KOÇ UNIVERSITY

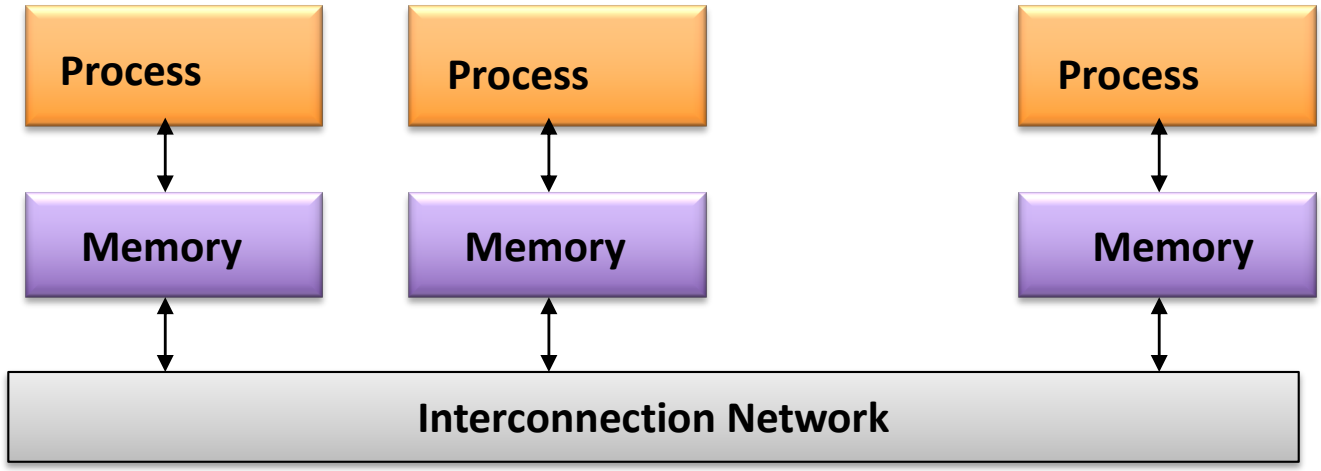
Non-Blocking Communication and Network Topologies

Didem Unat

COMP 429/529 Parallel Programming

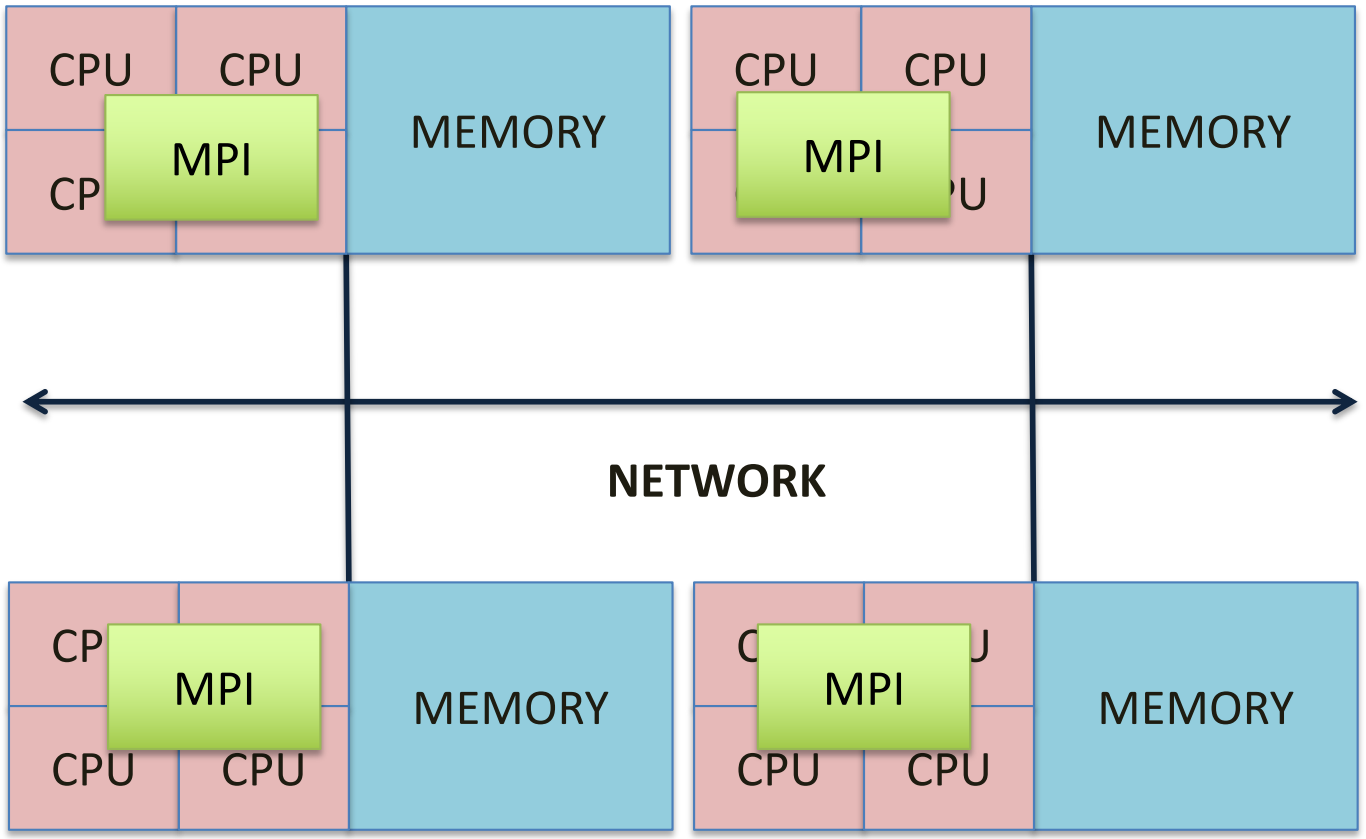
Message Passing Programming Model

- 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)



