Outline

- Synchronous/Asynchronous Simulation
- Brief review of Shared Memory Processors
- Distributed Memory Processors
- Programming using MPI

Synchronous Circuit Simulation

- Circuit is a graph made up of subcircuits connected by wires
  - Component simulations need to interact if they share a wire.
  - Data structure is (irregular) graph of subcircuits.
  - Parallel algorithm is timing-driven or synchronous:
    - Evaluate all components at every *timestep* (determined by known circuit delay)
- Graph partitioning assigns subgraphs to processors
  - Determines parallelism and locality.
  - Goal 1 is to evenly distribute subgraphs to nodes (load balance).
  - Goal 2 is to minimize edge crossings (minimize communication).
  - Easy for meshes, NP-hard in general, so we will approximate (future lecture)

Asynchronous Simulation

- Synchronous simulations may waste time:
  - Simulates even when the inputs do not change
- Asynchronous (event-driven) simulations update only when an event arrives from another component:
  - No global time steps, but individual events contain time stamp.
  - Example: Game of life in loosely connected ponds (don’t simulate empty ponds).
  - Example: Circuit simulation with delays (events are gates changing).
  - Example: Traffic simulation (events are cars changing lanes, etc.).
- Asynchronous is more efficient, but harder to parallelize
  - On distributed memory, events are naturally implemented as messages between processors (eg using MPI), but how do you know when to execute a “receive”?

Scheduling Asynchronous Circuit Simulation

- Conservative:
  - Only simulate up to (and including) the minimum time stamp of inputs.
  - Need deadlock detection if there are cycles in graph
  - Example: Pthor circuit simulator
- Speculative (or Optimistic):
  - Assume no new inputs will arrive and keep simulating.
  - May need to backup if assumption wrong, using timestamps
- Example: Timewarp [D. Jefferson], Parswec [Wen,Yelick].
- Optimizing load balance and locality is difficult:
  - Locality means putting tightly coupled subcircuit on one processor.
  - Since “active” part of circuit likely to be in a tightly coupled subcircuit, this may be bad for load balance.

Summary of Discrete Event Simulations

- Model of the world is discrete
  - Both time and space
- Approaches
  - Decompose domain, i.e., set of objects
  - Run each component ahead using
    - Synchronous: communicate at end of each timestep
    - Asynchronous: communicate on-demand
      - Conservative scheduling – wait for inputs
        - need deadlock detection
      - Speculative scheduling – assume no inputs
        - roll back if necessary

A Brief Review:
Shared memory multiprocessors

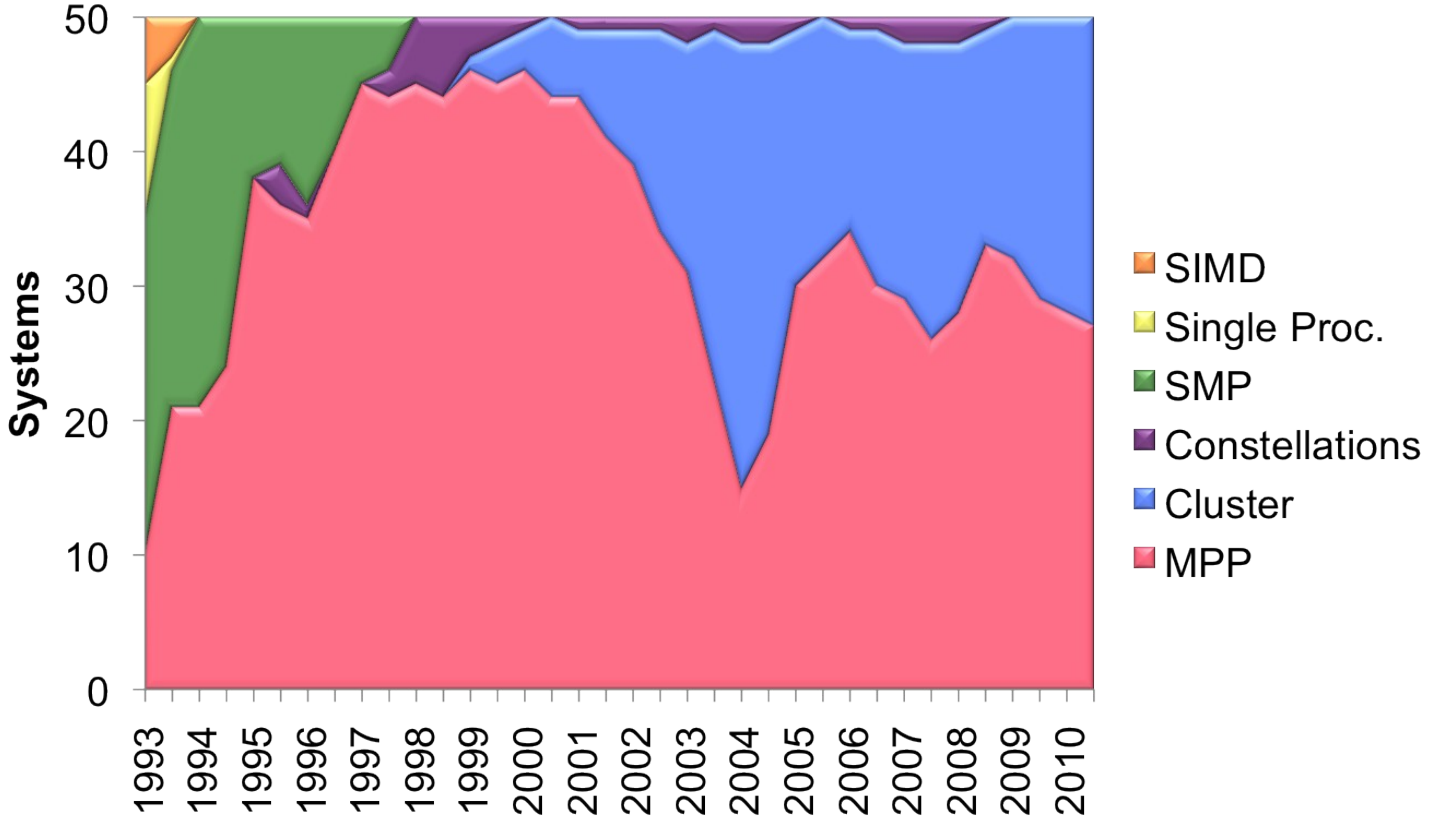
- Caches may be either shared or distributed
  - Multicore chips are likely to have shared caches
  - Cache hit performance is better if they are distributed (each cache is smaller/closer) but they must be kept coherent -- multiple cached copies of same location must be kept equal.
- Requires clever hardware
- Distant memory much more expensive to access
- Machines scale to 10s or 100s of processors

