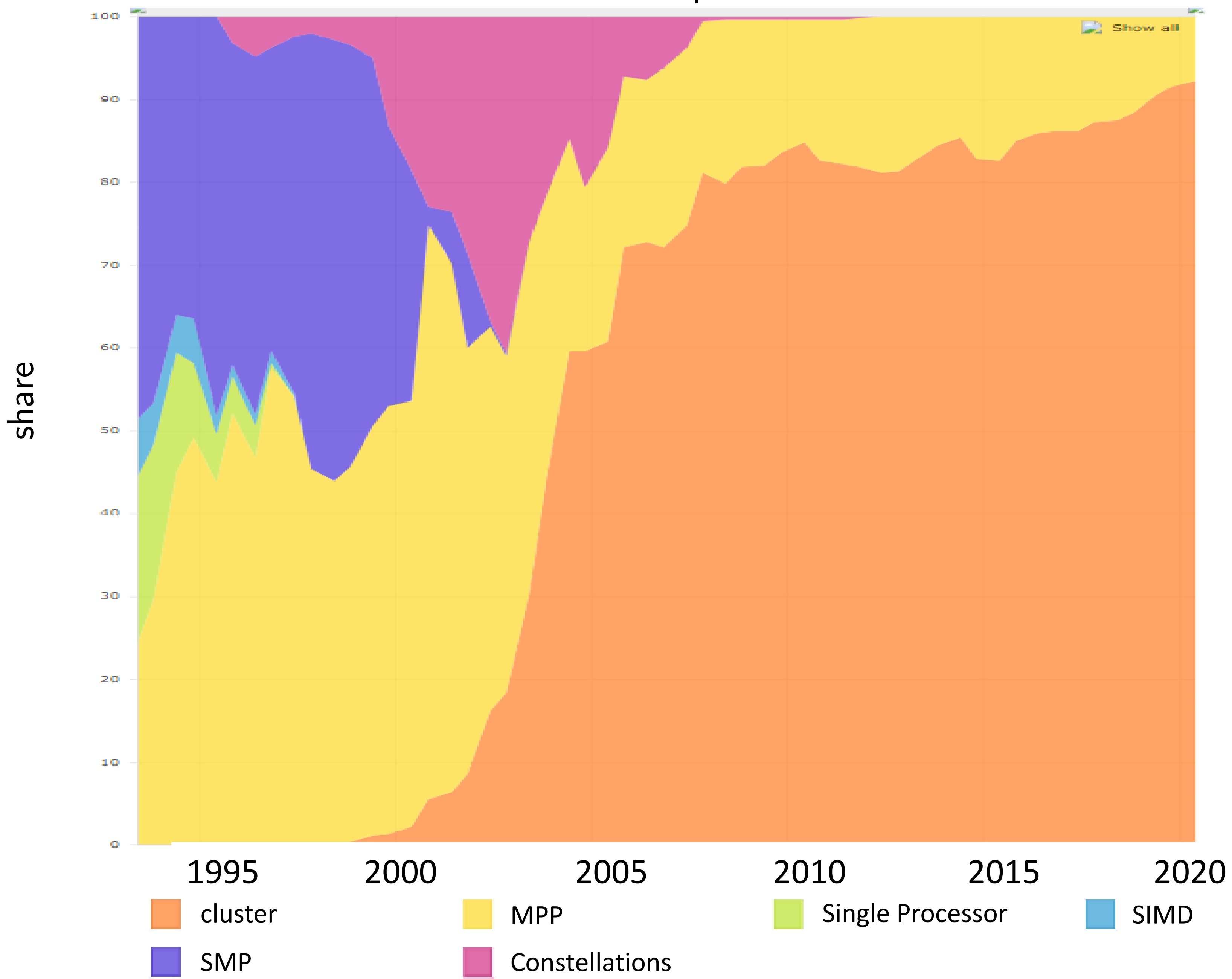
Lecture 5:
Intro to Distributed Memory Programming and MPI

Outline

- Distributed Memory Architectures
  - Properties of communication networks
  - Topologies

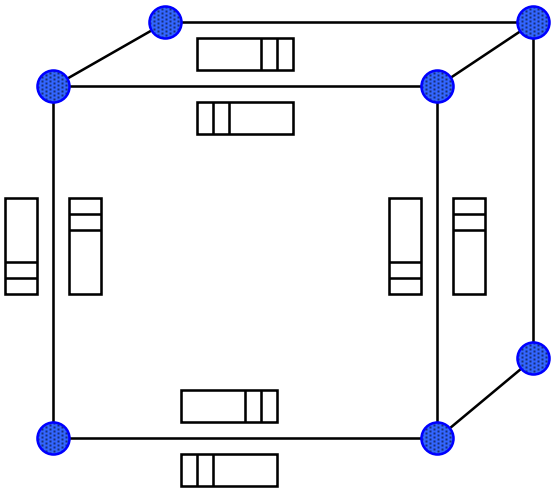
- MPI Intro

Architectures in Top 500 Over Time



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.
- Properties:
  - Latency: how long to get between nodes in the network.
    - Street: time for one car = dist (miles) / speed (miles/hr)
  - Bandwidth: how much data can be moved per unit time.
    - Street: cars/hour = density (cars/mile) * speed (miles/hr) * #lanes
    - Network bandwidth is limited by the bit rate per wire and #wires