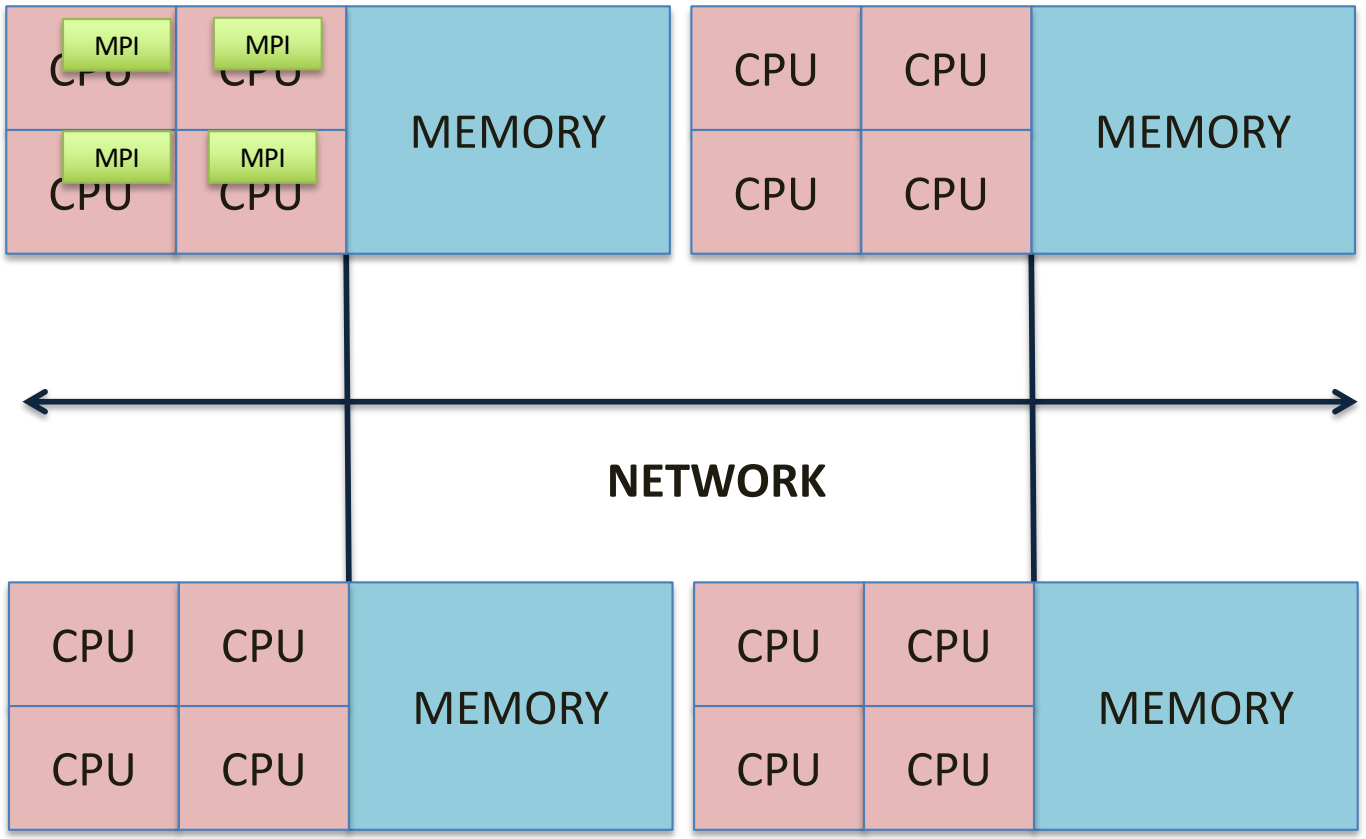
MPI on Distributed Memory

- 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)



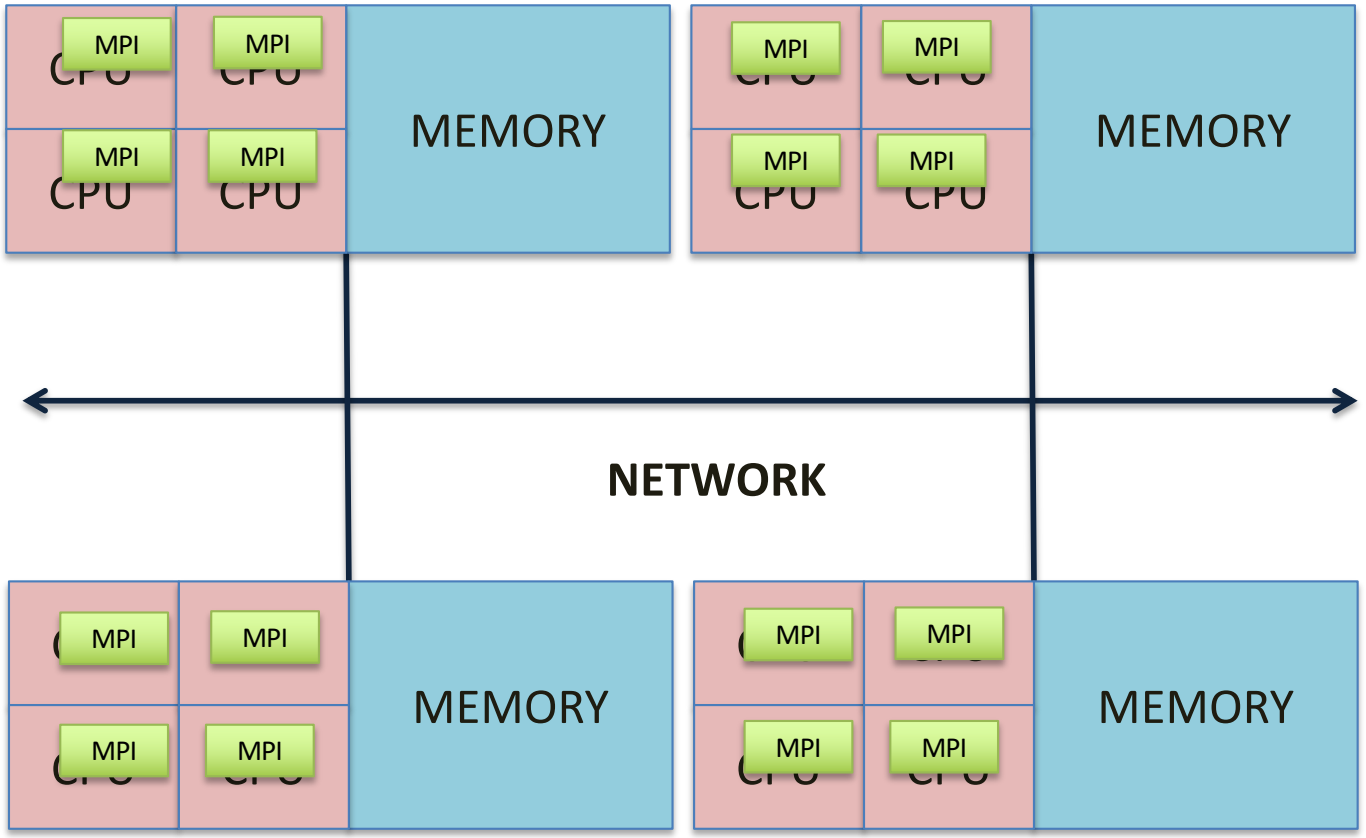
MPI on Shared Memory

- 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)



