JEFF SQUYRES

In Parallel, Everyone Hears You Scream II

A guy walks into a breakfast joint with 16 penguins. They sit down at the biggest table in the place. The guy orders coffee for himself and a bowl of cereal for each of the penguins. He then breaks out a newspaper and casually starts reading. Meanwhile, the penguins' breakfasts arrive and the first penguin starts eating while all the others look at him. After he finishes, all the penguins look at the second penguin while he eats. When the last penguin finishes his cereal, he emits a loud "gwank!" and all the penguins get up and file out of the restaurant.

The guy looks up, folds up his newspaper, and gets up to pay the bill. One of the other patrons had been watching the spectacle said, "Excuse me sir, I have to ask. What was that all about? Why did you just sit there while your penguins ate their breakfast?"

"Yeah, it always takes this long," he said. "It's cerealized."

The Story So Far

Last month, we started my Top-10, All-Time Favorite Evils to Avoid in Parallel. As promised, it's *so* big that it takes two months to cover. We covered the first five last month:

10 Inconsistent environment/"dot" files

9 Orphaning MPI requests

8 `MPI_PROBE`

7 Mixing Fortran (and C++) compilers

6 Blaming MPI for programmer errors

So without further ado, from the home office in Bloomington, Ind., let's continue with No. 5...

5: Re-Using a Buffer Prematurely

Recall that MPI's message passing behavior is mostly defined through buffer semantics. Specifically, the MPI standard makes it clear that a buffer can only be used in *one* communication at a time. It is erroneous to use the same buffer in multiple, ongoing communications.

A common error is for MPI programs to start a non-blocking communication to or from a buffer and then start a second one with the same buffer before the first completes. There are two common cases: concurrent reading and writing, and multiple concurrent reads.

Simultaneous reading and writing to the same buffer is clearly a race condition. For example, if both a non-blocking send and a non-blocking receive are simultaneously posted to the same buffer, there is no guarantee in which order they will complete. Indeed, it may be impossible to know exactly what is sent because it will depend on exactly when the incoming message was received vs. when the outgoing message was actually able to be sent.

Multiple concurrent readers is frequently seen as harmless (i.e., sending from the same buffer); surely multiple *readers* can't cause a problem for the MPI implementation, can it?

Probably not.

But MPI still says that it's illegal, and it *may* cause problems — even if the sends all complete normally. The rationale here is that the MPI implementation may do something with "special" memory in order to maximize performance. For example, networks based on OS-bypass mechanisms may require the use of "pinned" memory — memory that the operating system is disallowed from swapping out. This requirement allows the OS-bypass-capable NIC to find the memory and be guaranteed that it doesn't move while the network transfer takes place.

An MPI implementation typically has to maintain some kind of state to keep track of pinned memory. While such techniques usually involve reference counting - the memory is not "un-pinned" by the MPI implementation until all communications involving it have completed - it is conceivable that an MPI implementation will *not* reference count or otherwise perform error checking in order to decrease overhead (and therefore decrease latency). This process can result in the premature "un-pinning" of memory while other communications are still ongoing, leading to run-time errors or other unpredictable behavior.

4: Mixing MPI Implementations

It is not uncommon for someone to ask me a question about LA-MPI, FT-MPI, MPICH, or one of several other MPI implementations. I always politely reply that I work on LAM/MPI, and can't really answer questions about those implementations. This situation is typically more amusing to me than anything else, but it underscores the issue that many users frequently do not distinguish between different MPI implementations.

This misconception unfortunately spills over to the technology as well; users compile their application with one implementation, try to run it with another, and are confused when it does not work. Or, worse, they run their application on multiple machines, each with a different implementation installed (this is similar to but slightly different from point #10: inconsistent environment/"dot" files). This situation is almost guaranteed not to work.

Additionally, some users assume that the `mpi.h` and `mpif.h` header files are interchangeable between MPI implementations (or do not make the distinction). They are not; indeed, the differences in these files are among the top-level reasons that MPI implementations are incompatible with each other (e.g., types, constants, and macros will likely have conflicting values in different implementations). Even worse, an MPI application may *compile* properly with the wrong `mpi.h` file, but then fail at run time in strange and mysterious ways.

The most common way to avoid this problem is to use the MPI implementation's "wrapper" compilers for compiling and linking applications. Most (but not all) MPI implementations provide commands such as `mpicc` and `mpif77` to compile C and Fortran 77 programs, respectively. These commands do nothing other than add relevant command line arguments before invoking an underlying compiler. They are typically the easiest way to ensure that the "right" `mpi.h` and MPI library are used when compiling and linking.

3: MPI_ANY_SOURCE

The use of `MPI_ANY_SOURCE` is convenient for programmers; it is not uncommon for a message with the same signature to be able to arrive from multiple sources. However, depending on the underlying network and the MPI implementation, this may force extra overhead upon receipt of the message. For example, the MPI implementation may be required to associate the receive request with all possible communication devices (which may entail spinning on polling all devices). When a matching message arrives, the MPI implementation must disassociate the request from all other devices. Not only does this cause extra latency simply by necessitating N actions, it may involve costly locking and unlocking mechanisms in multi-threaded programs.

