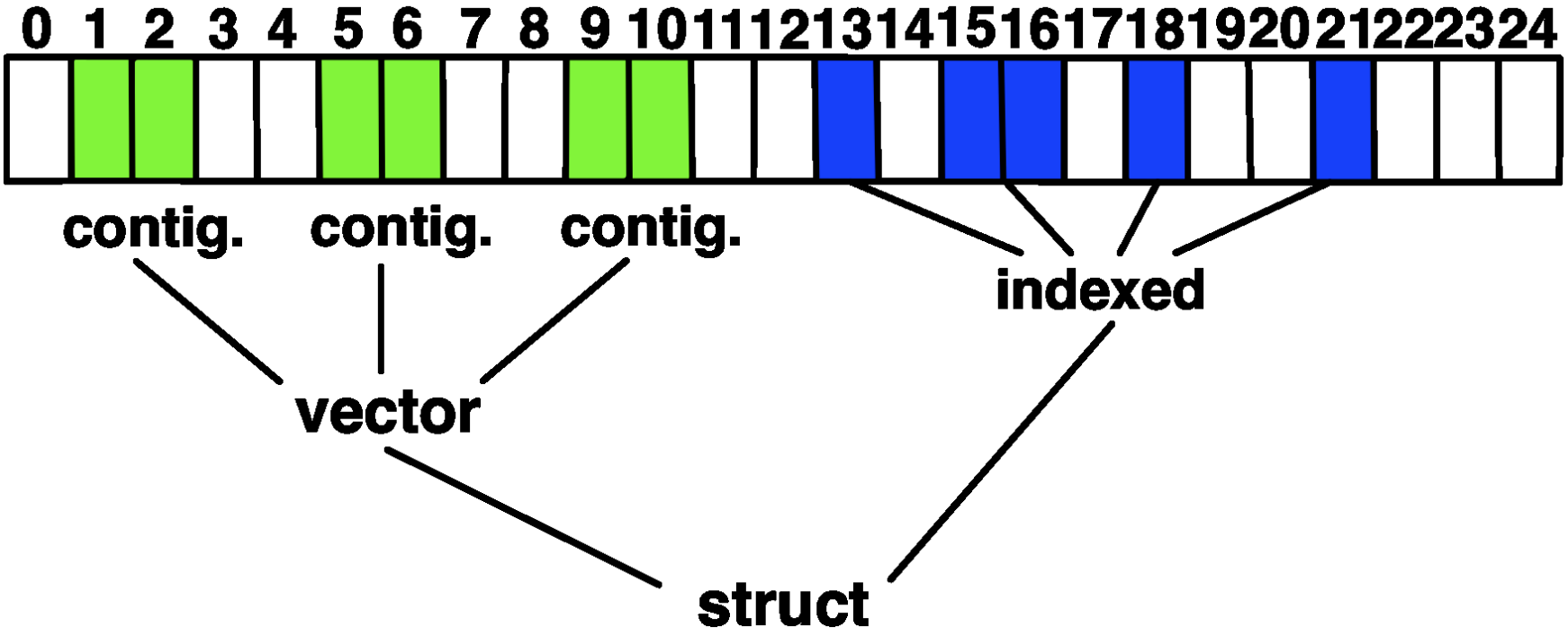
Derived Data Type Routines

- Define your own data structures based upon sequences of the MPI primitive data types.
  - Such user defined structures are called derived data types.

- Primitive data types are contiguous. Derived data types allow you to specify non-contiguous data in a convenient manner and to treat it as though it was contiguous.

- MPI provides several methods for constructing derived data types: (in ascending cost)
  - Contiguous < vector < indexed block < index < struct

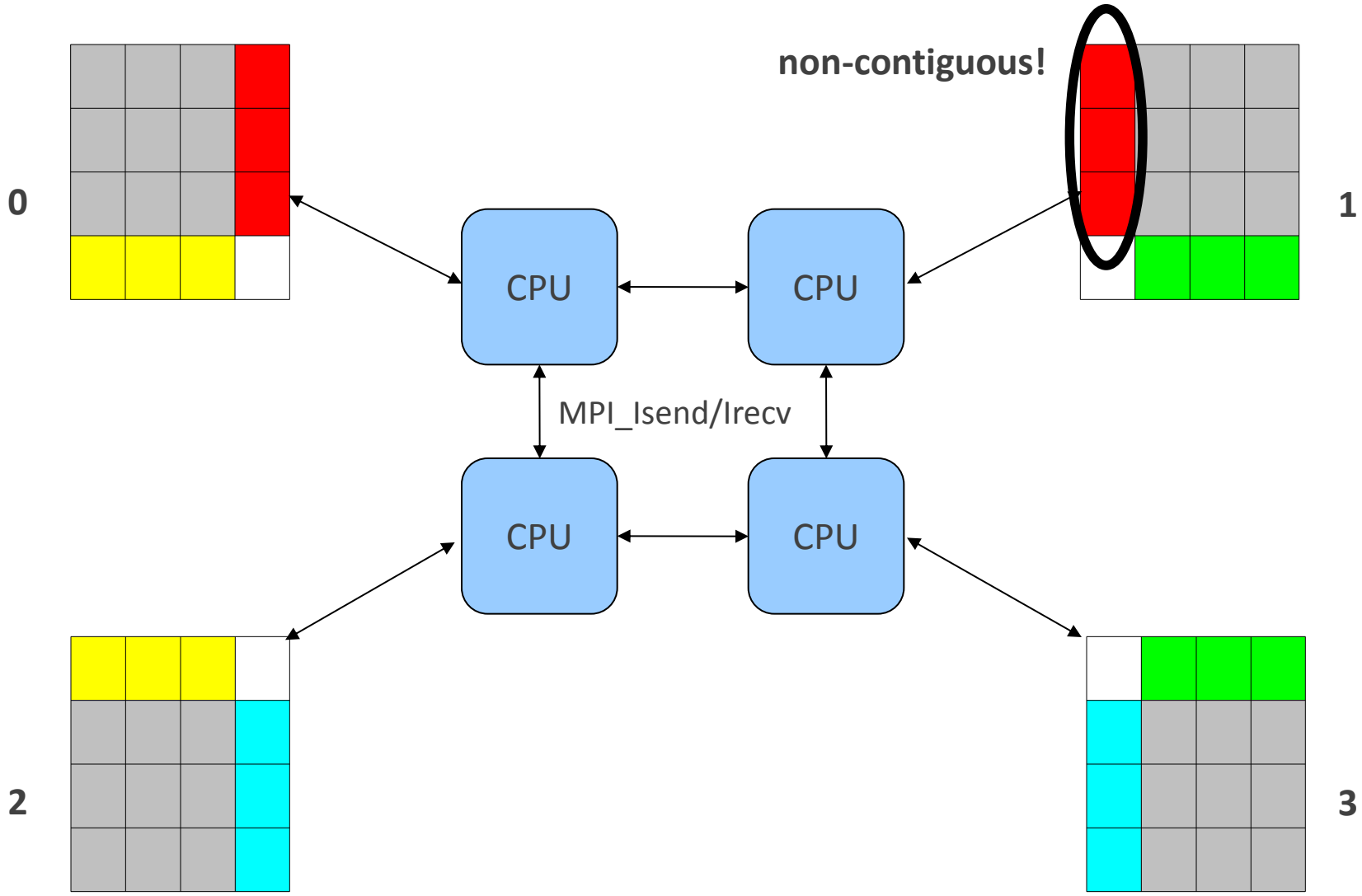
Derived data types



Basic derived data type constructs

- MPI_Type_contiguous - Produces a new data type by making count copies of an existing data type.
  - MPI_Type_contiguous (count,oldtype,&newtype)
  - MPI_TYPE_CONTIGUOUS (count,oldtype,newtype,ierr)
- MPI_Type_commit - Commits new datatype to the system. Required for all derived datatypes.
  - MPI_Type_commit (&datatype)
  - MPI_TYPE_COMMIT (datatype,ierr)
- MPI_Type_free - Deallocates the specified datatype object.
  - MPI_Type_free (&datatype)
  - MPI_TYPE_FREE (datatype,ierr)

MPI - Stencil Computation



Exercise 2: Contiguous Derived Data Type

MPI_ Type_contiguous

count = 4;
MPI_Type_contiguous(count, MPI_FLOAT, &rowtype);

1.0	2.0	3.0	4.0
5.0	6.0	7.0	8.0
9.0	10.0	11.0	12.0
13.0	14.0	15.0	16.0

a[4][4]

MPI_Send(&a[2][0], 1, rowtype, dest, tag, comm);