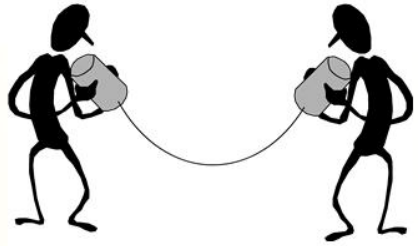
MPI on Shared+Distributed Memory

- 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)



Message Passing

- 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

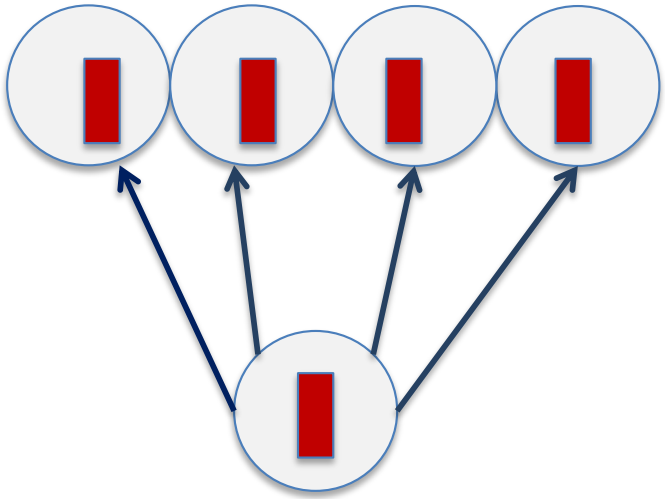


  - **Collective communication** involving multiple processors
    - Move data: Broadcast, Scatter/gather
    - Compute and move: Reduce, AllReduce

MPI Review

- Six most common MPI Commands
  - **MPI_Init**
  - **MPI_Finalize**
  - **MPI_Comm_size**
  - **MPI_Comm_rank**
  - **MPI_Send**
  - **MPI_Recv**
- One can implement rest of the MPI communication calls in terms of sends and receives though they won't be very efficient

Collective Communication

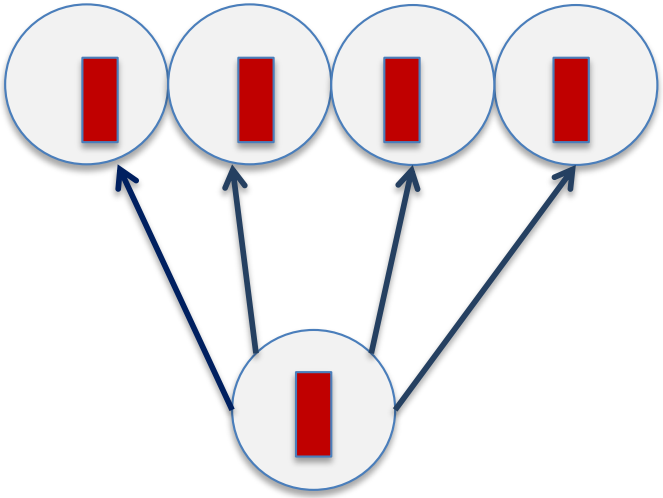


broadcast

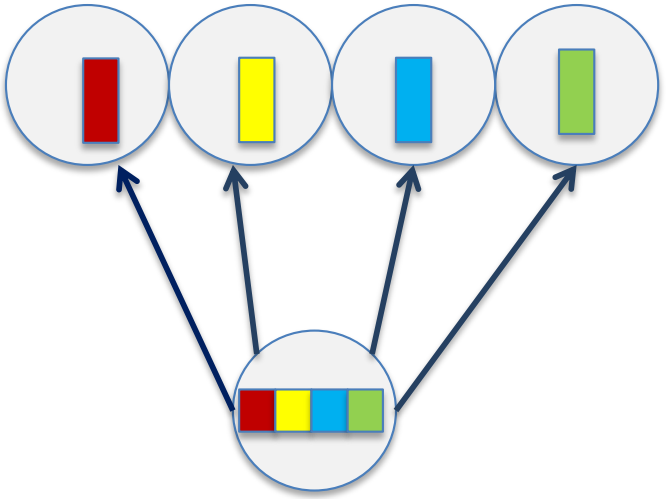
```
MPI_Bcast(void* data,int count,MPI_Datatype datatype,
          int root,MPI_Comm communicator)
```

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.

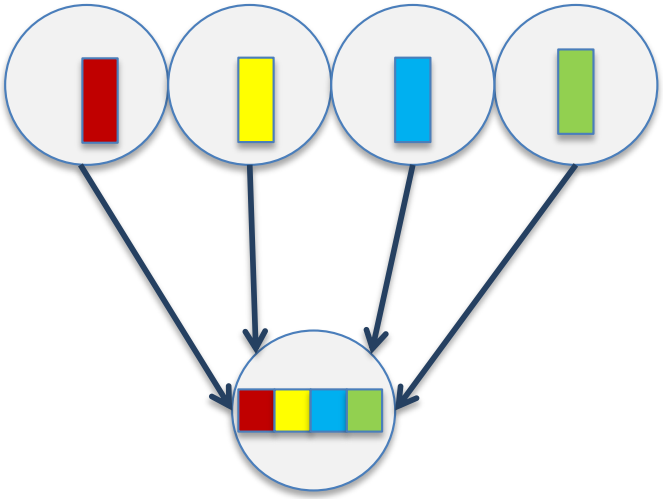
Collective Communication



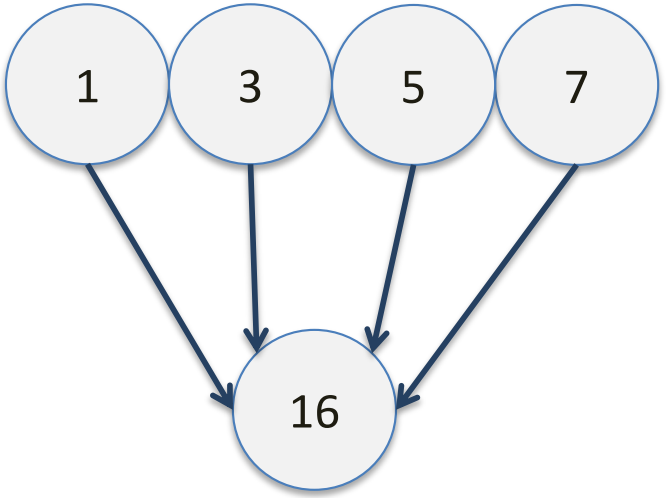
broadcast



scatter



gather



Reduction (sum)

MPI_Init()

- The MPI standard does not say what a program can do before an MPI_INIT or after an MPI_FINALIZE.
  - In the MPICH implementation, it is suggested that you should do as little as possible.
  - In particular, avoid anything that changes the external state of the program, such as opening files, reading standard input or writing to standard output.

