

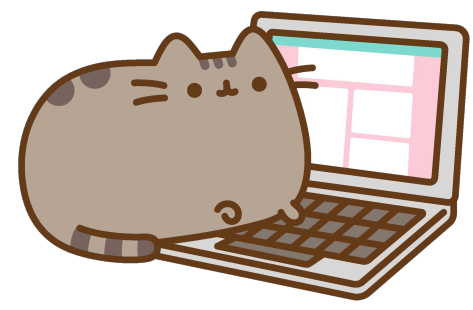
Announcements

- HW3 due Thursday

CSE 6230:
HPC Tools and Applications



+



Lecture 11: Distributed-Memory Programming

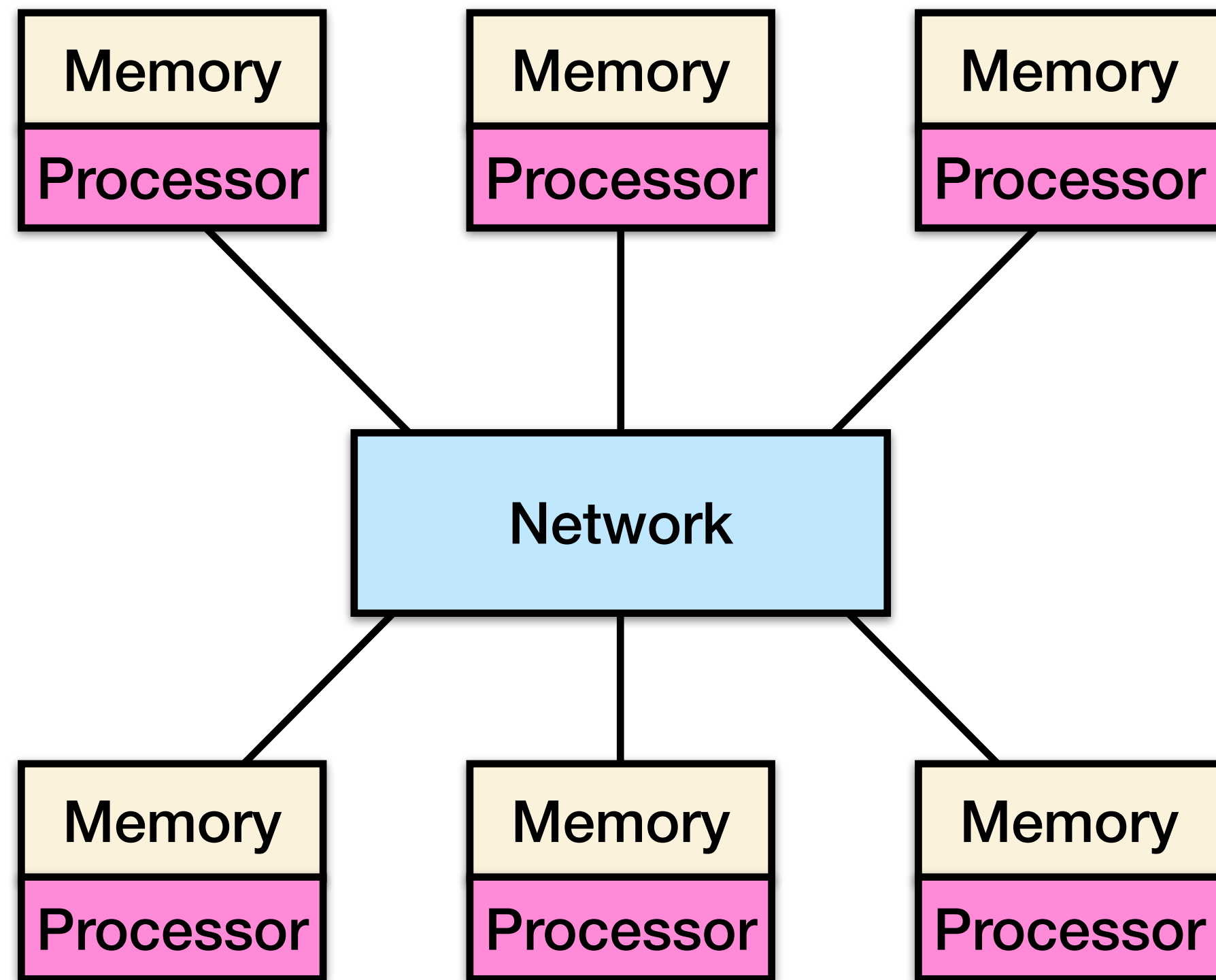
Helen Xu

hxu615@gatech.edu



Georgia Tech College of Computing
School of Computational
Science and Engineering

Recall: Distributed-Memory Machines

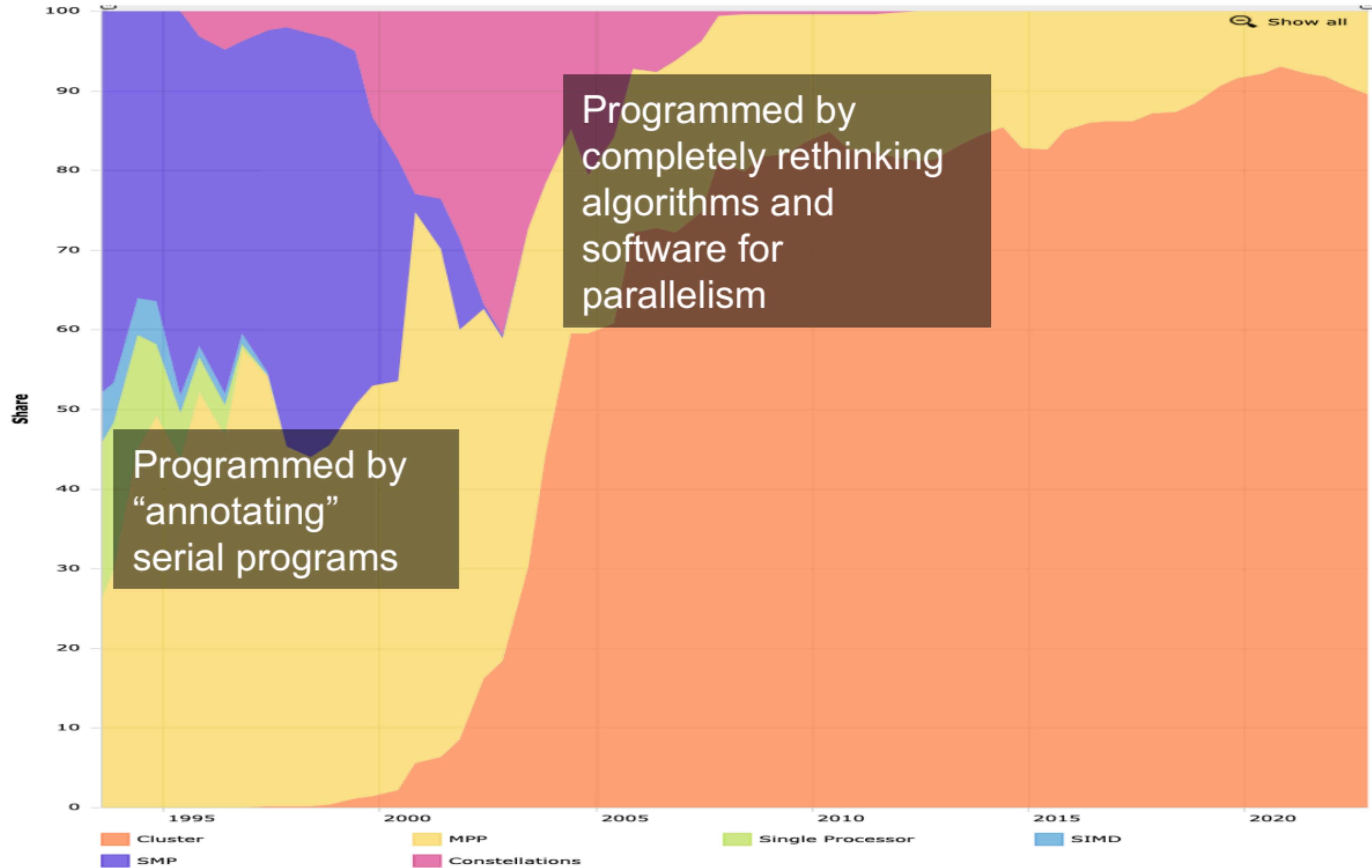


A **distributed-memory multiprocessor** has processors with their own memories connected by a high-speed network.

Also called a **cluster**.

A **high-performance computing** (HPC) system contains 100s or 1000s of such processors (nodes).

From vector machines to massively parallel accelerator systems



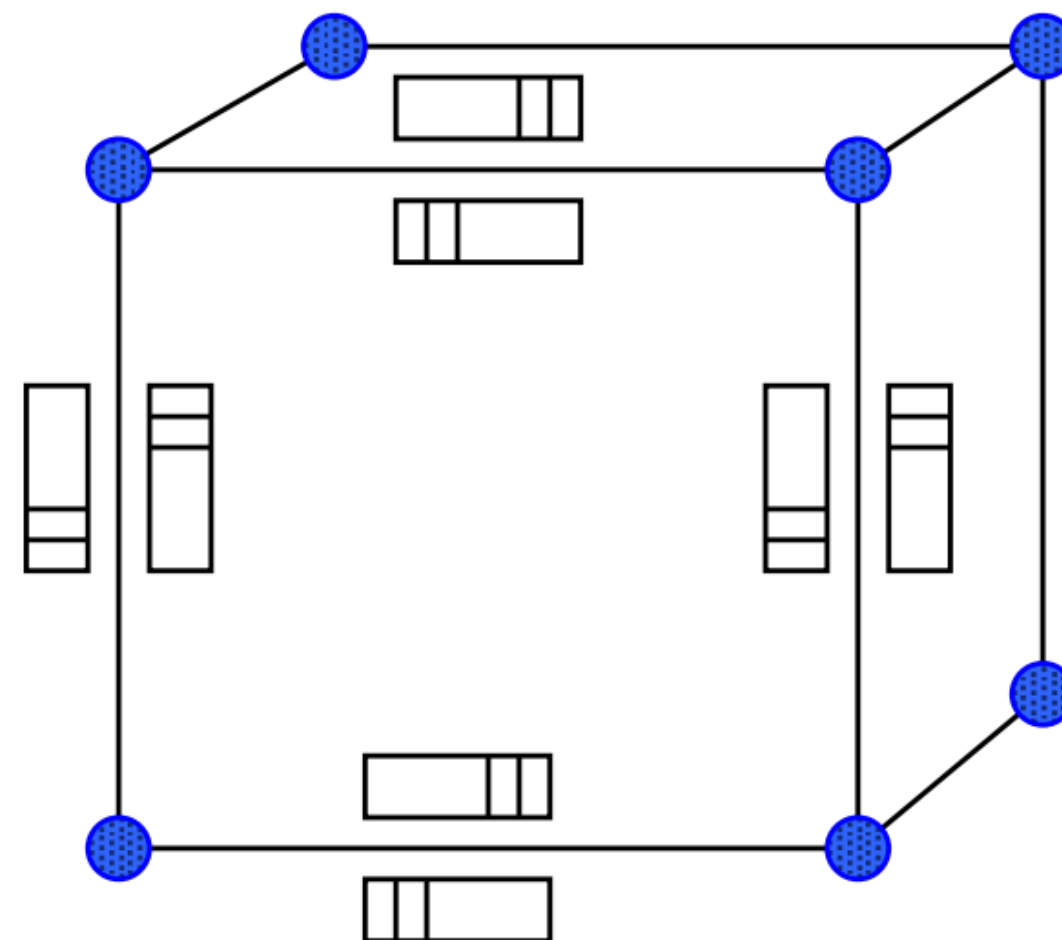
Example: Frontier System

System	Titan (2012)	Summit (2017)	Frontier (2021)
Peak	27 PF	200 PF	2 EF
# nodes	18,688	4,608	9,408
Node	1 AMD Opteron CPU 1 NVIDIA Kepler GPU	2 IBM POWER9™ CPUs 6 NVIDIA Volta GPUs	1 AMD EPYC CPU 4 AMD Radeon Instinct GPUs
Memory	0.6 PB DDR3 + 0.1 PB GDDR	2.4 PB DDR4 + 0.4 HBM + 7.4 PB On-node storage	4.6 PB DDR4 + 4.6 PB HBM2e + 36 PB On-node storage, 66 TB/s Read 62 TB/s Write
On-node interconnect	PCI Gen2 No coherence across the node	NVIDIA NVLINK Coherent memory across the node	AMD Infinity Fabric Coherent memory across the node
System Interconnect	Cray Gemini network 6.4 GB/s	Mellanox Dual-port EDR IB 25 GB/s	Four-port Slingshot network 100 GB/s
Topology	3D Torus	Non-blocking Fat Tree	Dragonfly
Storage	32 PB, 1 TB/s, <u>Lustre</u> Filesystem	250 PB, 2.5 TB/s, IBM Spectrum Scale™ with GPFS™	695 PB HDD+11 PB Flash Performance Tier, 9.4 TB/s and 10 PB Metadata Flash. <u>Lustre</u>
Power	9 MW	13 MW	29 MW

Distributed-Memory Architectures

Historical Perspective

- Early distributed memory machines were:
 - Collection of microprocessors.
 - Communication was performed using bi-directional **queues between nearest neighbors**.
- Messages were **forwarded by processors** on path - “Store and forward” networking
- There was a strong emphasis on **topology in algorithms**, in order to minimize the number of hops = minimize time



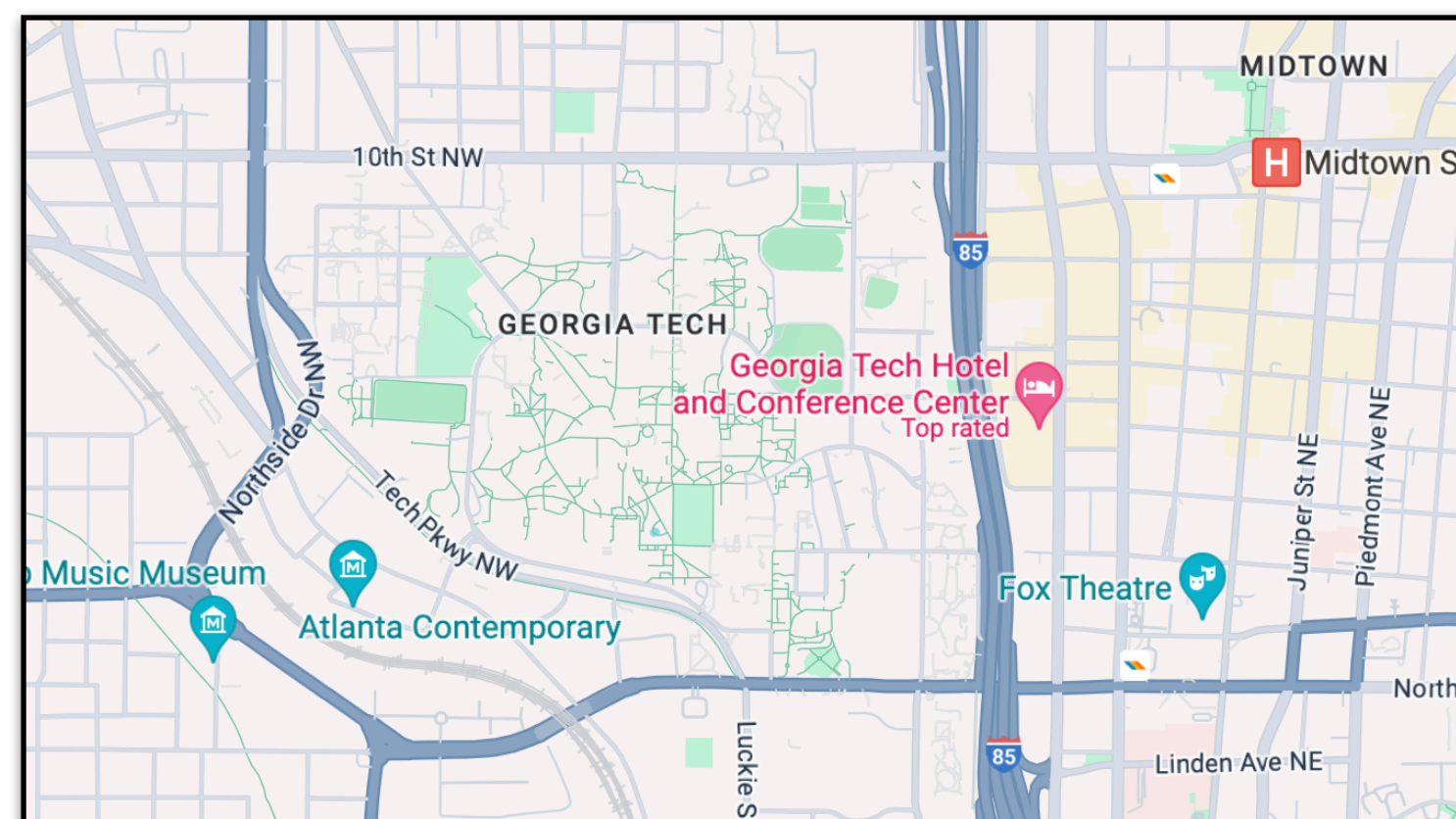
Network Analogy

To have a large number of different transfers occurring at once, you need a large number of distinct wires

- Not just a bus, as in shared memory

Networks are like streets:

- Link = street.
- Switch = intersection.
- Distances (hops) = number of blocks traveled.
- Routing algorithm = travel plan.



Network Properties

Latency - how long to get between nodes in the network

- Street - time for one car = $\text{dist (miles)} / \text{speed (miles/hr)}$

Bandwidth - how much data can be moved per unit time

- Street - cars / hour = $\text{density (cars / mile)} * \text{speed (miles / hr)} * \text{\#lanes}$
- Network bandwidth is limited by the bit rate per wire and \#wires

Bandwidth

\approx data throughput (bytes/second)



Low Bandwidth



High Bandwidth

Latency

\approx delay due data travel time (secs)



