CSE 613: Parallel Programming

Lecture 12
(The Message Passing Interface)

Guest Lecturer: Zafar Ahmad
Slides: Zafar Ahmad, Rezaul Chowdhury
Department of Computer Science
SUNY Stony Brook
Spring 2019
Principles of Message-Passing Programming

- One of the oldest and most widely used approaches for programming parallel computers
- Two key attributes
  - Assumes a partitioned address space
  - Supports only explicit parallelism
- Two immediate implications of partitioned address space
  - Data must be explicitly partitioned and placed to appropriate partitions
  - Each interaction (read-only and read/write) requires cooperation between two processes: process that has the data, and the one that wants to access the data

Source: Blaise Barney, LLNL
Structure of Message-Passing Programs

**Asynchronous**
- All concurrent tasks execute asynchronously
- Most general (can implement any parallel algorithm)
- Can be difficult to reason about
- Can have non-deterministic behavior due to races

**Loosely Synchronous**
- A good compromise between synchronous and asynchronous
- Tasks or subset of tasks synchronize to interact
- Between the interactions tasks execute asynchronously
- Easy to reason about these programs
Structure of Message-Passing Programs

Multiple Program Multiple Data (MPMD)
- Ultimate flexibility in parallel programming
- Unscalable

Single Program Multiple Data (SPMD)
- Most message-passing programs
- Loosely synchronous or completely asynchronous
The Building Blocks: Send & Receive Operations

send( &data, n, dest ):
Send n items pointed to by &data to a processor with id dest

receive( &data, n, src ):
Receive n items from a processor with id src to location pointed to by &data

But wait! What P1 prints when P0 and P1 execute the following code?

```
1    P0
2
3    a = 100;
4    send(&a, 1, 1);
5    a=0;
```
```
1    P0
2
3    receive(&a, 1, 0)
4    printf("%d\n", a);
```

Source: Grama et al., “Introduction to Parallel Computing”, 2nd Edition
Blocking Non-Buffered Send / Receive

Sending operation waits until the matching receive operation is encountered at the receiving process, and data transfer is complete.
Blocking Non-Buffered Send / Receive

May lead to idling:

(a) Sender comes first; idling at sender
Blocking Non-Buffered Send / Receive

May lead to idling:

(a) Sender comes first; idling at sender
(b) Sender and receiver come at about the same time; idling minimized
Blocking Non-Buffered Send / Receive

May lead to idling:

(a) Sender comes first; idling at sender
(b) Sender and receiver come at about the same time; idling minimized
(c) Receiver comes first; idling at receiver

Source: Grama et al., “Introduction to Parallel Computing”, 2nd Edition
**Blocking Non-Buffered Send / Receive**

May lead to deadlocks:

<table>
<thead>
<tr>
<th>P0</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>send(&amp;a, 1, 1);</td>
<td>send(&amp;a, 1, 0);</td>
</tr>
<tr>
<td>receive(&amp;b, 1, 1);</td>
<td>receive(&amp;b, 1, 0);</td>
</tr>
</tbody>
</table>

- The send at P0 waits for the matching receive at P1
- The send at P1 waits for the matching receive at P0

Source: Grama et al., "Introduction to Parallel Computing", 2nd Edition
**Blocking Buffered Send / Receive**

- Sending operation waits until data is copied into a pre-allocated communication buffer at the sending process.
- Data is first copied into a buffer at the receiving process as well, from where data is copied to the target location by the receiver.

Finite buffers lead to delays:

```
1      P0                      P1
2
3      for (i = 0; i < 1000; i++) {
4          produce_data(&a);
5          send(&a, 1, 1);
6      }
7      for (i = 0; i < 1000; i++) {
8          receive(&a, 1, 0);
9          consume_data(&a);
10     }
```

Source: Grama et al., “Introduction to Parallel Computing”, 2nd Edition

What happens if the receiver’s buffer can only hold 10 items?
Blocking Buffered Send / Receive

May still lead to deadlocks:

```
1   P0
2
3   receive(&a, 1, 1);       receive(&a, 1, 0);
4   send(&b, 1, 1);          send(&b, 1, 0);
```

Source: Grama et al., “Introduction to Parallel Computing”, 2nd Edition

— Blocks because the receive calls are always blocking in order to ensure consistency
Non-Blocking Non-Buffered Send / Receive

- Sending operation posts a pending message and returns
- When the corresponding receive is posted data transfer starts
- When data transfer is complete the check-status operation indicates that it is safe to touch the data

Source: Grama et al., “Introduction to Parallel Computing”, 2nd Edition
Non-Blocking Buffered Send / Receive

