JEFF SQUYRES

Doing More with Less

Have you ever searched for "MPI" on EBay? It's fantastic! You can buy cars, pens, deuterium lamps, PVC piping, power transformers — all with MPI! I offer this as proof positive to the message passing naysayers: with MPI integrated into all of these common, real-world products, MPI is here to stay!

Pardon me; I have to go place some bids...

The Story So Far

Last month, we talked about some of the issues involved with a single-threaded MPI implementation making "progress" on message passing. Specifically, the discussion centered around TCP sockets-based implementations. Unavoidable sources of overhead were discussed, such as MPI envelopes and operating system TCP handling.

This month, we'll discuss operating system-bypass types of networks such as Myrinet, Infiniband, and Quadrics. Although we won't cover many specifics of these three networks, the message passing issues are fairly similar across all three — and quite different than TCP-based networks.

What Are OS-bypass Networks?

One source of the overhead involved in sending and receiving data from a network is the time necessary to trap into the operating system kernel. In some cases, this trap can occur multiple times, further increasing overhead. OS-bypass networks attempt to minimize or eliminate the costly trap into the kernel — bypassing the operating system and using communication co-processors located on the network card to handle the work instead of the main CPU.

Ignoring lots of details, this typically involves either modifications to the kernel or (more recently) loading a module into the kernel to provide both underlying support for user-level message passing and a device driver to talk to the network card. In the user's application (typically within the MPI library), message sending occurs by placing information about the outgoing message in a control structure and then notifying the communication co-processor to start the send. Incoming messages are automatically received by the co-processor and a control structure is placed in the user application's memory, where it can be found when the application polls for progress.

Differences from TCP

The design of OS-bypass networks make their usage quite different than TCP-based networks. Here are some of the differences:

1. Messages are transferred as single units; there are no partial sends and receives.
2. Message and the ordering between them may not be guaranteed.
3. All sent messages must have a corresponding pre-posted receive (no message can be unexpected).
4. Resources are limited; e.g., "special" memory may have to be used for all message passing.

Let's look at each of these in detail.

Single-unit Messages

Unlike the other three, this characteristic generally makes message passing *easier* than TCP. One of the most annoying aspects of writing a TCP-based progression engine is the fact that it has to maintain state about who "owns" the socket, how far along a read or write is in a given buffer, etc. Each pass through the progression engine has to survey all these values, progress them, and then update the internal accounting information. It's not rocket science, but it is quite cumbersome and annoying.

Newer generation networks present interfaces that transfer messages as atomic units. If you send N bytes, the receiver will receive N bytes - no more, no less. That's a whole lot of logic that does not need to be included in the MPI transfer layer (as compared to its TCP analogue).

Message [non-]Guarantees

TCP guarantees that any bytes sent will be received - there is never any need for retransmission. Other networks do not necessarily provide this guarantee. Packets can be dropped or otherwise corrupted (solar and atmospheric activity, believe it or not, can actually be a factor at high altitudes!); the MPI layer is typically the entity that has to watch for — and correct — these kinds of problems. This requirement translates into at least some additional amount of code and error conditions that the MPI layer has to handle.

Dropped packets have to be handled by the MPI implementation. Some networks' default handling of lost packets is to drop *all* packets to a given peer. The MPI layer must then effectively replay all the messages that it sent to that peer — potentially in the same order that they were originally sent (depending on how the MPI implementation's wire protocols work) — to recover.

Data corruption monitoring, although it seems like a desirable trait, is not something that most users will tolerate in an MPI implementation. Because [potentially expensive] data integrity checking needs to be inserted in the critical code path of message passing, this step, by definition, increases message passing latency. Combined with the fact that data corruption is rare, few MPI implementations support data integrity and simply assume that the underlying network will always deliver correct data.

MPI community wiki!

A new MPI community resource has recently opened its doors: a wiki all about MPI — l<*www.mpi-comm-world.org*>. The intent is to provide an MPI support site for the community, by the community. Any information about MPI is fair game - information about the standard, questions (and answers) about specific MPI implementations, etc. You are hereby cordially invited to submit information that you wish you had known when you started with MPI.

There is a password on the site to prevent spam defacements: "MPI_COMM_WORLD" (without the quotes). Enjoy!

No Unexpected Messages

Most low latency networks derive at least some of their speed from the fact that they can assume that all messages are expected — that there is a corresponding user-provided buffer ready to receive each incoming message (unexpected messages are errors, and can lead to dropped packets). The MPI implementation therefore needs to pre-post receive buffers (for envelopes at least), potentially from each of its peers.

Limited Resources

Some networks can only use "special" memory for all message passing. Myrinet, Infinband, and older versions of Quadrics need to use memory that has been registered with the network driver. One of the key actions that registering memory accomplishes is that the operating system "pins" down the virtual page(s) where the memory is located and guarantees that it(they) will never be swapped out or moved elsewhere in physical memory. This restriction allows the communication co-processor on the network card to DMA transfer to and from the user's buffer without fear of the operating system moving it during the process. Operating systems have limits on how much memory can be pinned; typically anywhere from ¼ to ½ of physical memory. The MPI implementation is usually responsible for managing registered memory.

However, since all receives must be pre-posted, the MPI layer must setup to receive some number of envelopes (possibly on a per-peer basis). Two obvious questions that come from this:

- How many envelopes should be pre-posted?
- How large should the envelopes be?