- http://mpi.deino.net/mpi_functions/MPI_Init.html

Blocking Sends and Receives

- The sender *copies the data into a designated buffer* and *returns after the copy operation* has been completed.
  - When send() returns, it is safe to overwrite the message
  - Message is still in transit

- The data *must be buffered at the receiving end* as well.
  - When receive() returns, it is safe to read the message

- When there is no corresponding receive for a send or no corresponding send for a receive, a deadlock will occur

Send and Receive

```
. . .
int a[10], b[10], myrank;
MPI_Status status; ...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, tag1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, tag2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, tag2, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, tag1, MPI_COMM_WORLD);
}
...
```

- Deadlock?
  - MPI_Send is blocking, there is a deadlock. Tags do not match!

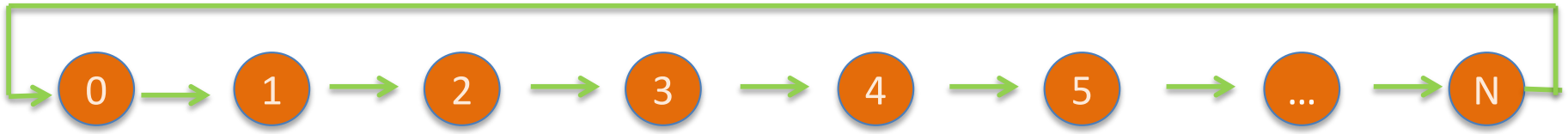
Send and Receive

```
. . .
int a[10], b[10], myrank;
MPI_Status status; ...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

if (myrank == 0) {
    MPI_Recv(a, 10, MPI_INT, 1, tag1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, tag2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, tag1, MPI_COMM_WORLD);
    MPI_Send(a, 10, MPI_INT, 0, tag2, MPI_COMM_WORLD);
}
...
```

- Deadlock? Yes- Recv is a blocking call

Ring

- Consider the following piece of code, in which process$_i$ sends a message to process$_{i+1}$ and receives a message from process$_{i-1}$.

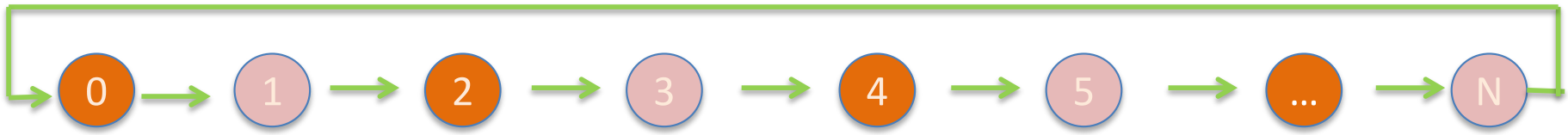



```
int a[10], b[10], np, myrank;
...
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

MPI_Send(a, 10, MPI_INT, (myrank+1)%np, 1, MPI_COMM_WORLD);

MPI_Recv(b, 10, MPI_INT, (myrank-1+np)%np, 1, MPI_COMM_WORLD);

...
```

We have a deadlock since MPI_Send is blocking

Ring – Avoid Deadlock

- We can break the circular wait to avoid deadlocks as follows:




```
int a[10], b[10], np, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &np);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

if (myrank%2 == 1) {
 MPI_Send(a, 10, MPI_INT, (myrank+1)%np, 1,MPI_COMM_WORLD);
 MPI_Recv(b, 10, MPI_INT, (myrank-1+np)%np, 1,MPI_COMM_WORLD);
}
else {
 MPI_Recv(b, 10, MPI_INT, (myrank-1+np)%np, 1, MPI_COMM_WORLD);
 MPI_Send(a, 10, MPI_INT, (myrank+1)%np, 1, MPI_COMM_WORLD);

} ...
```

Assumes even number of processes.

Non-Blocking Communications

- The class of non-blocking protocols returns from the send or receive operations before it is semantically safe to do so.
  - Thus copy operation may not be completed on return
- Non-blocking send and receive operations in MPI
  - "I" stands for "Immediate":

- `int MPI_Isend(. . .)`
  - Processing continues immediately without waiting for the message to be copied out from the application buffer.
- `int MPI_Irecv(. . .)`
  - Processing continues immediately without actually waiting for the message to be received and copied into the the application buffer.

Nearest neighbor exchange in a ring topology with Non-Blocking Messages

```
#include "mpi.h"
#include <stdio.h>
main(int argc, char *argv[])  {
int numtasks, rank, next, prev, recvbuf[2], sendbuf[2], tag1=1, tag2=2;
MPI_Request reqs[4];
MPI_Status stats[4];

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

initBuffers(sendbuf);
prev = rank-1;
next = rank+1;
if (rank == 0)  prev = numtasks - 1;
if (rank == (numtasks - 1))  next = 0;

MPI_Irecv(&recvbuf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&recvbuf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

MPI_Isend(&sendbuf[0], 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&sendbuf[1], 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

{  do some work  }

MPI_Waitall(4, reqs, stats);

MPI_Finalize();
}
```

Receive from left neighbor

Receive from right neighbor

Send to left neighbor

Send to right neighbor

Not safe to use recvbuf and sendbuf before wait!

MPI_Wait and _Test

- All asynchronous (non-blocking)operations are given a request handle to check the completion of the operations

- MPI_Wait(req, status)
  - Blocks and waits for an MPI send or receive to complete
- MPI_WaitAll(count, reqs, status)
  - Blocks and waits for all given communications to complete
- MPI_Test (req, flag, status)
  - Tests for the completion of a send or receive
- MPI_TestAll(count, reqs, flag, status)
  - Tests for the completion of all previously initiated communications

- If neither is done, the asynchronous calls may not be fully completed- message not sent, memory leaks etc.