When possible, try to avoid using `MPI_ANY_SOURCE`. Instead, it may be better to post N non-blocking receives - one for each source from where the message may be received. This arrangement allows the MPI to check only the relevant communication devices. Functions such as `MPI_WAITANY` and `MPI_TESTANY` can be used to determine when a message arrives. This situation is, of course, a trade-off — if you have a message that legitimately may arrive from *any* peer process, then `MPI_ANY_SOURCE` may actually be more efficient than posting N receives. Other factors also become relevant, such as the frequency of messages from each peer (including strategies to avoid unexpected messages) — it depends on the application.

LISTING ONE

Pseudocode of Communication and Computation Overlap

```
buffer_comm = A;
buffer_work = B;
for (...) {
  /* Send the communication buffer */
  MPI_Isend(buffer_comm, ..., &req);

  /* Do useful work on the other buffer */
  do_work(buffer_work);

  /* Finish the communication */
  MPI_Wait(&req, &status);

  /* Swap the buffers */
  buffer_tmp = buffer_comm;
  buffer_comm = buffer_work;
  buffer_work = buffer_tmp;
}
```

The real moral of the story is to understand your application and the run-time environment of the MPI implementation that you're using

2: Serialization

May users are nervous about using MPI's various modes of non-blocking communications and instead simply use `MPI_SEND` and `MPI_RECV`. This habit can lead to performance degradation by unknowingly serializing parallel applications. Processes blocked in `MPI_SEND` or `MPI_RECV` may be wasting valuable CPU cycles while simply waiting for communication with peer processes. This situation can even lead to a domino-like effect where a series of processes are waiting for each other and progress only occurs in a peer-by-peer fashion — just like the penguins in the beginning of this article.

This behavior can almost always be fixed in the application. While some algorithms simply cannot avoid this problem, most can be re-factored to allow a true overlap of computation and communication. Specifically: allow the MPI to perform message passing "in the background" while the user application is performing useful work. A common technique is to use multiple pairs of buffers, swapping between them on successive iterations. For example, in iteration N, initiate communication using buffer A and perform useful local work on buffer B. In iteration N+1, swap the buffers: communicate with buffer B and work on buffer A. See the pseudocode in *Listing One* for an example.

And the No. 1, All-Time Favorite Evil to Avoid in Parallel is...

1: Assuming MPI_SEND Will [Not] Block

In the February 2004 edition of this column, I included a sidebar entitled "To Block or Not To Block" describing typical user confusion as to whether `MPI_SEND` is supposed block or not. It still remains a popular question, frequently asked in multiple forms:

- "My application blocks in `MPI_SEND` — but only sometimes. Why?"
- "Why does my application work fine with Foo MPI, but deadlock in Bar MPI?"
- "When `MPI_SEND` returns, has the destination received the message?"

`MPI_SEND` and `MPI_RECV` are called "blocking" by the MPI-1 standard, but they may or may not actually block. Whether or not an unmatched send will block typically depends on how much buffering the implementation provides. For example, short messages are usually sent "eagerly" — regardless of whether a matching receive has been posted or not.

Long messages may be sent with a rendezvous protocol, meaning that it will not actually complete until the target has initiated a matching receive.

This behavior is legal because the semantics of `MPI_SEND` do not actually define whether message has been sent when it returns. The only guarantee that MPI makes is that the buffer is able to be re-used when `MPI_SEND` returns.

Receives, by their definition, will not return until a matching message has actually been received. If a matching short message was previously eagerly sent then it may be received "immediately" for example.

This case is called an "unexpected" message, and MPI implementations typically provide some level of implicit buffering for this condition: eagerly sent, unmatched messages are simply stored in internal buffering at the target until a matching receive is posted by the application. A local memory copy is all that is necessary to complete the receive.

Note that it is also legal for an MPI implementation to provide zero buffering — to effectively disallow unexpected messages and block `MPI_SEND` until a matching receive is posted (regardless of the size of the message).

MPI applications that assume at least some level of underlying buffering are not conformant (i.e., applications that assume that `MPI_SEND` will or will not block), and may run to completion under one MPI implementation but block in another.

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.

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.

NCSA MPI tutorial webct.ncsa.uiuc.edu:8900/public/MPI

Where to Go From Here?

There you have it — my canonical list of things to avoid while programming in parallel. Note that even though this is *my* favorite list, your mileage may vary — every parallel application is different. The real moral of the story here is to thoroughly understand both your application and the run-time environment of the MPI implementation that you're using. This understanding is the best way to obtain the best performance.

Next month, we'll launch into the nitty-gritty details of non-blocking communication. Stay tuned!

Jeff Squyres is a post-doctoral research associate at Indiana University and is the lead developer for the LAM implementation of MPI. Reach him at jsquyres@lam-mpi.org.