Overview

- Today’s topic: communication patterns that involve all processes in an application
  - broadcast (one to all)
  - scatter (one to all) and gather (all to one)
  - reduce (all to one)
  - synchronization

- Techniques for avoiding deadlock

- Application using collective communication: Jacobi iteration
  (method for solving system of linear equations)

- Reading:
  - Wilkinson & Allen: parts of Ch 2 (pp 49-61), Ch 6 (synchronization)
  - MPI documentation

Broadcast

- A message pattern known as “broadcast” has one process send the same information to all other processes

  ![Broadcast diagram]

- The MPI function:
  ```c
  MPI_Bcast(buf,count,type,root,comm)
  ```
  - `buf`: the address of the data to communicate
  - `count`: number of items
  - `type`: type of items
  - `root`: id of the process that will send the data

Broadcast (cont’d)

- An example from a previous lecture:
  ```c
  void broadcast(char *s)
  {
      int n = strlen(s) + 1;
      MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
      MPI_Bcast(s,n,MPI_CHAR,0,MPI_COMM_WORLD);
  }
  
broadcast("hello");
  ```

- Important note: all processes execute the function
Implementing Broadcast

- The abstract view: message is copied to all receivers simultaneously
- Full parallel implementation would require low-level interaction with host system hardware
- Example: cluster using ethernet
  - ethernet hardware protocol supports (is actually defined by) broadcast
  - software layers (TCP/IP) do point-to-point communication
  - fancier switches may implement broadcast

```
switch
 Myrinet, others
```

Linear Implementation of Broadcast

- In a straightforward implementation the root sends one message to all other processes, e.g. if root is process 0:
  ```
  if (rank == 0)
  for (i = 1; i < np; i++)
    MPI_Send(...)
  ```
- Other processes just wait for the message:
  ```
  MPI_Recv(...)
  ```
- Requires $O(n)$ sends from root process

Tree Implementation of Broadcast

- An alternative implementation would have receivers forward information to other processes
  - 0 sends to 1
  - 0 and 1 send to 2 and 3
  - 0..3 send to 4..7
- Requires $O(\log n)$ sends from process 0

Aside: Hypercubes

- A good way to visualize the tree-based broadcast is to see how it is done on a hypercube
- An $n$-dimensional hypercube is defined inductively
  - A 1D “cube” has two nodes, labeled 0 and 1
  - An $n$-dimensional cube is has two copies of an $(n-1)$-dimensional cube, with connections between corresponding nodes
  - To make labels: prefix with 0 in one copy, 1 in the other copy
Hypercubes (cont’d)

- A 3-D cube:

  ![3-D Hypercube Diagram]

- Some facts (apparent from the inductive construction):
  - $n$-dimensional cube has $2^n$ nodes
  - each node has $n$ neighbors ("degree $n$")
  - longest path traverses $n$ links

Scatter / Gather

- Two operations related to broadcast are named *scatter* and *gather*
  - scatter: send elements of an array to other processes
  - gather: collect elements of an array from other processes

Scatter / Gather in MPI

- The MPI functions are `MPI_Scatter` and `MPI_Gather`

```c
char ch;
char *s;
if (rank == 0) {
  s = "hello";
}
MPI_Scatter(s, 1, MPI_CHAR,
            &ch, 1, MPI_CHAR, 0, MPI_COMM_WORLD);
```

- Sender buffer, number to send, and type
- Receiver buffer, size, and type
- Sender ID ("root")
Scatter / Gather in MPI

```c
char ch;
char *s;
MPI_Scatter(s, 1, MPI_CHAR,
    &ch, 1, MPI_CHAR, 0, MPI_COMM_WORLD);
```

```
  s  h  ch  e  l  ch  l
   h e l l
```

process 0 1 2 3

Scatter / Gather Notes
- Call to MPI_Scatter or MPI_Gather is executed by all processes
- Root also participates in the communication
  - Note "h" sent to `ch` on process 0 in previous example
- For scatter, send params (buffer address, etc) are ignored by all processes except root
  - For gather, receive params ignored by all but root
- Types and counts must be compatible
- Target buffer locations should be written at most once
- See MPI Report for details...

