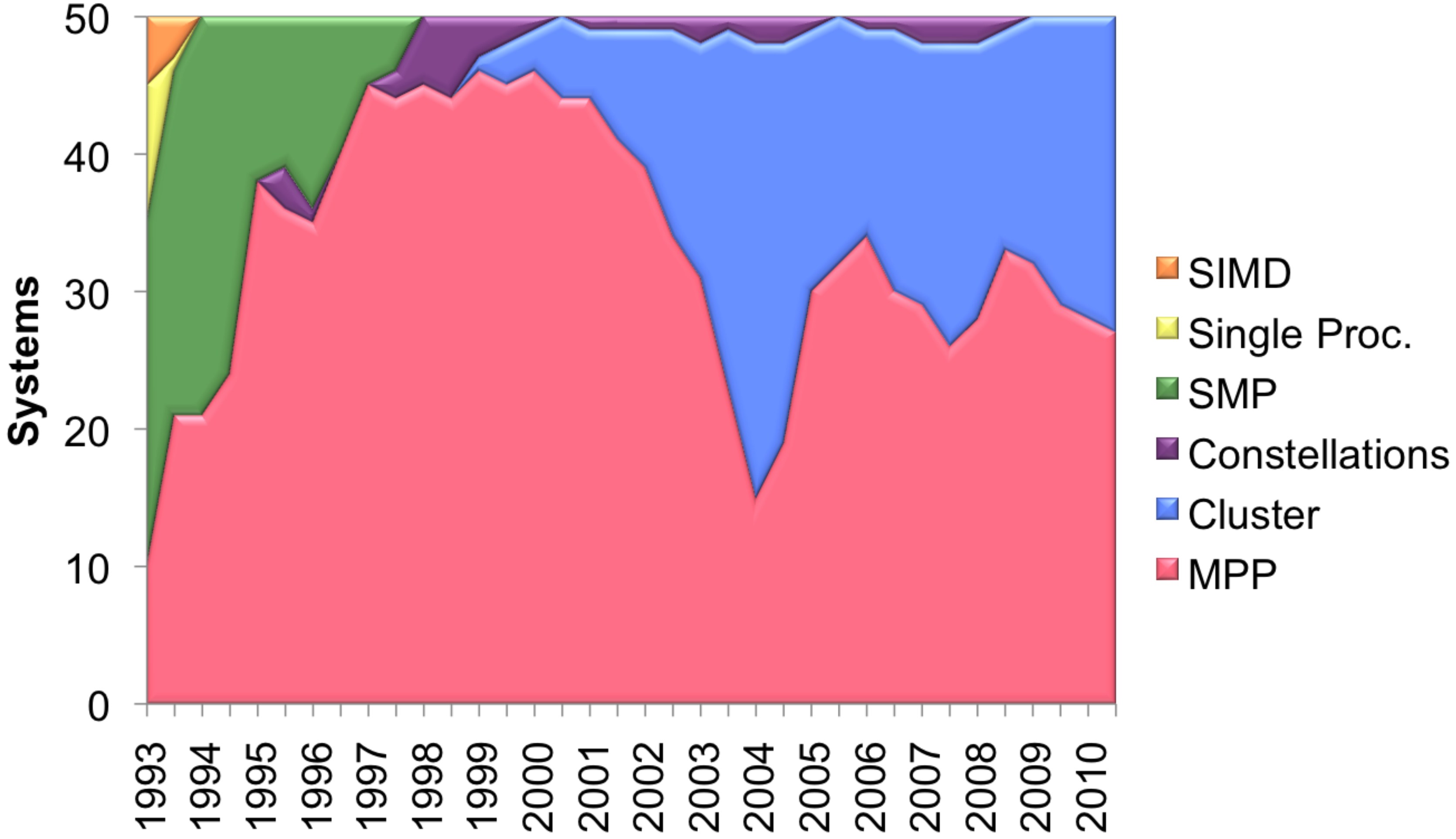
Distributed Memory Machines and Programming

James Demmel and Kathy Yelick
www.cs.berkeley.edu/~demmel/cs267_Spr11

Recap of Last Lecture

- 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.
- Shared memory programming
    - Starting, stopping threads.
    - Communication by reading/writing shared variables.
    - Synchronization with locks, barriers.

Architectures (TOP50)

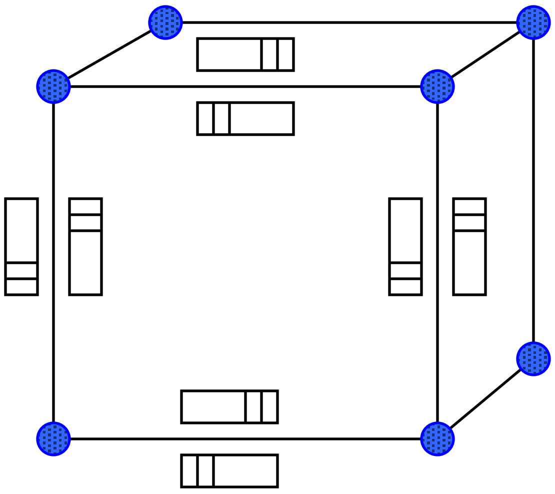


Outline

- Distributed Memory Architectures
  - Properties of communication networks
  - Topologies
  - Performance models
- Programming Distributed Memory Machines using Message Passing
  - Overview of MPI
  - Basic send/receive use
  - Non-blocking communication
  - Collectives

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.
  - Bandwidth: how much data can be moved per unit time.
    - Bandwidth is limited by the number of wires and the rate at which each wire can accept data.

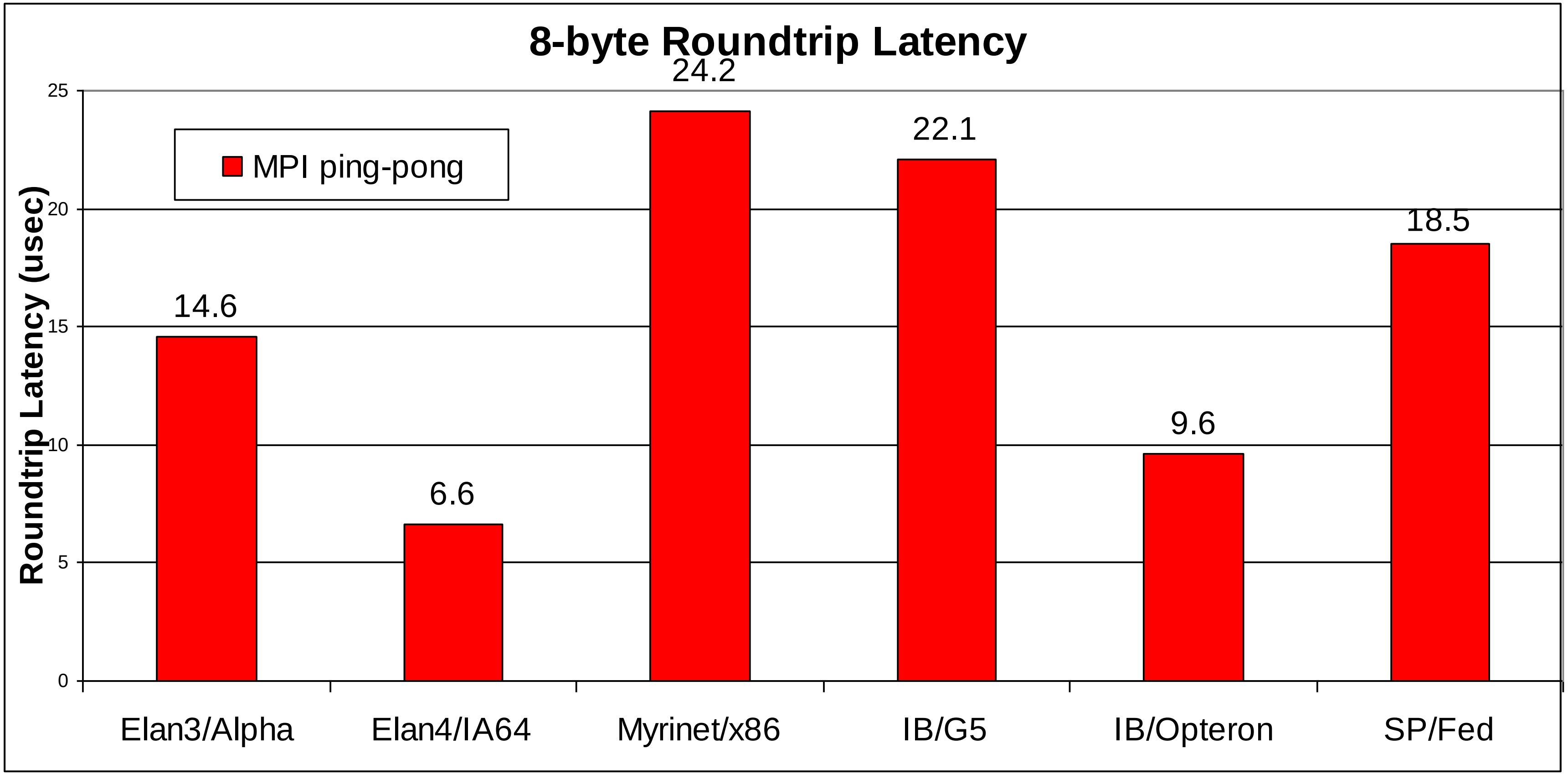
Design Characteristics of a Network

- Topology (how things are connected)
  - Crossbar, ring, 2-D and 3-D mesh or torus, hypercube, tree, butterfly, perfect shuffle ....
- Routing algorithm:
- 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.
- 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