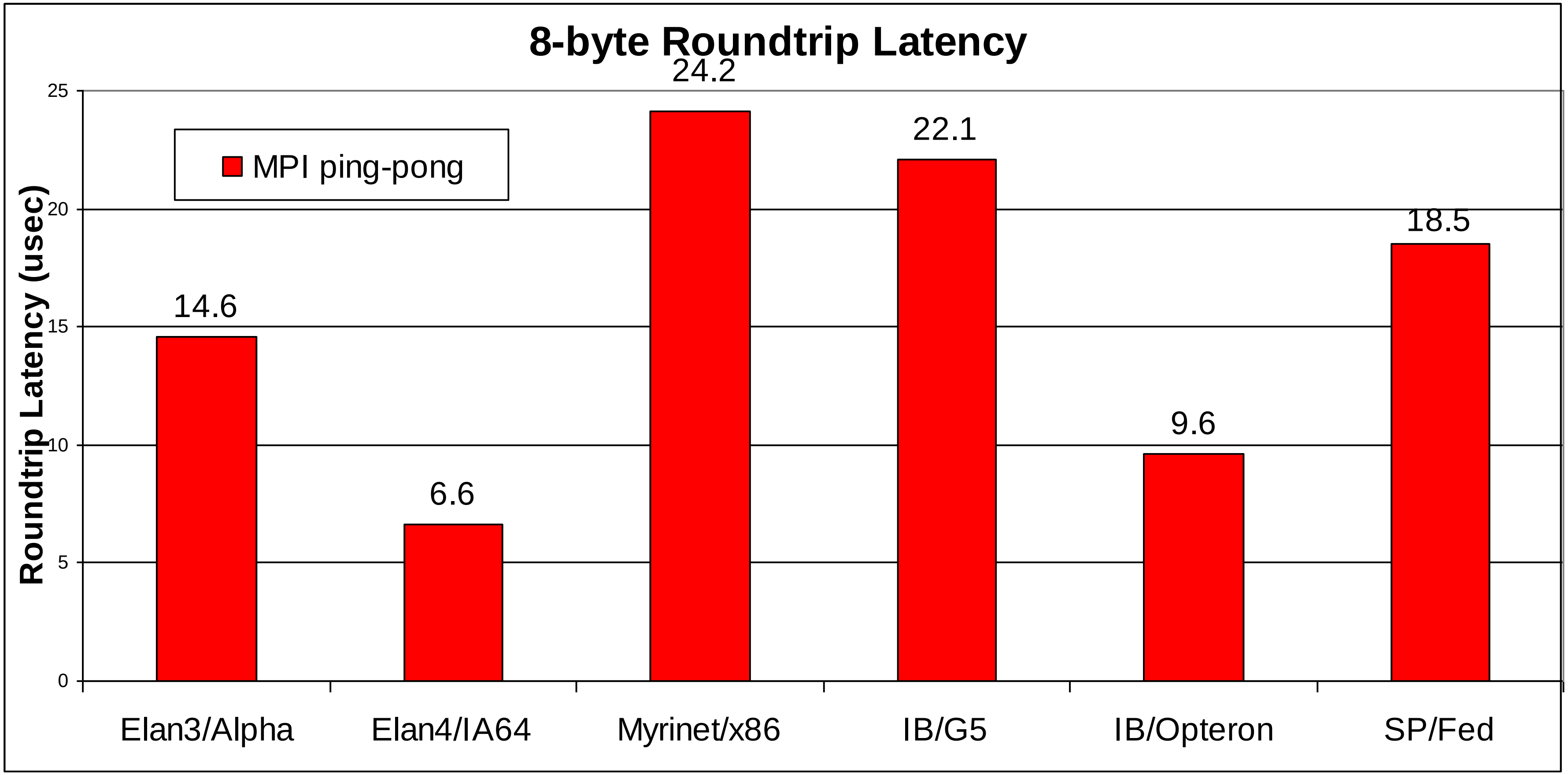
Design Characteristics of a Network

- Topology (how things are connected)
  - Crossbar; ring; 2-D, 3-D, higher-D mesh or torus; hypercube; tree; butterfly; perfect shuffle, ...
- Routing algorithm:
  - Example in 2D torus: all east-west then all north-south (avoids deadlock).
- 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 of a Network: Latency

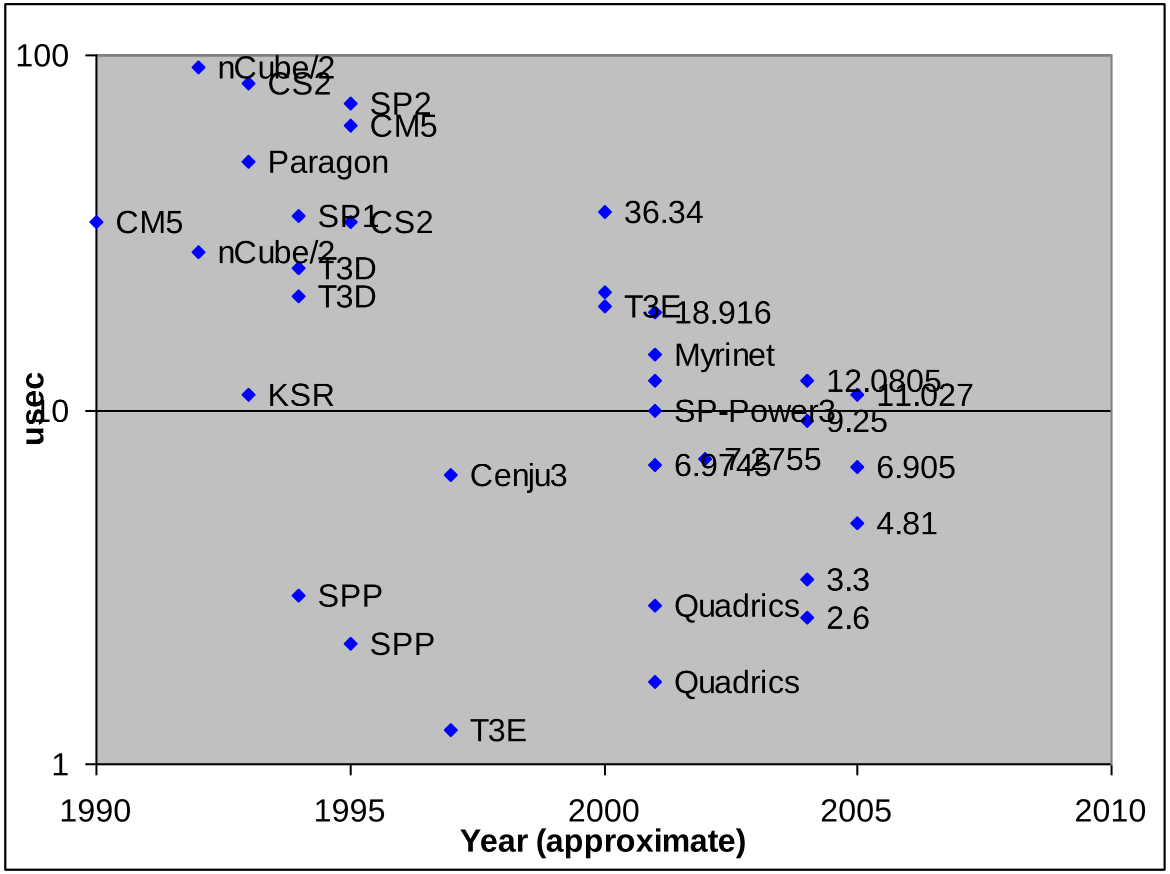
- Diameter: the maximum (over all pairs of nodes) of the shortest path between a given pair of nodes.
- Latency: delay between send and receive times
  - Latency tends to vary widely across architectures
  - Vendors often report hardware latencies (wire time)
  - Application programmers care about software latencies (user program to user program)
- Observations:
  - Latencies differ by 1-2 orders across network designs
  - 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

Latency on Some Machines/Networks



- Latencies shown are from a ping-pong test using MPI
- These are roundtrip numbers: many people use ½ of roundtrip time to approximate 1-way latency (which can't easily be measured)

End to End Latency (1/2 roundtrip) Over Time

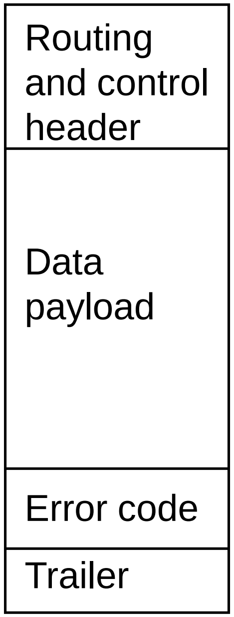


- Latency has not improved significantly, unlike Moore's Law
    - T3E (shmem) was lowest point – in 1997

Data from Kathy Yelick, UCB and NERSC

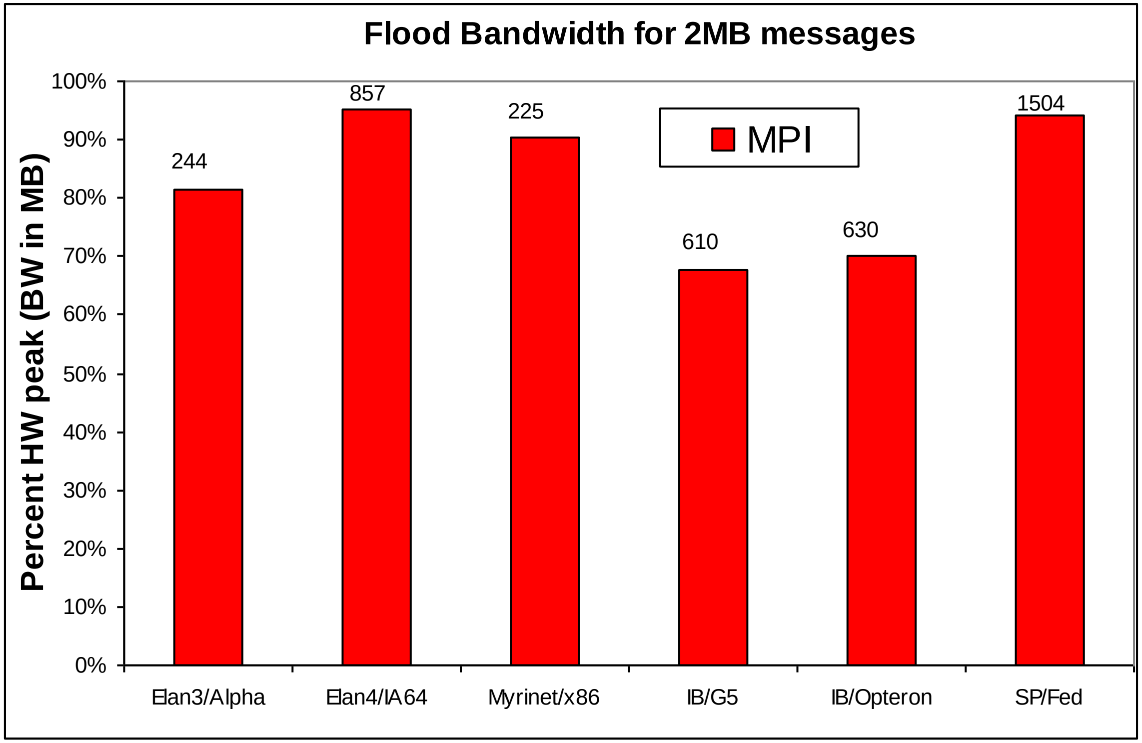
Performance Properties of a Network: Bandwidth