- Sending operation initiates a DMA (Direct Memory Access) operation and returns immediately
- Data becomes safe as soon as the DMA operation completes
- The receiver initiates a transfer from sender’s buffer to receiver’s target location
- Reduces the time during which the data is unsafe to touch
Possible Protocols for Send & Receive Operations

<table>
<thead>
<tr>
<th>Buffered</th>
<th>Non-Buffered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocking Operations</td>
<td>Non-Blocking Operations</td>
</tr>
<tr>
<td>Sending process returns after data has been copied into communication buffer</td>
<td>Sending process returns after initiating DMA transfer to buffer. This operation may not be completed on return</td>
</tr>
<tr>
<td>Sending process blocks until matching receive operation has been encountered</td>
<td>Programmer must explicitly ensure semantics by polling to verify completion</td>
</tr>
</tbody>
</table>

Send and Receive semantics assured by corresponding operation

Source: Grama et al., “Introduction to Parallel Computing”, 2nd Edition
The Minimal Set of MPI Routines

- The MPI library contains over 125 routines
- But fully functional message-passing programs can be written using only the following 6 MPI routines

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Init</td>
<td>Initializes MPI.</td>
</tr>
<tr>
<td>MPI_Finalize</td>
<td>Terminates MPI.</td>
</tr>
<tr>
<td>MPI_Comm_size</td>
<td>Determines the number of processes.</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>Determines the label of the calling process.</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>Sends a message.</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>Receives a message.</td>
</tr>
</tbody>
</table>

- All 6 functions return `MPI_SUCCESS` upon successful completion, otherwise return an implementation-defined error code
- All MPI routines, data-types and constants are prefixed by `MPI_`
- All of them are defined in `mpi.h` (for C/C++)
Starting and Terminating the MPI Library

1. `#include <mpi.h>`
2. 
3. `main( int argc, char *argv[ ] )`
4. {
5.     `MPI_Init( &argc, &argv );`
6.     ... ... ...  // do some work
7.     `MPI_Finalize( );`
8. }

- Both `MPI_Init` and `MPI_Finalize` must be called by all processes
- Command line should be processed only after `MPI_Init`
- No MPI function may be called after `MPI_Finalize`
Communicators

- A communicator defines the scope of a communication operation
- Each process included in the communicator has a rank associated with the communicator
- By default, all processes are included in a communicator called `MPI_COMM_WORLD`, and each process is given a unique rank between 0 and \( p - 1 \), where \( p \) is the number of processes
- Additional communicator can be created for groups of processes
- To get the size of a communicator:
  ```
  int MPI_Comm_size( MPI_Comm comm, int *size )
  ```
- To get the rank of a process associated with a communicator:
  ```
  int MPI_Comm_rank( MPI_Comm comm, int *rank )
  ```
Communicators

1. 

2. 

3. `main( int argc, char *argv[ ] )`

4. {

5.    int p, myrank;

6.    `MPI_Init( &argc, &argv );`

7.    `MPI_Comm_size( MPI_COMM_WORLD, &p );`

8.    `MPI_Comm_rank( MPI_COMM_WORLD, &myrank );`

9.    `printf( “This is process %d out of %d!\n”, p, myrank );`

10.   `MPI_Finalize( );`

11. }

MPI Standard Blocking Send Format

```
int MPI_Send( void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm )
```

data parameters

- address of send buffer
- number of items to send
- datatype of each item

envelope parameters

- rank of destination process
- message tag
- communicator
MPI Standard Blocking Receive Format

Data parameters:
- Address of receive buffer
- Number of items to receive
- Datatype of each item

Envelope parameters:
- Rank of source process
- Message tag
- Communicator
- Status after operation

Function:
```c
int MPI_Recv( void *buf, int count, MPI_Datatype datatype,
              int src, int tag, MPI_Comm comm, MPI_Status *status )
```
## MPI Datatypes

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>C Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI UNSIGNED SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>
Blocking Send/Receive between Two Processes

1. `#include <mpi.h>
2. 
3. `main(int argc, char *argv[])`
4. {
5.   `int myrank, v = 121;`
6.   `MPI_Status status;`
7.   `MPI_Init(&argc, &argv);`
8.   `MPI_Comm_rank(MPI_COMM_WORLD, &myrank);`
9.   `if (myrank == 0) {
10.      `MPI_Send(&v, 1, MPI_INT, 1, MPI_ANY_TAG, MPI_COMM_WORLD);`
11.      `printf("Process %d sent %d\n", p, myrank, v);
12.   } else if (myrank == 1) {
13.      `MPI_Recv(&v, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
14.      `printf("Process %d received %d\n", p, myrank, v);
15.   }
16.   `MPI_Finalize();
17. }`
**MPI Status**

