Distributed Memory Machines and Programming

Lecture 7

James Demmel

www.cs.berkeley.edu/~demmel/cs267_Spr14

Slides from Kathy Yelick

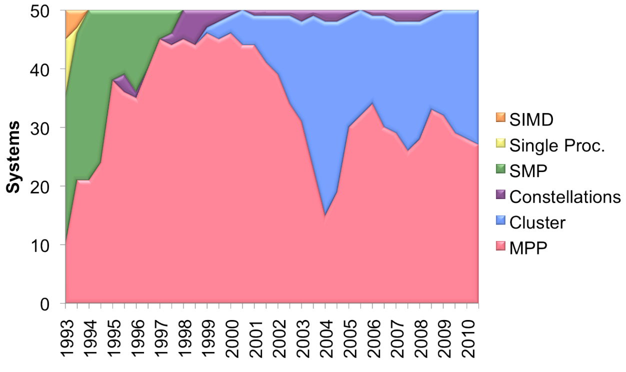
Recap of Lecture 6

- 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 (see CS258, CS252).
  - 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.

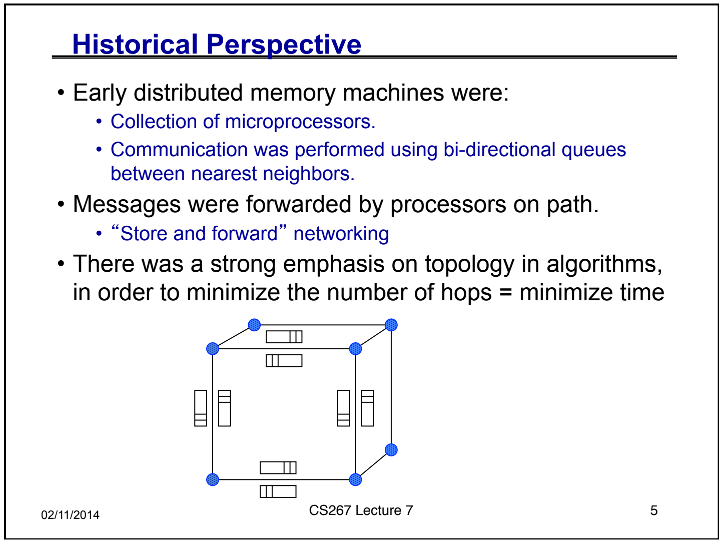
Outline

- 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

Architectures (TOP50)

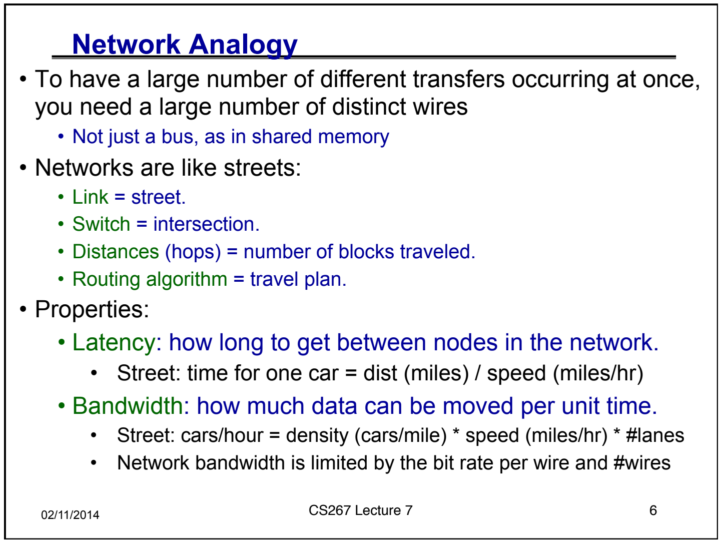


Top500 similar: 100% Cluster + MPP since 2009



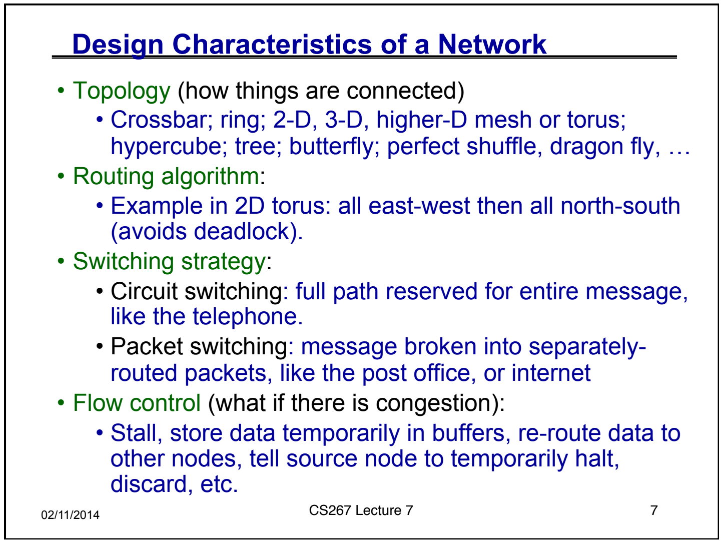
Historical Perspective

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



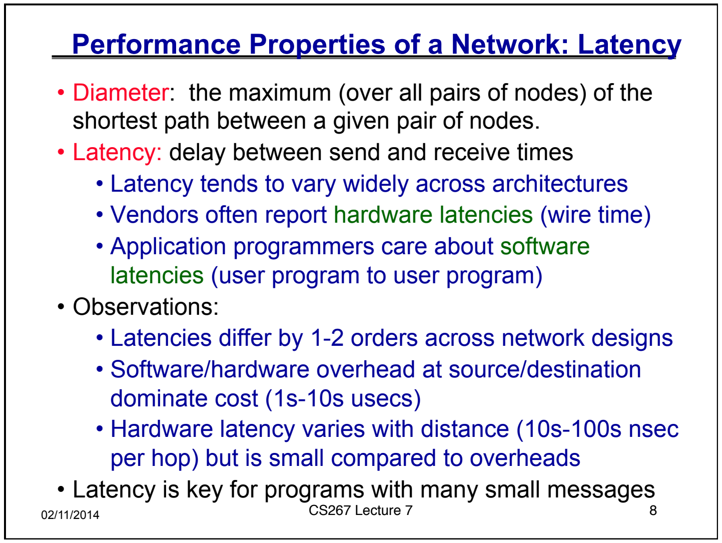
Network Analogy

- To have a large number of different transfers occurring at once, you need a large number of distinct wires
  - Not just a bus, as in shared memory
- Networks are like streets:
  - Link = street.
  - Switch = intersection.
  - Distances (hops) = number of blocks traveled.
  - Routing algorithm = travel plan.
- Properties:
  - Latency: how long to get between nodes in the network.
    - Street: time for one car = dist (miles) / speed (miles/hr)
  - Bandwidth: how much data can be moved per unit time.
    - Street: cars/hour = density (cars/mile) * speed (miles/hr) * #lanes
    - Network bandwidth is limited by the bit rate per wire and #wires



Design Characteristics of a Network

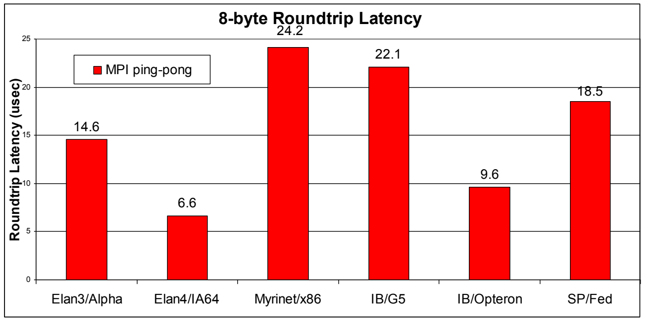
- Topology (how things are connected)
  - Crossbar; ring; 2-D, 3-D, higher-D mesh or torus; hypercube; tree; butterfly; perfect shuffle, dragon fly, …
- Routing algorithm:
  - Example in 2D torus: all east-west then all north-south (avoids deadlock).
- Switching strategy:
  - Circuit switching: full 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: Latency