9.0	10.0	11.0	12.0

1 element of rowtype

- Create derived data type to pack the rows of the matrix and send each row to corresponding mpi process.
  - Start from exercise2.c or exercise2.f
  - Remember Fortran is column major

Sample program output:

rank= 0 b= 1.0 2.0 3.0 4.0
rank= 1 b= 5.0 6.0 7.0 8.0
rank= 2 b= 9.0 10.0 11.0 12.0
rank= 3 b= 13.0 14.0 15.0 16.0

Solution 2

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

main(int argc, char *argv[])  {
.
.
.
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
MPI_Type_commit(&rowtype);
.
.
.
MPI_Type_free(&rowtype);
MPI_Finalize();
}
```

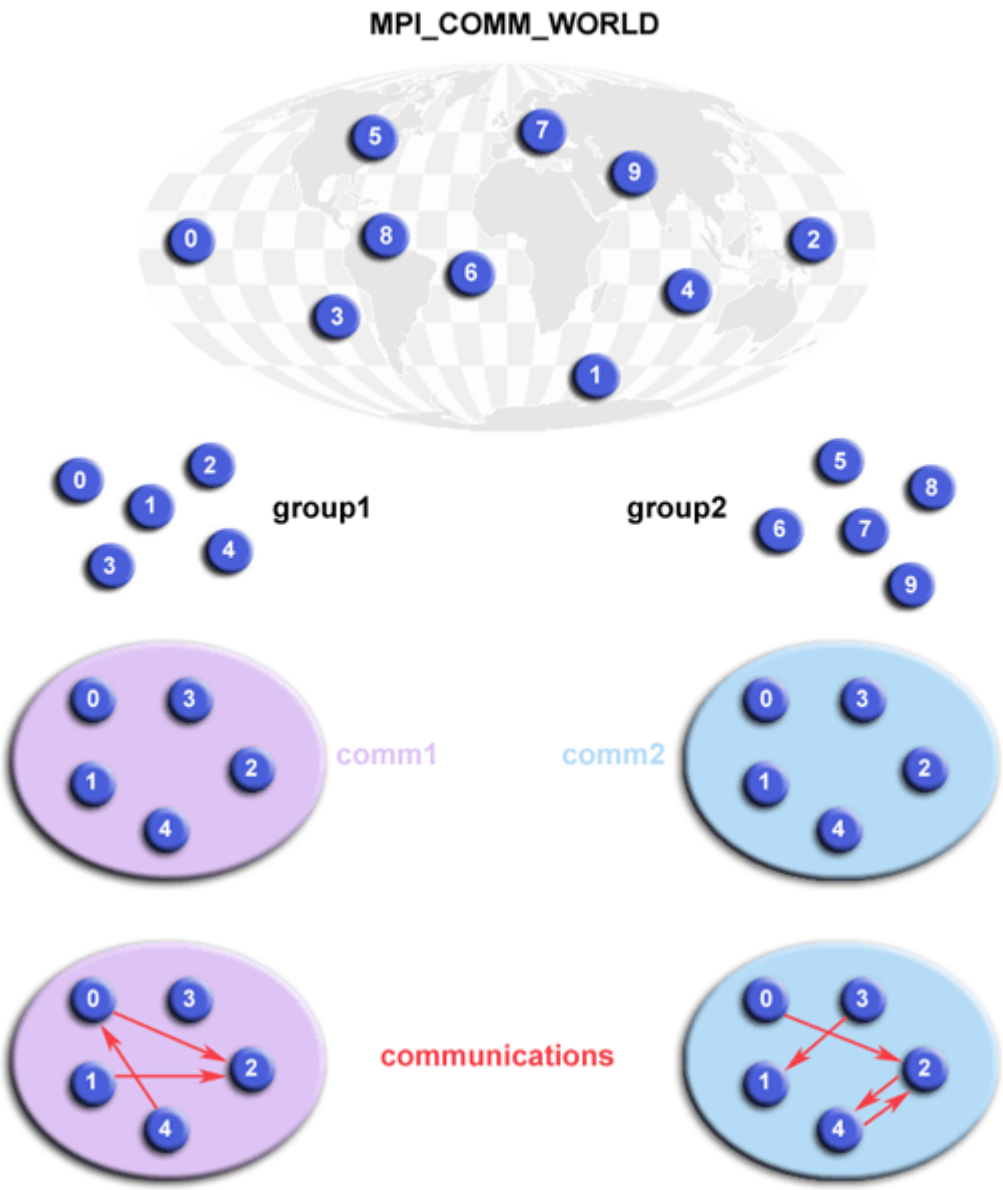
```
program contiguous
  include 'mpif.h'
.
.
.
  call MPI_INIT(ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, numtasks, ierr)

  call MPI_TYPE_CONTIGUOUS(SIZE, MPI_REAL, columntype, ierr)
  call MPI_TYPE_COMMIT(columntype, ierr)
.
.
.
  call MPI_TYPE_FREE(columntype, ierr)
  call MPI_FINALIZE(ierr)

  end
```

Group and Communicator Management Routines

- Groups vs. Communicators
  - A group is an ordered set of processes.
  - A communicator encompasses a group of processes that may communicate with each other.
- Allow to organize tasks, based upon function, into task groups.
- Enable Collective Communications operations across a subset of related tasks.
- Provide for safe communications
- Provide basis for implementing user defined virtual topologies

Groups/Communicators

- Programming considerations
  - Groups/communicators can be created and destroyed during program execution
  - Processes may be in more than one group/communicator.
    - They will have a unique rank within each group/communicator.
- Typical usage:
  - Extract handle of global group from MPI_COMM_WORLD using MPI_Comm_group
  - Form new group as a subset of global group using MPI_Group_incl
  - Create new communicator for new group using MPI_Comm_create
  - Determine new rank in new communicator using MPI_Comm_rank
  - Conduct communications using any MPI message passing routine
  - When finished, free up new communicator and group (optional) using MPI_Comm_free and MPI_Group_free


MPI_COMM_WORLD
group1
group2
comm1
comm2
communications

Group and communicator creation

```
/* Extract the original group handle */
MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

/* Divide tasks into two distinct groups based upon rank */
if (rank < NPROCS/2) {
  MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
  }
else {
  MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);
  }

/* Create new new communicator and then perform collective
communications */
MPI_Comm_create(MPI_COMM_WORLD, new_group,
&new_comm);
MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM,
new_comm);

MPI_Group_rank (new_group, &new_rank);
printf("rank= %d newrank= %d recvbuf=
%d\n",rank,new_rank,recvbuf);
```

```
C     Extract the original group handle
      call MPI_COMM_GROUP(MPI_COMM_WORLD, orig_group, ierr)

C     Divide tasks into two distinct groups based upon rank
      if (rank .lt. NPROCS/2) then
        call MPI_GROUP_INCL(orig_group, NPROCS/2, ranks1,
     &                  new_group, ierr)
      else
        call MPI_GROUP_INCL(orig_group, NPROCS/2, ranks2,
     &                  new_group, ierr)
      endif

      call MPI_COMM_CREATE(MPI_COMM_WORLD, new_group,
     &                  new_comm, ierr)
      call MPI_ALLREDUCE(sendbuf, recvbuf, 1, MPI_INTEGER,
     &                  MPI_SUM, new_comm, ierr)

      call MPI_GROUP_RANK(new_group, new_rank, ierr)
      print *, 'rank= ',rank,' newrank= ',new_rank,' recvbuf= ',
     &     recvbuf
```

Exercise 3

- Run the given sample code (exercise4) for 8 processors
- Modify the program to run on 4 processors

Sample program output 1:

```
rank= 7 newrank= 3 recvbuf= 22
rank= 0 newrank= 0 recvbuf= 6
rank= 1 newrank= 1 recvbuf= 6
rank= 2 newrank= 2 recvbuf= 6
rank= 6 newrank= 2 recvbuf= 22
rank= 3 newrank= 3 recvbuf= 6
rank= 4 newrank= 0 recvbuf= 22
rank= 5 newrank= 1 recvbuf= 22
```

Virtual Topologies

- In terms of MPI, a virtual topology describes a mapping/ordering of MPI processes into a geometric "shape".
- The two main types of topologies supported by MPI are Cartesian (grid) and Graph.
- MPI topologies are virtual - there may be no relation between the physical structure of the parallel machine and the process topology.
- Virtual topologies are built upon MPI communicators and groups.
- Must be "programmed" by the application developer.

Use of Virtual Topology

- Convenience
  - Virtual topologies may be useful for applications with specific communication patterns - patterns that match an MPI topology structure.
  - For example, a Cartesian topology might prove convenient for an application that requires 4-way nearest neighbor communications for grid based data.

- Communication Efficiency
  - Some hardware architectures may impose penalties for communications between successively distant "nodes".
  - A particular implementation may optimize process mapping based upon the physical characteristics of a given parallel machine.
  - The mapping of processes into an MPI virtual topology is dependent upon the MPI implementation, and may be totally ignored.

Example

- A simplified mapping of processes into a Cartesian virtual topology

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)
12 (3,0)	13 (3,1)	14 (3,2)	15 (3,3)

Exercise 4

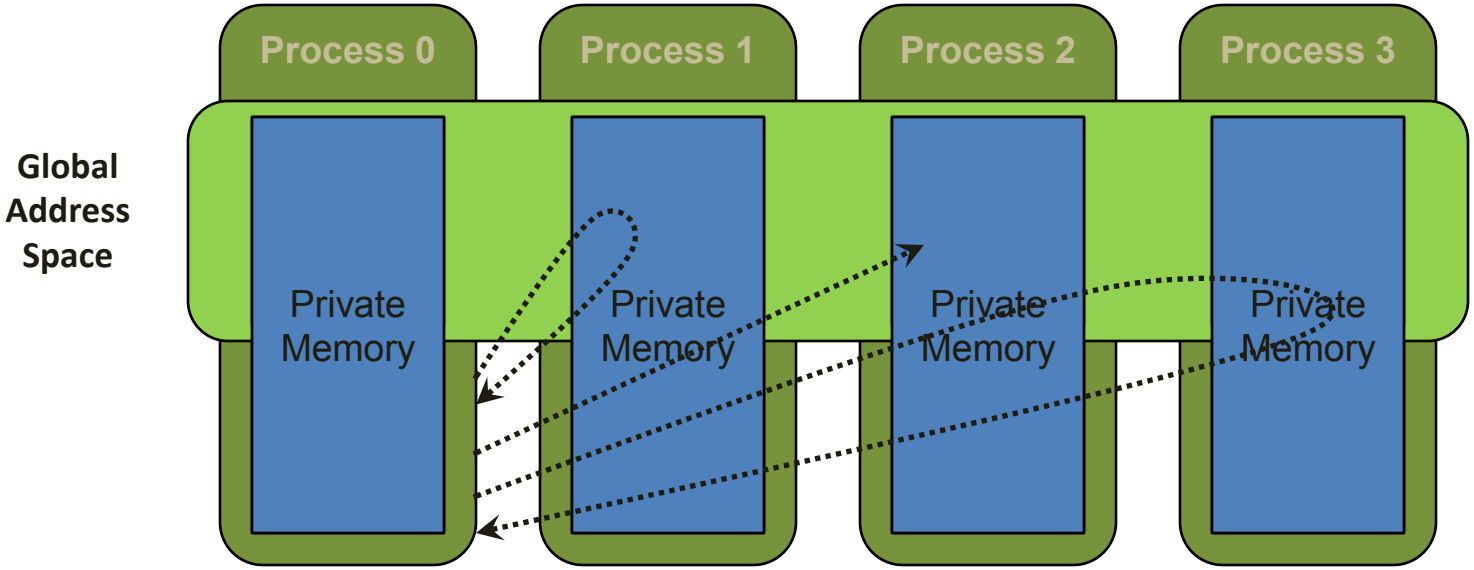
- Run exercise4 which creates a 4 x 4 Cartesian topology from 16 processors and have each process exchange its rank with four neighbors.

Sample program output: (partial)

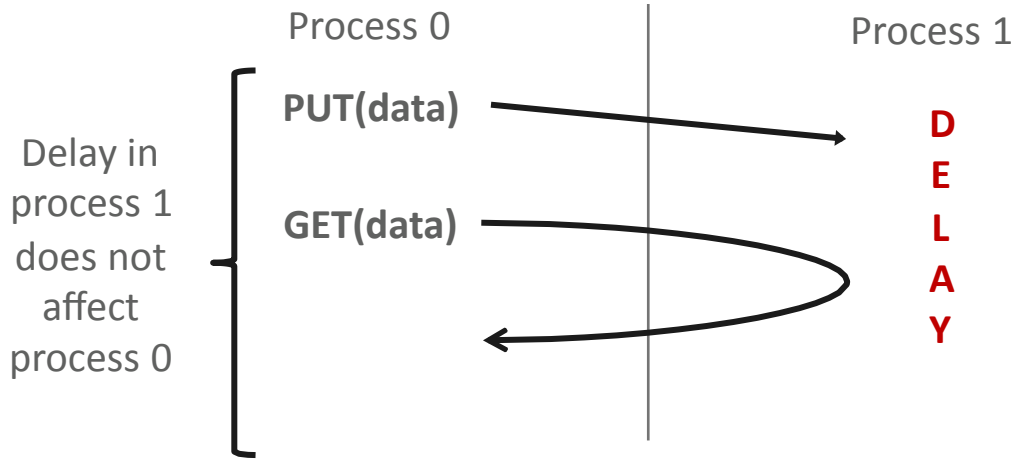
```
rank=   0 coords=  0 0 neighbors(u,d,l,r)=  -1  4 -1  1
rank=   0              inbuf(u,d,l,r)=  -1  4 -1  1
rank=   8 coords=  2 0 neighbors(u,d,l,r)=   4 12 -1  9
rank=   8              inbuf(u,d,l,r)=   4 12 -1  9
rank=   1 coords=  0 1 neighbors(u,d,l,r)=  -1  5  0  2
rank=   1              inbuf(u,d,l,r)=  -1  5  0  2
...
...
rank=   3 coords=  0 3 neighbors(u,d,l,r)=  -1  7  2 -1
rank=   3              inbuf(u,d,l,r)=  -1  7  2 -1
rank=  11 coords=  2 3 neighbors(u,d,l,r)=   7 15 10 -1
rank=  11              inbuf(u,d,l,r)=   7 15 10 -1
rank=  10 coords=  2 2 neighbors(u,d,l,r)=   6 14  9 11
rank=  10              inbuf(u,d,l,r)=   6 14  9 11
rank=   9 coords=  2 1 neighbors(u,d,l,r)=   5 13  8 10
rank=   9              inbuf(u,d,l,r)=   5 13  8 10
```

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



Comparing One-sided and Two-sided Programming



Irregular Communication Patterns with RMA (remote memory access)

- If communication pattern is not known a priori, the send-recv model requires an extra step to determine how many sends-recvs to issue
- RMA, however, can handle it easily because only the origin or target process needs to issue the put or get call
- This makes dynamic communication easier to code in RMA
- Can be significantly faster than send/receive on systems with hardware support for remote memory access, such as shared memory systems

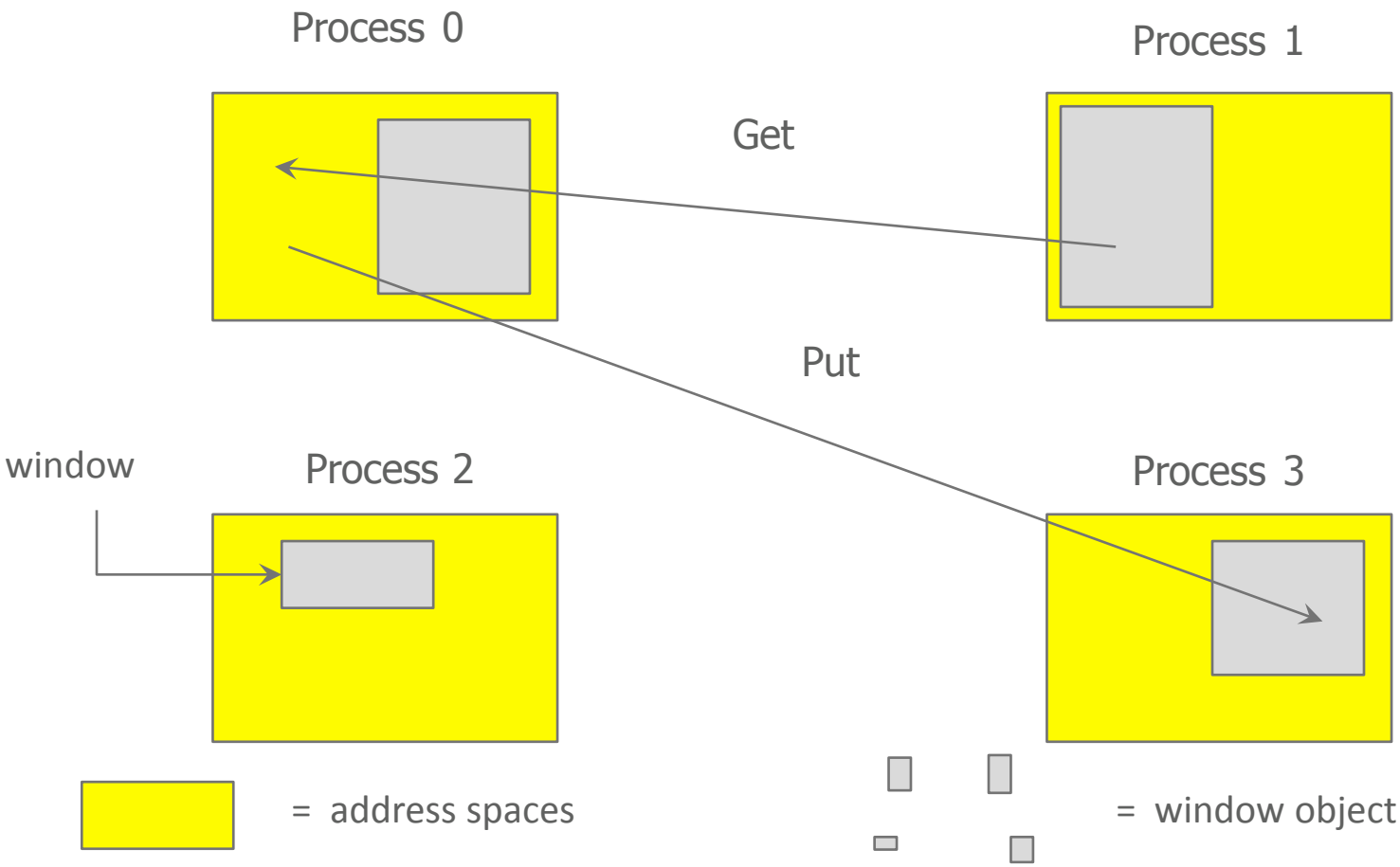
What we need to know in MPI RMA?

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

Creating Public Memory

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

Remote Memory Access Windows and Window Objects



Basic RMA Functions for Communication

- MPI_Win_create: Exposes local memory to RMA operation by other processes in a communicator
    - Collective operation
    - Creates window object
- MPI_Win_free deallocates window object
- MPI_Put moves data from local memory to remote memory
- MPI_Get retrieves data from remote memory into local memory
- MPI_Accumulate updates remote memory using local values
- Data movement operations are non-blocking
- Subsequent synchronization on window object needed to ensure opera*on is complete

Window creation routines

- MPI_Win_create: You already have an allocated buffer that you would like to make remotely accessible
- MPI_Win_allocate: MPI allocates the memory associated with the window (instead of the user passing allocated memory)
- MPI_Win_create_dynamic: Creates a window without memory attached. User can dynamically attach and detach memory to/from the window by calling MPI_Win_attach and MPI_Win_detach
- MPI_Win_allocate_shared: Creates a window of shared memory (within a node) that can be can be accessed simultaneously by direct load/store accesses as well as RMA ops

MPI_WIN_CREATE_DYNAMIC