Outline:

Distributed Memory Architectures

- Properties of communication networks
- Topologies
- Performance models

Architectures (as of 2010)
(a long time ago)



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

An analogy: Networks as streets

- To have a large number of simultaneous transfers, 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

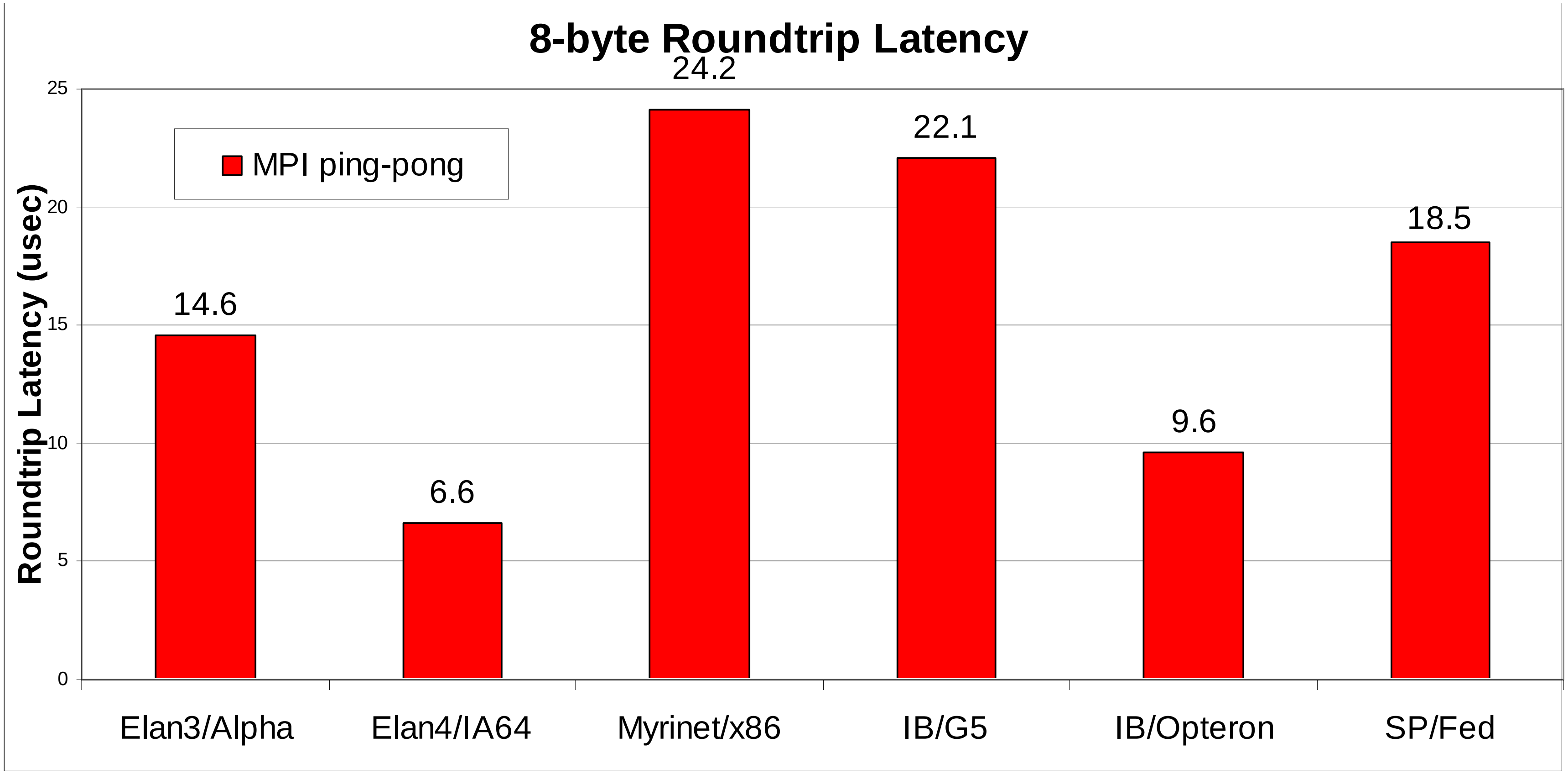
Network Design

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

- 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

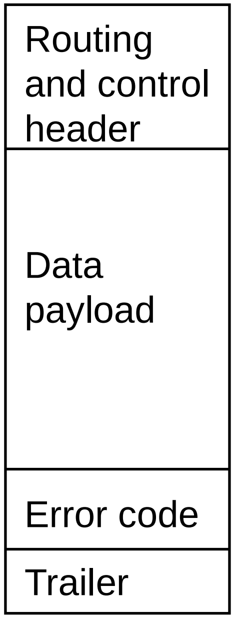
Latency on earlier Machines/Networks



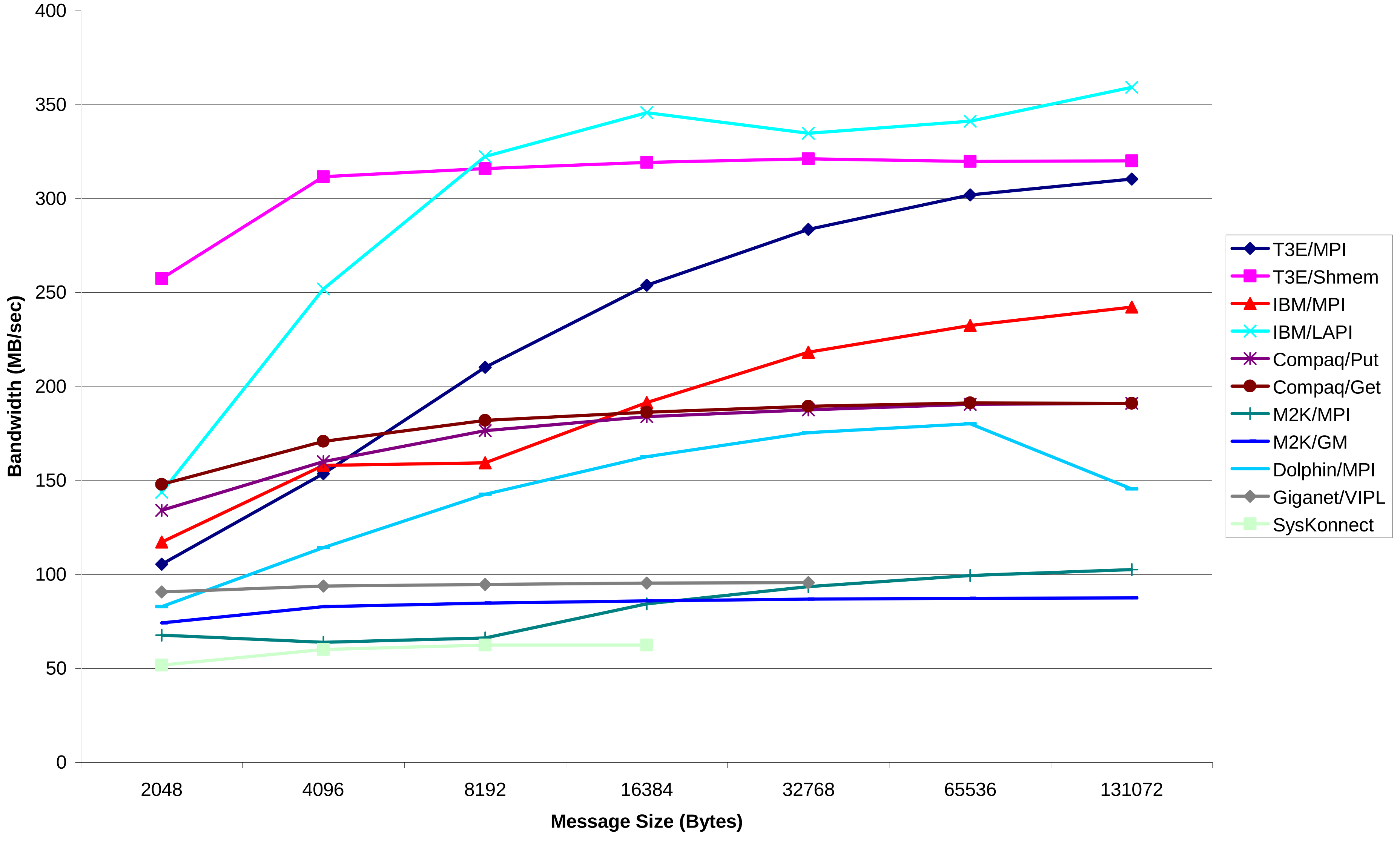
- Latencies shown are from a ping-pong test using MPI
- These are roundtrip numbers: many people use ½ of roundtrip time to approximate 1-way latency (which can't easily be measured)
