Message Passing

ICS632: Principles of High Performance Computing

Henri Casanova (henric@hawaii.edu)

Fall 2015

Outline

Not enough memory, not enough time

- Many applications can't run on a shared-memory machine
  - There are reasonably large (and expensive!) shared-memory machines (SGI Altix)
  - But it's been ages since they've been on Top500
- Application may need a lot of memory and/or a lot of computational power
- A practical, economical way of achieving these needs is to put a bunch of small machines together in a room, wire them with a fast network, have good cooling, maintain an OS image across the machines, and let users write code in the form of communicating processes
- And, just like that, we loose the shared memory abstraction

Shared- vs. Distributed-Memory Programming

- Shared-memory: threads that share memory
  - Programming is more or less convenient as threads "see" the same memory
  - Parallelizing a code can be easy
    - e.g., a few OpenMP pragmas
  - Incremental parallelization is natural
  - Understanding parallel code is not too hard
- Distributed-memory: processes that do not share memory
  - Processes must exchange message explicitly
    - Your code contains calls like `send()`, `recv()`, often with complex communication patterns
  - Understanding parallel code is typically difficult
  - Incremental parallelization can be very challenging
  - Parallelization often entails major algorithmic modifications

Abstractions

- Message-passing programming with "point-to-point" communication can be messy
  - A fun way to produce "spaghetti code" that's really difficult to debug
- As a result there has been standard APIs to attempt to formalize it
- Abstractions have been built on top of message passing:
  - Collective communications (broadcasts, all-to-all, etc.)
  - Distributed objects
  - Distributed Shared Memory!
- A lot of HPC code is being built using only point-to-point and collective communications
  - And tons of legacy code of course

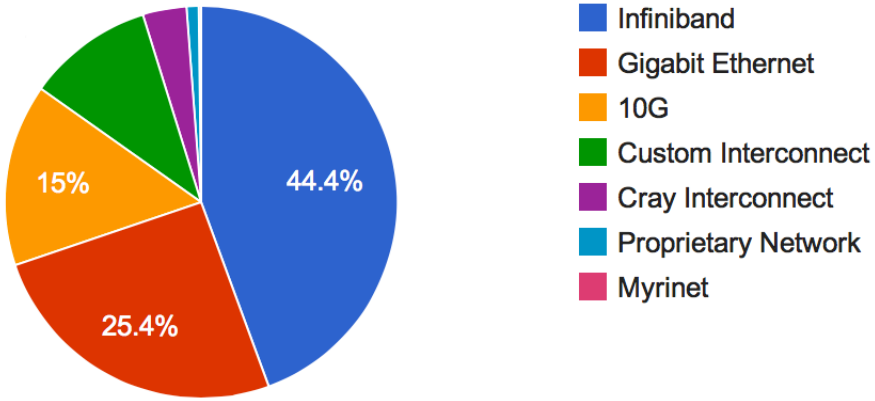
Outline

HPC Interconnects

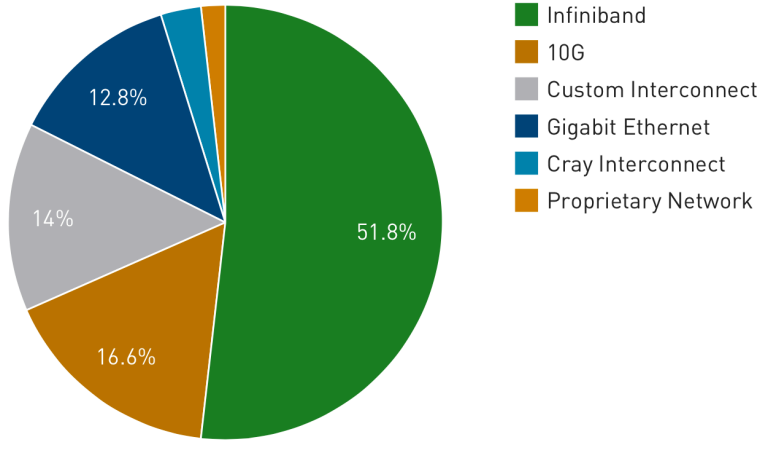
- Distributed-memory systems (e.g., clusters, MPPs) are build with a network
- Network of compute nodes, network of small "routers", network of high-radix switches, combinations of them all
- Custom interconnects
  - Tianhe-2 (China): compute nodes connected in groups via switches, themselves connected to 13 576-port central switches in a fat tree fashion
  - Titan (ORNL): 2 nodes share a router, routers are in a 3-D torus
  - Sequoia (LLNL): nodes are connected in a 5-D torus
  - K-Computer (Japan): nodes are connected in a 3-D torus of 3-D tori
- Commodity interconnects (networks of switches)
  - 1G Ethernet, 10G Ethernet, Infiniband
- Let's look at the Top500 distribution

Top500 Topology Families (June 2014 & June 2015)

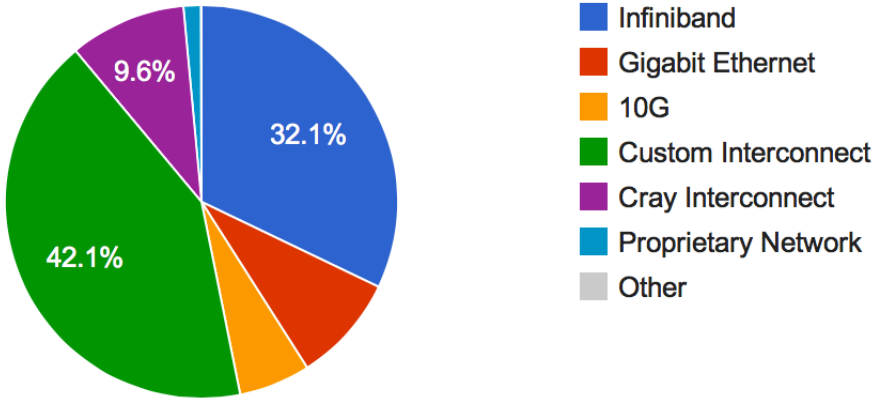
Interconnect Family System Share



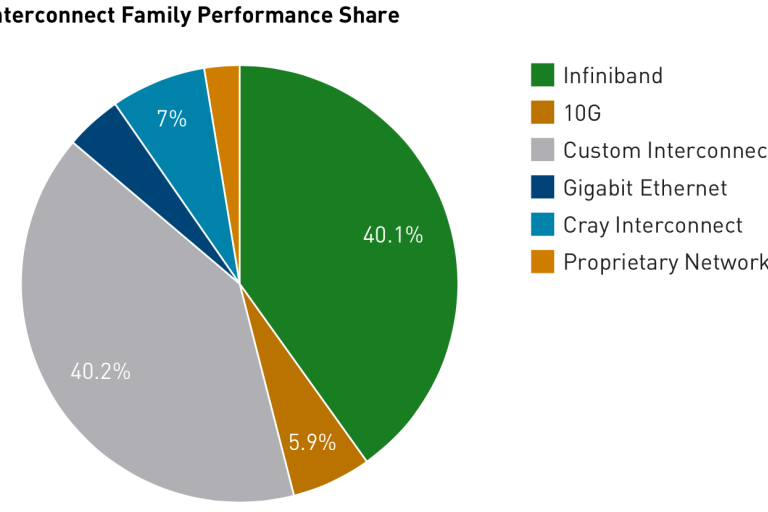
Interconnect Family System Share



Interconnect Family Performance Share



Interconnect Family Performance Share



What is a good network?

- So we have various topologies interconnecting compute nodes directly, compute nodes together via switches, switches together via other switches, etc.
- In the end, we have a graph with vertices and edges, where each edge is labeled by a latency and a bandwidth
  - Somewhat of a simplification
- Latencies and bandwidths are defined by the technology in use and the budget in use
- The overall shape of the graph is called the topology

Topology Metrics

- Diameter: maximum shortest path length (in hops)
  - High diameter means that there are "distant" compute nodes that perhaps shouldn't talk to each other
  - Maybe no a problem given the application's communication pattern
- Degree
  - Often though of as related to cost (NICs, used switch ports)
  - Regular? Non-regular?
- Bisection
  - Minimum number of edges over all possible balanced cuts
  - Should be indicative of the throughput that can be achieved when a bunch of communication happens all over the place

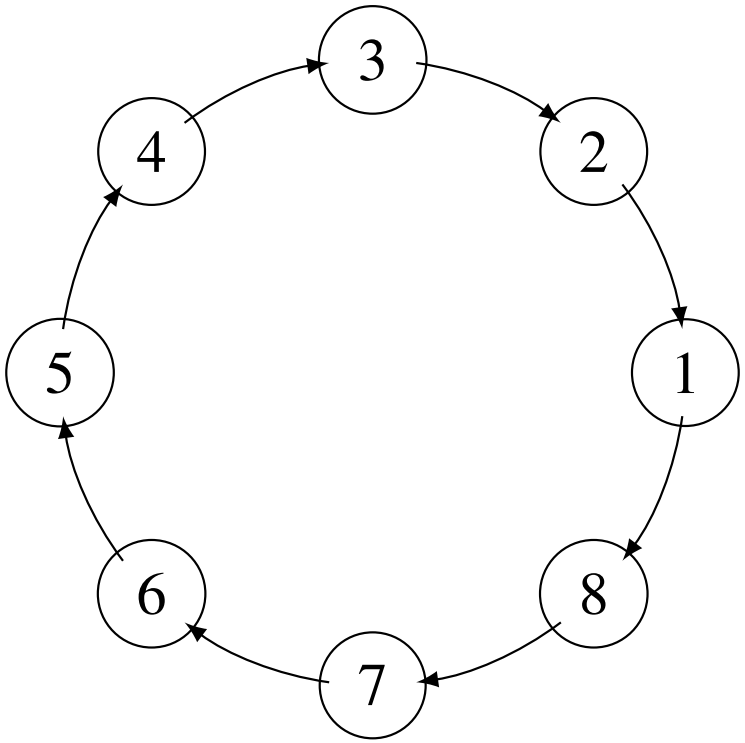
Topology Metrics

- Diameter: maximum shortest path length (in hops)
  - High diameter means that there are "distant" compute nodes that perhaps shouldn't talk to each other
  - Maybe no a problem given the application's communication pattern
- Degree
  - Often though of as related to cost (NICs, used switch ports)
  - Regular? Non-regular?
- Bisection
  - Minimum number of edges over all possible balanced cuts
  - Should be indicative of the throughput that can be achieved when a bunch of communication happens all over the place