Low Latency



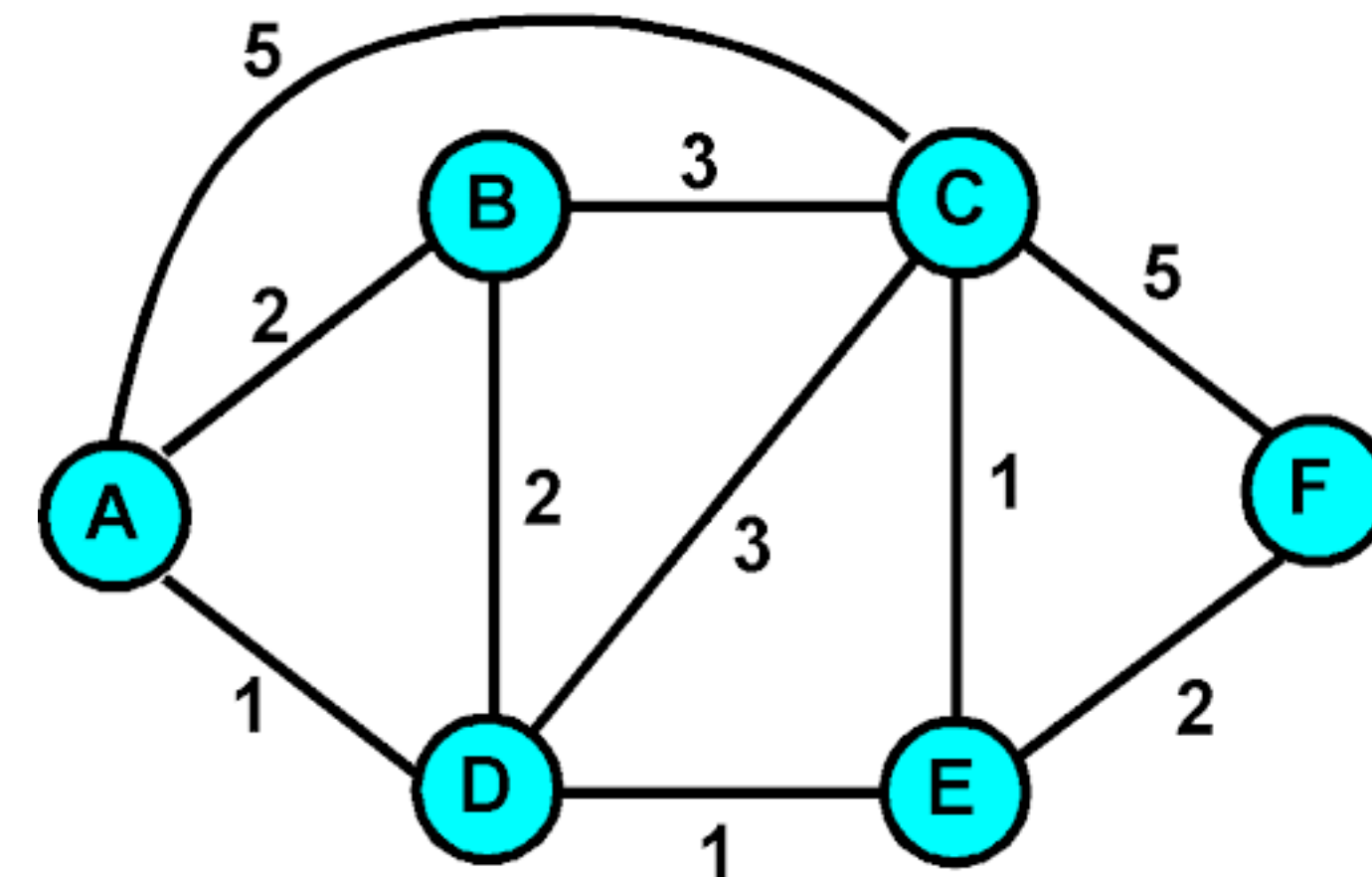
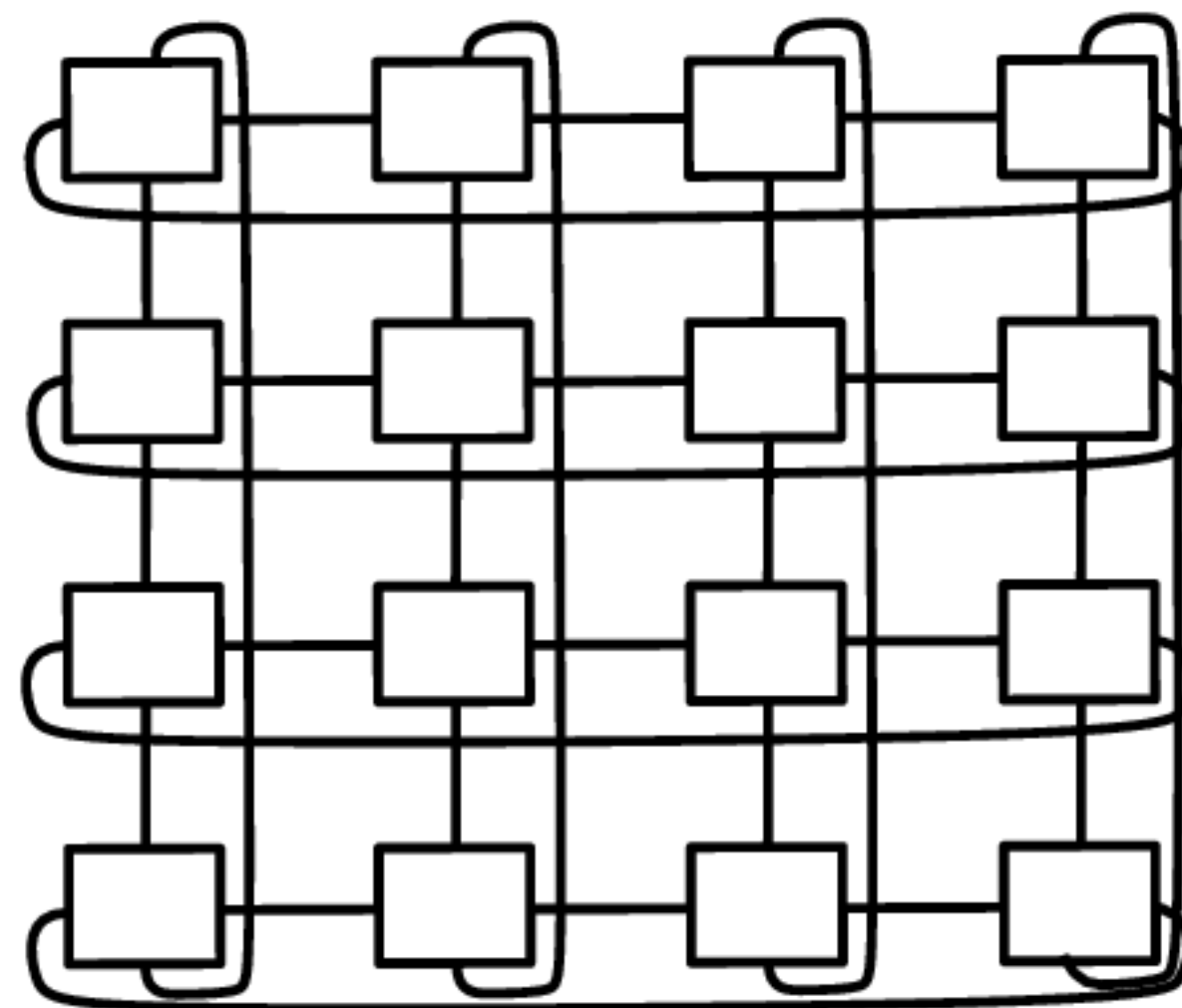
High Latency

Network Design Characteristics

Topology (how things are connected) - examples include crossbar; ring; 2-D, 3-D, higher-D mesh or torus; hypercube; tree; butterfly; perfect shuffle, dragon fly, ...

Routing algorithm - how messages move through the network

- Example in 2D torus: all east-west then all north-south (avoids deadlock).



http://www2.ic.uff.br/~michael/kr1999/4-network/4_02-algor.htm

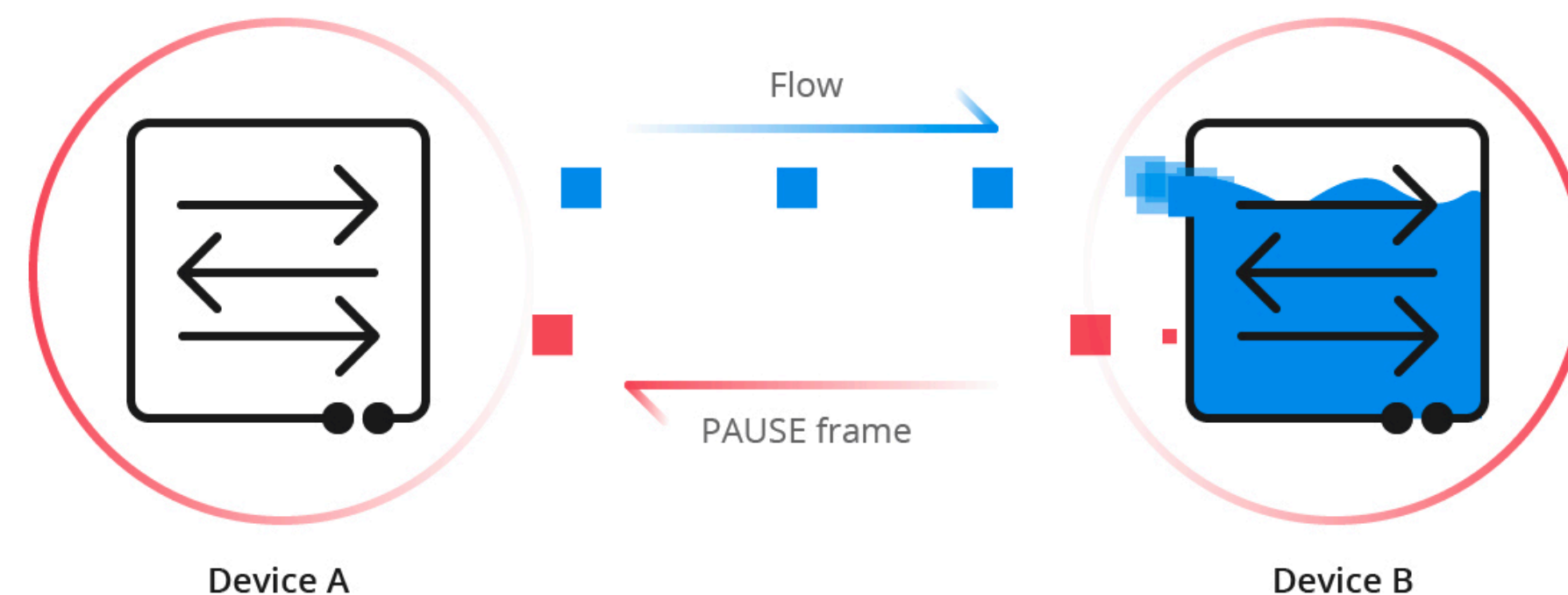
Network Design Characteristics

Switching strategy:

- Circuit switching: full path reserved for entire message, like the telephone.
- Packet switching: message broken into separately-routed packets, like the post office, or internet

Flow control (what if there is congestion):

- Stall, store data temporarily in buffers, re-route data to other nodes, tell source node to temporarily halt, discard, etc.



Performance Properties: Latency

Diameter: the maximum (over all pairs of nodes) of the shortest path between a given pair of nodes.

Latency: delay between send and receive times

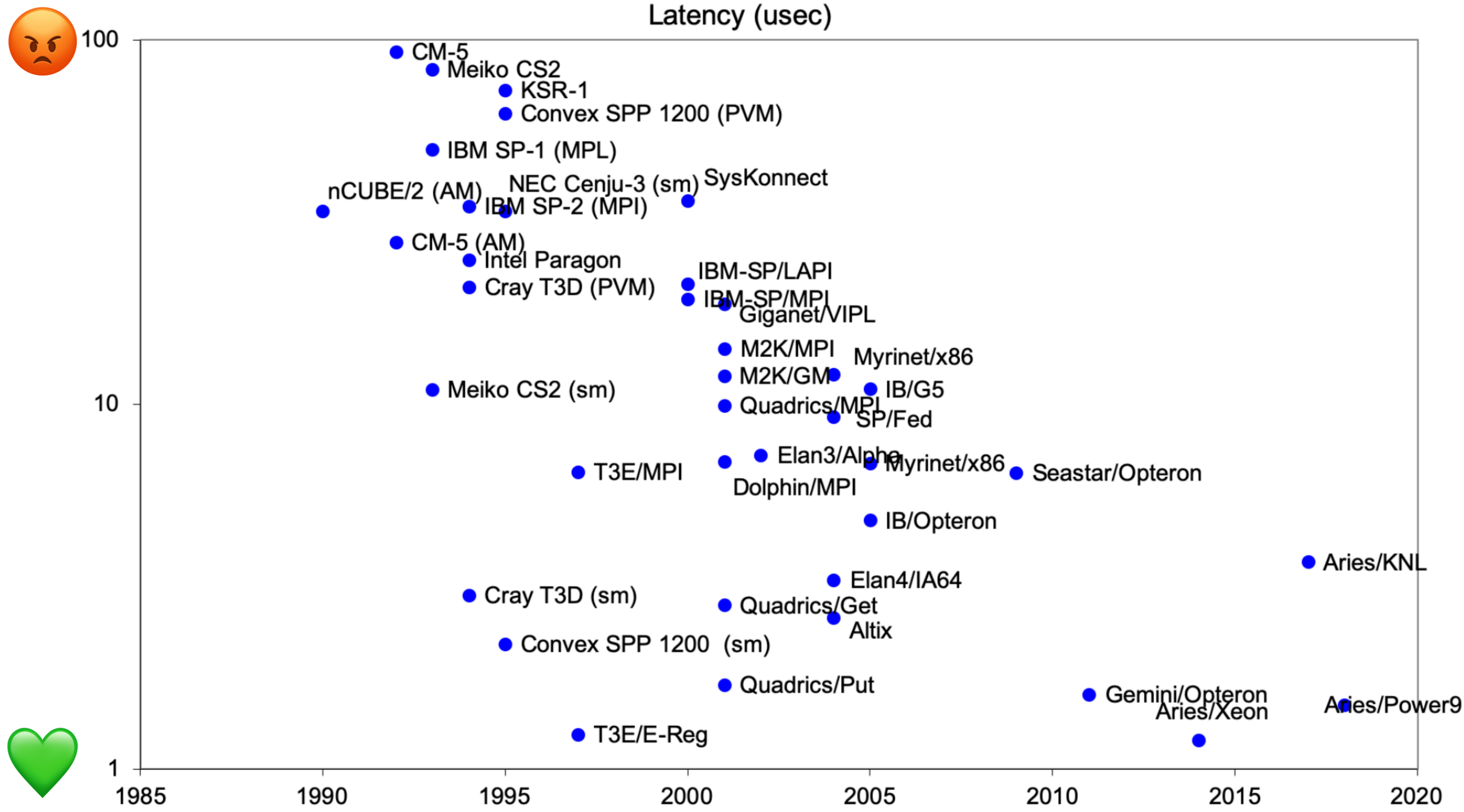
- Vendors often report **hardware latencies** (wire time).
- Application programmers care about **software latencies** (user program to user program).

Software/hardware overhead at source/destination dominate cost (1s-10s usecs).

- Hardware latency varies with distance (10s-100s nsec per hop) but is small compared to overheads.

Latency is key for programs **with many small messages.**

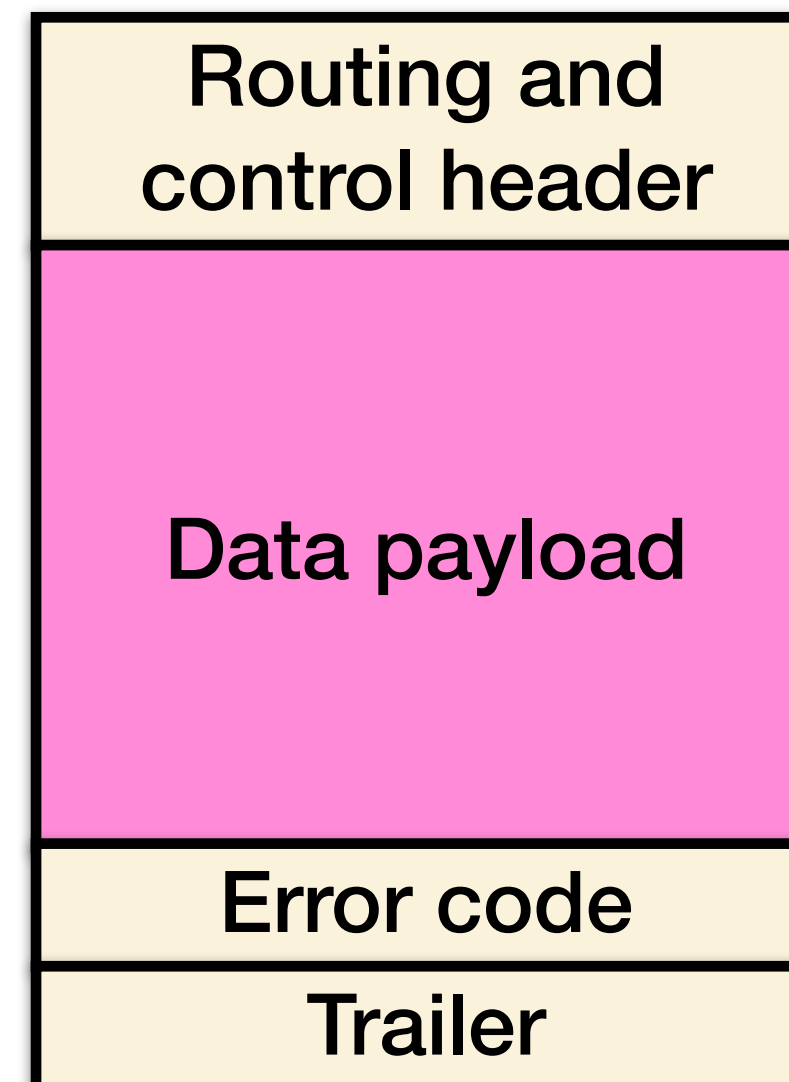
End-to-End Latency (1/2 Roundtrip) Over Time



Latency has not improved as much as transistor counts (i.e., Moore's Law)

Performance Properties: Bandwidth

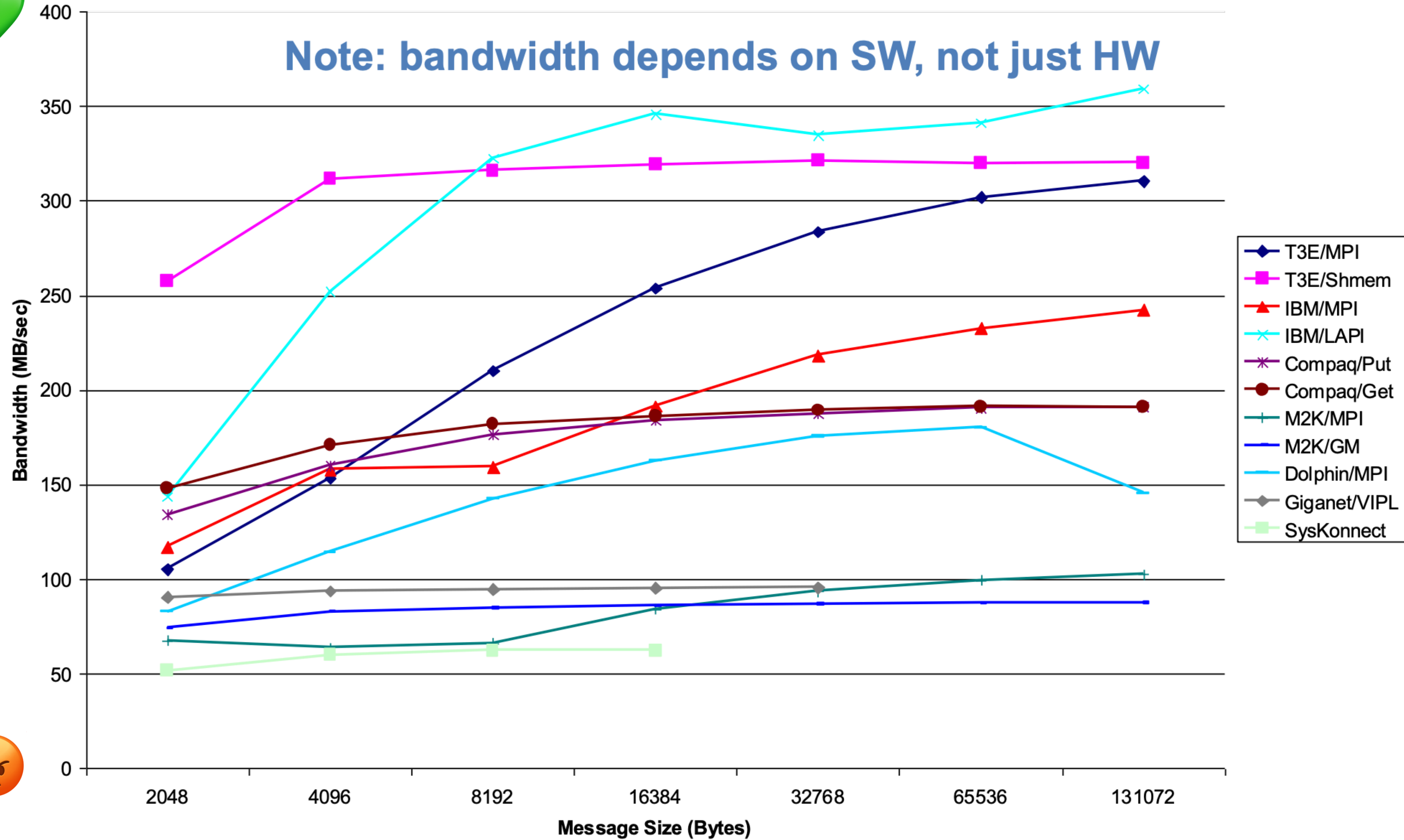
- The bandwidth of a link = # wires / time-per-bit
- Bandwidth typically in **Gigabytes/sec** (GB/s)
- **Effective bandwidth is usually lower** than physical link bandwidth due to packet overhead.
- Bandwidth is important for applications with mostly **large messages**.