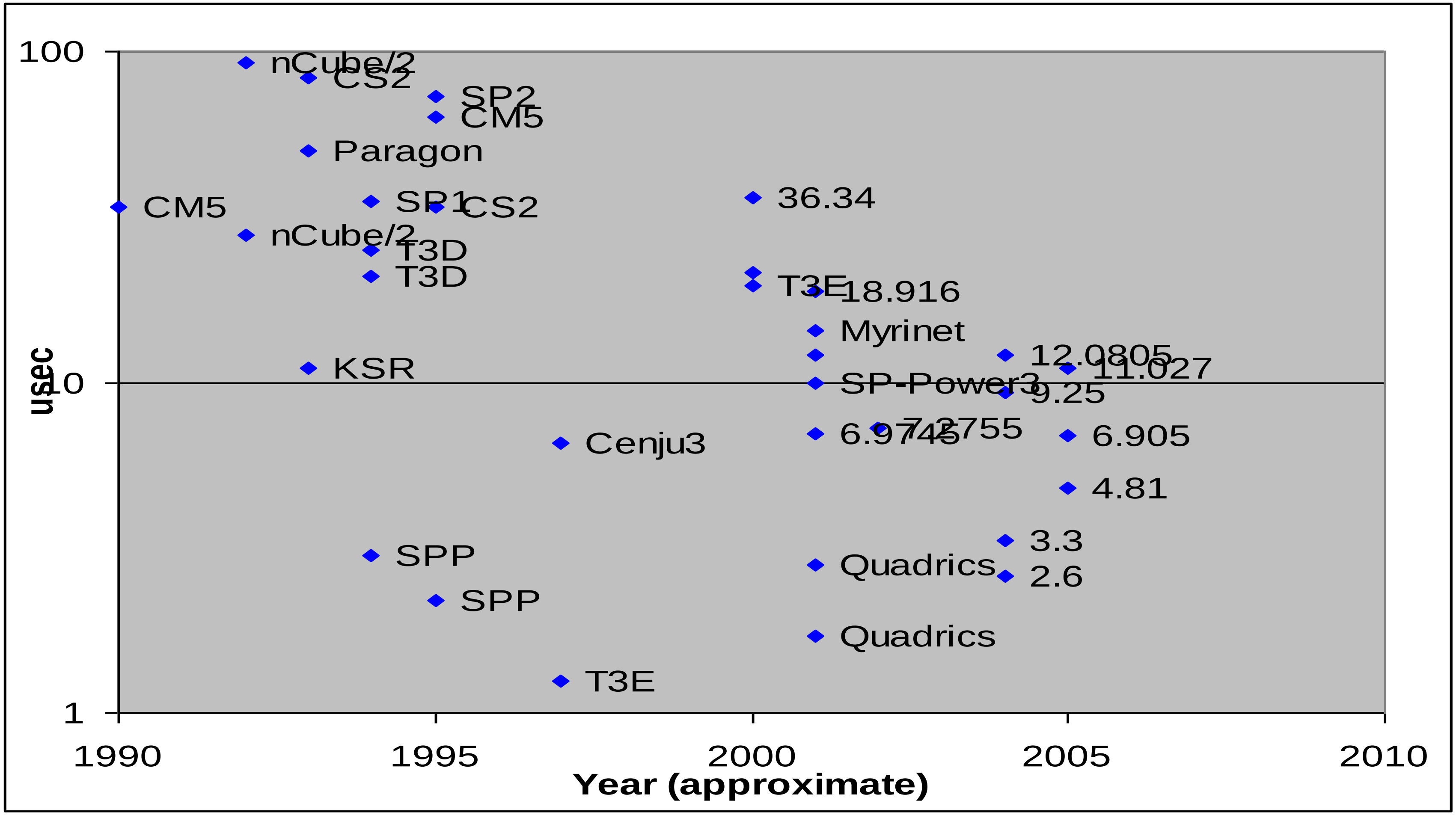
- Diameter: the maximum (over all pairs of nodes) of the shortest path between a given pair of nodes.
- Latency: delay between send and receive times
  - Latency tends to vary widely across architectures
  - Vendors often report hardware latencies (wire time)
  - Application programmers care about software 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 Recent 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

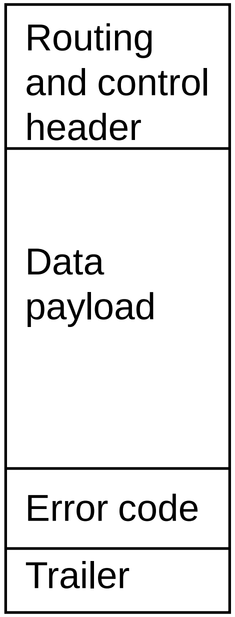


- Latency has not improved significantly, unlike Moore's Law
  - T3E (shmem) was lowest point – in 1997

Data from Kathy Yelick, UCB and NERSC

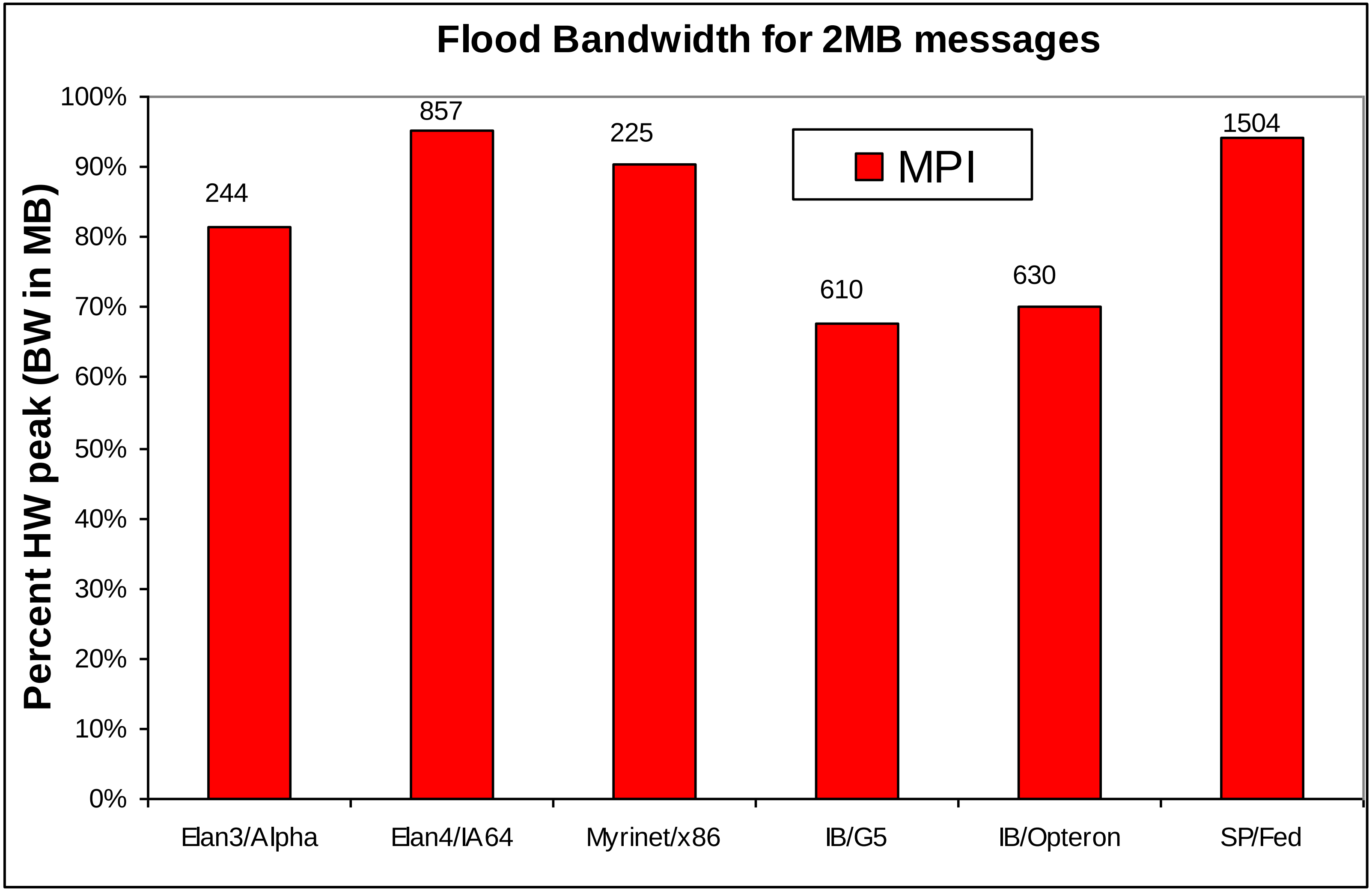
Performance Properties of a Network: Bandwidth

- 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.



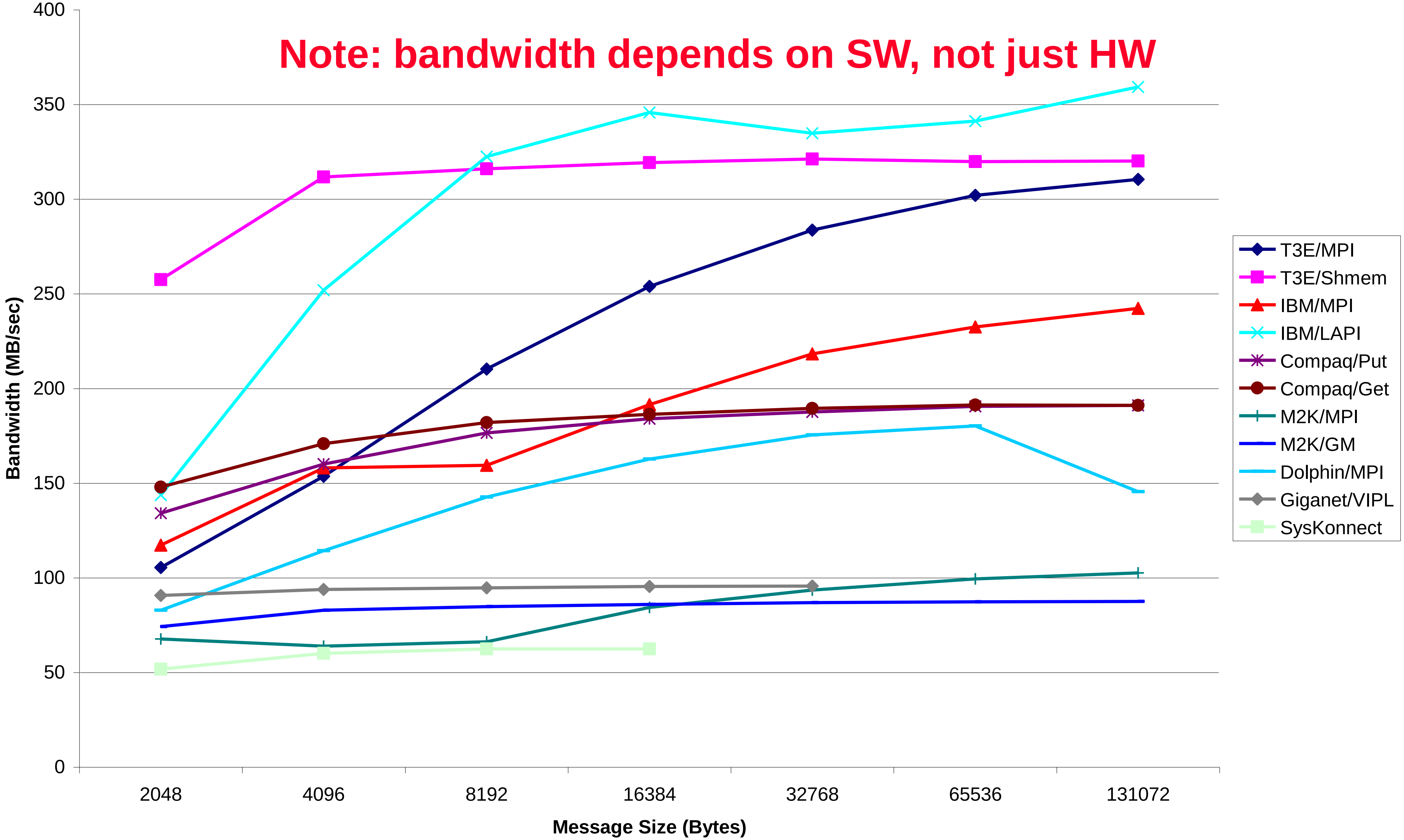
- Bandwidth is important for applications with mostly large messages