- Latency hasn't improved!

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* 220 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

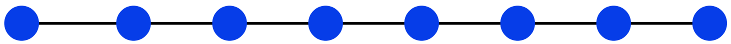
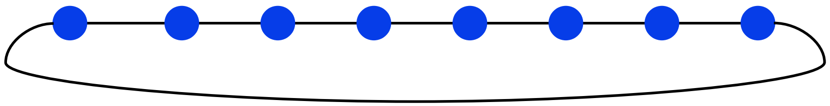


Bandwidth (MB/sec)
0
50
100
150
200
250
300
350
400
2048
4096
8192
16384
32768
65536
131072
Message Size (Bytes)
T3E/MPI
T3E/Shmem
IBM/MPI
IBM/LAPI
Compaq/Put
Compaq/Get
M2K/MPI
M2K/GM
Dolphin/MPI
Giganet/VIPL
SysKonnect

Performance: Bisection Bandwidth

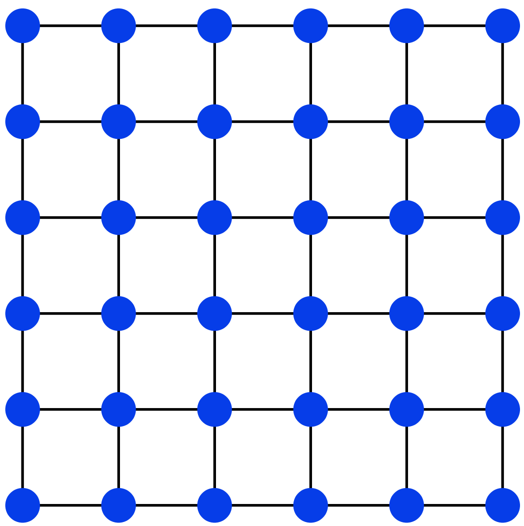
- Bisection bandwidth: bandwidth across smallest cut that divides network into two equal halves
- Bandwidth across “narrowest” part of the network
- Bisection bandwidth is important for algorithms in which all processors need to communicate with all others