MPI_Status holds few primary pieces of information: MPI_SOURCE, MPI_TAG, count, cancelled and MPI_ERROR. If we look at the **MPI_Recv**, it has source, tag and count as parameters.

```c
int MPI_Recv( void *buf, int count, MPI_Datatype datatype,
              int src, int tag, MPI_Comm comm, MPI_Status *status )
```

- **number of items to receive**
- **rank of source process**
- **message tag**

MPI_STATUS_IGNORE can be used as status parameter. But, why do we need these information inside MPI_Status?
Dynamic Receiving and MPI Status

It turns out that MPI_Recv can have MPI_ANY_SOURCE and MPI_ANY_TAG for receiving data for any tag and from any source. Later, source and tag can be extracted from status parameter by directly accessing status.MPI_SOURCE and status.MPI_TAG.

What if the receiver doesn’t know the size of the received data?
It turns out that `MPI_Recv` can have `MPI_ANY_SOURCE` and `MPI_ANY_TAG` for receiving data for any tag and from any source. Later, source and tag can be extracted from status parameter by directly accessing `status.MPI_SOURCE` and `status.MPI_TAG`.

What if the receiver doesn’t know the size of the received data? In that case, it one can use a buffer of maximum possible size and extract the received data size from `status` using `MPI_Get_count`.

```c
int MPI_Get_count( MPI_Status *status, MPI_Datatype datatype, int *count )
```

But isn’t it a waste to allocate that unused receive buffer?
Dynamic Receiving and MPI_Probe

Instead of using a large buffer to handle all the cases, we can use MPI_Probe to query the message size before receiving it.

```c
int MPI_Probe( int source, int tag, MPI_Comm comm, MPI_Status *status )
```

MPI_Probe does everything MPI_Recv does but receiving the actual message. Element count can be extracted from status using MPI_Get_count.
Dynamic Receive and MPI_Probe

1. #include <mpi.h>
2. main( int argc, char *argv[ ] )
3. {
4.   int myrank, v = 121, count;
5.   MPI_Status status;
6.   MPI_Init( &argc, &argv );
7.   MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
8.   if ( myrank == 0 ) {
9.     MPI_Send( &v, 1, MPI_INT, 1, MPI_ANY_TAG, MPI_COMM_WORLD, &status );
10.   } else if ( myrank == 1 ) {
11.     MPI_Probe( 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status );
12.     MPI_Get_count( &status, MPI_INT, &count );
13.     int recv_buf = (int *) malloc(sizeof(int) * count);
14.     MPI_Recv( &recv_buff, count, MPI_INT, 0, MPI_ANY_TAG,
15.                  MPI_COMM_WORLD, MPI_STATUS_IGNORE );
16.   }
17.   MPI_Finalize( );
Non-Blocking Send / Receive

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

int MPI_Irecv( void *buf, int count, MPI_Datatype datatype, int src, int tag, MPI_Comm comm, MPI_Request *req )

The MPI_Request object is used as an argument to the following two functions to identify the operation whose status we want to query or to wait for its completion.

int MPI_Test( MPI_Request *req, int *flag, MPI_Status *status )
  — Returns *flag = 1, if the operation associated with *req has completed, otherwise returns *flag = 0

int MPI_Wait( MPI_Request *req, MPI_Status *status )
  — Waits until the operation associated with *req completes
# Non-Blocking Send and Blocking Receive

1. `#include < mpi.h >`

2. 

3. `main( int argc, char *argv[ ] )`

4. `{`

5. `int myrank, v = 121;`

6. `MPI_Status status;`

7. `MPI_Request req;`

8. `MPI_Init( &argc, &argv );`

9. `MPI_Comm_rank( MPI_COMM_WORLD, &myrank );`

10. `if ( myrank == 0 ) {`

11. `   MPI_Isend( &v, 1, MPI_INT, 1, MPI_ANY_TAG, MPI_COMM_WORLD, &req );`

12. `   compute( );            /* but do not modify v */`

13. `   MPI_Wait( &req, &status );`

14. `} else if ( myrank == 1 ) MPI_Recv( &v, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status );`

15. `MPI_Finalize( );`

16. `}`
# include < mpi. h >

main( int argc, char *argv[ ] )

{ 
 int myrank, v = 121;

 MPI_Status status;

 MPI_Request req;

 MPI_Init( &argc, &argv );