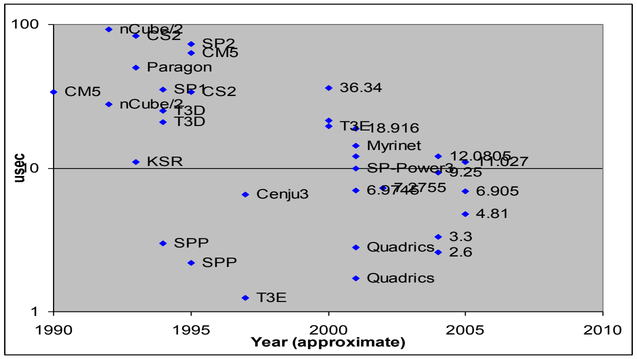
- Diameter: the maximum (over all pairs of nodes) of the shortest path between a given pair of nodes.
- Latency: delay between send and receive times
  - Latency tends to vary widely across architectures
  - Vendors often report hardware latencies (wire time)
  - Application programmers care about software latencies (user program to user program)
- Observations:
  - Latencies differ by 1-2 orders across network designs
  - Software/hardware overhead at source/destination dominate cost (1s-10s usecs)
  - Hardware latency varies with distance (10s-100s nsec per hop) but is small compared to overheads
- Latency is key for programs with many small messages

Latency on Some Machines/Networks



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

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

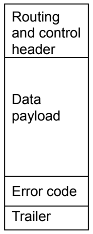


- Latency has not improved significantly, unlike Moore's Law
  - T3E (shmem) was lowest point – in 1997

Data from Kathy Yelick, UCB and NERSC

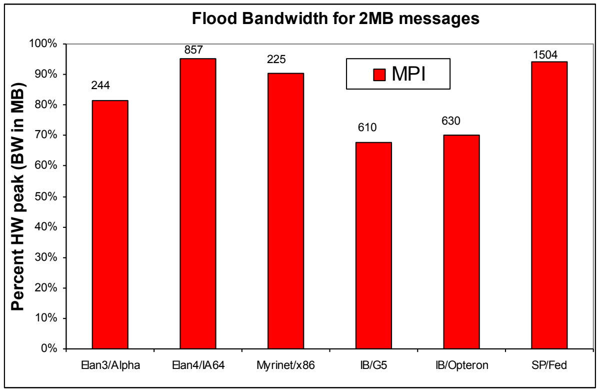
Performance Properties of a Network: Bandwidth

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



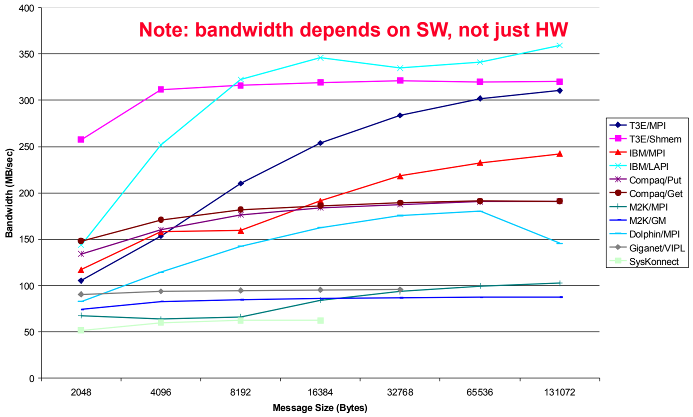
- Bandwidth is important for applications with mostly large messages

Bandwidth on Existing Networks



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

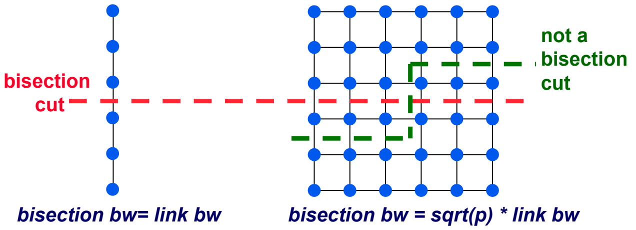
Bandwidth Chart



Data from Mike Welcome, NERSC

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 cut

not a bisection cut

bisection bw= link bw

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

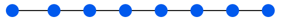
- Bisection bandwidth is important for algorithms in which all processors need to communicate with all others

Network Topology

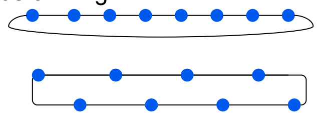
- In the past, there was considerable research in network topology and in mapping algorithms to topology.
  - Key cost to be minimized: number of "hops" between nodes (e.g. "store and forward")
  - Modern networks hide hop cost (i.e., "wormhole routing"), so topology less of a factor in performance of many algorithms
- 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
  - Example later of big performance impact

Linear and Ring Topologies

- Linear array



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

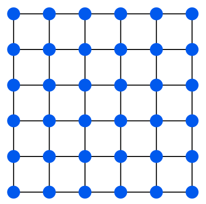


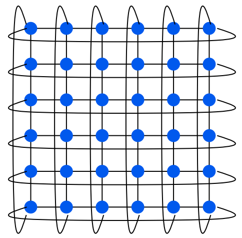
  - Diameter = n/2; average distance ~ n/4.
  - Bisection bandwidth = 2.
  - Natural for algorithms that work with 1D arrays.

Meshes and Tori – 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)**

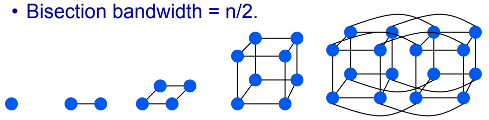




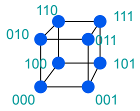
- **Generalizes to higher dimensions**
  - **Cray XT (eg Hopper@NERSC) uses 3D Torus**
- **Natural for algorithms that work with 2D and/or 3D arrays (matmul)**

Hypercubes

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

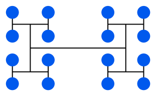


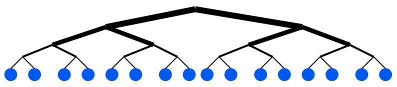
- 0d 1d 2d 3d 4d
- Popular in early machines (Intel iPSC, NCUBE).
  - Lots of clever algorithms.
  - See 1996 online CS267 notes.
- Greycode addressing:
  - Each node connected to d others with 1 bit different.

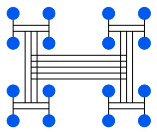


Trees

- 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.



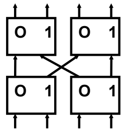




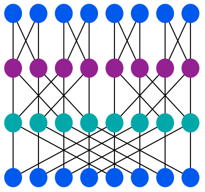
Butterflies

- 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



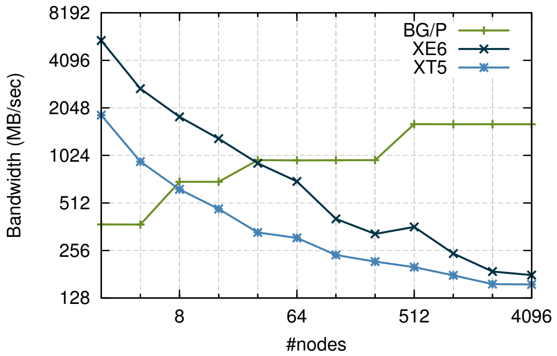
butterfly switch



multistage butterfly network

Does Topology Matter?