Topologies: Linear and Ring

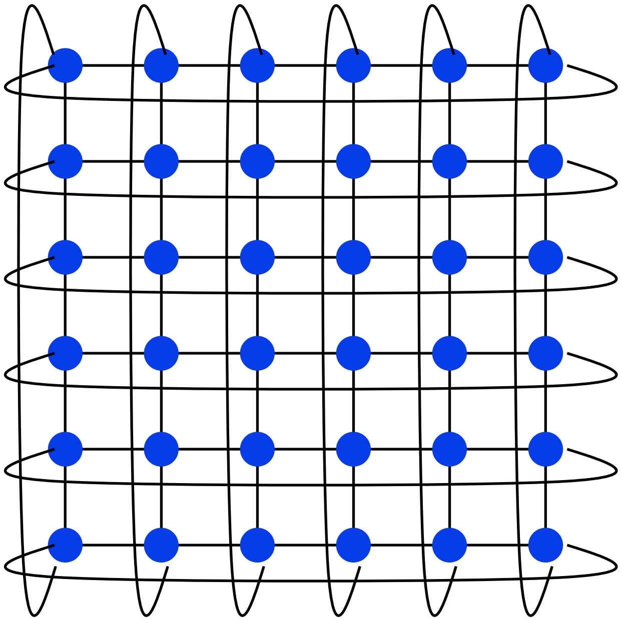
- Linear array 
  - Diameter = n-1; average distance ~n/3.
  - Bisection bandwidth = 1 (in units of link bandwidth).
- Torus or Ring 
  - Diameter = n/2; average distance ~ n/4.
  - Bisection bandwidth = 2.
  - Natural for algorithms that work with 1D arrays.

Topologies: Meshes and Tori

- Two dimensional mesh
  - Diameter = 2 * (sqrt(n) – 1)
  - Bisection bandwidth = sqrt(n)



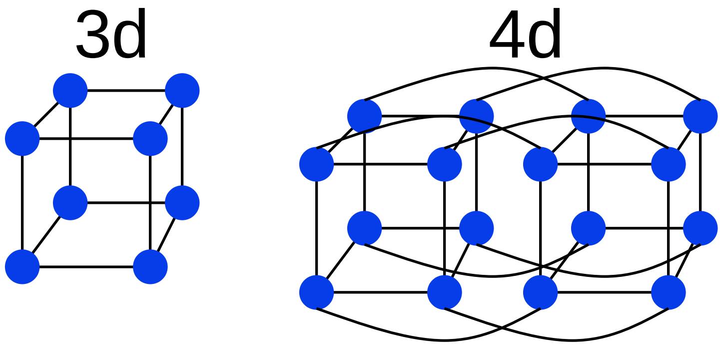
- Two dimensional torus
  - Diameter = sqrt(n)
  - Bisection bandwidth = 2* sqrt(n)



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

Hypercubes

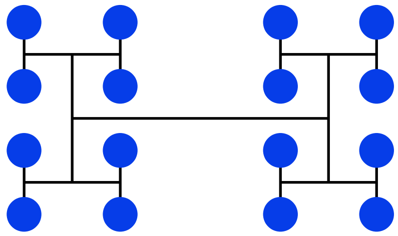
- Number of nodes $n = 2^d$ for dimension d.
- Diameter = d.
- Bisection bandwidth = n/2.

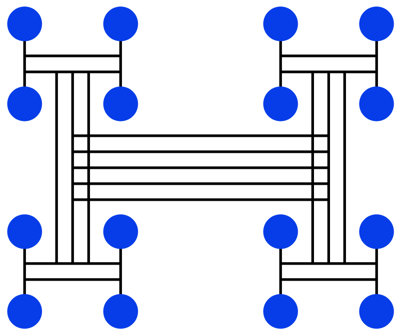


- Popular in early machines (Intel iPSC, NCUBE).
- Lots of clever algorithms
- Greycode addressing: Each node connected to d others with 1 bit different.

Trees

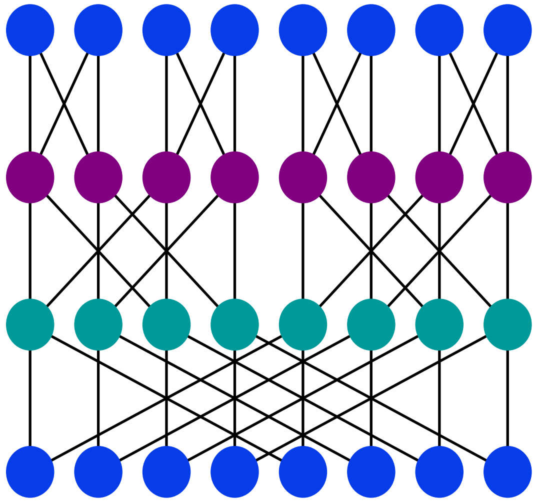
- Diameter = log n.
- Bisection bandwidth = 1.
- Easy layout as planar graph (H-trees)
- Many tree algorithms (e.g., summation).
- Fat trees avoid bisection bandwidth problem:
  - More (or wider) links near top.
  - Example: Thinking Machines CM-5