Topology Metrics

- Diameter: maximum shortest path length (in hops)
  - High diameter means that there are "distant" compute nodes that perhaps shouldn't talk to each other
  - Maybe no a problem given the application's communication pattern
- Degree
  - Often though of as related to cost (NICs, used switch ports)
  - Regular? Non-regular?
- Bisection
  - Minimum number of edges over all possible balanced cuts
  - Should be indicative of the throughput that can be achieved when a bunch of communication happens all over the place

A Ring Topology

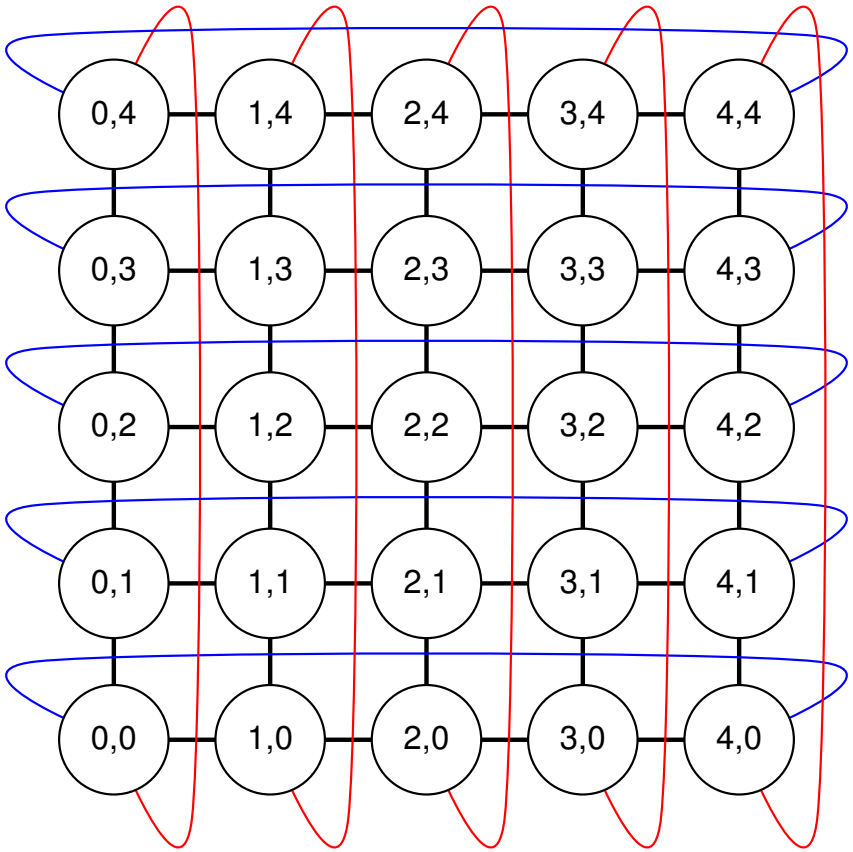


- diameter = $N/2$, degree = 2, bisection bandwidth = 2

k-ary, n-cube

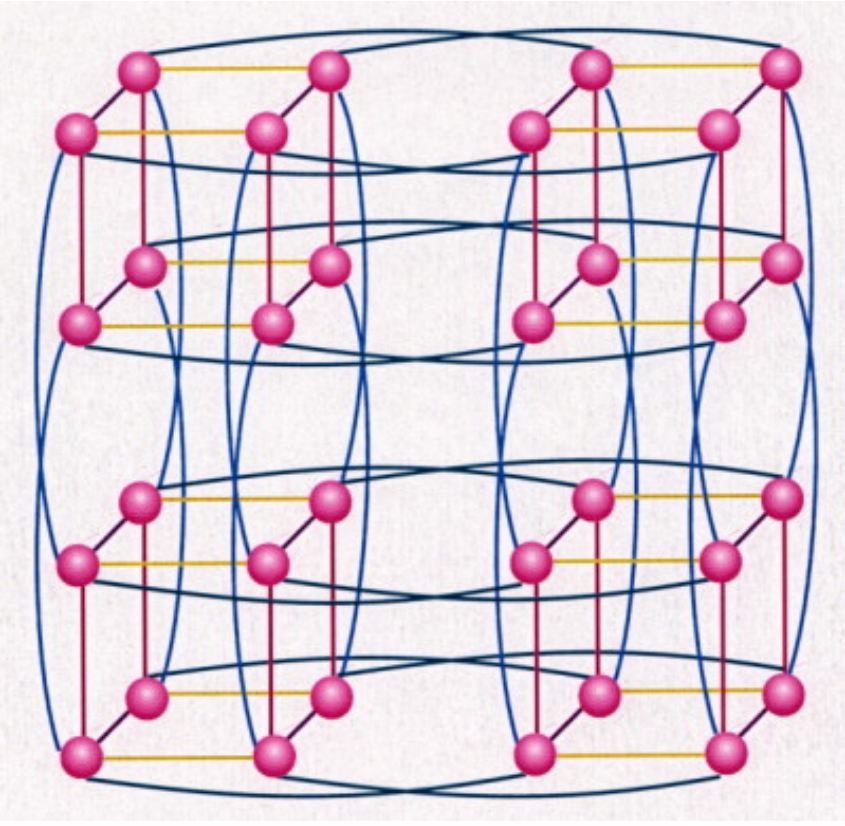
- A very popular topology used in HPC systems
- k^n nodes, arranged in n dimensions
  - A ring is a n-ary, 1-cube with wrap-around, i.e., a 1-D torus
- Each node is identified by an n-digit address where each digit is in between 1 and k
- Connect two nodes if their addresses differ only by 1 in only one digit
- Typically called a n-D grid
- Wrap-around (i.e., k and 1 do differ by 1) to obtain what is typically called an n-D torus
- degree: $2n$, diameter: $nk/2$, bisection: $2k^{n-1}$ (for $k > 2$)

5-ary, 2-cube



- k^2 nodes, degree: 4, diameter: 5, bisection: 10

2-ary, n-cube: The Hypercube

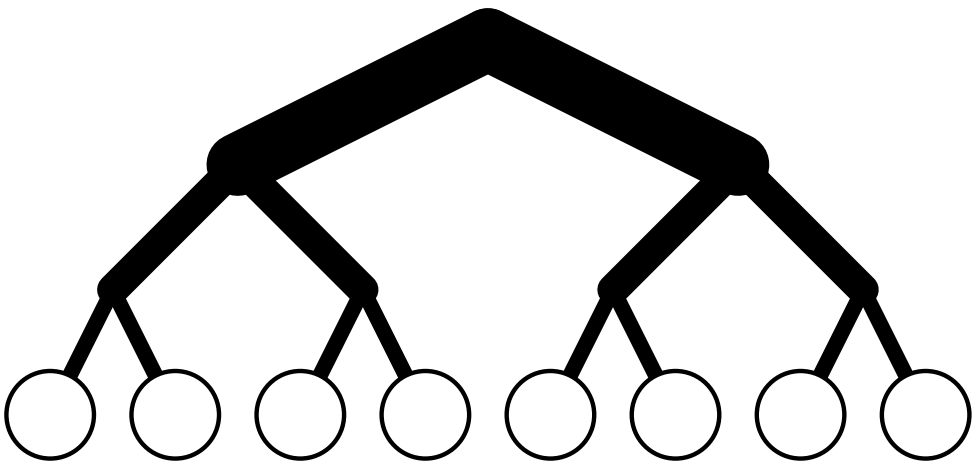


- 2^n nodes, degree: n, diameter: n, bisection: $2^n/2$

The Hypercube

- The hypercube is widely studied because it has all kinds of nice properties
- Diameter increases logarithmically with scale
- Degree increases logarithmically with scale
- There is an elegant dimension-order routing scheme
  - Route to a neighbor that reduces the Hamming distance between the address of the source and that of the destination
  - (0,1,1,0) $\rightarrow$ (1,1,1,0) $\rightarrow$ (1,1,0,0) $\rightarrow$ (1,1,0,1) $\rightarrow$ (1,1,0,1)
- Several tweaks on the hypercube
  - e.g., "folded" hypercube (add on edge between each node and its most distant neighbor) to reduce diameter

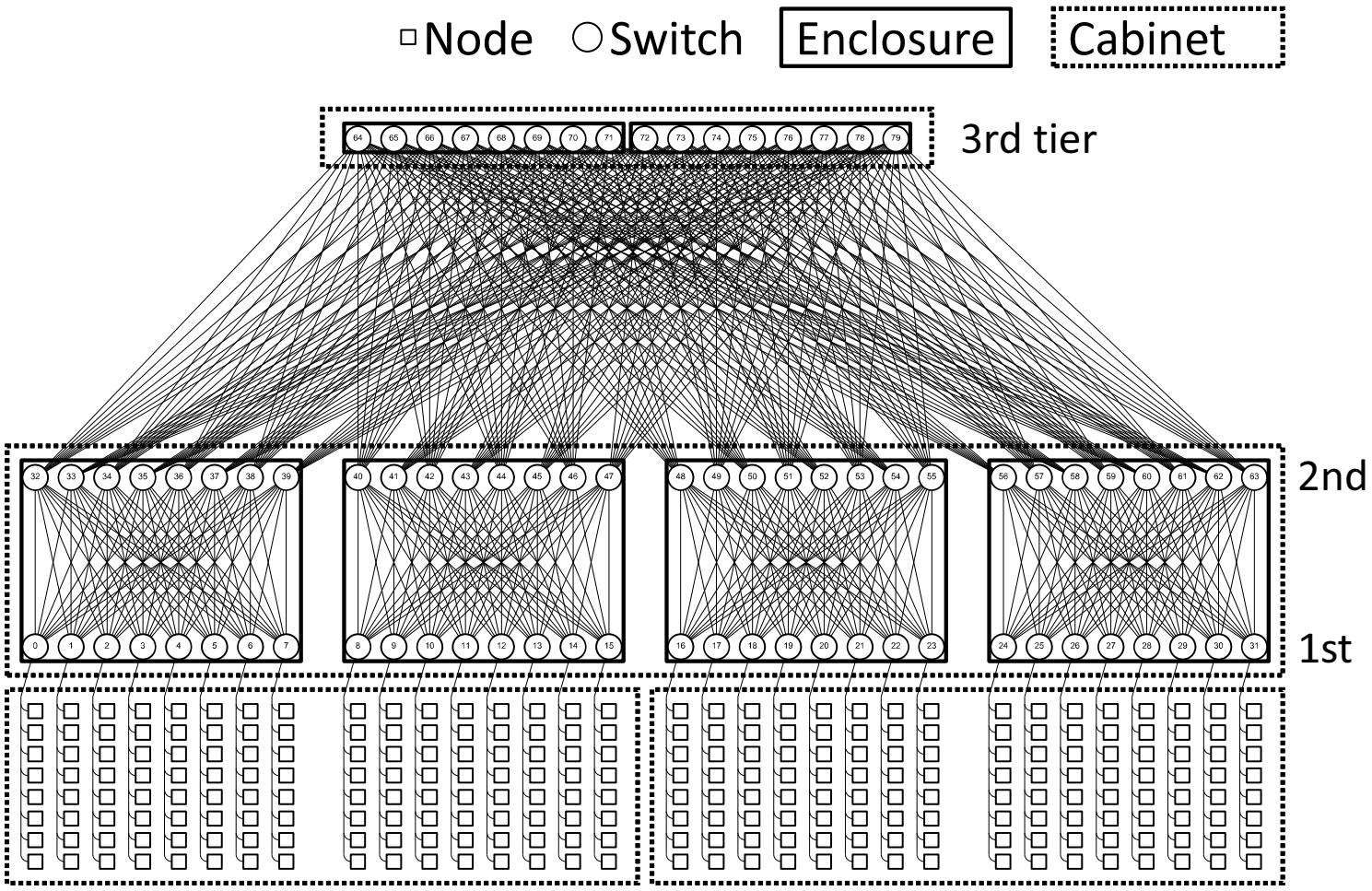
Fat Tree Topology



- A k-ary tree structure
- Bandwidth on "links" is higher at higher levels
- This is achieved via bandwidth aggregation with multiple physical links