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



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

Dragonflies – used in Edison

- Motivation: Exploit gap in cost and performance between optical interconnects (which go between cabinets in a machine room) and electrical networks (inside cabinet)
  - Optical more expensive but higher bandwidth when long
  - Electrical networks cheaper, faster when short
- Combine in hierarchy
  - One-to-many via electrical networks inside cabinet
  - Just a few long optical interconnects between cabinets
- Clever routing algorithm to avoid bottlenecks:
  - Route from source to randomly chosen intermediate cabinet
  - Route from intermediate cabinet to destination
- Outcome: programmer can (usually) ignore topology, get good performance
  - Important in virtualized, dynamic environment
  - Programmer can still create serial bottlenecks
- Details in "Technology-Drive, Highly-Scalable Dragonfly Topology," J. Kim. W. Dally, S. Scott, D. Abts, ISCA 2008

Evolution of Distributed Memory Machines

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

Performance Models

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

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 "α−β model" and written

Time = α + n*β

- Usually α >> β >> time per flop.
  - One long message is cheaper than many short ones.

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

  - Can do hundreds or thousands of flops for cost of one message.
- Lesson: Need large computation-to-communication ratio to be efficient.
- LogP – more detailed model (**L**atency/**o**verhead/**g**ap/**P**roc.)

Alpha-Beta Parameters on Current Machines

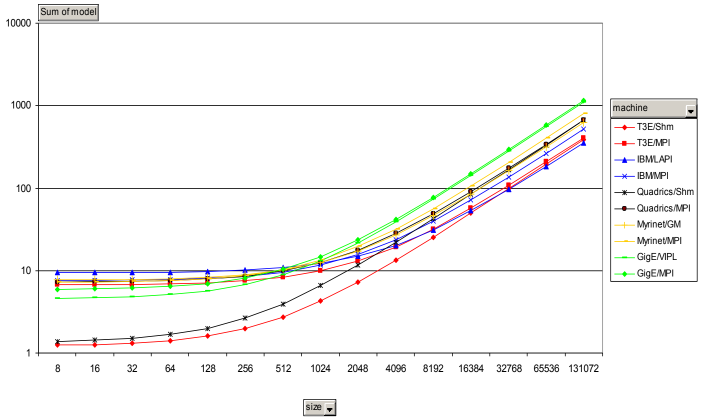
- These numbers were obtained empirically

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

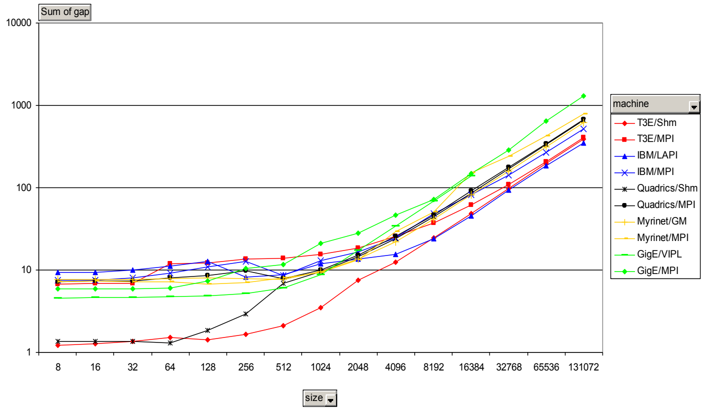
α is latency in usecs
β is BW in usecs per Byte

How well does the model
Time = α + n*β
predict actual performance?

Model Time Varying Message Size & Machines



Measured Message Time



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, prefix sum, … of data on many processors
  - Synchronization
    - Barrier
    - No locks because there are no shared variables to protect
  - Enquiries
    - How many processes? Which one am I? Any messages waiting?

Novel Features of MPI

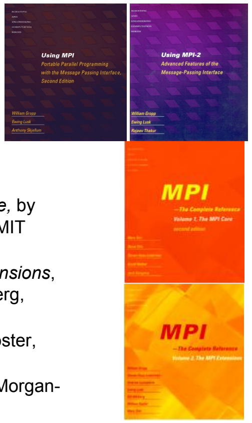
- Communicators encapsulate communication spaces for library safety
- Datatypes reduce copying costs and permit heterogeneity
- Multiple communication modes allow precise buffer management
- Extensive collective operations for scalable global communication
- Process topologies permit efficient process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

MPI References

- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
  - Latest version MPI 3.0, released Sept 2012
- 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

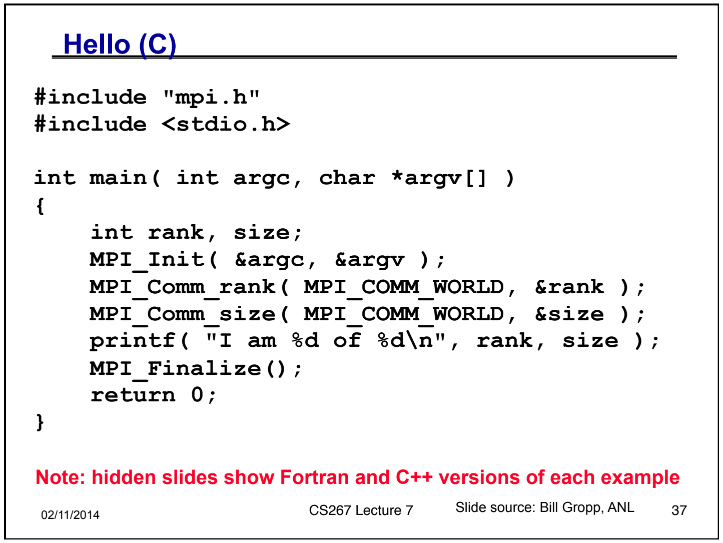
Books on MPI

- *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.



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



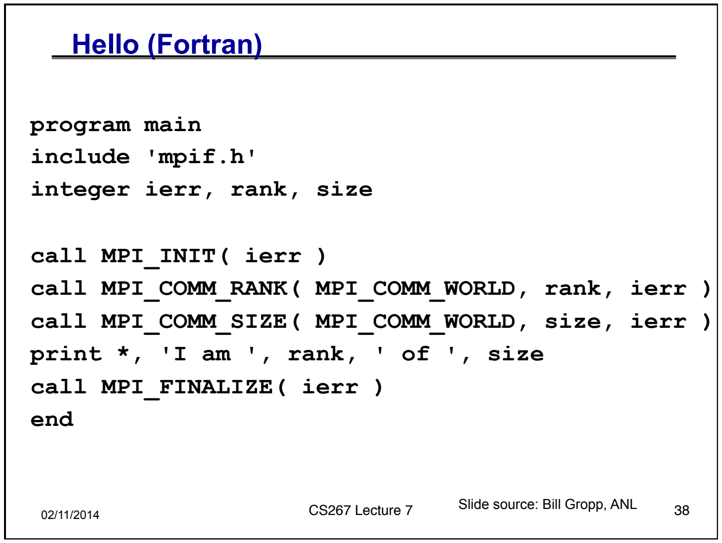
Hello (C)

```
#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;
}
```