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

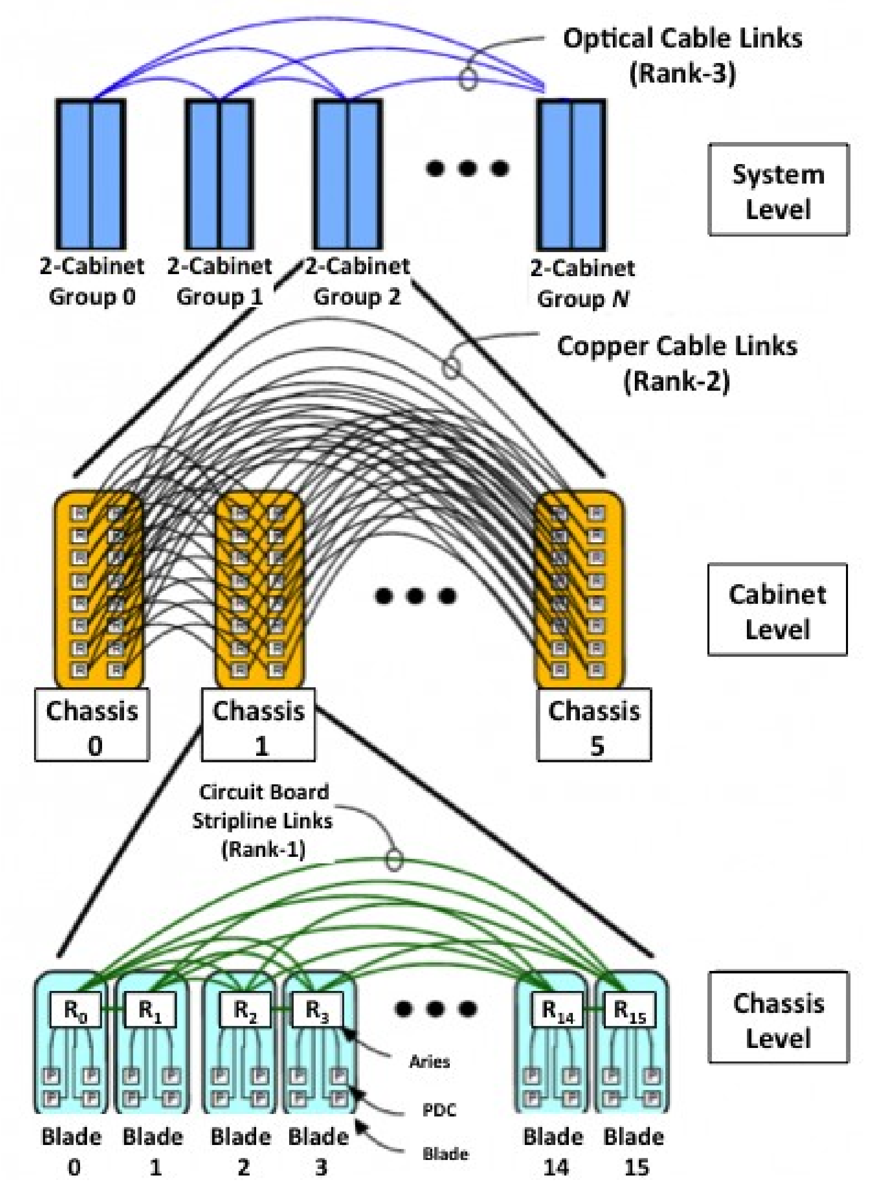


Dragonflies

- Motivation: Exploit gap in cost and performance between optical interconnects (which go between cabinets in a machine room) and electrical networks (inside cabinet)
  - Optical (fiber) more expensive but higher bandwidth when long
  - Electrical (copper) networks cheaper, faster when short
- 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.
- Uses a randomized routing algorithm
- Outcome: programmer can (usually) ignore topology, get good performance
  - Important in virtualized, dynamic environment
  - Drawback: variable performance

Dragonflies: Cray XC30

- Each router (Rx) is connected to four processors nodes (P). Sixteen blades, each with one router, are connected together at the chassis level by circuit board links (Rank-1 Subtree)
- Six chassis are connected together to form the two-cabinet group by using copper cabling at the cabinet level (Rank-2 Subtree)
- Finally, the two-cabinet groups are connected to each other by using optical cables for the global links (Rank-3 Subtree)
- Rank 1 routing is characterized by one electrical link between routers. Rank 2 is characterized by three electrical links and Rank 3 is characterized by two optical links between routers.



Why so many topologies?

- Different systems have different needs
  - 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

Shared Memory Performance Models

- Parallel Random Access Memory (PRAM)
  - All memory access operations complete in one clock period -- no concept of memory hierarchy ("too good to be true").
  - OK for understanding whether an algorithm has enough parallelism at all
- Parallel algorithm design strategy: first do a PRAM algorithm, then worry about memory/communication time
- Slightly more realistic versions exist
  - E.g., Concurrent Read Exclusive Write (CREW) PRAM
  - Still missing the memory hierarchy

Latency and Bandwidth Model

- Time to send message of length n is roughly

$$\text{Time} = \text{latency} + n*\text{cost\_per\_word}$$
$$= \text{latency} + n/\text{bandwidth}$$

- Topology is assumed irrelevant
- Often called "α-β model" and written

$$\text{Time} = \alpha + n*\beta$$

- Usually $\alpha >> \beta >>$ time per flop.
  - One long message is cheaper than many short ones.