Bandwidth on Existing Networks



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

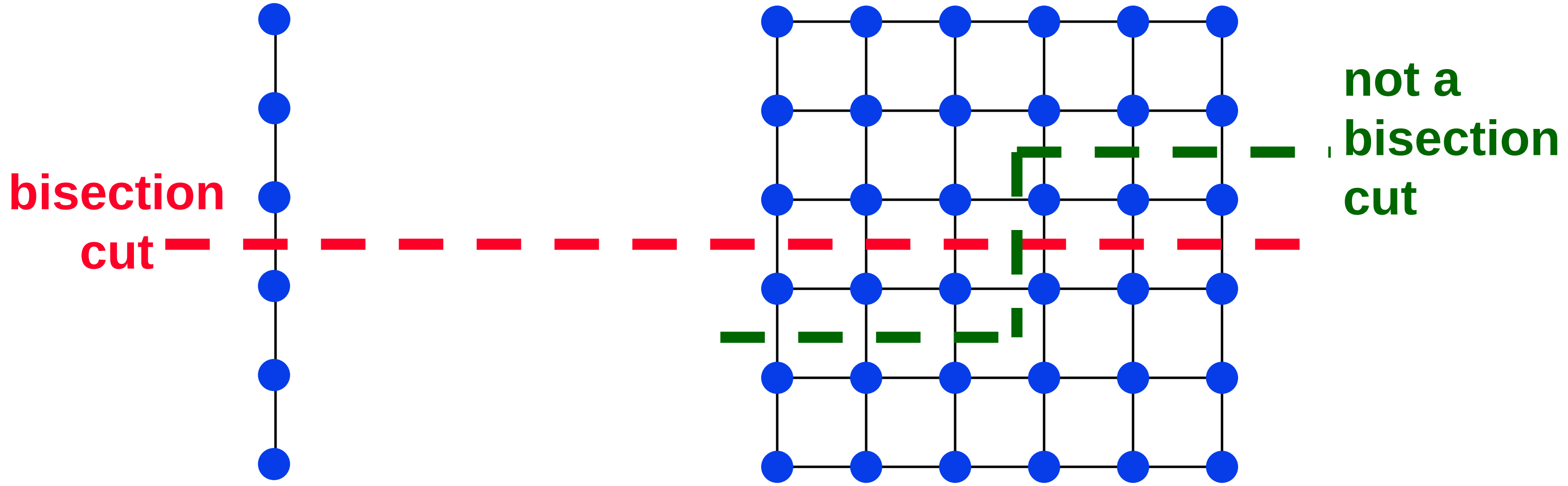
Bandwidth Chart



Data from Mike Welcome, NERSC

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 bw= link bw ***bisection bw = sqrt(n) * link bw***

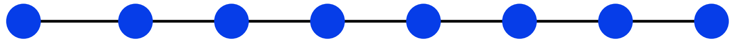
- Bisection bandwidth is important for algorithms in which all processors need to communicate with all others

Network Topology

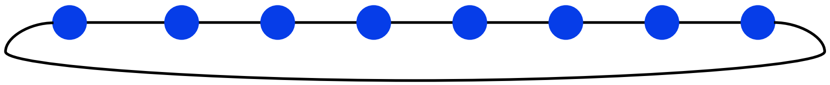
- In the past, there was considerable research in network topology and in mapping algorithms to topology.
  - Key cost to be minimized: number of “hops” between nodes (e.g. “store and forward”)
  - Modern networks hide hop cost (i.e., “wormhole routing”), so topology is no longer a major factor in algorithm performance.
- Example: On IBM SP system, hardware latency varies from 0.5 usec to 1.5 usec, but user-level message passing latency is roughly 36 usec.
- Need some background in network topology
  - Algorithms may have a communication topology
  - Topology affects bisection bandwidth.

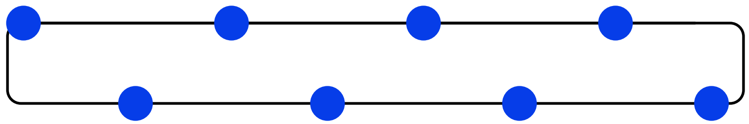
Linear and Ring Topologies

- Linear array



  - Diameter = n-1; average distance ~n/3.
  - Bisection bandwidth = 1 (in units of link bandwidth).
- Torus or Ring



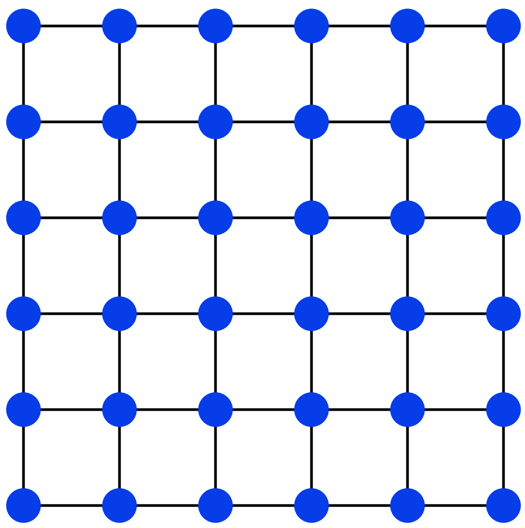


  - Diameter = n/2; average distance ~ n/4.
  - Bisection bandwidth = 2.
  - Natural for algorithms that work with 1D arrays.

Meshes and Tori

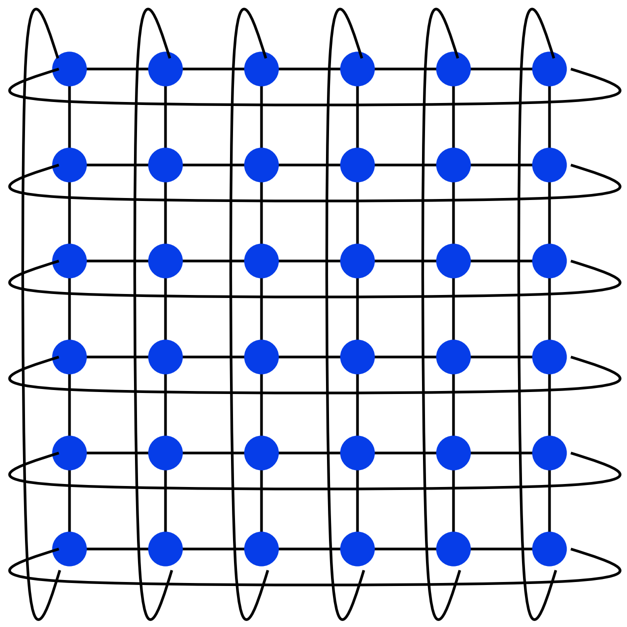
Two dimensional mesh

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



Two dimensional torus

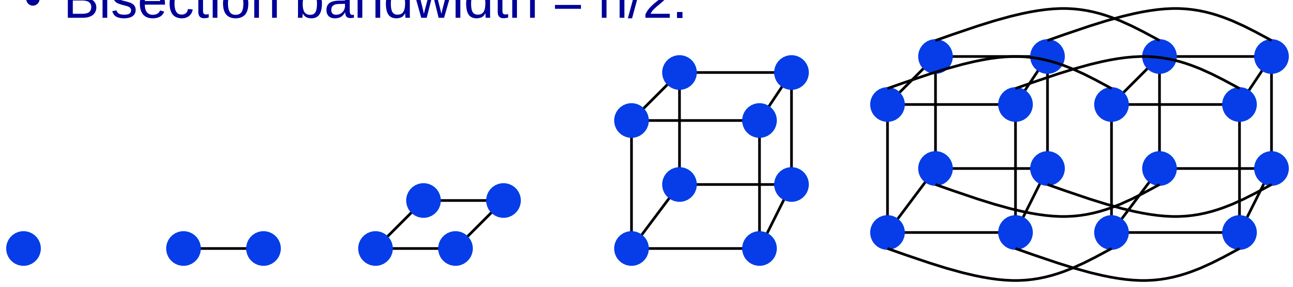
- **Diameter = sqrt(n)**
- **Bisection bandwidth = 2* sqrt(n)**



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

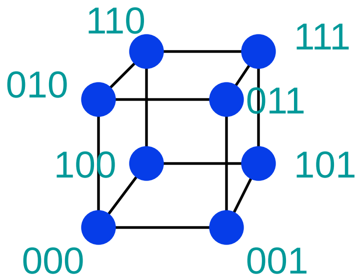
Hypercubes

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



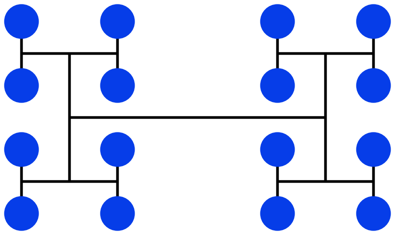
- 0d 1d 2d 3d 4d

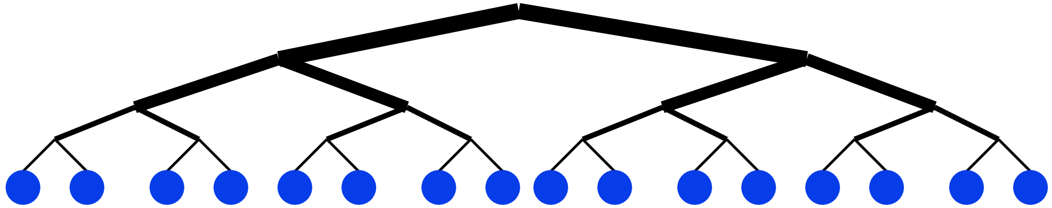
- Popular in early machines (Intel iPSC, NCUBE).
  - Lots of clever algorithms.
  - See 1996 online CS267 notes.
- Greycode addressing:
  - Each node connected to d others with 1 bit different.

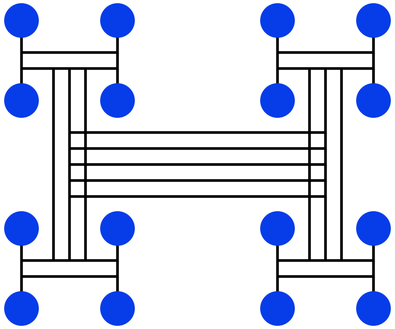


Trees

- 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.



