# MPI Debugging: Can You Hear Me Now?

It's been said that if debugging is the process of removing bugs from a program, then programming must be the process of putting them in. Although I don't personally write bugs (ahem), I understand from others that they can be quite difficult to track down and fix. My personal recommendation is to avoid writing them in the first place; steer clear of popular assembler instruction such as "BFM" (branch on full moon), "WTR" (write to random), and "MM" (more magic).

## The Story So Far

Over the course of the past year, we have talked about all kinds of aspects of MPI — ranging from what MPI is all the way through advanced communication patterns and techniques. Presumably, loyal readers who have been following this column are now among the World's Greatest MPI Hackers (and should feel free to add "Member, WGMH" to resumes and vitae).

But now that you're a hotshot MPI programmer, the reality of writing code sets in: debugging. Debugging any application can be a difficult and arduous task; multiple orthogonal dimensions of factors combine to create unique — and sometimes elusive — unforeseen circumstances. Assumptions in code can therefore turn out to be false leading to system crashes, corrupted output, or, even worse, subtly incorrect answers.

Parallel applications add several more dimensions to the mix — bugs can be the result of complex interactions between the individual processes in a parallel job. Typical parallel bugs manifest themselves in race conditions, unexpected or un-handled messages, and the ever-popular deadlock and *live-lock* scenarios.

While many of these errors can also occur in serial applications, the fact that these scenarios can simultaneously occur in one or more processes of a parallel job dramatically increases the difficulty of isolating exactly what the bug is. Even after the bug is identified, understanding the events causing it to occur can be equally difficult, because it may be the direct (or indirect) result of interactions between multiple (semi-)independent processes. Hence, examining the state of the single process where the bug occurred (perhaps through a core dump file) may be neither sufficient to understand *why* it happened, nor how to fix it.

In short, parallel bugs span multiple processes. This condition is, unfortunately, simply the nature of parallel computing; it is not specific to MPI applications. Multi-process bug dependency can create a multiplicative effect both in terms of system complexity and difficulty in tracking down even simple problems.

All that being said — fear not! Parallel applications, just like serial applications, only do exactly what they are told to do. They are discrete creatures that, even though they seem to be devious and malicious (particularly to the programmer who is debugging them), are bound by finite rules and operating procedures. For every bug, there is a reason. For every reason, there is a bug fix.

This month we'll examine some of the more popular and some of the more effective techniques of parallel debugging (note that "popular" is not always the same as "effective"!).

## printf Debugging

Using `printf` is perhaps one of the most common forms of debugging. Want to know what this variable is at that point in the program? Put in a `printf`! What to check and see if that conditional was taken? Put in another `printf`!

Variations on this theme include sending `printf`-like output to files for postmortem analysis, selectively enabling and disabling specific classes of output, and compile-time disabling all debugging output. The end result is inevitably the same: an ever-growing set of output messages that must be sorted through in the hopes that one or more of the messages will reveal the exact location and/or conditions where bugs occur.

`printf`-style debugging can be even less effective in parallel because of the multiplicative effect: every process will print messages, resulting in potentially *N* times as much output to sort through. Worse, because standard output is not always guaranteed to be in order by process (or may even be delayed until `MPI_FINALIZE`), output may be interleaved from multiple sources.

All of these can be solved with workarounds (e.g., labeling each output message with the process' rank in `MPI_COMM_WORLD`, sending the output of each process to a different file, etc.), but still results in the same problem: `printf`-style solutions can only display a limited subset of the process' state.

If the programmer wants to display something else, the application must be edited, re-compiled, and re-run.

Even worse, inserting `printf` (and friends) can change an application so that the bug does not occur, or, even worse, changes symptoms. This situation can be quite common in MPI applications, especially with bugs that are due to race conditions. For example, a `printf` can slow one process down just enough that the bug mysteriously disappears (remember that sending to the standard output is a relatively time-consuming task).

## Using Serial Debuggers in Parallel

Remember what your Introduction to Programming instructor told you: use a debugger! Debuggers allow you to single-step through your running application, examining just about anything in the process. Even a serial debugger can be useful in parallel. Most vendor-provided compilers, as well as the well-known GNU debugger (`gdb`) and its GUI counterpart the Data Display Debugger (`ddd`) can be used with parallel applications in one of two ways.