- The bandwidth of a link = # wires / time-per-bit
- Bandwidth typically in Gigabytes/sec (GB/s), i.e., 8* 2^{20} bits per second
- Effective bandwidth is usually lower than physical link bandwidth due to packet overhead.



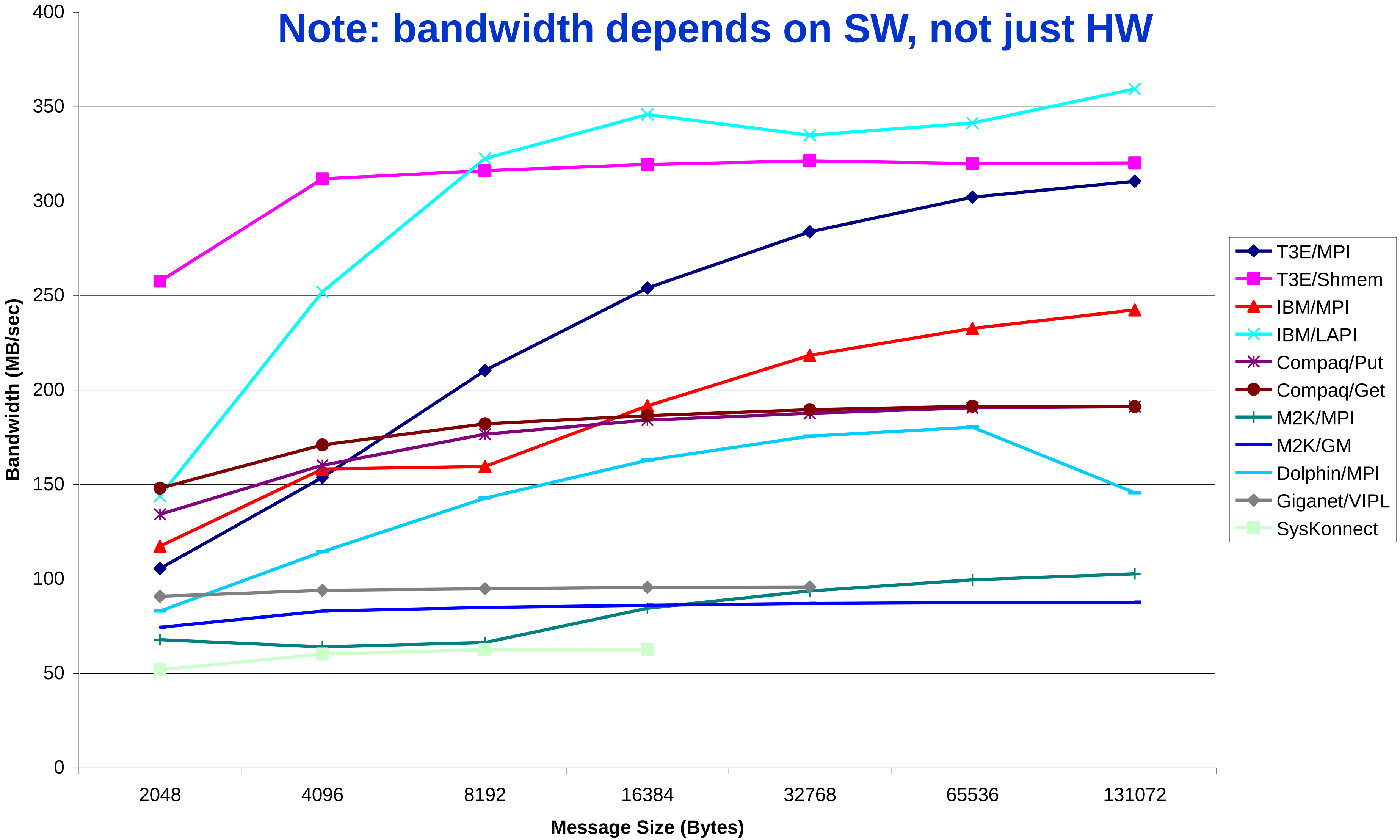
- Bandwidth is important for applications with mostly large messages

Bandwidth on Existing Networks



- Flood bandwidth (throughput of back-to-back 2MB messages)

Bandwidth Chart



Data from Mike Welcome, NERSC

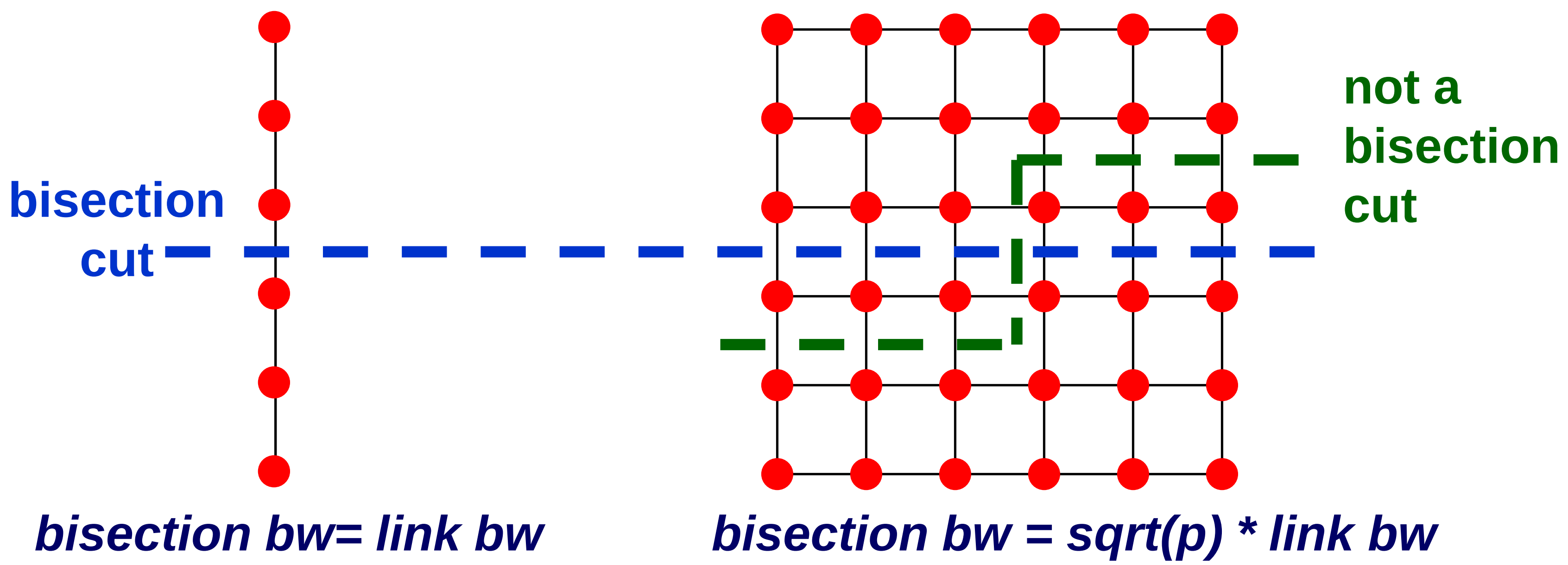
Exascale Systems

	Petascale Systems	Exascale Systems	Factor Improvement
System Peak	10^{16} flops/s	10^{18} flops/s	100
Node Memory Bandwidth	10^{2} GB/s	10^{3} GB/s	10
Interconnect Bandwidth	10^{1} GB/s	10^{2} GB/s	10
Memory Latency	10^{-7} s	$5 \cdot 10^{-8}$ s	2
Interconnect Latency	10^{-6} s	$5 \cdot 10^{-7}$ s	2

