

CS 267

Lecture 9: Distributed Memory Machines and Programming

Aydin Buluc

<https://sites.google.com/lbl.gov/cs267-spr2021/>

02/16/2021

1

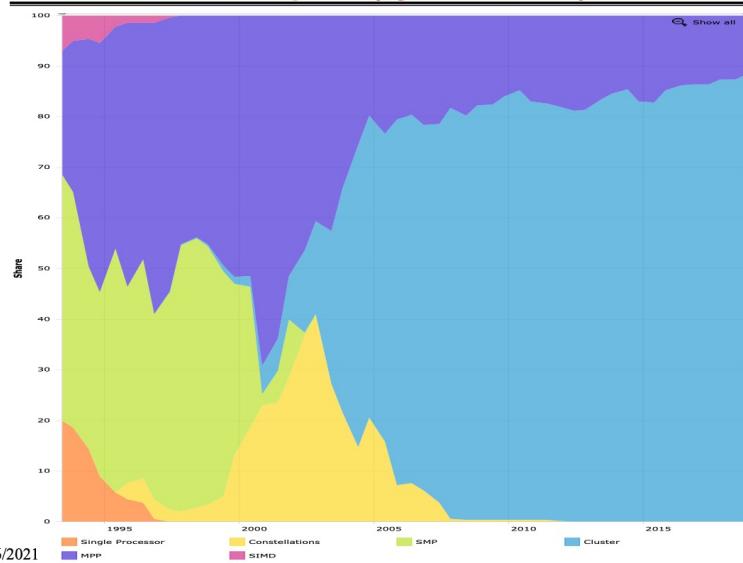
Outline

- **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

02/16/2021

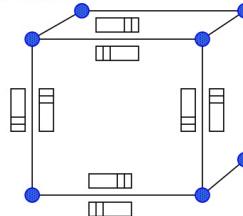
2

Architectures in Top 500 (systems share)



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**



02/16/2021

4

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

02/16/2021

5

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, 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.

02/16/2021

6

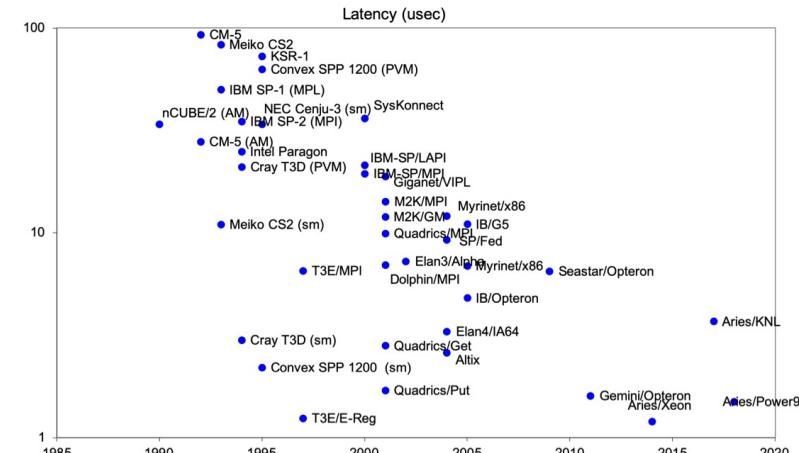
Performance Properties of a Network: Latency