$$\alpha + n*\beta \quad << \quad n*(\alpha + 1*\beta)$$

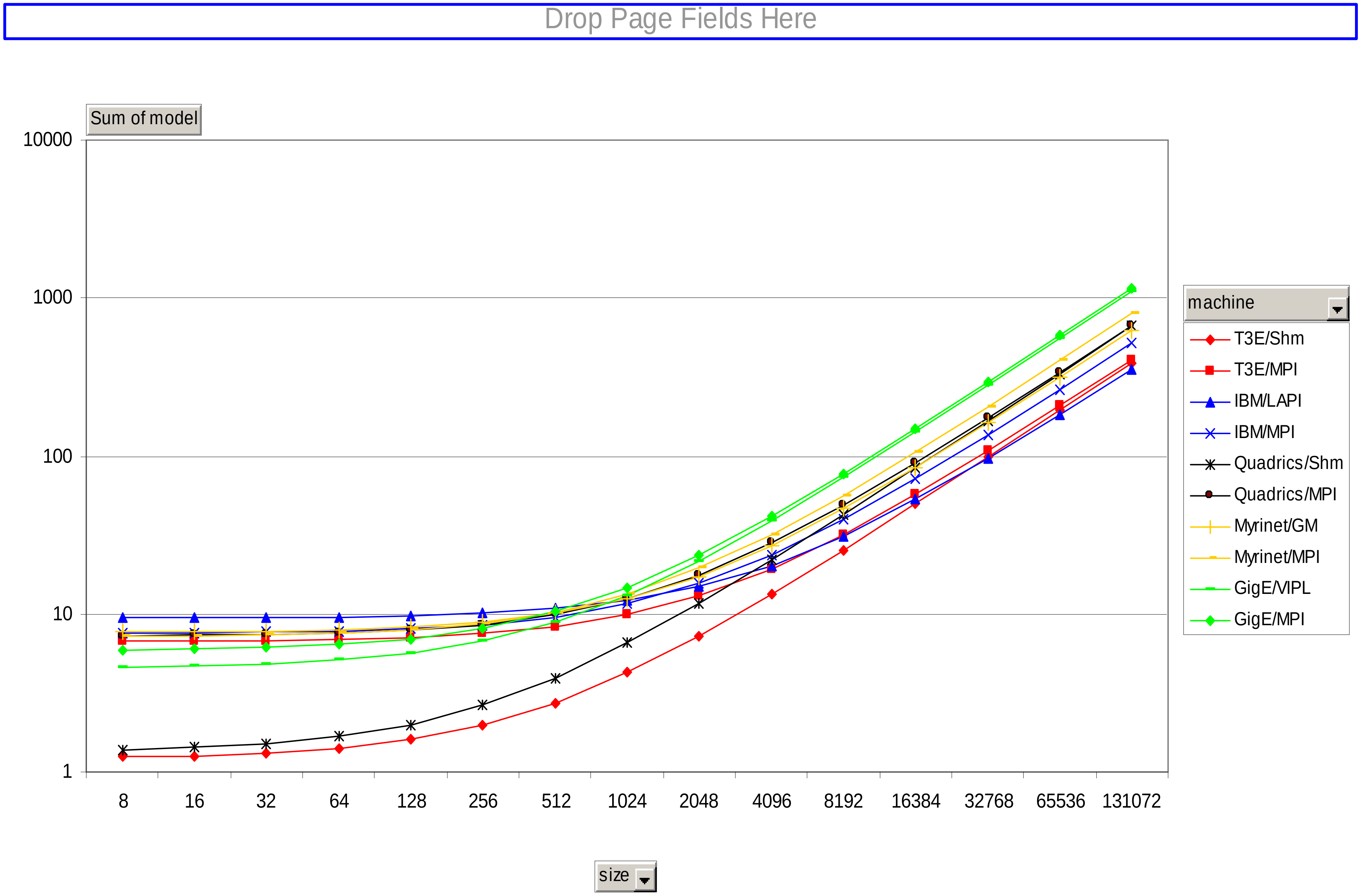
  - Can do hundreds or thousands of flops for cost of one message.
- Lesson: *Need large computation-to-communication ratio to be efficient*

Example Alpha-Beta parameters

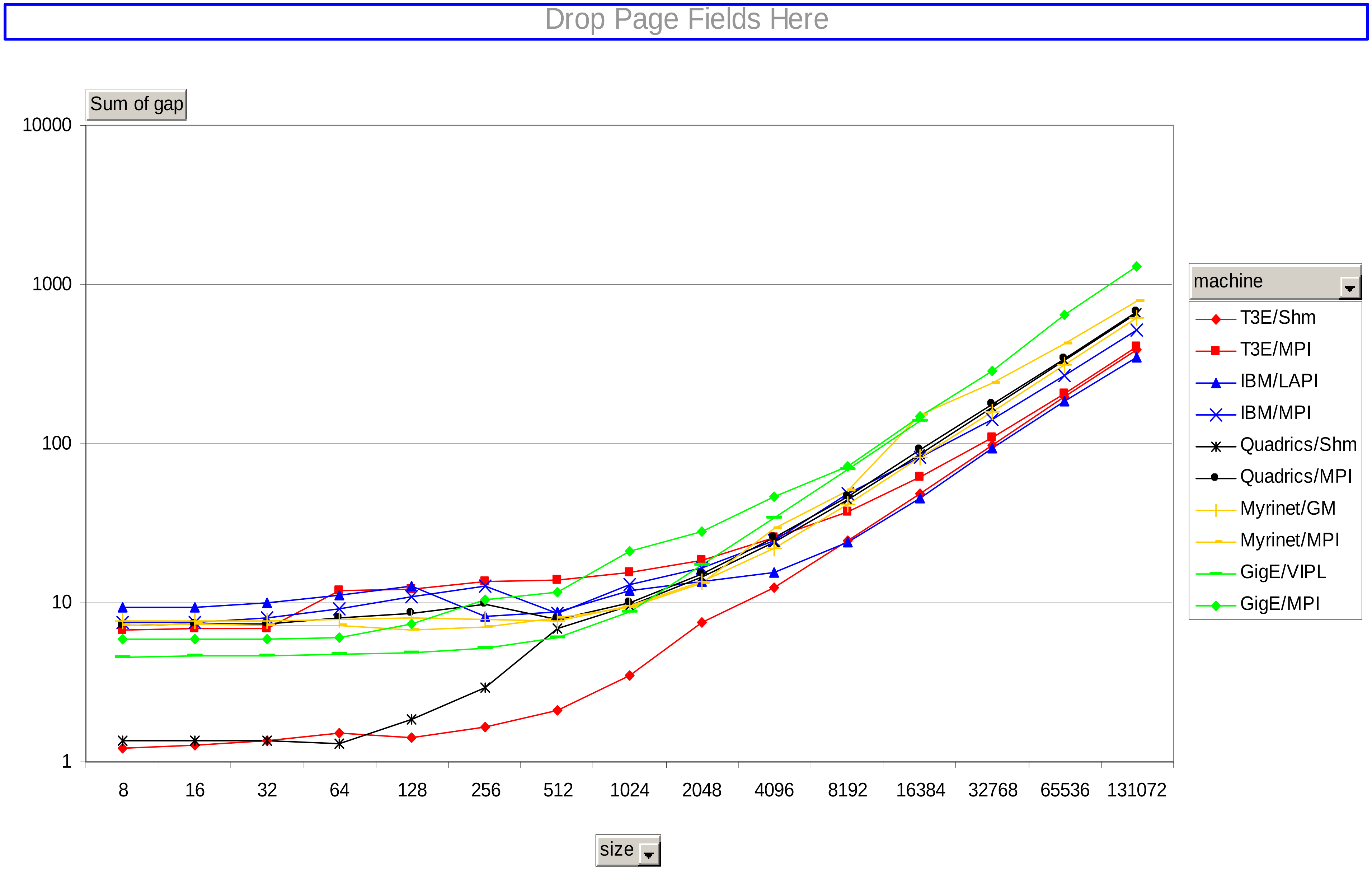
machine	α	β
T3E/Shm	1.2	0.003
T3E/MPI	6.7	0.003
IBM/LAPI	9.4	0.003
IBM/MPI	7.6	0.004
Quadrics/Get	3.267	0.00498
Quadrics/Shm	1.3	0.005
Quadrics/MPI	7.3	0.005
Myrinet/GM	7.7	0.005
Myrinet/MPI	7.2	0.006
Dolphin/MPI	7.767	0.00529
Giganet/VIPL	3.0	0.010
GigE/VIPL	4.6	0.008
GigE/MPI	5.854	0.00872