Note: hidden slides show Fortran and C++ versions of each example

02/11/2014 CS267 Lecture 7 Slide source: Bill Gropp, ANL 37

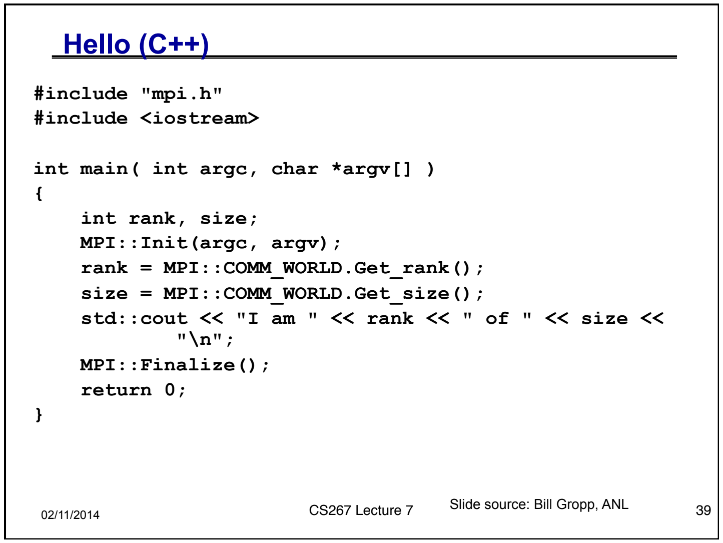


Hello (Fortran)

```
program main
include 'mpif.h'
integer ierr, rank, size

call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )
print *, 'I am ', rank, ' of ', size
call MPI_FINALIZE( ierr )
end
```

02/11/2014 CS267 Lecture 7 Slide source: Bill Gropp, ANL 38

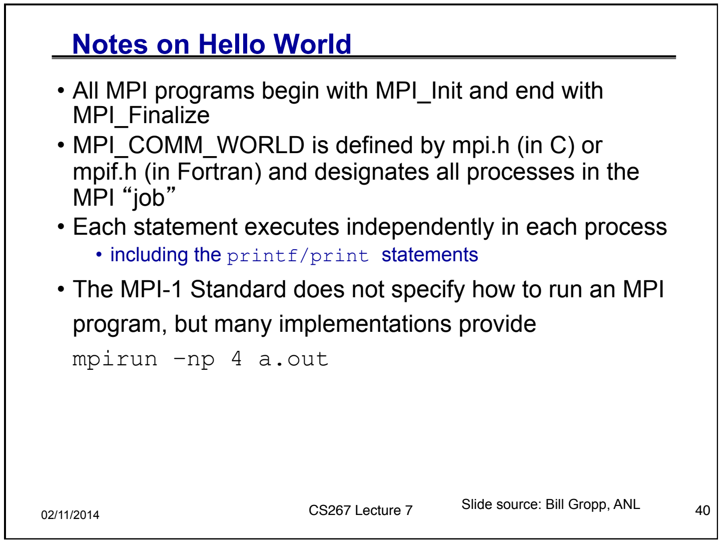


Hello (C++)

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI::Init(argc, argv);
    rank = MPI::COMM_WORLD.Get_rank();
    size = MPI::COMM_WORLD.Get_size();
    std::cout << "I am " << rank << " of " << size <<
            "\n";
    MPI::Finalize();
    return 0;
}
```

02/11/2014 CS267 Lecture 7 Slide source: Bill Gropp, ANL 39



Notes on Hello World

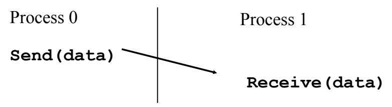
- All MPI programs begin with MPI_Init and end with MPI_Finalize
- MPI_COMM_WORLD is defined by mpi.h (in C) or mpif.h (in Fortran) and designates all processes in the MPI "job"
- Each statement executes independently in each process
  - including the `printf/print` statements
- The MPI-1 Standard does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

```
mpirun -np 4 a.out
```

02/11/2014 CS267 Lecture 7 Slide source: Bill Gropp, ANL 40

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?

Slide source: Bill Gropp, ANL

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`**

Slide source: Bill Gropp, ANL

MPI Datatypes

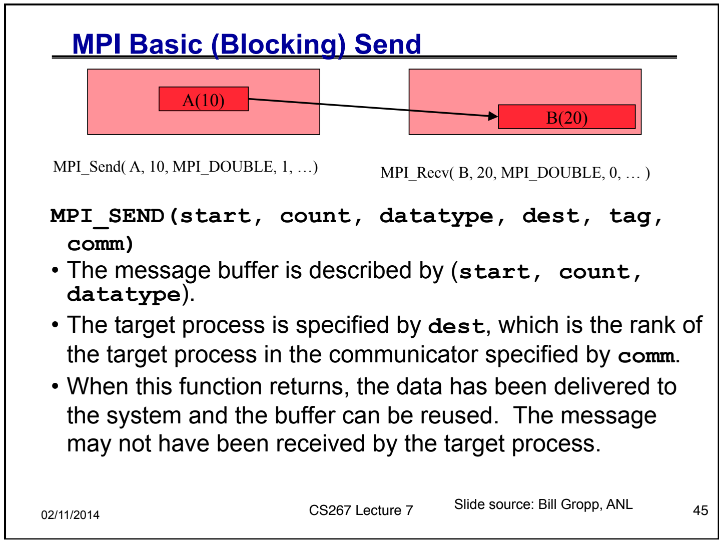
- The data in a message to send or receive is described by a triple (address, count, datatype), where
- An MPI datatype is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
  - a contiguous array of MPI datatypes
  - a strided block of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

Slide source: Bill Gropp, ANL

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

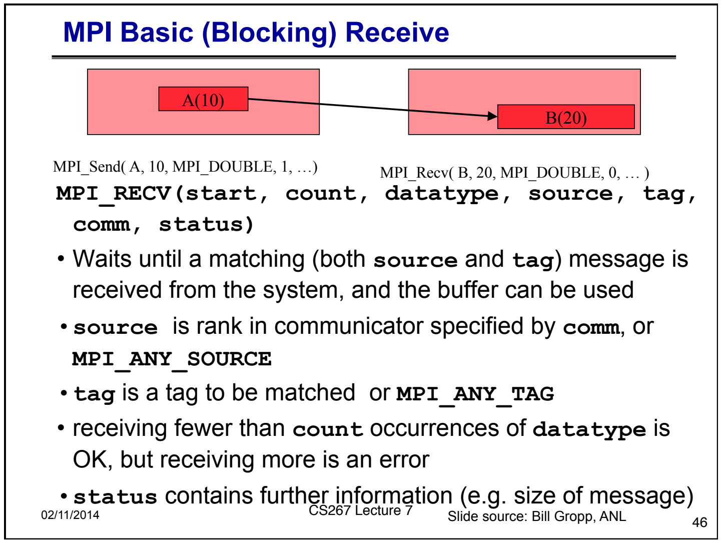
MPI_Send(A, 10, MPI_DOUBLE, 1, …)

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.

Slide source: Bill Gropp, ANL



MPI Basic (Blocking) Receive

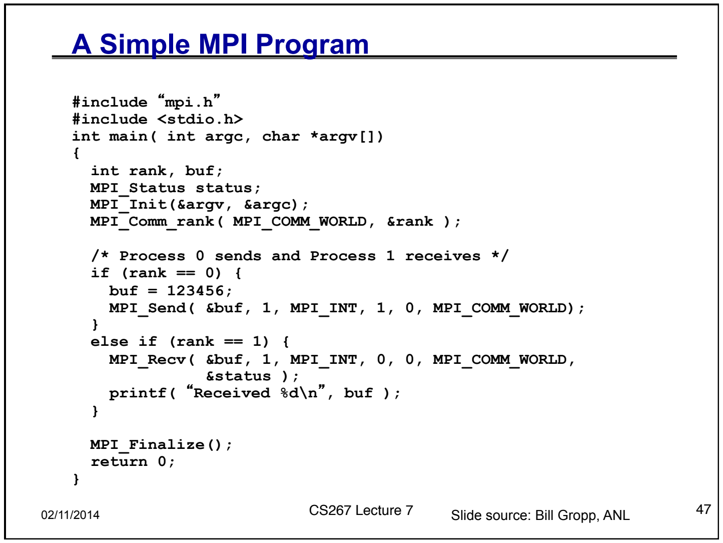
MPI_Send(A, 10, MPI_DOUBLE, 1, …)

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)

Slide source: Bill Gropp, ANL



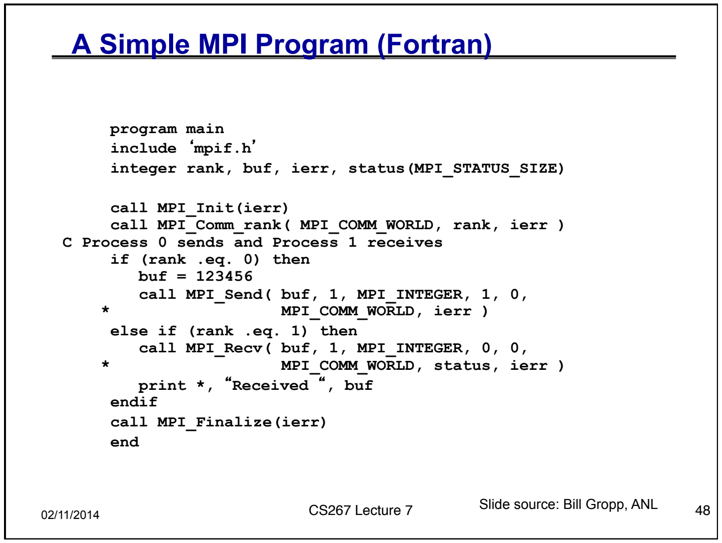
A Simple MPI Program