MyriClos “Fat Tree” Network

- Topology of switches with bandwidth-aggregation



Practical Topology Issues

- Aggregate cable length (cost) and packaging complexity
- Cable medium (copper $<$10m, optical < 100m)
- Cable redundancy for resilience

Our Cray's network

- Our Cray machine uses Infiniband (the most common technology in the Top500)
  - Switches, all optical cables
- Our system has a single switch!
  - A so-called star topology
- Larger systems would have a topology of many switches
  - Building a fast huge switch is expensive
- Having a single switch removes many interesting/vexing network issues

Some New Topology Research Directions

- Design topologies that explicitly account for the physical layout to reduce cable length
  - e.g., Dragonfly, HyperX
- Use random topologies
  - One of my research areas
  - I may talk about it at some point in the course if there is interest
- Use cable-less communication
  - Fast Wifi, Free-Space Optics?
  - One of my research areas
- Should we focus on cable delay rather than hop counts (because switches are getting really fast)?
  - Fast Wifi, Free-Space Optics?
  - One of my research areas

Outline

A Standard Message Passing Interface?

- Processes have communicated over the network for decades, e.g., using low-level IP sockets
- Why would we need some special API?
- Reason #1: Sockets are too low-level
  - No notion of data types, message tags, etc.
  - Only point-to-point communication
  - So we need a higher-level API
- Reason #2: Socket API is non-trivial
  - Non-computer scientists who write scientific computing code should be able to develop distributed-memory code
- Reason #3: Need for a standard that can be optimized by vendors
  - A supercomputer should come with a good message passing implementation with bells and whistles
  - IP is not even necessarily the right protocol!

A Brief History of Message Passing

- Many vendors started building distributed-memory machines in the mid 80's
- Each provided a message passing library
  - Caltech's Hypercube and Crystalline Operating System (CROS) - 1984
  - Meiko CS-1 and Occam - circa 1990
    - Transputer based (32-bit processor with 4 communication links, with fast multitasking/multithreading)
    - Occam: formal language for parallel processing:
- Lessons:
  - Getting people to embrace a parallel language is difficult
  - Better to do extensions to an existing (popular) language
  - Better to just design a library?

A Brief History of Message Passing

- The Intel iPSC1, Paragon and NX
  - Originally close to the Caltech Hypercube and CROS
  - iPSC1's message passing system hides underlying communication topology (process rank), multiple processes per node, any-to-any message passing, non-synchronous messages, message tags, variable message lengths, etc.
  - The Paragon's NX2 added interrupt-driven communications, some notion of filtering of messages with wildcards, global synchronization, arithmetic reduction operations
- Many other systems: IBM SPs and EUI, Meiko CS-2 and CSTools, Thinking Machine CM5 and the CMMD Active Message Layer (AML), etc.
- Many of their features are part of modern message passing

Toward a Standard

- We went from highly restrictive systems to systems close to today's state-of-the-art of message passing
- The main problem was: impossible to write portable code!
  - Programmers became expert of one system
  - The systems would die eventually and one had to relearn a new system (I learned NX!)
- People started writing their own "portable" libraries
  - Tricks with macros (PICL, P4, PVM, CHIMPS, ...)
- Performance issue: if I invest millions in an IBM-SP, do I really want to use some library that uses (slow) sockets??
- There was no clear winner for a long time although PVM (Parallel Virtual Machine) had won in the end
- After years of activity/competition it was agreed that a standard should be developed (committee design)

The MPI Standard

- MPI Forum (1992) established to create a standard:
  - Source-code portability
  - Allow for efficient implementation (e.g., by vendors)
  - Support for heterogeneous platforms
  - C and FORTRAN bindings
- MPI is not:
  - A language
  - An implementation (but provides hints for implementers)
- June 1995: MPI v1.1 (we're now at MPI v3.0)
- http://www.mpi-forum.org
- Well-adopted by vendors
- Free implementations for clusters: MPICH, OpenMPI
- Tons of research versions/extensions (e.g., fault-tolerance)

SPMD

- It is rare for a programmer to write a different program for each process of a parallel application
- Single Program Multiple Data (SPMD) programs
  - The same program runs on all participating processors
  - Processes can be identified by some rank
  - Based on rank each process knows which piece of the problem to work on

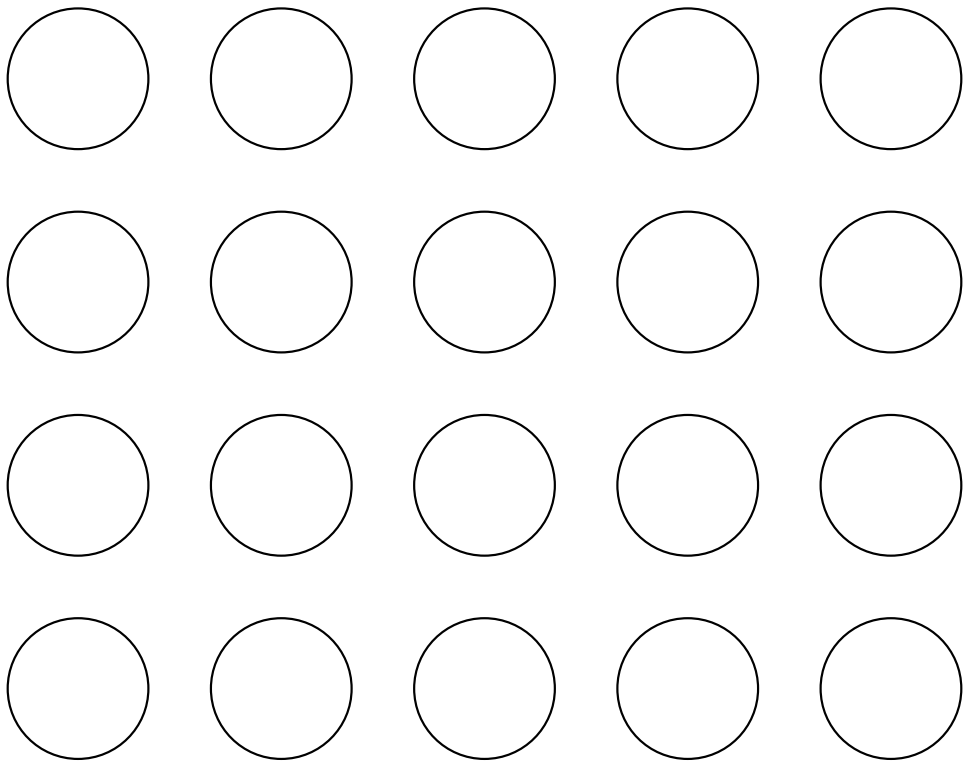
Master-Worker SPMD Program

```
main(int argc, char **argv) {
   if (my_rank == 0) { /* master */
     ... load input and dispatch ...
   } else { /* workers */
     ... wait for data and compute ...
   }
```

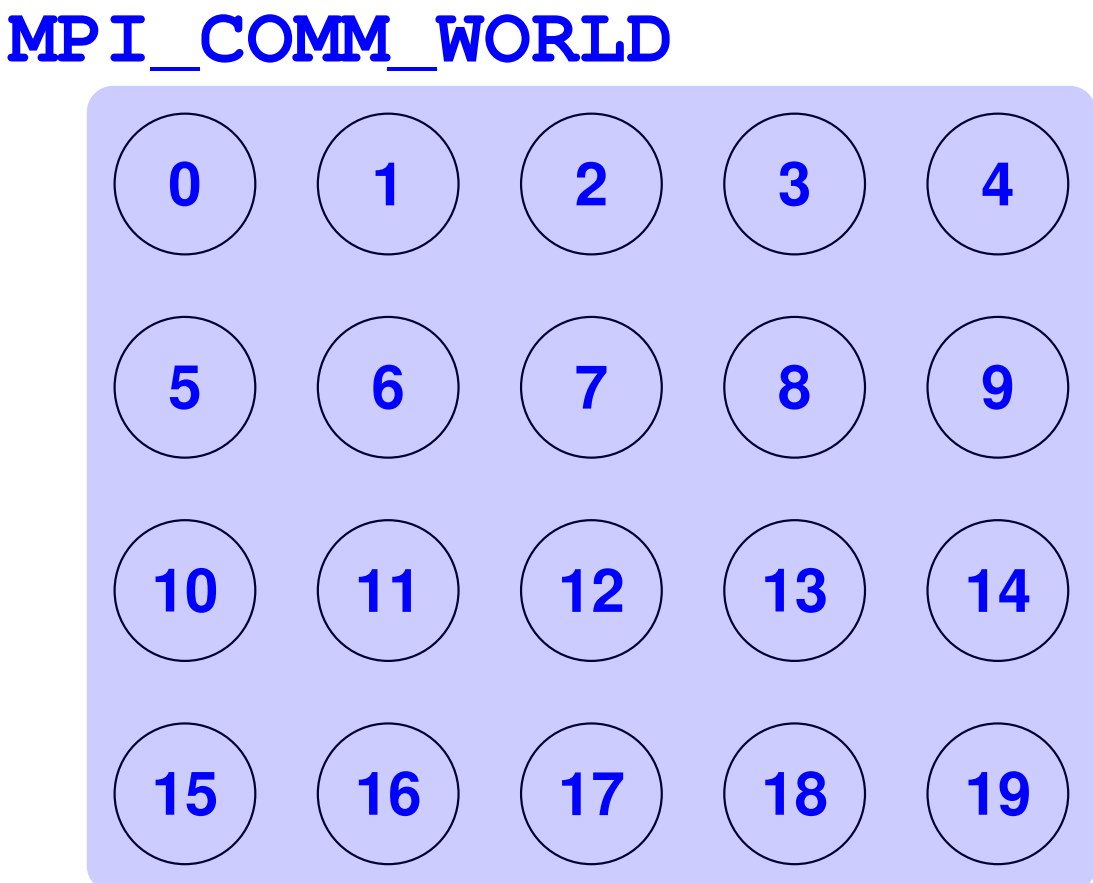
MPI Concepts

- Fixed number of processors
  - When launching the application one must specify the number of processors to use, which remains unchanged throughout execution
- Communicator
  - Abstraction for a group of processes that can communicate
  - A process can belong to multiple communicators
  - Makes is easy to partition/organize the application in multiple layers of communicating processes
  - Default and global communicator: `MPI_COMM_WORLD`
- Process Rank
  - The index of a process within a communicator