Pre-posting the buffers consumes system resources (e.g., registered memory), but posting too few buffers for envelopes can degrade performance. If too few are posted, flow control issues can consume too much processing power and cause "dead air" time on the network; if too many are posted, the system can run out of registered memory.

The size of the envelopes is another factor. MPI implementations commonly have three message sizes: tiny, small, and large. Tiny messages are included in the payload of the envelope itself, and can therefore be sent eagerly in a single network transfer. Small messages are also sent eagerly, but in a second network transfer (i.e., immediately after the envelope). Large messages are sent via rendezvous protocols.

Hence, the size of the envelope is really the *maximum* size of the envelope. Specifically, the MPI implementation will transfer only as much of the envelope as is necessary. For tiny messages, normally the header and the payload are sent; for short and long messages, only the header is sent. Choosing the maximum sizes for tiny and short messages is therefore a complicated choice (and will not be covered in this month's column).

Increasing the maximum size of envelopes can increase performance for applications that send relatively small messages. That is, if the user application's common message size is N, setting the maximum envelope size to be greater than or equal to N means that the application's messages will be sent eagerly with a single message transfer (subject to flow control, of course). But this also consumes system resources and can potentially exhaust available registered memory.

Short and Long Messages

Keep in mind that pre-posted envelopes are only one place where registered memory is consumed. Any messages that are not transferred as part of the envelope (i.e., short and long messages) need to operate in registered memory as well.

There are a variety of different flow control schemes to handle such issues. Here's one simplistic example:

- Sender: sends envelope to the receiver indicating "I've got a message of X bytes to send to you."
- Receiver: upon finding a matching MPI receive, registers the receive buffer (if it wasn't already registered) and sends back an ACK indicating "Ready to receive; send to address Y."
- Sender: sends the message to address Y.
- Sender: upon completion of the send, replies to the ACK with an envelope indicating "Transfer complete."
- Receiver: de-registers the receive buffer (if necessary).

Registering and de-registering memory is typically an expensive operation. MPI implementations typically expend a good deal of effort optimizing caching and fast lookup systems in an attempt to minimize time spent managing registered memory.

Progress

The good news is that once a message is given to the communication co-processor, it will progress "in the background" without intervention from the user application (and therefore from the MPI implementation). Since all messages are expected, it will eventually show up in a buffer and the network interface will inform the MPI layer (typically when the MPI implementation polls asking for progress). The MPI implementation will then process the buffer, depending on its type and content.

This aspect helps with asynchronous progress of tiny and short messages. The sender fires and forgets; the receiver will find the message has already arrived once a matching receive is posted. But this behavior does not necessarily help in rendezvous protocols — only single network messages are progressed independent of the main CPU. So the flow control messages described in the simplistic rendezvous protocol (above) are only triggered when the MPI implementation's progress engine is run. In a single threaded MPI implementation, this usually only occurs when the application enters an MPI library function.

Simply put: single-threaded MPI implementations receive a nice benefit from eagerly-sent messages when communication co-processors are used. They do not necessarily receive the same benefit when rendez-

See MPI, page 50

MPI Quiz

Last month, I asked in what order messages would be received at `MPI_COMM_WORLD` rank 0 from the following code:

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

Do MPI's message ordering guarantees provide any insight into the receipt order?

No. The answer is that since all the messages are coming from different processes, no fine-grained ordering between them is preserved by MPI (assuming that all messages were sent at roughly the "same" time) -- the use of `MPI_ANY_SOURCE` was somewhat of a red herring.

It is a not-uncommon problem to assume that this kind of code pattern (especially when using `MPI_ANY_SOURCE`) will receive messages "in order," meaning that `buffer[i]` will correspond to the message sent by `MPI_COMM_WORLD` rank `i`. While the above code is not a bad technique to avoid bottleneck delays from slow processes, it does not guarantee the source for any given `buffer[i]`. If guaranteeing the source is necessary, the following may be more appropriate:

```
if (rank == 0) {
 for (i = 0; i < size - 1; ++i)      Should be: i+1
   MPI_Irecv(buffer[i], ..., ~~MPI_ANY_SOURCE~~,
   tag, MPI_COMM_WORLD, &reqs[i]);
 MPI_Waitall(size - 1, reqs, MPI_STATUSES_IGNORE)
} else
  MPI_Send(buffer, ..., 0, tag, MPI_COMM_WORLD);
```

Next question...

Will the following code deadlock? Why or why not? What will the communication pattern be? How efficient is it?

```
left = (rank == 0) ? MPI_PROC_NULL : rank - 1;
right = (rank == size - 1) ? MPI_PROC_NULL : rank + 1;
MPI_Recv(rbuf, ..., left, tag, comm, &status);
MPI_Send(sbuf, ..., right, tag, comm, ~~&status~~);
```

MPI, from page 34

vous protocols are used (especially in conjunction with non-blocking MPI communication) because the MPI progress engine still has to poll to effect progress.

Where to Go From Here?

The same disclaimer from last month applies: 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 issues described in this column are one set of reasons why MPI implementations are so complex. Management of resources can sometimes be directly at odds with performance; the settings and management algorithms to maximize performance for one application may cause horrendous performance in another. As an MPI implementer, I beg you to remember this the next time you curse your MPI implementation for being slow.

Got any MPI questions you want answered? Wondering why one MPI does *this* and another does *that*? Send them to jsquyres@open-mpi.org.

Jeff Squyres is a post-doctoral research associate at Indiana University and is the one of the lead technical architects of the Open MPI project. 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
- **MPI Community Wiki:** www.mpi-comm-world.org