Collective Communication Operations

- Up until MPI-3, all collective calls used to be blocking
- Now, there exists the non-blocking counterparts
  - At least they are in the specification

- Another useful collective communication is barrier synchronization in MPI

```
int MPI_Barrier(MPI_COMM comm);
```

Ping-Pong and Ring Examples

- ssh to KUACC
- Copy the MPI folder to your home
  - /kuacc/users/dunat/COMP429/MPI

```
module avail
module load mpich/3.2.1
make
mpiexec –n proc# ./executable
```

Modeling Communication

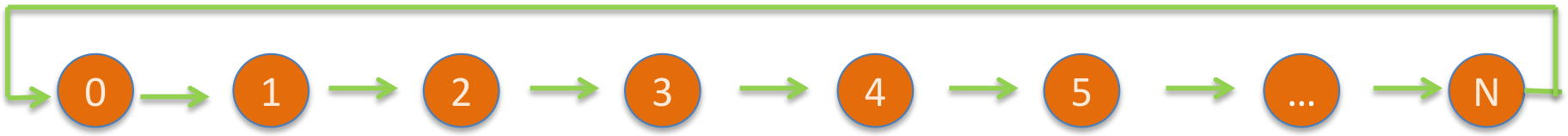
- Simple model for sending a message
  - Latency cost (α = message startup time)
    - Fixed cost for getting a message from point A to B.
    - Proportional to number of messages
  - Bandwidth cost (β = network bandwidth (bytes/sec))
    - Throughput for sending the bits over the wire
    - Proportional to message size

```
Msg Cost = Latency + MsgSize / Bandwidth
```

- Limitations of the model
  - Ignores the cost of packing/unpacking of messages
  - Ignores the network contention and topology

Benchmark: Ring Program

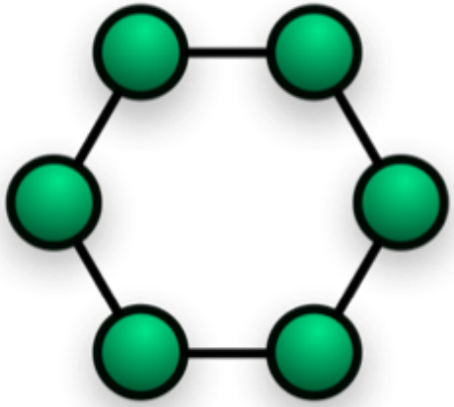
- Ring program can measure the communication performance
  - Repeatedly send and receive messages



Short messages can be used to measure startup time (latency). Long messages can be used to measure bandwidth of the network.

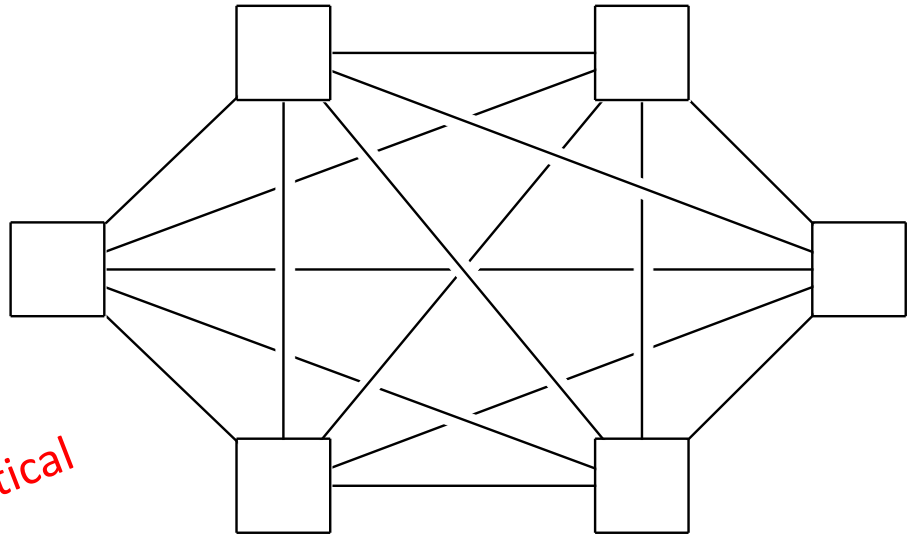
Network Topology

- Communication cost also depends on network topology
  - Messaging with close neighbors are fast while remote neighbors takes more time
  - Minimize latency, diameter, and cost



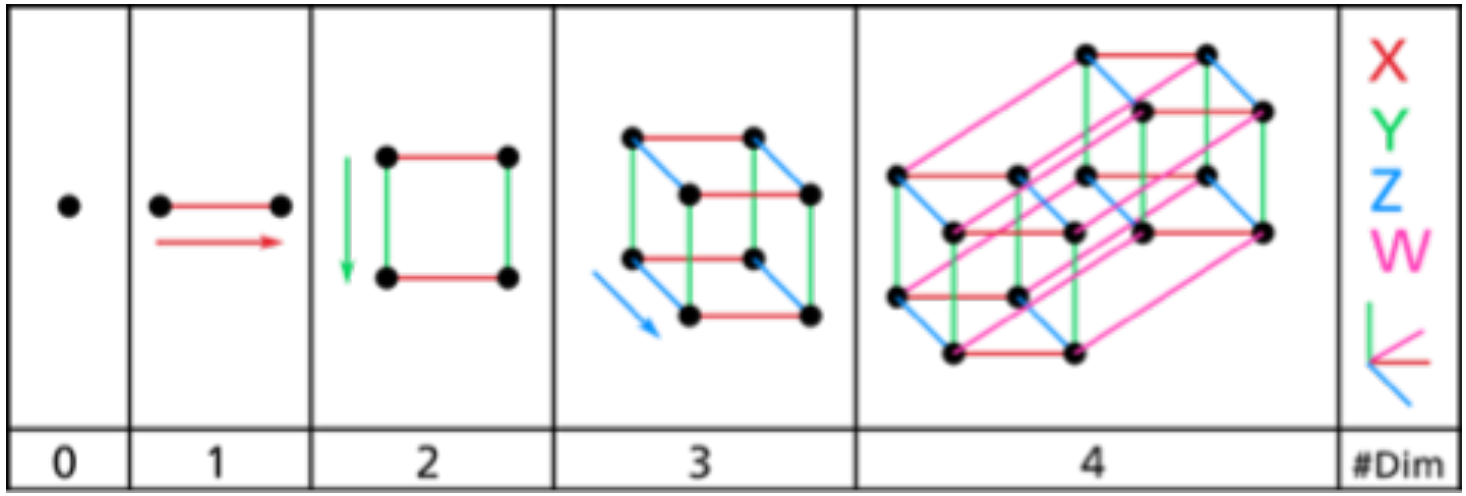
ring

impractical



Fully connected network

Hypercube Networks

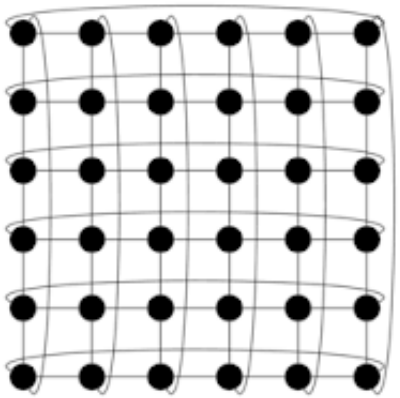


one- two- three-dimensional

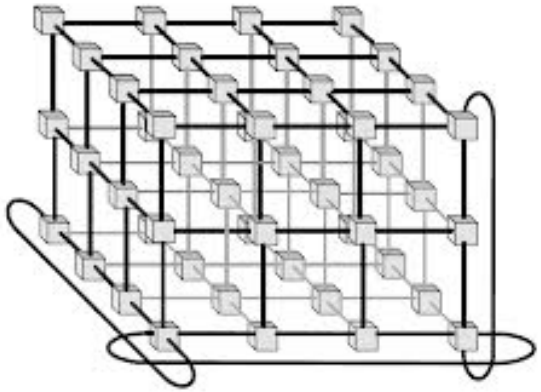
- Each node has *d* neighbors (d is the dimension)
- The distance between any two nodes is at most d
- Was very popular early years of parallel computing
  - Expensive and doesn't scale in today's machines