- **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**

02/16/2021

7

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

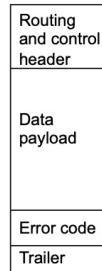


- Latency has not improved significantly, unlike Moore's Law

8

Performance Properties of a Network: Bandwidth

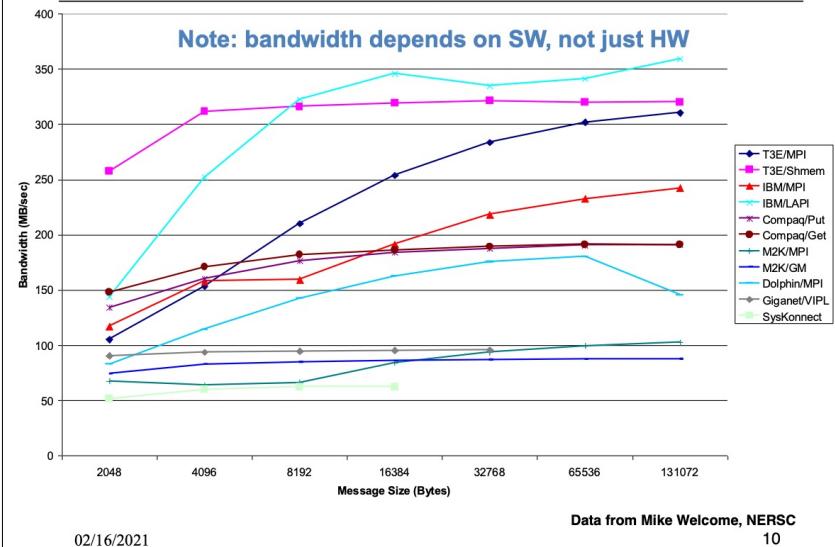
- The **bandwidth** of a link = # wires / time-per-bit
- Bandwidth typically in Gigabytes/sec (GB/s), i.e., $8 \cdot 2^{20}$ bits per second
- Effective bandwidth** is usually lower than physical link bandwidth due to packet overhead.
- Bandwidth is important for applications with mostly large messages



02/16/2021

9

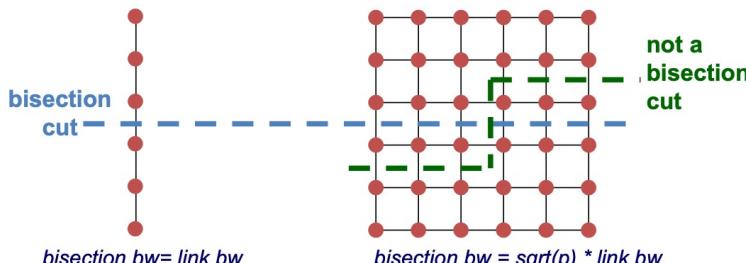
Bandwidth Chart



10

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

02/16/2021

11

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.

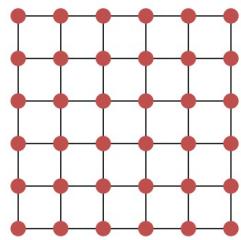
02/16/2021

12

Meshes and Tori – used in Hopper

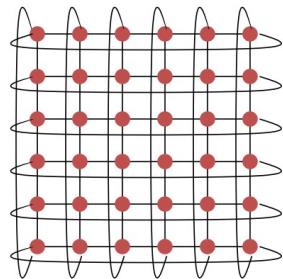
Two dimensional mesh

- 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 Hopper@NERSC) uses 3D Torus
- Natural for algorithms that work with 2D and/or 3D arrays (matmul)

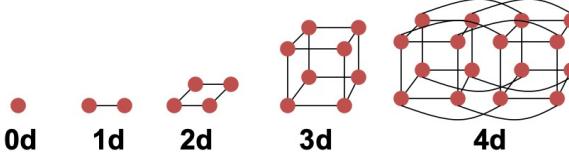
02/16/2021

13

Hypercubes

- Number of nodes $n = 2^d$ for dimension d .

- Diameter = d .
- Bisection bandwidth = $n/2$.



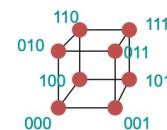
0d 1d 2d 3d 4d

- 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.

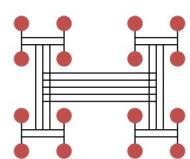
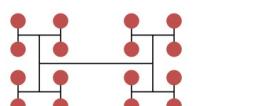


02/16/2021

14

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.