```
#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;
}
```

Slide source: Bill Gropp, ANL

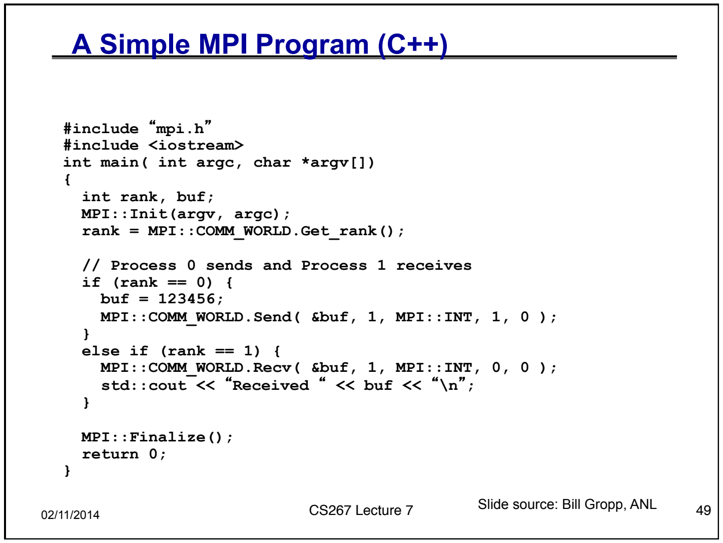


A Simple MPI Program (Fortran)

```
      program main
      include ‘mpif.h’
      integer rank, buf, ierr, status(MPI_STATUS_SIZE)

      call MPI_Init(ierr)
      call MPI_Comm_rank( MPI_COMM_WORLD, rank, ierr )
C Process 0 sends and Process 1 receives
      if (rank .eq. 0) then
         buf = 123456
         call MPI_Send( buf, 1, MPI_INTEGER, 1, 0,
     *                  MPI_COMM_WORLD, ierr )
      else if (rank .eq. 1) then
         call MPI_Recv( buf, 1, MPI_INTEGER, 0, 0,
     *                  MPI_COMM_WORLD, status, ierr )
         print *, “Received “, buf
      endif
      call MPI_Finalize(ierr)
      end
```

Slide source: Bill Gropp, ANL



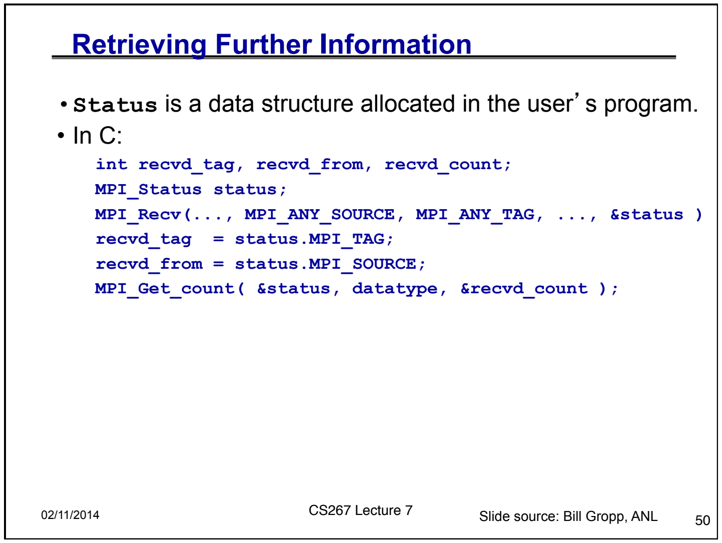
A Simple MPI Program (C++)

```
#include "mpi.h"
#include <iostream>
int main( int argc, char *argv[])
{
  int rank, buf;
  MPI::Init(argv, argc);
  rank = MPI::COMM_WORLD.Get_rank();

  // Process 0 sends and Process 1 receives
  if (rank == 0) {
    buf = 123456;
    MPI::COMM_WORLD.Send( &buf, 1, MPI::INT, 1, 0 );
  }
  else if (rank == 1) {
    MPI::COMM_WORLD.Recv( &buf, 1, MPI::INT, 0, 0 );
    std::cout << "Received " << buf << "\n";
  }

  MPI::Finalize();
  return 0;
}
```

02/11/2014 CS267 Lecture 7 Slide source: Bill Gropp, ANL 49

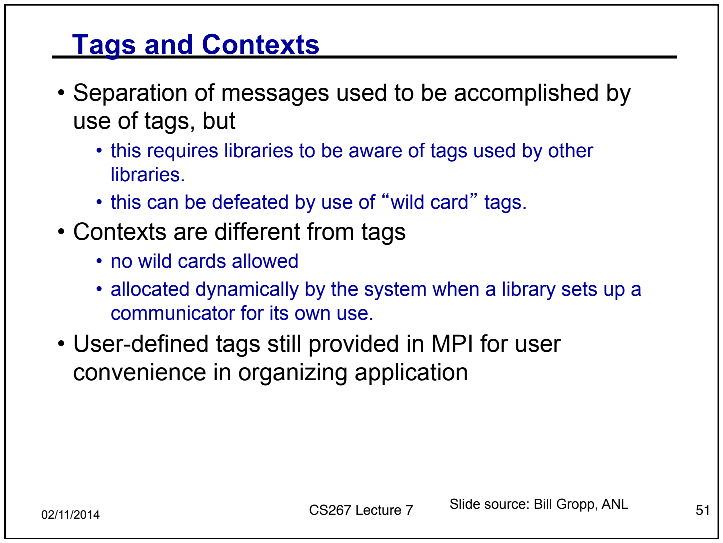


Retrieving Further Information

- `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 );
```