Torus Networks

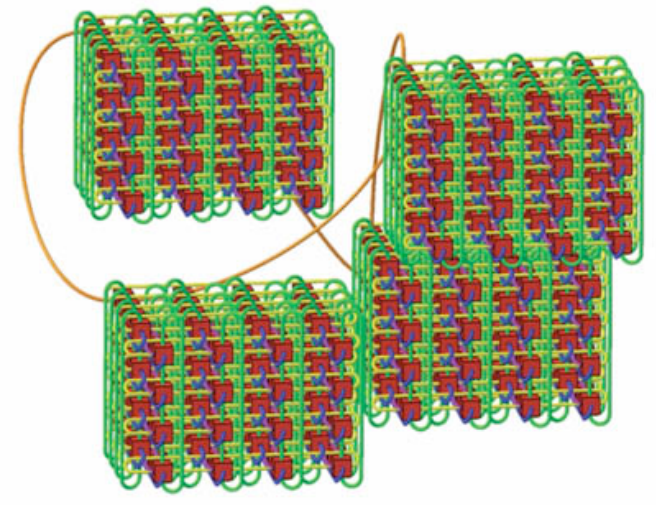
- Torus is a mesh with wrap-around links
- k-D p-array torus: k dimensional torus with p processors in each dimension
- Japanese K-computer uses 6D torus connecting 80K nodes