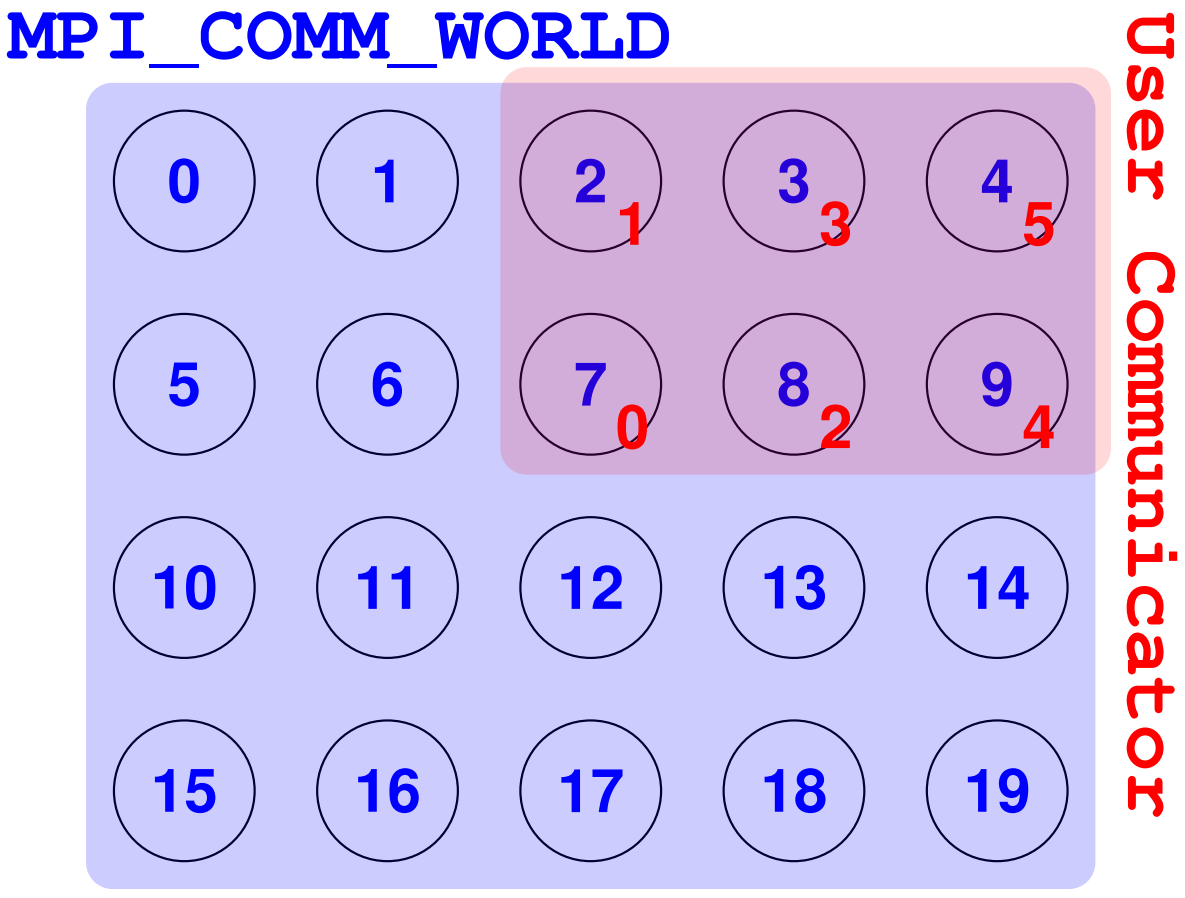
MPI Communicators



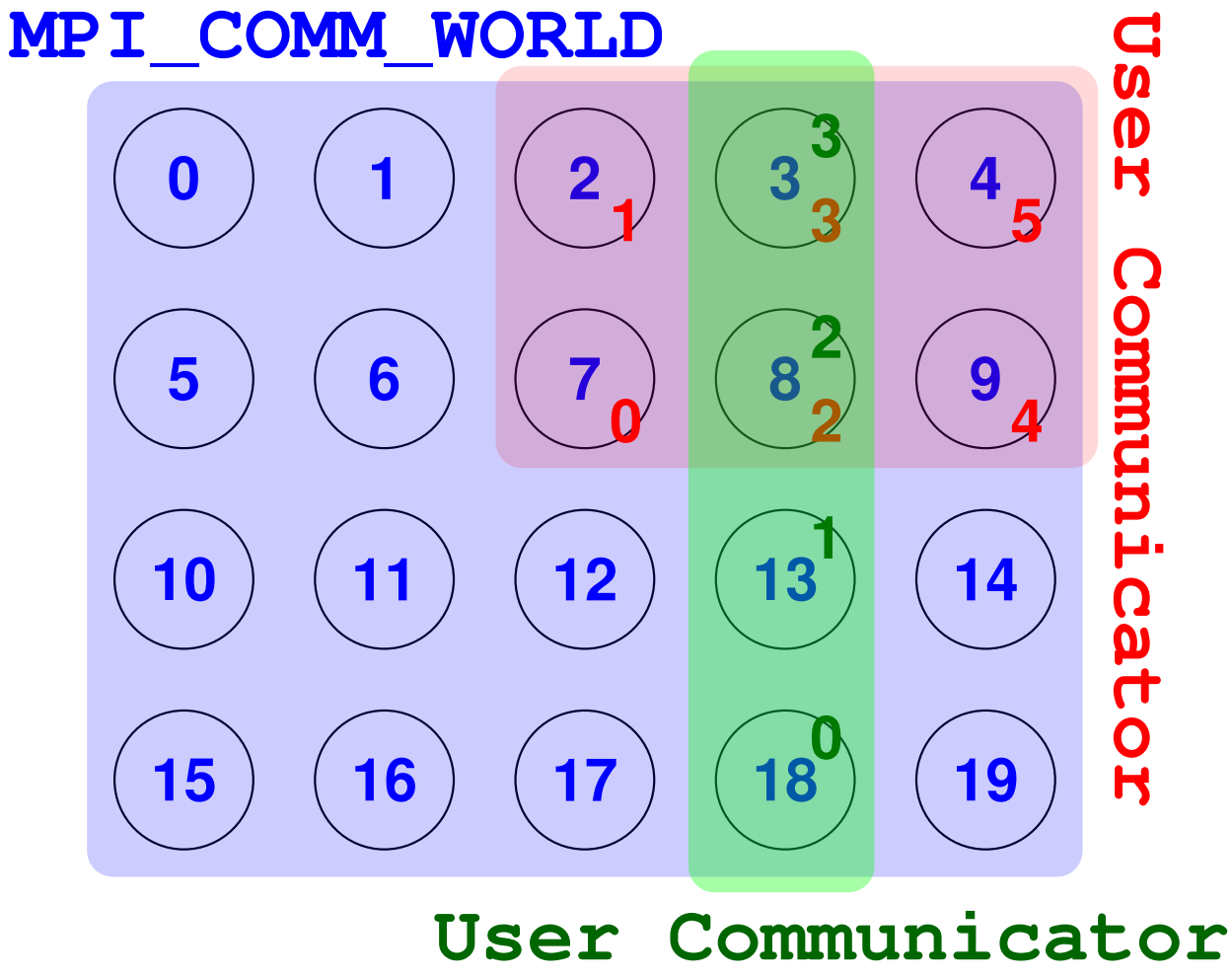
MPI Communicators



MPI Communicators



MPI Communicators



“Hello World” in MPI

```
#include <unistd.h>
#include <mpi.h>
int main(int argc, char **argv) {
  int my_rank, n;
  char hostname[128];
  int namelen;
  MPI_Init(&argc,&argv); // Called once
  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
  MPI_Comm_size(MPI_COMM_WORLD,&n);
  MPI_Get_processor_name(hostname,&namelen);
  if (my_rank == 0) { // Master
    printf("I am the master: %s\n",hostname);
  } else { // Worker
    printf("I am a worker: %s (rank=%d/%d) at time %lf\n",
           hostname,my_rank,n-1,MPI_Wtime());
  }
  MPI_Finalize();   // Call once
  exit(0);
}
```

Compiling/Running an MPI Program

Compiling

```
% mpicc my_program.c -o my_program
```

Running

```
% mpirun -np <num processes> -hostfile <hostfile> my_program  <program arguments>
```

- `<hostfile>`: the name/IP-address of one host on each line
- `<num processes>`: number of MPI processes to start
- See the man pages for all details

Previous example program:

```
% mpirun -np 3 -hostfile hosts my_program
  I am the master: somehost1
  I am a worker: somehost2 (rank=2/2)
  I am a worker: somehost3 (rank=1/2)
```

Mapping of Processes to Hosts: `mpirun`

- `mpirun` places MPI tasks/processes on hosts round-robin
  - Specifying more processes than hosts causes processes to double up on the host
- Say you have 4 hosts, each with 8 cores, to use everything you have two options:
- **Option #1:** Use 4 MPI processes, and each process uses OpenMP (or whatever) to use all 8 cores
- **Option #2:** Use 32 MPI processes
  - Typically not as good due, e.g., to overhead of extra communication (but MPI can be smart about inter-core, intra-host communication)
- You should not exceed RAM capacity on each host!