- Movement of data (communication) is much more expensive than floating point operations (computation), in terms of both time and energy
- Gaps will only grow larger

Performance Properties of a Network: Bisection Bandwidth

- Bisection bandwidth: bandwidth across smallest cut that divides network into two equal halves
- Bandwidth across "narrowest" part of the network



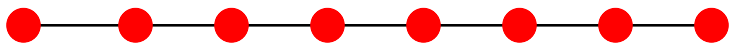
- Bisection bandwidth is important for algorithms in which all processors need to communicate with all others

Network Topology

- In the past, there was considerable research in network topology and in mapping algorithms to topology.
  - Key cost to be minimized: number of "hops" between nodes (e.g. "store and forward")
  - Modern networks hide hop cost, and user-level latency depends more on overheads than toplogy, so topology less of a factor in performance of many algorithms
- Need some background in network topology
  - Algorithms may have a communication topology

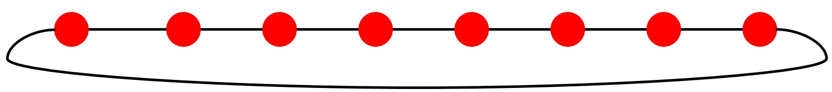
Linear and Ring Topologies

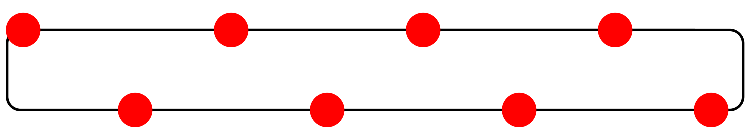
- Linear array



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

- Torus or Ring



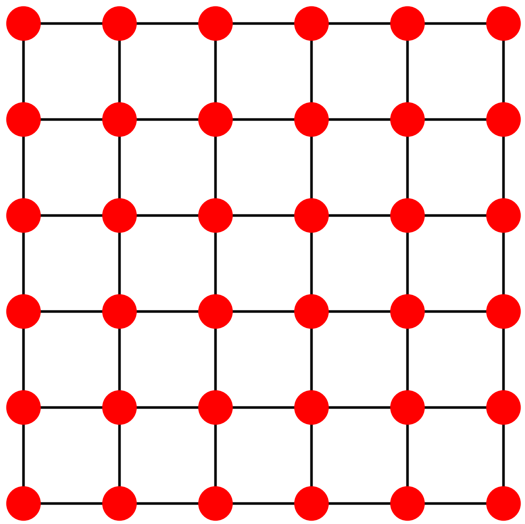


  - Diameter $= n/2$; average distance $\approx n/4$.
  - Bisection bandwidth $= 2$.
  - Natural for algorithms that work with 1D arrays.

Meshes and Tori

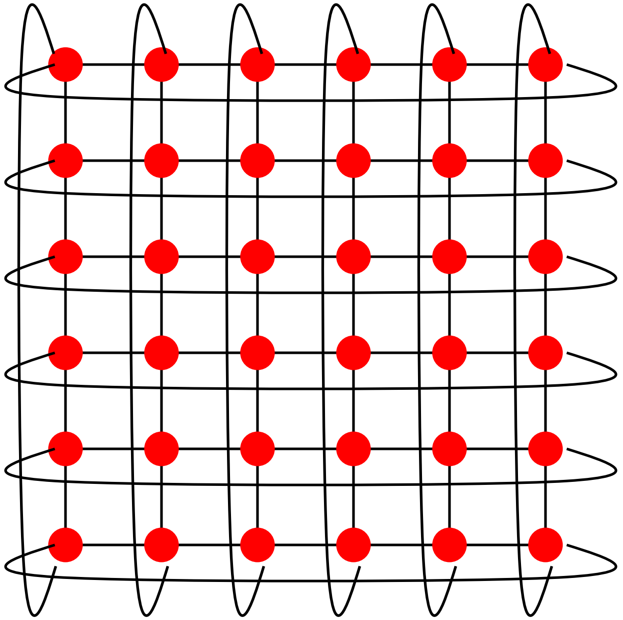
Two dimensional mesh

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



Two dimensional torus

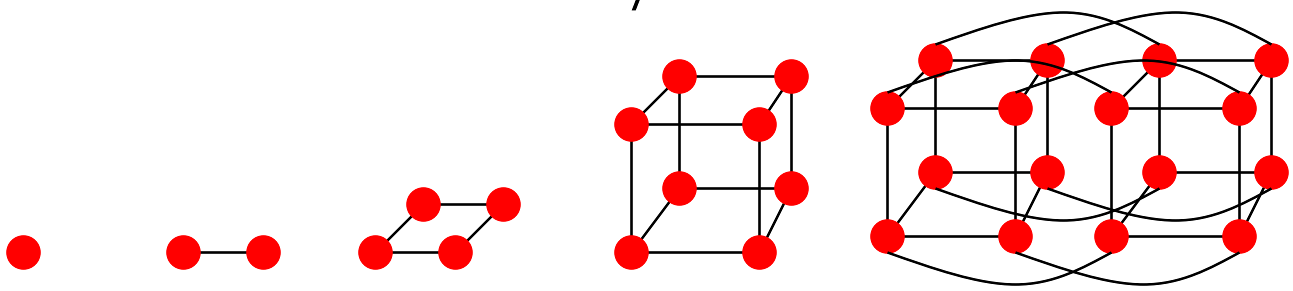
- Diameter $= \sqrt{n}$
- Bisection bandwidth $= 2\sqrt{n}$



- Generalizes to higher dimensions
  - Natural for algorithms that work with 2D and/or 3D arrays (matmul)

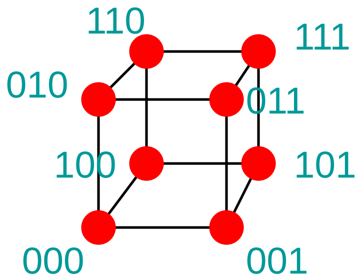
Hypercubes

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



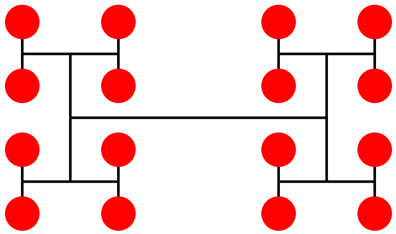
- 0d 1d 2d 3d 4d

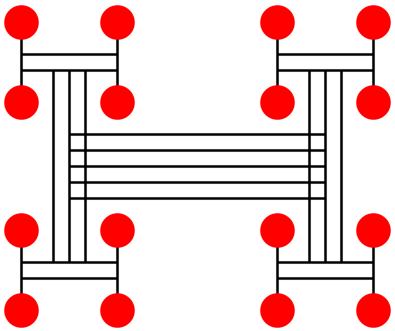
- Popular in early machines (Intel iPSC, NCUBE).
  - Lots of clever algorithms.
- Greycode addressing:
  - Each node connected to d others with 1 bit different.

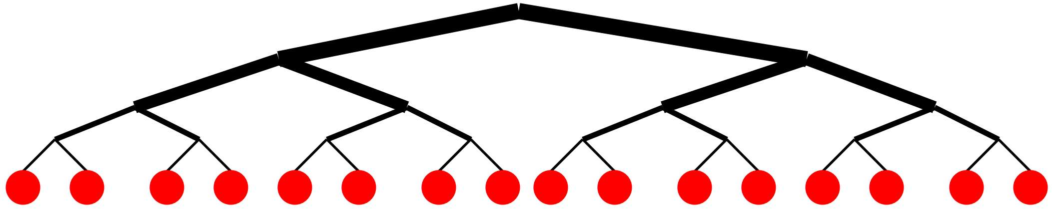


Trees

- 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.