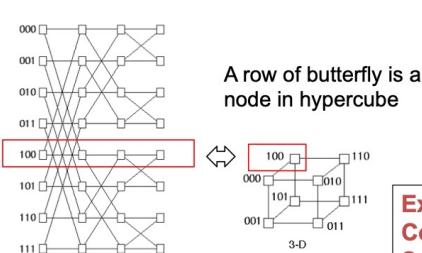


02/16/2021

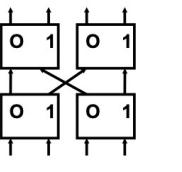
15

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



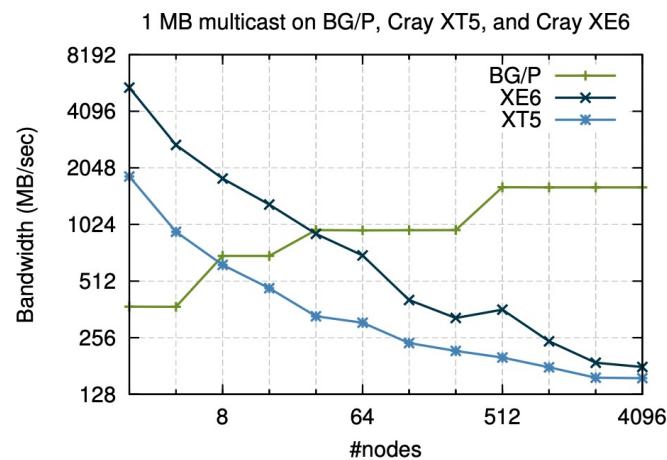
butterfly switch

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

02/16/2021

16

Does Topology Matter?



See EECS Tech Report *UCB/EECS-2011-92*, August 2011

02/16/2021

17

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

02/16/2021

18

Dragonflies – used in Edison and Cori

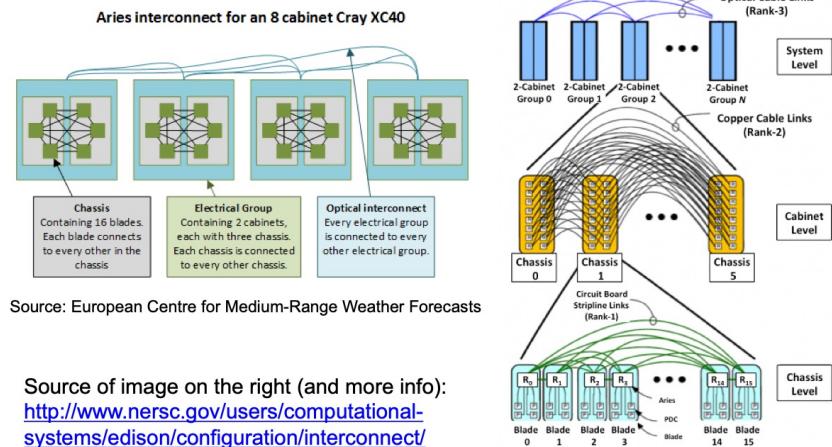
- **Motivation:** Exploit gap in cost and performance between optical interconnects (which go between cabinets in a machine room) and electrical networks (inside cabinet)
 - Optical (fiber) more expensive but higher bandwidth when long
 - Electrical (copper) networks cheaper, faster when short
- **Combine 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

"Technology-Driven, Highly-Scalable Dragonfly Topology," ISCA 2008

02/16/2021

19

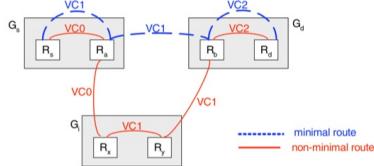
Dragonfly in practice



02/16/2021

20

Why randomized routing?



- Minimal routing:**
1. If $G_s \neq G_d$ and R_s has no connection to G_d , route within G_s from R_s to R_a , which has a global channel to G_d .
 2. If $G_s \neq G_d$, traverse the global channel from R_a to router R_b in G_d .
 3. If $R_b \neq R_d$, route from R_b to R_d .

Minimal routing works well when things are load balanced, potentially catastrophic in adversarial traffic patterns.