Bandwidth Performance



Note: bandwidth depends on SW, not just HW

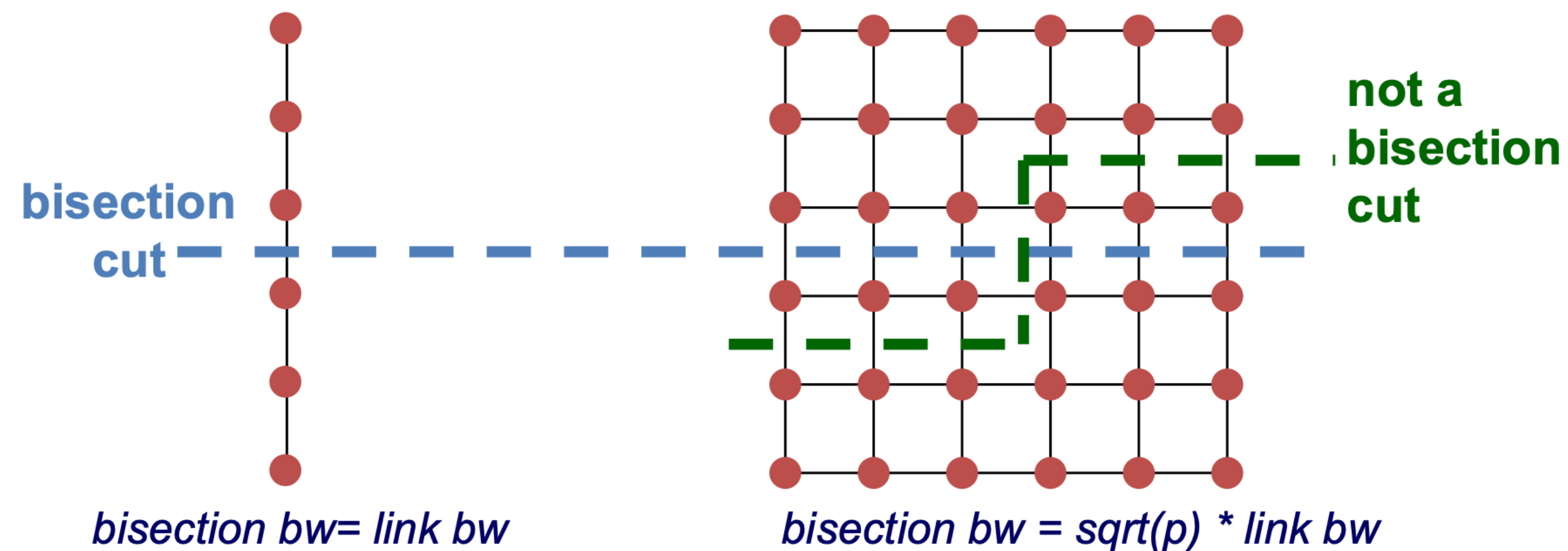


Data from Mike Welcome, NERSC

Performance Properties: Bisection Bandwidth

Bisection bandwidth: **bandwidth across smallest cut** that divides network into two equal halves

Bandwidth across “narrowest” part of the network



Bisection bandwidth is important for algorithms in which **all processors need to communicate with all others.**

Linear and Ring Topologies

Linear array



- Diameter = $n-1$; average distance $\sim n/3$.
- Bisection bandwidth = 1 (in units of link bandwidth).

Torus or Ring

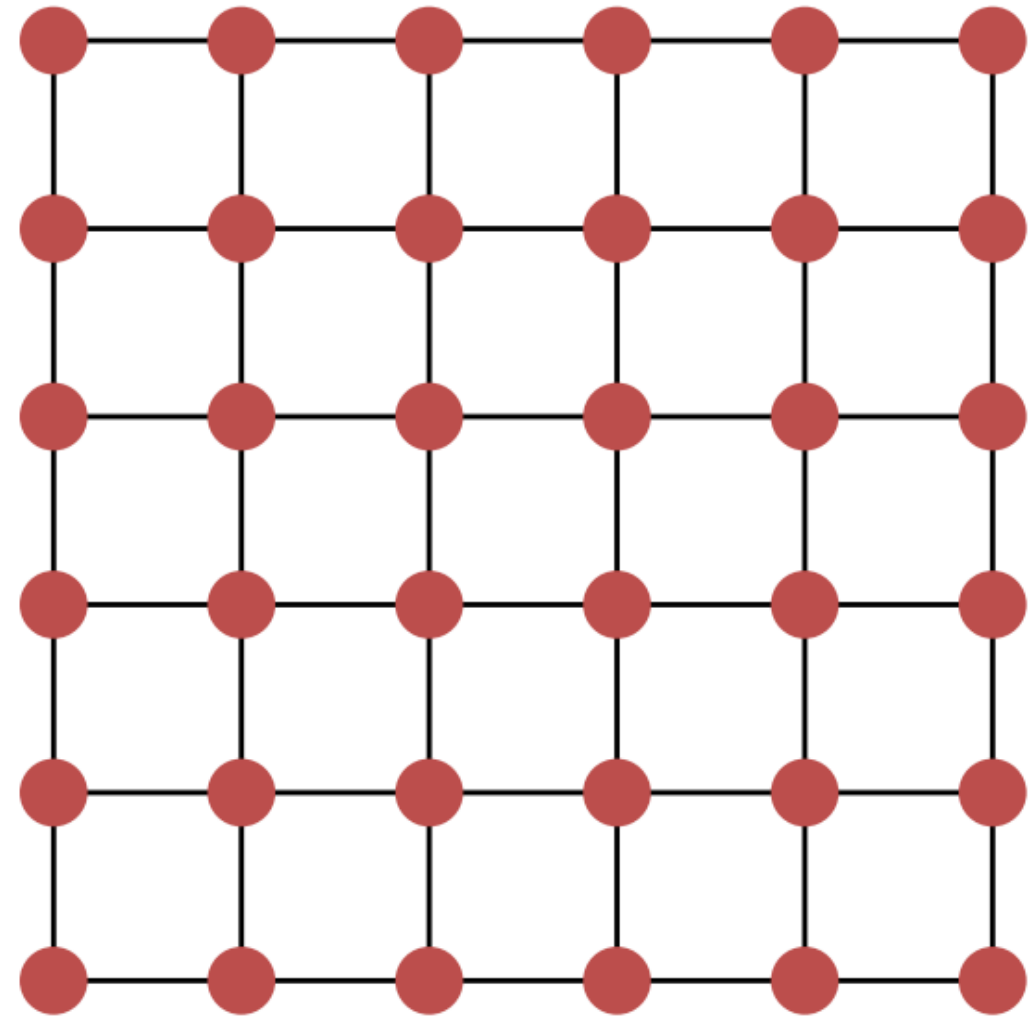


- Diameter = $n/2$; average distance $\sim n/4$.
- Bisection bandwidth = 2.
- Natural for algorithms that work with 1D arrays.

Meshes and Tori

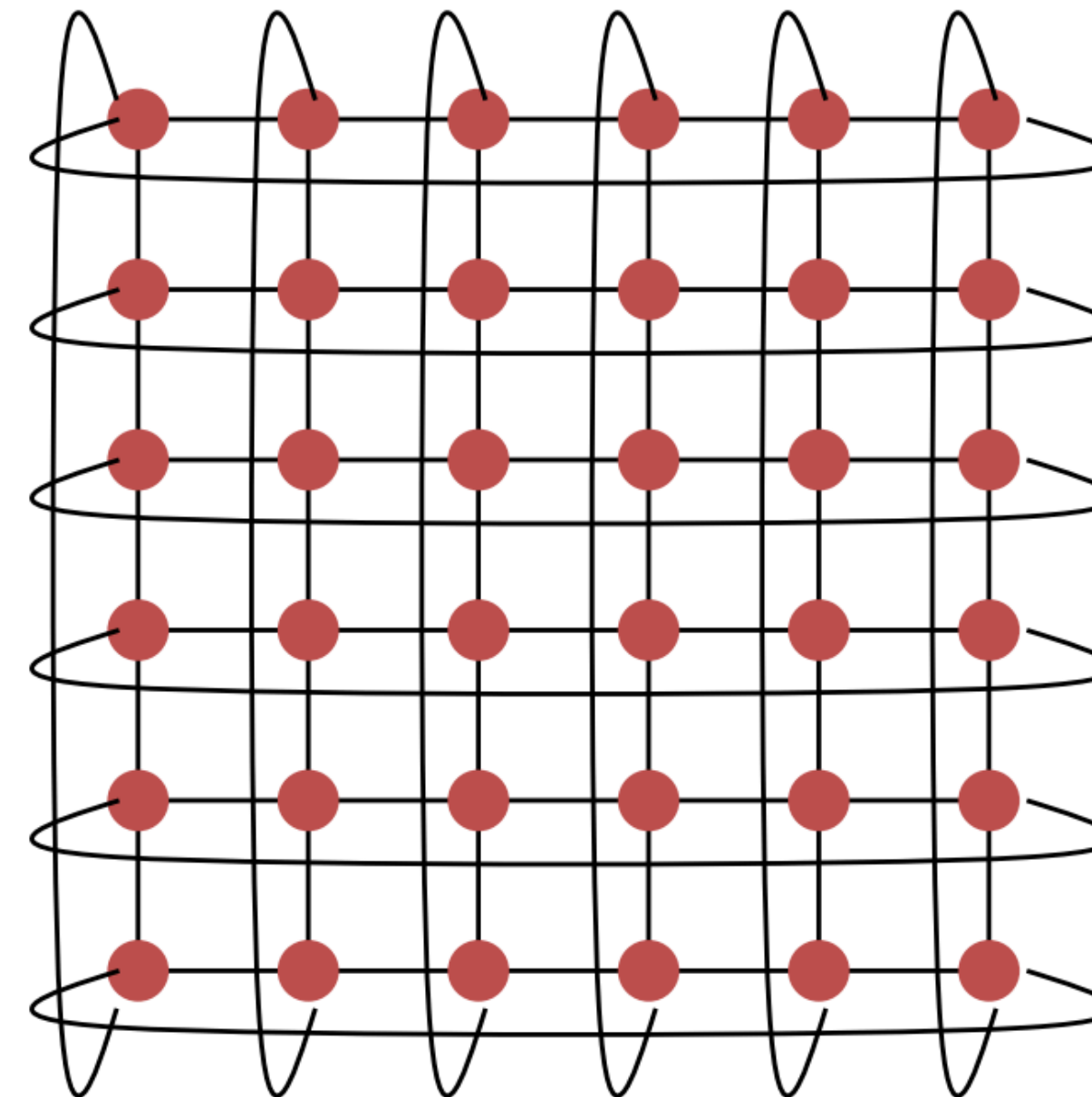
Two dimensional mesh

- Diameter = $2 * (\text{sqrt}(n) - 1)$
- Bisection bandwidth = $\text{sqrt}(n)$



Two dimensional torus

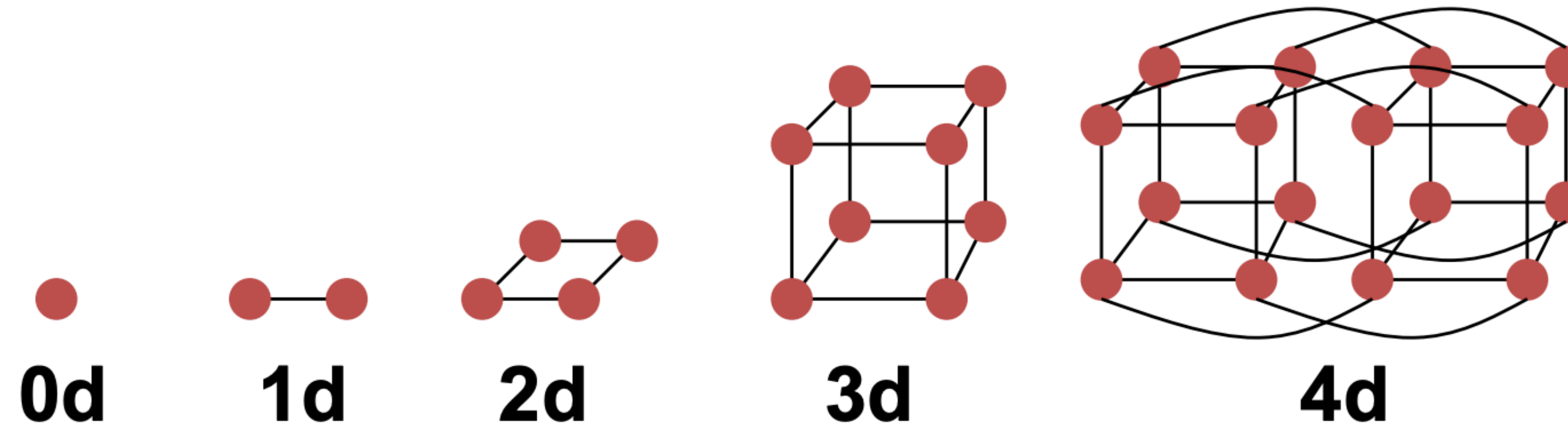
- Diameter = $\text{sqrt}(n)$
- Bisection bandwidth = $2 * \text{sqrt}(n)$



- Generalizes to higher dimensions
 - Cray XT (eg Hopper@NERSC) uses 3D Torus
- Natural for algorithms that work with 2D and/or 3D arrays (matmul)

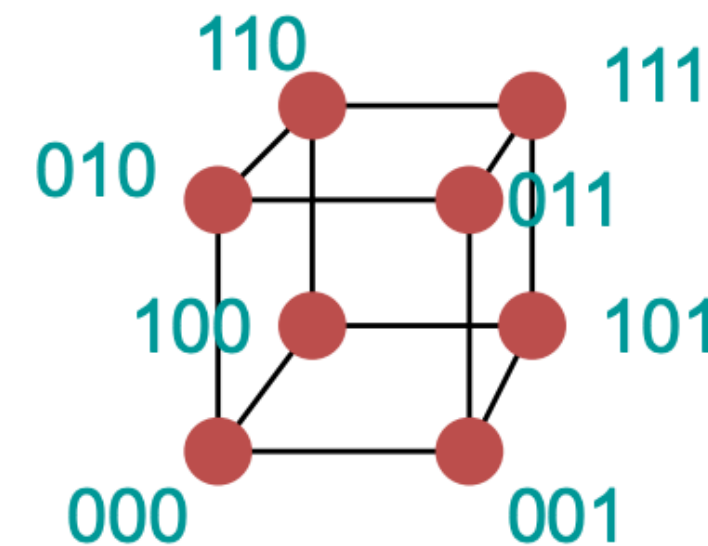
Hypercubes

- **Number of nodes $n = 2^d$ for dimension d .**
 - **Diameter = d .**
 - **Bisection bandwidth = $n/2$.**



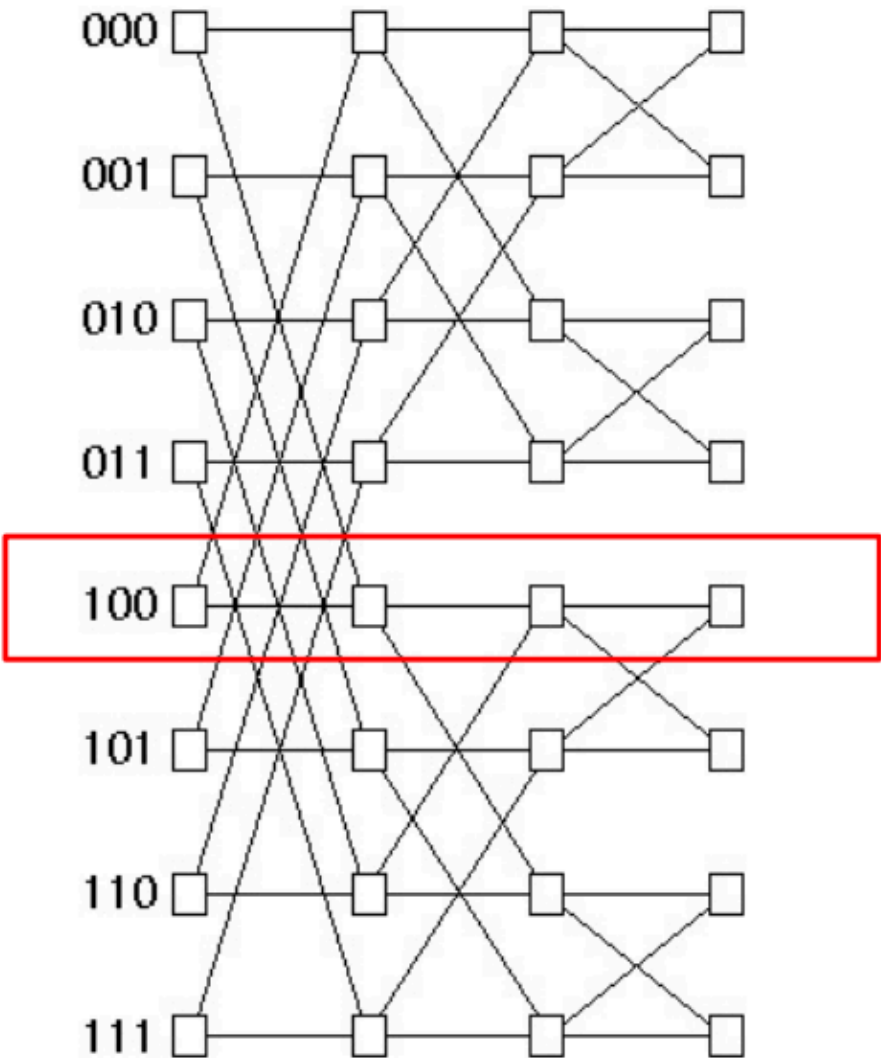
- **Popular in early machines (Intel iPSC, NCUBE).**
 - **Lots of clever algorithms.**
 - **See 1996 online CS267 notes.**

- **Greycode addressing:**
 - **Each node connected to d others with 1 bit different.**

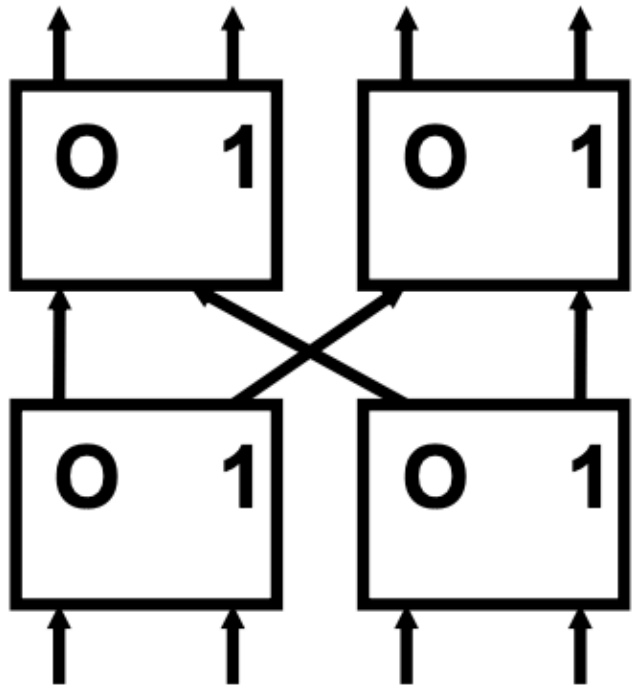
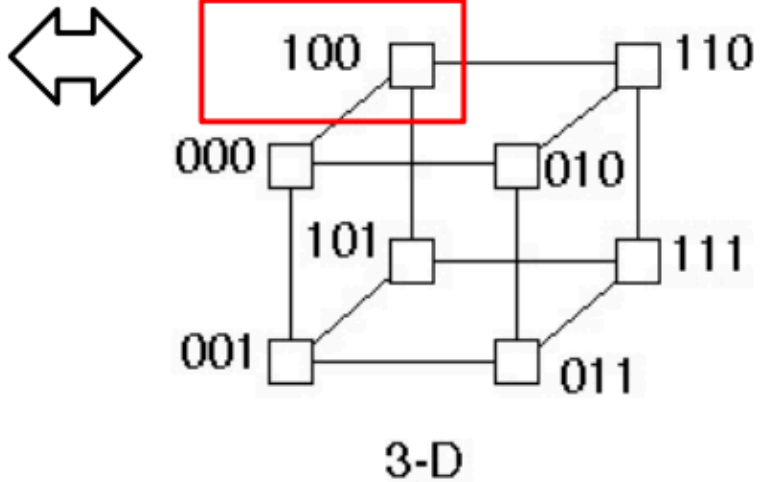


Butterflies

- Really an unfolded version of hypercube.
- A d-dimensional butterfly has $(d+1) 2^d$ "switching nodes" (not to be confused with processors, which is $n = 2^d$)
- Butterfly was invented because hypercube required increasing radix of switches as the network got larger; prohibitive at the time
- Diameter = $\log n$. Bisection bandwidth = n
- No path diversity: bad with adversarial traffic



