KOÇ UNIVERSITY

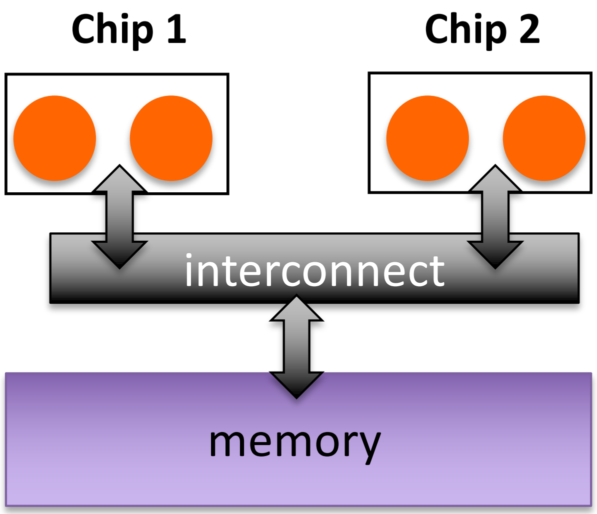
Message Passing Programming Model- Intro

Didem Unat

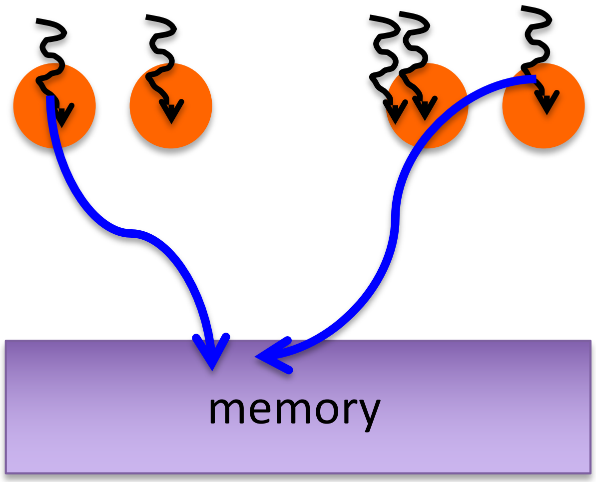
COMP 429/529 Parallel Programming

Shared-Memory Programming Model

- More correct name: Shared-address space programming
  - Threads communicate through shared memory as opposed to messages
  - Threads coordinate through synchronization (also through shared memory).

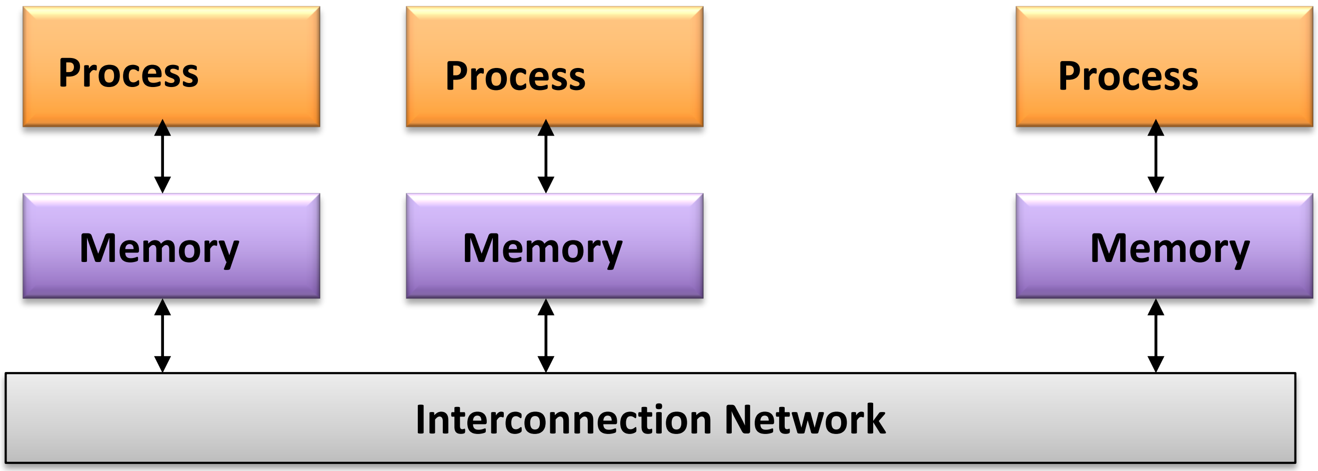


Recall shared memory system
(can be either UMA, NUMA)



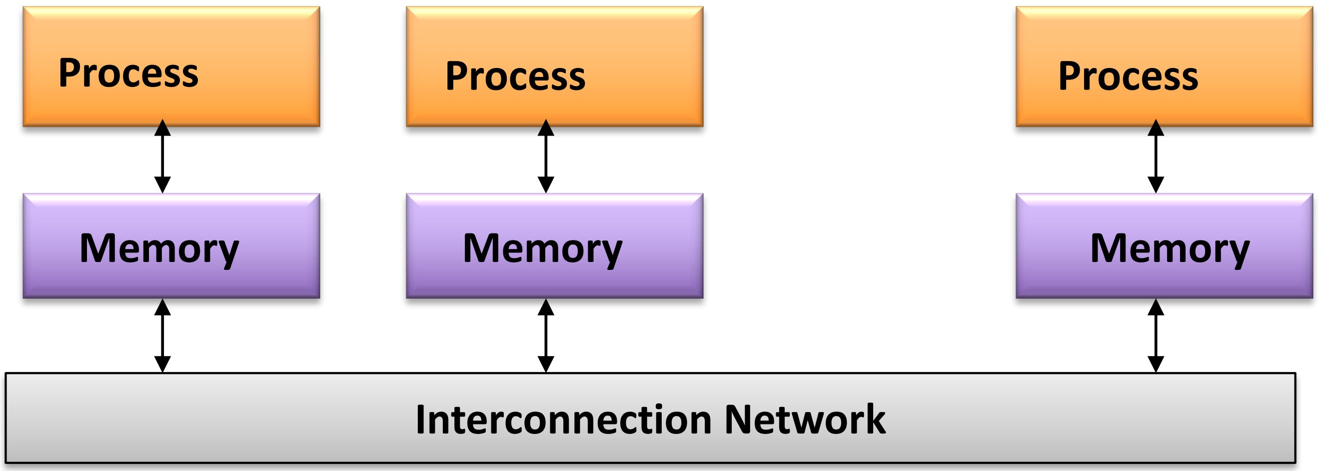
Message Passing Programming Model

- Programs execute as a set of P processes (user specifies P)
- Each process has its own private address space
  - Processes share data by *explicitly* sending and receiving information (message passing)
  - Coordination is built into message passing primitives (message send and message receive)



Message Passing Programming Model

- Each data element must belong to one of the partitions of the space; hence data must be explicitly partitioned and placed.
- Most message-passing programs are written using single program multiple data (SPMD model)



Message Passing

- All communication, synchronization require library calls
  - No shared variables
  - Program runs on a single processor just like any uniprocessor program, except for calls to message passing library
  - Barriers
  - No locks because no shared variables to protect
- Isolation of separate address spaces

  + No data races, but communication errors possible

  - Complexity and code size growth!