Randomization idea: For each packet sourced at router $R_s \in G_s$ and addressed to a router in another group $R_d \in G_d$, first route it to an intermediate group G_i .

- This requires at most two group-level link traversals
- And at most 5 total link traversals

Valiant, Leslie G. "A scheme for fast parallel communication." *SIAM journal on computing* 11.2 (1982): 350-361.

21

Performance Models

02/16/2021

22

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 (see CS273).
 - Parallel algorithm design strategy: first do a PRAM algorithm, then worry about memory/communication time (sometimes works)
- Slightly more realistic versions exist
 - E.g., Concurrent Read Exclusive Write (CREW) PRAM.
 - Still missing the memory hierarchy

02/16/2021

23

Latency and Bandwidth Model

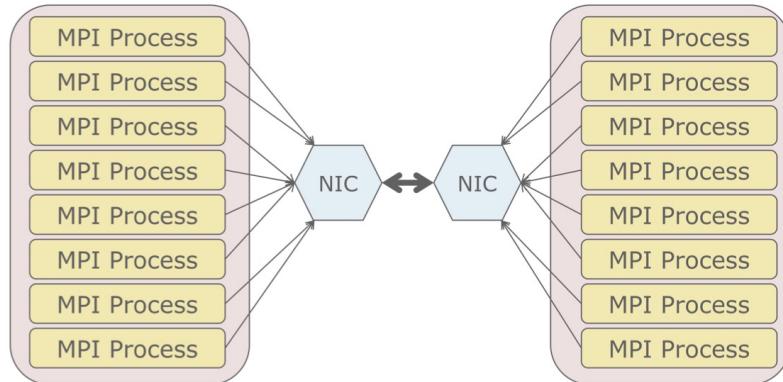
- Time to send message of length n is roughly
$$\text{Time} = \text{latency} + n * \text{cost_per_word}$$
$$= \text{latency} + n / \text{bandwidth}$$
- Topology is assumed irrelevant.
- Often called " α - β model" and written
$$\text{Time} = \alpha + n * \beta$$
- Usually $\alpha \gg \beta \gg \text{time per flop}$.
 - One long message is cheaper than many short ones.
 - Can do hundreds or thousands of flops for cost of one message.
$$\alpha + n * \beta \ll n * (\alpha + 1 * \beta)$$
- Lesson: Need large computation-to-communication ratio to be efficient.
- LogP – more detailed model (Latency/overhead/gap/Proc.)

02/16/2021

24

Latency and Bandwidth Model in 2010 and beyond+

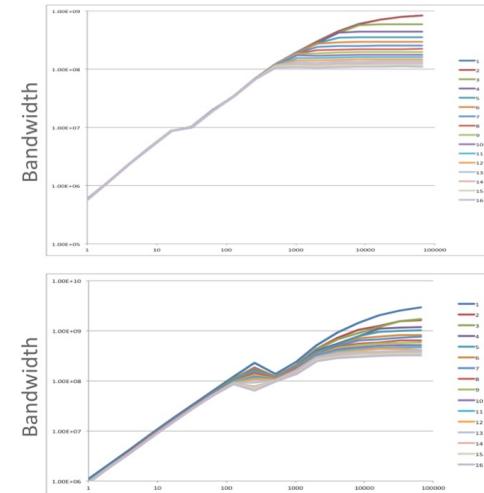
Processors are multi-core and many nodes are multi-chip



02/16/2021

Figure source: Balaji, Gropp, Thakur 25

NIC bandwidth bottleneck



02/16/2021

Figure source: Balaji, Gropp, Thakur 26

Programming Distributed Memory Machines with Message Passing

Slides by

Aydin Buluc, Jonathan Carter, Jim Demmel,
Bill Gropp, Kathy Yelick

02/16/2021

27

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.

02/16/2021

28

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, 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?

02/16/2021

29

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

02/16/2021

30

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

02/16/2021

31

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.