```
int MPI_Win_create_dynamic(…, 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

Application can dynamically aHach memory to this window

Application can access data on this window only a]er a memory region has been aHached

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 = (void *) 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 accessibly by all processes in MPI_COMM_WORLD*/

    /* undeclare public memory */
    MPI_Win_detach(win, 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

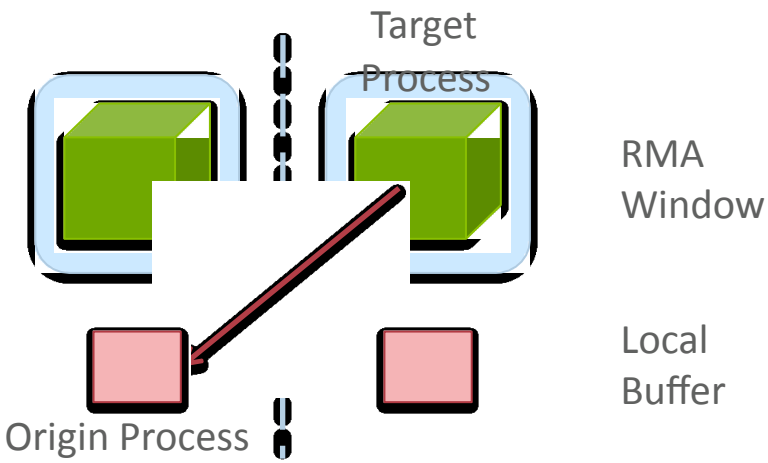
- MPI_GET
- MPI_PUT
- MPI_ACCUMULATE
- MPI_GET_ACCUMULATE
- MPI_COMPARE_AND_SWAP
- MPI_FETCH_AND_OP

Data movement: Get

```
MPI_Get(
    origin_addr,  origin_count, origin_datatype,
    target_rank,
    target_disp,  target_count, target_datatype,
    win)
```

Move data <u>to</u> origin, <u>from</u> target

Separate Data description triples for origin and target

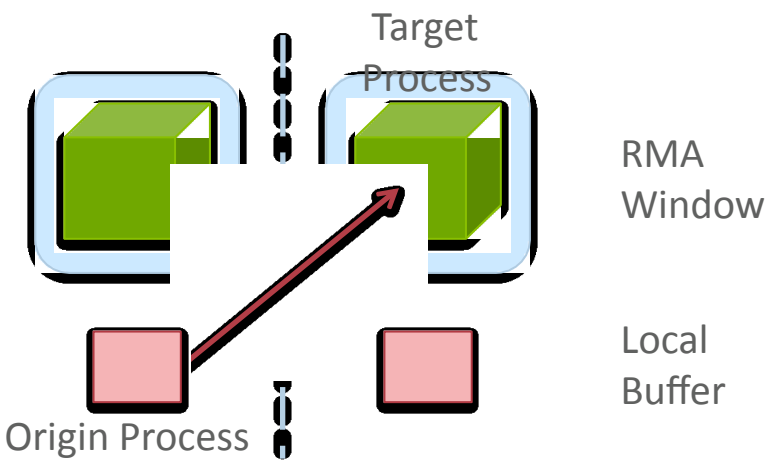


Data movement: Put

```
MPI_Put(
    origin_addr,  origin_count, origin_datatype,
    target_rank,
    target_disp,  target_count, target_datatype,
    win)
```

Move data <u>from</u> origin, <u>to</u> target

Same arguments as MPI_Get



Data aggregation: Accumulate

Like MPI_Put, but applies an MPI_Op instead

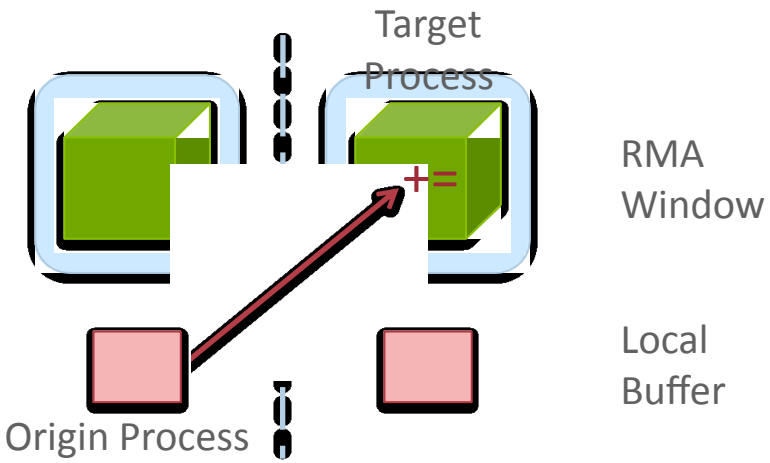
- Predefined ops only, no user-defined!