α is latency in usecs
β is BW in usecs per Byte

Model Time Varying Message Size



Measured Message Time



Programming Distributed Memory Machines with Message Passing

- Overview of MPI
- Basic send/receive use
- Non-blocking communication
- Collectives

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 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
    - No locks because there are no shared variables to protect
  - Inquiries
    - 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: http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
  - Latest version MPI 3.0, released Sept 2012
- Other information on Web: http://www.mcs.anl.gov/mpi
  - 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 (2nd edition)*, by Gropp, Lusk, and Skjellum, MIT Press, 1999.
- *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.

Environmental Inquiries

- 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

Environmental Inquiries (C)

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );  // Required
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize(); // Required
    return 0;
}
```

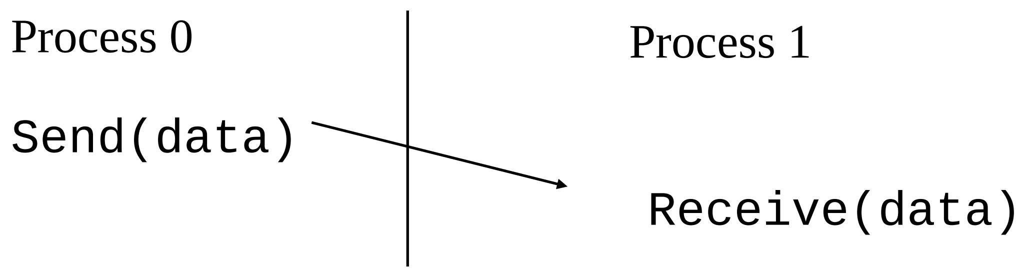
Each statement executes independently in each process including the `printf/print` statements

The MPI-1 Standard does not specify how to run an MPI program, but many implementations provide

`mpirun –np 4 a.out`

MPI Basic Send / Receive

- We need to fill in the details in



- 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 Basic Concepts

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

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
- Some non-MPI message-passing systems have called tags “message types”. MPI calls them tags to avoid confusion with datatypes

MPI Blocking Send

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

- The message buffer is described by (**`start, count, datatype`**).
- The target process is specified by **`dest`**, which is the rank of the target process in the communicator specified by **`comm`**.
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may *not* have been received by the target process.

MPI Blocking Receive

- **`MPI_RECV(start, count, datatype, source, tag, comm, status)`**
- Waits until a matching (both `source` and `tag`) message is received from the system, and the buffer can be used
- `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)

Simple Send/Receive

```
#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;
}
```

Interpreting status

- **`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 );
```

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

Collectives

- *Collective* routines provide a higher-level way to organize a parallel program
- Each process executes the same communication operations
- MPI provides a rich set of collective operations…

Collectives in MPI

- Collective operations are called by all processes in a communicator
- **`MPI_BCAST`** distributes data from one process (the root) to all others in a communicator
- **`MPI_REDUCE`** combines data from all processes in communicator and returns it to one process
- In many numerical algorithms, **`SEND/RECEIVE`** can be replaced by **`BCAST/REDUCE`**, improving both simplicity and efficiency

MPI *can* be simple

Many parallel programs can be written using just these six functions, only two of which are non-trivial

Point to point:

MPI_INIT

MPI_FINALIZE

MPI_COMM_SIZE

MPI_COMM_RANK

MPI_SEND

MPI_RECV

Using collectives:

MPI_INIT

MPI_FINALIZE

MPI_COMM_SIZE

MPI_COMM_RANK

MPI_BCAST

MPI_REDUCE

Pi in MPI

- Simple program written in a data parallel style in MPI
- E.g., for a reduction, each process will first reduce (sum) its own values, then call a collective to combine them
- Estimates pi by approximating the area of the quadrant of a unit circle
- Each process gets 1/p of the intervals (mapped round robin, i.e., a cyclic mapping)