02/16/2021



32

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

02/16/2021

33

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;
}
```

02/16/2021

34

Notes on Hello World

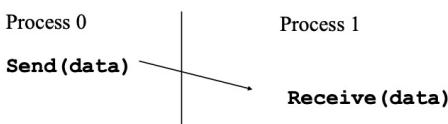
- All MPI programs begin with MPI_Init and end with MPI_Finalize
- MPI_COMM_WORLD is defined by mpi.h (in C) or mpif.h (in Fortran) and designates all processes in the MPI “job”
- Each statement executes independently in each process
 - including the printf/print statements
- The MPI-1 Standard does not specify how to run an MPI program, but many implementations provide `mpirun -np 4 a.out`

02/16/2021

35

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?

02/16/2021

36

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`

02/16/2021

37

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

02/16/2021

38

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

02/16/2021

39

MPI Basic (Blocking) Send



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

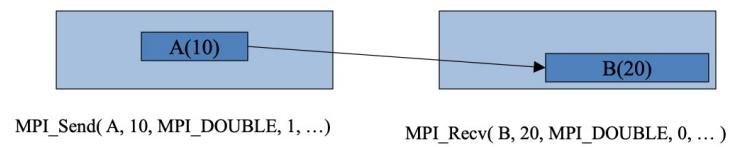
`MPI_Recv(B, 20, MPI_DOUBLE, 0, ...)`

`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.

40

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 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)
```

41

## A Simple MPI Program

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
 int rank, buf;
 MPI_Status status;
 MPI_Init(&argc, &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;
}
```

02/16/2021

42

## Retrieving Further Information

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

```
int recv_tag, recv_from, recv_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status)
recv_tag = status.MPI_TAG;
recv_from = status.MPI_SOURCE;
MPI_Get_count(&status, datatype, &recv_count);
```

02/16/2021

43

## 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

02/16/2021

44

## But is that small subset the practical usage?

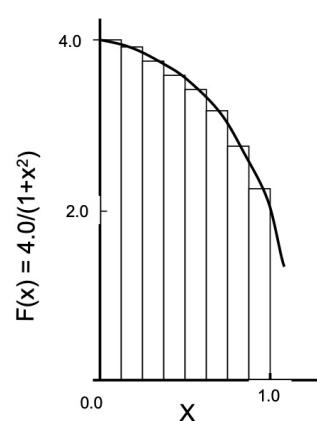
- 35 What aspects of the MPI standard do you use in your application in its current form? \* (multiple)  
 36 What aspects of the MPI standard appear in performance-critical sections of your current application? \* (multiple)  
 37 What aspects of the MPI standard do you anticipate using in the "exascale"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

| Responses                                      | Q35: Current Usage |     |         | Q37: Exascale Usage |     |         | Q38: 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%      |

02/16/2021

45

## PI redux: Numerical integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^N F(x_i) \Delta x \approx \pi$$

Where each rectangle has width  $\Delta x$  and height  $F(x_i)$  at the middle of interval  $i$ .

02/16/2021

47

## But is that small subset the practical usage?

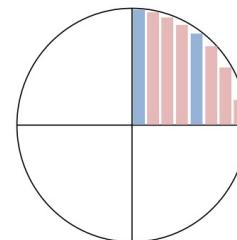
| No. | Question and Responses                                                                                                                                                                 | AD  | ST  | Overall |
|-----|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|-----|---------|
| 38  | What is the dominant communication in your application? Check all that apply, recognizing that many applications have different communication patterns in different phases. (multiple) |     |     |         |
|     | 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... (multiple)                                                                                            |     |     |         |
|     | 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... (multiple)                                                                                                |     |     |         |
|     | To overlap communication with computation?                                                                                                                                             | 71% | 46% | 59%     |
|     | To allow asynchronous progress?                                                                                                                                                        | 46% | 29% | 38%     |
|     | To allow event-based programming?                                                                                                                                                      | 25% | 14% | 20%     |

<https://info.ornl.gov/sites/publications/Files/Pub108588.pdf>

02/16/2021

46

## Example: Calculating Pi



E.g., in a 4-process run, each process gets every 4<sup>th</sup> interval. Process 0 slices are in red.

- Simple program written in a data parallel style in MPI
  - E.g., for a reduction (recall “data parallelism” lecture), 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)

02/16/2021

48

## Example: PI in C – 1/2