Send and Receive Operations

- The prototypes of these operations are as follows:

```
send(void *sendbuf, int num_elems, int dest)
receive(void *recvbuf, int num_elems, int source)
```

- Consider the following code segments:

```
//P0
a = 100;
send(&a, 1, P1);
a = 0;
```

```
//P1
receive(&a, 1, P0);
printf("%d\n", a);
```

- The semantics of the send operation require that the value received by process P1 must be 100 as opposed to 0.
- This motivates the design of the send and receive protocols.

Message Passing and MPI

- **MPI (Message Passing Interface)**
  - A specification of message passing programming model
  - Library implementations are available for conventional sequential languages (Fortran, C, C++)
  - Portable
  - Low-level but universal
  - Predominant library for implementing message passing on large clusters
  - Scales on millions of cores

Like OpenMP, MPI arose as a standard to replace a large number of proprietary message passing libraries.

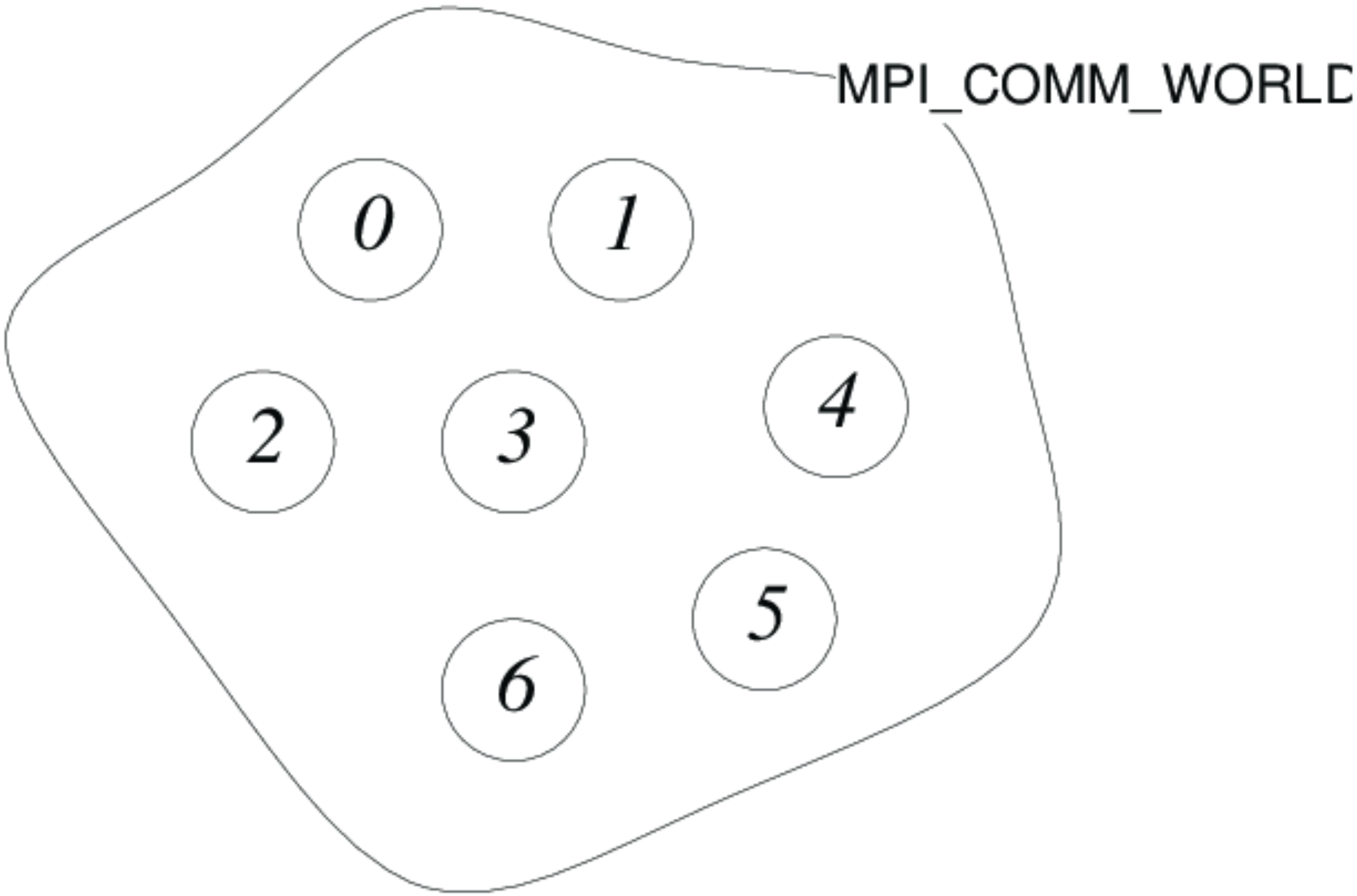
Environment

- Two important questions that arise in every parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?