The first method is to initially launch the MPI processes under the serial debugger (if the MPI implementation supports it). For example, the following works in LAM/MPI:

```
$ mpirun -np 2 xterm \
  -e gdb my_mpi_application
```

This command will launch two `xterms`, each of which will launch a copy of `gdb` and load `my_mpi_application` (note that this example assumes that you have proper X authentication between the nodes that you are running on; see the LAM/MPI FAQ for more information). You can then individually control each MPI process. This method is quite helpful, but only for small to mid-sized parallel runs; attempting to use 32 `gdbs` in 32 `xterms` can be quite difficult to manage.

This concept can be extended (again, if the MPI implementation supports it) by `mpirun`'ing a script instead of an `xterm`:

```
$ mpirun -np 32 my_debug.sh
  my_mpi_application
```

where `my_debug.sh` only launches a debugger for a subset of the processes, not all of them. Hence, although the MPI job consists of 32 applications, you can set `my_debug.sh` to only launch a debugger on `MPI_COMM_WORLD` ranks 0, 1, and 2.

Note, however, that this method will only work for a single run. Once the processes complete, you will need to exit `gdb` and re-`mpirun` the MPI job. You cannot re-run the application from within `gdb`.

The second method of using a serial debugger is to attach to an already running MPI process. This method is more portable than the first because debuggers can always attach to processes; it does not depend on the capabilities of the MPI implementation.

A common scenario where this is helpful is in a deadlock scenario — if your MPI application "hangs" for no apparent reason, you can at-

### Why Bother With Parallelism?

Given the potentially significant increase in difficulty of debugging parallel applications (as compared to debugging serial applications), why bother writing in parallel? What would justify the added time, resources, and expense required to obtain a correctly functioning parallel application?

The rationale behind parallel computing is that it enables two main things:

- Applications that are too large to fit on a single machine
- Decreased execution time as compared to serial applications

Oil companies, for example, gather vast datasets from surveying equipment to determine where to drill. Such datasets can reach into the terabyte realm — far too large to fit in the RAM of single node at a time. A serial application would have to iteratively request subsets of the data, store partial results, and later combine them into a final result. By definition, this process is (at least) linear. Extending this scenario to allow multiple copies of the processing application to simultaneously request and process subsets of the data can result in a significant decrease of execution time. Hence, what used to take weeks to run can now be accomplished in hours.

Such gains can also be translated into increasing the resolution or accuracy of the results. For example, consider a serial process that takes 1,000 hours (41 days). Say that running this process in parallel on a large cluster takes 10 hours to get the same results. With so much speedup, it seems natural to increase the resolution of the computation. Perhaps it will take 20, 40, or 80 hours to obtain the finer-grained results, but it is still significantly less than 41 days.

Bottom line: although some up-front investment is required, parallel solutions can result in better results in less time — a competitive advantage that directly impacts the overall cost of a project.

tach a serial debugger to one (or more) of the MPI processes and simply see what it is doing.

Another typical example is where one MPI process repeatedly crashes (e.g., rank 3 in `MPI_COMM_WORLD`). Use code similar to the following:

```
int rank;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,
              &rank);
if (rank == 3) {
  int i = 0;
  printf("PID %d waiting\n",
         getpid());
  fflush(stdout);
  while (i == 0)
    sleep(10);
}
```

This code causes `MPI_COMM_WORLD` rank 3 to loop forever. However, it prints out its PID, enabling you to attach a debugger to it, change the value of the variable `i`, and continue single-stepping through the process to examine the events leading up to the crash.

Admittedly, this is not an elegant approach, but it is quite practical.

### 1-800-DBUG-HPC

Debugging is a frustrating, maddening task. Run the application. Watch the crash. Examine the core dump. Make a fix. Compile the application. Run, watch, examine, fix, compile. Repeat. Repeat. Repeat...