 MPI_Comm_rank( MPI_COMM_WORLD, &myrank );

 if ( myrank == 0 ) {
  MPI_Isend( &v, 1, MPI_INT, 1, MPI_ANY_TAG, MPI_COMM_WORLD, &req );
  compute( ); /* but do not modify v */
  MPI_Wait( &req, &status );
  }
 else if ( myrank == 1 ) {
  MPI_Irecv( &v, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &req );
  compute( ); /* but do not read or modify v */
  MPI_Wait( &req, &status );
  }
}

MPI_Finalize( );
MPI Collective Communication & Computation Operations

Synchronization
  - Barrier

Data Movement
  - Broadcast
  - Scatter
  - Gather
  - All-to-all

Global Computation
  - Reduce
  - Scan

These routines must be called by all processes in the communication group.
**Barrier Synchronization**

```c
int MPI_Barrier( MPI_Comm comm )
```

Returns only after all processes in the communication group have called this function.
Broadcast

int MPI_Bcast( void *buf,
              int count,
              MPI_Datatype datatype,
              int src,
              MPI_Comm comm )

Sends the data stored in the buffer *buf* of process *src* to all the other processes in the group.

Broadcast
int MPI_Scatter( void *sendbuf,
int sendcount,
MPI_Datatype sendtype,
void *recvbuf,
int recvcount,
MPI_Datatype recvtype,
int src,
MPI_Comm comm )

The src process sends a different part of sendbuf to each process, including itself. Process i receives sendcount contiguous elements starting from $i \times \text{sendcount}$. The received data are stored in recvbuf.

The opposite of scatter.

Every process, including \textit{dest} sends data stored in \textit{sendbuf} to \textit{dest}.

Data from process \textit{i} occupy \textit{sendcount} contiguous locations of \textit{recvbuf} starting from \textit{i} × \textit{sendcount}.

\begin{verbatim}
int MPI_Gather( void *sendbuf, int sendcount, MPI_Datatype sendtype, 
                void *recvbuf, int recvcount, MPI_Datatype recvtype, 
                int dest, MPI_Comm comm )
\end{verbatim}

Reduce

Combines the elements stored in `sendbuf` of each process using the operation `op`, and stores the combined values in `recvbuf` of the process with rank `dest`.

```c
int MPI_Reduce( void *sendbuf,
                void *recvbuf,
                int count,
                MPI_Datatype datatype,
                MPI_Op op,
                int dest,
                MPI_Comm comm )
```
int MPI_Reduce( void *sendbuf,  
void *recvbuf,  
int count,  
MPI_Datatype datatype,  
MPI_Op op,  
int dest,  
MPI_Comm comm )

Combines the elements stored in `sendbuf` of each process using the operation `op`, and stores the combined values in `recvbuf` of the process with rank `dest`. 

### Reduce

**MPI Reduce**

```c
MPI_Reduce( vals, sums, 4, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD )
```

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$a_0$</th>
<th>$b_0$</th>
<th>$c_0$</th>
<th>$d_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_1$</td>
<td>$d_1$</td>
</tr>
<tr>
<td>$P_2$</td>
<td>$a_2$</td>
<td>$b_2$</td>
<td>$c_2$</td>
<td>$d_2$</td>
</tr>
<tr>
<td>$P_3$</td>
<td>$a_3$</td>
<td>$b_3$</td>
<td>$c_3$</td>
<td>$d_3$</td>
</tr>
</tbody>
</table>

| $a_0 + a_1 + a_2 + a_3$ | $b_0 + b_1 + b_2 + b_3$ | $c_0 + c_1 + c_2 + c_3$ | $d_0 + d_1 + d_2 + d_3$ |
### 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>
intMPI_Scan(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

Performs a prefix reduction of the data stored in sendbuf at each process and returns the results in recvbuf of the process with rank dest.

\[
\begin{array}{cccc}
  P_0 & a_0 & b_0 & c_0 & d_0 \\
  P_1 & a_1 & b_1 & c_1 & d_1 \\
  P_2 & a_2 & b_2 & c_2 & d_2 \\
  P_3 & a_3 & b_3 & c_3 & d_3 \\
\end{array}
\quad\rightarrow\quad
\begin{array}{cccc}
  a_0 & b_0 & c_0 & d_0 \\
  a_0+a_1 & b_0+b_1 & c_0+c_1 & d_0+d_1 \\
  a_0+a_1+a_2 & b_0+b_1+b_2 & c_0+c_1+c_2 & d_0+d_1+d_2 \\
  a_0+a_1+a_2+a_3 & b_0+b_1+b_2+b_3 & c_0+c_1+c_2+c_3 & d_0+d_1+d_2+d_3 \\
\end{array}
\]

MPI_Scan(vals, sums, 4, MPI_INT, MPI_SUM, MPI_COMM_WORLD)