Reduce
- A very common operation is known as reduction
- Similar to gather, except values sent to root are combined with an arithmetic or logical operation
- Example: compute the total number of black pixels in a Mandelbrot image
  - each process does a local count, stores result in `lcount`
  - all processes call the reduce function, specifying `lcount` as input parameter and `add` as the operation
  - after the call one process will have the sum of counts from all processes

Reduce in MPI
- The MPI function is MPI_Reduce
  ```c
  int i = rank;
  int n;
  MPI_Reduce(&i, &n, 1, MPI_INT,
     MPI_SUM, 0, MPI_COMM_WORLD);
  ```

  Address of value sent to root
  Where root will find result
  Number and type to combine
  Combining operation
  ```c
  root process ID
  ```
Reduce in MPI (cont’d)

- The MPI function is `MPI_Reduce`
  ```cpp
  int i = rank;
  int n;
  MPI_Reduce(&i, &n, 1, MPI_INT, 
  MPI_SUM, 0, MPI_COMM_WORLD);
  cout << rank << ": " << n << endl;
  ```

- Notes:
  - the call is executed by all processes
  - `i` and `n` are declared on all processes
  - final value of `n` is well-defined only for the root process (process 0 here)

Reduce in MPI (cont’d)

- Why do all processes have to allocate a buffer for the result?
  - Reduce (and scatter, gather, and other collective operations) can be implemented in a tree-like pattern
    - processes on back plane send value to front, where it is combined
    - processes on lower row send value up, where it is combined again
    - process on right sends value left for final combination
  - Intermediate steps require space for partial results...

Other Collective Operations

- MPI has many other group operations and variations on those presented here
  - all-to-all communication
    - `MPI_Alltoall`, `MPI_Allgather`
  - Scattering and gathering vectors
    - `MPI_Scatterv`, `MPI_Gatherv`
  - Scan (aka prefix or partial reduction), variation on reduce
    - `MPI_Scan`
      - topic for future lecture: clever dynamic task allocation scheme that uses scan to locate excess tasks

Reduce in MPI (cont’d)

- Reduce can apply binary associative operators (add, multiply, min, max, boolean and, ...)
  - also accepts user-defined operations

- What happens if the size parameter is greater than 1?
  ```cpp
  MPI_Reduce(x, y, 3, MPI_INT, 
  MPI_SUM, 0, MPI_COMM_WORLD);
  ```
  - are local x arrays combined first? or is the result an array of pair-wise combined values?

- See MPI Report for more information
**Barrier**

- A barrier is a method for synchronizing processes
  - all processes call the barrier function
  - a process is blocked until every other process reaches the barrier
  - all processes are then unblocked and proceed in parallel again
- No data is exchanged
- In MPI:
  - `MPI_Barrier(comm)`
    - the only parameter is the communicator ID (e.g. `MPI_COMM_WORLD`)

**Barrier (cont’d)**

- When do you need a barrier?
  - Scatter, gather, reduce, etc are implied barriers
    - all processes in the communicator cooperate to exchange information
  - Places where you might need a barrier:
    - organizing print statements: make sure all messages printed before the group proceeds
      - same for graphics or other output
    - when an application uses communicators that don’t involve all the processes
      - scatter, etc will synchronize processes in that communicator, but not all processes in `MPI_COMM_WORLD`

**Barrier Implementation**

- A straightforward implementation uses send and receive primitives
- Select one process to be the controller
  - all processes send an empty message to the controller, then wait for a response
  - controller maintains a counter
  - after controller receives a message from all processes it sends a reply (another empty message) to each process
- Can use a linear communication pattern or a tree-based pattern