- MPI provides functions to answer these questions:
  - **`MPI_Comm_size`** reports the number of processes.
  - **`MPI_Comm_rank`** reports the *rank (process ID)*, a number between 0 and size-1, identifying the calling process

Slide source: Bill Gropp

MPI Communication World



MPI_Comm_size()

MPI_Comm_rank()

Hello World in MPI

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

int main( int argc, char *argv[] )
{
    int rank=0, size=1;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( ”Hello from process %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Compilation and Execution

wrapper script to compile

source file

```
mpicc  -o  mpi_hello  mpi_hello.c
```

create this executable file name

(as opposed to default a.out)

Compilation

Execution

```
mpiexec  -n  <number of processes>   <executable>

mpiexec  -n  4  ./mpi_hello
```

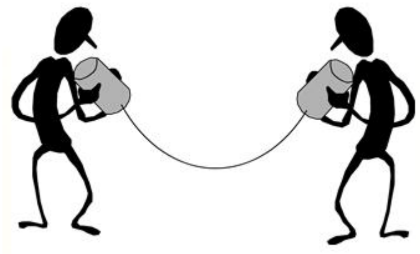
run with 4 processes

MPI in C++

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI::Init(argc, argv);
    rank = MPI::COMM_WORLD.Get_rank();
    size = MPI::COMM_WORLD.Get_size();
    std::cout<< ”Hello from process”<<rank <<”of ”<<size <<"\n";
    MPI::Finalize();
    return 0;
}
```

Message Types

- Two kinds of communication patterns
  - **Pairwise or point-to-point**: A message is sent from a specific sending process (point a) to a specific receiving process (point b)
    - Send/Receive



  - **Collective communication** involving multiple processors
    - Move data: Broadcast, Scatter/gather
    - Compute and move: Reduce, AllReduce

MPI Send()

`MPI_SEND(start, count, datatype, dest, tag, comm)`

- The message buffer is described by (**`start, count, datatype`**).
- The target process is specified by **`dest`**, which is the rank of the target process in the communicator specified by **`comm`**.
- When this function returns, the message is 'in transit' and the buffer can be reused. The message may not have been received by the target process.
- **Tags** assist the receiving process in identifying the message

MPI Receive()

`MPI_RECV(start, count, datatype, source, tag, comm, status)`

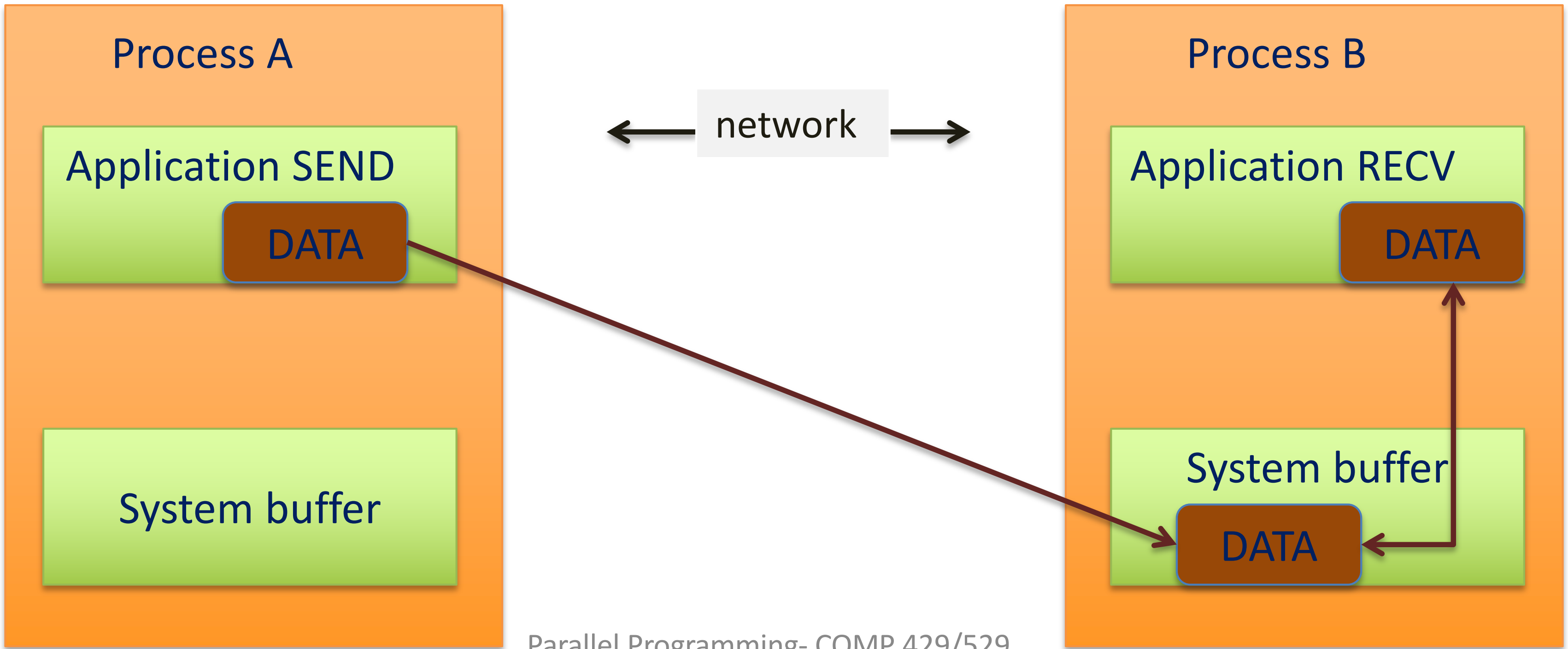
- Blocks until a matching (both **`source`** and **`tag`**) message is received from the system
- When the function returns, the buffer can be used
- **`source`** is rank in communicator specified by **`comm`**, or **`MPI_ANY_SOURCE`**
- **`tag`** is a tag to be matched on or **`MPI_ANY_TAG`**
- receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error
- **`status`** contains further information (e.g. size of message)

Buffering of Messages

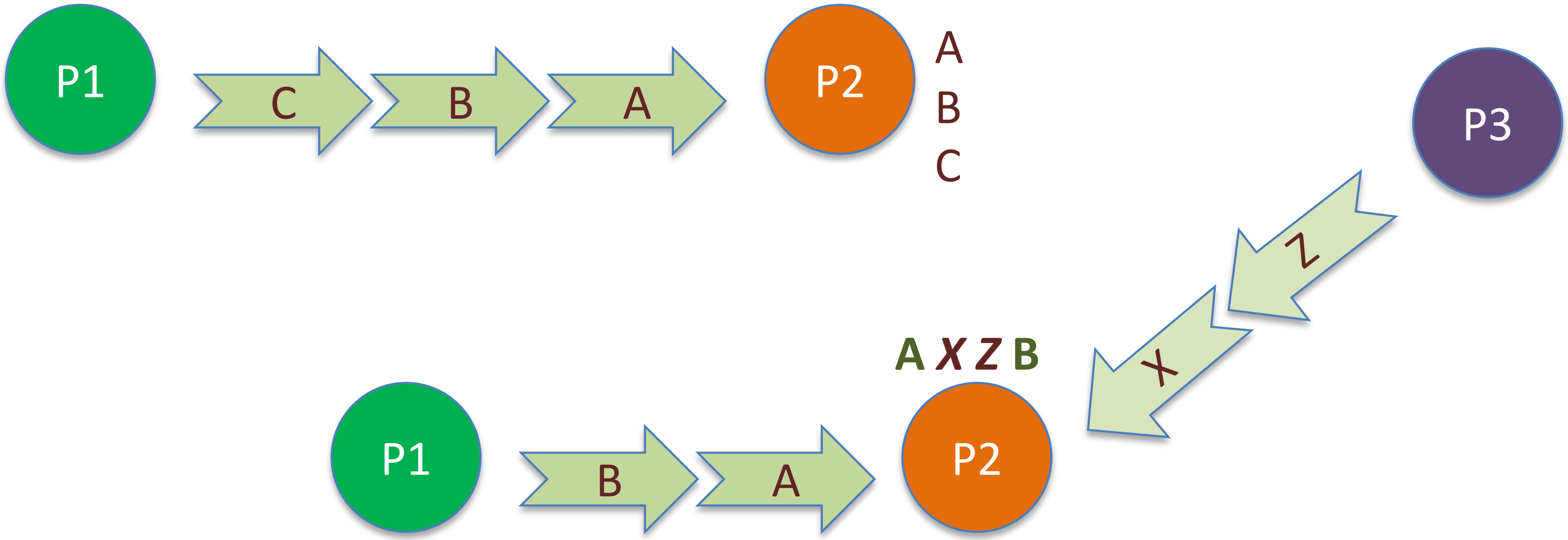
- Consider the following two cases:
  - A send operation occurs 5 seconds before the receive is ready - where is the message while the receive is pending?
  - Multiple sends arrive at the same receiving task which can only accept one send at a time - what happens to the messages that are "backing up"?
- The MPI implementation (not the MPI standard) decides what happens to data in these types of cases.

Point-to-Point Communication

- A single pair of communicating processes copy data between address space
- There are buffers at send and receive ends
  - Managed by the MPI library (not by the programmer)
  - The sender simply copies the data into the designated buffer and returns after the copy operation has been completed



Point-to-Point Comm.



MPI guarantees that the messages arrive in order from the same sender but not multiple senders

- Message Tags
  – Helps matching the messages from the same receiver

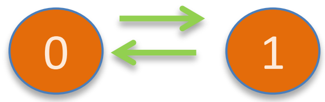
MPI Datatypes

- The data in a message to send or receive is described by a triple (address, count, datatype)
- The message unit size depends on the type of the data
  - The length in bytes is sizeof(type) * # of elements

- An MPI datatype can correspond to a data type from the language (e.g., MPI_INT, MPI_DOUBLE, MPI_FLOAT)
- There are MPI functions to construct custom datatypes (e.g. structs)

Ping-Pong Example

- Task 0 pings task 1 and awaits return ping




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

main(int argc, char *argv[])  {

int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

Get the number of processes

Get my process ID

Ping-Pong (cont.)