Mapping of Processes to Hosts: Batch Scheduler

- Batch schedulers “know” about MPI and you don’t pass arguments to `mpirun`
  - Our Cray cluster is managed using SLURM, a well-known popular scheduler
- SLURM job submission:

SLURM and MPI

```
#SBATCH -n xxx # Number of MPI tasks requested (think "-np" for mpirun)
#SBATCH -c xxx # Number of cores requested per task (think "omp_set_num_threads()")
#SBATCH -N xxx # Number of nodes requested (think "I want to be neatly packed onto nodes")
...
mpirun ./my_program arg1 arg2
```

- Our cluster has **20-core** nodes

SLURM/MPI Examples

- 40 MPI tasks, each on a core, over 2 20-core nodes:

```
#SBATCH -n 40 # Number of MPI tasks requested
#SBATCH -c 1  # Number of cores requested per task
#SBATCH -N 2  # Number of nodes requested
```

- 2 MPI tasks, each on 20 cores, over 2 20-core nodes:

```
#SBATCH -n 2  # Number of MPI tasks requested
#SBATCH -c 20 # Number of cores requested per task
#SBATCH -N 2  # Number of nodes requested
```

Different queues

- Our cluster has different queues, in particular:
  - **community.q**: May share node with other users (i.e., some cores on your nodes are used by somebody else)
  - **exclusive.q**: Garanteed dedicated nodes even if some cores are unused
  - **ics632.q**: Like exclusive.q, but we have higher priority (but only 4 nodes)
- **All MPI jobs should be submitted to exclusive.q**

SLURM and MPI

```
$#SBATCH -n xxx          # Number of MPI tasks requested (think "-np" for mpirun)
#SBATCH -c xxx           # Number of cores requested per task (think "omp_set_num_threads()")
#SBATCH -N xxx           # Number of nodes requested (think "I want to be neatly packed onto nodes")
#SBATCH -p exclusive.q # Queue ('p' stands for 'partition')
...
mpirun ./my_program arg1 arg2
```

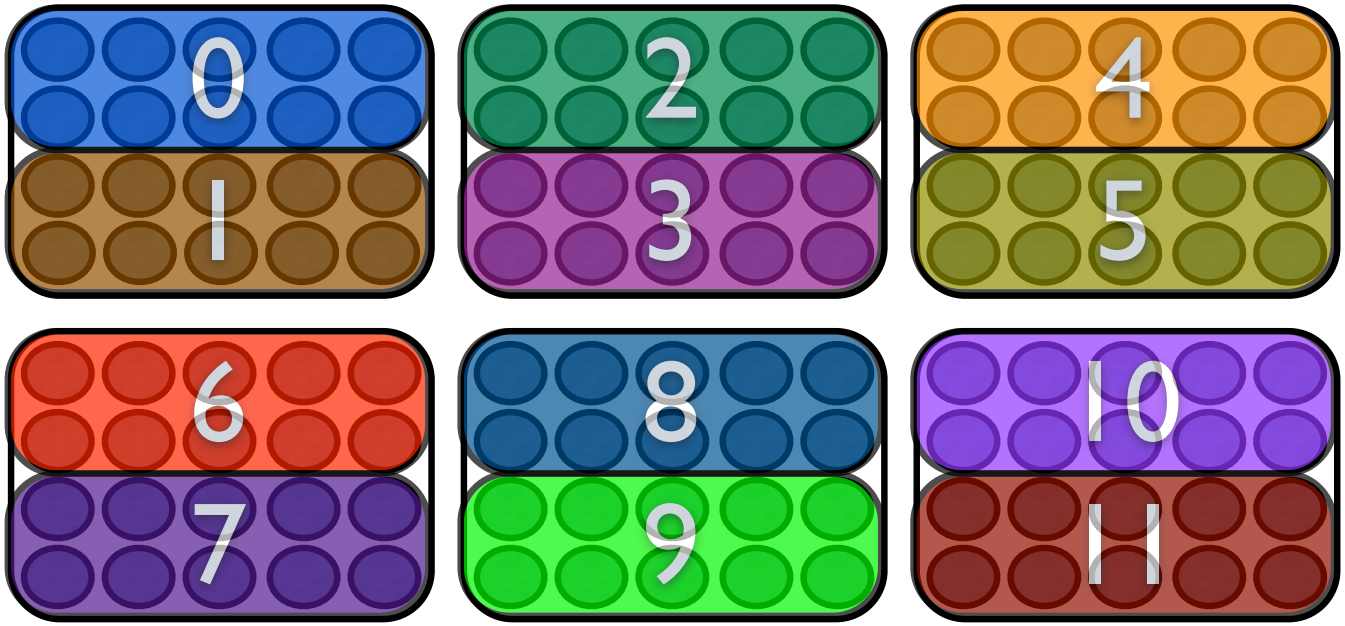
SLURM/MPI Quiz

```
#SBATCH -n 12            # Number of MPI tasks requested
#SBATCH -c 10            # Number of cores requested per task
#SBATCH -N 6             # Number of nodes requested
#SBATCH -p exclusive.q   # Queue
```

- What does the execution look like?

SLURM/MPI Quiz

```
#SBATCH -n 12           # Number of MPI tasks requested
#SBATCH -c 10           # Number of cores requested per task
#SBATCH -N 6            # Number of nodes requested
#SBATCH -p exclusive.q  # Queue
```



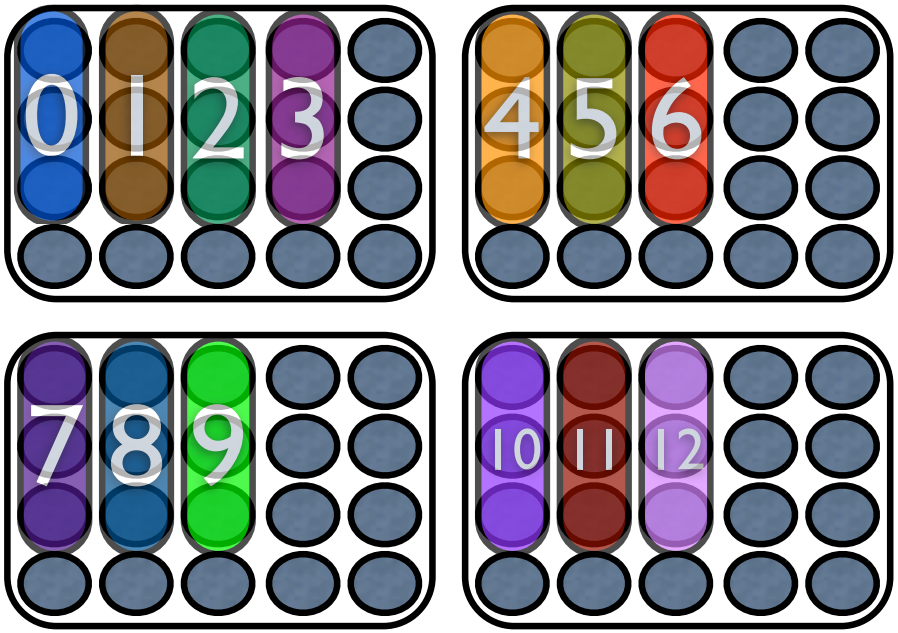
SLURM/MPI Quiz

```
#SBATCH -n 13          # Number of MPI tasks requested
#SBATCH -c 3           # Number of cores requested per task
#SBATCH -N 4           # Number of nodes requested
#SBATCH -p ics632.q    # Queue
```

- What does the execution look like?

SLURM/MPI Quiz

```
#SBATCH -n 13          # Number of MPI tasks requested
#SBATCH -c 3           # Number of cores requested per task
#SBATCH -N 4           # Number of nodes requested
#SBATCH -p ics632.q    # Queue
```



- Note that ranks are "contiguous"
- What about intra-node communications?
  - Some MPI implementations are very smart...

Is our MPI smart?

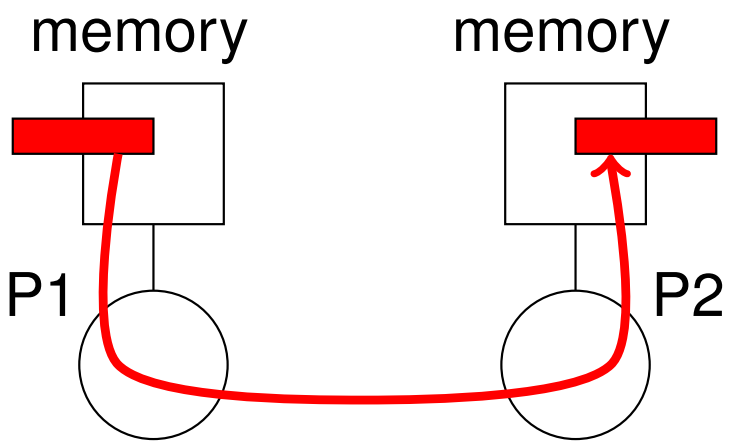
- Simple MPI program to pass along a message on a *logical* ring (or simply does a `memcpy`).

	Avg. elapsed time (sec)		
#bytes	1 node memcpy	1 node	20 nodes
10^4	0.000003	0.000042 (14x)	0.000223 (53x)
10^6	0.000318	0.001963 (6x)	0.006638 (3x)
10^8	0.086983	0.340864 (4x)	0.607999 (2x)
10^9	1.186134	3.567302 (3x)	7.393857 (2x)

- Going across the network is not terrible compared to being all within on node
- On 1 node we're $\sim$3x slower than memory bandwidth, so our MPI perhaps use `memcpy` with internal buffers?
  - We would have to dig deeper to truly figure it out (and look at physical characteristics of the machine)

Outline

Point-to-Point Communication



- Data to be communicated is described by:
  - address in memory of sender
  - data type of the elements in the message
  - length of the message in numbers of elements
- Involved processes are described by:
  - communicator
  - rank
- Message “tagged” (integer) as chosen by the user

Communication Modes

- Two modes of communication:
  - Synchronous: Communication does not complete until the message has been received
  - Non-blocking: Completes as soon as the message is "on its way," and hopefully it gets to destination
- MPI provides 4 versions regardless of the mode:
  - synchronous
  - buffered
  - standard
  - ready
- So we have $4 \times 2 = 8$ possibilities

Synchronous/Buffered sending in MPI

- Synchronous with `MPI_Ssend`
  - The send completes only once the receive has succeeded
  - Copy data to the network, wait for an ack
  - The sender has to wait for a receive to be posted
  - No buffering of data (but for the OS)

- Buffered with `MPI_Bsend`
  - The send completes once the message has been buffered internally by MPI
  - Buffering incurs an extra memory copy
  - Doe not require a matching receive to be posted
  - May cause buffer overflow if many Bsends and no matching receives have been posted yet

Standard/Ready Send

- Standard with `MPI_Send`
  - Up to MPI to decide whether to do synchronous or buffered, for performance reasons
  - The rationale is that a correct MPI program should not rely on buffering to ensure correct semantics
- Ready with `MPI_Rsend`
  - May be started only if the matching receive has been posted
  - Can be done efficiently on some systems as no hand-shaking is required