A row of butterfly is a node in hypercube

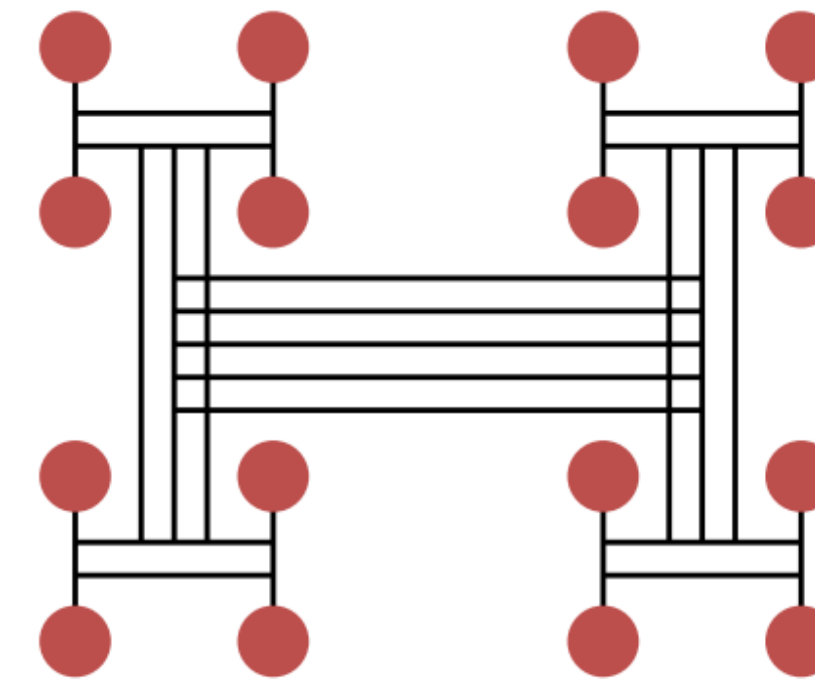
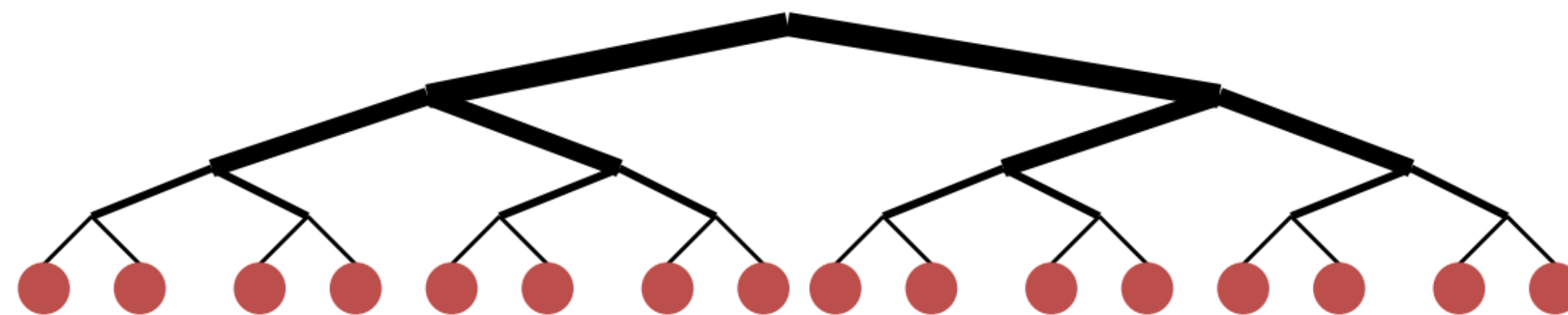
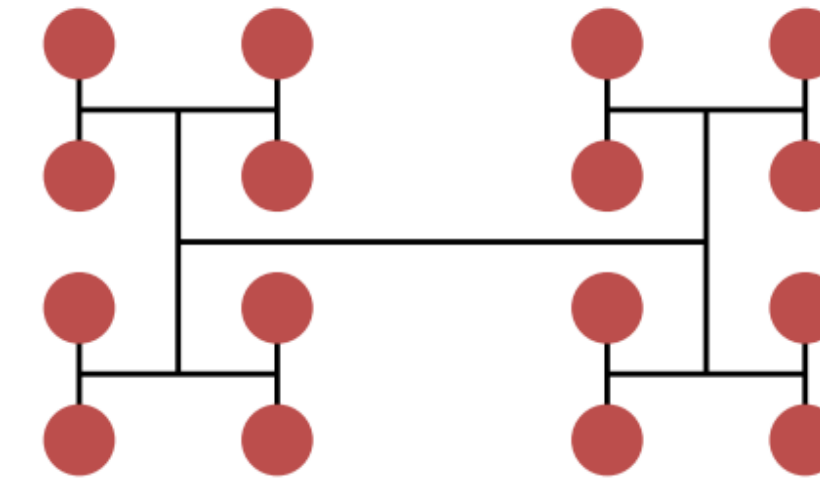


butterfly switch

Ex: to get from proc 101 to 110, Compare bit-by-bit and Switch if they disagree, else not

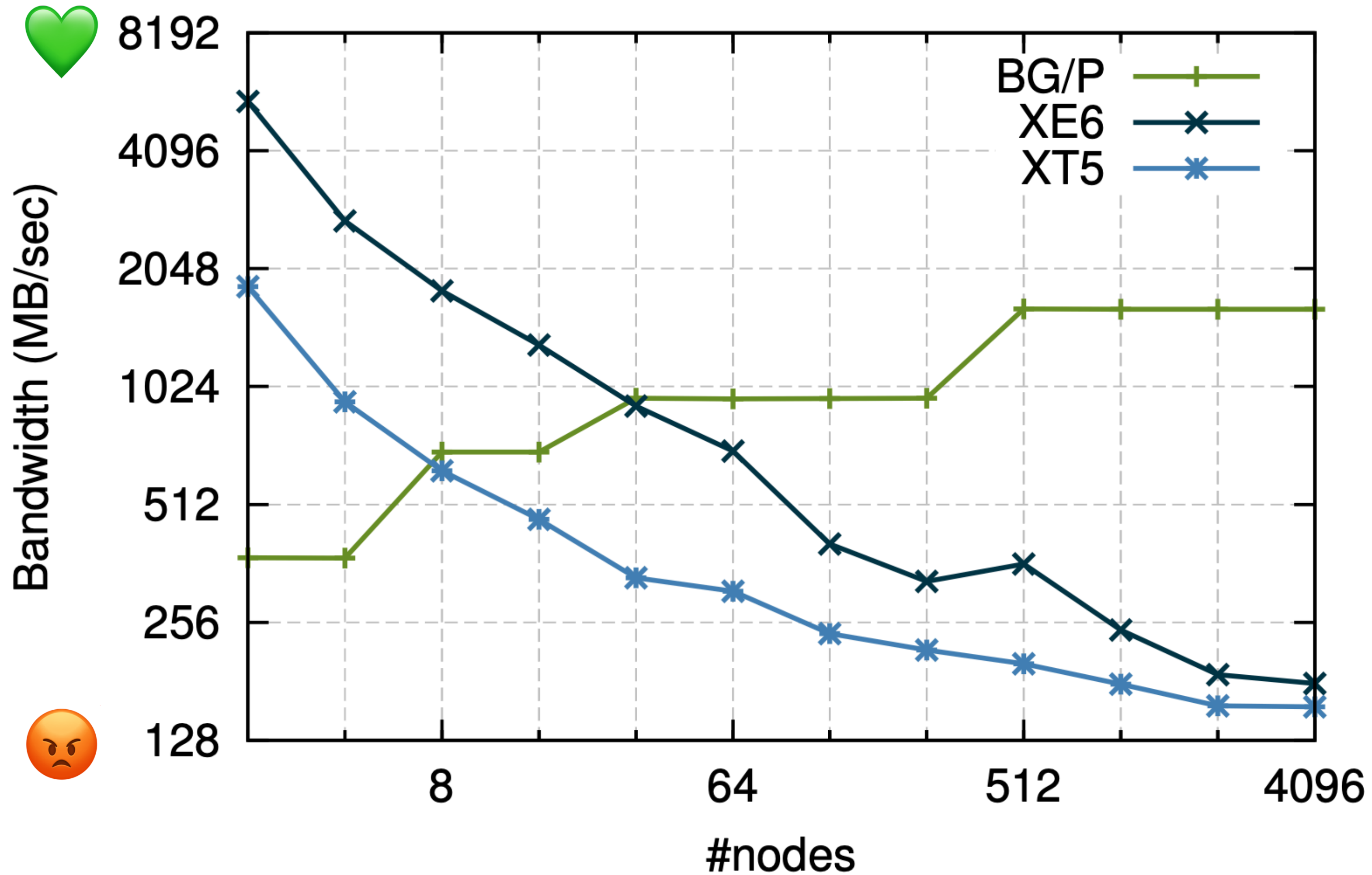
Trees

- **Diameter = $\log n$.**
- **Bisection bandwidth = 1.**
- **Easy layout as planar graph.**
- **Many tree algorithms (e.g., summation).**
- **Fat trees avoid bisection bandwidth problem:**
 - **More (or wider) links near top.**
 - **Example: Thinking Machines CM-5.**



Performance Effect of Topology

1 MB multicast on BG/P, Cray XT5, and Cray XE6



Why so many topologies?

Different systems have different needs depending on the **size of the system** (data center vs. NIC)

Complexity vs. optimality

Physical constraints - **innovations in HW** enable previously infeasible technologies

Two recent technological changes:

- Higher **radix** (number of ports supported) switches economical, which is really a consequence of Moore's law
- **Fiber optic** is feasible -> distance doesn't matter

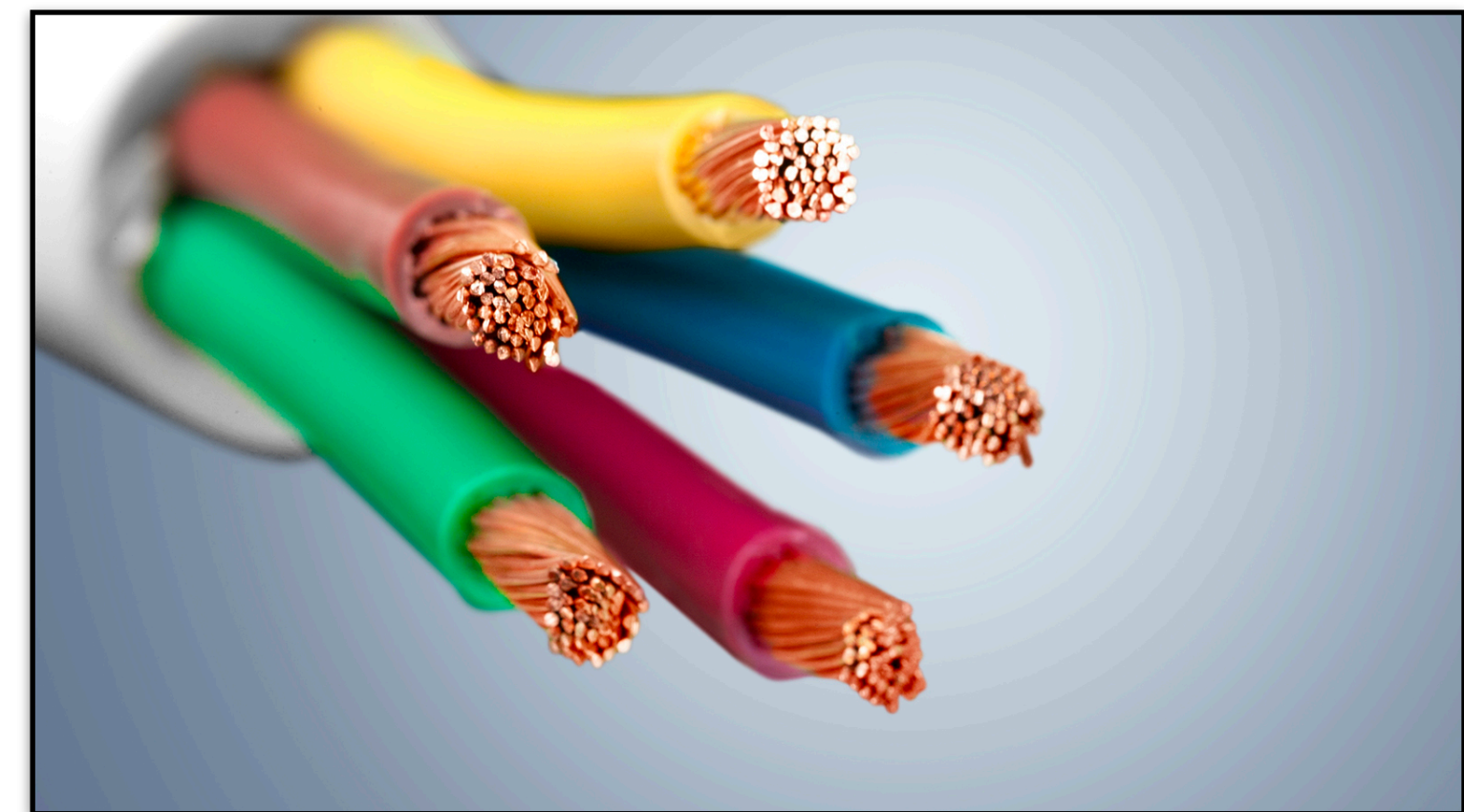
Dragonfly Networks

Motivation: **Exploit gap in cost and performance** between optical interconnects (which go between cabinets in a machine room) and electrical networks (inside cabinet)

- Optical (fiber) more expensive but higher bandwidth when long
- Electrical (copper) networks cheaper, faster when short



<https://www.alliedtelesis.com/us/en/blog/quick-guide-fiber-networking>

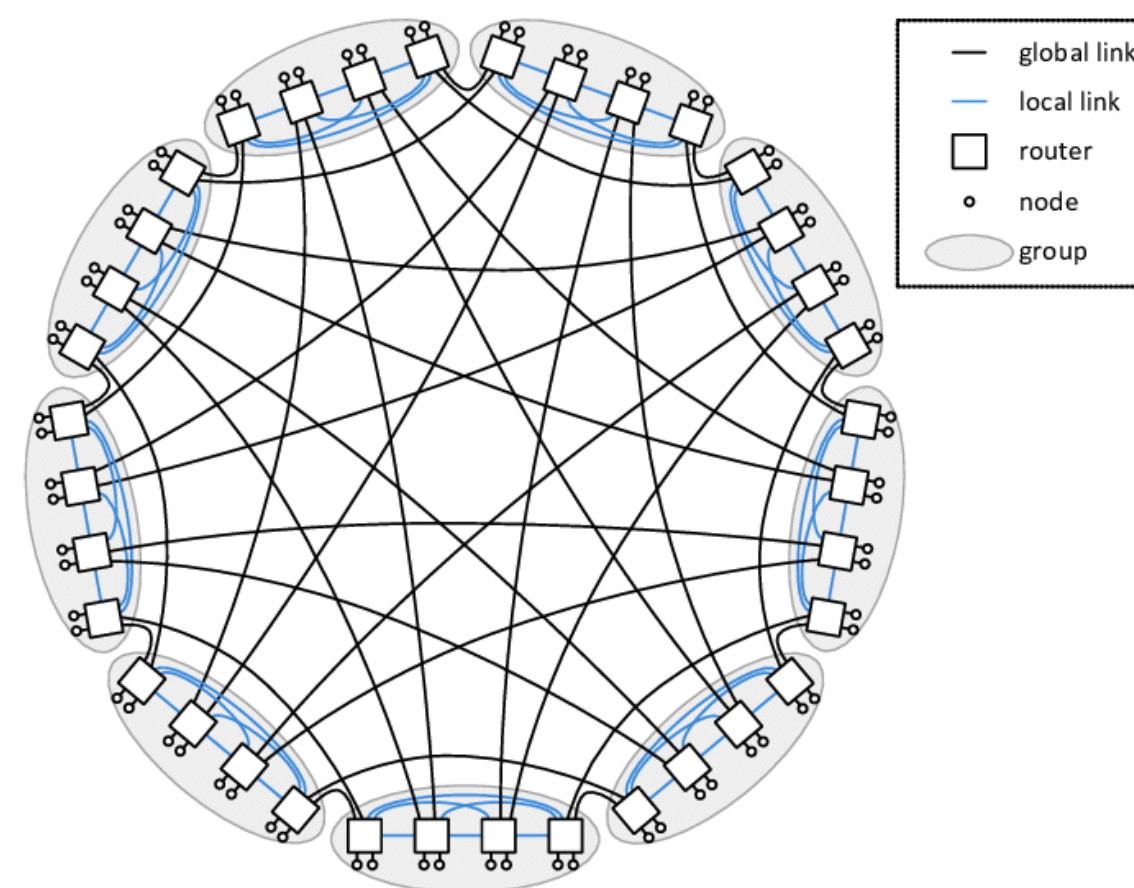


<https://www.actiontec.com/what-is-the-future-of-the-copper-network/>

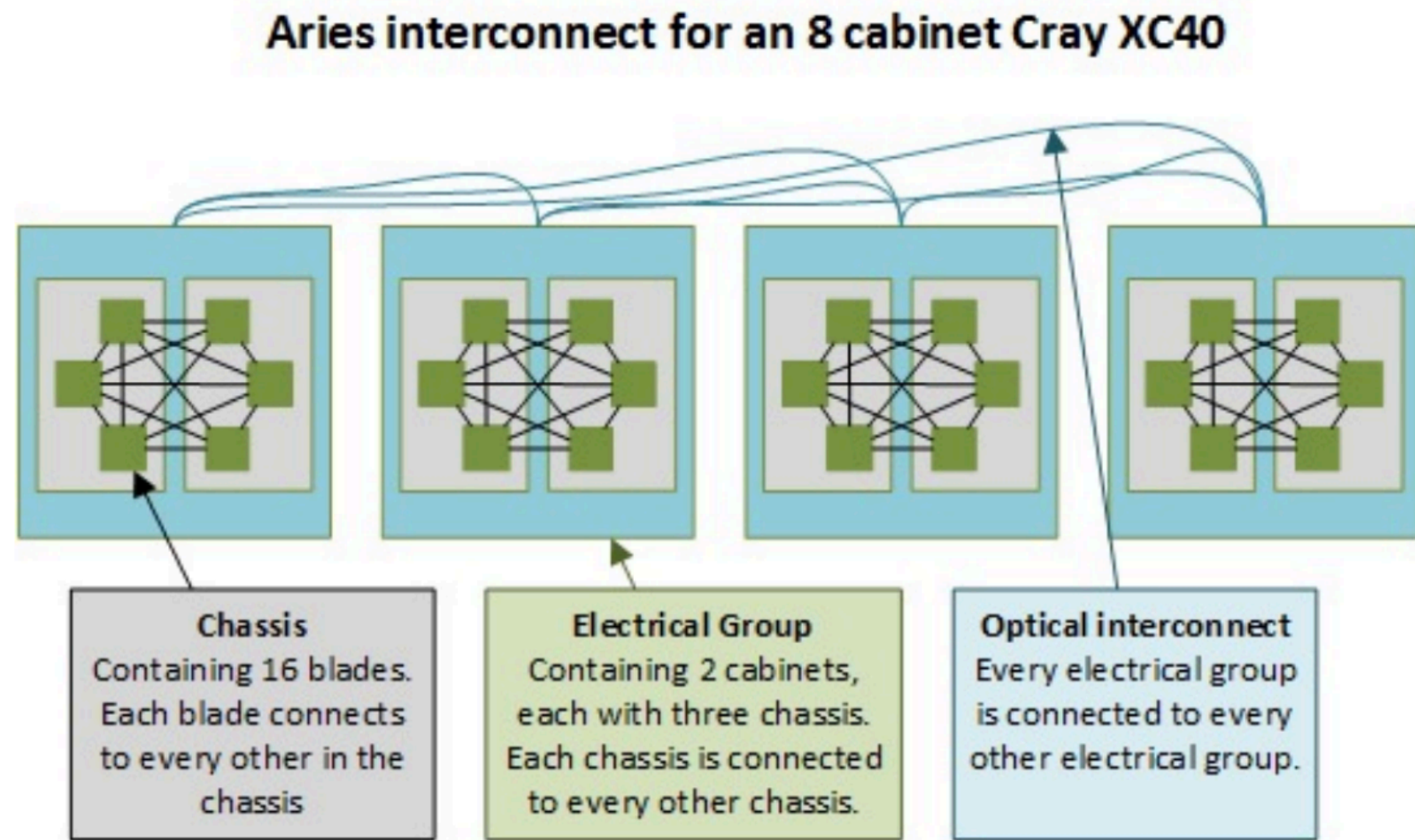
Dragonfly Networks

Combine in **hierarchy**:

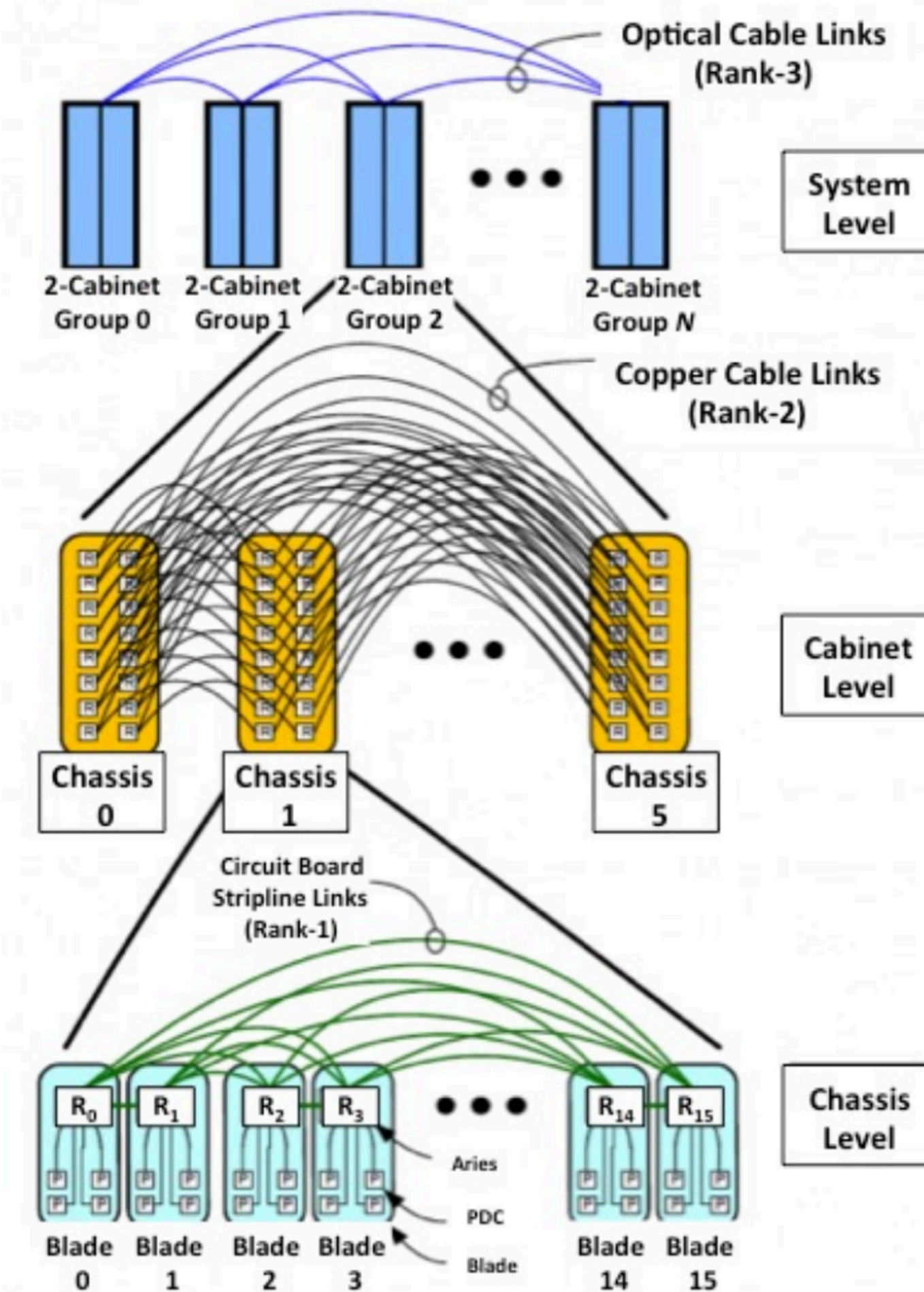
- Several groups are connected together using all to all links, i.e. each group has at least one link directly to each other group.
- The topology inside each group can be any topology.



Dragonfly Examples



Source: European Centre for Medium-Range Weather Forecasts



<https://docs.nersc.gov/systems/cori/interconnect/>

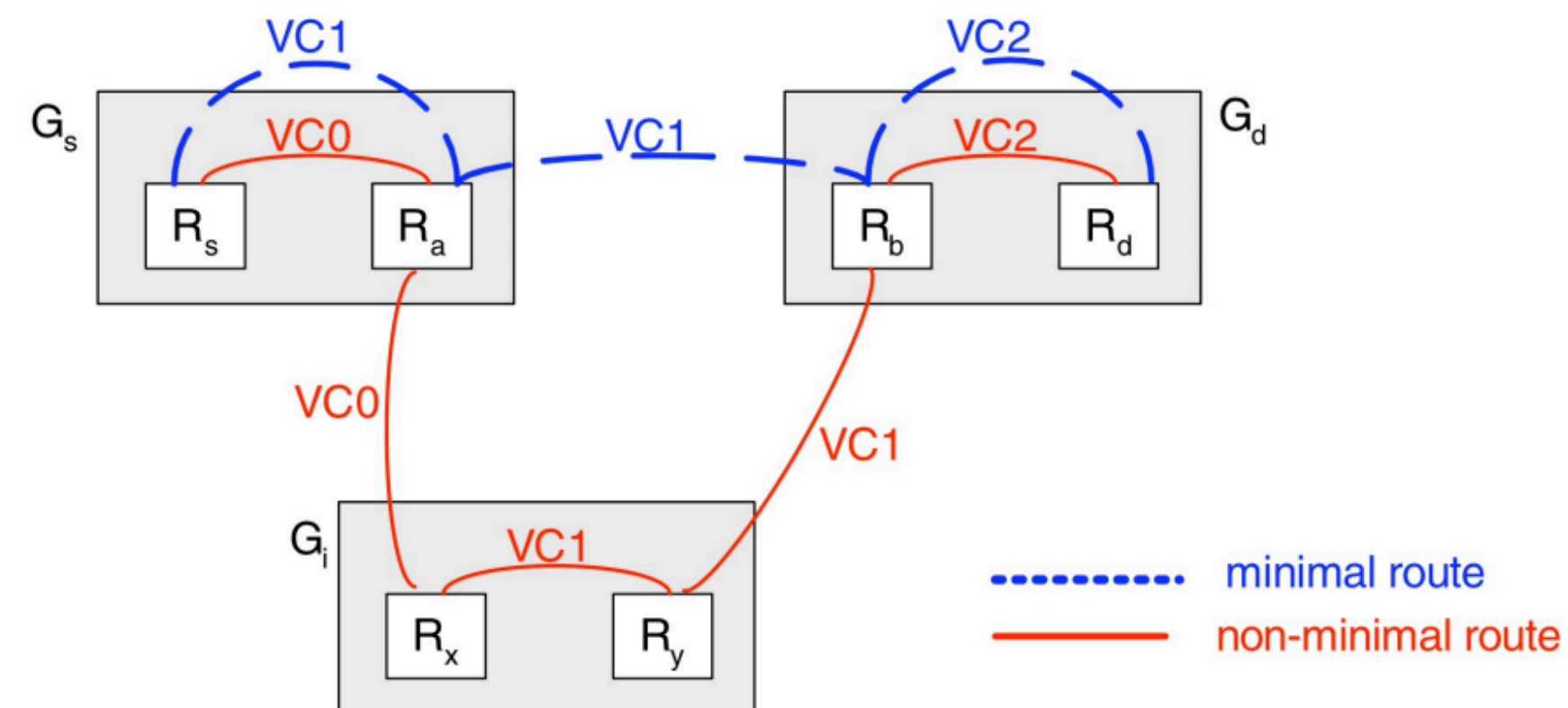
Minimal Routing

Suppose we want to route from source node s attached to router R_s in group G_s to destination node d attached to router R_d in group G_d .

Step 1: If $G_s \neq G_d$ and R_s does not have a connection to G_d , route within G_s from R_s to R_a , a router that has a global channel to G_d .

Step 2: If $G_s \neq G_d$, traverse the global channel from R_a to reach router R_b in G_d .

Step 3: If $R_b \neq R_d$, route within G_d from R_b to R_d .



Good for load-balanced traffic, but bad on **adversarial traffic.**

Randomized Routing

Suppose we want to route from source node s attached to router R_s in group G_s to destination node d attached to router R_d in group G_d .

Idea: First route to some **randomly-selected intermediate group** G_i .

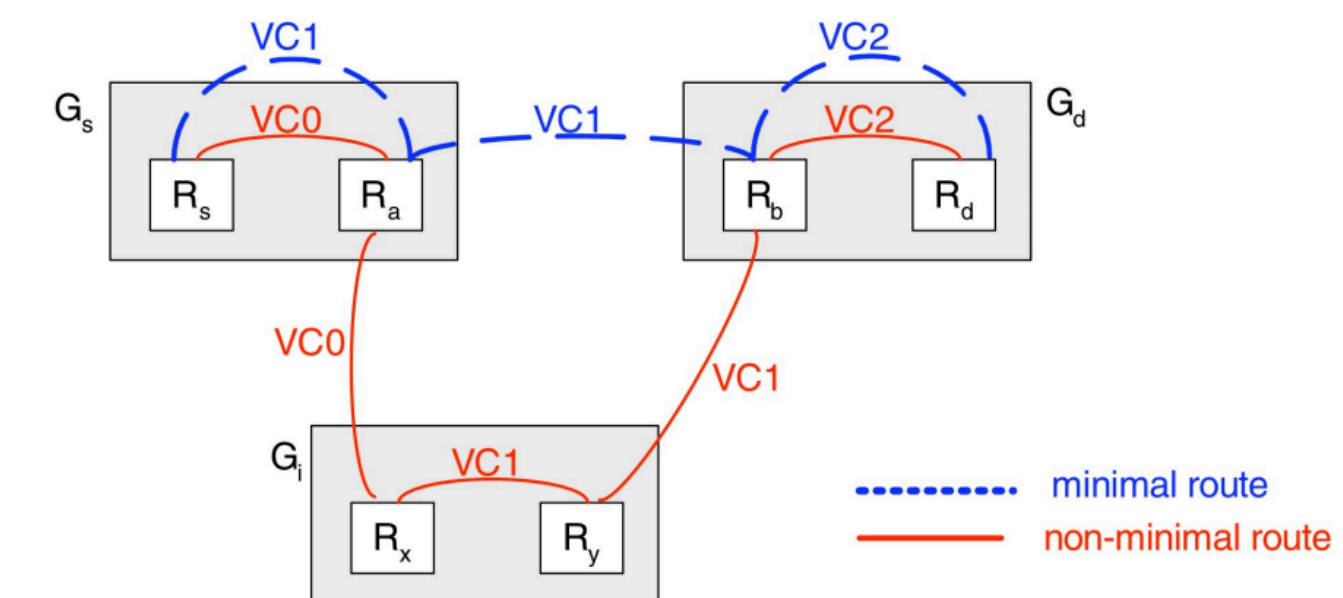
Step 1: If $G_s \neq G_i$ and R_s does not have a connection to G_i , route within G_s from R_s to R_a , a router that has a global channel to G_i .

Step 2: If $G_s \neq G_i$, traverse the global channel from R_a to reach router R_x in G_i .

Step 3: If $G_i \neq G_d$ and R_x does not have a connection to G_d , route within G_i from R_x to R_y , a router that has a global channel to G_d .

Step 4: If $G_i \neq G_d$, traverse the global channel from R_y to R_b in G_d .

Step 5: If $R_b \neq R_d$, route within G_d from R_b to R_d .



Randomized Routing

Suppose we want to route from source node s attached to router R_s in group G_s to destination node d attached to router R_d in group G_d .

Idea: First route to some **randomly-selected intermediate group** G_i .

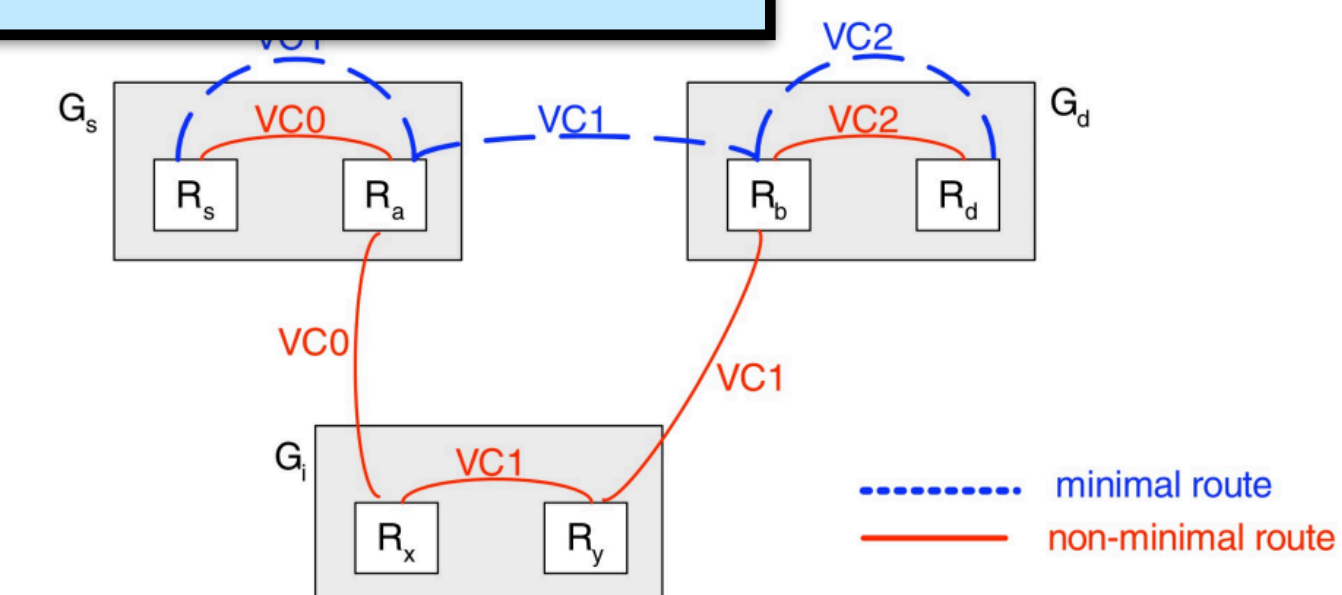
- Programmer can (usually) ignore topology, get good performance - important in virtualized, dynamic environment

- Drawback: variable performance

router that has a global channel to G_d .

Step 4: If $G_i \neq G_d$, traverse the global channel from R_y to R_b in G_d .

Step 5: If $R_b \neq R_d$, route within G_d from R_b to R_d .



Routing Performance

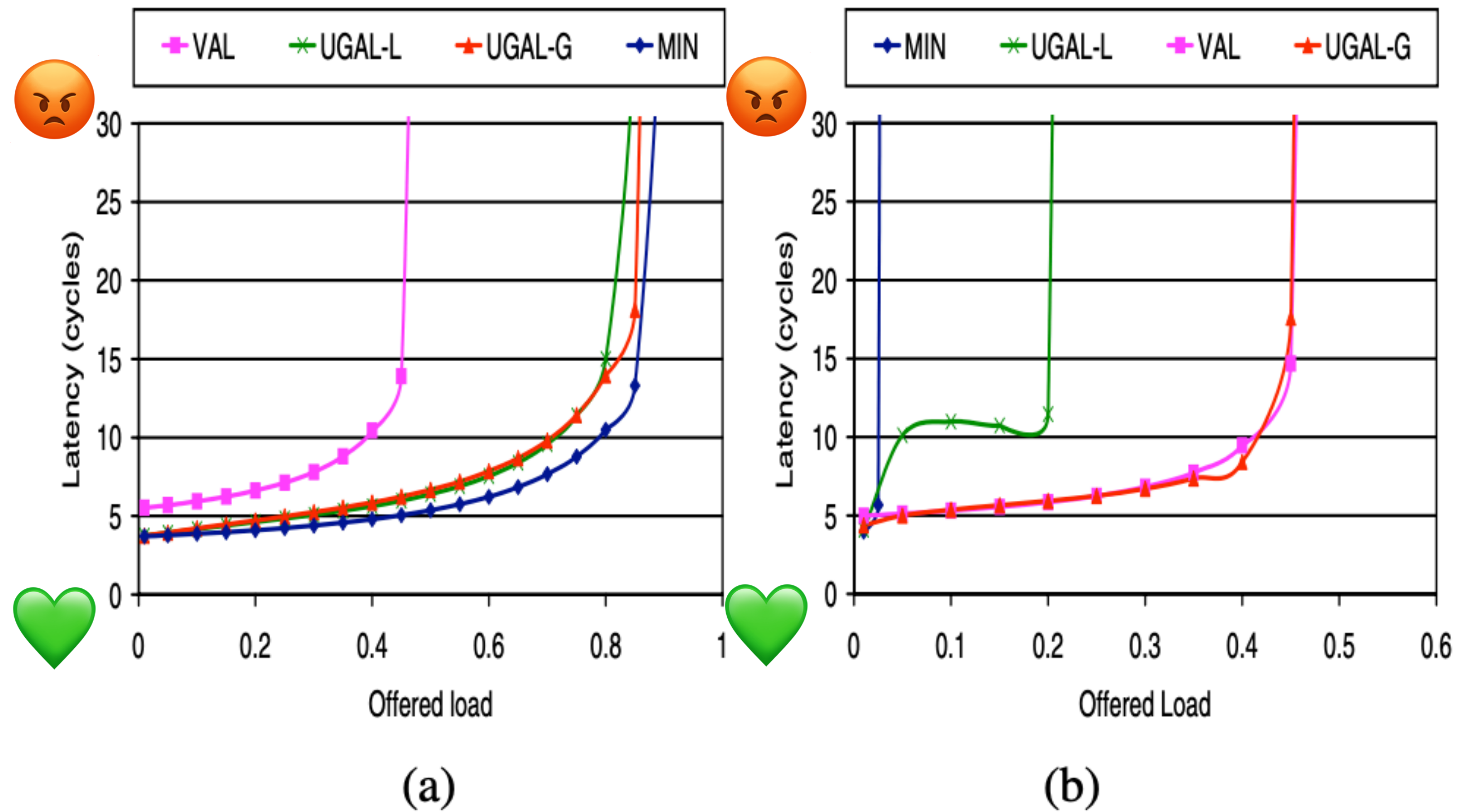
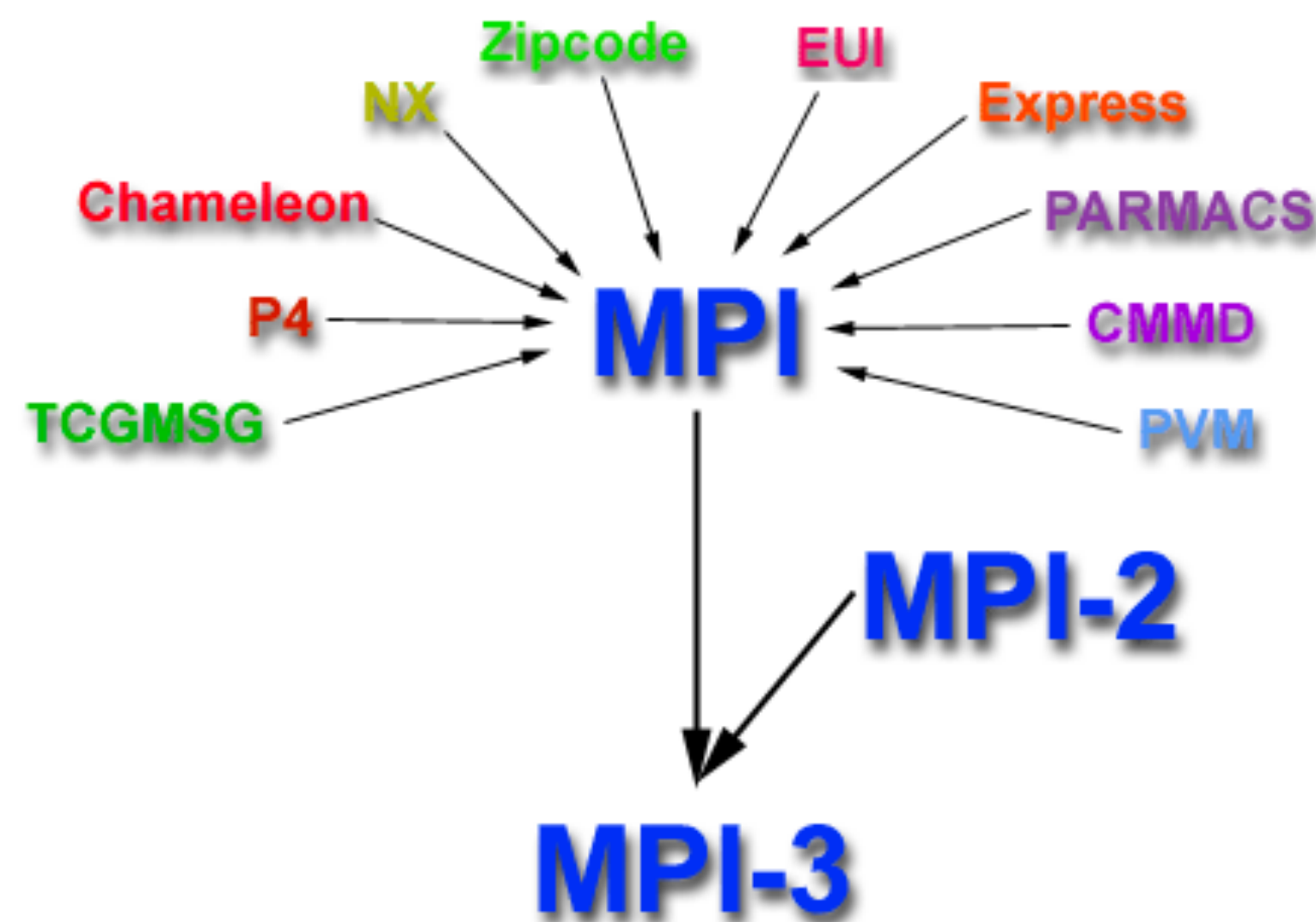


Figure 8. Routing algorithm comparison on the dragonfly for (a) uniform random traffic and (b) adversarial traffic pattern.