```
if (rank == 0) {
  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
}
```

message

1 char (byte)

Destination PID

Source PID

Status

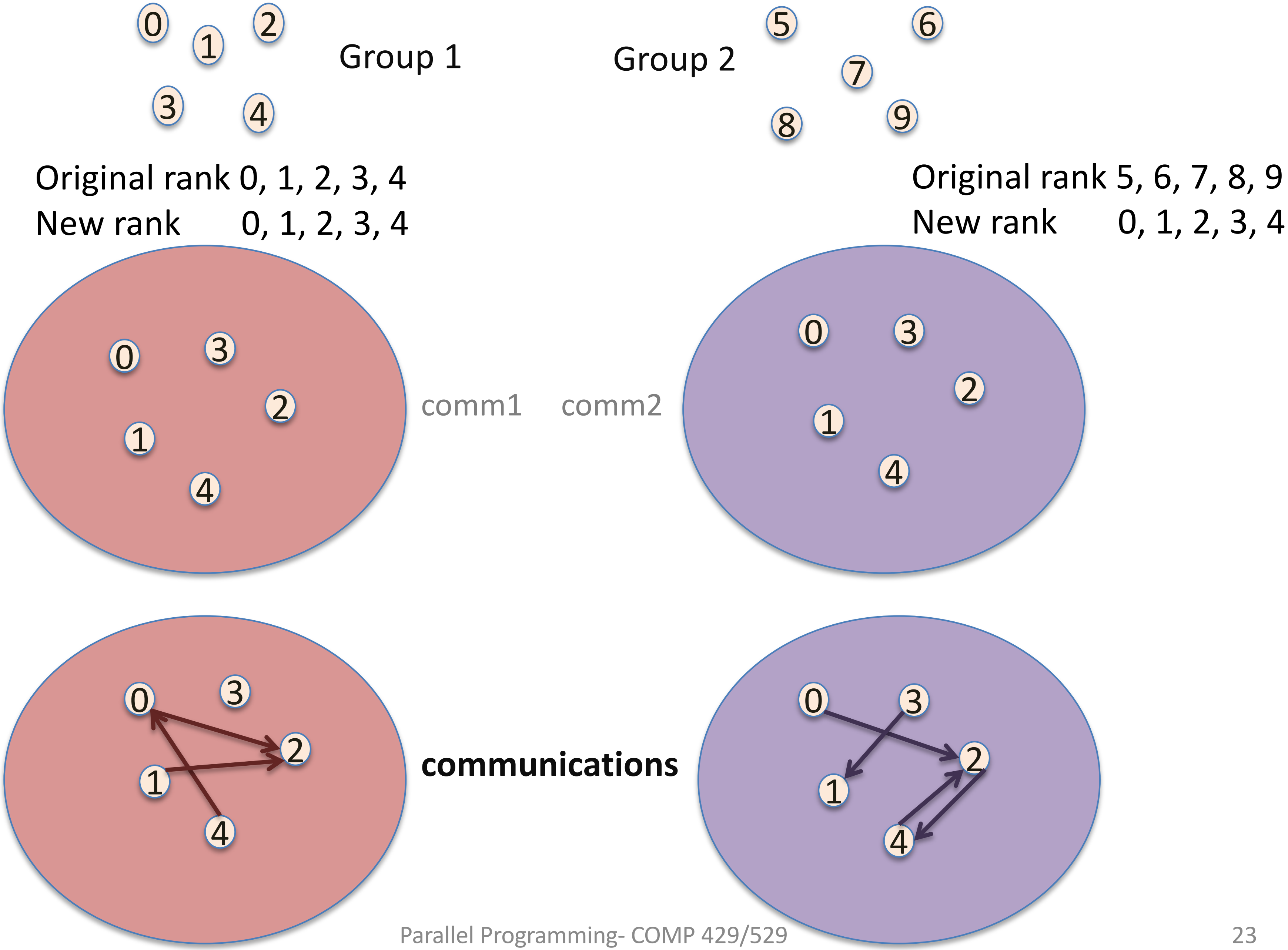
```
else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
}

rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
printf("Task %d: Received %d char(s) from task %d with tag %d \n",
       rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);