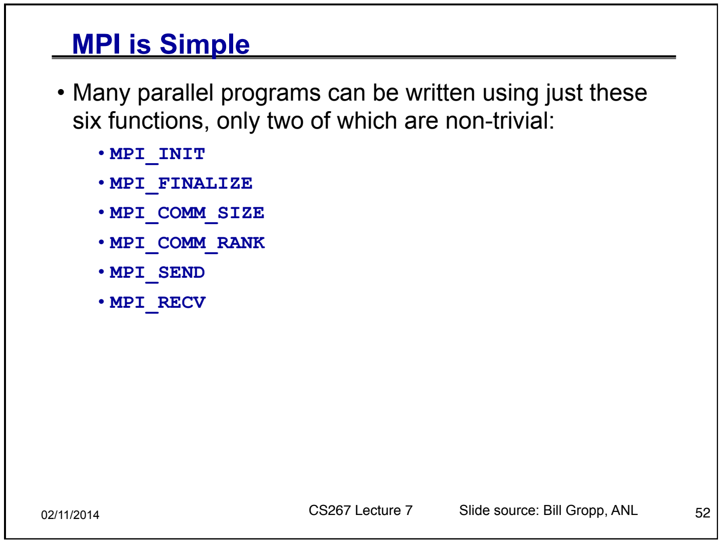
02/11/2014 CS267 Lecture 7 Slide source: Bill Gropp, ANL 50



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

02/11/2014 CS267 Lecture 7 Slide source: Bill Gropp, ANL 51



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**

02/11/2014 CS267 Lecture 7 Slide source: Bill Gropp, ANL 52

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…

Slide source: Bill Gropp, ANL

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

Alternative Set of 6 Functions

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

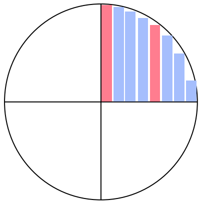
- Using point-to-point:
  - **`MPI_INIT`**
  - **`MPI_FINALIZE`**
  - **`MPI_COMM_SIZE`**
  - **`MPI_COMM_RANK`**
  - **`MPI_SEND`**
  - **`MPI_RECEIVE`**

- Using collectives:
  - **`MPI_INIT`**
  - **`MPI_FINALIZE`**
  - **`MPI_COMM_SIZE`**
  - **`MPI_COMM_RANK`**
  - **`MPI_BCAST`**
  - **`MPI_REDUCE`**

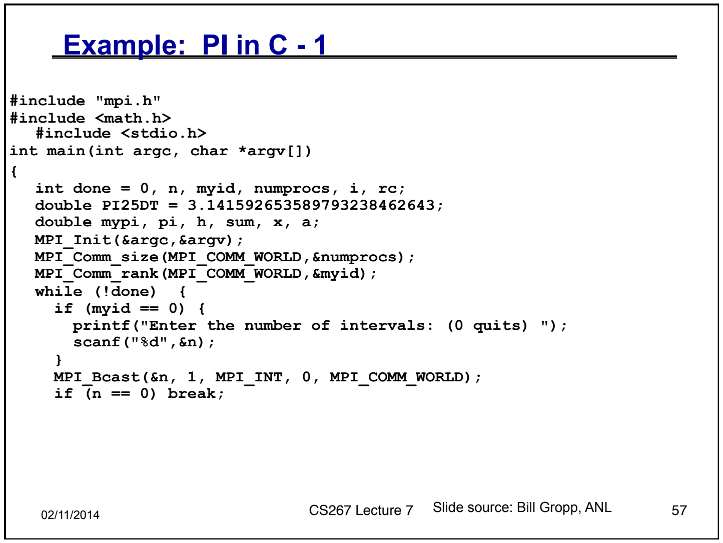
- You may use more for convenience or performance

Example: Calculating Pi

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



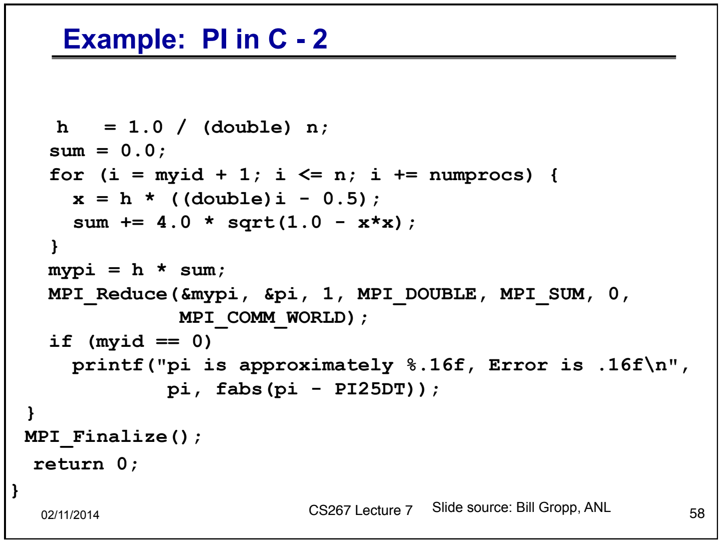
- Simple program written in a data parallel style in MPI
  - E.g., for a reduction (recall “tricks with trees” lecture), each process will first reduce (sum) its own values, then call a collective to combine them
- Estimates pi by approximating the area of the quadrant of a unit circle
- Each process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



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;
```

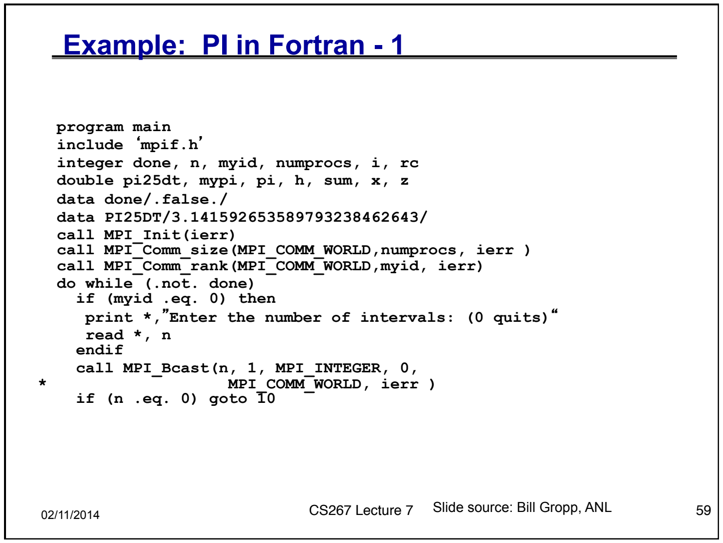
Slide source: Bill Gropp, ANL



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 * 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;
}
```

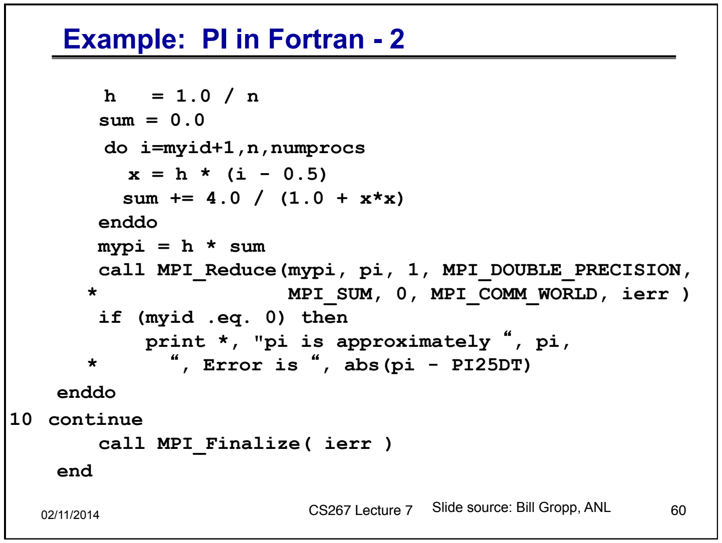
Slide source: Bill Gropp, ANL



Example: PI in Fortran - 1

```
   program main
   include ‘mpif.h’
   integer done, n, myid, numprocs, i, rc
   double pi25dt, mypi, pi, h, sum, x, z
   data done/.false./
   data PI25DT/3.141592653589793238462643/
   call MPI_Init(ierr)
   call MPI_Comm_size(MPI_COMM_WORLD,numprocs, ierr )
   call MPI_Comm_rank(MPI_COMM_WORLD,myid, ierr)
   do while (.not. done)
     if (myid .eq. 0) then
      print *,”Enter the number of intervals: (0 quits)“
      read *, n
     endif
     call MPI_Bcast(n, 1, MPI_INTEGER, 0,
*                   MPI_COMM_WORLD, ierr )
     if (n .eq. 0) goto 10
```

