Group Operations in MPI

Collective Communication
Synchronization

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

The Message-Passing Model
- A process is a program counter and address space
- Processes may have multiple threads (program counters and associated stacks) sharing a single address space
- MPI is for communication among processes (not threads)
- Interprocess communication consists of
  - Synchronization
  - Data movement

Message Passing Programming
- Defined by communication requirements
  - Data communication
  - Control communication
- Program behavior determined by communication patterns
- Message passing infrastructure attempts to support the forms of communication most often used or desired
  - Basic forms provide functional access
    - Can be used most often
  - Complex form provide higher-level abstractions
    - Serve as basis for extension
  - Extensions for greater programming power
Cooperative Operations for Communication

- Data is cooperatively exchanged in message-passing
- Explicitly sent by one process and received by another
- Advantage of local control of memory
  - Any change in the receiving process’s memory is made with the receiver’s explicit participation
- Communication and synchronization are combined

```
Process 0          Process 1
  Send(data)       Receive(data)
```

time

What is MPI (Message Passing Interface)?

- Message-passing library (interface) specification
  - Extended message-passing model
  - Not a language or compiler specification
  - Not a specific implementation or product
- Targeted for parallel computers, clusters, and NOWs
- Specified in C, C++, Fortran 77, F90
- Full-featured and robust
- Designed to access to advanced parallel hardware
  - End users
  - Library writers
  - Tool developers

Some Basic Concepts

- Processes can be collected into groups
- Each message is sent in a context
  - Must be received in the same context
- A group and context together form a communicator
- A process is identified by its rank
  - With respect to the group associated with a communicator
- There is a default communicator `MPI_COMM_WORLD`
  - Contains all initial processes

MPI Datatypes

- Message data (sent or received) is described by a triple
  - address, count, datatype
- An MPI datatype is recursively defined as:
  - Predefined data type from the language
  - 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
  - Array of (int, float) pairs
  - Row of a matrix stored columnwise
MPI Tags

- Messages are sent with an accompanying user-defined integer tag
  - Assist the receiving process in identifying the message
- Messages can be screened at the receiving end by specifying a specific tag
  - `MPI_ANY_TAG` matches any tag in a receive
- Tags are sometimes called “message types”
  - MPI calls them “tags” to avoid confusion with datatypes

MPI Basic (Blocking) Send

- `MPI_SEND (start, count, datatype, dest, tag, comm)`
  - The message buffer is described by
    - start, count, datatype
  - The target process is specified by dest
    - rank of the target process in the communicator specified by comm
  - When this function returns
    - data has been delivered to the system
    - buffer can be reused
  - Message may not have been received by target process

MPI Basic (Blocking) Receive

- `MPI_RECV(start, count, datatype, source, tag, comm, status)`
  - Waits until a matching message is received from system
    - Matches on source and tag
    - Buffer must be available
  - source is rank in communicator specified by comm
    - Or `MPI_ANY_SOURCE`
  - Status contains further information
  - Receiving fewer than count is OK, more is not

Retrieving Further Information

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

  ```c
  int recvd_tag, recvd_from, recvd_count;
  MPI_Status status;
  MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
  recvd_tag = status.MPI_TAG;
  recvd_from = status.MPI_SOURCE;
  MPI_Get_count( &status, datatype, &recvd_count );
  ```
Tags and Contexts

- Separation of messages by use of tags
  - Requires libraries to be aware of tags of 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
- Use `MPI_Comm_split` to create new communicators

Broadcast

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

![Diagram of broadcast process]

- 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

Programming MPI with Only Six Functions

- Many parallel programs can be written using:
  - `MPI_INIT()`
  - `MPI_FINALIZE()`
  - `MPI_COMM_SIZE()`
  - `MPI_COMM_RANK()`
  - `MPI_SEND()`
  - `MPI_RECV()`

- Point-to-point (send/recv) isn’t the only way...
  - Add more support for communication

Broadcast (cont’d)

- An example: process 0 sends a string to all the others
  ```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
```

nodes may use NIC (network interface cards)

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

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
    ```
    0
    ```
  - An $n$-dimensional cube is has two copies of an $(n-1)$-dimensional cube, with connections between corresponding nodes
    ```
    00 01
    ```
  - To make labels: prefix with 0 in one copy, 1 in the other copy

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
    ```
    0
    ```
  - An $n$-dimensional cube is has two copies of an $(n-1)$-dimensional cube, with connections between corresponding nodes
    ```
    00 01
    ```
  - To make labels: prefix with 0 in one copy, 1 in the other copy
Hypercubes (cont’d)

- A 3-D cube:

- 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

Hypercubes (cont’d)

- Each bit in the binary label corresponds to a dimension

- Many algorithms use labels to organize communication
  - broadcast: send to neighbor in \( i \)th dimension on step \( i \)

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);
```
Scatter / Gather in MPI

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

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

```c
int i = rank;
int n;
MPI_Reduce(&i, &n, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```
Reduce in MPI (cont’d)

- 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);
  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...

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?
  ```c++
  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

Other Collective Operations

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