MPI_Finalize();
}
```

Returns the precise count of data items received

Communicators

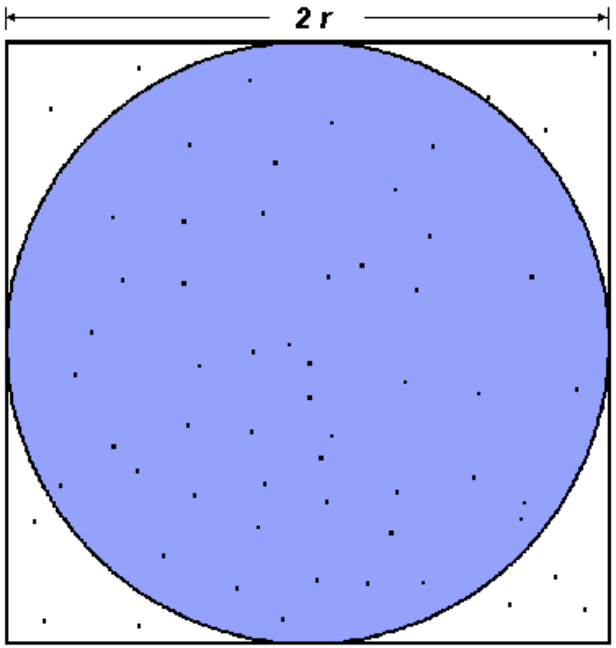
- How to organize processes
  - Processes can be collected into groups
  - Each message is sent in a context, and must be received in the same context
    - Provides necessary support for libraries
  - A group and context together form a communicator
  - A process is identified by its rank in the group associated with a communicator
- **`MPI_COMM_WORLD`** is the default communicator whose group contains all initial processes

Communicators



Pi Calculation Example

```
npoints = 10000
circle_count = 0

do j = 1,npoints
        generate 2 random numbers between 0 and 1
        xcoordinate = random1
        ycoordinate = random2
        if (xcoordinate, ycoordinate) inside circle
          then circle_count = circle_count + 1
end do

PI = 4.0*circle_count/npoints
```

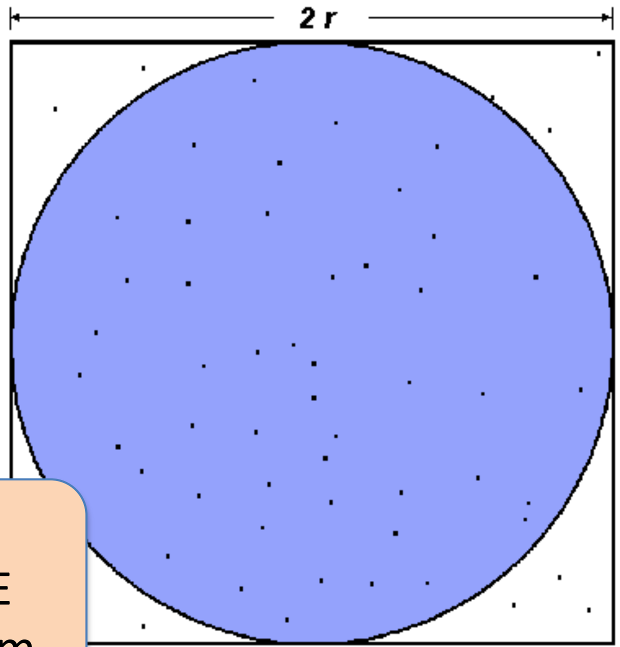


Area (circle) = πr^2
Area (square) = $(2r)^2$
π = 4 Area(circle) / Area(square)

Parallel Pi Calculation Example

```
npoints = 10000
circle_count = 0

p = number of tasks
darts = npoints/p

find out if I am MASTER or WORKER

do j = 1, darts
  generate 2 random numbers between 0 and 1
  xcoordinate = random1
  ycoordinate = random2
  if (xcoordinate, ycoordinate) inside circle
  then circle_count = circle_count + 1
end do

if I am MASTER
  receive from WORKERS their circle_counts
  compute PI (use MASTER and WORKER calculations)
else if I am WORKER
  send to MASTER circle_count
endif
```

Can use MPI_REDUCE collective comm.



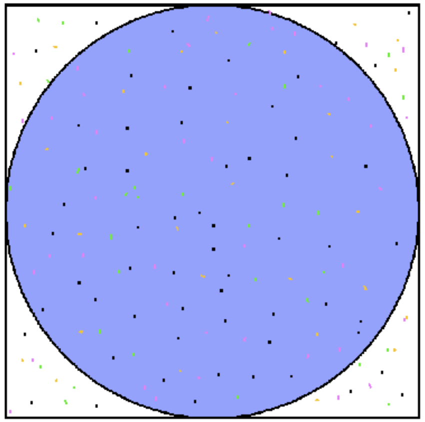
Area (circle) = πr^2

Area (square) = $(2r)^2$

π = 4 Area(circle) / Area(square)

Pi Calculation - Example

