CME213/ME339
Lecture 18

Eric Darve
Erich Elsen
Austin Gibbons

Department of Mechanical Engineering
Institute for Computational and Mathematical Engineering
Stanford University

Spring 2012

MPI library

The MPI library is used to allow nodes to communicate and exchange information.

There are a number of subtle points involved in this process.

Blocking and non-blocking

Although the details are not always available, in general there are two ways to communicate data.

Say that some data is residing in some memory location. You want to send that data.

1. You try to send that data directly. Pro: minimal use of resources, fast. Con: you cannot continue execution of the program until the transfer is complete.
2. You copy that data to a system buffer and then do the communication. Pro: you can continue executing the code as soon as the memory copy is complete. Con: it requires some memory resources.

Buffered communications

The ability to use systems buffers for communications can be essential.

It allows the sending and receiving processes to send and receive at different times, minimizing idle times.

Without such buffers, communication would automatically require that both senders and receivers execute the communication instruction at exactly the same time, leading possibly to a lot of idle time.

This leads to two separate concepts: blocking and non-blocking communications.

- Blocking: a blocking send will not return until the send buffer can safely be used. This means that either the data was copied to a system buffer or the communication completed.
- Non-blocking: returns more or less immediately. The user can check later on whether the communication is complete or not.

This discussion involves only the view point of the local process.

Synchronous and asynchronous communication involve a global view point.

Synchronous and asynchronous communications

- Asynchronous: no coordination between the sending and receiving process.
- Synchronous: the operation completes when the sending buffer can be reused and the received process has started its receive operation.

Synchronous communications provide natural synchronization points if they are needed.

Point-to-point communication

The simplest form of communication: a process sends some data to another process through the network.

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

`count`: number of elements
`MPI_Datatype`: see next slide
`dest`: destination process number
`tag`: a tag that can be used to distinguish different messages from the same sender
`MPI_Comm`: see later

Data types

`MPI_CHAR`	signed char
`MPI_SHORT`	signed short int
`MPI_INT`	signed int
`MPI_LONG`	signed long int
`MPI_UNSIGNED_CHAR`	unsigned char
`MPI_UNSIGNED_SHORT`	unsigned short int
`MPI_UNSIGNED`	unsigned int
`MPI_UNSIGNED_LONG`	unsigned long int
`MPI_FLOAT`	float
`MPI_DOUBLE`	double
`MPI_LONG_DOUBLE`	long double
`MPI_BYTE`	8 binary digits
`MPI_PACKED`	data packed or unpacked with `MPI_Pack()`/`MPI_Unpack`

Receive instruction

Similar command on the receiving process:

```
1  int MPI_Recv(void* buf, int count, MPI_Datatype datatype,
2               int source, int tag, MPI_Comm comm,
3               MPI_Status *status)
```

`MPI_Status`: contains information about the status of the message after having been received.

Blocking

These two commands are blocking asynchronous operations.

This typically offers lower performance than non-blocking but is easier to use.

Once the call is complete and returns, the code can continue executing as if the communication had taken place.

Example

```
1  #include "mpi.h"
2  main( argc, argv )
3  int argc;
4  char **argv;
5  {
6      char message[20];
7      int myrank;
8      MPI_Status status;
9      MPI_Init( &argc, &argv);
10     MPI_Comm_rank( MPI_COMM_WORLD, &myrank);
```

Example

```
1     if (myrank == 0)     /* code for process zero */
2     {
3         strcpy(message,"Hello, there");
4         MPI_Send(message, strlen(message), MPI_CHAR, 1, 99,
5                  MPI_COMM_WORLD);
6     }
7     else                 /* code for process one */
8     {
9         MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD,
10                 &status);
11        printf("received :%s:\n", message);
12    }
13    MPI_Finalize();
14 }
```

99 is the tag (some arbitrary number here).

ANY

There are two special keywords:

1. `MPI_ANY_SOURCE`: will receive from any source
2. `MPI_ANY_TAG`: any tag will do.