2D Torus



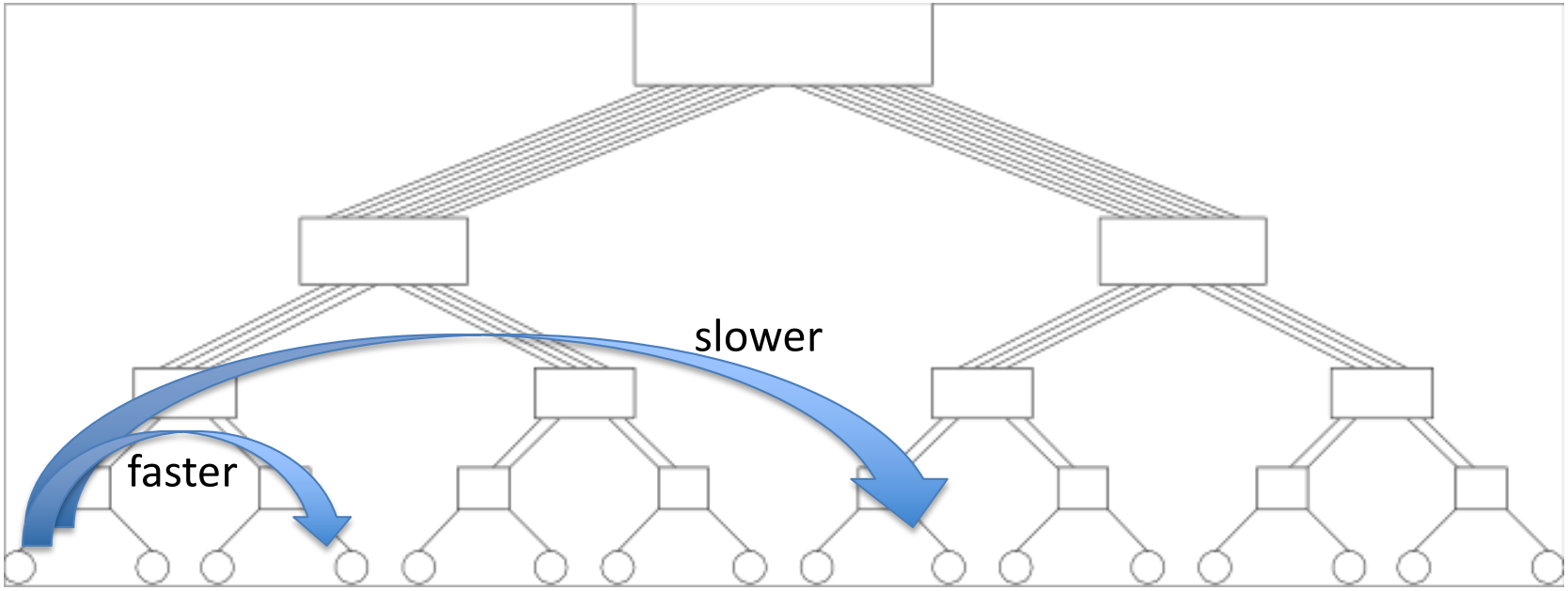
3D Torus



4x4x4x4x2 Torus Network

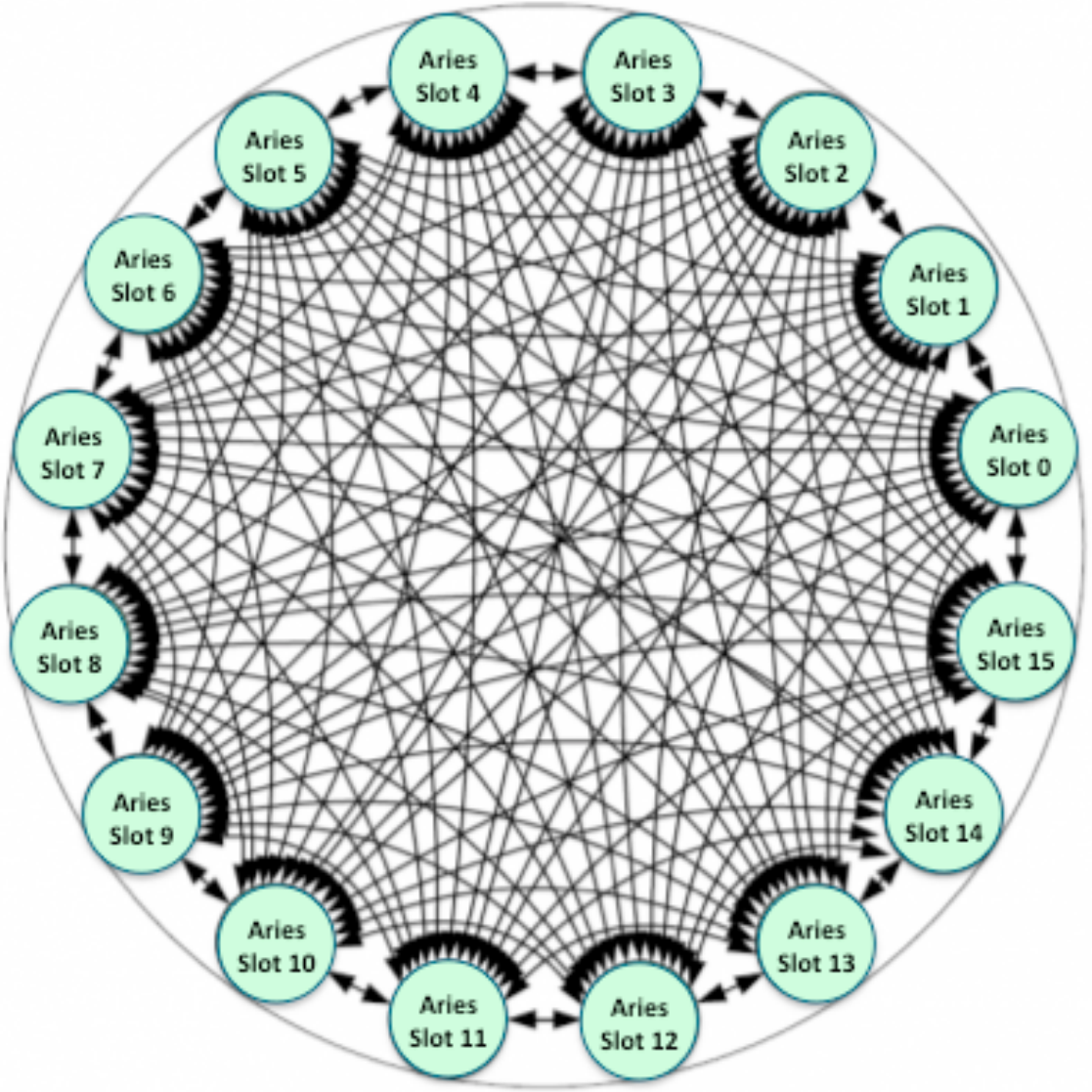
Tree Networks

- Complete binary tree networks
- If there are p processors, the depth of the tree is *log(p)*
- The distance between any two nodes is no more than *2log(p)*
- Links higher up the tree potentially carry more traffic than those at the lower levels.
  - A variant called a fat-tree, fattens the links as we go up the tree



Other Networks

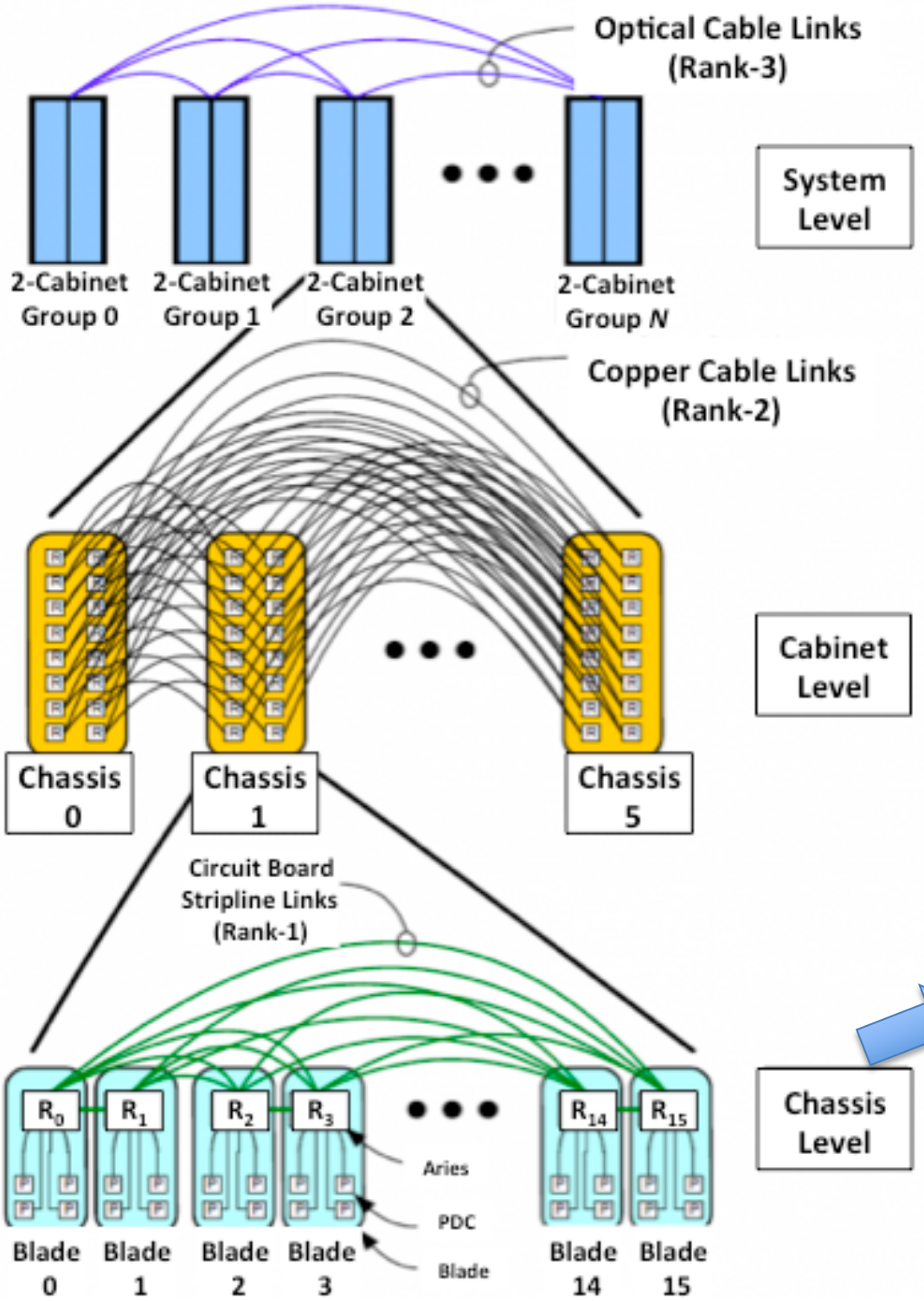
- Omega Networks
- Butterfly Networks
- Dragonfly Networks
- Custom Networks