```
#define MASTER 0
. . .
double localpi = compute(DARTS);

rc = MPI_Reduce(&localpi, &pisum, 1, MPI_DOUBLE, MPI_SUM,
                 MASTER, MPI_COMM_WORLD);
```



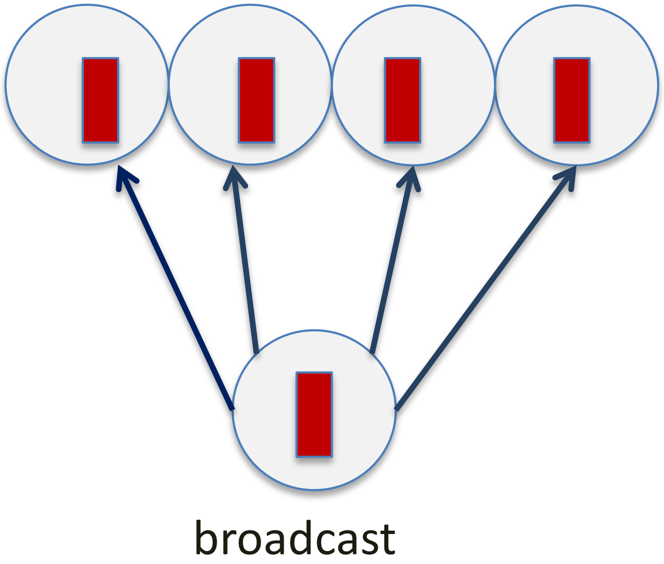
/* Use MPI_Reduce to sum values of **`localpi`** across all tasks
 * Master will store the accumulated value in **`pisum`**
 * - **`localpi`** is the send buffer
 * - **`pisum`** is the receive buffer (used by the receiving task only)
 * - the size of the message is sizeof(double)
 * - MASTER is the task that will receive the result of the reduction
 *   operation
 * - MPI_SUM is a pre-defined reduction function (double-precision
 *   floating-point vector addition).
 * - MPI_COMM_WORLD is the group of tasks that will participate.
 */

Collective Communications

- All the processes in the communicator must call the same collective function.
  - For example, a program that attempts to match a call to MPI_Reduce on one process with a call to MPI_Recv on another process is erroneous, and, in all likelihood, the program will hang or crash.

- The arguments passed by each process to an MPI collective communication must be "compatible."
  - For example, if one process passes in 0 as the dest_process and another passes in 1, then the outcome of a call to MPI_Reduce is erroneous, and, once again, the program is likely to hang or crash.

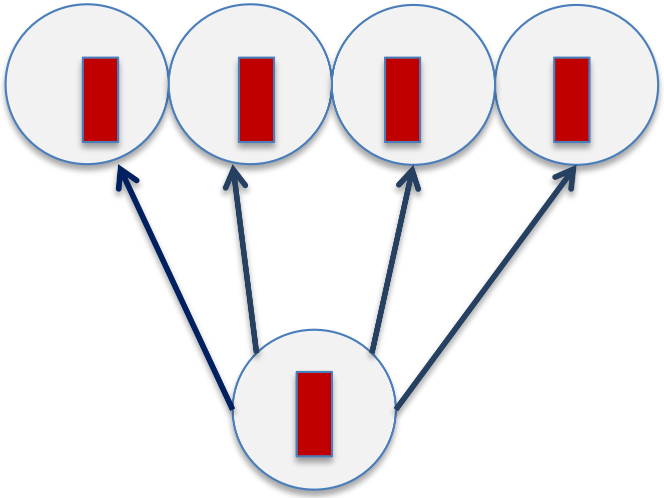
Collective Communication



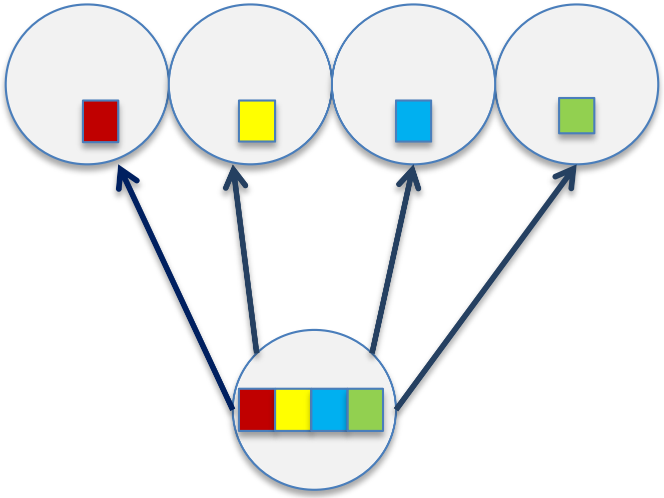