Programming Distributed-Memory Machines with Message Passing

Message Passing Interface (MPI)

- In the 1980s-early 90s, there used to be many software tools for writing distributed-memory programs.
- 1994 - MPI 1.0 is released - works on distributed and shared memory
- Standards are needed to write portable code.



Message Passing Libraries

All **communication, synchronization require subroutine calls**

- **No shared variables**
- Program run on a single processor just like any uniprocessor program, except for calls to message passing library

Subroutines for

- **Communication**
 - Pairwise or point-to-point: Send and Receive
 - Collectives all processor get together to
 - Move data: Broadcast, Scatter/gather
 - Compute and move: sum, product, max, prefix sum, ... of data on many processors
- **Synchronization** (barrier)
- **Enquiries** - How many processes? Which one am I? Any messages waiting?

Compile/Run MPI

In your code:

```
#include <mpi.h>

rc = MPI_XXXX(parameter, ...)
```

To compile:

```
$ mpicc mpi_hello_world.c -o mpi_hello_world
```

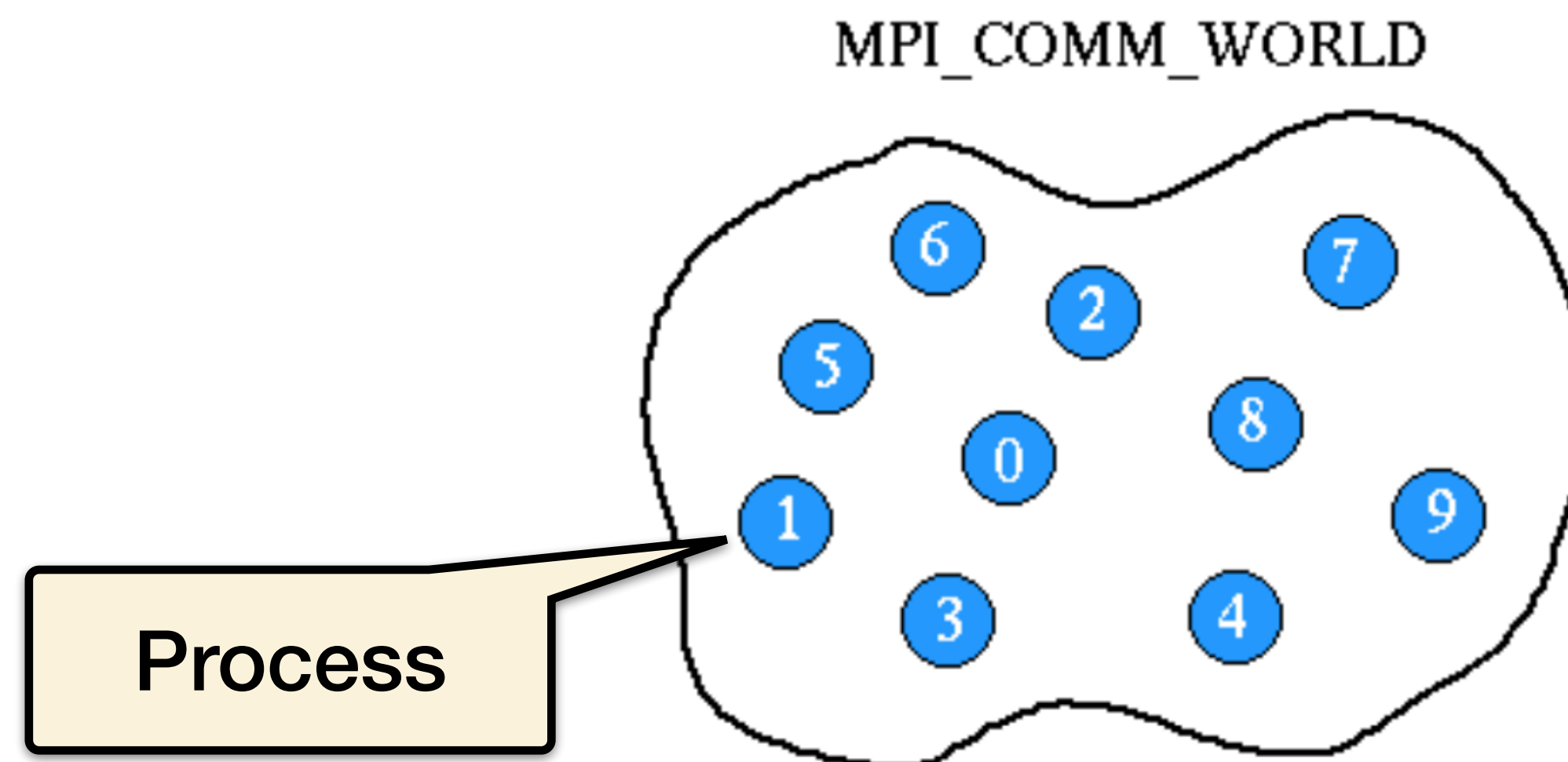
To run:

```
$ srun mpi_hello_world
```

(The default implementation of MPI on PACE ICE is mvapich2.
Different implementations (e.g., OpenMPI) may have slightly different usage)

MPI Communicator

- An **MPI communicator**: a "communication universe" for a group of processes
- `MPI_COMM_WORLD` – name of the default MPI communicator, i.e., the collection of all processes
- Each process in a communicator is identified by its **rank**
- Almost every MPI command needs to provide **a communicator as input argument**



MPI Process Rank

- Each process has a unique **rank**, i.e. an integer identifier, within a communicator
- The rank value is between 0 and #procs-1
- The rank value is used to distinguish one process from another
- Commands `MPI_Comm_size` & `MPI_Comm_rank` are very useful

Example:

```
int size, my_rank;

MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

if (my_rank==0) { ... }
```

The 6 Most Important MPI Commands

- `MPI_Init` - initiate an MPI computation
- `MPI_Finalize` - terminate the MPI computation and clean up
- `MPI_Comm_size` - how many processes participate in a given MPI communicator?
- `MPI_Comm_rank` - which one am I? (A number between 0 and size-1.)
- `MPI_Send` - send a message to a particular process within an MPI communicator
- `MPI_Recv` - receive a message from a particular process within an MPI communicator

MPI Hello World Example

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

int main(int argc, char** argv) {
    MPI_Init(NULL, NULL); // Initialize the MPI environment.

    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size); // Get the number of processes

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank); // Get the rank of the process

    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len); // Get the name of the processor

    printf("Hello world from processor %s, rank %d out of %d processors\n",
           processor_name, world_rank, world_size); // Print hello world message

    // Finalize the MPI environment. No more MPI calls can be made after this
    MPI_Finalize();
}
```


MPI Hello World Example

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

int main(int argc, char** argv) {
    MPI_Init(NULL, NULL); // Initialize the MPI environment.

    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size); // Get the number of processes

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank); // Get the rank of the process

    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len); // Get the name of the processor

    printf("Hello world from processor %s, rank %d out of %d processors\n",
           processor_name, world_rank, world_size); // Print hello world message

    // Finalize the MPI environment. No more MPI calls can be made after this
    MPI_Finalize();
}
```

Initialize environment

MPI Hello World Example

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

int main(int argc, char** argv) {
    MPI_Init(NULL, NULL); // Initialize the MPI environment.

    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank); // Get the rank of the process

    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len); // Get the name of the processor

    printf("Hello world from processor %s, rank %d out of %d processors\n",
           processor_name, world_rank, world_size); // Print hello world message

    // Finalize the MPI environment. No more MPI calls can be made after this
    MPI_Finalize();
}
```

Initialize environment

How many processors are there?

MPI Hello World Example

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

Initialize environment

```
int main(int argc, char** argv) {
    MPI_Init(NULL, NULL); // Initialize the MPI environment.
```

```
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
```

How many processors are there?

```
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

What is my rank?

```
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len); // Get the name of the processor
```

```
    printf("Hello world from processor %s, rank %d out of %d processors\n",
           processor_name, world_rank, world_size); // Print hello world message
```

```
    // Finalize the MPI environment. No more MPI calls can be made after this
    MPI_Finalize();
```

```
}
```

MPI Hello World Example

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

Initialize environment

```
int main(int argc, char** argv) {
    MPI_Init(NULL, NULL); // Initialize the MPI environment.
```

```
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
```

How many processors are there?

```
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

What is my rank?

the process

```
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);
```

What is my name?

processor

```
    printf("Hello world from processor %s, rank %d out of %d processors\n",
           processor_name, world_rank, world_size); // Print hello world message
```

```
    // Finalize the MPI environment. No more MPI calls can be made after this
    MPI_Finalize();
```

```
}
```


MPI Hello World Example

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

Initialize environment

```
int main(int argc, char** argv) {
    MPI_Init(NULL, NULL); // Initialize the MPI environment.
```

```
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
```

How many processors are there?

```
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

What is my rank?

```
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);
```

What is my name?

```
    printf("Hello world from processor %s, rank %d out of %d processors\n",
           processor_name, world_rank, world_size);
```

Print name, rank, total processor count

```
    // Finalize the MPI environment. No more MPI calls can be made after this
    MPI_Finalize();
```

```
}
```

MPI Hello World Example

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

Initialize environment

```
int main(int argc, char** argv) {
    MPI_Init(NULL, NULL); // Initialize the MPI environment.
```

```
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
```

How many processors are there?

```
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

What is my rank?

```
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);
```

What is my name?

```
    printf("Hello world from processor %s, rank %d out of %d processors\n",
           processor_name, world_rank, world_size);
```

Print name, rank, total processor count

```
    // Finalize the MPI environment. No more MPI calls can be made after this
    MPI_Finalize();
```

Finalize after we are done

```
}
```

Parallel Execution Model

The same MPI program is executed concurrently **on each process**

Process 0

Process 1

Process P-1

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

int main(int argc, char** argv) {
    MPI_Init(NULL, NULL); // Initialize the MPI environment.

    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size); // Get the number
of processes

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank); // Get the rank of
the process

    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len); // Get the
name of the processor

    printf("Hello world from processor %s, rank %d out of %d
processors\n",
           processor_name, world_rank, world_size); // Print hello
world message

    // Finalize the MPI environment. No more MPI calls can be made
after this
    MPI_Finalize();
}
```

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

int main(int argc, char** argv) {
    MPI_Init(NULL, NULL); // Initialize the MPI environment.

    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size); // Get the number
of processes

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank); // Get the rank of
the process

    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len); // Get the
name of the processor

    printf("Hello world from processor %s, rank %d out of %d
processors\n",
           processor_name, world_rank, world_size); // Print hello
world message

    // Finalize the MPI environment. No more MPI calls can be made
after this
    MPI_Finalize();
}
```

...

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

int main(int argc, char** argv) {
    MPI_Init(NULL, NULL); // Initialize the MPI environment.

    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size); // Get the number
of processes

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank); // Get the rank of
the process

    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len); // Get the
name of the processor

    printf("Hello world from processor %s, rank %d out of %d
processors\n",
           processor_name, world_rank, world_size); // Print hello
world message

    // Finalize the MPI environment. No more MPI calls can be made
after this
    MPI_Finalize();
}
```

MPI Hello World Example Output

How to run it:

```
$ salloc -N2 --ntasks-per-node=4 -t1:00:00  
salloc: Pending job allocation 1471  
salloc: job 1471 queued and waiting for resources  
$ srun mpi_hello_world
```

2 node * 4 processes per node = 8

Example output:

Order not guaranteed

```
Hello world from processor at10, rank 0 out of 8 processors  
Hello world from processor at10, rank 2 out of 8 processors  
Hello world from processor at10, rank 3 out of 8 processors  
Hello world from processor at11, rank 4 out of 8 processors  
Hello world from processor at11, rank 7 out of 8 processors  
Hello world from processor at10, rank 1 out of 8 processors  
Hello world from processor at11, rank 5 out of 8 processors  
Hello world from processor at11, rank 6 out of 8 processors
```


Synchronization

- Many parallel algorithms require that **no process proceeds before all the processes** have reached the same state at certain points of a program.

- Explicit synchronization :

```
int MPI_Barrier (MPI_Comm comm)
```

- Implicit synchronization through use of e.g. pairs of MPI_Send and MPI_Recv.

- Ask yourself the following question: *“If Process 1 progresses 100 times faster than Process 2, will the final result still be correct?”*

Example: Ordered Output

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

int main(int argc, char** argv) {
    // init as before

    for(int i = 0; i < world_size; i++) {
        MPI_Barrier(MPI_COMM_WORLD);
        if (i == world_rank) {
            printf("Hello world from processor %s, rank %d out of %d processors\n",
                processor_name, world_rank, world_size); // Print hello world message
            fflush(stdout);
        }
    }

    // Finalize the MPI environment. No more MPI calls can be made after this
    MPI_Finalize();
}
```

Iterate over processor count

Example: Ordered Output

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

int main(int argc, char** argv) {
    // init as before

    for(int i = 0; i < world_size; i++) {
        MPI_Barrier(MPI_COMM_WORLD);
        if (i == world_rank) {
            printf("Hello world from processor %s, rank %d out of %d processors\n",
                processor_name, world_rank, world_size); // Print hello world message
            fflush(stdout);
        }
    }

    // Finalize the MPI environment. No more MPI calls can be made after this
    MPI_Finalize();
}
```

Iterate over processor count

Synchronization

Example: Ordered Output

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

int main(int argc, char** argv) {
    // init as before

    for(int i = 0; i < world_size; i++) {
        MPI_Barrier(MPI_COMM_WORLD);
        if (i == world_rank) {
            printf("Hello world from processor %s, rank %d out of %d processors\n",
                processor_name, world_rank, world_size); // Print hello world message
            fflush(stdout);
        }
    }

    // Finalize the MPI environment. No more MPI calls can be made after this
    MPI_Finalize();
}
```

Iterate over processor count

Synchronization

Only print if we are on our rank

Example: Ordered Output

The processes synchronize between themselves P times:

Process 0

Process 1

Process P-1

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

int main(int argc, char** argv) {
    // init as before

    for(int i = 0; i < size; i++) {
        MPI_Barrier(MPI_COMM_WORLD);
        if (i == world_rank) {
            printf("Hello world from processor %s, rank %d out of
%d processors\n",
                processor_name, world_rank, world_size); // Print
hello world message
                fflush(stdout);
        }
    }

    // Finalize the MPI environment. No more MPI calls can be made
after this
    MPI_Finalize();
}
```

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

int main(int argc, char** argv) {
    // init as before

    for(int i = 0; i < size; i++) {
        MPI_Barrier(MPI_COMM_WORLD);
        if (i == world_rank) {
            printf("Hello world from processor %s, rank %d out of
%d processors\n",
                processor_name, world_rank, world_size); // Print
hello world message
                fflush(stdout);
        }
    }

    // Finalize the MPI environment. No more MPI calls can be made
after this
    MPI_Finalize();
}
```

...

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

int main(int argc, char** argv) {
    // init as before

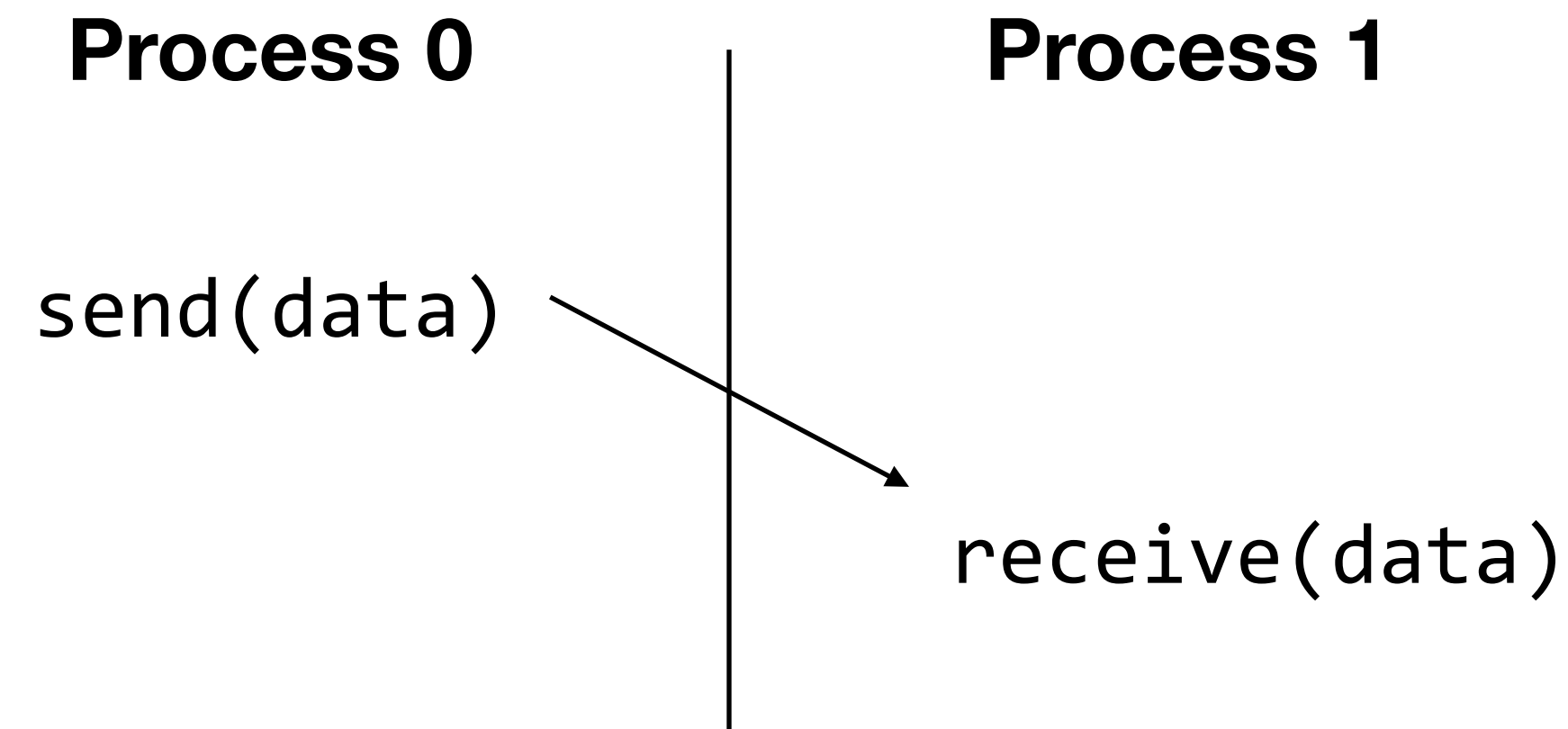
    for(int i = 0; i < size; i++) {
        MPI_Barrier(MPI_COMM_WORLD);
        if (i == world_rank) {
            printf("Hello world from processor %s, rank %d out of
%d processors\n",
                processor_name, world_rank, world_size); // Print
hello world message
                fflush(stdout);
        }
    }

    // Finalize the MPI environment. No more MPI calls can be made
after this
    MPI_Finalize();
}
```

Output:

```
Hello world from processor at10, rank 0 out of 8 processors
Hello world from processor at10, rank 1 out of 8 processors
Hello world from processor at10, rank 2 out of 8 processors
Hello world from processor at10, rank 3 out of 8 processors
Hello world from processor at11, rank 4 out of 8 processors
Hello world from processor at11, rank 5 out of 8 processors
Hello world from processor at11, rank 6 out of 8 processors
Hello world from processor at11, rank 7 out of 8 processors
```

MPI Basic Send/Receive



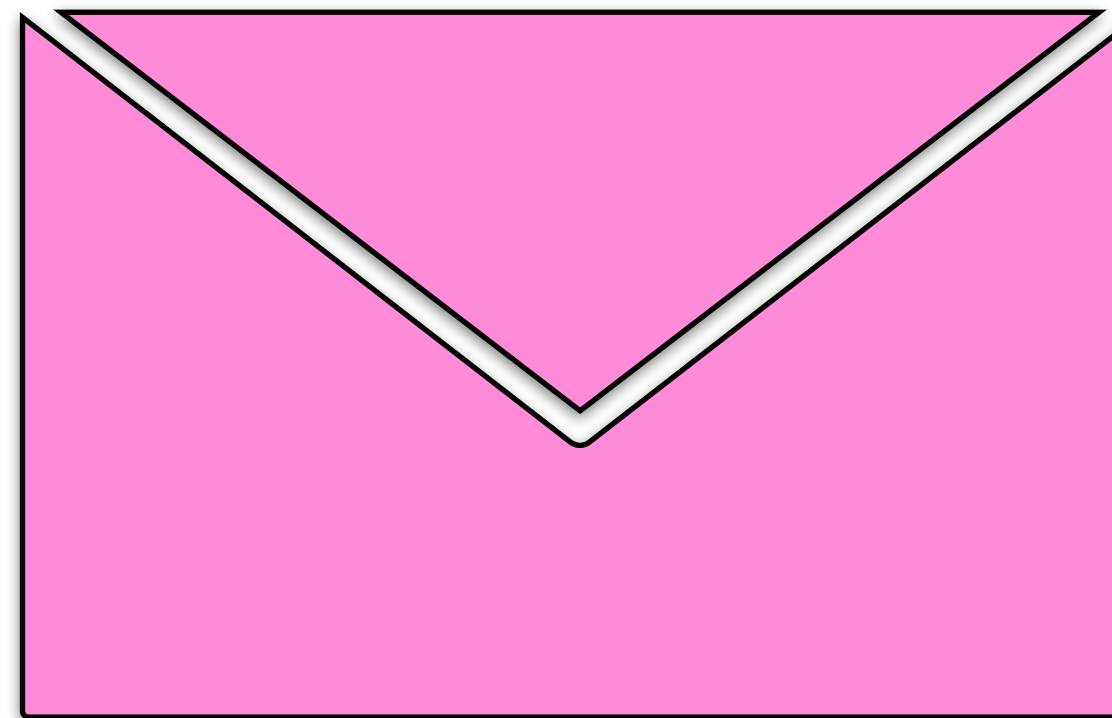
Things that need specifying:

- How will “data” be described?
- How will processes be identified?
- How will the receiver recognize/screen messages?
- What will it mean for these operations to complete?