Result ends up at target buffer

Different data layouts between target/origin OK, basic type elements must match

Put-like behavior with MPI_REPLACE (implements */(a,b)=b*)

- Atomic PUT



Data aggregation: Get Accumulate

Like MPI_Get, but applies an MPI_Op instead

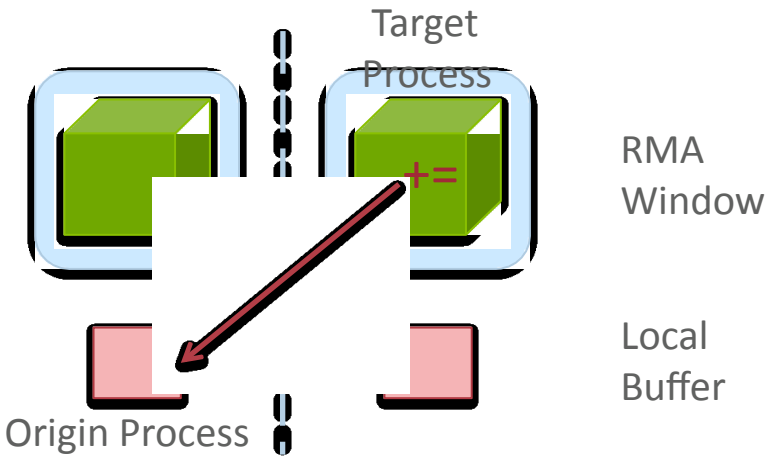
- Predefined ops only, no user-defined!

Result at target buffer; original data comes to the source

Different data layouts between target/origin OK, basic type elements must match

Get-like behavior with MPI_NO_OP

- Atomic GET



RMA Synchronization Models

RMA data visibility

- When is a process allowed to read/write from remotely accessible memory?
- How do I know when data wriHen by process X is available for process Y to read?
- RMA synchronization models provide these capabilities

MPI RMA model allows data to be accessed only within an "epoch"

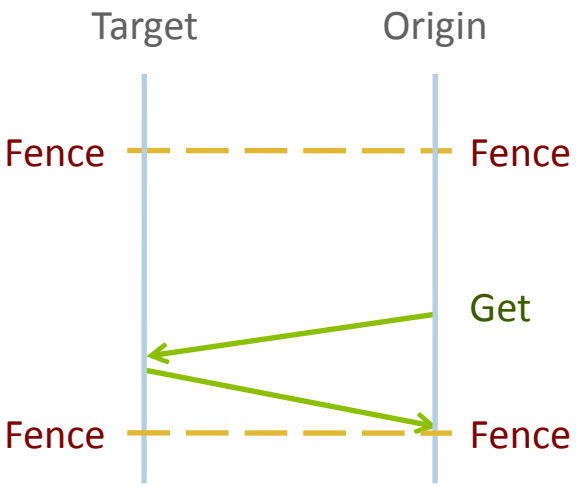
- Three types of epochs possible:
  - Fence (active target)
  - Post-start-complete-wait (active target)
  - Lock/Unlock (passive target)

Data visibility is managed using RMA synchronization primitives

- MPI_WIN_FLUSH, MPI_WIN_FLUSH_ALL
- Epochs also perform synchronization

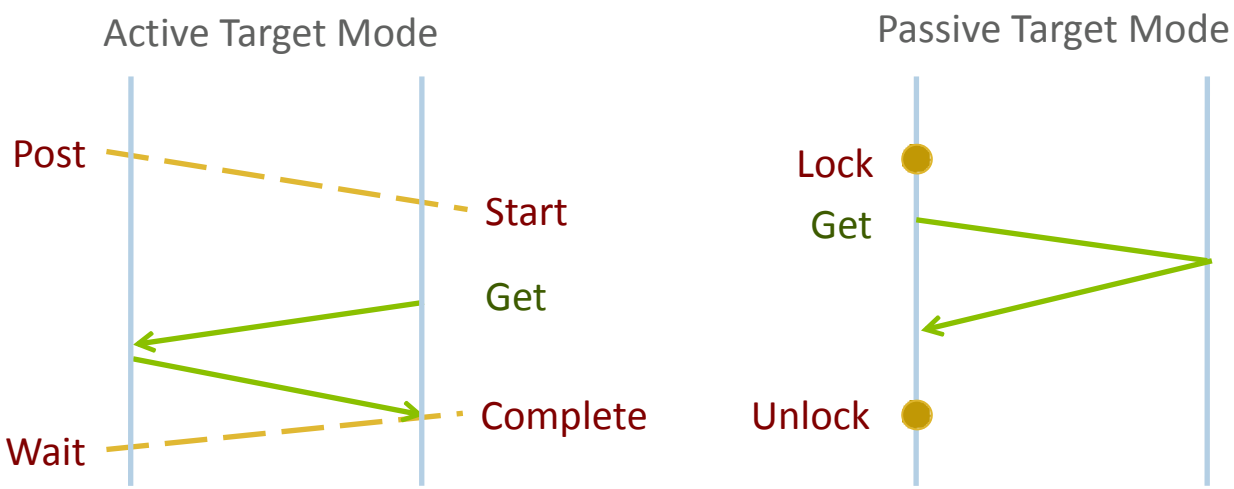
Fence Synchronization

- MPI_Win_fence(assert, win)
- Collective synchronization model -- assume it
- synchronizes like a barrier
- Starts *and* ends access & exposure epochs (usually)



- Everyone does an MPI_WIN_FENCE to open an epoch
- Everyone issues PUT/GET operations to read/write data
- Everyone does an MPI_WIN_FENCE to close the epoch

