JEFF SQYURES

In Parallel, Everyone Hears You Scream

Parallel coding
Mysteries of MPI
Truly, life enow.

This, ladies and gentlemen, is what happens when a classical education collides with righteous code.

The Story So Far

So now you think you know MPI. We've covered a lot of ground in this column, including the MPI basics, startup and shutdown, collective operations, communicators and groups, and we just spent two fantastic months on datatypes (really, is there anything better?). This month, we'll start my Top 10, All-Time Favorite Evils To Avoid In Parallel. It's *so big* that it'll take two months to cover.

Many of these are common mistakes that either befuddle users or subtly cause performance degradation (and sometimes go unnoticed). Some of them are easy to explain, some are just due to how MPI implementations are typically crafted on the inside. Some have to do with running MPI programs, others have to do with writing them. It's a motley collection. From the home office in Bloomington, Ind., let's start with No. 10...

10: Inconsistent Environment/"Dot" Files

A common method of launching MPI applications - particularly across commodity Linux clusters - is with `rsh` (or `ssh`). Most new users to MPI simply invoke `mpirun` (or whatever startup command is relevant to your MPI implementation) and are surprised/dismayed/frustrated when it tries to invoke `rsh` (or `ssh`) behind the scenes and doesn't work. It doesn't matter which shell you use (they all work equally well with MPI), you must set it up to work properly with remote processes. The top two reasons why `rsh`/`ssh`-based MPI application startups fail are:

- The `PATH` environment variable is not set properly in the user's so-called "dot" files (e.g., `.tcshrc`, `.profile`, or `.bashrc` -- the specific file name depends on which shell you are using). Specifically, you may need to set the `PATH` in your "dot" file to include the directory where your MPI installation is installed on the remote nodes; *it may not be sufficient to set the* `PATH` *in the shell where you invoke* `mpirun`.

- Remote authentication and/or `rsh`/`ssh` is not setup properly. Error messages such as "Permission denied" typically indicate that the user has not setup remote logins properly (e.g., a `.rhosts` file or SSH keys). Error messages such as "Connection refused" usually mean that remote logins using a specific protocol are not enabled (e.g., trying to use `rsh` in a cluster where only `ssh` remote logins are enabled).

Both of these kinds of errors are show-stoppers; you won't be able to run MPI programs until they are solved. Usually a few Google searches will find the right answer. If all else fails, seek out your local neighborhood system administrator for advice.

9: Orphaning MPI Requests

When using non-blocking MPI communication (i.e., you tell MPI to *start* a communication), MPI gives you back a request that you can use later to find out if the communication has completed. It is important to always poll MPI later and see if it has completed. Not only is this necessary so that you can know when you re-use your message buffer, MPI allocates resources to track non-blocking communications that are not released until the user application is notified that it has completed.

The moral of the story: if you start a non-blocking communication and then never check the request for completion, your application is leaking resources. Always, always, always remember to poll for completion of non-blocking communications.

8: MPI_PROBE

For a specific (tag, source rank, communicator) triple, the `MPI_PROBE` function returns when a message matching that triple is ready to be received (a similar non-blocking version is available as well: `MPI_IPROBE`) and reports, among other values, the size of the pending incoming message. `MPI_PROBE` is commonly used to receive variable-length messages — where the receiver does not know how large the message is that will be received. For example, post an `MPI_PROBE` and then use the size that is returned to allocate a buffer of the correct size and then `MPI_RECV` into it.

Although convenient, `MPI_PROBE` (and `MPI_IPROBE`) may actually force the MPI implementation to allocate a temporary buffer and fully receive the message into it before reporting its size. Hence, when the matching receive is finally posted, the MPI implementation simply performs a memory copy to transfer the message to the user's buffer (and then frees the temporary buffer). This can add significant latency, particularly for large messages or in low-latency networking environments.

Avoid the use of `MPI_PROBE` when possible. It may be more efficient to actually send *two* messages: first send a fixed-size message that simply contains the size of the second message, then immediately follow it with the real message. This method prevents the MPI implementation from needing to allocate temporary buffers and perform unnecessary memory copies.

7: Mixing Fortran (and C++) Compilers

This problem is not so much a problem with MPI as it is the state of compiler technology. Fortran compilers may resolve global variables and function names differently. For example, the GNU Fortran 77 compiler silently transforms the name to lower case and appends two underscores to all global variable and function names. This is in contrast to, for example, the Solaris Forte Fortran compiler only adds *one* underscore. It is possible that an MPI implementation was configured for a specific Fortran compiler's resolution scheme. Hence, functions such as `MPI_INIT` may actually be exist as `mpi_init__`.

As a direct result, your MPI implementation may be configured to only work with a single Fortran compiler (which is only relevant if you are writing Fortran MPI programs). Attempting to use a different Fortran compiler may result in "Unresolved symbol" kinds of errors when attempting to link MPI executables.

To fix this, either only use the Fortran compiler that your MPI installation was configured with, or re-configure/re-install your MPI with the Fortran compiler that you want to use. The issue is almost identical for C++ compilers (similarly, this is only relevant if you are writing C++ MPI programs that use the C++ MPI bindings).

6: Blaming MPI for Programmer Errors

A natural tendency when an application breaks is to blame the MPI implementation, particularly when your application "works" with one MPI implementation and (for example) seg faults in another. While no MPI implementation is perfect, they do typically go through heavy testing before release. It is quite possible (and likely) that your application actually has a latent bug that is simply not tripped on some architectures/MPI implementations.

This sounds arrogant (especially coming from an MPI implementer), but the vast majority of "bug reports" that we receive are actually due to errors in the user's application (and sometimes they are very subtle errors). For example, some compilers initialize variables to default values (such as zero). Others do not. If your code accidentally depends on a variable having a default value, it may work fine under some platforms or compilers, yet cause errors on others.

Before submitting a bug report to the maintainers, double and triple check your application. Use a memory-checking debugger, such as the Linux Valgrind package, the Solaris `bcheck` command-line checker, or the Purify system. All of these debuggers will report on the memory usage in your application, including buffer overflows, reading from uninitialized memory, and so on. You'd be surprised what will turn up in your application.

Resources

Valgrind project
- valgrind.kde.org

MPI Forum
- www.mpi-forum.org

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

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.

Where to Go From Here?

So what did we learn here?

10 Ensure your environment is set up correctly. You only need to do this once.

9 Always check non-blocking communication for completion. Don't leak resources.

8 Avoid `MPI_PROBE` and `MPI_IPROBE`; they're evil.

7 Ensure that you are using the right compilers.

6 Don't blame MPI for your errors. Use memory-checking debuggers.

If anything, realize that you are not alone if you run into MPI problems. The problems discussed this month are all relatively easy to fix. So even if you can't get your MPI application to run, don't despair. The solution is probably just a few Google searches or a system administrator away.

Stay tuned: next month, we'll continue with my Top 5 All-Time Favorite Evils to Avoid in Parallel.

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