MPI Message

An MPI message is an array of data elements "inside an envelope"

- Data: start address of the message buffer, counter of elements in the buffer, data type
- Envelope: source/destination process, message tag, communicator



MPI Datatypes

- The data in a message to send or receive is described by a triple (address, count, datatype), where
- An MPI datatype is recursively defined as:
 - **predefined**, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
 - a contiguous array of MPI datatypes
 - a strided block of datatypes
 - an indexed array of blocks of datatypes
 - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, in particular ones for subarrays
- May hurt performance if datatypes are complex

The Simplest MPI Send Command

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

Returns only when communication is finished

This **blocking send function** returns when the data has been delivered to the system and the buffer can be reused. The message may not have been received by the destination process.

The Simplest MPI Receive Command

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

- This **blocking receive function** waits until a matching message is received from the system so that the buffer contains the incoming message.
- Match of data type, source process (or `MPI_ANY_SOURCE`), message tag (or `MPI_ANY_TAG`).
- Receiving fewer datatype elements than count is ok, but receiving more is an error.

A Simple MPI Send/Receive Program

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

int main( int argc, char *argv[] ) {
    int rank, buf;
    MPI_Status status;
    MPI_Init(&argv, &argc);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );

    /* Process 0 sends and Process 1 receives
    if (rank == 0) {
        buf = 123456;
        MPI_Send( &buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
    } else if (rank == 1) {
        MPI_Recv( &buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
        printf( "Received %d\n ", buf );
    }
    MPI_Finalize();
    return 0;
}
```

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

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

MPI Tags

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

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

Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message.

Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying `MPI_ANY_TAG` as the tag in a receive.

MPI Status

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

In C, the `MPI_Status` is a structure that contains at least 3 attributes: `MPI_SOURCE`, `MPI_TAG` and `MPI_ERROR`.

- `MPI_SOURCE` contains the rank of the process that sent the message.
- `MPI_TAG` contains the tag of the message received.
- `MPI_ERROR` contains the error code of the receive operation.

Querying MPI_Status

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

In `MPI_Get_count`, the user passes the `MPI_Status` structure, the `datatype` of the message, and `count` is returned. The `count` variable is the total number of `datatype` elements that were received.

`MPI_Recv` can take `MPI_ANY_SOURCE` for the rank of the sender and `MPI_ANY_TAG` for the tag, so `MPI_Status` would be the only way to find the sender and tag in that case.

Example of Querying MPI_Status

```
const int MAX_NUMBERS = 100;
int numbers[MAX_NUMBERS];
int number_amount;
if (world_rank == 0) {
    // Pick a random amount of integers to send to process one
    srand(time(NULL));
    number_amount = (rand() / (float)RAND_MAX) * MAX_NUMBERS;

    // Send the amount of integers to process one
    MPI_Send(numbers, number_amount, MPI_INT, 1, 0, MPI_COMM_WORLD);
    printf("0 sent %d numbers to 1\n", number_amount);
}
...
```

Example of Querying MPI_Status

```
...
} else if (world_rank == 1) {
    MPI_Status status;
    // Receive at most MAX_NUMBERS from process zero
    MPI_Recv(numbers, MAX_NUMBERS, MPI_INT, 0, 0, MPI_COMM_WORLD,
             &status);

    // After receiving the message, check the status to determine
    // how many numbers were actually received
    MPI_Get_count(&status, MPI_INT, &number_amount);

    // Print off the amount of numbers, and also print additional
    // information in the status object
    printf("1 received %d numbers from 0. Message source = %d, "
           "tag = %d\n",
           number_amount, status.MPI_SOURCE, status.MPI_TAG);
}
```

Output Example:

```
0 sent 93 numbers to 1
1 dynamically received 93 numbers from 0
```


MPI_Probe

```
MPI_Probe(  
    int source,  
    int tag,  
    MPI_Comm comm,  
    MPI_Status* status)
```

- MPI_Probe will block for a message with a matching tag and sender. When it receives it, it will fill the status structure with information. Then the user can use MPI_Recv to receive the actual message.
- MPI_Probe forms the basis of many **dynamic MPI applications**. For example, manager/worker programs will often make heavy use of MPI_Probe when exchanging variable-sized worker messages.

MPI_Probe Example

```
...
} else if (world_rank == 1) {
    MPI_Status status;
    // Probe for an incoming message from process zero
    MPI_Probe(0, 0, MPI_COMM_WORLD, &status);

    // When probe returns, the status object has the size and other
    // attributes of the incoming message. Get the message size
    MPI_Get_count(&status, MPI_INT, &number_amount);

    // Allocate a buffer to hold the incoming numbers
    int* number_buf = (int*)malloc(sizeof(int) * number_amount);

    // Now receive the message with the allocated buffer
    MPI_Recv(number_buf, number_amount, MPI_INT, 0, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("1 dynamically received %d numbers from 0.\n",
           number_amount);
    free(number_buf);
}
```

MPI Can Be Simple

Claim: Most MPI applications can be written with only 6 functions (although which 6 may differ)

Using point-to-point:

- MPI_INIT
- MPI_FINALIZE
- MPI_COMM_SIZE
- MPI_COMM_RANK
- MPI_SEND
- MPI_RECEIVE

Using collectives:

- MPI_INIT
- MPI_FINALIZE
- MPI_COMM_SIZE
- MPI_COMM_RANK
- MPI_BCAST
- MPI_REDUCE

You may use more for convenience or performance.

But is that small subset the practical usage?

- 35 What aspects of the MPI standard do you use in your application in its current form?
* (multiple)
- 36 What aspects of the MPI standard appear in performance-critical sections of your current application? * (multiple)
- 37 What aspects of the MPI standard do you anticipate using in the "exascale" version of your application? * (multiple)

Responses	Q35: Current Usage			Q37: Exascale Usage			Q36: Performance Critical		
	AD	ST	Overall	AD	ST	Overall	AD	ST	Overall
Point-to-point communications	96%	79%	88%	89%	71%	80%	93%	75%	84%
MPI derived datatypes	25%	21%	23%	21%	21%	21%	14%	7%	11%
Collective communications	86%	75%	80%	96%	68%	82%	64%	64%	64%
Neighbor collective communications	14%	14%	14%	32%	25%	29%	7%	11%	9%
Communicators and group management	68%	54%	61%	61%	50%	55%	29%	7%	18%
Process topologies	14%	7%	11%	32%	11%	21%	4%	4%	4%
RMA (one-sided communications)	36%	7%	21%	50%	36%	43%	21%	7%	14%
RMA shared windows	18%	7%	12%	21%	18%	20%	7%	7%	7%
MPI I/O (called directly)	25%	18%	21%	21%	18%	20%	4%	7%	5%
MPI I/O (called through a third-party library)	32%	21%	27%	36%	25%	30%	7%	11%	9%
MPI profiling interface	11%	0%	14%	11%	21%	16%	0%	4%	2%

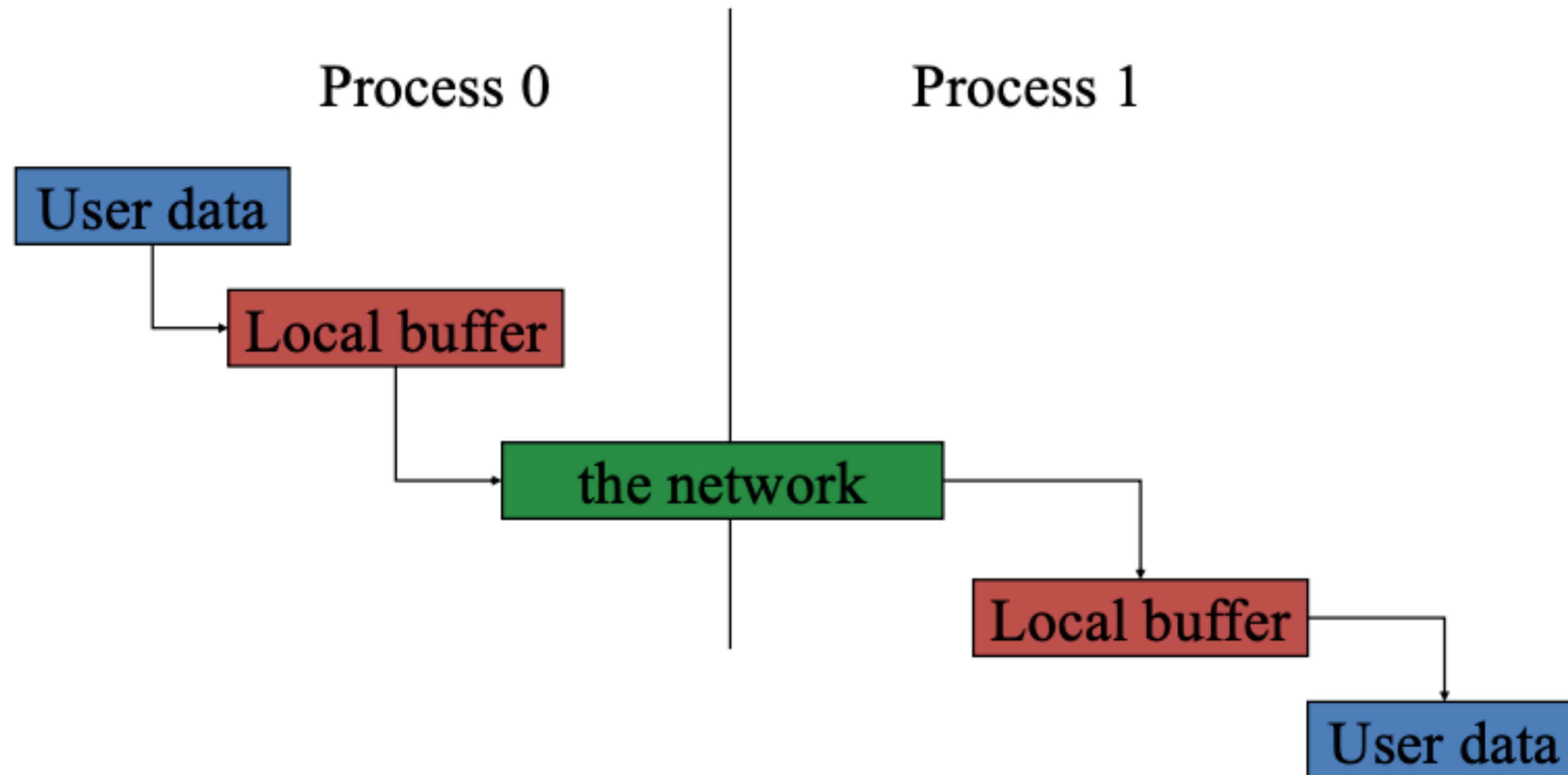
But is that small subset the practical usage?

No.	Question and Responses	AD	ST	Overall
38	What is the dominant communication in your application? Check all that apply, recognizing that many applications have different communication patterns in different phases. <i>(multiple)</i>			
	Each process talks to (almost) every other process	25%	21%	23%
	Processes communicate in fixed "neighborhoods" of limited size	46%	46%	46%
	Processes communicate in "neighborhoods" of limited size that may change in different phases of the application or evolve over the course of a run	42%	36%	39%
	Communication is largely irregular	18%	21%	20%
39	Can your application take advantage of non-blocking point-to-point operations... <i>(multiple)</i>			
	To overlap communication with computation?	89%	71%	80%
	To allow asynchronous progress?	64%	64%	64%
	To allow event-based programming?	43%	29%	36%
40	Can your application take advantage of non-blocking collective operations... <i>(multiple)</i>			
	To overlap communication with computation?	71%	46%	59%
	To allow asynchronous progress?	46%	29%	38%
	To allow event-based programming?	25%	14%	20%

FIRST LECTURE ENDED HERE

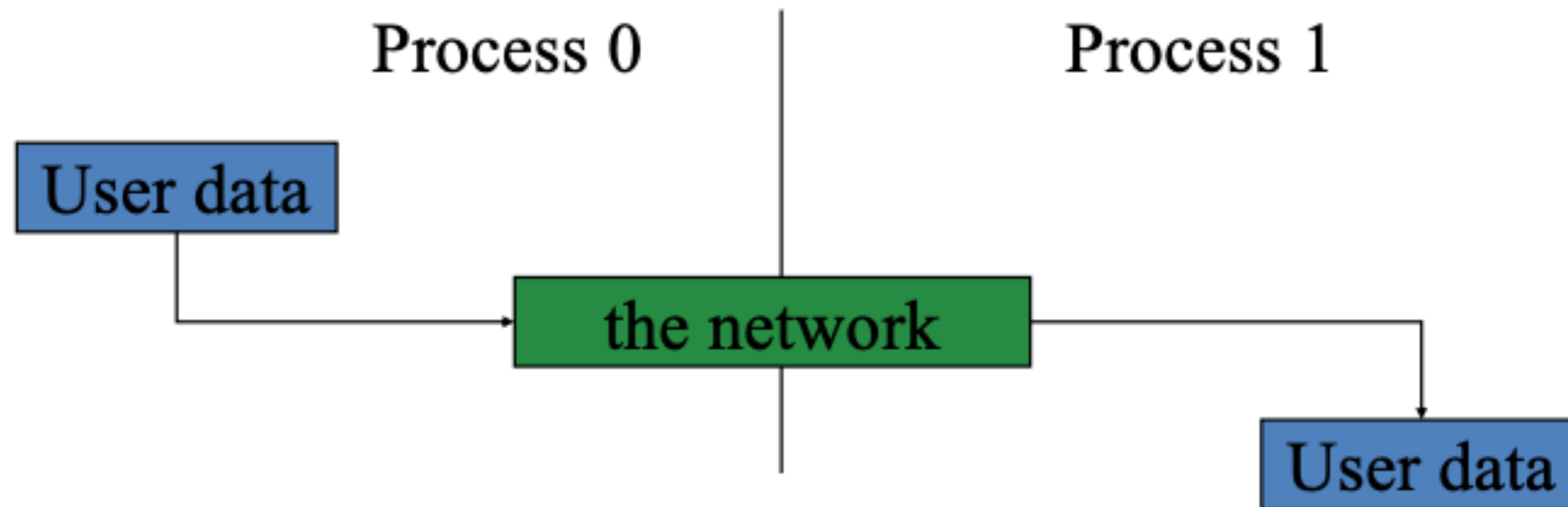
Buffers

When you send data, where does it go? One possibility is:



Avoiding Buffering

Avoiding copies uses less memory (and may or may not save time).



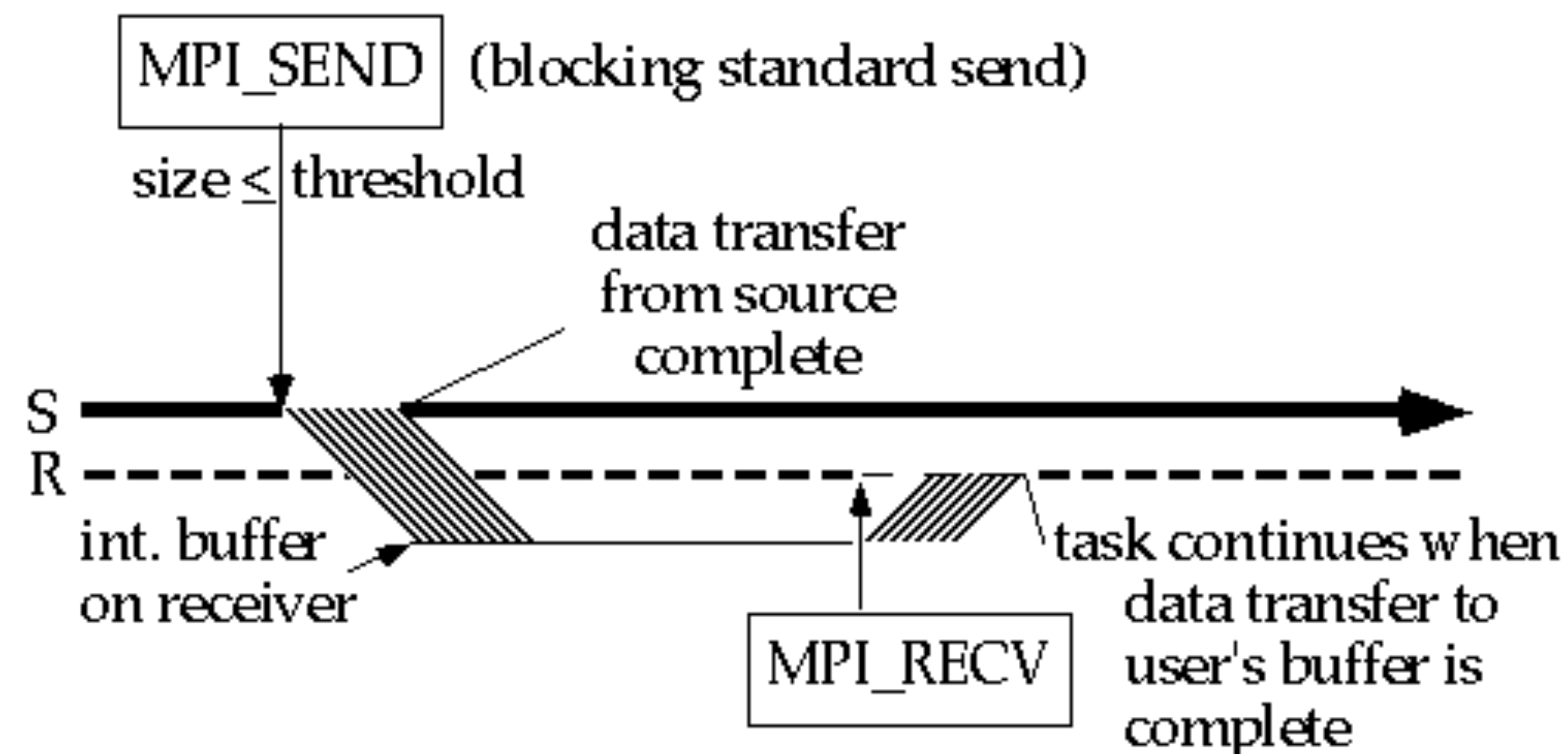
This requires that `MPI_Send` wait on delivery, or that `MPI_Send` return before transfer is complete, and we wait later.

Blocking and Non-Blocking Communication

So far we have been using blocking communication:

- MPI_Recv does not complete until the buffer is full (available for use).
- MPI_Send does not complete until the buffer is empty (available for use).

Completion depends on size of message and amount of system buffering.

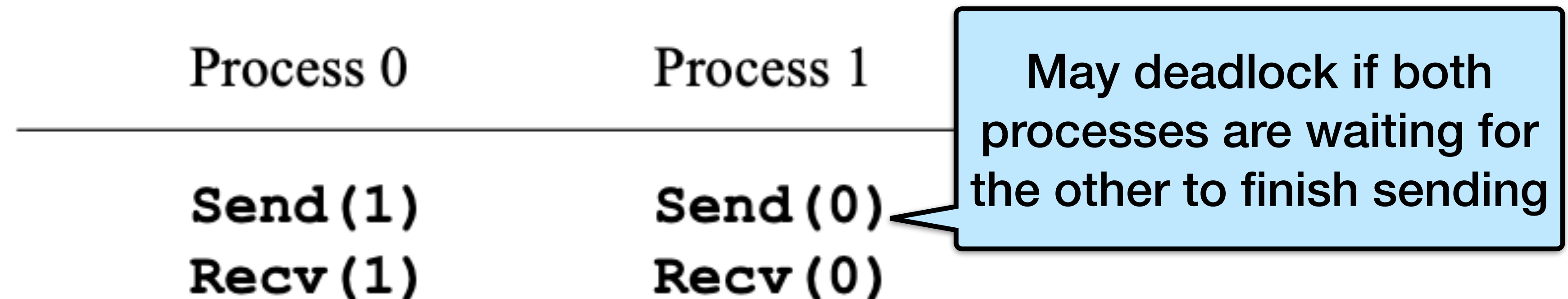


<https://www.codingame.com/playgrounds/47058/have-fun-with-mpi-in-c/blocking-communication>

Deadlocks

Send a large message from process 0 to process 1

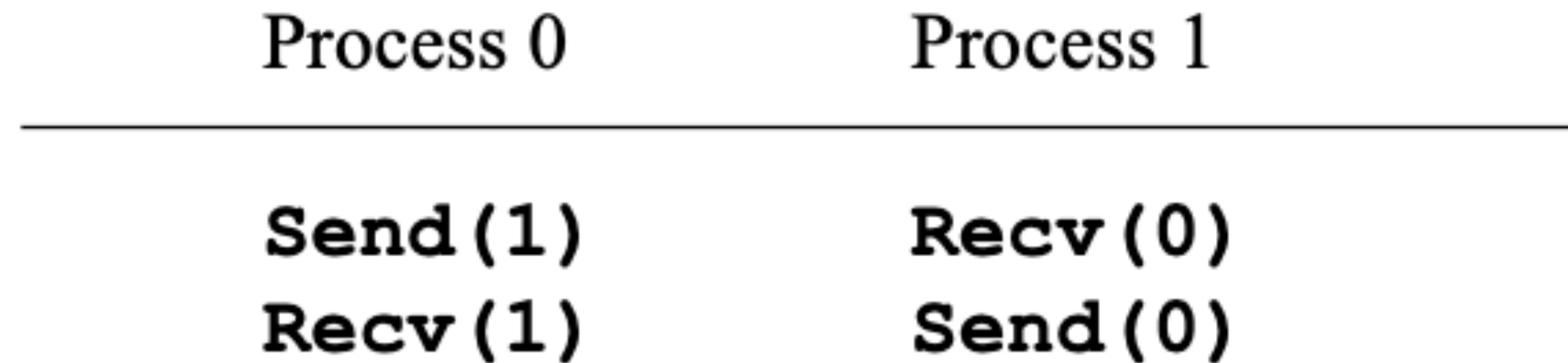
- If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)



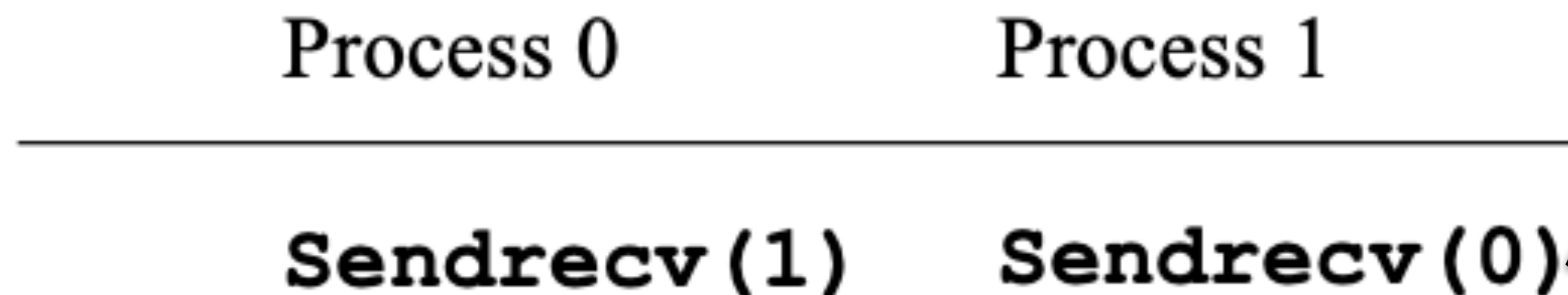
This is called “unsafe” because it **depends on the availability of system buffers** in which to store the data sent until it can be received.

Some Solutions to Deadlock in MPI

Order the operations more carefully:



Supply receive buffer at same time as send:



Can think of it as both
send and receive
executed concurrently

MPI_Sendrecv