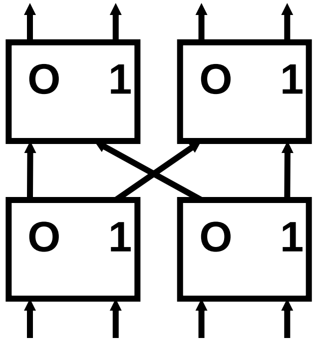




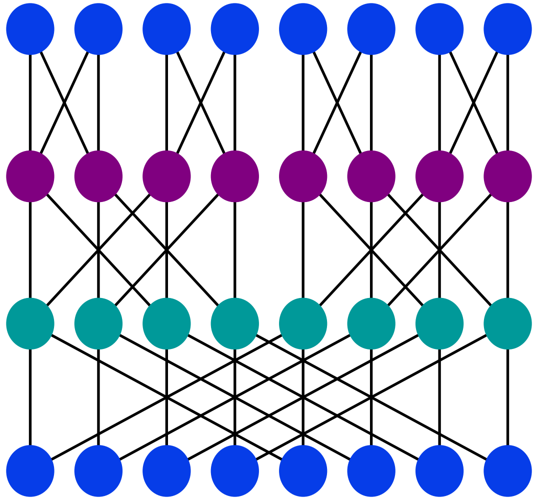
Butterflies

- 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

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

Performance Models

Latency and Bandwidth Model

- Time to send message of length n is roughly

Time = latency + n*cost_per_word
= latency + n/bandwidth

- Topology is assumed irrelevant.
- Often called "$\alpha-\beta$ model" and written

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

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

$$\alpha + n*\beta \ll n*(\alpha + 1*\beta)$$

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

Alpha-Beta Parameters on Current Machines

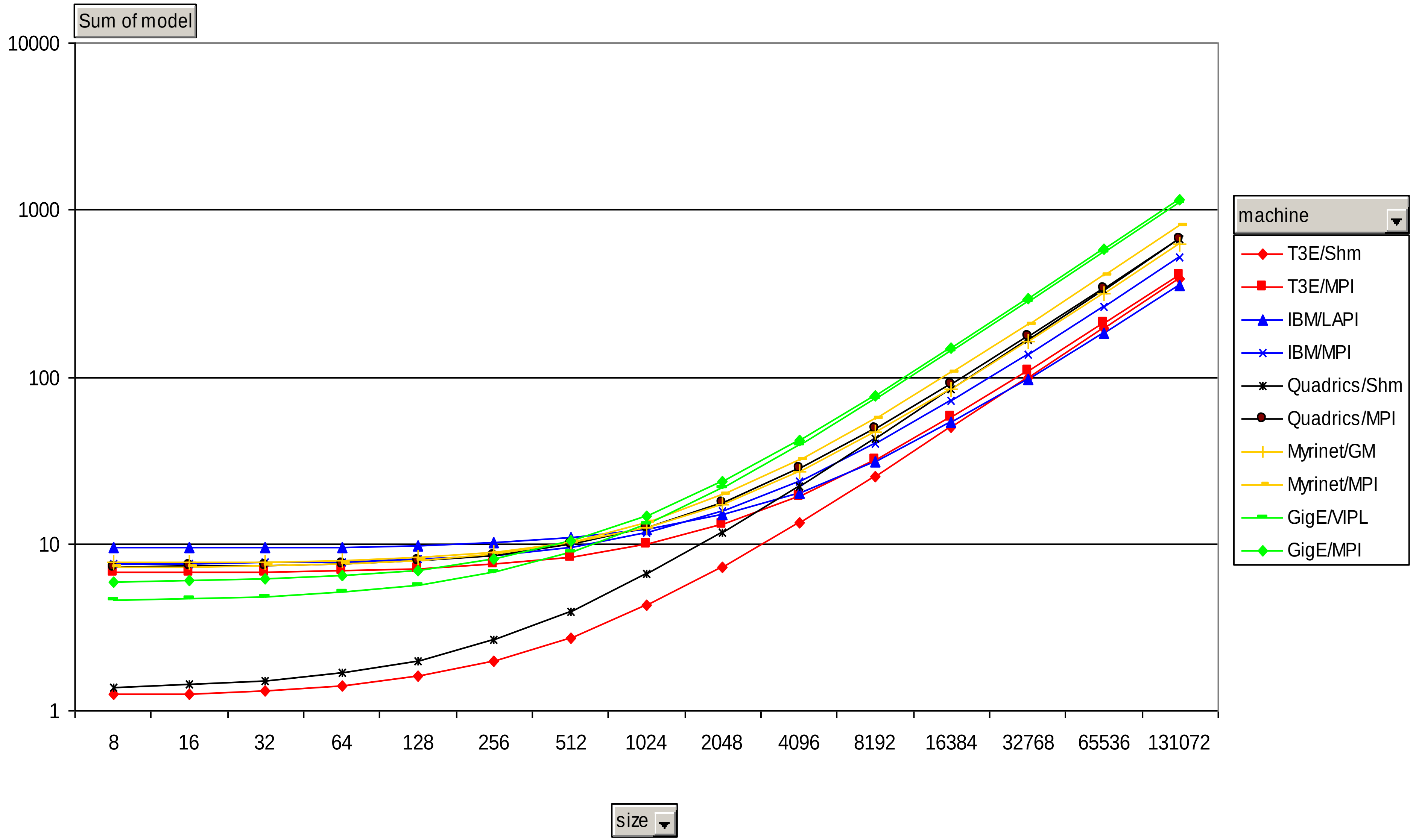
- These numbers were obtained empirically

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

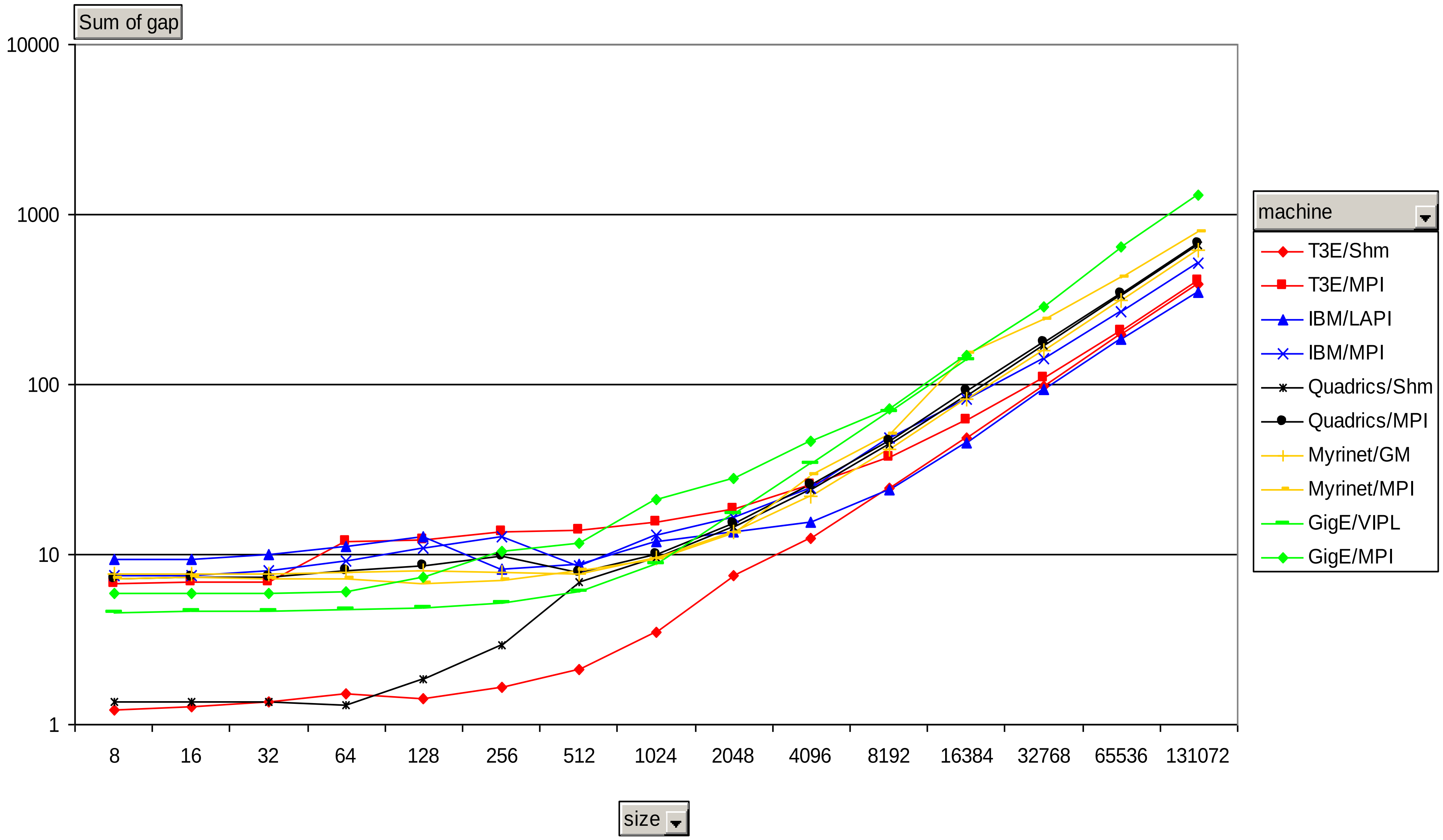
α is latency in usecs
β is BW in usecs per Byte

How well does the model
Time = $\alpha + n*\beta$
predict actual performance?