```
#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;
 }
```

02/16/2021

49

## Example: PI in C – 2/2

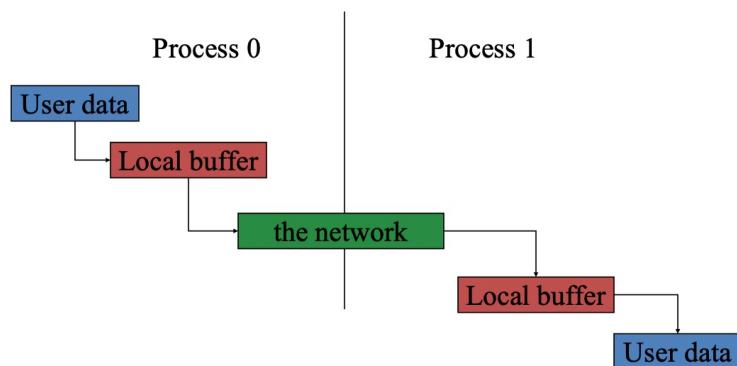
```
h = 1.0 / (double) n;
sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs) {
 x = h * ((double)i - 0.5);
 sum += 4.0 * sqrt(1.0 - x*x);
}
mypi = h * sum;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
 MPI_COMM_WORLD);
if (myid == 0)
 printf("pi is approximately %.16f, Error is %.16f\n",
 pi, fabs(pi - PI25DT));
MPI_Finalize();
return 0;
}
```

02/16/2021

50

## Buffers

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

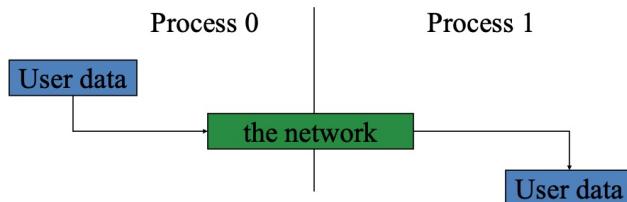


02/16/2021

Slide source: Bill Gropp, ANL 51

## 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.

02/16/2021

Slide source: Bill Gropp, ANL 52

## 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.

02/16/2021

53

## 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      |
|----------------|----------------|
| <b>Send(1)</b> | <b>Send(0)</b> |
| <b>Recv(1)</b> | <b>Recv(0)</b> |

- 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

02/16/2021

54

## Some Solutions to the “unsafe” Problem

- Order the operations more carefully:

| Process 0      | Process 1      |
|----------------|----------------|
| <b>Send(1)</b> | <b>Recv(0)</b> |
| <b>Recv(1)</b> | <b>Send(0)</b> |

- Supply receive buffer at same time as send:

| Process 0          | Process 1          |
|--------------------|--------------------|
| <b>Sendrecv(1)</b> | <b>Sendrecv(0)</b> |

02/16/2021

55

## More Solutions to the “unsafe” Problem

- Supply own space as buffer for send

| Process 0       | Process 1       |
|-----------------|-----------------|
| <b>Bsend(1)</b> | <b>Bsend(0)</b> |
| <b>Recv(1)</b>  | <b>Recv(0)</b>  |

- Use non-blocking operations:

| Process 0       | Process 1       |
|-----------------|-----------------|
| <b>Isend(1)</b> | <b>Isend(0)</b> |
| <b>Irecv(1)</b> | <b>Irecv(0)</b> |
| <b>Waitall</b>  | <b>Waitall</b>  |

02/16/2021

56

## 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);
```

- Accessing the data buffer without waiting is undefined

02/16/2021

57

## 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.

02/16/2021

58

## 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](http://www mpi-forum.org) for summary of all flavors of send/receive

02/16/2021

59