```
MPI_Bcast(void* data,int count,MPI_Datatype datatype,
          int root,MPI_Comm communicator)
```

MPI_Bcast's tree implementation provides additional network utilization so do not try to implement broadcast with send and receives but use collective communication routines.

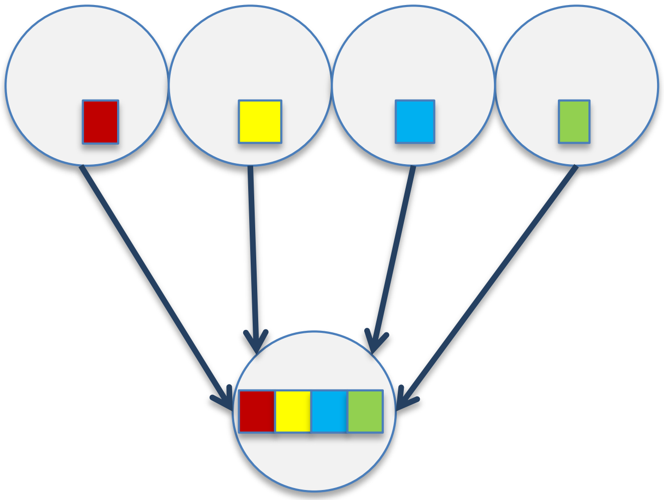
Collective Communication



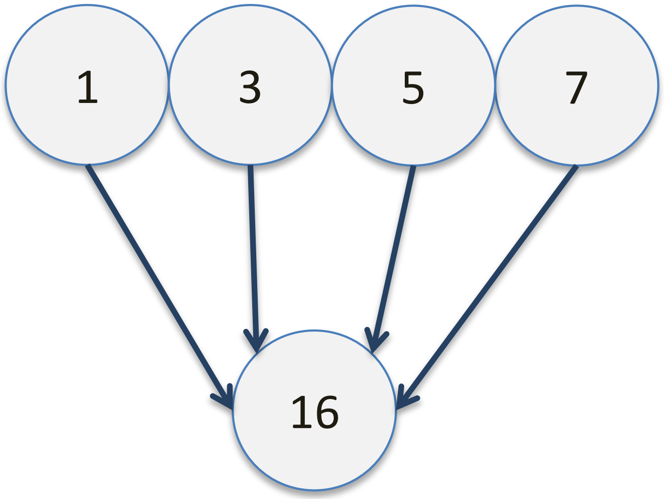
broadcast



scatter



gather



Reduction (sum)

NCCL (Nvidia Collective Communication Library)

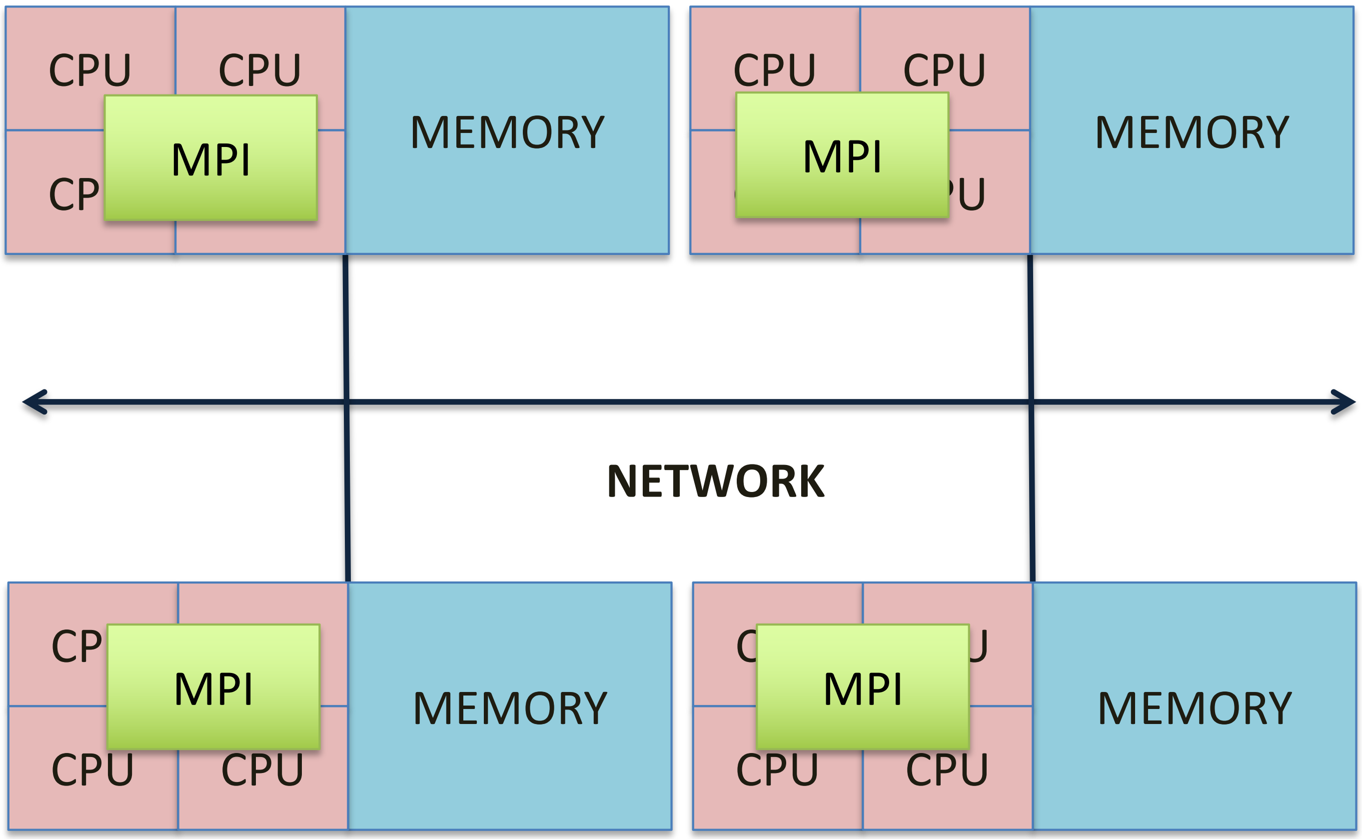
- NCCL (pronounced "Nickel")
  - Optimized collective communication library between CUDA devices
- Supports
  - all-reduce, all-gather, reduce, broadcast, reduce-scatter, as well as any send/receive based communication pattern.
  - High bandwidth on platforms using PCIe, NVLink, NVswitch, as well as networking using InfiniBand Verbs or TCP/IP sockets.
- NCCL supports an arbitrary number of GPUs installed in a single node or across multiple nodes, and can be used in either single- or multi-process (e.g., MPI) applications.

Usage of Collectives

- Collectives are central to scalability in a variety of key applications:
    - Deep Learning (All-reduce, broadcast, gather)
    - Parallel FFT (Transposition is all-to-all)
    - Molecular Dynamics (All-reduce)