```
int MPI_Sendrecv(const void* buffer_send,
                 int count_send,
                 MPI_Datatype datatype_send,
                 int recipient,
                 int tag_send,
                 void* buffer_recv,
                 int count_recv,
                 MPI_Datatype datatype_recv,
                 int sender,
                 int tag_recv,
                 MPI_Comm communicator,
                 MPI_Status* status);
```

- MPI_Sendrecv is a combination of an MPI_send and MPI_Recv.
- It can be seen as both subroutines executing concurrently.
- The buffers for send and receive must be different.

MPI_Sendrecv Example

```
// NOT SHOWN - INIT AND MAKE SURE YOU HAVE TWO PROCESSES

// Prepare parameters
int my_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
int buffer_send = (my_rank == 0) ? 12345 : 67890;
int buffer_recv;
int tag_send = 0;
int tag_recv = tag_send;
int peer = (my_rank == 0) ? 1 : 0;

// Issue the send + receive at the same time
printf("MPI process %d sends value %d to MPI process %d.\n", my_rank, buffer_send, peer);
MPI_Sendrecv(&buffer_send, 1, MPI_INT, peer, tag_send,
             &buffer_recv, 1, MPI_INT, peer, tag_recv, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
printf("MPI process %d received value %d from MPI process %d.\n", my_rank, buffer_recv,
peer);

// NOT SHOWN - FINALIZE
```

MPI_Sendrecv Example

```
// NOT SHOWN - INIT AND MAKE SURE YOU HAVE TWO PROCESSES

// Prepare parameters
int my_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
int buffer_send = (my_rank == 0) ? 12345 : 67890;
int buffer_recv;
int tag_send = 0;
int tag_recv = tag_send;
int peer = (my_rank == 0) ? 1 : 0;

// Issue the send + receive at the same time
printf("MPI process %d sends value %d to MPI process %d.\n", my_rank, buffer_send, peer);
MPI_Sendrecv(&buffer_send, 1, MPI_INT, peer, tag_send,
             &buffer_recv, 1, MPI_INT, peer, tag_recv, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
printf("MPI process %d received value %d from MPI process %d.\n", my_rank, buffer_recv,
peer);

// NOT SHOWN - FINALIZE
```

P0 sends 12345, receives 67890
P1 sends 67890, receives 12345

More Solutions to Deadlock in MPI

Supply own space as buffer for send

Process 0

Process 1

Bsend(1)

Bsend(0)

Recv(1)

Recv(0)

Use non-blocking operations:

Process 0

Process 1

Isend(1)

Isend(0)

Irecv(1)

Irecv(0)

Waitall

Waitall

MPI_Bsend

```
int MPI_Bsend(const void* buffer,  
             int count,  
             MPI_Datatype datatype,  
             int recipient,  
             int tag,  
             MPI_Comm communicator);
```

- MPI_Bsend is the asynchronous **blocking** send (the capital B stands for buffered).
- It will block until a copy of the buffer passed is made.
- MPI will send the buffer at a later point (unknown to the user).
- It guarantees that the buffer can be safely reused once MPI_Bsend returns.

Two processes

MPI_Bsend Example - Sender

```
// Get my rank and do the corresponding job
enum role_ranks { SENDER, RECEIVER };
int my_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
switch(my_rank)
{
    case SENDER:
    {
        // Declare the buffer and attach it
        int buffer_attached_size = MPI_BSEND_OVERHEAD + sizeof(int);
        char* buffer_attached = (char*)malloc(buffer_attached_size);
        MPI_Buffer_attach(buffer_attached, buffer_attached_size);

        // Issue the MPI_Bsend
        int buffer_sent = 12345;
        printf("[MPI process %d] I send value %d.\n", my_rank, buffer_sent);
        MPI_Bsend(&buffer_sent, 1, MPI_INT, RECEIVER, 0, MPI_COMM_WORLD);

        // Detach the buffer. It blocks until all messages stored are sent.
        MPI_Buffer_detach(&buffer_attached, &buffer_attached_size);
        free(buffer_attached);
        break;
    }
}
```

Memory overhead
generated by MPI_Bsend

Two processes

MPI_Bsend Example - Sender

```
// Get my rank and do the corresponding job
enum role_ranks { SENDER, RECEIVER };
int my_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
switch(my_rank)
{
    case SENDER:
    {
        // Declare the buffer and attach it
        int buffer_attached_size = MPI_BSEND_OVERHEAD + sizeof(int);
        char* buffer_attached = (char*)malloc(buffer_attached_size);
        MPI_Buffer_attach(buffer_attached, buffer_attached_size);

        // Issue the MPI_Bsend
        int buffer_sent = 12345;
        printf("[MPI process %d] I send value %d.\n", my_rank, buffer_sent);
        MPI_Bsend(&buffer_sent, 1, MPI_INT, RECEIVER, 0, MPI_COMM_WORLD);

        // Detach the buffer. It blocks until all messages stored are sent.
        MPI_Buffer_detach(&buffer_attached, &buffer_attached_size);
        free(buffer_attached);
        break;
    }
}
```

Memory overhead
generated by MPI_Bsend

Value is 12345

Two processes

MPI_Bsend Example - Receiver

```
case RECEIVER:
{
    // Receive the message and print it.
    int received;
    MPI_Recv(&received, 1, MPI_INT, SENDER, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("[MPI process %d] I received value: %d.\n", my_rank, received);
    break;
}
}
```

Should be 12345

Non-Blocking Send/Receive

Non-blocking operations **return (immediately)** “request handles” that can be tested and waited on:

- `MPI_Request request;`
- `MPI_Status status;`
- `MPI_Isend(...);`
- `MPI_Irecv(...);`
- `MPI_Wait(&request, &status);` (each request must be Waited on)

One can also test without waiting:

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

Accessing the data buffer without waiting is undefined

MPI_Isend

```
int MPI_Isend(const void* buffer,  
             int count,  
             MPI_Datatype datatype,  
             int recipient,  
             int tag,  
             MPI_Comm communicator,  
             MPI_Request* request);
```

Handle on non-blocking operation (used later to check for completion)

- MPI_Isend is the standard **non-blocking send** (I stands for immediate return).
- The user must not attempt to reuse the buffer after MPI_Isend returns without explicitly **checking for completion**.

MPI_Wait and MPI_Test

```
int MPI_Wait(MPI_Request* request,  
             MPI_Status* status);
```

MPI_Wait **waits** for a non-blocking operation to complete and will block until the underlying operation is done.

```
int MPI_Test(MPI_Request* request,  
             int* flag,  
             MPI_Status* status);
```

MPI_Test checks if a non-blocking operation is complete at a given time. **It will not wait** for the underlying non-blocking operation to complete.

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(count, array_of_requests, array_of_indices,  
array_of_statuses)
```

There are corresponding versions of test for each of these.

Two processes

MPI_Isend Example - Sender

```
switch(my_rank)
{
    case SENDER:
    {
        int buffer_sent = 12345;
        MPI_Request request;
        printf("MPI process %d sends value %d.\n", my_rank, buffer_sent);
        MPI_Isend(&buffer_sent, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);

        // Do other things while the MPI_Isend completes
        // <...>

        // Let's wait for the MPI_Isend to complete before progressing
        further.
        MPI_Wait(&request, MPI_STATUS_IGNORE);
        break;
    }
    // RECEIVER CASE ON NEXT SLIDE
}
```

Two processes

MPI_Isend Example - Sender

MPI process 0 sends value 12345

```
switch(my_rank)
{
    case SENDER:
    {
        int buffer_sent = 12345;
        MPI_Request request;
        printf("MPI process %d sends value %d.\n", my_rank, buffer_sent);
        MPI_Isend(&buffer_sent, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);

        // Do other things while the MPI_Isend completes
        // <...>

        // Let's wait for the MPI_Isend to complete before progressing
        further.
        MPI_Wait(&request, MPI_STATUS_IGNORE);
        break;
    }
    // RECEIVER CASE ON NEXT SLIDE
}
```

Two processes

MPI_Isend Example - Sender

```
switch(my_rank)
{
  case SENDER:
  {
    int buffer_sent = 12345;
    MPI_Request request;
    printf("MPI process %d sends value %d.\n", my_rank, buffer_sent);
    MPI_Isend(&buffer_sent, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &request);

    // Do other things while the MPI_Isend completes
    // <...>

    // Let's wait for the MPI_Isend to complete before progressing
    further.
    MPI_Wait(&request, MPI_STATUS_IGNORE);
    break;
  }
  // RECEIVER CASE ON NEXT SLIDE
}
```

MPI process 0 sends value 12345

Only continue after the send is done

Two processes

MPI_Isend Example - Receiver

```
switch(my_rank)
{
    // CASE SENDER FROM BEFORE NOT SHOWN
    case RECEIVER:
    {
        int received;
        MPI_Recv(&received, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("MPI process %d received value: %d.\n", my_rank, received);
        break;
    }
}
```

Does not need to be a special recv

MPI process 1 received value 12345

MPI_Irecv

```
int MPI_Irecv(void* buffer,  
             int count,  
             MPI_Datatype datatype,  
             int sender,  
             int tag,  
             MPI_Comm communicator,  
             MPI_Request* request);
```

Handle on non-blocking operation (used later to check for completion)

- MPI_Irecv is the standard **non-blocking receive** (I stands for immediate return).
- To know whether the message has been received, you must use MPI_Wait or MPI_Test on the MPI_Request.

MPI_Irecv Example

```
switch(my_rank)
{
    // NOT SHOWN - The primary MPI process sends the value 12345 synchronously.
    case RECEIVER:
    {
        // The secondary MPI process receives the message.
        int received;
        MPI_Request request;
        printf("[Process %d] I issue the MPI_Irecv to receive the message as a
background task.\n", my_rank);
        MPI_Irecv(&received, 1, MPI_INT, SENDER, 0, MPI_COMM_WORLD, &request);

        // Do other things while the MPI_Irecv completes.
        printf("[Process %d] The MPI_Irecv is issued, I now moved on to print this
message.\n", my_rank);

        // Wait for the MPI_Recv to complete.
        MPI_Wait(&request, MPI_STATUS_IGNORE);
        printf("[Process %d] The MPI_Irecv completed, therefore so does the underlying
MPI_Recv. I received the value %d.\n", my_rank, received);
        break;
    }
}
```

MPI_Irecv Example

```
switch(my_rank)
{
    // NOT SHOWN - The primary MPI process sends the value 12345 synchronously.
    case RECEIVER:
    {
        // The secondary MPI process receives the message.
        int received;
        MPI_Request request;
        printf("[Process %d] I issue the MPI_Irecv to receive the message as a
background task.\n", my_rank);
        MPI_Irecv(&received, 1, MPI_INT, SENDER, 0, MPI_COMM_WORLD, &request);

        // Do other things while the MPI_Irecv completes.
        printf("[Process %d] The MPI_Irecv is issued, I now moved on to print this
message.\n", my_rank);

        // Wait for the MPI_Recv to complete.
        MPI_Wait(&request, MPI_STATUS_IGNORE);
        printf("[Process %d] The MPI_Irecv completed, therefore so does the underlying
MPI_Recv. I received the value %d.\n", my_rank, received);
        break;
    }
}
```

Can do local work while waiting

MPI_Irecv Example

```
switch(my_rank)
{
    // NOT SHOWN - The primary MPI process sends the value 12345 synchronously.
    case RECEIVER:
    {
        // The secondary MPI process receives the message.
        int received;
        MPI_Request request;
        printf("[Process %d] I issue the MPI_Irecv to receive the message as a
background task.\n", my_rank);
        MPI_Irecv(&received, 1, MPI_INT, SENDER, 0, MPI_COMM_WORLD, &request);

        // Do other things while the MPI_Irecv completes.
        printf("[Process %d] The MPI_Irecv is issued, I now moved on to print this
message.\n", my_rank);

        // Wait for the MPI_Recv to complete.
        MPI_Wait(&request, MPI_STATUS_IGNORE);
        printf("[Process %d] The MPI_Irecv completed, therefore so does the underlying
MPI_Recv. I received the value %d.\n", my_rank, received);
        break;
    }
}
```

Can do local work while waiting

value 12345

Summary

- There are many different network topologies, each with their own strengths and weaknesses.
- The main performance metrics of a network are latency and bandwidth.
- MPI defines a standard for programming distributed-memory machines.
- The core functionality for message passing is send and receive (and variants)
- Next lecture: Advanced MPI via collectives

MPI References

- MPI Standard: <https://www.mpi-forum.org/>
- Other information: <https://www.mcs.anl.gov/research/projects/mpi/index.htm>
- LLNL tutorial: <https://hpc-tutorials.llnl.gov/mpi/>
- <https://mpitutorial.com/>

Backup Slides & More Examples

More MPI Send/Receive Examples

```
int ping_pong_count = 0;
int partner_rank = (world_rank + 1) % 2;
while (ping_pong_count < PING_PONG_LIMIT) {
    if (world_rank == ping_pong_count % 2) {
        // Increment the ping pong count before you send it
        ping_pong_count++;
        MPI_Send(&ping_pong_count, 1, MPI_INT, partner_rank, 0,
                MPI_COMM_WORLD);
        printf("%d sent and incremented ping_pong_count "
               "%d to %d\n", world_rank, ping_pong_count,
               partner_rank);
    } else {
        MPI_Recv(&ping_pong_count, 1, MPI_INT, partner_rank, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("%d received ping_pong_count %d from %d\n",
               world_rank, ping_pong_count, partner_rank);
    }
}
```

MPI Ping Pong Output

```
0 sent and incremented ping_pong_count 1 to 1
0 received ping_pong_count 2 from 1
0 sent and incremented ping_pong_count 3 to 1
0 received ping_pong_count 4 from 1
0 sent and incremented ping_pong_count 5 to 1
0 received ping_pong_count 6 from 1
0 sent and incremented ping_pong_count 7 to 1
0 received ping_pong_count 8 from 1
0 sent and incremented ping_pong_count 9 to 1
0 received ping_pong_count 10 from 1
1 received ping_pong_count 1 from 0
1 sent and incremented ping_pong_count 2 to 0
1 received ping_pong_count 3 from 0
1 sent and incremented ping_pong_count 4 to 0
1 received ping_pong_count 5 from 0
1 sent and incremented ping_pong_count 6 to 0
1 received ping_pong_count 7 from 0
1 sent and incremented ping_pong_count 8 to 0
1 received ping_pong_count 9 from 0
1 sent and incremented ping_pong_count 10 to 0
```

MPI Ring Send/Receive Example

```
int token;
if (world_rank != 0) {
    MPI_Recv(&token, 1, MPI_INT, world_rank - 1, 0,
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process %d received token %d from process %d\n",
           world_rank, token, world_rank - 1);
} else {
    // Set the token's value if you are process 0
    token = -1;
}
MPI_Send(&token, 1, MPI_INT, (world_rank + 1) % world_size,
        0, MPI_COMM_WORLD);

// Now process 0 can receive from the last process.
if (world_rank == 0) {
    MPI_Recv(&token, 1, MPI_INT, world_size - 1, 0,
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process %d received token %d from process %d\n",
           world_rank, token, world_size - 1);
}
```


MPI Ring Send/Receive Output

```
Process 1 received token -1 from process 0  
Process 2 received token -1 from process 1  
Process 3 received token -1 from process 2  
Process 4 received token -1 from process 3  
Process 0 received token -1 from process 4
```