Lock/Unlock Synchronization



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

Passive Target Synchronization

```
int MPI_Win_lock(int lock_type, int rank, int assert,
      "MPI_Win win)

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

Begin/end passive mode epoch

- Doesn’t function like a mutex, name can be confusing
- Communication operations within epoch are all nonblocking

Lock type

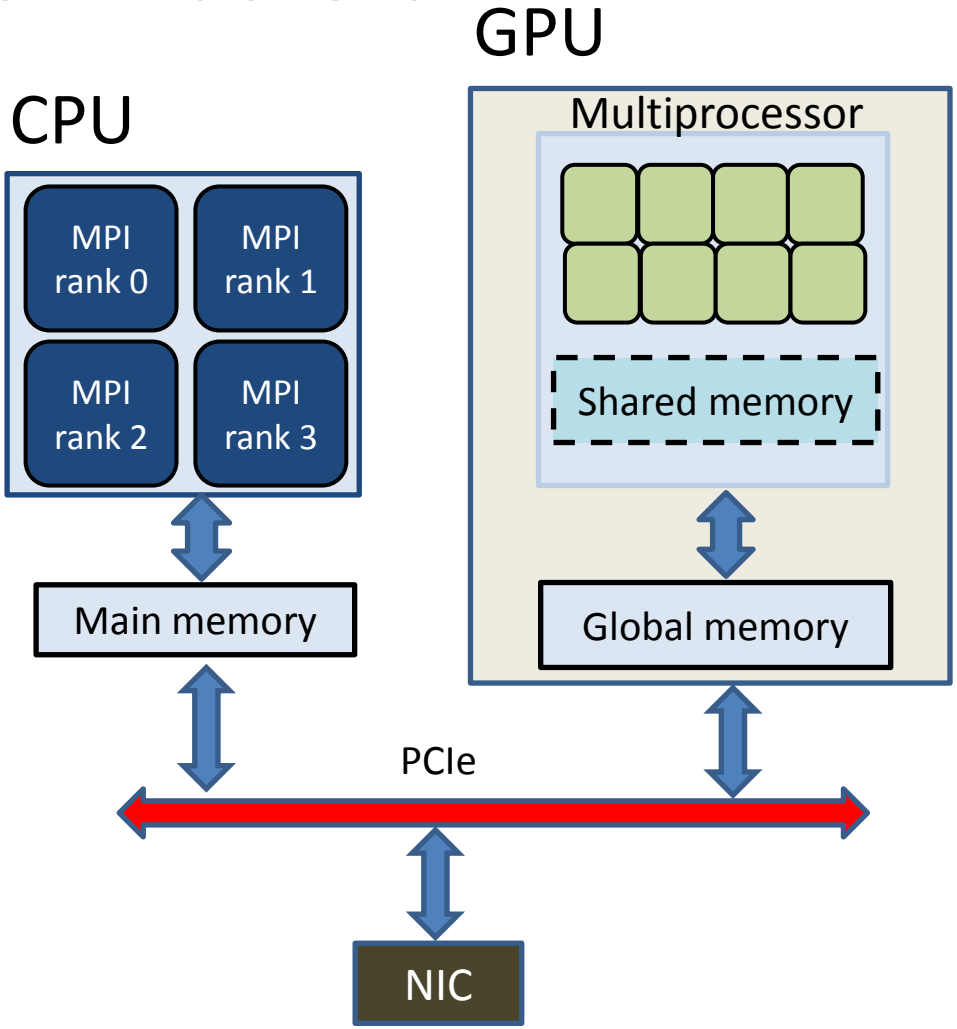
- SHARED: Other processes using shared can access concurrently
- EXCLUSIVE: No other processes can access concurrently

Homework Exercise

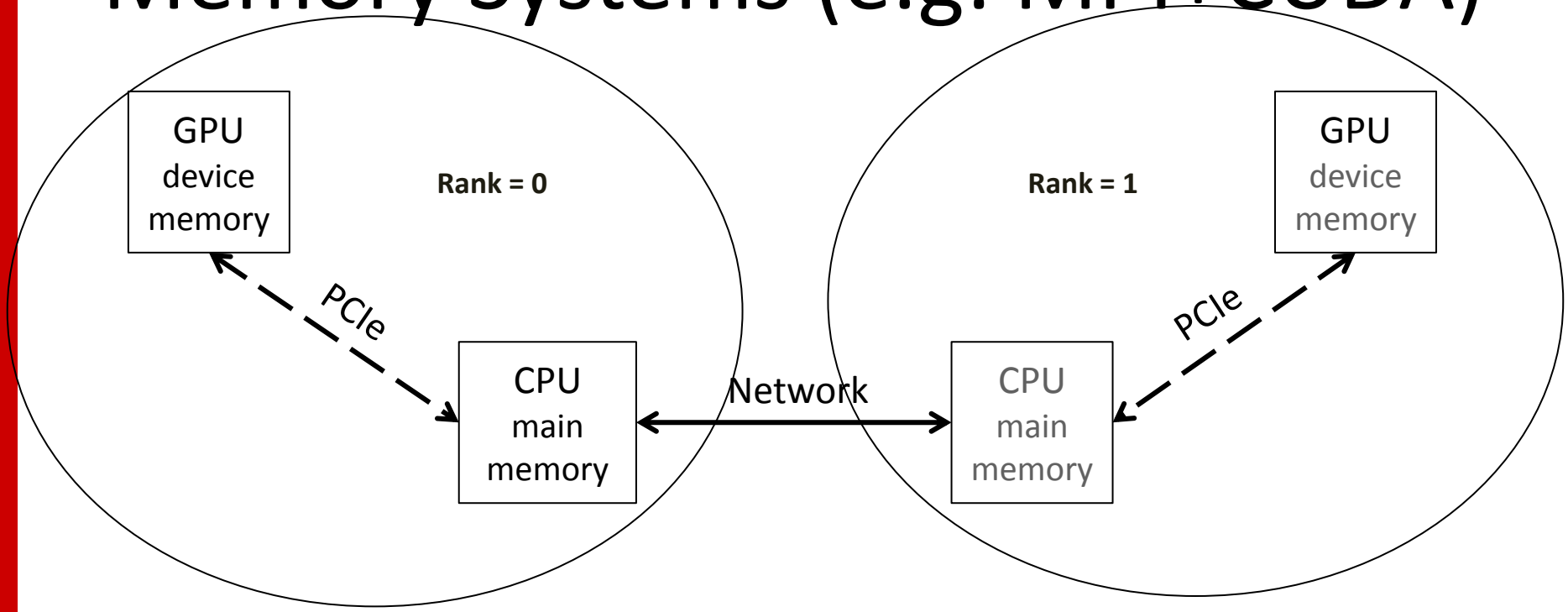
- Create a program with one way communication to exchange process rank value and print the sum of ranks with passive target synchronization
  - Start from exercise5.c (create a dynamic window)
  - Attach the process rank to the window
  - Use MPI_Get_Accumulate to find the sum of ranks
  - Pad the calls with mpi_win_lock and mpi_win_unlock

Heterogeneous Architecture: Accelerator Clusters

- Graphics Processing Units (GPUs)
  - Many-core architecture for high performance and efficiency (FLOPs, FLOPs/Watt, FLOPs/$)
  - Prog. Models: CUDA, OpenCL, OpenACC
  - Explicitly managed global memory and separate address spaces
- CPU clusters
  - Most popular parallel prog. model: Message Passing Interface (MPI)
  - Host memory only
- Disjoint Memory Spaces!



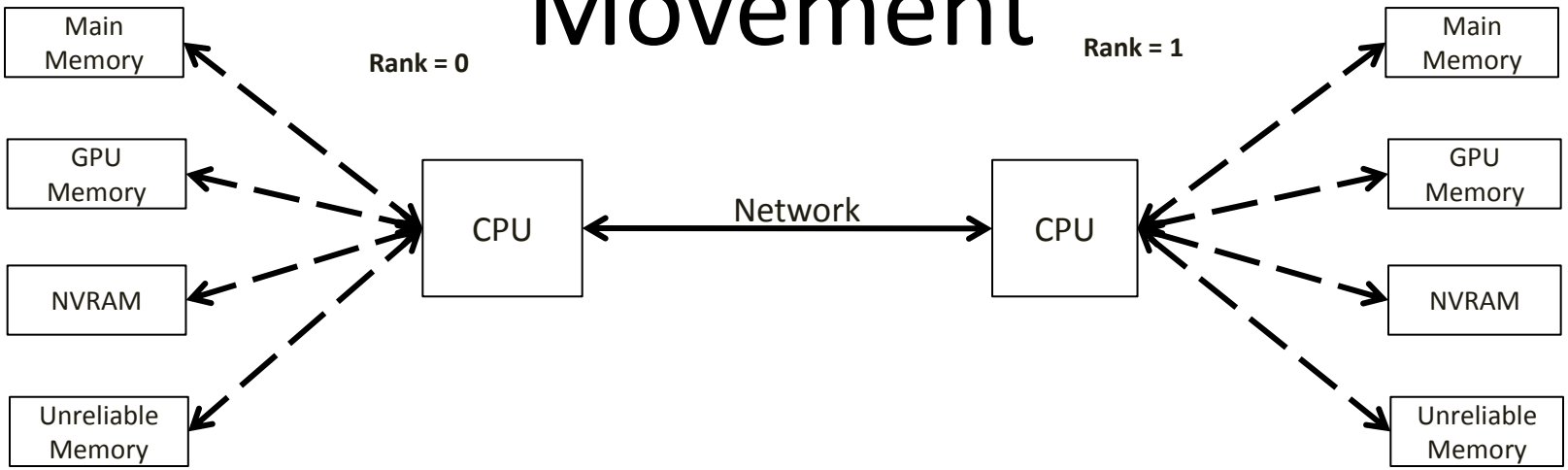
Programming Heterogeneous Memory Systems (e.g: MPI+CUDA)




```
if(rank == 0)
{
  cudaMemcpy(host_buf, dev_buf, D2H)
  MPI_Send(host_buf, .. ..)
}
```

```
if(rank == 1)
{
  MPI_Recv(host_buf, .. ..)
  cudaMemcpy(dev_buf, host_buf, H2D)
}
```

MPI-ACC: A Model for Unified Data Movement



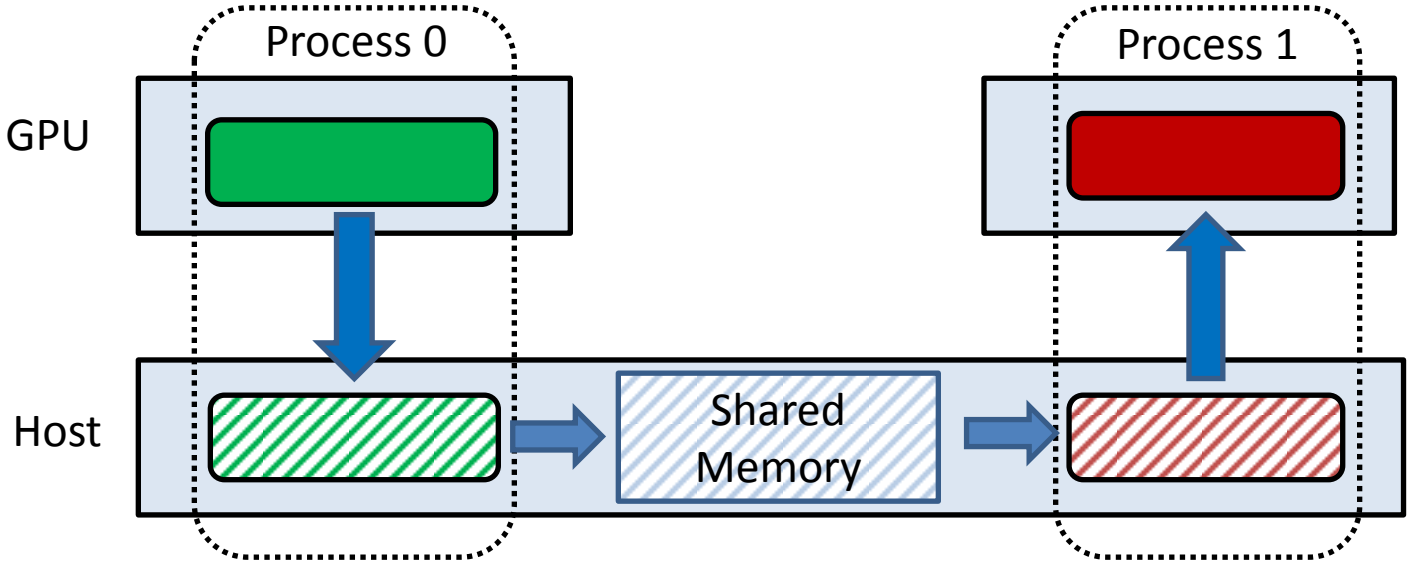

