COMP 322: Fundamentals of Parallel Programming

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Lecture 37: Introduction to MPI (contd)

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu
Acknowledgments for Today’s Lecture

- “Principles of Parallel Programming”, Calvin Lin & Lawrence Snyder
  — Includes resources available at http://www.pearsonhighered.com/educator/academic/product/0,3110,0321487907,00.html

- “Parallel Architectures”, Calvin Lin
  — Lectures 5 & 6, CS380P, Spring 2009, UT Austin

- mpiJava home page: http://www.hpjava.org/mpiJava.html

- “MPI-based Approaches for Java” presentation by Bryan Carpenter
  — http://www.hpjava.org/courses/ar1

- MPI lectures given at Rice HPC Summer Institute 2009, Tim Warburton, May 2009
Announcements

• Homework 7 due by 5pm on Friday, April 22\textsuperscript{nd}
  — Send email to comp322-staff if you’re running into issues with accessing SUG@R nodes, or anything else

• Take-home final exam will be given on Friday, April 22\textsuperscript{nd}
  — Content will cover second half of semester
    - Come to Friday’s lecture for review of final exam material!
  — Due by 5pm on Friday, April 29\textsuperscript{th}
Organization of a Distributed-Memory Multiprocessor

Figure (a)

- Host node (Pc) connected to a *cluster* of processor nodes (P₀ ... Pₘ)
- Processors P₀ ... Pₘ communicate via an *interconnection network*
  - Supports much lower latencies and higher bandwidth than standard TCP/IP networks

Figure (b)

- Each processor node consists of a processor, memory, and a Network Interface Card (NIC) connected to a router node (R) in the interconnect
Use of MPI on an SMP

- Memory hierarchy for a single Intel Xeon Quad-core E5440 HarperTown processor chip
  - A SUG@R node contains two such chips
Recap of mpiJava Send() and Recv()

- **Send and receive members of Comm:**
  
  ```java
  void Send(Object buf, int offset, int count, Datatype type,
            int dst, int tag) ;
  
  Status Recv(Object buf, int offset, int count, Datatype type,
               int src, int tag) ;
  ```

- The arguments `buf`, `offset`, `count`, `type` describe the data buffer—the storage of the data that is sent or received. They will be discussed on the next slide.

- `dst` is the rank of the destination process relative to this communicator. Similarly in `Recv()`, `src` is the rank of the source process.

- An arbitrarily chosen tag value can be used in `Recv()` to select between several incoming messages: the call will wait until a message sent with a matching tag value arrives.

- The `Recv()` method returns a `Status` value, discussed later.

- Both `Send()` and `Recv()` are **blocking** operations by default—Analogous to a phaser next operation.
Deadlock Scenario #1 (C version)

Consider:

```c
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, MPI_COMM_WORLD);
}
...

If MPI_Send is blocking, there is a deadlock.
```
Deadlock Scenario #2 (C version)

Consider the following piece of code, in which process $i$ sends a message to process $i + 1$ (modulo the number of processes) and receives a message from process $i - 1$ (modulo the number of processes).

```c
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
         MPI_COMM_WORLD);
MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
         MPI_COMM_WORLD);
...

Once again, we have a deadlock if MPI_Send is blocking.
```
Approach #1 to Deadlock Avoidance ---
Reorder Send and Recv calls

We can break the circular wait to avoid deadlocks as follows:

```c
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank%2 == 1) {
    MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
              MPI_COMM_WORLD);
    MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
              MPI_COMM_WORLD);
}
else {
    MPI_Recv(b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
              MPI_COMM_WORLD);
    MPI_Send(a, 10, MPI_INT, (myrank+1)%npes, 1,
              MPI_COMM_WORLD);
}
...
```

...
Approach #2 to Deadlock Avoidance ---
a combined Sendrecv() call

• Since it is fairly common to want to simultaneously send one message while receiving another (as illustrated on the previous slide, MPI provides a more specialized operation for this.

• In mpiJava the corresponding method of Comm has the complicated signature:

  Status Sendrecv(Object sendBuf, int sendOffset, int sendCount,
  Datatype sendType, int dst, int sendTag,
  Object recvBuf, int recvOffset, int recvCount,
  Datatype recvType, int src, int recvTag) ;

  — This can be more efficient that doing separate sends and receives, and it can be used to avoid deadlock conditions in certain situations
    - Analogous to phaser “next” operation, where programmer does not have access to individual signal/wait operations
    — There is also a variant called Sendrecv_replace() which only specifies a single buffer: the original data is sent from this buffer, then overwritten with incoming data.
Using Sendrecv for Deadlock Avoidance in Scenario #2 (C version)