Model Time Varying Message Size & Machines

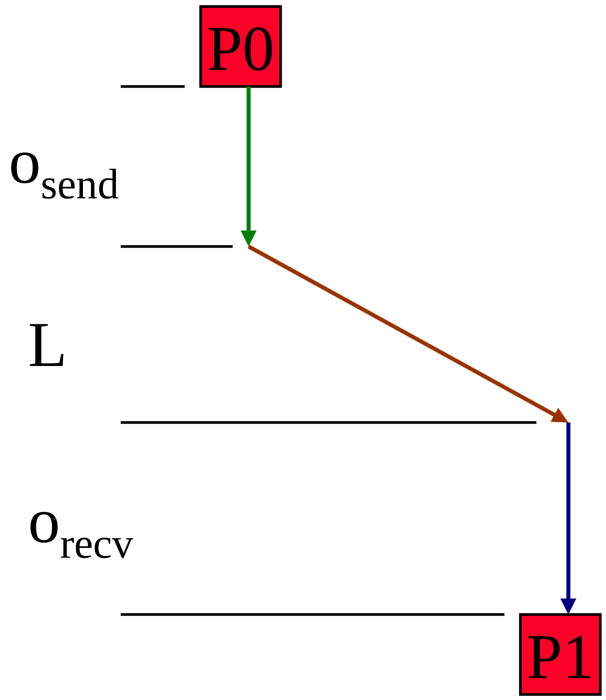


Measured Message Time



LogP Parameters: Overhead & Latency

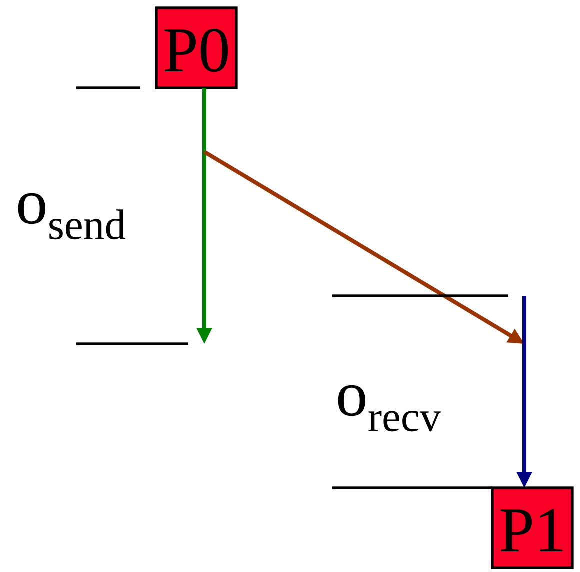
- Non-overlapping overhead



EEL = End-to-End Latency

$$= o_{send} + L + o_{recv}$$

- Send and recv overhead can overlap



$$EEL = f(o_{send}, L, o_{recv})$$

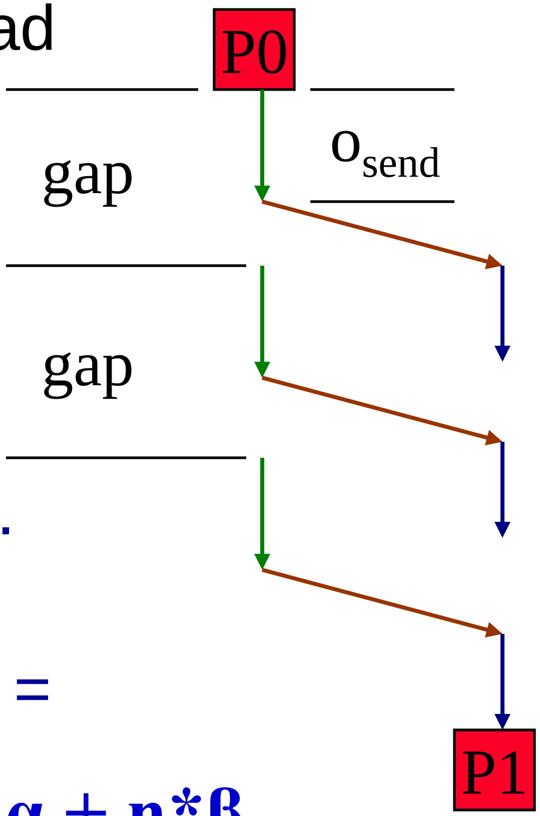
$$\geq \max(o_{send}, L, o_{recv})$$

LogP Parameters: gap

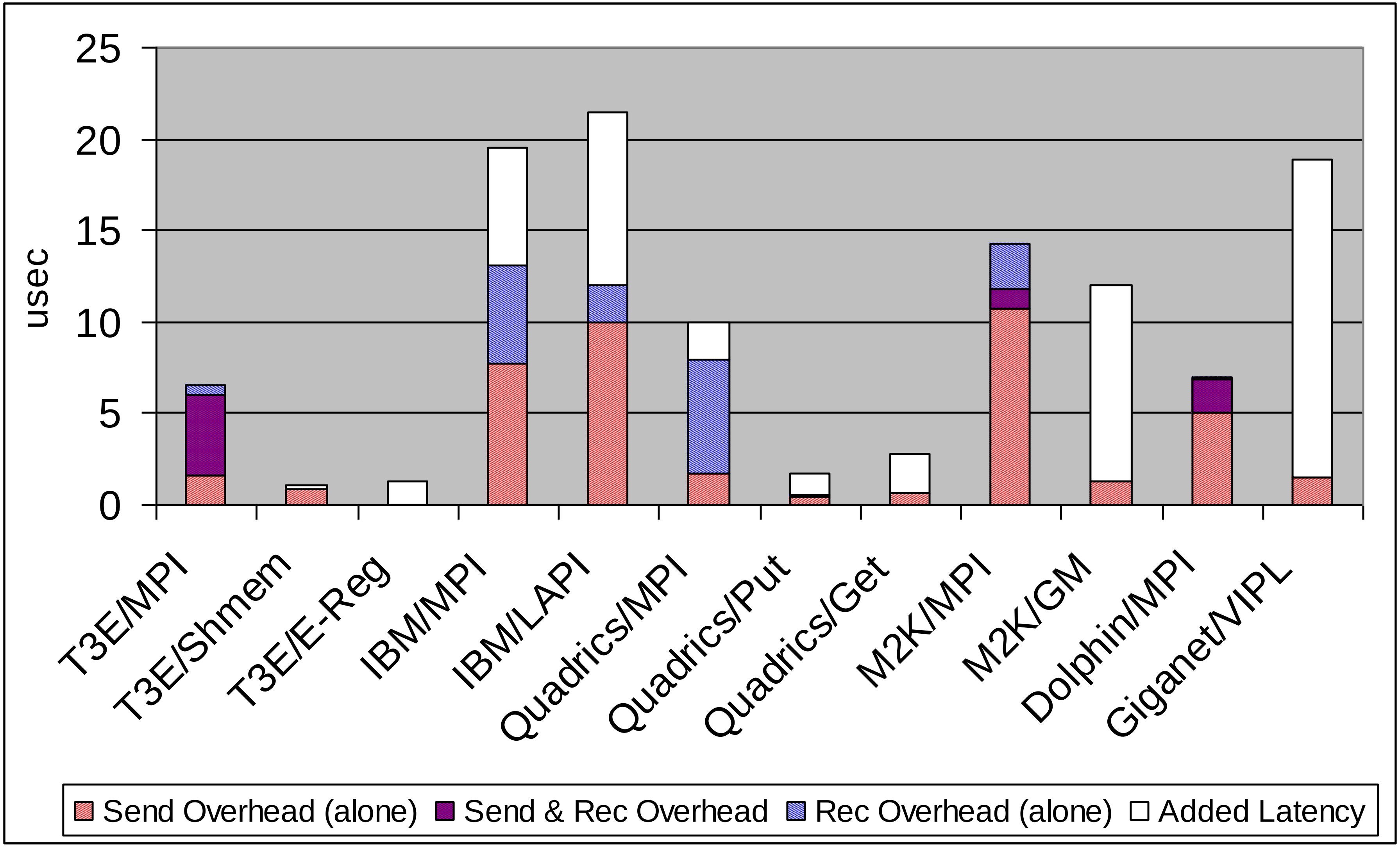
- The Gap is the delay between sending messages
- Gap could be greater than send overhead
  - NIC may be busy finishing the processing of last message and cannot accept a new one.
  - Flow control or backpressure on the network may prevent the NIC from accepting the next message to send.
- No overlap $\Rightarrow$

  time to send n messages (pipelined) =

$$(o_{send} + L + o_{recv} - gap) + n*gap = \alpha + n*\beta$$

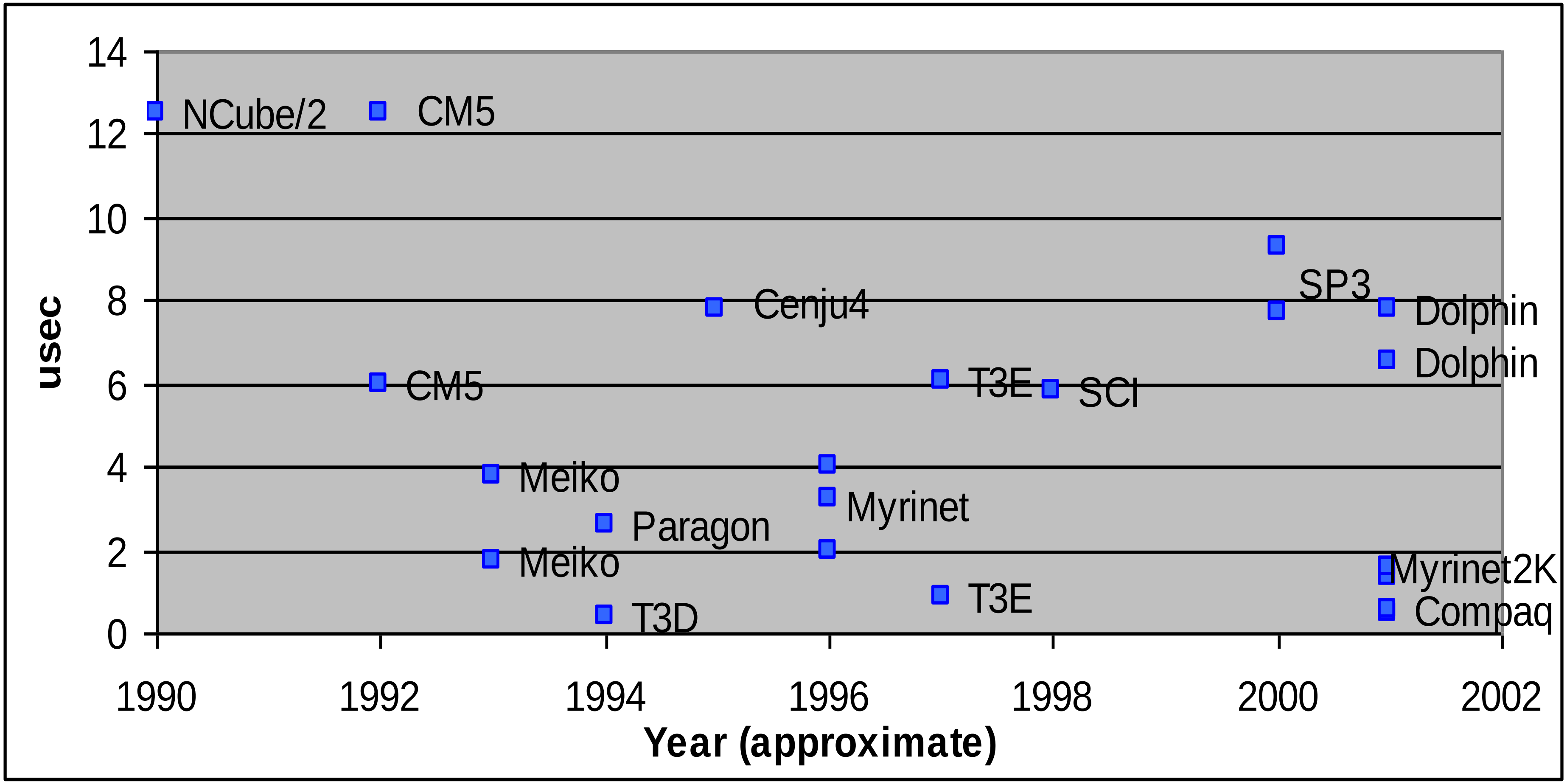


Results: EEL and Overhead



Data from Mike Welcome, NERSC

Send Overhead Over Time



- Overhead has not improved significantly; T3D was best
  - Lack of integration; lack of attention in software

Data from Kathy Yelick, UCB and NERSC

Limitations of the LogP Model

- The LogP model has a fixed cost for each message
  - This is useful in showing how to quickly broadcast a single word
  - Other examples also in the LogP papers
- For larger messages, there is a variation LogGP
  - Two gap parameters, one for small and one for large messages
  - The large message gap is the β in our previous model
- No topology considerations (including no limits for bisection bandwidth)
  - Assumes a fully connected network
  - OK for some algorithms with nearest neighbor communication, but with "all-to-all" communication we need to refine this further
- This is a flat model, i.e., each processor is connected to the network
  - Clusters of multicores are not accurately modeled

Programming Distributed Memory Machines with Message Passing

Slides from

Jonathan Carter (jtcarter@lbl.gov),
Katherine Yelick (yelick@cs.berkeley.edu),
Bill Gropp (wgropp@illinois.edu)

Message Passing Libraries (1)

- 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 (2)

- 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, … 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
- Other information on Web:
  - at http://www.mcs.anl.gov/mpi
  - pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages

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

Hello (C)