MPI_Recv

- There is only one `MPI_Recv`, which returns when the data has been received (specifies max number of elements)
- Why all this junk?
  - Performance, performance, performance
  - MPI was designed with constructors in mind, who would endlessly tune code to extract the best out of the platform (LINPACK benchmark)
  - Playing with the different versions of `MPI_?send` can improve performance without modifying program semantics
  - Playing with the different versions of `MPI_?send` can modify program semantics
  - Typically parallel codes do not face very complex distributed system problems and it's often more about performance than correctness.
  - You can play with these in your code

Simple Send-Receive MPI Program

```
#include <unistd.h>
#include <mpi.h>
int main(int argc, char **argv) {
  // IMPORTANT: each process has its own i, my_rank, nprocs, x
  int i, my_rank, nprocs, x[4];

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

  if (my_rank == 0) { // Master
    x[0]=42; x[1]=43; x[2]=44; x[3]=45;
    MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
    for (i=1;i<nprocs;i++)
      // send 4 ints at address x to process i in default communicator with tag 0
      MPI_Send(x,4,MPI_INT,i,0,MPI_COMM_WORLD);
  } else { // Worker
    MPI_Status status;
    // receive 4 ints at address x from process 0 in default communicator with tag 0
    MPI_Recv(x,4,MPI_INT,0,0,MPI_COMM_WORLD,&status);
  }

  MPI_Finalize();
  exit(0);
}
```

Possible Deadlocks

```
...
MPI_Ssend(...);
MPI_Recv(...);
...
```

DEADLOCK

```
...
MPI_Ssend(...);
MPI_Recv(...);
...
```

```
...
MPI_Buffer_attach(...);
MPI_Bsend(...);
MPI_Recv(...);
...
```

OK

```
...
MPI_Buffer_attach(...);
MPI_Bsend(...);
MPI_Recv(...);
...
```

```
...
MPI_Buffer_attach(...);
MPI_Bsend(...);
MPI_Recv(...);
...
```

OK

```
...
MPI_Ssend(...);
MPI_Recv(...);
...
```

What About MPI_Send?

```
...
MPI_Send(...);
MPI_Recv(...);
...
```

Deadlock behavior depends on message size

```
...
MPI_Send(...);
MPI_Recv(...);
...
```

- MPI implementations choose how `MPI_Send` is implemented based on message time
  - If message is small: buffered
  - If message is large: synchronous
- A program shouldn't rely on the particular sending behavior for correctness
- This is why we have asynchronous sends/recvs!

What About MPI_Send?

```
...
MPI_Send(...);
MPI_Recv(...);
...
```

Deadlock behavior depends on message size

```
...
MPI_Send(...);
MPI_Recv(...);
...
```

- MPI implementations choose how `MPI_Send` is implemented based on message time
  - If message is small: buffered
  - If message is large: synchronous
- A program shouldn't rely on the particular sending behavior for correctness
- This is why we have asynchronous sends/recvs!

Non-Blocking Communications

- So far we've seen blocking communication
  - The call returns whenever its operation is complete (`MPI_Ssend` returns once the message has been received, `MPI_Bsend` returns once the message has been buffered, etc.)
- MPI provides non-blocking communication: the call returns immediately and there is another call that can be used to check on completion.
- Rationale: Non-blocking calls let the sender/receiver do something useful while waiting for completion of the operation (without playing with threads, etc.)

Isend, Irecv

- MPI_Issend, MPI_Ibsend, MPI_Isend, MPI_Irsend, MPI_Irecv

```
MPI_Request request1=MPI_REQUEST_NULL, request2=MPI_REQUEST_NULL;
MPI_Isend(&x,1,MPI_INT,dest,tag,communicator,&request1);
MPI_Irecv(&x,1,MPI_INT,src,tag,communicator,&request2);
```

- MPI_Wait, MPI_Test, MPI_Waitany, MPI_Testany, MPI_Waitall, MPI_Testall, MPI_Waitsome, MPI_Testsome

```
MPI_Status status1, status2;
MPI_Test(&request2, &status2) // doesn't block
MPI_Wait(&request1, &status1) // block
```

Example with No Deadlock

```
#include <unistd.h>
#include <mpi.h>
int main(int argc, char **argv) {
  int i, my_rank, x, y;
  MPI_Status status;
  MPI_Request request=MPI_REQUEST_NULL;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);

  if (my_rank == 0) { // P0
    x=42;
    MPI_Isend(&x,1,MPI_INT,1,0,MPI_COMM_WORLD,&request);
    MPI_Recv(&y,1,MPI_INT,1,0,MPI_COMM_WORLD,&status);
    MPI_Wait(&request,&status);
 } else if (my_rank == 1) { // P1
    y=41;
    MPI_Isend(&y,1,MPI_INT,0,0,MPI_COMM_WORLD,&request);
    MPI_Recv(&x,1,MPI_INT,0,0,MPI_COMM_WORLD,&status);
    MPI_Wait(&request,&status);
  }

  MPI_Finalize();
  exit(0);
}
```

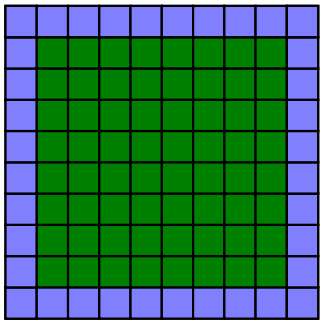
Usefulness of Non-Blocking Communication

- In our example, why not swap sends and receives?
  - Example: A logical linear array of N processors, needing to exchange data with their neighbor at each iteration of an application
  - One would need to orchestrate the communications: all odd-numbered processors send first, all even-numbered processors receive first
  - Sort of cumbersome and can lead to complicated patterns for more complex examples
  - Just use `MPI_Isend` and write much simpler code
- Using `MPI_Isend` makes it possible to overlap useful work with communication delays

```
MPI_Isend()
<useful work>
MPI_Wait()
```

Iterative Application Example

```
for (iteration)
  update my cells
  send boundary values to neighbors
  receive boundary values from neighbors
```



- Would deadlock with `MPI_Ssend`, maybe with `MPI_Send`
- Better version with non-blocking communication (no deadlock, latency hiding):

```
for (iterations)
  initiate sending of boundary values to neighbors
  initiate receipt of boundary values from neighbors
  update non-boundary cells
  await completion of sending of boundary values
  await completion of receipt of boundary values
  update boundary cells
```

More on Point-to-Point Communication

- Message ordering is guaranteed between each pair of processes
- There are many more functions that allow fine control of point-to-point communication
- Detailed API descriptions: http://www.mcs.anl.gov/research/projects/mpi/
  - Note that you should check error codes, etc.
- Tons of on-line MPI material

Outline

Multiple Processes Communicating

- A single operation in which multiple processes communicate is called a collective communication
- Examples: One-to-All (broadcast), All-to-One (gather), All-to-All, Synchronization (barrier), Reduction
- All these can be built using point-to-point communications, but typical MPI implementations have optimized them, and it's a good idea to use them
  - Optimized by a vendor to their interconnect
  - Optimized in open implementations given commonplace interconnect properties
- In all of these, all processes place the same call (in good SPMD fashion), although depending on the process, some arguments may not be used

Barrier

- Synchronization of the calling processes
- The call blocks until all of the processes have placed it
- No data is exchanged

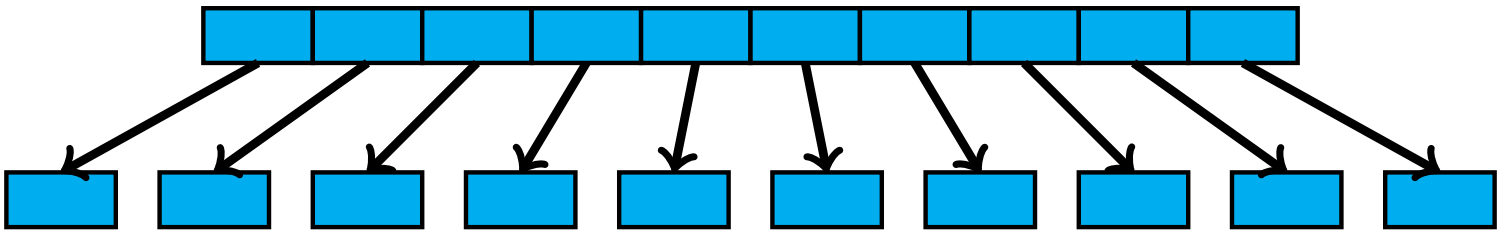
```
...
MPI_Barrier(MPI_COMM_WORLD);
...
```

Broadcast

- One-to-many communication
  - The source process is called the *root*
  - Multicast (One-to-some communication) can be implemented by using communicators
- All processes place the same call (typical of MPI)

```
...
// Process 0 sends 4 ints at address x to all processes
// (semantically, also to itself)
MPI_Bcast(x, 4, MPI_INT, 0, MPI_COMM_WORLD);
...
```

Scatter



```
...
// x is an array with N*100 elements
//   Process 0 sends x[0:99] to process 0, x[100:199] to process 1, x[200:299] to process 2, etc.
// y is an array with 100 elements
//   Process 0 stores what it receives in y[0:99], process 1 in y[0:99], process 2 in y[0:99], etc.
//
MPI_Scatter(x, 100, MPI_INT, y, 100, MPI_INT, 0, MPI_COMM_WORLD)
...
```

- The first 3 arguments are meaningful only at the root

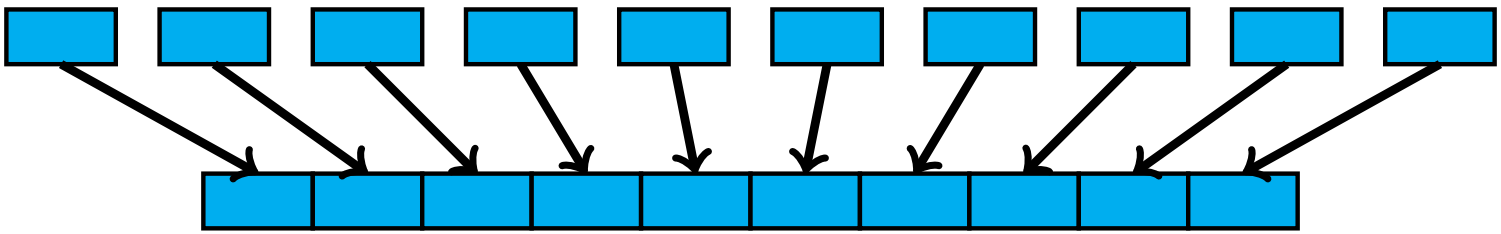
Scatter Example

```
int main(int argc,char **argv) {
  int *a;
  int *recvbuffer;

  ...
  MPI_Comm_size(MPI_COMM_WORLD,&n);
  <allocate destination array recvbuffer of size N/n>

  if (my_rank == 0) { // root
    <allocate source array a of size N>
  }

  MPI_Scatter(a, N/n, MPI_INT, recvbuffer, N/n, MPI_INT, 0, MPI_COMM_WORLD);
  ...
}
```