```
if(rank == 0)
{
  MPI_Send(any_buf, .. ..);
}
```

```
if(rank == 1)
{
  MPI_Recv(any_buf, .. ..);
}
```

"MPI-ACC: An Integrated and Extensible Approach to Data Movement in Accelerator-Based Systems", Ashwin Aji, James S. Dinan, Darius T. Buntinas, Pavan Balaji, Wu-chun Feng, Keith R. Bisset and Rajeev S. Thakur. IEEE International Conference on High Performance Computing and Communications (HPCC), 2012

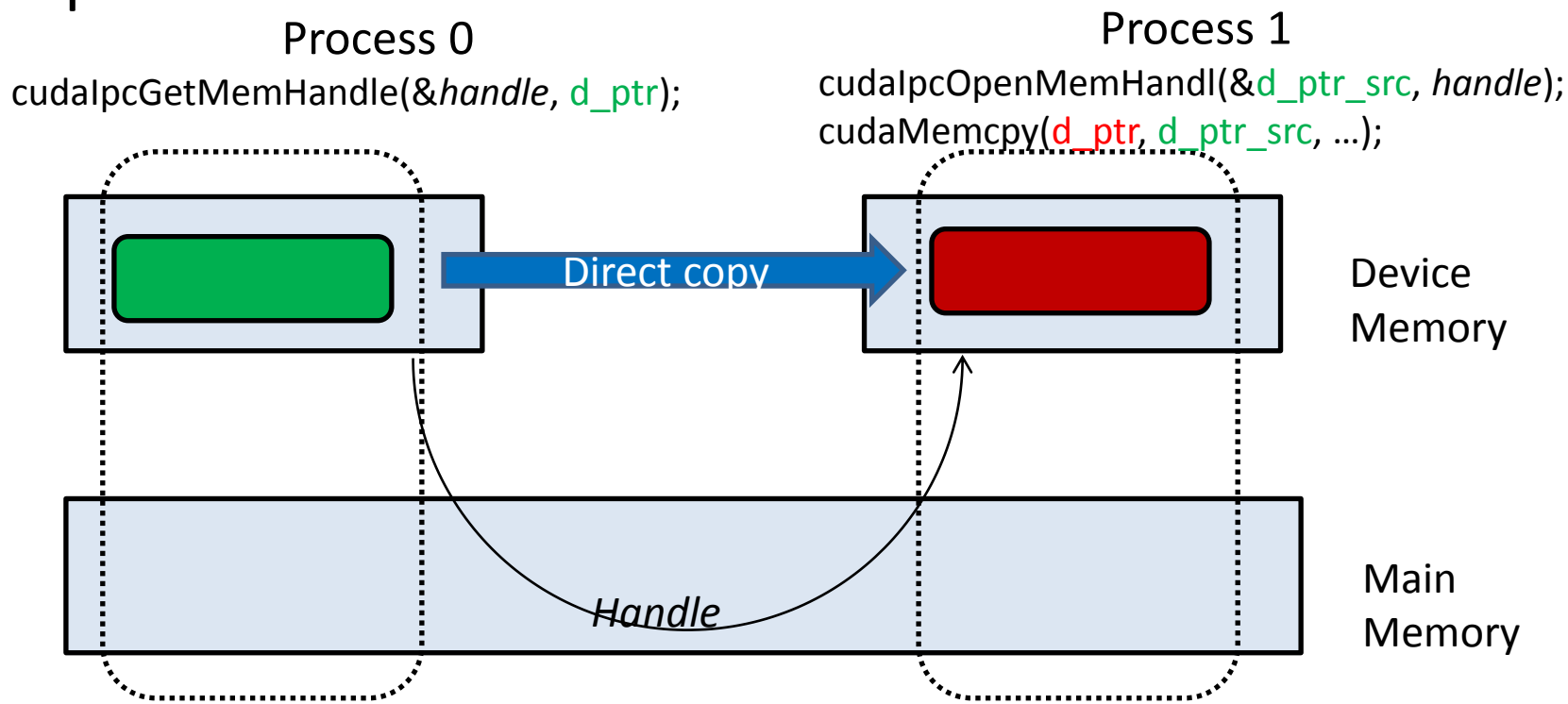
Traditional Intranode Communication

- Communication without accelerator integration
  - 2 PCIe data copies + 2 main memory copies
  - Transfers are serialized



GPU Direct and CUDAIPC optimizations

- GPUDirect: DMA-driven peer GPU copy
- CUDAIPC: exporting a GPU buffer to a different process



Mapping, Ranking, and Binding

- **MPI implementation dependent**
- **mapping**
  - Assigns a default location to each process
  - mpirun -hostfile myhostfile ./a.out (by node)
- **ranking**
  - Assigns an MPI_COMM_WORLD rank value to each process
  - flexibility in the relative placement of MPI processes

Process Binding

- **binding**
  - Constrains each process to run on specific processors
- To avoid suboptimal process placement by OS

  --report-bindings
  - mpirun --report-bindings -np 16 a.out

  --bind-to <foo>
  - Bind processes to the specified object, defaults to core. Supported options include slot, hwthread, core, l1cache, l2cache, l3cache, socket, numa, board, and none.
- Exercise – get the binding report for exercise4

Binding report