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    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();
    return 0;
}
```

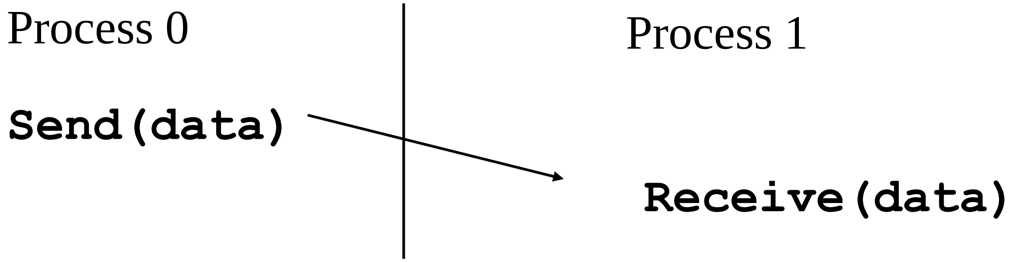
Notes on Hello World

- All MPI programs begin with MPI_Init and end with MPI_Finalize
- MPI_COMM_WORLD is defined by mpi.h (in C) or mpif.h (in Fortran) and designates all processes in the MPI "job"
- Each statement executes independently in each process
  - including the `printf/print` statements
- I/O not part of MPI-1but is in MPI-2
  - print and write to standard output or error not part of either MPI-1 or MPI-2
  - output order is undefined (may be interleaved by character, line, or blocks of characters),
- 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?

Some 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

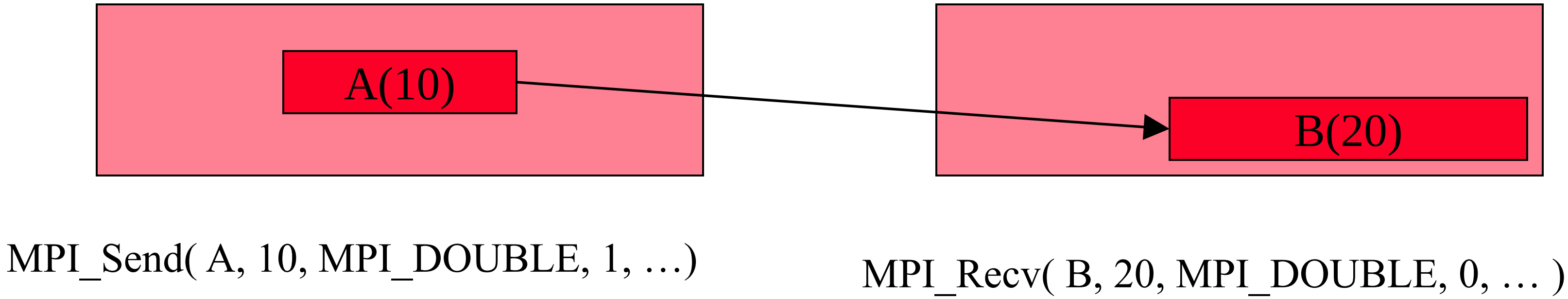
- The data in a message to send or receive is described by a triple (address, count, datatype), where
- An MPI datatype is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
  - a contiguous array of MPI datatypes
  - a strided block of 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

Slide source: Bill Gropp, ANL

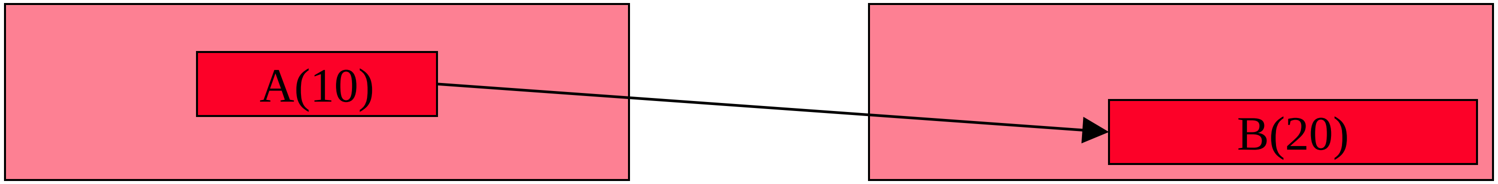
MPI Basic (Blocking) Send



`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 Basic (Blocking) Receive



MPI_Send(A, 10, MPI_DOUBLE, 1, …)

MPI_Recv(B, 20, MPI_DOUBLE, 0, …)

`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 on or **`MPI_ANY_TAG`**
- receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error
- **`status`** contains further information (e.g. size of message)

A Simple MPI 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;
}
```

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 );
```

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

Slide source: Bill Gropp, ANL

MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
    - **MPI_INIT**
    - **MPI_FINALIZE**
    - **MPI_COMM_SIZE**
    - **MPI_COMM_RANK**
    - **MPI_SEND**
    - **MPI_RECV**

Another Approach to Parallelism

- *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…

Collective Operations 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

Slide source: Bill Gropp, ANL

Example: PI in C - 1

```
#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;
```

Example: PI in C - 2

```
    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 / (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;
}
```

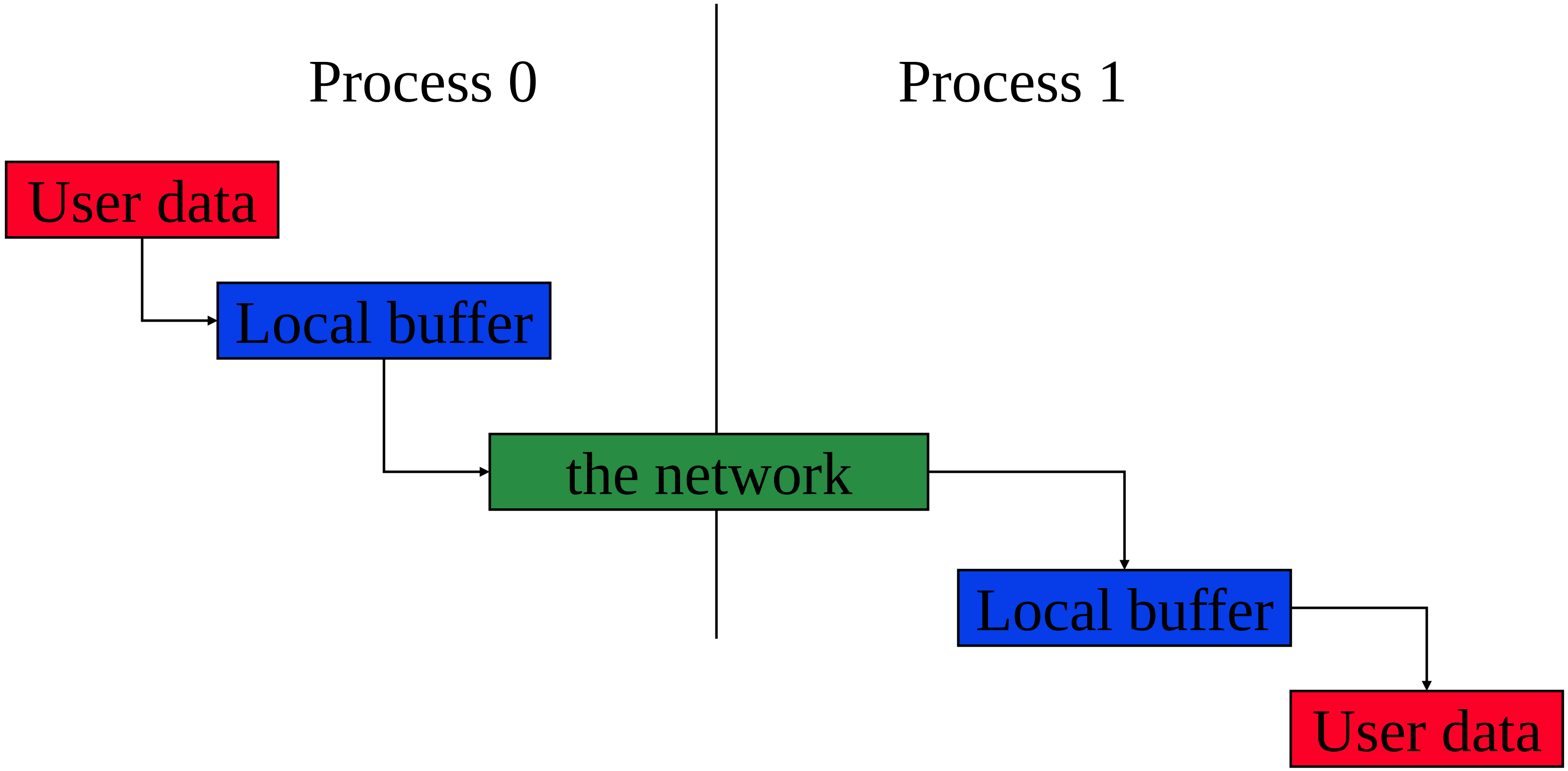

Slide source: Bill Gropp, ANL

More on Message Passing

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

Buffers

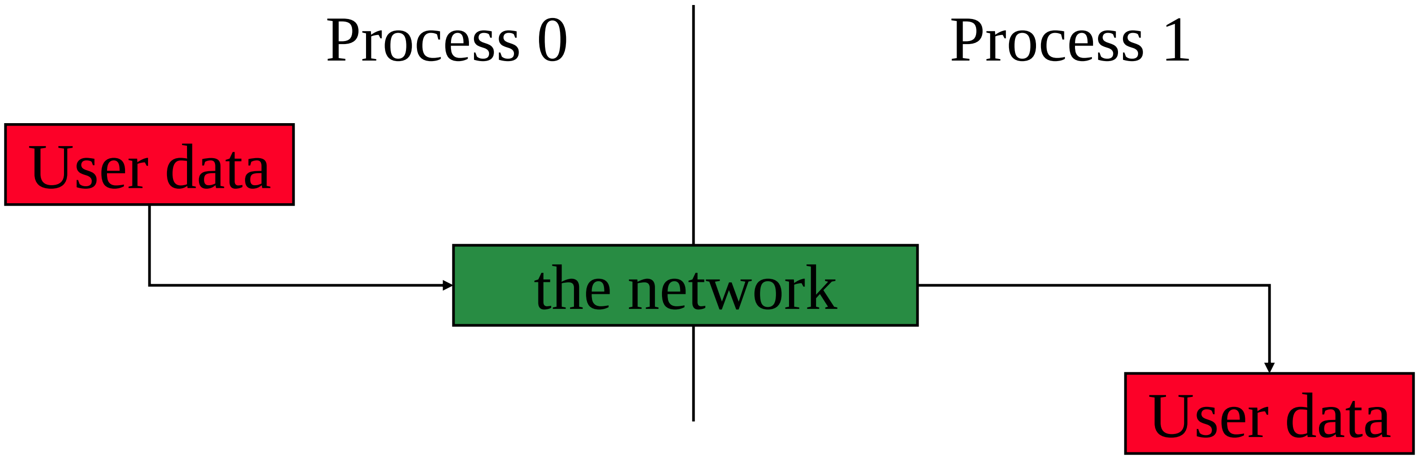
- When you send data, where does it go? One possibility is:



Slide source: Bill Gropp, ANL

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.

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

Slide source: Bill Gropp, ANL

Sources of Deadlocks

- Will there be a deadlock?
- Assume tag/process ID is assigne properly.

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

Slide source: Bill Gropp, ANL

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)`

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;
MPI_Status status;
 MPI_Isend(start, count, datatype,
     dest, tag, comm, &request);
 MPI_Irecv(start, count, datatype,
     dest, tag, comm, &request);
 MPI_Wait(&request, &status);
```

  (each request must be Waited on)

- One can also test without waiting:

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

Communication Modes

- MPI provides multiple *modes* for sending messages:
  - Synchronous mode (**MPI_Ssend**): 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_Issend**, etc.)
- **MPI_Recv** receives messages sent in any mode.
- See www.mpi-forum.org for summary of all flavors of send/receive