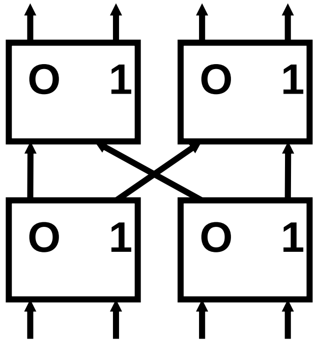




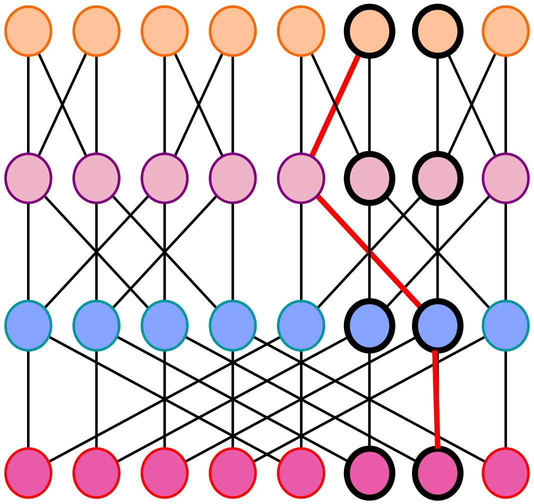
Butterflies

- Diameter $= \log n$
- Bisection bandwidth $= n$
- Cost: lots of wires.
- Used in BBN Butterfly.
- Natural for FFT.

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

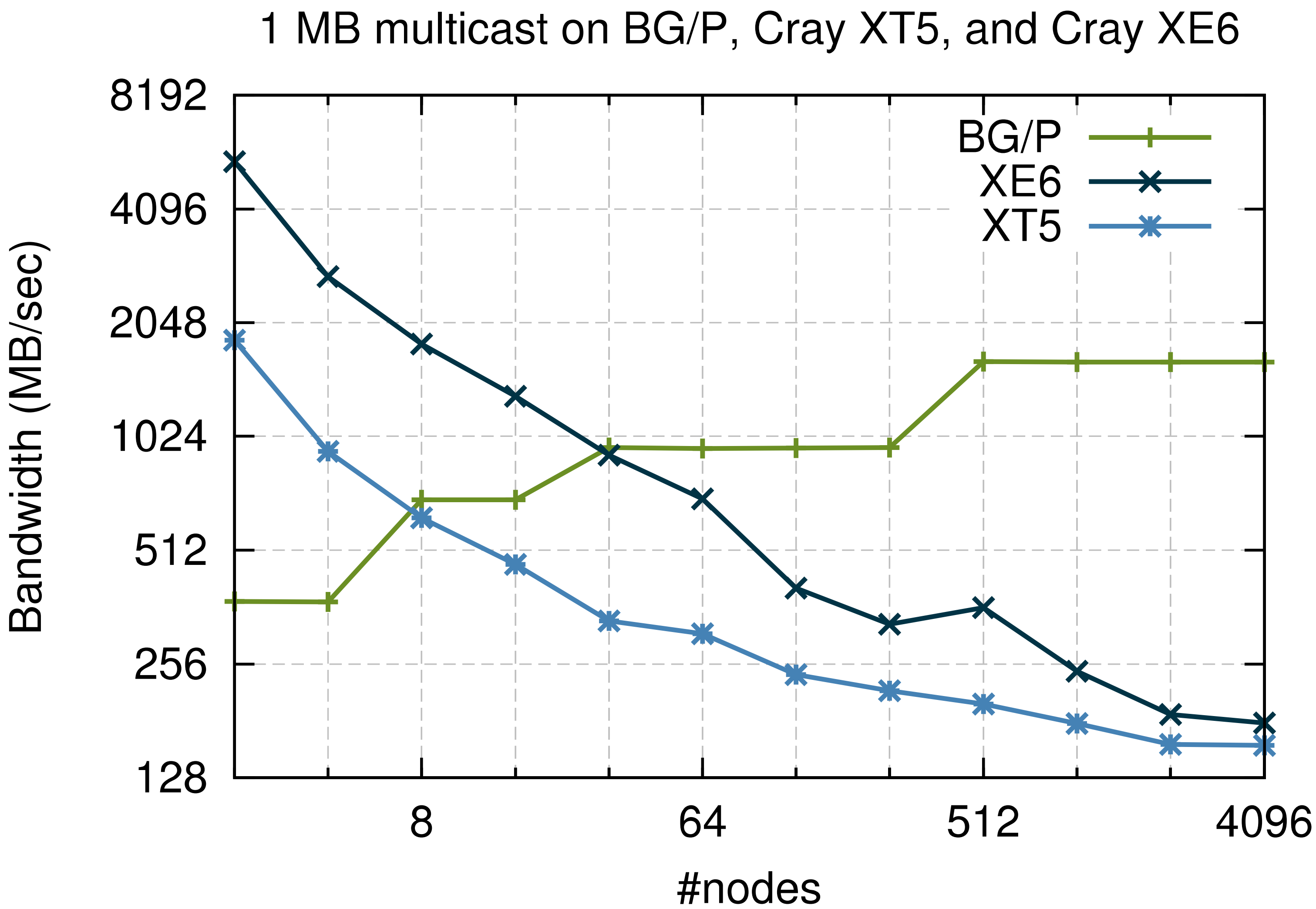


butterfly switch



multistage butterfly network

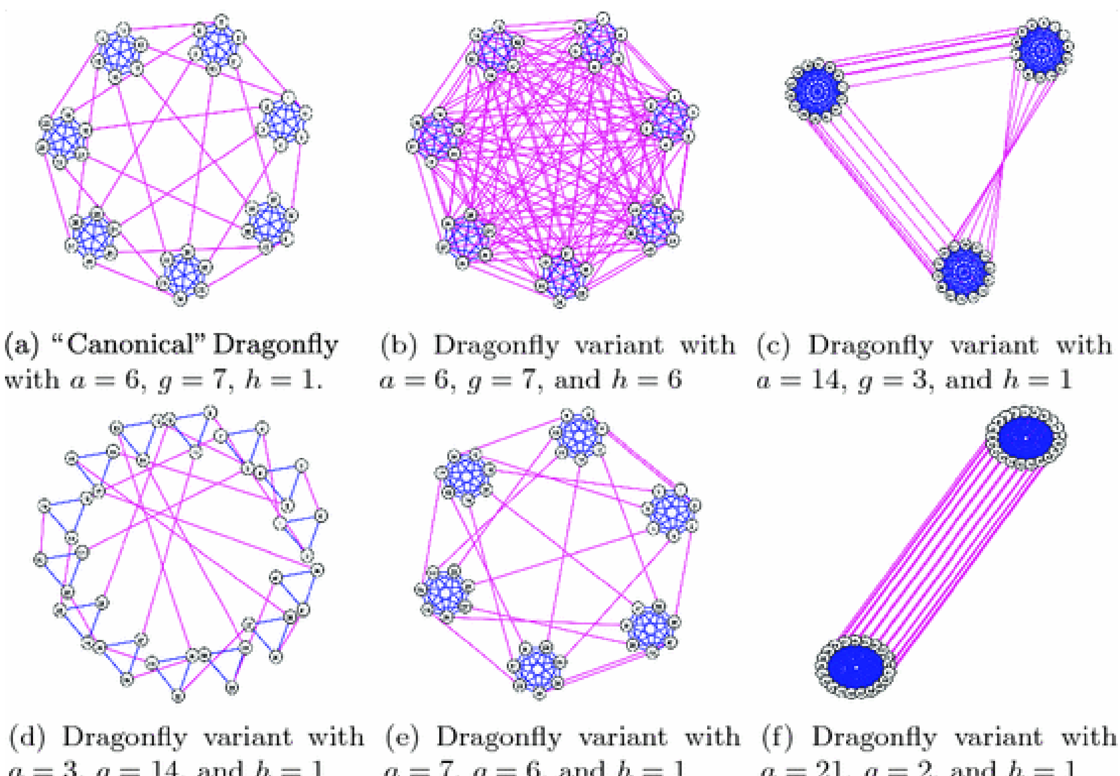
Does Topology Matter?