```
MCW rank 12 bound to socket 0[core 6[hwt 0-1]]: [../../../../../../BB/../../..][../../../../../../../../../..]
MCW rank 13 bound to socket 1[core 16[hwt 0-1]]: [../../../../../../../../../..][../../../../../../BB/../../..]
MCW rank 14 bound to socket 0[core 7[hwt 0-1]]: [../../../../../../../BB/../..][../../../../../../../../../..]
MCW rank 15 bound to socket 1[core 17[hwt 0-1]]: [../../../../../../../../../..][../../../../../../../BB/../..]
MCW rank 0 bound to socket 0[core 0[hwt 0-1]]: [BB/../../../../../../../../..][../../../../../../../../../..]
MCW rank 1 bound to socket 1[core 10[hwt 0-1]]: [../../../../../../../../../..][BB/../../../../../../../../..]
MCW rank 2 bound to socket 0[core 1[hwt 0-1]]: [../BB/../../../../../../../..][../../../../../../../../../..]
MCW rank 3 bound to socket 1[core 11[hwt 0-1]]: [../../../../../../../../../..][../BB/../../../../../../../..]
MCW rank 4 bound to socket 0[core 2[hwt 0-1]]: [../../BB/../../../../../../..][../../../../../../../../../..]
MCW rank 5 bound to socket 1[core 12[hwt 0-1]]: [../../../../../../../../../..][../../BB/../../../../../../..]
MCW rank 6 bound to socket 0[core 3[hwt 0-1]]: [../../../BB/../../../../../..][../../../../../../../../../..]
MCW rank 7 bound to socket 1[core 13[hwt 0-1]]: [../../../../../../../../../..][../../../BB/../../../../../..]
MCW rank 8 bound to socket 0[core 4[hwt 0-1]]: [../../../../BB/../../../../..][../../../../../../../../../..]
MCW rank 9 bound to socket 1[core 14[hwt 0-1]]: [../../../../../../../../../..][../../../../BB/../../../../..]
MCW rank 10 bound to socket 0[core 5[hwt 0-1]]: [../../../../../BB/../../../..][../../../../../../../../../..]
MCW rank 11 bound to socket 1[core 15[hwt 0-1]]: [../../../../../../../../../..][../../../../../BB/../../../..]
```

Current Situation with Production Applications (1)

- The vast majority of DOE's production parallel scientific applications today use MPI
  - Increasing number use (MPI + OpenMP) hybrid
  - Some exploring (MPI + accelerator) hybrid
- Today's largest systems in terms of number of regular cores (excluding GPU cores)

Sequoia (LLNL)	1,572,864 cores
Mira (ANL)	786,432 cores
K computer	705,024 cores
Jülich BG/Q	393,216 cores
Blue Waters	386,816 cores
Titan (ORNL)	299,008 cores

- **MPI already runs in production on systems with up to 1.6 million cores**

Courtesy: Pavan Balaji, ANL

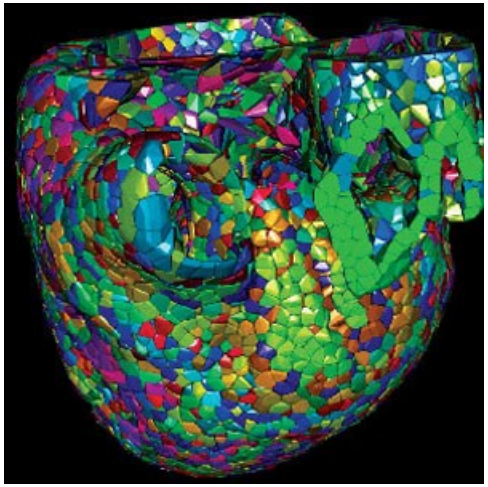
Current Situation with Production Applications (2)

- IBM has successfully scaled the LAMMPS application to over 3 million MPI ranks
- Applications are running at scale on LLNL's Sequoia and achieving **12 to 14 petaflops** *sustained* performance

- HACC cosmology code from Argonne (PI: Salman Habib) achieved **14 petaflops** on Sequoia
  - Ran on full Sequoia system using MPI + OpenMP hybrid
  - Used 16 MPI ranks * 4 OpenMP threads on each node, which matches the hardware architecture: 16 cores per node with 4 hardware threads each
  - http://www.hpcwire.com/hpcwire/2012-11-29/sequoia_supercomputer_runs_cosmology_code_at_14_petaflops.html
  - SC12 Gordon Bell prize finalist

Courtesy: Pavan Balaji, ANL

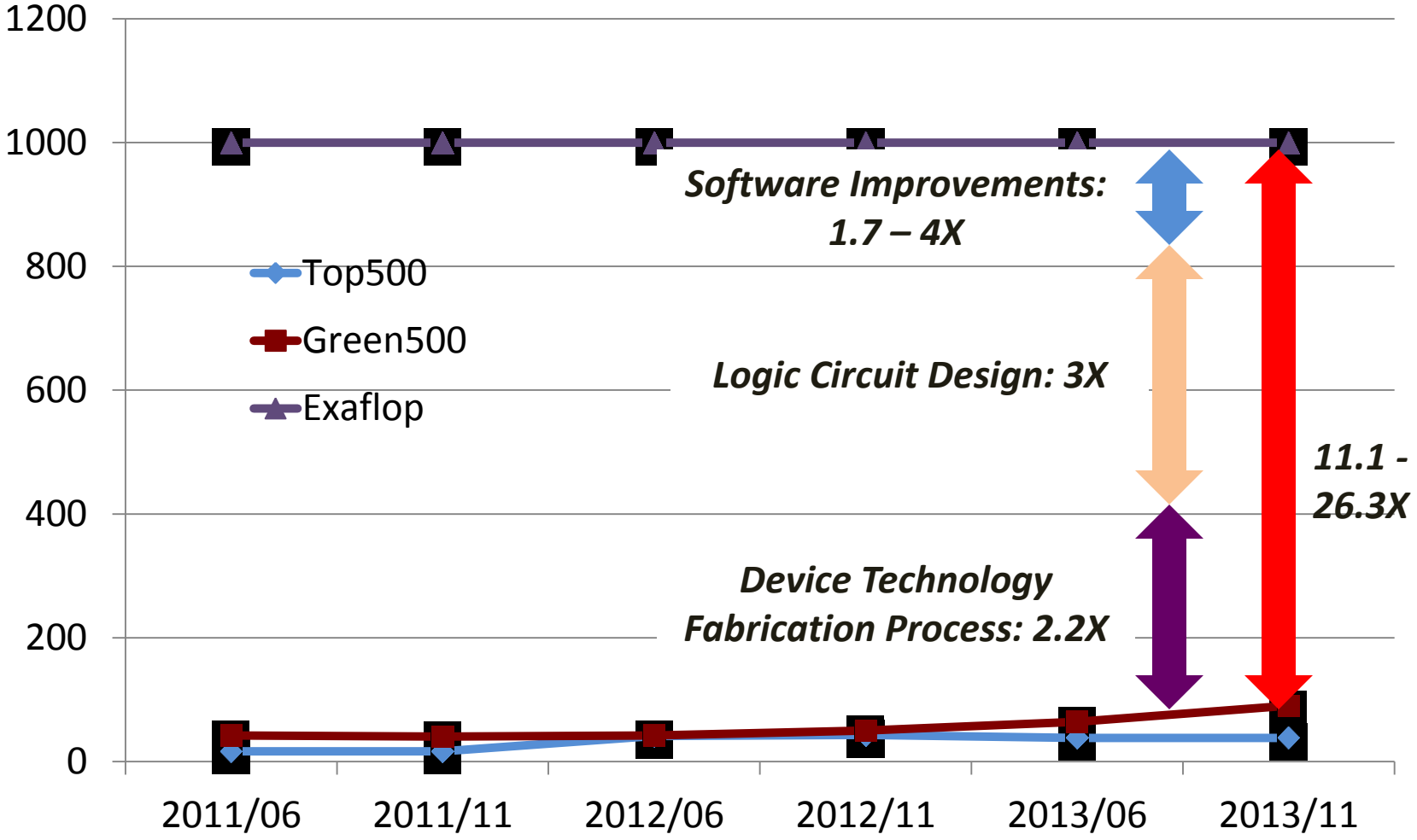
Current Situation with Production Applications (3)



- Cardioid cardiac modeling code (IBM & LLNL) achieved **12 petaflops** on Sequoia
  - Models a beating human heart at near-cellular resolution
  - Ran at scale on full system (96 racks)
  - Used MPI + threads hybrid: 1 MPI rank per node and 64 threads
  - OpenMP was used for thread creation only; all other thread choreography and synchronization used custom code, not OpenMP pragmas
  - http://nnsa.energy.gov/mediaroom/pressreleases/sequoia112812
  - SC12 Gordon Bell Prize finalist
- And there are other applications running at similar scales…

Courtesy: Pavan Balaji, ANL

On the path to Exascale



Courtesy: Pavan Balaji, ANL

MPI in the Exascale Era

- Under a lot of scrutiny (good!)
  - Lots of myths floating around (bad!)
- Push to get new programming models designed and developed for exascale
- The truth is that MPI today is a new programming model (compared to 2004) , and MPI in 2020 will be a new programming model (compared to today)
- Strengths of MPI
  - Composability
    - Ability to build tools and libraries above and around MPI
    - No “do everything under the sun” attitude
  - Continuous evolution
    - The standard incorporates best research ideas

Courtesy: Pavan Balaji, ANL

References

The Standard itself: http://www.mpi-forum.org
General MPI reference: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

Books:

- Using MPI: Portable Parallel Programming with the Message-Passing Interface, by Gropp, Lusk, and Skjellum, MIT Press, 1994.
- MPI: The Complete Reference, by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1996.
- Designing and Building Parallel Programs, by Ian Foster, Addison-Wesley, 1995.
- Parallel Programming with MPI, by Peter Pacheco, Morgan-Kaufmann, 1997.
- MPI: The Complete Reference Vol 1 and 2,MIT Press, 1998 (Fall).

References

Other web resources:

http://static.msi.umn.edu/tutorial/scicomp/general/MPI/content_communicator.html

https://www.cs.kent.ac.uk/people/staff/trh/MPI/mpitutorial.pdf

http://condor.cc.ku.edu/~grobe/docs/intro-MPI.shtml

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

https://computing.llnl.gov/tutorials/mpi/

Support

- CACDS support
  - http://support.cacds.uh.edu/

- Amit Amritkar
  - PGH 223
  - aramritkar@uh.edu
  - (713)743-7547

Thank you

- Please fill the course assessment forms, *just click the home icon in your browser*

- Email yourselves the **slides & example programs** from
  /home/hpc_userXX/mpi_2.zip
  /home/hpc_userXX/mpi_2.pdf

- Upper menu > System > **Log Out** hpc_user...

Upcoming training events at CACDS

Visualization with Paraview II, 3/27, 2pm, PGH 235

Intro to high performance numerical libraries, 3/31, 10am, PGH 200
GPGPU parallel programming with OpenACC I, 3/31, 2pm, PGH 235