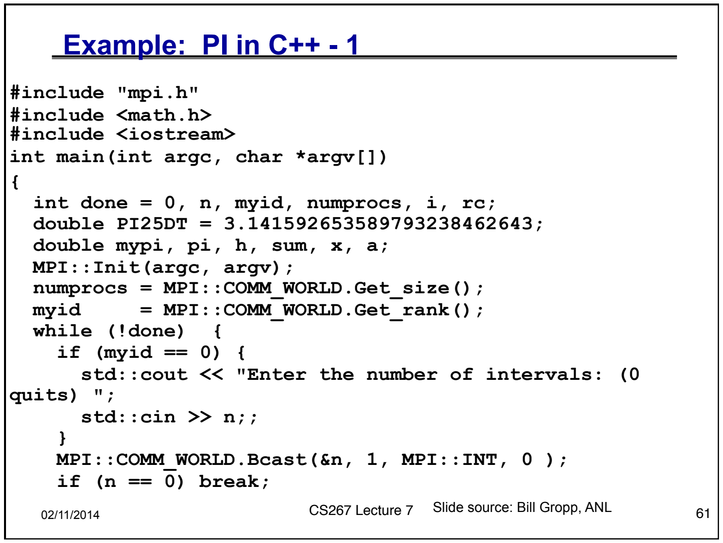
Slide source: Bill Gropp, ANL



Example: PI in Fortran - 2

```
        h   = 1.0 / n
       sum = 0.0
        do i=myid+1,n,numprocs
          x = h * (i - 0.5)
         sum += 4.0 / (1.0 + x*x)
       enddo
       mypi = h * sum
       call MPI_Reduce(mypi, pi, 1, MPI_DOUBLE_PRECISION,
      *                MPI_SUM, 0, MPI_COMM_WORLD, ierr )
       if (myid .eq. 0) then
           print *, "pi is approximately “, pi,
      *       “, Error is “, abs(pi - PI25DT)
    enddo
10  continue
       call MPI_Finalize( ierr )
    end
```

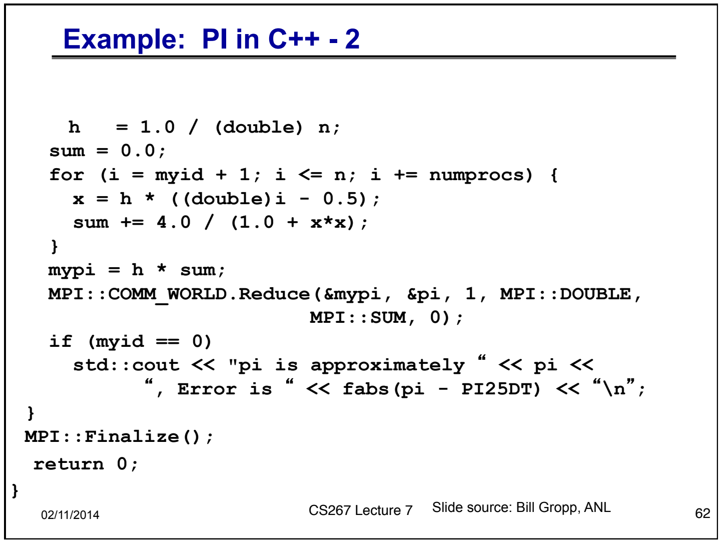
Slide source: Bill Gropp, ANL



Example: PI in C++ - 1

```
#include "mpi.h"
#include <math.h>
#include <iostream>
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);
  numprocs = MPI::COMM_WORLD.Get_size();
  myid     = MPI::COMM_WORLD.Get_rank();
  while (!done)  {
    if (myid == 0) {
      std::cout << "Enter the number of intervals: (0
quits) ";
      std::cin >> n;;
    }
    MPI::COMM_WORLD.Bcast(&n, 1, MPI::INT, 0 );
    if (n == 0) break;
```

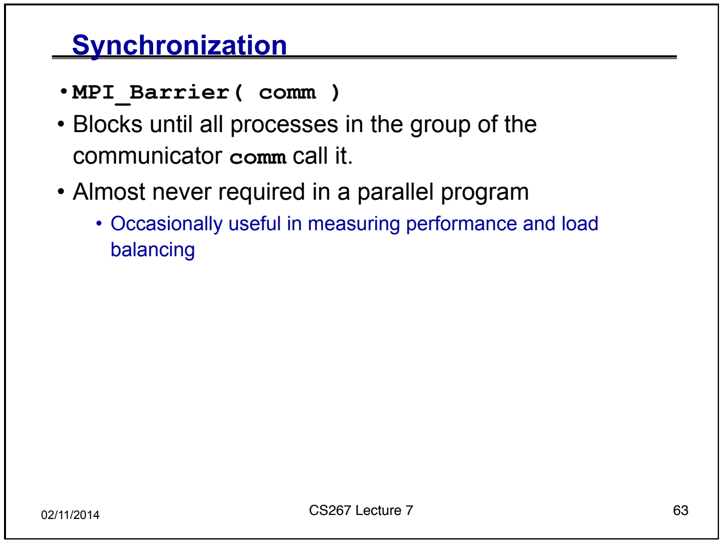
Slide source: Bill Gropp, ANL



Example: PI in C++ - 2

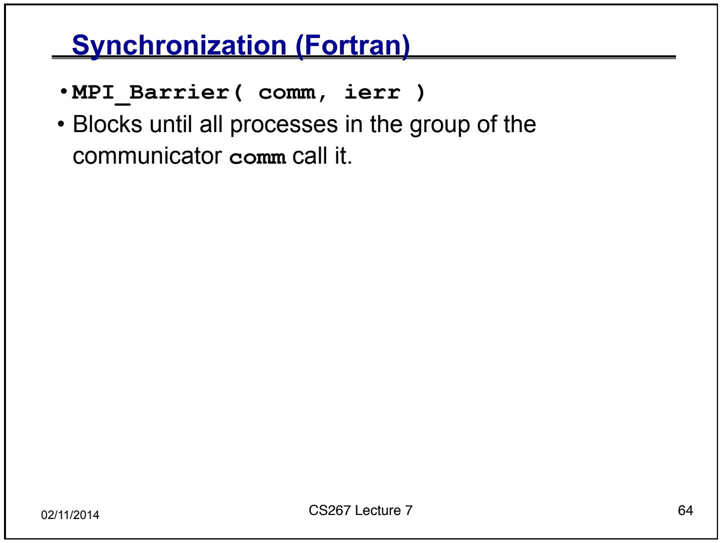
```
    h   = 1.0 / (double) n;
   sum = 0.0;
   for (i = myid + 1; i <= n; i += numprocs) {
     x = h * ((double)i - 0.5);
     sum += 4.0 / (1.0 + x*x);
   }
   mypi = h * sum;
   MPI::COMM_WORLD.Reduce(&mypi, &pi, 1, MPI::DOUBLE,
                          MPI::SUM, 0);
   if (myid == 0)
     std::cout << "pi is approximately “ << pi <<
           “, Error is “ << fabs(pi - PI25DT) << “\n”;
 }
 MPI::Finalize();
  return 0;
}
```

Slide source: Bill Gropp, ANL



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



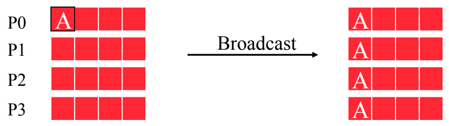
Synchronization (Fortran)

- **MPI_Barrier(comm, ierr)**
- Blocks until all processes in the group of the communicator **comm** call it.