Scatter Example with No Redundant Send

```
int main(int argc,char **argv) {
  int *recvbuffer;

  ...
  MPI_Comm_size(MPI_COMM_WORLD,&n);
  if (my_rank == 0) { // root
    int *a;

    <allocate array a of size N>
    <allocate array recvbuffer of size N/n>
    MPI_Scatter(a, N/n, MPI_INT, MPI_IN_PLACE, N/n, MPI_INT, 0, MPI_COMM_WORLD);

  } else { // others

    <allocate array recvbuffer of size N/n>
    MPI_Scatter(NULL, 0, MPI_INT, recvbuffer, N/n, MPI_INT, 0, MPI_COMM_WORLD);

  }
  ...
}
```

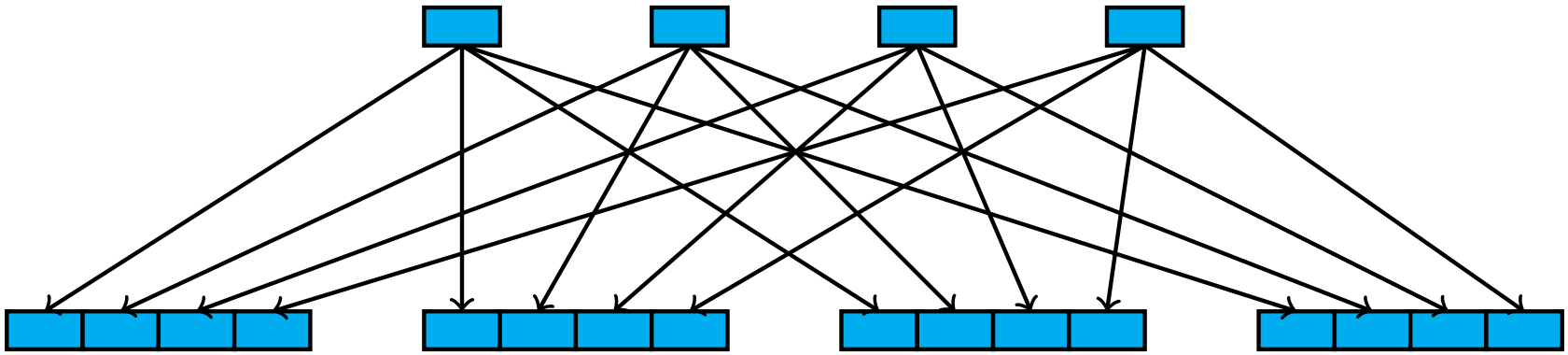
Gather



```
...
// x is an array with 100 elements
//   Process 0 sends x[0:99] to process 0, process 1 sends x[100:199] to process 0, process x[200:299] to
        process 0, etc.
// y is an array with N*100 elements
//   Process 0 stores what it receives from process 0 in y[0:99], from process 1 in y[100:199], from process 2
        in y[200:299], etc.
//
MPI_Gather(x, 100, MPI_INT, y, 100, MPI_INT, 0, MPI_COMM_WORLD)
...
```

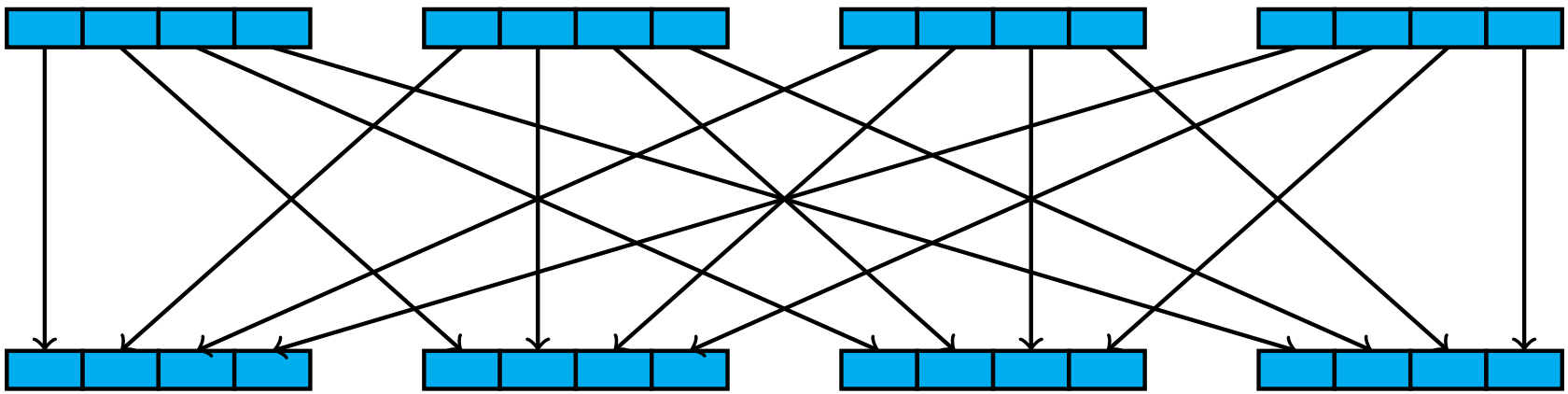
- The 4th to 6th arguments are meaningful only at the root

Gather to All



```
...
// x is an array with 100 elements
//   Process i sends x[0:99] to process j=1,...,N
// y is an array with N*100 elements
//   Process i stores what it receives from process j in y[i*100:i*100+99]
//
MPI_Allgather(x, 100, MPI_INT, y, 100, MPI_INT, MPI_COMM_WORLD)
...
```

All to All



```
...
// block i from process j goes to block j on process i
//
MPI_Alltoall(x, 100, MPI_INT, y, 100, MPI_INT, MPI_COMM_WORLD)
...
```

Reduction Operations

- Used to compute a result from data that is distributed among processors
- Often what a user wants to do anyway, e.g., compute the sum of a distributed array
- So why not provide the functionality as a single API call rather than having people keep re-implementing the same things?
- Predefined operations:
  - `MPI_MAX`, `MPI_MIN`, `MPI_SUM`, etc.
- Possible to have user-defined operations
- Must be associative and commutative

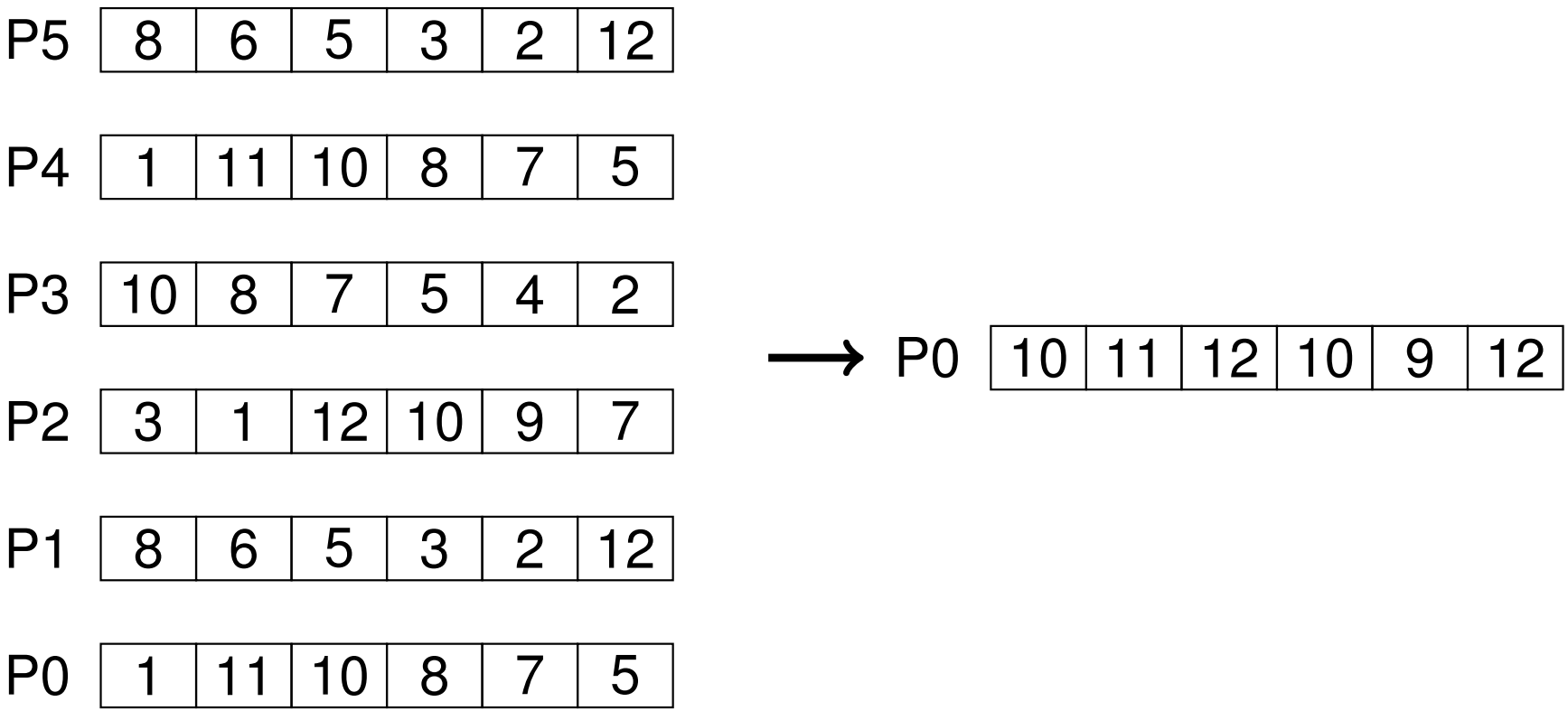
Reduce, All-reduce

- `MPI_Reduce`: Result is collected at the root
  - The operation is applied element-wise for each element of the input arrays on each processor
    - including the root
  - An output array is returned
- `MPI_Allreduce`: result is sent out to everyone

```
// root: process 0; x: input array; r: output array
MPI_Reduce(x, r, 10, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)
```

```
// x: input array; r: output array
MPI_Allreduce(x, r, 10, MPI_INT, MPI_SUM, MPI_COMM_WORLD)
```

Reduction Example: Maximum

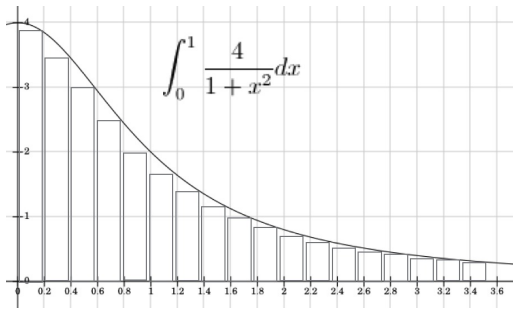


More Collectives...