No guaranteed delivery order

```
1  MPI_Comm_rank( MPI_COMM_WORLD, &myrank);
2  if (myrank == 0) {
3    MPI_Send(sendbuf1, count, MPI_INT, 2, tag, MPI_COMM_WORLD);
4    MPI_Send(sendbuf2, count, MPI_INT, 1, tag, MPI_COMM_WORLD);
5  } else if (myrank == 1) {
6    MPI_Recv(recvbuf1, count, MPI_INT, 0, tag, MPI_COMM_WORLD,
7             &status);
8    MPI_Send(recvbuf1, count, MPI_INT, 2, tag, MPI_COMM_WORLD);
9  } else if (myrank == 2) {
10   MPI_Recv(recvbuf1, count, MPI_INT, MPI_ANY_SOURCE, tag,
11            MPI_COMM_WORLD, &status);
12   MPI_Recv(recvbuf2, count, MPI_INT, MPI_ANY_SOURCE, tag,
13            MPI_COMM_WORLD, &status);
14 }
```

Deadlocks

We have seen this concept before. With MPI, deadlocks are pretty easy to generate.

```
1  MPI_Comm_rank( MPI_COMM_WORLD, &myrank);
2  if (myrank == 0) {
3    MPI_Recv(recvbuf, count, MPI_INT, 1, tag, MPI_COMM_WORLD,
4             &status);
5    MPI_Send(sendbuf, count, MPI_INT, 1, tag, MPI_COMM_WORLD);
6  } else if (myrank == 1) {
7    MPI_Recv(recvbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD,
8             &status);
9    MPI_Send(sendbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD);
10 }
```

This code is not going to make much progress.

Better

```
1  MPI_Comm_rank( MPI_COMM_WORLD, &myrank);
2  if (myrank == 0) {
3    MPI_Send(sendbuf, count, MPI_INT, 1, tag, MPI_COMM_WORLD);
4    MPI_Recv(recvbuf, count, MPI_INT, 1, tag, MPI_COMM_WORLD,
5             &status);
6  } else if (myrank == 1) {
7    MPI_Send(sendbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD);
8    MPI_Recv(recvbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD,
9             &status);
10 }
```

Will work if MPI decides to use buffers for these communications.

Secure implementation

A secure implementation always works regardless of the buffering strategy.

```
1  MPI_Comm_rank( MPI_COMM_WORLD, &myrank);
2  if (myrank == 0) {
3    MPI_Send(sendbuf, count, MPI_INT, 1, tag, MPI_COMM_WORLD);
4    MPI_Recv(recvbuf, count, MPI_INT, 1, tag, MPI_COMM_WORLD,
5             &status);
6  } else if (myrank == 1) {
7    MPI_Recv(recvbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD,
8             &status);
9    MPI_Send(sendbuf, count, MPI_INT, 0, tag, MPI_COMM_WORLD);
10 }
```

Send-receive

Since this situation where processes send and receive is common, a special command is provided:

```
1  int MPI_Sendrecv(
2            void *sendbuf, int sendcount, MPI_Datatype sendtype,
3            int dest, int sendtag, void *recvbuf, int recvcount,
4            MPI_Datatype recvtype, int source, int recvtag,
5            MPI_Comm comm, MPI_Status *status)
```

This is a blocking operation.

Non-blocking operations

Similar non-blocking operations are provided:

`MPI_send` $\rightarrow$ `MPI_Isend`

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

`MPI_Request*`: use to get information later on about the status of that operation.

What does `I` stand for?

Non-blocking operations

Similar non-blocking operations are provided:

`MPI_send` $\rightarrow$ `MPI_Isend`

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

`MPI_Request*`: use to get information later on about the status of that operation.

What does `I` stand for? Immediate.

Testing and waiting

There is a similar

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

Test using:

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

`flag` is 1 if completed, 0 otherwise.

Wait until operation completes:

```
1 int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

Wait all

`MPI_Waitall`: waits for all given communications to complete.

Syntax:

```
1 int MPI_Waitall(int count, MPI_Request *array_of_requests,
2                  MPI_Status *array_of_statuses)
```

Input Parameters:
`count`: lists length (integer)
`array_of_requests`: array of requests (array of handles).

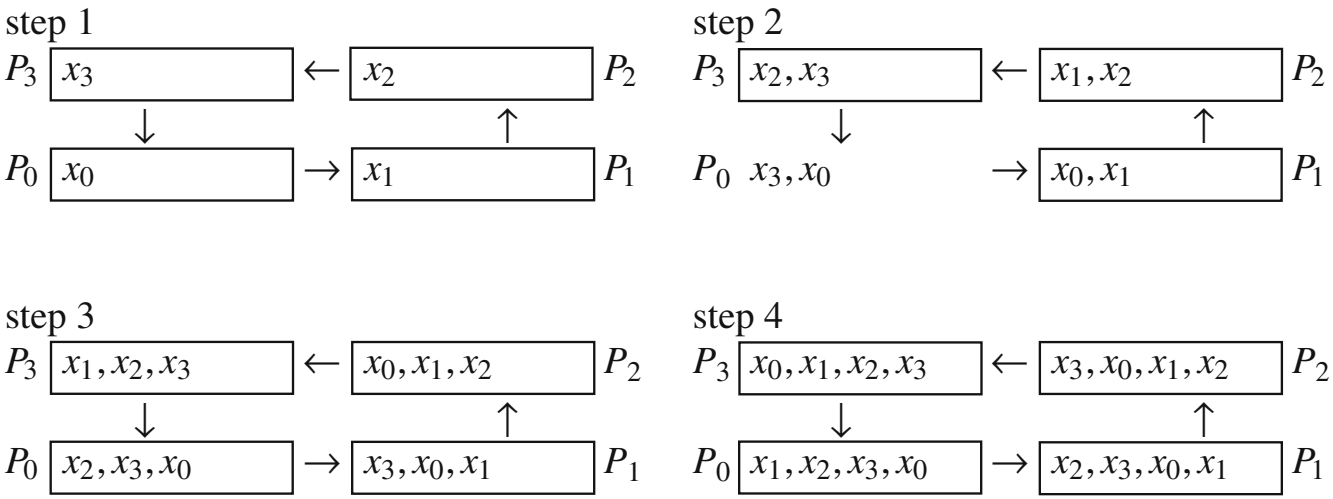
Output Parameters:
`array_of_statuses`: array of status objects (array of status).

Example

Non-blocking gather ring.

Assume we have a ring of processes. Each process has some data x_i. We want all processes to have all the data.

At each step, a process receives from its left and sends to its right the data it received at the previous step.



Code

```
1  void Gather_ring(float * x, int blocksize, float * y) {
2    int i, p, my_rank, next, prev;
3    int send_offset, recv_offset;
4    MPI_Status status;
5    MPI_Request send_request, recv_request;
6
7    MPI_Comm_size(MPI_COMM_WORLD, &p);
8    // Number of processes
9    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
10   // My process ID or rank
11   ...
```

Code

```
1   ...
2   for (i=0; i<blocksize; i++)
3     y[i+my_rank*blocksize] = x[i];
4   /* Copying data that was produced or is owned
5      by process my_rank */
6
7   next = (my_rank+1) % p;    // next process in ring
8   prev = (my_rank-1+p) % p; // previous process
9   send_offset = my_rank * blocksize;
10  recv_offset = ((my_rank-1+p) % p) * blocksize;
11  ...
```

Code

```
1    for (i=0; i<p-1; i++) {
2      // Sending and receiving the different x blocks
3      MPI_Isend(y+send_offset, blocksize, MPI_FLOAT, next,
4              0, MPI_COMM_WORLD, &send_request);
5      MPI_Irecv(y+recv_offset, blocksize, MPI_FLOAT, prev,
6              0, MPI_COMM_WORLD, &recv_request);
7      send_offset = ((my_rank-i-1+p) % p) * blocksize;
8      recv_offset = ((my_rank-i-2+p) % p) * blocksize;
9      MPI_Wait(&send_request, &status);
10     MPI_Wait(&recv_request, &status);
11   }
12 }
```

Communication modes

This is a more advanced topic. We cover the only the key concepts.

Standard: this the mode we have used up to now. This is in most cases sufficient. It provided good performance and relies on the MPI library for several optimizations. For example, MPI will decide whether or not buffers should be used.

Synchronous mode

In synchronous mode, a send operation will be completed not before the corresponding receive operation has been started and the receiving process has started to receive the data sent.

This leads to a form of synchronization between the sending and the receiving processes: the completion of a send operation in synchronous mode indicates that the receiver has started to store the message in its local receive buffer.

Note: completion does not imply that the receiving node has finished receiving the data.

Blocking: returns when completed (sending buffer can be reused + receiving process started executing the matching receive).

Non-blocking: an `MPI_Wait` is required to check completion.

See documentation for more info. Commands include for example `MPI_Ssend()`.

Two other modes

Buffered mode: the user can allocated space for the MPI system buffer. This guarantees that a buffer is used. This is therefore a **guaranteed local operation.** It never depends on the receiving node. `MPI_Bsend()`

Ready mode: it can only be used if the user can guarantee that a matching receive has already been posted. The user is responsible for writing a correct program. Ready mode aims to minimize system overhead and synchronization overhead incurred by the sending task. `MPI_Rsend()`