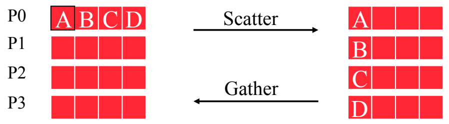
Synchronization (C++)

- `comm.Barrier();`
- Blocks until all processes in the group of the communicator `comm` call it.

Collective Data Movement



P0	A					A			
P1					Broadcast →	A			
P2						A			
P3						A			

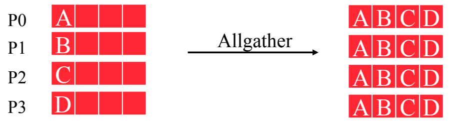


P0	A	B	C	D	Scatter →	A			
P1						B			
P2					← Gather	C			
P3						D			

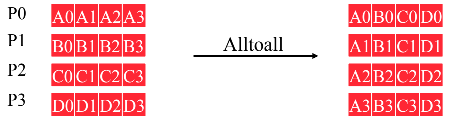
Comments on Broadcast

- 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
  - "root" argument is the rank of the sender; this tells MPI which process originates the broadcast and which receive

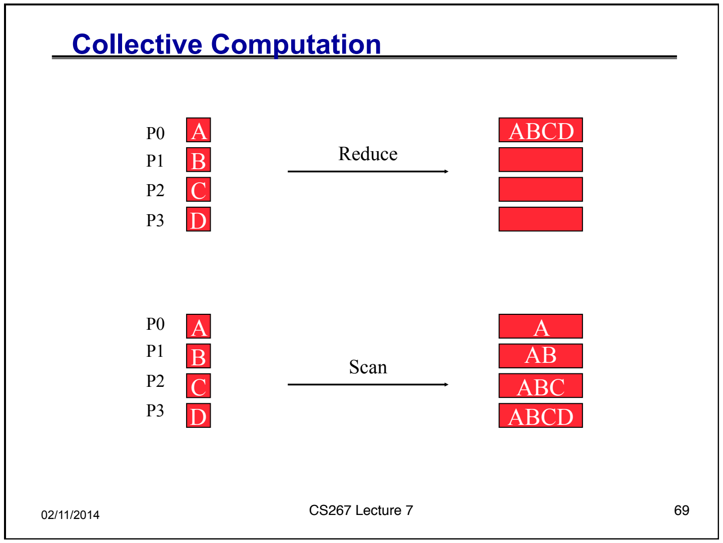
More Collective Data Movement



P0	A					A	B	C	D
P1	B				Allgather →	A	B	C	D
P2	C					A	B	C	D
P3	D					A	B	C	D



P0	A0	A1	A2	A3		A0	B0	C0	D0
P1	B0	B1	B2	B3	Alltoall →	A1	B1	C1	D1
P2	C0	C1	C2	C3		A2	B2	C2	D2
P3	D0	D1	D2	D3		A3	B3	C3	D3



Collective Computation

P0 A
P1 B
P2 C
P3 D

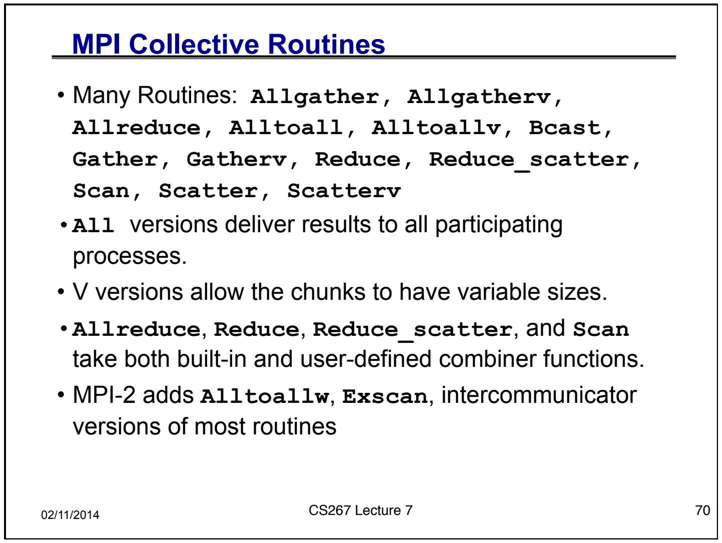
Reduce

ABCD

P0 A
P1 B
P2 C
P3 D

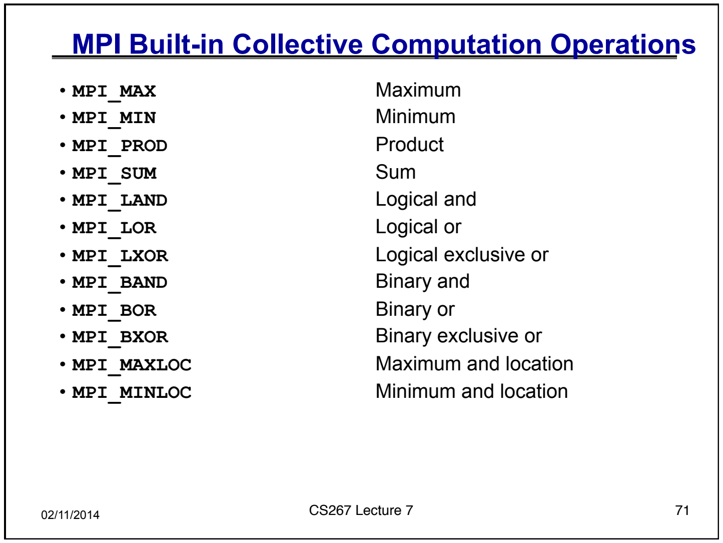
Scan

A
AB
ABC
ABCD



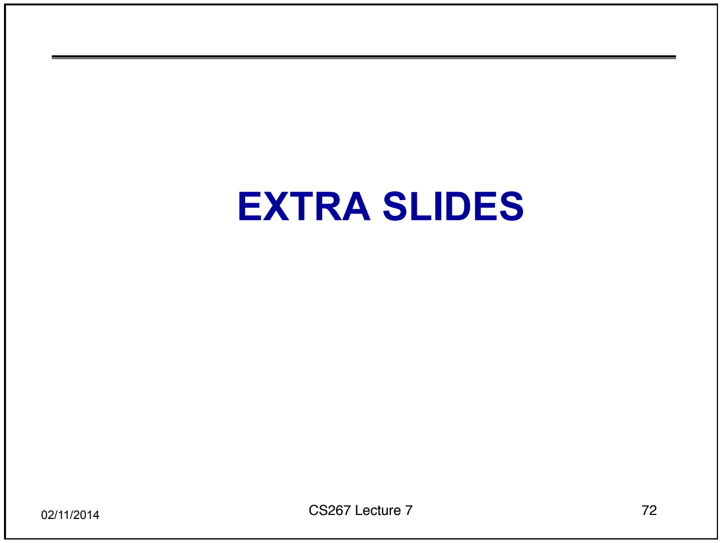
MPI Collective Routines

- 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 chunks 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



EXTRA SLIDES