Dragonfly Topology

- A hierarchical topology with properties:
  - Several "groups" of nodes are connected using all-to-all links
  - Topology inside each group can be any topology

[John Kim et al. "Technology-Driven, Highly-Scalable Dragonfly Topology", 2008]



(a) "Canonical" Dragonfly with $a = 6$, $g = 7$, $h = 1$.
(b) Dragonfly variant with $a = 6$, $g = 7$, and $h = 6$
(c) Dragonfly variant with $a = 14$, $g = 3$, and $h = 1$
(d) Dragonfly variant with $a = 3$, $g = 14$, and $h = 1$
(e) Dragonfly variant with $a = 7$, $g = 6$, and $h = 1$
(f) Dragonfly variant with $a = 21$, $g = 2$, and $h = 1$

[Teh, Wilke, Bergman, Rumley, 2017]

Dragonflies

- Motivation: Exploit gap in cost and performance between optical interconnects (which go between cabinets in a machine room) and electrical networks (inside cabinet)
  - Optical more expensive but higher bandwidth when long
  - Electrical networks cheaper, faster when short
- Combine in hierarchy
  - One-to-many via electrical networks inside cabinet
  - Just a few long optical interconnects between cabinets
- Clever routing algorithm to avoid bottlenecks:
  - Route from source to randomly chosen intermediate cabinet
  - Route from intermediate cabinet to destination
- Outcome: programmer can (usually) ignore topology, get good performance
  - Important in virtualized, dynamic environment
  - Programmer can still create serial bottlenecks
  - Drawback: variable performance

Topologies in Real Machines

older ⟵ ⟶ newer

Cray XT3 and XT4	3D Torus (approx)
Blue Gene/L	3D Torus
SGI Altix	Fat tree
Cray X1	4D Hypercube*
Myricom (Millennium)	Arbitrary
Quadrics (in HP Alpha server clusters)	Fat tree
IBM SP	Fat tree (approx)
SGI Origin	Hypercube
Intel Paragon (old)	2D Mesh
BBN Butterfly (really old)	Butterfly

Many of these are approximations: E.g., the X1 is really a "quad bristled hypercube" and some of the fat trees are not as fat as they should be at the top

Topologies in More Modern Machines

- Frontier (#1): Dragonfly
- Fugaku (#2): 6D Torus
- LUMI (#3): Dragonfly
- Summit (#4): Fat tree
- Sierra (#5): Fat tree
- Sunway TaihuLight (#6): Fat tree
- Perlmutter (#7): Dragonfly
- Selene (#8): Fat tree
- Tianhe-2 (#9): Fat tree

Evolution of Distributed Memory Machines

- Special queue connections replaced by direct memory access (DMA):
  - Network Interface (NI) processor packs or copies messages.
  - CPU initiates transfer, goes on computing.
- Wormhole routing in hardware:
  - NIs do not interrupt CPUs along path.
  - Long message sends are pipelined.
  - NIs don't wait for complete message before forwarding
- Message passing libraries provide store-and-forward abstraction:
  - Can send/receive between any pair of nodes, not just along one wire.
  - Time depends on distance since each NI along path must participate.

Programming Distributed Memory Machines

Message Passing Libraries

- Many "message passing libraries" were once available
  - Chameleon, from ANL.
  - CMMD, from Thinking Machines.
  - Express, commercial.
  - MPL, native library on IBM SP-2.
  - NX, native library on Intel Paragon.
  - Zipcode, from LLL.
  - PVM, Parallel Virtual Machine, public, from ORNL/UTK.
  - Others...
  - MPI, Message Passing Interface, now the industry standard.
- Need standards to write portable code.

Message Passing Libraries

- All communication, synchronization require subroutine calls
    - No shared variables
    - Program runs 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
        - No locks because there are no shared variables to protect
    - Enquiries
        - How many processes? Which one am I? Any messages waiting?

Novel Features of MPI

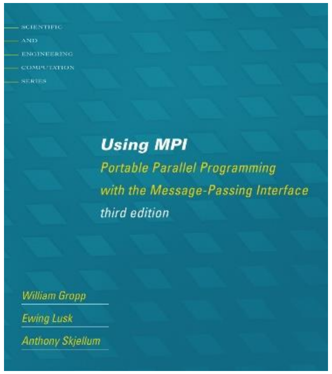
- Communicators encapsulate communication spaces for library safety
- Datatypes reduce copying costs and permit heterogeneity
- Multiple communication modes allow precise buffer management
- Extensive collective operations for scalable global communication
- Process topologies permit efficient process placement, user views of process layout
- Profiling interface encourages portable tools

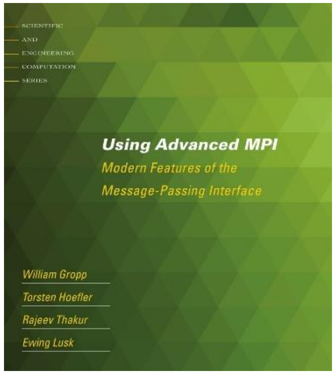
MPI References

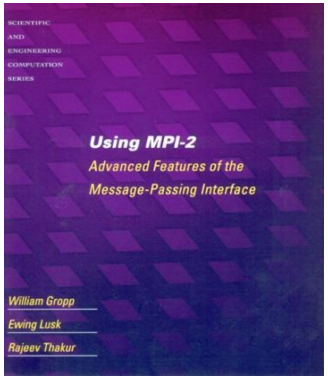
- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
  - Latest version MPI 3.1, released June 2015
- Other information on Web:
  - at http://www.mcs.anl.gov/research/projects/mpi/index.htm
  - pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages

Books on MPI

- *Using MPI: Portable Parallel Programming with the Message-Passing Interface (third edition)*, by Gropp, Lusk, and Skjellum, MIT Press, 2014.
- *Using Advanced MPI: Modern Features of the Message-Passing Interface, by* Gropp, Hoefler, Thakur, and Lusk, MIT Press, 2014
- *Using MPI-2: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Thakur, MIT Press, 1999.
- *MPI: The Complete Reference - Vol 1 The MPI Core,* by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1998.
- *MPI: The Complete Reference - Vol 2 The MPI Extensions*, by Gropp, Huss-Lederman, Lumsdaine, Lusk, Nitzberg, Saphir, and Snir, MIT Press, 1998.
- *Designing and Building Parallel Programs*, by Ian Foster, Addison-Wesley, 1995.
- *Parallel Programming with MPI*, by Peter Pacheco, Morgan-Kaufmann, 1997.