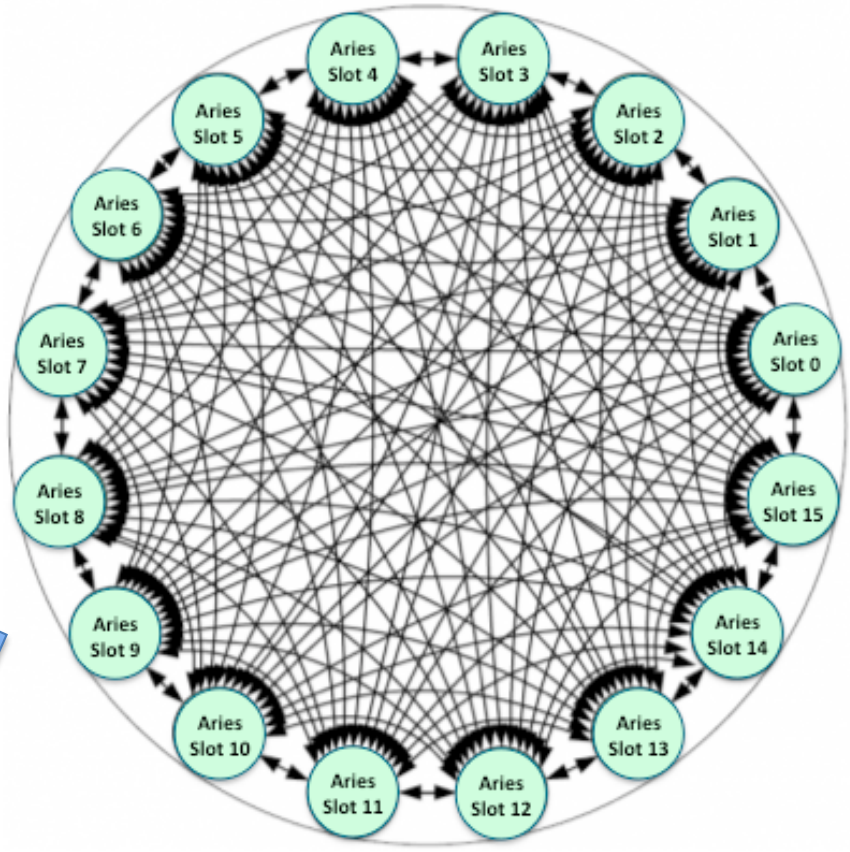


Dragonfly Network

The groups are arranged such that data transfer from one group to another requires only one route through a global link.

Cori Supercomputer employs the "Dragonfly" topology





Wiring Supercomputers

Trinity supercomputer

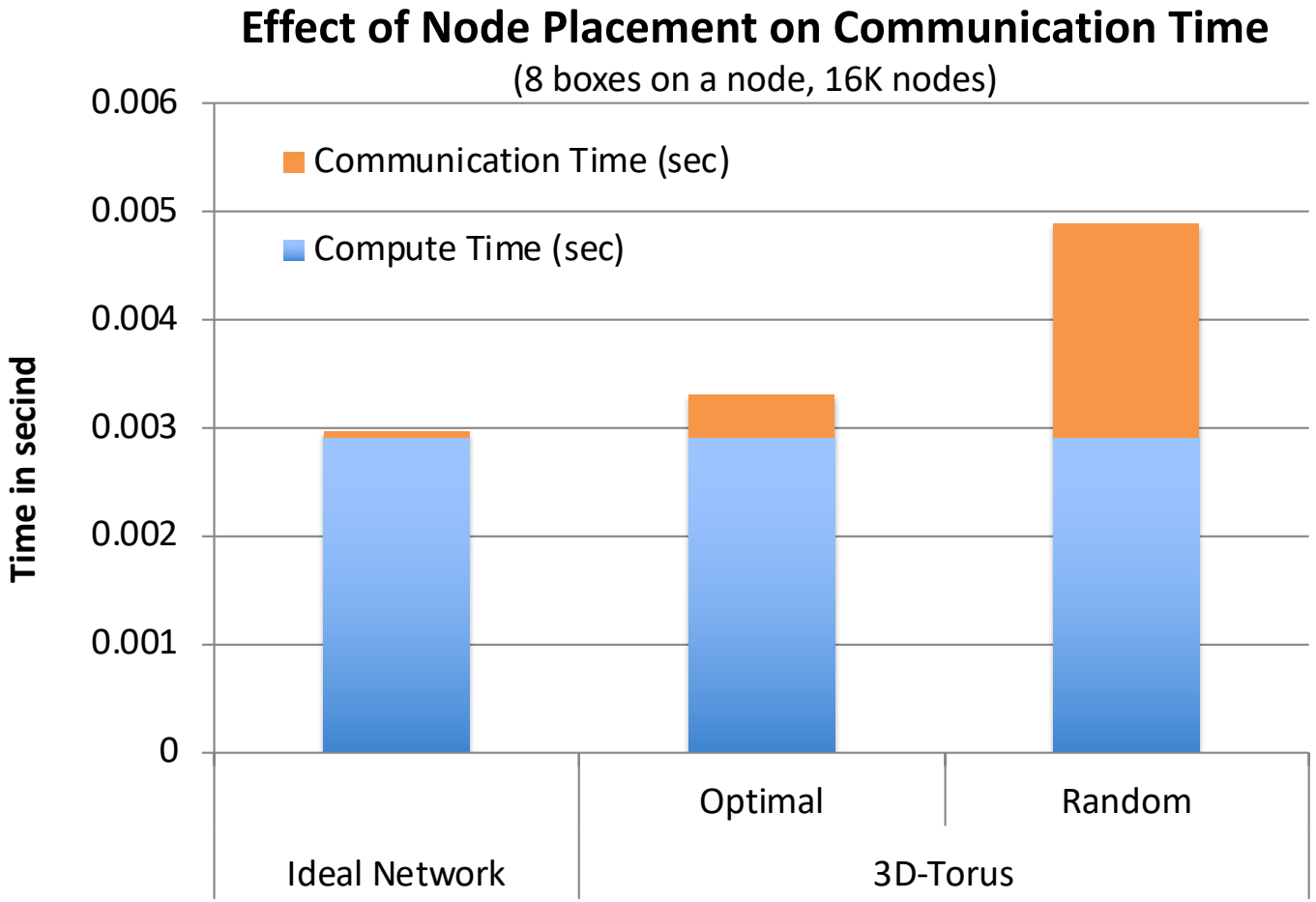
Mira Supercomputer

A beautiful example

- https://www.bsc.es/marenostrum/marenostrum

Rank Placement on Network

- An example application running with 16000 MPI processes
  - Ideal network is fully connected
  - Optimal places communicating ranks closer to each other on the network
  - Random randomly maps MPI ranks to network nodes



Acknowledgments

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