Lecture 14: Message Passing

Message passing is a parallel programming style used typically on distributed memory machines. Processors communicate using messages. Message passing is the most popular parallel programming paradigm. PVM and MPI are message passing libraries.

1 Generic Message Passing

Message passing libraries support basic primivites to send and receive data through complementary operations. Message passing libraries support a machine model in which each pair of processors have a virtual communications channel between them.

1.1 Typical Primitives

```
void send (processor, void *msg, size)
int recv (processor, void *msg_buffer, max_buffer_size)
```

Blocking send and receive operations are the most common. Blocking send does not return control until it is safe to modify the message buffer. Blocking receive does not return control until the message is received.

1.2 Pseudo-implementation

```
int recv (processor, void *msg_buffer, max_buffer_size)
while channel from processor empty
      do nothing
if max_buffer_size < msg_size
   then error
read message from channel into msg_buffer
return msg_size
```

The while loop which waits until the channel is not empty is a poll.

```
send (processor, void *msg, size)
while channel to processor busy
      do nothing
while recv not called by processor
```

```
        do nothing
    write size, msg into channel
    return
```

The while loop which waits until the channel is not busy is a poll. The while loop that waits the receiving processor calls recv is a non-local operation, we will see how to implement this later.

This is a *synchronous* blocking send. It does not complete until a recv is initiated by the receiving processor. This is the most common type of send used in message passing libraries.

2 Message Passing Programming

Message passing programming is appealing becuase it is a very simple model. Message passing programs are usually SPMD (Single Program, Multiple Data) style.

Here is a simple example which exchanges data between processors 0 and 1.

```
if self = 0
   then send (1, &data, 100)
        recv (1, &data, 100)
if self = 1
   then recv (1, &data, 100)
        send (1, &data, 100)
```

If the order of the send and the recv in the second **if** are switched, this program will deadlock.

Synchronous blocking communication is the most common protocol for sending large amounts of data. Programmer must explicity program both ends of the communication and avoid deadlock.

A three-phase protocol is used to implement the non-local component of synchronous blocking sends.

There are number inefficiencies in synchronous blocking send receives

- Three messages to send one peice of data
- Three message latencies
- Cannot overlap activities on multiple channels

But synchronous blocking send requiresly only one request-to-send buffer per processor, $\Theta(P)$ buffer space.

3 Example: circular shift

Idea:

Processor	0	1	2	3	4	5
	Send1	Recv0	Send3	Recv2	Send5	Recv4
	Send5	Recv2	Send1	Recv4	Send3	Recv0

There is no overlap in communication, cicular shift requires two phases. Circular shift with an odd number of processors is more complicated.

4 Buffered Blocking Sends

Buffered blocking sends move message data to a system buffer before sending, and returns to user code immediately.

send (processor, void *msg, size)
while buffer full
  do nothing
copy msg to buffer
call synchronous blocking send on buffer, while polling run user code

Buffered blocking sends make circular shift easier to implement.

Processor	0	1	2	3	4	5
	Send1	Send2	Send3	Send3	Send5	Send0
	Recv5	Recv0	Recv1	Recv2	Recv3	Recv4

Consider code which sends date to it's nieghbors:

Processor	0	1	2	3	4	5
	Send1	Send2	Send3	Send3	Send5	Send0
	Send5	Send0	Send1	Send2	Send3	Send3
	Recv1	Recv2	Recv3	Recv4	Recv5	Recv0
	Recv5	Recv0	Recv1	Recv2	Recv3	Recv4

This code requires buffer space for two messages, or it will deadlock. Copying messages into the buffer can be costly, but the advantage is the code is simpler for than the synchronous, unbuffered case.

There can be problems other than deadlock, such as out-of-order message delivery. Most message passing implementations guarantee FIFO message delivery.

5 Nonselective receive

int recv (void *msg_buffer, max_buffer_size)

Nonselective receive performs a blocking poll for a message on all virtual channels. When message is detected, it performs a receive on the appropriate channel.

6 Collective Communication

Message passing libraries also support global, collective communication operations such as broadcast, reduce, scan, and barriers (block until all processors execute barrier).

7 Non-blocking Operations

Given these operations, you still cannot implement a sparce histagram, since the receiving processor does not know if a message will be sent to it, and therefore cannot know to do a receive operation.

The solution to this are non-blocking operations. Non-blocking send and receive operations return immediately, and can be tested for completion later. This leads to a "split phase" style, where initiating and completing communications are seperate operations.

8 Why?

Why is message passing successful?

- Message passing progams are very portable, the message passing model makes few hardare assumptions.
- Message passing provides a simple model. Most users want grid communications for scientific programming.
- pIt is easy to get a simple message passing program working.
- Simple solution to deadlocks: increase buffer sizes.
- Exploits hierarchical memory.