Finding Out About the Environment

- Two important questions that arise early in a 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*, a number between 0 and size-1, identifying the calling process

Some Basic Terminology

- Processes can be collected into groups
- Each message is sent in a context, and must be received in the same context
- A group and context together form a communicator
- A process is identified by its rank in the group associated with a communicator
- There is a default communicator whose group contains all initial processes, called **MPI_COMM_WORLD**

helloworld_mpi.c

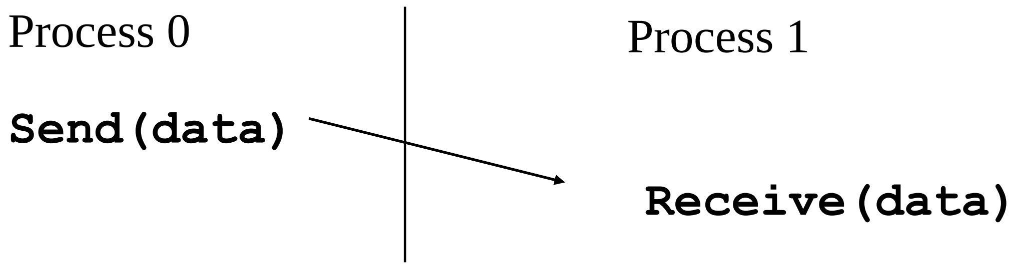
```
#include <mpi.h>
#include <stdio.h>

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

Notes on Hello World

- All MPI programs begin with MPI_Init and end with MPI_Finalize
- MPI_COMM_WORLD is defined by mpi.h and designates all processes in the MPI "job"
- Each statement executes independently in each process
  - including the `printf/print` statements
- The MPI Standard does not specify how to run an MPI program

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?

API

```
int MPI_Send(      const void *address,
                   int count,
                   MPI_Datatype datatype,
                   int dest_rank,
                   int tag,
                   MPI_Comm comm)

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

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

MPI Tags

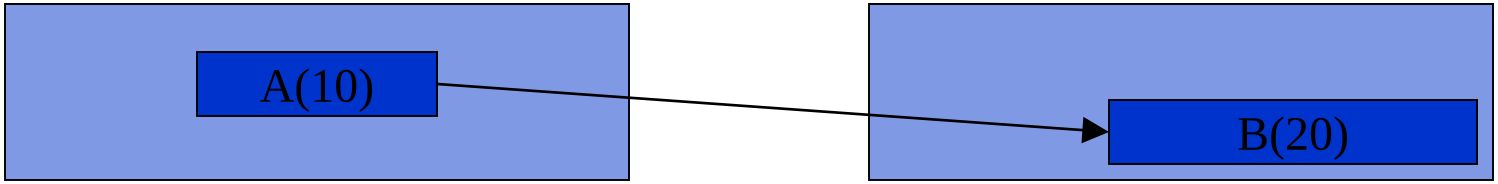
- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message

- 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

Tags and Contexts

- Separation of messages used to be accomplished by use of tags, but
  - this requires libraries to be aware of tags used by other libraries.
  - this can be defeated by use of "wild card" tags.
- Contexts are different from tags
  - no wild cards allowed
  - allocated dynamically by the system when a library sets up a communicator for its own use.
- User-defined tags still provided in MPI for user convenience in organizing application

MPI Basic (Blocking) Send



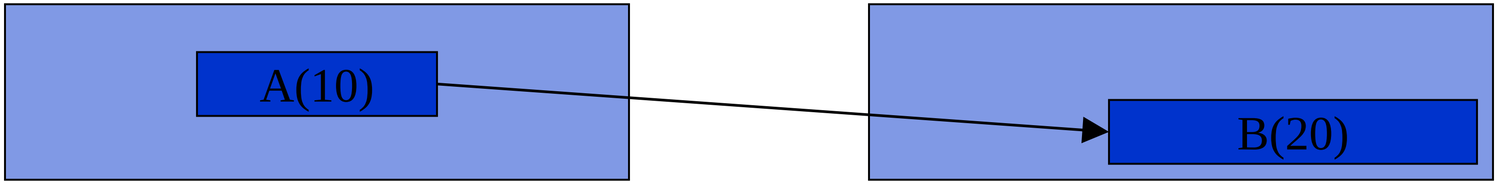
MPI_Send(A, 10, MPI_DOUBLE, 1, ...)

MPI_Recv(B, 20, MPI_DOUBLE, 0, ...)

`MPI_Send(address, count, datatype, dest, tag, comm)`

- 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 data has been delivered to the system and the memory in **`address`** can be reused. The message may not have been received by the target process.

MPI Basic (Blocking) Receive



MPI_Send(A, 10, MPI_DOUBLE, 1, ...)

MPI_Recv(B, 20, MPI_DOUBLE, 0, ...)

`MPI_Recv(address, count, datatype, source,tag, comm, status)`

- Waits until a matching (both **`source`** and **`tag`**) message is received from the system
- **`source`** is rank in communicator specified by **`comm`**, or **`MPI_ANY_SOURCE`**
- **`tag`** is a tag to be matched 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)

Retrieving Further Information

- **`status`** is a data structure allocated in the user's program.
- In C:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ...,
  &status )
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

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_RECV**

- (Next class) Using collectives:
  - **MPI_INIT**
  - **MPI_FINALIZE**
  - **MPI_COMM_SIZE**
  - **MPI_COMM_RANK**
  - **MPI_BCAST**
  - **MPI_REDUCE**

- You may use more for convenience or performance

Thinking in terms of distributed memory

Problem: want to create an array of size N and set every entry a[i]=i^2

In shared memory with OpenMP, we would still allocate an array of size N, and then have threads parallelize the setting of the entries

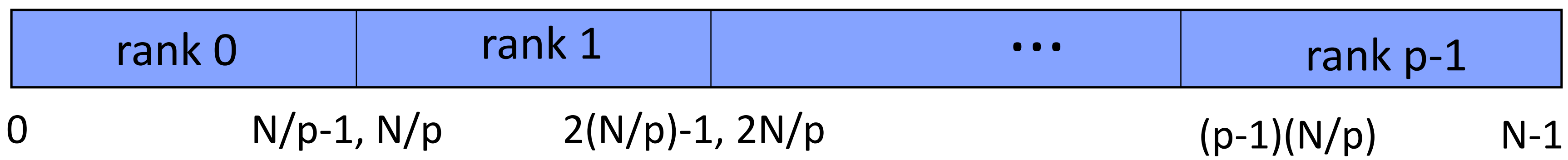
```
#pragma omp parallel for
for(int i = 0; i < N; i++)
   a[i] = i*i;
```

Now we have distributed memory. *There is no global array.*

- Each process will have a local array of size `my_N = N/mpisize`
  Each process will have to know which entries of the global array they are responsible for

Distributed Memory

Simple distribution: by contiguous chunks:




```
int my_N = N/mpisize;
int start = mpirank*my_N;

int *local_arr = (int*) calloc(sizeof(int), my_N);

for (i = 0; i < my_N; i++)
      local_arr[i] = (start+i)*(start+i);
```