    - Graph Analytics (All-to-all)
- ….

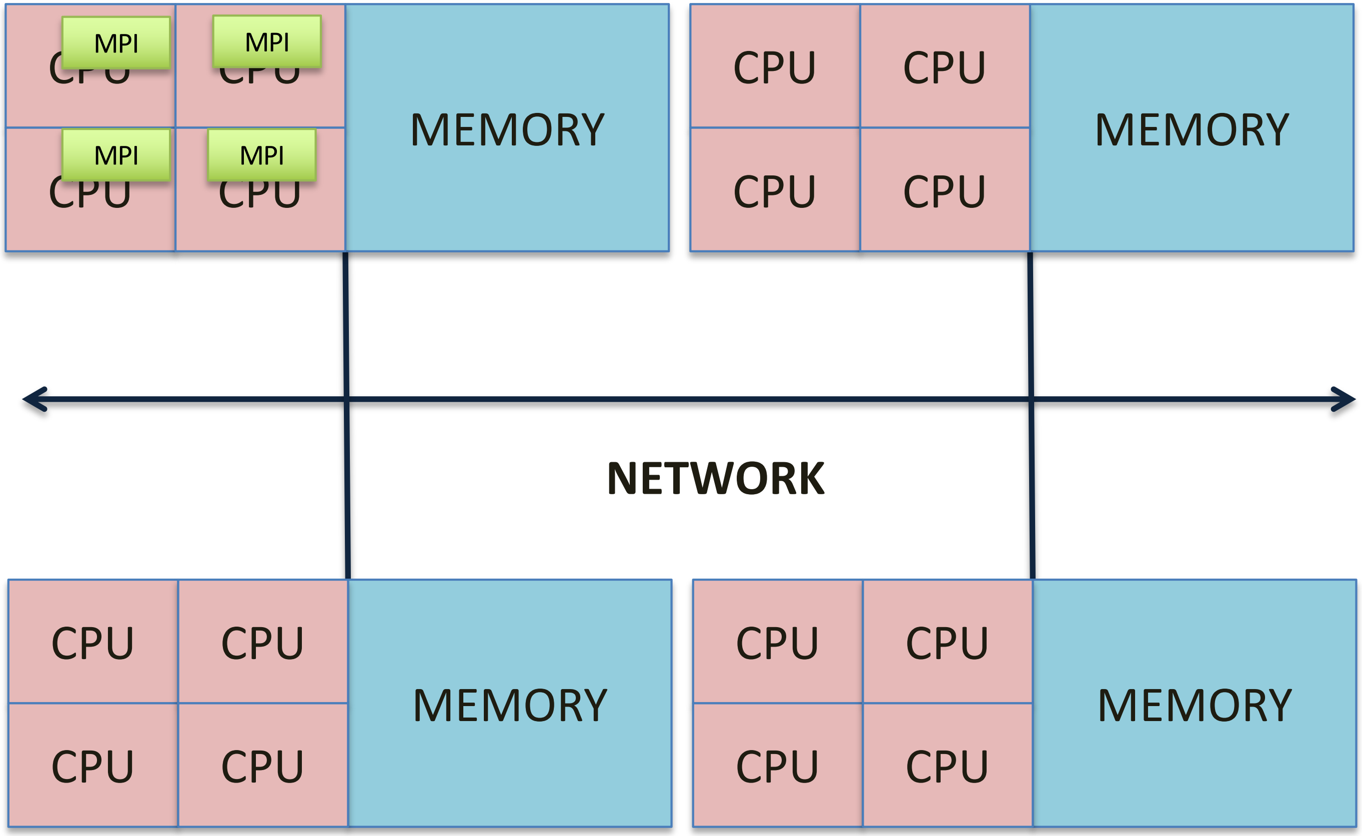
MPI on Distributed Memory

- MPI can be used within a shared memory node and distributed memory node
  - Processes will copy data within the shared memory because messages are the means of communication (not shared address space)



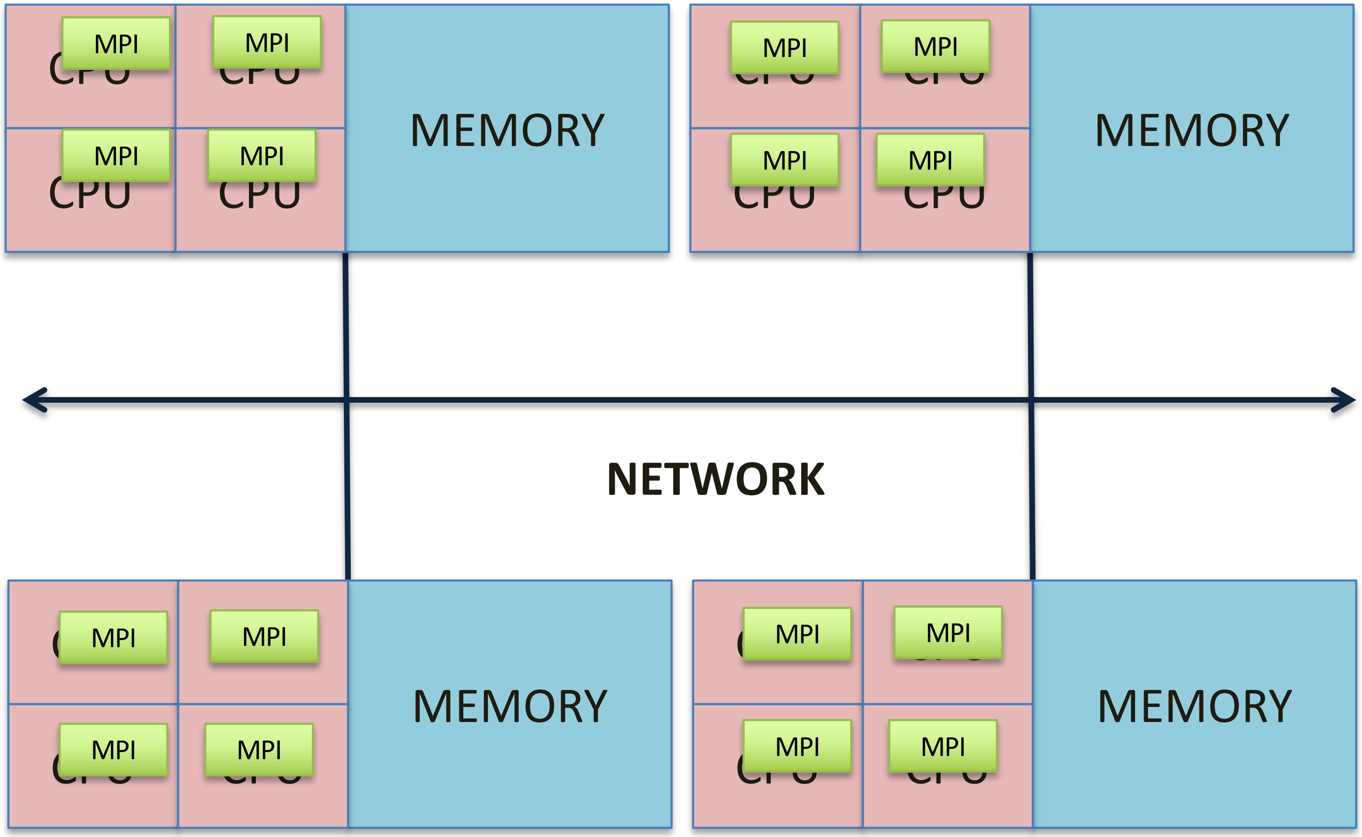
MPI on Shared Memory

- MPI can be used within a shared memory node and distributed memory node
  - Processes will copy data within the shared memory because messages are the means of communication (not shared address space)



MPI on Shared+Distributed Memory

- MPI can be used within a shared memory node and distributed memory node
  - Processes will copy data within the shared memory because messages are the means of communication (not shared address space)



Readings

- Read Chapter 3
- MPI tutorial
  - https://computing.llnl.gov/tutorials/mpi/
  - http://mpitutorial.com/tutorials/

- Next lectures
  - More on collective communication
  - Asynchronous, non-blocking communication

Acknowledgments

- These slides are inspired and partly adapted from
  - Mary Hall (Univ. of Utah)
  - Scott Baden (UCSD)
  - Vivek Sarkar (Rice Univ.)
  - The course book (Pacheco)
  - https://computing.llnl.gov/tutorials/mpi/