Consider the following piece of code, in which process $i$ sends a message to process $i + 1$ (modulo the number of processes) and receives a message from process $i - 1$ (modulo the number of processes).

```c
int a[10], b[10], npes, myrank;
MPI_Status status;
...
MPI_Comm_size(MPI_COMM_WORLD, &npes);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Sendrecv(a, 10, MPI_INT, (myrank+1)%npes, 1,
             b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
             MPI_COMM_WORLD);
...

A combined Sendrecv() call avoids deadlock in this case.
MPI Nonblocking Point-to-point Communication
Latency in Blocking vs. Nonblocking Communication

Blocking communication

Nonblocking communication
Non-Blocking Send and Receive operations

- In order to overlap communication with computation, MPI provides a pair of functions for performing non-blocking send and receive operations (“I” stands for “Immediate”):

  ```c
  int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
  ```

  ```c
  int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)
  ```

- These operations return before the operations have been completed. Function MPI_Test tests whether or not the non-blocking send or receive operation identified by its request has finished.

  ```c
  int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
  ```

- MPI_Wait waits for the operation to complete.

  ```c
  int MPI_Wait(MPI_Request *request, MPI_Status *status)
  ```
Non-blocking Example

Example pseudo-code on process 0:

```c
if(procid==0){
    MPI_Isend outgoing to 1
    MPI_Irecv incoming from 1
    .. compute ..
    MPI_Wait until Irecv has received incoming
    .. compute ..
    MPI_Wait until Isend does not need outgoing
}
```

Example pseudo-code on process 1:

```c
if(procid==1){
    MPI_Isend outgoing to 0
    MPI_Irecv incoming from 0
    .. compute ..
    MPI_Wait until Irecv has filled incoming
    .. compute ..
    MPI_Wait until Isend does not need outgoing
}
```

Using the “non-blocked” send and receives allows us to overlap the latency and buffering overheads with useful computation.
Avoiding Deadlocks (C version)

Using non-blocking operations removes most deadlocks. Consider:

```c
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
    MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
    MPI_Recv(b, 10, MPI_INT, 0, 2, &status, MPI_COMM_WORLD);
    MPI_Recv(a, 10, MPI_INT, 0, 1, &status, MPI_COMM_WORLD);
}
...

Replacing either the send or the receive operations with non-blocking counterparts fixes this deadlock.
```
MPI Collective Communication
Collective Communications

• A popular feature of MPI is its family of collective communication operations.

• Each of these operations is defined over a communicator.
  — All processes in a communicator must perform the same operation
  — Implicit barrier (next)

• The simplest example is the broadcast operation: all processes invoke the operation, all agreeing on one root process. Data is broadcast from that root.