**Butterfly**

- A process interconnection pattern known as a “butterfly” can be used for synchronization
- Butterfly switch (e.g. connect processors to memories in an SMP):
  - BBN Butterfly
  - up to 256 processors (ca. 1985)
Butterfly Topology for Barrier

- Use $\log(n)$ steps to synchronize $n$ processes
  - On each step a process exchanges a message with one other process
  - Process inverts bit $i$ of its ID to determine ID of its partner on step $i$
  - e.g. process 2 (010) exchanges with:
    - 3 (011)
    - 0 (000)
    - 6 (110)

Butterfly Barrier (cont’d)

- After each step larger groups will be synchronized
  - Step 1:
    - (0,1) (2,3) (4,5) (6,7)
  - Step 2:
    - (0,1,2,3) (4,5,6,7)
  - Step 3:
    - (0,1,2,3,4,5,6,7)
  - Advantage: eliminates the “release” phase of tree implementation

Butterfly Barrier (cont’d)

- Test your understanding:
  - what happens if one process is very far behind the others? will this scheme delay all others until the straggler catches up?
  - can this topology be used for other collective operations?
    - broadcast
    - reduction

Deadlock

- Operations that rely on exchanging messages between pairs of processes need to be careful to avoid deadlock
  - A set of two or more communicating processes may deadlock if
    - there is a cycle in the communication pattern
    - processes block until a message arrives
  - Example:
    ```
    process 0 process 1
    MPI_Send(&x,...,1) MPI_Send(&x,...,0)
    MPI_Recv(&x,...,0) MPI_Recv(&x,...,1)
    ```
Techniques for Avoiding Deadlock
- Put sends and receives in different order
- Example (for ring of processes): processes with odd IDs execute send before receive, even IDs do the opposite

```c
if (rank % 2) {
    MPI_Send(...);
    MPI_Recv(...);
} else {
    MPI_Recv(...);
    MPI_Send(...);
}
```

Avoiding Deadlock (cont’d)
- Use asynchronous sends
  - sender does not block when message sent
- Use MPI_Sendrecv
  - calling process sends one message and receives another
  - atomic operation wrt calling process
  - parameters from both MPI_Send and MPI_Recv:
    ```c
    MPI_Sendrecv(sbuf, scnt, stype, destid, stag,
    rbuf, rcnt, rtype, srcid, rtag, comm, status)
    ```

Application: Jacobi Iteration
- See Wilkinson and Allen § 6.3
- Many scientific applications need to solve systems of linear equations
- Find values of \( x \) that simultaneously satisfy
  \[
  a_{0,0}x_0 + a_{0,1}x_1 + \ldots + a_{0,n-1}x_{n-1} = b_0 \\
  a_{1,0}x_0 + a_{1,1}x_1 + \ldots + a_{1,n-1}x_{n-1} = b_1 \\
  \vdots \\
  a_{n-1,0}x_0 + a_{n-1,1}x_1 + \ldots + a_{n-1,n-1}x_{n-1} = b_{n-1}
  \]
  \( (n \text{ equations with } n \text{ unknowns}) \)

Jacobi Iteration (cont’d)
- Rewrite each row so row \( i \) has only \( x_i \) on the left side:
  \[
  x_i = \left( b_i - \sum_{j \neq i} a_{i,j}x_j \right) / a_{i,i}
  \]
- The new form can be viewed as a way of estimating \( x_i \) as a function of all the other \( x \)'s
- Iteration:
  - start with an initial guess for each \( x \)
  - apply the formulas to update the estimates
  - repeat until the values converge (i.e. successive rounds make little or no changes to each \( x \))
Parallel Iteration with MPI

- Use \( n \) processes (one per unknown)
- Each process initializes its \( x \)
- Iterate:
  - use `MPI_Allgather` to collect \( x_j \) from each process
  - update estimate of \( x \)
  - use `MPI_Reduce` to sum total change in \( x \)’s
  - stop when processes converge