Here's a few of my favorite mantras that I like to repeat while working out particularly nasty bugs:

- A computer is a labor-saving device
- Asynchronous behavior is my friend
- Computers do not exhibit malice
- MPI means Misbehaving Program Interface
- I love my job
- I will bend the computer to my will

### Resources

**LAM/MPI FAQ** (more information on debugging in parallel):
- www.lam-mpi.org/faq

- **Valgrind:** valgrind.kde.org
- **Purify:** www-306.ibm.com/software/awdtools/purify/unix
- **DDT:** www.streamline-computing.com
- **Totalview:** www.etnus.com
- **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

## Memory-Checking Debuggers

Memory-checking debuggers are the greatest thing since sliced bread. Once you start using memory-checking debuggers, you'll wonder how you programmed without them. In addition to identifying all the "normal" causes of cashing that regular debuggers provide (segmentation faults, bus errors, etc.), memory-checking debuggers look for erroneous patterns such as accessing memory outside of an array or the local stack, using heap memory that was already freed, freeing memory that was already freed, using uninitialized variables, and so on. Best of all, they will tell you these things by file and line number in your source code.

Popular memory-checking debuggers include Valgrind (Linux), bcheck (Solaris, part of the Forte compiler suite), Rational Purify (a commercial product available for several operating systems), and various forms of "malloc debug" (e.g., Mac OS X has native support). Others are also available.

Most memory-checking debuggers are typically intended to be used non-interactively and cannot be attached to already-running processes. As such, depending on your MPI implementation, they can only

## Should I Compile My MPI With "-g"?

Many users ask me if they need to compile their MPI implementation with "-g" to enable them to debug their MPI application.

The answer is no. The application being debugged must be compiled with "-g", however. The MPI implementation itself should not be compiled with "-g"; this will disallow the debugger from stepping into MPI functions. Hence, if you try to "step" into `MPI_SEND`, the debugger won't let you and will likely execute the entire `MPI_SEND` function call. This result is what most users want anyway — you're attempting to debug your application, not the MPI implementation.

If you compile your MPI implementation with "-g", you'll be able to step into functions such as `MPI_SEND`, but this may not provide as much useful information as you would think — the internals of an MPI implementation are quite complex.

Note that this sidebar really only applies to MPI implementations that provide their source code; binary-only MPI implementation distributions are most likely compiled without "-g" (and, by definition, the debugger will not be able to find the source code to display).

be launched via `mpirun`. For example:

```
$ mpirun -np 2 valgrind \
  -num-callers=100 \
  -tool-memcheck \
  -leak-check=yes \
  -show-reachable=yes \
  -log-file=output my_mpi_app
```

This command will run two copies of Valgrind, which will, in turn, each launch a copy of `my_mpi_app`. Each of the Valgrind instances will monitor their child process and send their output to a file named `foo.pid[PID]`. After the application completes, the `foo` files can be examined to see the errors that Valgrind detected.

Note that LAM/MPI should be configured with the *-with-purify* switch to be used with memory-checking debuggers. This switch eliminates many false positives at the expense of a slight performance loss (i.e., LAM uses some optimizations that are known to be safe, but tools such as Valgrind will interpret them as reading from uninitialized memory).

Although memory-checking debuggers cannot catch *all* errors, they can help find a *lot* of errors even before you know that they exist (even for serial applications). My own personal experience has shown that it can extremely helpful to use memory-checking debuggers frequently during an application's development — even when you are not aware of any current bugs.

## Parallel Debuggers

Finally, there are debuggers specifically created to operate on parallel MPI applications. Two commonly-used suites are the Distributed Debugging Tool (DDT) from Streamline Computing and Totalview from Etnus. These packages have the significant advantage over the prior approaches in that they can natively understand an entire parallel process.

Specifically, in addition to all the normal functionality of a debugger (setting breakpoints, examining variables, stepping through code, etc.), you can individually monitor and control all processes in a running MPI job.

For example, you can step through the code in process A (while blocking all other processes) and watch a message being sent. Then you can step through process B and watch the message being received. In this manner, you have complete control over the entire parallel job.

This kind of tool is invaluable for serious parallel application development, but tend to be somewhat expensive. If you can afford them, parallel debuggers are extremely helpful tools.

Memory-checking debuggers are the greatest thing since sliced bread. Once you start using memory-checking debuggers, you'll wonder how you programmed without them

## Where to Go From Here?

The moral of this column is: fear not. There really is more to parallel debugging than `printf`, Virginia. Debugging is a tricky task, but using the proper tools can greatly reduce the task to something that is manageable.

Next month, we'll continue the debugging discussion and describe some common MPI programming errors and how you can use the techniques described here to find them.

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 him at jsquyres@open-mpi.org.*