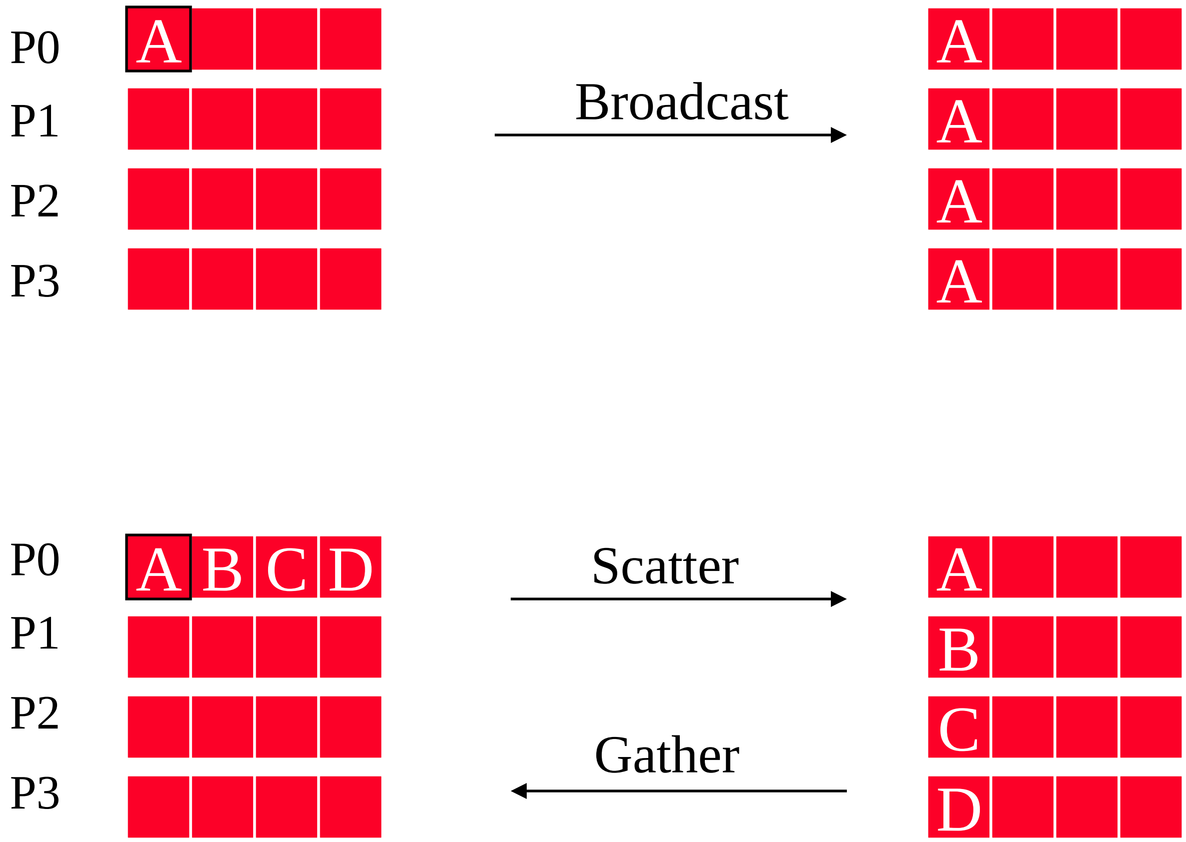
MPI Collective Communication

- Communication and computation is coordinated among a group of processes in a communicator.
- Groups and communicators can be constructed "by hand" or using topology routines.
- Tags are not used; different communicators deliver similar functionality.
- No non-blocking collective operations.
- Three classes of operations: synchronization, data movement, collective computation.

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

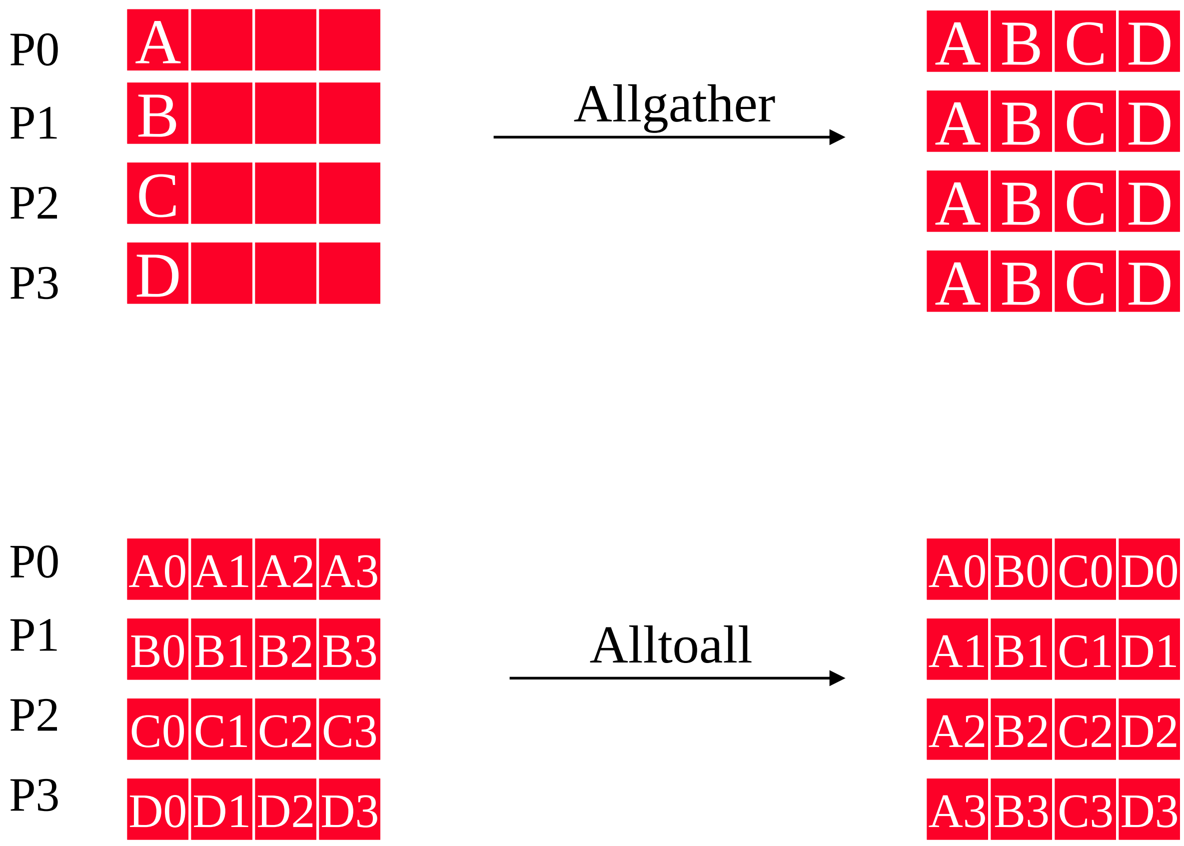
Collective Data Movement



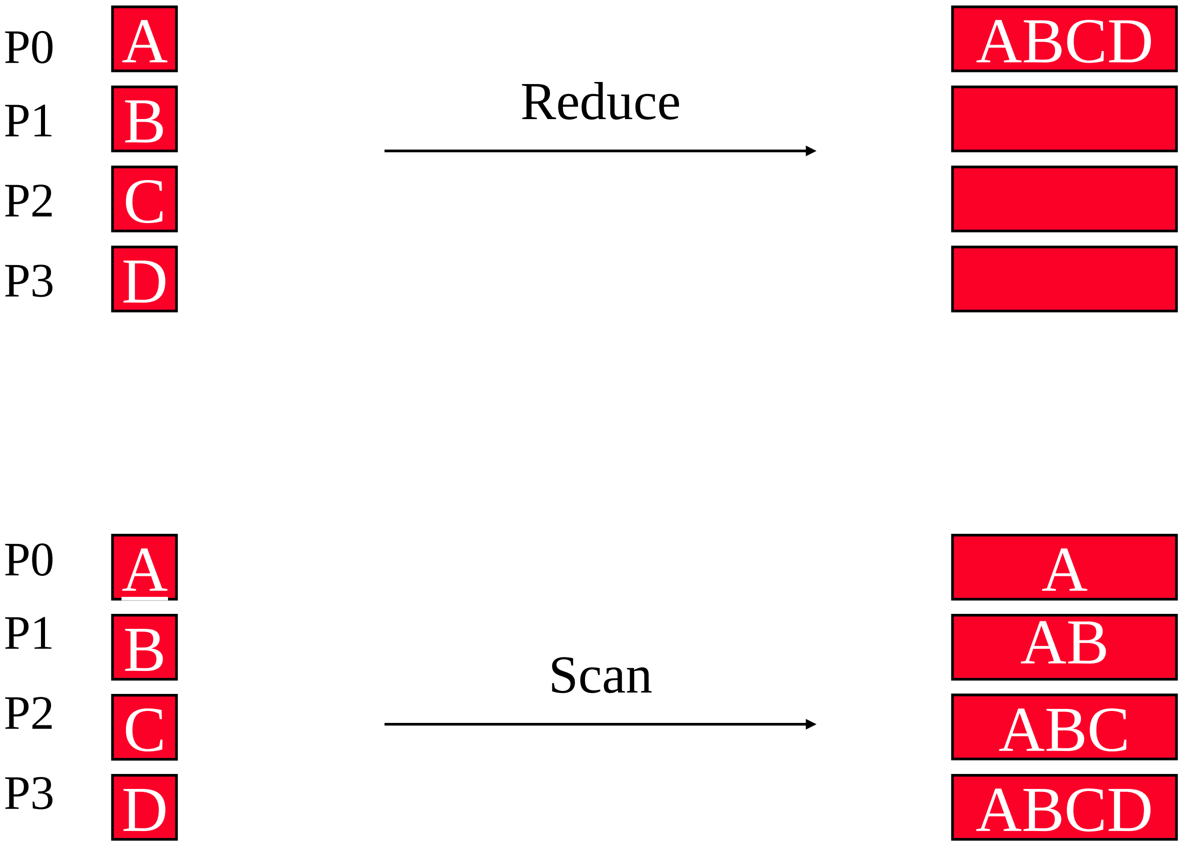
Comments on Broadcast

- All collective operations must be called by *all* processes in the communicator
- MPI_Bcast is called by both the sender (called the root process) and the processes that are to receive the broadcast
  - MPI_Bcast is not a “multi-send”
  - “root” argument is the rank of the sender; this tells MPI which process originates the broadcast and which receive

More Collective Data Movement



Collective Computation



MPI Collective Routines

- Many Routines: **`Allgather, Allgatherv, Allreduce, Alltoall, Alltoallv, Bcast, Gather, Gatherv, Reduce, Reduce_scatter, Scan, Scatter, Scatterv`**
- **`All`** versions deliver results to all participating processes.
- V versions allow the hunks to have variable sizes.
- **`Allreduce`**, **`Reduce`**, **`Reduce_scatter`**, and **`Scan`** take both built-in and user-defined combiner functions.
- MPI-2 adds **`Alltoallw`**, **`Exscan`**, intercommunicator versions of most routines

MPI Built-in Collective Computation Operations

• **MPI_MAX**	Maximum
• **MPI_MIN**	Minimum
• **MPI_PROD**	Product
• **MPI_SUM**	Sum
• **MPI_LAND**	Logical and
• **MPI_LOR**	Logical or
• **MPI_LXOR**	Logical exclusive or
• **MPI_BAND**	Binary and
• **MPI_BOR**	Binary or
• **MPI_BXOR**	Binary exclusive or
• **MPI_MAXLOC**	Maximum and location
• **MPI_MINLOC**	Minimum and location

Not Covered

- Topologies: map a communicator onto, say, a 3D Cartesian processor grid
  - Implementation can provide ideal logical to physical mapping
- Rich set of I/O functions: individual, collective, blocking and non-blocking
  - Collective I/O can lead to many small requests being merged for more efficient I/O
- One-sided communication: puts and gets with various synchronization schemes
  - Implementations not well-optimized and rarely used
  - Redesign of interface is underway
- Task creation and destruction: change number of tasks during a run
  - Few implementations available

Backup Slides

Implementing Synchronous Message Passing

- Send operations complete after matching receive and source data has been sent.
- Receive operations complete after data transfer is complete from matching send.

