JEFF SQUYRES

'Progress' is the opposite of 'Congress'

My mother called me the other day, tremendously concerned. "Are you going to get sued?" she demanded. "Are you going to have to pay a fine, or go to jail?"

"Err... Mom, I have no idea what you're talking about."

"That 'work' you do. I heard on the radio today that the recording industry is suing hundreds of people for illegally downloading music on the Internet."

After a long conversation detailing the difference between MP3 files and MPI, my mom is still convinced that I'm going to jail.

The Story So Far

Last month, we talked about some of the fundamental architectural decisions that an MPI implementation has to make during its design phases. This month, we'll take a closer look at some point-to-point message passing issues and implementation details. The performance on various platforms and networks can vary wildly, not just because of the quality of the MPI implementation (although this is a frequently overlooked issue) but because of how *progress* is made in the underlying MPI engine, which is frequently defined in terms of how the underlying network(s) allow messages to be sent and received.

The inner workings of how an MPI implementation performs the seemingly simple task of moving bytes from one process to another may surprise you. More importantly, having at least a basic understanding of what your MPI is doing may enable you to write your application to match its underlying semantics, and therefore squeeze just a little bit more performance out of your environment.

Message Envelopes

Keep in mind that the body of the message is not the only content that needs to be sent across the network. A prologue — frequently called an *envelope* — must also be sent so that the receiver knows what the message is. For example, the envelope usually contains identifiers for the message's communicator and tag. It usually contains a small amount of other information as well, but this varies from implementation to implementation (and potentially even between network types in the same MPI implementation). Regardless, the overall envelope size is kept fairly small; enlarging it will only increase the latency of small messages — consider that sending a 1-byte user message must *also* send a corresponding envelope.

Some MPI implementations send different types (and sizes) of envelopes depending on the message; *Sidebar One* shows a list of common elements that may be included in envelopes. An implementation's goal is typically to send the shortest envelope possible to reduce the latency of "simple" messages, such as short, non-synchronous messages.

MPI Progress

"Progress" is typically defined as the advancement of pending communications — completing sends and/or receives that were initiated previously. This includes both blocking and non-blocking communications. For simplicity, we'll focus on single-threaded MPI implementations here; the issues facing multi-threaded implementations are best described as "analogous" — they're similar in spirit, but the deep, dark details are different.

What exactly is an MPI implementation *doing* while it is "making progress"? How do the bytes get from process A to process B?

The answer is inevitably different in each MPI implementation, but let's look at a few common examples to see what is happening under the covers in a typical TCP sockets-based implementation (next month, we'll delve into non-sockets based implementations). These examples are likely not *exactly* how any one MPI implementation works, but are influenced by several real-world MPI progress engines.

Sockets

Most sockets-based implementations open at most one socket for point-to-point MPI messages between pairs of processes; all traffic is multiplexed across the socket.

If no other communications are pending, blocking sends are [deceptively] easy — the `writev(2)` system call can send both the envelope and payload in a single function call (and potentially in a single TCP packet). The operating system may or may not block... but who cares? MPI is allowed to block, so that point is irrelevant. Also recall that when `write()` and `writev()` return, we have no guarantees from the operating system that the message has actually been received (or even sent!). All we know is that the buffer is available for re-use, and therefore MPI can return from the blocking send.

Similarly, blocking receives can be implemented with calls to `read(2)`. Using eager or rendezvous protocols, the algorithm is simple because the MPI is allowed to block in `writev(2)` and `read(2)`. The following simplistic pseudocode outlines both cases:

```
1 if (size <= eager_size) {
2  fill_envelope(env,iov);
3  writev(fd,iov,2);
4 } else {
5  fill_envelope(env,iov);
6  write(fd,env,sizeof(env));
7  wait_for_ack(fd);
8  writev(fd,iov,2);
9 }
```

A real MPI implementation will likely have a more complex state machine than this, such as allowing other communications to progress (e.g., other non-blocking communication, unexpected receives, etc.) during significant blocking periods (e.g., the `wait_for_ack()` function could block for an arbitrarily long time). But you get the idea.

Non-Blocking Communication

Non-blocking communication is where a lot of the bookkeeping complexity of an MPI implementation originates. The MPI "immediate" send functions are supposed to return immediately (e.g., MPI_ISEND). Most MPI implementations try to send the message immediately, but may not be able to do so. That is, the socket will be set to non-blocking mode and the implementation will invoke `write()` (or `writev()`). The operating system will write as many bytes as it can before it would block and then return.