More on Message Passing

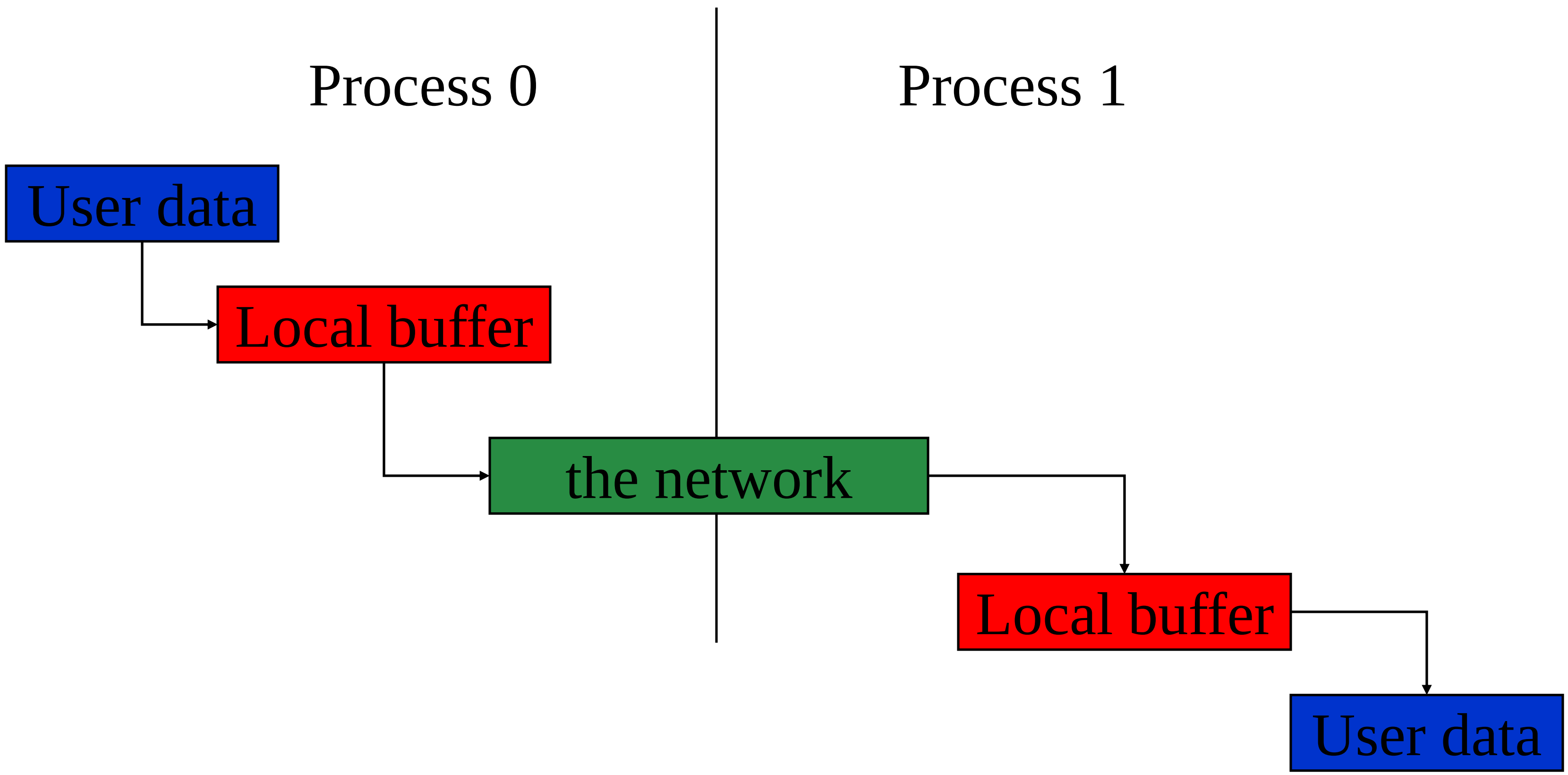
- Message passing is a simple programming model, but there are some special issues
  - Buffering and deadlock
  - Deterministic execution
  - Performance

Synchronization

- **`MPI_Barrier( comm )`**
- Blocks until all processes in the group of the communicator **`comm`** call it.
- Almost never required in a parallel program
  - Occasionally useful in measuring performance and load balancing

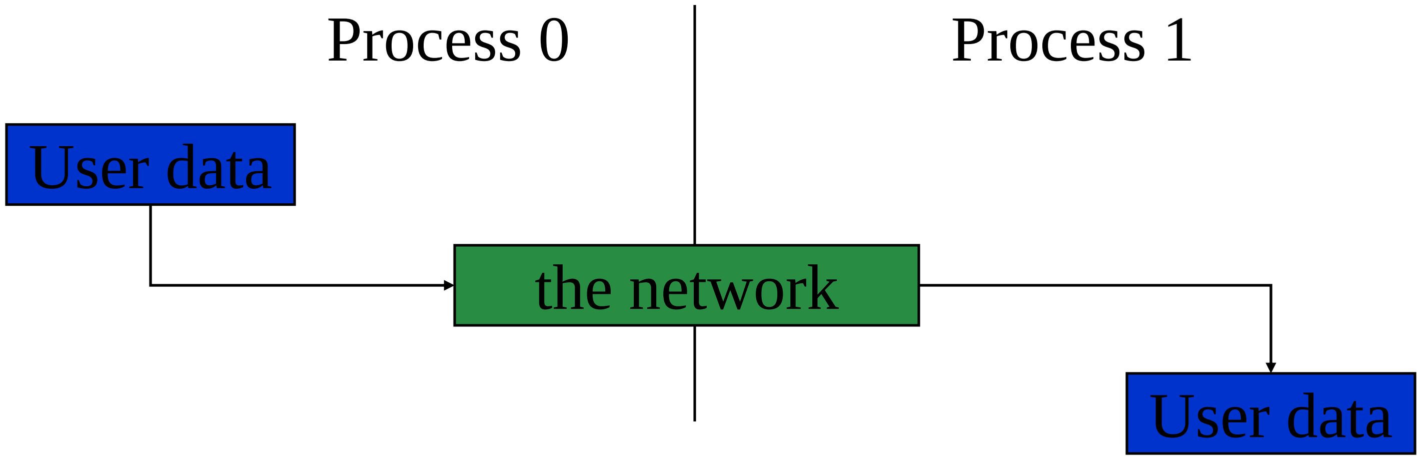
Buffers

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



Avoiding Buffering

- Avoiding copies uses less memory
- May use more or less 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.

Sources of 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)
- What happens with this code?

Process 0	Process 1
Send(1)	**Send(0)**
Recv(1)	**Recv(0)**

- 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 the "unsafe" Problem

- Order the operations more carefully:

Process 0	Process 1
Send(1)	**Recv(0)**
Recv(1)	**Send(0)**

- Supply receive buffer at same time as send:

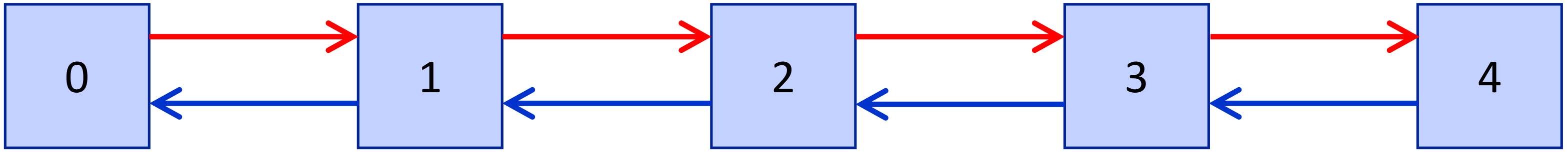
Process 0	Process 1
Sendrecv(1)	**Sendrecv(0)**

MPI_Sendrecv()

```
int MPI_Sendrecv(
            const void *sendbuf,
            int sendcount,
            MPI_Datatype sendtype,
            int dest,
            int sendtag,
            void *recvbuf,
            int recvcount,
            MPI_Datatype recvtype,
            int source,
            int recvtag,
            MPI_Comm comm,
            MPI_Status *status  )
```

Example

Row of processors. Each wants to exchange 1 double number with its neighbors.




```
int left, right;
if (mpirank == 0)
        left = MPI_PROC_NULL;
else
        left = mpirank - 1;
if (mpirank == mpisize -1)
        right = MPI_PROC_NULL;
else
        right = mpirank +1;

MPI_Sendrecv(&sendbufR,  1, MPI_DOUBLE, right, 0, &recvbufR, 1,
        MPI_DOUBLE, right, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

MPI_Sendrecv(&sendbufL,  1, MPI_DOUBLE, left, 0, &recvbufL, 1,
        MPI_DOUBLE, left, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

More Solutions to the "unsafe" Problem

- 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's Non-blocking Operations

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

```
MPI_Request request, request2;
MPI_Status status;

MPI_Isend(start, count, datatype,
    dest, tag, comm, &request);

MPI_Irecv(start, count, datatype,
    dest, tag, comm, &request2);

MPI_Wait(&request, &status);

MPI_Wait(&request2, &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

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.

Summary: Communication Modes

- MPI provides multiple *modes* for sending messages:
  - Synchronous mode (**MPI_Send**): the send does not complete until a matching receive has begun. (Unsafe programs deadlock.)
  - Buffered mode (**MPI_Bsend**): the user supplies a buffer to the system for its use. (User allocates enough memory to make an unsafe program safe.)
  - Ready mode (**MPI_Rsend**): user guarantees that a matching receive has been posted.
    - Allows access to fast protocols
    - undefined behavior if matching receive not posted
- Non-blocking versions (**MPI_Isend**, etc.)
- **MPI_Recv** receives messages sent in any mode.
- See www.mpi-forum.org for summary of all flavors of send/receive