```c
void Bcast(Object buf, int offset, int count, Datatype type, int root)
  - Broadcast a message from the process with rank root to all processes of the group.
```
MPI_Bcast

A root process sends same message to all

29 represents an array of values

Simple tree broadcast

0
1
2
3
4
5
6
7

proc
More Examples of Collective Operations

- All the following are instance methods of Intracom:
  - `void Barrier()`
    - Blocks the caller until all processes in the group have called it.
  - `void Gather(Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, Object recvbuf, int recvoffset, int recvcount, Datatype recvtype, int root)`
    - Each process sends the contents of its send buffer to the root process.
  - `void Scatter(Object sendbuf, int sendoffset, int sendcount, Datatype sendtype, Object recvbuf, int recvoffset, int recvcount, Datatype recvtype, int root)`
    - Inverse of the operation Gather.
  - `void Reduce(Object sendbuf, int sendoffset, Object recvbuf, int recvoffset, int count, Datatype datatype, Op op, int root)`
    - Combine elements in send buffer of each process using the reduce operation, and return the combined value in the receive buffer of the root process.
MPI_Gather

• On occasion it is necessary to copy an array of data from each process into a single array on a single process.

• Graphically:

  • Note: only process 0 (PO) needs to supply storage for the output
MPI_Reduce

void MPI.COMM_WORLD.Reduce(
    Object[] sendbuf /* in */,
    int sendoffset /* in */,
    Object[] recvbuf /* out */,
    int recvoffset /* in */,
    int count /* in */,
    MPI.Datatype datatype /* in */,
    MPI.Op operator /* in */,
    int root /* in */) 

MPI.COMM_WORLD.Reduce( msg, 0, result, 0, 1, MPI.INT, MPI.SUM, 2);
# Predefined Reduction Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
<th>Datatypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>Maximum</td>
<td>C integers and floating point</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>Minimum</td>
<td>C integers and floating point</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>Sum</td>
<td>C integers and floating point</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>Product</td>
<td>C integers and floating point</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>Logical AND</td>
<td>C integers</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>Bit-wise AND</td>
<td>C integers and byte</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>Logical OR</td>
<td>C integers</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>Bit-wise OR</td>
<td>C integers and byte</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>Logical XOR</td>
<td>C integers</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>Bit-wise XOR</td>
<td>C integers and byte</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>max-min value-location</td>
<td>Data-pairs</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>min-min value-location</td>
<td>Data-pairs</td>
</tr>
</tbody>
</table>
The operation **MPI_MAXLOC** combines pairs of values \((v_i, l_i)\) and returns the pair \((v, l)\) such that \(v\) is the maximum among all \(v_i\)'s and \(l\) is the corresponding \(l_i\) (if there are more than one, it is the smallest among all these \(l_i\)'s).

**MPI_MINLOC** does the same, except for minimum value of \(v_i\).

### Example Use

<table>
<thead>
<tr>
<th>Value</th>
<th>15</th>
<th>17</th>
<th>11</th>
<th>12</th>
<th>17</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

\[
\text{MinLoc(Value, Process)} = (11, 2) \\
\text{MaxLoc(Value, Process)} = (17, 1)
\]
Datatypes for MPI\_MAXLOC and MPI\_MINLOC

**MPI datatypes for data-pairs used with the MPI\_MAXLOC and MPI\_MINLOC reduction operations.**

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>C Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_2INT</td>
<td>pair of ints</td>
</tr>
<tr>
<td>MPI_SHORT_INT</td>
<td>short and int</td>
</tr>
<tr>
<td>MPI_LONG_INT</td>
<td>long and int</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE_INT</td>
<td>long double and int</td>
</tr>
<tr>
<td>MPI_FLOAT_INT</td>
<td>float and int</td>
</tr>
<tr>
<td>MPI_DOUBLE_INT</td>
<td>double and int</td>
</tr>
</tbody>
</table>
More Collective Communication Operations

- If the result of the reduction operation is needed by all processes, MPI provides:
  
  ```c
  int MPI_Allreduce(void *sendbuf, void *recvbuf,
                    int count, MPI_Datatype datatype, MPI_Op op,
                    MPI_Comm comm)
  ```

- MPI also provides the `MPI_Allgater` function in which the data are gathered at all the processes.
  
  ```c
  int MPI_Allgater(void *sendbuf, int sendcount,
                   MPI_Datatype senddatatype, void *recvbuf,
                   int recvcount, MPI_Datatype recvdatatype,
                   MPI_Comm comm)
  ```

- To compute prefix-sums, MPI provides:
  
  ```c
  int MPI_Scan(void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op,
               MPI_Comm comm)
  ```
MPI\_Alltoall

```c
int MPI\_Alltoall(void *sendbuf, int sendcount,
                  MPI\_Datatype senddatatype, void *recvbuf,
                  int recvcount, MPI\_Datatype recvdatatype,
                  MPI\_Comm comm)
```

- Each process submits an array to MPI\_Alltoall.
- The array on each process is split into \textit{nprocs} sub-arrays.
- Sub-array \textit{n} from process \textit{m} is sent to process \textit{n} placed in the \textit{m}'th block in the result array.
Groups and Communicators

• In many parallel algorithms, communication operations need to be restricted to certain subsets of processes.

• MPI provides mechanisms for partitioning the group of processes that belong to a communicator into subgroups each corresponding to a different communicator.

• The simplest such mechanism is:

  int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)

• This operation groups processors by color and sorts resulting groups on the key.
Groups and Communicators

• In many parallel algorithms, communication operations need to be restricted to certain subsets of processes.

• MPI provides mechanisms for partitioning the group of processes that belong to a communicator into subgroups.

• The simplest such mechanism is:
  
  ```c
  int MPI_Comm_split(MPI_Comm comm, int color, int key,
                          MPI_Comm *newcomm)
  ```

• This operation groups processors by color and sorts resulting groups on the key.
Summary of MPI Collective Communications

- A large number of collective operations are available with MPI

- Too many to mention...

- This table summarizes some of the most useful collective operations

<table>
<thead>
<tr>
<th>Collective Function</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Gather</td>
<td>gather together arrays from all processes in comm</td>
</tr>
<tr>
<td>MPI_Reduce</td>
<td>reduce (elementwise) arrays from all processes in communicator</td>
</tr>
<tr>
<td>MPI_Scatter</td>
<td>a “root” process sends consecutive chunks of an array to all processes</td>
</tr>
<tr>
<td>MPI_Alltoall</td>
<td>Block transpose</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>a “root” process sends the same array of data to all processes.</td>
</tr>
</tbody>
</table>