1) Initiate send

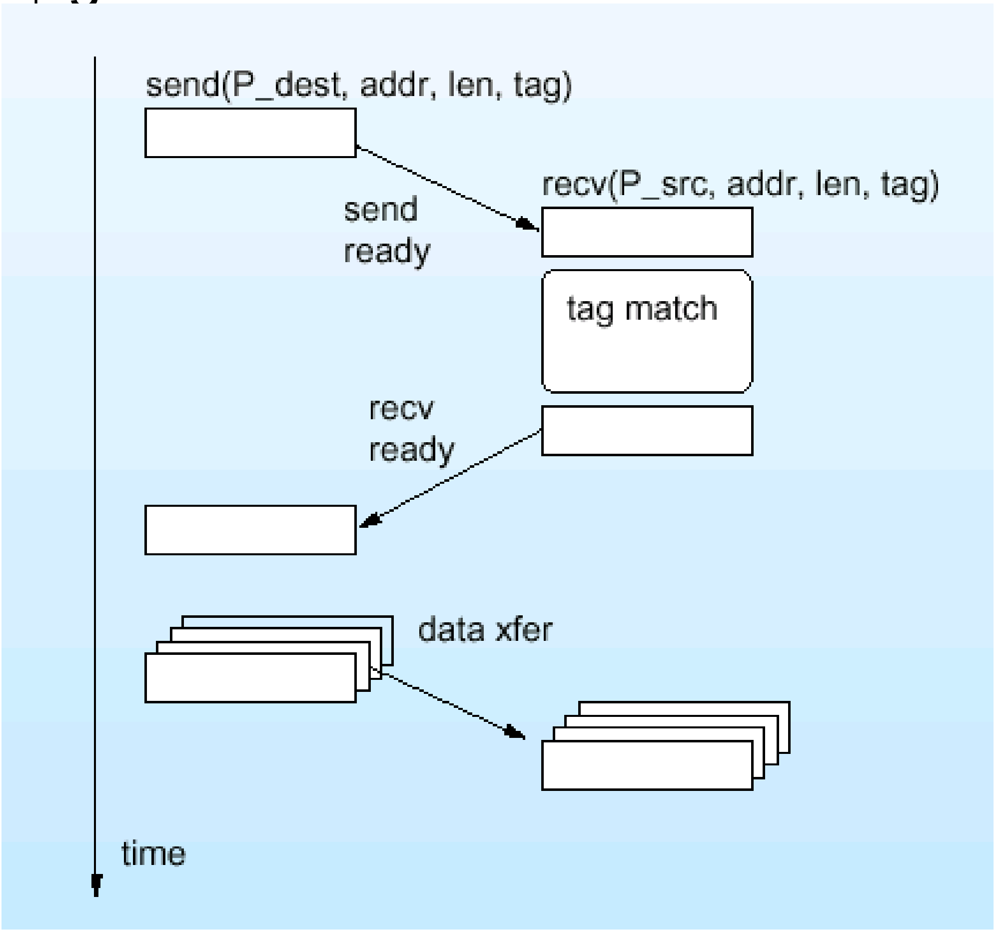
2) Address translation on P_{dest}

3) Send-Ready Request

4) Remote check for posted receive

5) Reply transaction

6) Bulk data transfer

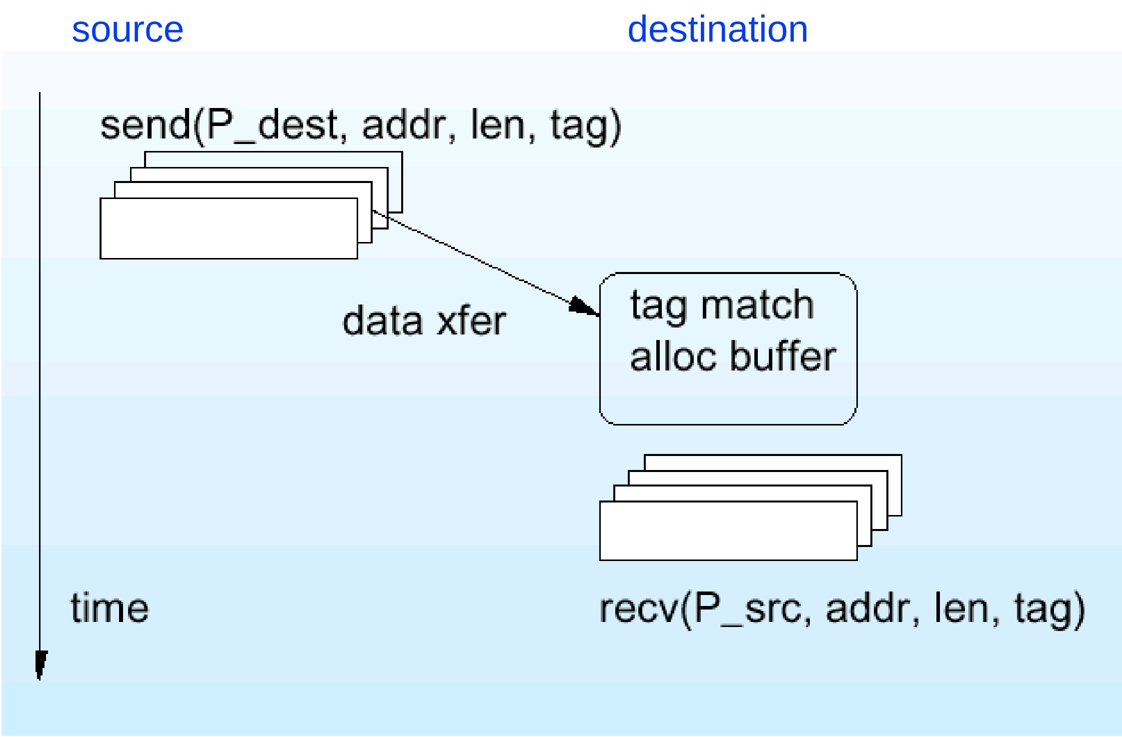


Implementing Asynchronous Message Passing

- Optimistic single-phase protocol assumes the destination can buffer data on demand.

1) Initiate send
2) Address translation on P_{dest}
3) Send Data Request

4) Remote check for posted receive
5) Allocate buffer (if check failed)
6) Bulk data transfer



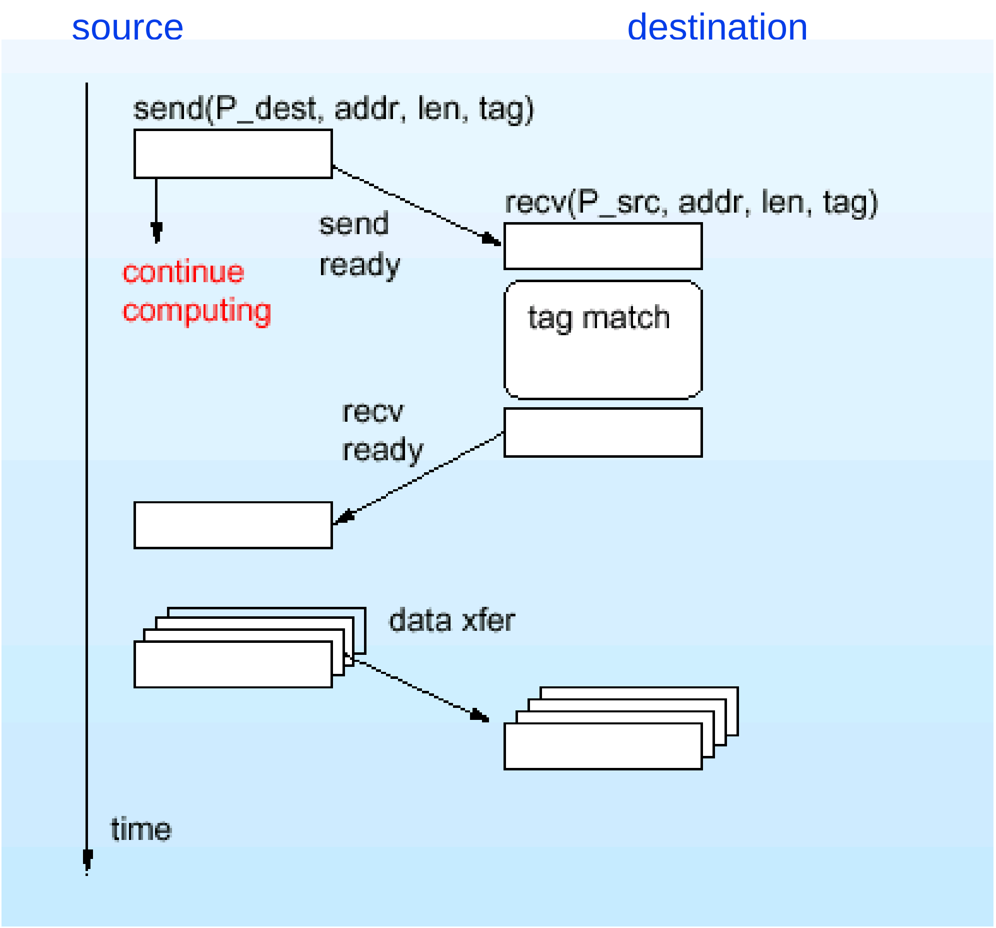
Safe Asynchronous Message Passing

- Use 3-phase protocol
- Buffer on sending side
- Variations on send completion
  - wait until data copied from user to system buffer
  - don't wait -- let the user beware of modifying data

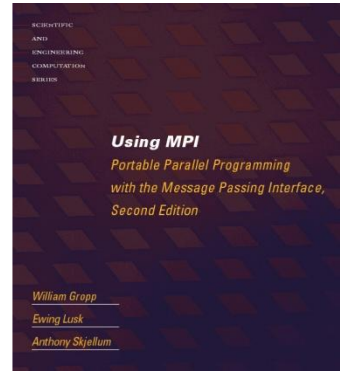
1) Initiate send
2) Address translation on P_{dest}
3) Send-Ready Request

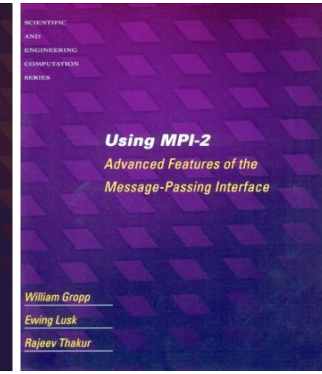
4) Remote check for posted receive
   record send-rdy
5) Reply transaction

6) Bulk data transfer



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.