It is up to the MPI implementation to track the fact that a given message has "claimed" the socket (i.e., no other traffic should be written down the socket until this message has been completely written), how many bytes have been written, how many are left to write, etc. Complexity arises when multiple outgoing messages are destined for the same peer - since they all have to be multiplexed across the same socket, the MPI must queue them up and progress them in order (potentially in a non-blocking manner - sending as many bytes of a envelope/message as possible each time through the progression engine).

These issues are multiplied when you consider that there are multiple sockets and messages that must be polled for progress simultaneously in order to provide some degree of fairness of message delivery to all peers. The MPI implementation typically invokes `poll(2)` or `select(2)` to see which sockets can be written. If any are available, it tries to progress the pending send queue for each ready file descriptor as far as possible until it would block.

Taking this into account, re-consider the above blocking example. What if other non-blocking sends are still waiting to be sent? Unless the MPI can handle out-of-order sending, the blocking send must progress them all until it can send its own message. So the simplistic pseudocode from above is likely only relevant once the blocking send is allowed to be sent (i.e., can "claim" the socket *and* all prior messages have been sent).

Receiver Logic

The receiver has non-trivial logic to implement as well. TCP allows for partial receives — a `read()` call may receive *some* bytes, but not all. Analogous to the non-blocking sending logic, the MPI implementation must maintain bookkeeping for which receiving envelopes / messages have "claimed" the socket, how many bytes have been received so far, how many are left to receive, etc. The same `poll()` or `select()` used for check-

Common MPI envelope data

Envelope type: If the MPI implementation uses different types (and sizes) of envelopes for different situations, an identifier must be included to indicate what kind of envelope is being used.

Communicator ID: Almost always required (may not be required if the communicator can be otherwise identified, such as having a different TCP socket for each communicator, but this may be tremendously inefficient in terms of resources).

Tag: Almost always required (ditto with communicator ID).

Originator: Some networks can naturally provide from whom an envelope was received (e.g., TCP), but others cannot (e.g., gm/Myrinet). If the network cannot provide the information, the originator of the envelope must be included in the envelope.

Message length: Even if the network or operating system can provide this, it is frequently more efficient if the MPI implementation knows how many bytes to receive in the payload.

Flags: Typically a bit field, indicates information such as: whether the envelope contains an ACK, whether the message requires an ACK, etc.

Sequence number: Some MPI implementation may send messages out of order (for efficiency, or if messages may take multiple paths to the destination); a sequence number may be required to enforce MPI's message ordering guarantees.

Peer request pointer: For messages that require multiple sends (rendezvous protocols, synchronous messages, etc.), including a pointer to the source request can avoid searching in memory for the matching message. For example, if a sender includes a pointer to its MPI request entry in the envelope, and that address is echoed back in the ACK from the receiver, the sender knows exactly which request has been acknowledged.

ing write progress can also be used to check for pending incoming data.

Once an envelope has been fully received, it must either be matched with a previously posted receive or placed in a special queue for "unexpected" messages. For eager sends, the body of the message must be received as well. If the message is expected, the body can be received directly into the target buffer. But if the message is unexpected, a temporary buffer must be allocated to receive the message. This temporary buffer is then associated with the envelope in the unexpected queue.

If the message body was *not* sent eagerly and was expected, the receiver must mark the matching receive as "in progress" and send an ACK back to the sender. Consider the following valid MPI code:

```
1 if (i_am_sender) {
2   MPI_Isend(long_msg, ...,
    peer, tag, comm, &req);
3   MPI_Send(short_msg, ...,
     peer, tag, comm);
4   MPI_Wait(&req, &status);
5 } else {
6   MPI_Recv(msg1, ...,
    peer, tag, comm);
7   MPI_Recv(msg2, ...,
    peer, tag, comm);
8 }
```

Now consider how the traffic will actually flow across the socket:

- Sender sends envelope for the long message
- Sender sends envelope + message for the short message
- Receiver receives envelope for the long message
- Receiver sends back ACK for long message
- Receiver receives envelope + message for the short message
- Sender receives ACK
- Sender sends envelope + message for the long message
- Receiver receives envelope + message for the long message