Pi in MPI

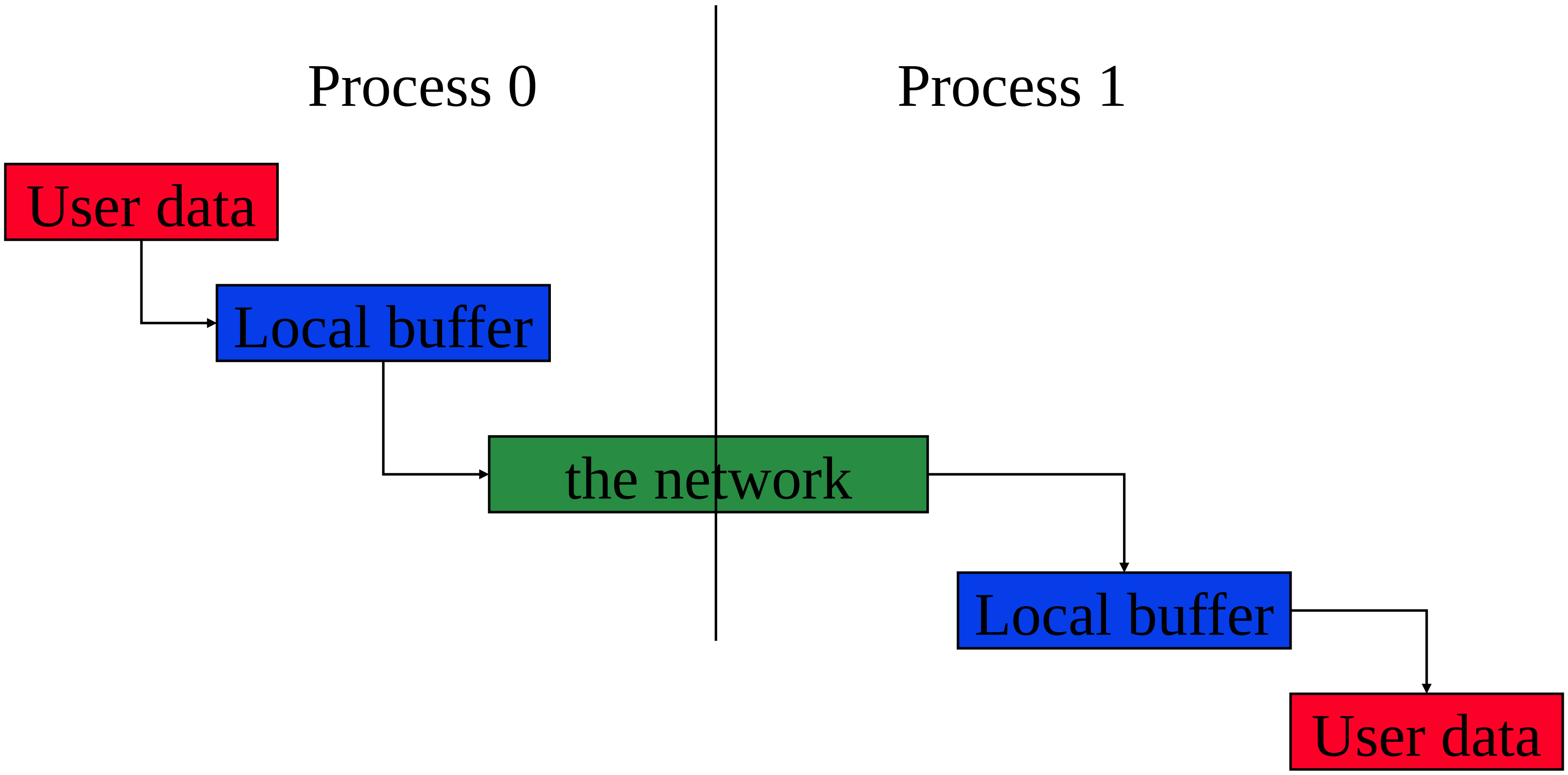
```
#include "mpi.h"
#include <math.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
  int done = 0, n, myid, numprocs, i, rc;
  double PI25DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x, a;
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  while (!done)  {
    if (myid == 0) {
      printf("Enter the number of intervals: (0 quits) ");
      scanf("%d",&n);
    }
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0) break;h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
      x = h * ((double)i - 0.5);
      sum += 4.0 * sqrt(1.0 - x*x);
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
               MPI_COMM_WORLD);
    if (myid == 0)
      printf("pi is approximately %.16f, Error is .16f\n",
             pi, fabs(pi - PI25DT));
  }
  MPI_Finalize();
  return 0;
}
```

More on Message Passing

Message passing is a simple programming model, but there are issues

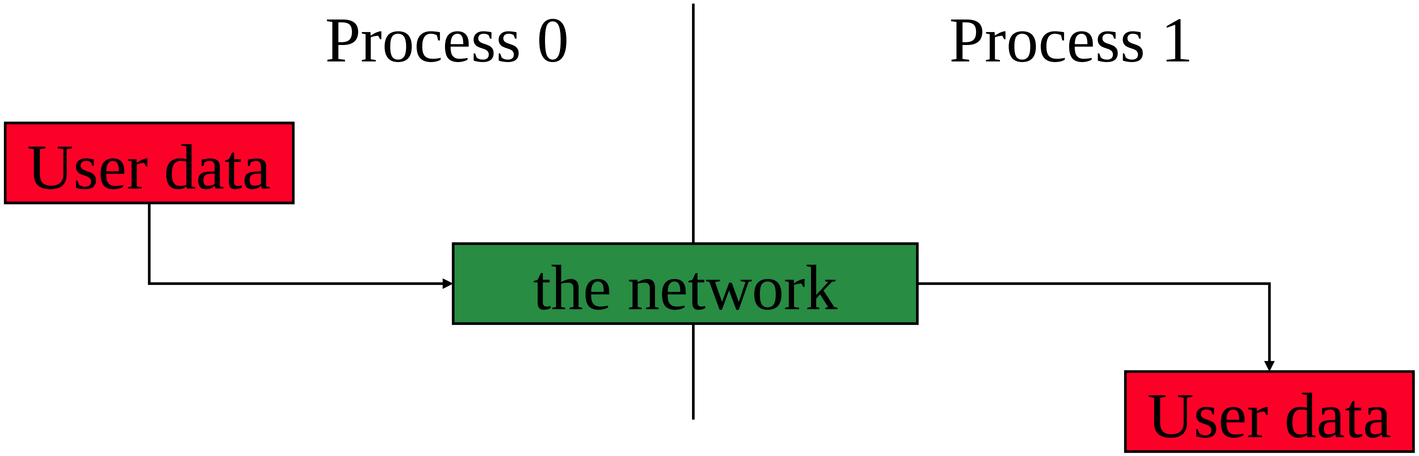
- Buffering and deadlock
- Deterministic execution
- Performance

Buffers: Where does data go?



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 Deadlock

- 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