- `MPI_Scan`: Prefix reduction
- Most broadcast operations come with a version that allows for a stride (so that blocks do not need to be contiguous)
  - `MPI_Gatherv()`, `MPI_Scatterv()`, `MPI_Allgatherv()`, `MPI_Alltoallv()`
- `MPI_Reduce_scatter()`: functionality equivalent to a reduce followed by a scatter
- All the above have been created because they occur in scientific applications and save user effort
- All details at: http://www.mcs.anl.gov/mpi/

Example: Computing π

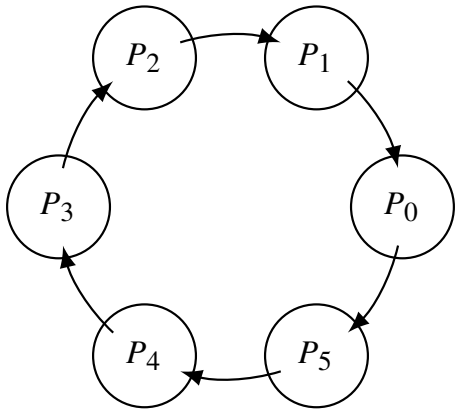
```
int n; // Number of rectangles
int nprocs, myrank;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
MPI_Comm_size(MPI_COMM_WORLD,&nprocs);

if (my_rank == 0)
  read_from_keyboard(&n);

// Broadcast number of rectangles from root process to everybody else
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

// Compute my part of the integral
mypi = integral((n/nprocs) * my_rank,
                (n/nprocs) * (1+my_rank) - 1)

// Sum mypi across all processes, storing result as pi on root process
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```



Outline

Implementing Collectives

- MPI implements collectives for us, but it's useful to think about how to implement them
  - Good idea to implement them "by hand" in some algorithms
- Collective communications have been studied formally in idealized models
  - How well the models match reality is a huge topic we'll talk about later in the semester if time
- Let's consider a ring topology:



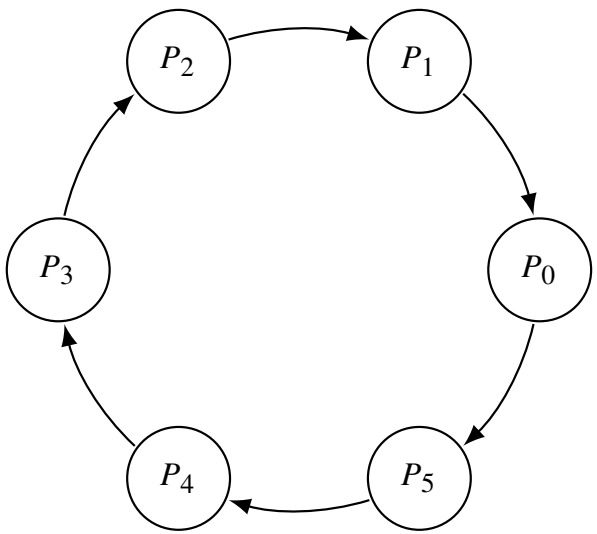
Ring Communication Assumption

- Let us pick one set of possible assumptions
- Message delay assumption:
  - $time(m)$: time to send a message of size m
  - $time(m) = \alpha + m\beta$
    - α: latency
    - β: time to transfer 1 byte (inverse of the data transfer rate)
- Let us assume that a process can receive, send, and compute at the same time
  - Provided there are no race conditions
  - Implemented easily with non-blocking communication
- In our pseudo-code we use "$||$" to indicate concurrent activities
- Let us assume blocking receives

Broadcast on a Ring

- Let's write a program that has P_k send the same message of length m to all other processes
- On the (unidirectional) ring a broadcast is straightforward:

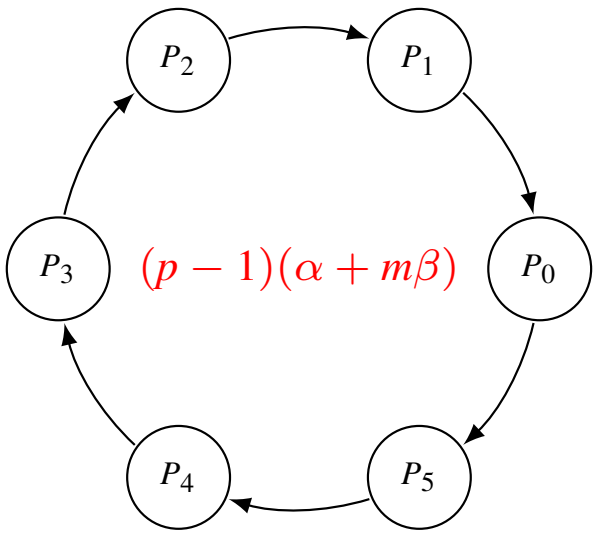
```
q = my_rank()
p = num_procs()
if (q == k)
   SEND(addr,m)
else
   if (q == k-1 mod p)
      RECV(addr,m)
   else
      RECV(addr,m)
      SEND(addr,m)
   endif
endif
```



Broadcast on a Ring

- Let's write a program that has P_k send the same message of length m to all other processes
- On the (unidirectional) ring a broadcast is straightforward:

```
q = my_rank()
p = num_procs()
if (q == k)
   SEND(addr,m)
else
   if (q == k-1 mod p)
      RECV(addr,m)
   else
      RECV(addr,m)
      SEND(addr,m)
   endif
endif
```



A Faster Broadcast

- How can we improve performance?
- One can cut the message in many small pieces, say in r pieces where m is divisible by r
- The root process just sends r messages
- The wall-clock time is as follows:
  - Consider the last processor to get the last piece of the message
  - There needs to be $p-1$ steps for the first piece to arrive, which takes time $(p-1)(\alpha+\beta m/r)$
  - Then the remaining $r-1$ pieces arrive one after another, which takes time $(r-1)(\alpha+\beta m/r)$
  - For a total of: $(p-2+r)(\alpha+\beta m/r)$

Fastest Broadcast?

- Wall-clock time: $(p-2+r)(\alpha+\beta m/r)$
- Question: What value of r minimizes the wall-clock time?
- Answer: $\sqrt{m(p-2)\beta/\alpha}$
  - Take the derivative and find the zero
- The optimal wall-clock time is: $(\sqrt{(p-2)\alpha}+\sqrt{m\beta})^2$
- When m is large, this tends to $m\beta$
- This means that for a large message, cutting it into many pieces makes is possible to achieve close to the maximum one-hop data transfer rate

Splitting Messages into Chunks

- We have “rediscovered” a fundamental principle of network protocols
  - Cut messages into frames to maximize throughput on multi-hop network paths
- Our original, inefficient broadcast implementation uses a “store-and-forward” strategy that does not exploit the network since ultimately network paths are multi-hop
- In the real world, collective communication run on top of network protocols, and most of these protocols cut messages into frames
  - This is the case of, for instance, in SimGrid MPI simulations since simulated networks are TCP + GigaBit Ethernet
- So in the end, you can cut messages into pieces yourself when writing the code, and the underlying network protocol may cut it into pieces as well

Scatter/All-to-All on a Ring

- Implementing other collectives on a ring can be done like for a broadcast by sending data along the ring
- Scatter:
  - The root first sends the message for the most distant recipient, then for the second most distant recipient, etc.
  - Each message is relayed by each processor until the final recipient is reached
  - Communications are thus pipelined along the ring
- All-to-all:
  - Each process first sends the message for its most distant recipient, then for its second most distant recipient, etc.
  - The code is a bit more complex, but the same pipelining principle applies

Now What?

- Now we can do fast collectives on a ring
- If our physical network is a ring, then we're done
- Unfortunately, our physical network is likely not a ring
- A ring could be embedded in our physical platform
  - Find a one-to-one vertex mapping between a logical graphs and a physical graphs so that no two logical edges share a physical edge and so that path lengths are conserved
  - e.g., A ring can be embedded in any torus
  - e.g., A torus can be embedded into a hypercube provided some dimensions are powers of 2
- Or we can acknowledge that our platform is not a ring but try the ring-based broadcast anyway
- We explore these issues in the programming assignment

Outline

Increasing Memory

- One reason to go distributed-memory is limited memory
- Example: I want to sort a 10GiB array using 10 computers each with 1GiB of memory
- Question: how do I write the code?
  - I cannot declare: `char array[10*1024*1024*1024]`
- Solution: keep the data distributed
- Since each node gets only 1/10th of the array, each node declares only an array on 1/10th of the size
  - each process: `char array[SIZE/10];`
- When processor 0 references `array[0]` it means the first element of the global array
- When processor i references `array[0]` it means element at index (SIZE/10*i) in the global array

Global vs. Local Indices

- There is a mapping from/to local indices and global indices
- It can be a mental gymnastic that requires some potentially complex arithmetic expressions for indices
- Often one writes functions to do this
  - e.g., global2local()
  - When you would write `a[i] * b[k]` for the sequential version of the code, you should write `a[global2local(i)]*b[global2local(k)]`
  - This may become necessary when index computations become too complicated
- More on this when we see some parallel algorithms

MPI Data Types

- MPI allows users to define custom data types that can be used for sending data that is not simply contiguous arrays of scalar types
  - Strided arrays, data structures, sets of variables, etc.
- Goal: not force users to pack data into temporary arrays, which is cumbersome and slow
- There are lots of features here, so let's just look at one example
  - See http://www.mcs.anl.gov/mpi/ for all details

MPI Data Type Example

```
int MPI_Type_vector(int count, int blocklength, int stride   MPI_Datatype oldtype,
      MPI_Datatype *newtype)
```

Sending the 5th Column of a Matrix

```
double results[M][N];
MPI_Datatype newtype;
MPI_Type_vector(M, 1, N, MPI_DOUBLE, &newtype);
MPI_Type_commit(&newtype);
MPI_Send(&(results[0][4]), 1, newtype, dest, tag, comm);
```

MPI-2, MPI-3

- Remote memory (put/get) to take advantage of hardware
- Parallel I/O
- Dynamic processes with growing/shrinking communicators
- Thread support
- More collective communications
- Inter-language operation, C++ bindings
- Socket-style communication for client-server
- Non-blocking collectives
- Topology graphs
- More communicator functions
- ...

Conclusion

- MPI is a decent API (very consistent)
- Many argue that MPI programing is too hard and many higher-level abstractions have been proposed
- But MPI remains the workhorse of distributed memory programming, for now