This is typically called the "overtaking" problem — although the long message was sent first, because of the use of the rendezvous protocol, the short message payload actually arrives first. MPI's message ordering guarantee states that the receives will match the order in which the sends were issued. Put simply, `msg1` must receive the long message and `msg2` must receive the short message (regardless of what or-

MPI Quiz

MPI has message ordering guarantees. But do you know what they are, exactly? Consider the following pseudocode, with an `MPI_COMM_WORLD` containing `size` processes:

```
1  if (rank == 0)
2    for (i = 0; i < size - 1; ++i)
3      MPI_Recv(buffer[i], ...,
       MPI_ANY_SOURCE,
       tag, MPI_COMM_WORLD);
4  else
5    MPI_Send(buffer, ...,
     0, tag, MPI_COMM_WORLD);
```

In what order will the messages be received at `MPI_COMM_WORLD` rank 0? See next month's column for the answer.

Rendezvous Protocols: Why Bother?

I am frequently asked why we bother using rendezvous protocols for sending messages — they're more complicated than eager sends (where the entire message is sent immediately) and, at a minimum, add one round-trip latency to the overall message transference time.

The answer is: because of limited resources. Particularly in operating system bypass networks (such as Myrinet and Infiniband), only "special" memory can be used to send or receive messages across the network. But only so much "special" memory is available — sending a single, enormous message may easily consume all of it and starve other message passing efforts.

Put even more simply — even for TCP networks — it is unattractive to send enormous messages without the receiver's permission. Consider sending a 32MB message eagerly. If the peer process has not posted a matching MPI receive, the MPI implementation will have to allocate a temporary 32MB buffer to receive it. When a matching receive is posted, the MPI application then has to copy the entire 32MB buffer and then deallocate the temporary buffer.

If this situation is repeated multiple times (e.g., multiple senders send enormous messages to a single receiver), the receiver can easily run out of memory. For MPI implementations that cannot guarantee that applications will not perform this way (and most cannot), rendezvous protocols prevent resource exhaustion at the receiver.

der they were actually sent across the underlying network).

The receiver must do appropriate bookkeeping to ensure that this condition is met. As a side effect of the sample code shown above, by the time the first `MPI_RECV` completes, the short message will have been received into a temporary buffer in the unexpected queue. So when the second `MPI_RECV` executes, all that is required is copying the message from the temporary buffer to the user's buffer.

Where to Go From Here?

A disclaimer on this column: the careful reader will notice that there were a *lot* of assumptions in the explanations given in this column. For each assumption listed above, there are real-world MPI implementations with different assumptions.

The number of assumptions provide some insight into why there are so many different MPI implementations: every system performs differently. Tweaking to receive the best possible performance is a compromise between the network behavior, operating system characteristics, management of finite resources, and behavior of the MPI application.

Next month, we'll explore the same MPI progression kinds of issues, but for non-sockets based kinds of networks (e.g., Myrinet, Inifiband). Such networks typically have lower latency and more tangible finite resources than TCP networks, so issues such as determining the short/eager message size become quite important.

Email Jeff at jsquyres@open-mpi.org

Resources

- **MPI Forum** (including the MPI-1 and MPI-2 specification documents): www.mpi-forum.org
- **MPI — The Complete Reference: Volume 1**, The MPI Core (2nd ed) (The MIT Press) by Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. ISBN 0-262-69215-5.
- **MPI — The Complete Reference: Volume 2**, The MPI Extensions (The MIT Press) by William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. ISBN 0-262-57123-4.
- **NCSA MPI tutorial:** webct.ncsa.uiuc.edu:8900/public/MPI

Advertisers' Index

The Advertisers' Index lists each company's Web address and advertisement page. To advertise in ClusterWorld Magazine, please contact adsales@clusterworld.com for a media kit containing an editorial schedule, rate card, and ad close dates. For subscription inquiries, please call 800-925-8215 or e-mail klwr@kable.com.

ClusterWorld Magazine is published monthly by InfoStrada LLC at 330 Townsend St. Suite 112, San Francisco, CA 94107. The U.S. subscription rate is $39.95 for 12 issues. In Canada and Mexico, a one-year subscription is $89.95 U.S. In all other countries, the annual rate is $119.95 US. Non-U.S. subscriptions must be pre-paid in U.S. funds drawn on a U.S. bank.

POSTMASTER: Send address changes to ClusterWorld Magazine, P.O. Box 592, Mt. Morris, IL 61054-0592