Debugging in Parallel (in Parallel)

Thus spake the master programmer: "Though a program be but three lines long, someday it will have to be maintained."

The Story So Far

Last month we started discussing the cold, hard reality of high performance computing: debugging in parallel. Like Weird Uncle Joe, no one wants to talk about it (every family has a Weird Uncle Joe). All the same kinds of nasty bugs that can happen in serial applications can also happen in parallel environments — magnified many times because they can happen in any process in a parallel job. Even worse, bugs can be the result of complex interaction between processes and possibly occur in separate processes simultaneously. Simply put: parallel bugs typically span multiple processes. The analysis of a single core file or subroutine may not yield the root causes behind a bug.

But to repeat myself from last month: fear not. For every bug, there is a reason. For every reason, there is a bug fix. Using the right tools, you can spin a web to catch bugs.

Last month we briefly discussed four parallel debugging techniques:

- `printf`-style output debugging
- launching serial debuggers in parallel
- attaching serial debuggers to individual parallel processes
- using parallel debuggers

The last one — parallel debuggers — extends the traditional serial debugger concept by encompassing all the processes in a parallel job under a single debugging session. A variety of commercial parallel debuggers are available. This column is not an advertisement, so I won't be displaying screen shots or reviewing the functionality of these products — you can visit their web sites and see the material for yourself. But suffice it to say I strongly recommend the use of a parallel debugger (see the Resources sidebar for more information).

That being said, most of us don't have access to parallel debuggers, so this month we'll concentrate more on the common-man approach to parallel debugging.

Debugging A Classic MPI Mistake

As mentioned several times previously in this column, the following is a fairly common MPI programming mistake when exchanging messages between a pair of processes:

```
1 MPI_Comm_size(comm, &size);
2 MPI_Comm_rank(comm, &rank);
3 if (2 == size) {
4  peer = 1 - rank;
5  MPI_Send(sbuf, ...,
            peer, comm);
6  MPI_Recv(rbuf, ...,
           peer, comm, &status);
7 }
```

An MPI implementation may perform the send on line 5 "in the background," *but is also allowed to block*. Many MPI implementations will implicitly buffer messages up to a certain size before blocking; if the message sent on line 5 is less than N bytes, the send will return more or less immediately (regardless of whether the message has actually transferred to the receiver or not). But once the message is larger than N bytes, the implementation may block in a rendezvous protocol while waiting for the target to post a matching receive. In this case, the code above will deadlock.

The solution is simple: have one process execute a send followed by a receive; have the other execute a receive followed by a send. But the problem is still the same: this error may be buried in many thousands (or millions) of lines of code. Assuming that the messages are large enough to force the MPI implementation to block, how would one *find* this problem in the first place?

Depending on the logic of the overall application, a binary search with `printf`-style debugging can probably [eventually] locate the bug in some finite amount of time. In the final iterations of the search, inserting `printf` statements before and after the `MPI_SEND` would likely positively identify the problem (i.e., the first `printf` message would be displayed, but the second would not).

The same result, however, can be obtained in far less time by using a debugger. `printf`-style debugging, by definition, is trial-and-error — think of it as searching for the location of the bug, as compared to a debugger which (at least in this case) can directly query "where is the bug?"

Launching a serial debugger in parallel, for example:

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

will launch 2 xterms, each running a GNU debugger (`gdb`) with your

MPI application loaded. In this case, you can run the application in both `gdb` instances and when it deadlocks, hit control-C. The debugger will show that both processes are stuck in the `MPI_SEND` on line 5. There is no guesswork involved.

Note that this example assumes that your MPI implementation allows X applications to be run in parallel. This task is easy if you are running on a single node (in which case X authentication is usually *automagically* handled), or, if running on multiple nodes, either X authentication is either disabled or setup such that X credentials are passed properly. Consult your MPI implementations documentation for more details — not all MPI implementations support this feature.

A slightly simpler, albeit more manual, method is to `mpirun` the MPI application as normal. When it deadlocks, login to one or more nodes where the application is running and attach a debugger to the live process. This example assumes Linux `ps` command line syntax:

```
$ mpirun -np 2 my_mpi_app &
$ ssh node17 ps -C my_mpi_app
  PID TTY      TIME CMD
 1234 ?   00:00:12 my_mpi_app
```

You'll need to use the "attach" feature of your debugger. With `gdb`:

```
$ ssh node17
Welcome to node17.
$ gdb -pid 1234
```

This action will attach the debugger to that process and interrupt it. You can list where the program counter is, view variables, etc. As with the case above, it will immediately identify that the application is stuck in the `MPI_SEND` on line 5.

Serialized Debugging

It is somewhat of an epiphany to realize that once applied in parallel, debuggers can be just as powerful — if not more so — than when used in serial. Consider other common MPI mistakes: mismatching the tag or communicator between a send and receive, freeing or otherwise modifying buffers used in non-blocking communications before they have been completed with `MPI_TEST` or `MPI_WAIT` (or their variants), receiving unexpected messages with `MPI_ANY_SOURCE` or `MPI_ANY_TAG`, and so on. All of these can be caught with a debugger.

> Remember that the bug(s) may span multiple processes — it is frequently not enough to examine a single process.

Debuggers can be used to effectively serialize a parallel application in order to help find bugs. By

To printf or not to printf?

I'll begin by saying that you should not use `printf` as a debugging tool. However, I know that most everyone will ignore me, so you might as well be aware of some potential "gotchas" that occur with `printf` when running in parallel.

Remember that the node where your `printf` was issued may not be the same node as where `mpirun` is executing (or whatever mechanism is used to launch your MPI application). This condition means that the standard output generated from `printf` will need to be transported back to `mpirun`, possibly across a network. This process has three important side effects:

1. The standard output from `printf` will take some time before it appears in `mpirun`'s standard output,
2. Standard output from `printf`s in different processes may therefore appear interleaved in the standard output of `mpirun`, and
3. Individual `printf` outputs may be buffered by the run-time system or MPI implementation.

The last item is the most important: many a programmer has been tricked into thinking that sections of code did not execute because they did not see the output from an embedded `printf`. Little did they realize that the code (and the `printf`) *did* execute, but the output of `printf` was buffered and not displayed. Although most MPI implementations make a "best effort" to display it, remember that the behavior of standard output and standard error is not defined by MPI. Some implementations handle it better than others.

If you are going to use `printf` debugging, it is safest to follow all `printf` statements with explicit `fflush(stdout)` statements. While this statement does not absolutely guarantee that your message will appear, it usually causes most MPI implementations and run-time systems to force the message to be displayed.

stepping through individual processes in the parallel job, a developer can literally watch a message being sent from one process and received in another. If the transfer does not occur as expected, the debugger provides the flexibility to look around to figure out why (e.g., the tags did not match). And even if the message does transfer properly, buffers can be examined to ensure that the received contents match what were expected.

Memory-Checking Debuggers

This type of "serialized debugging" is useful to catch flaws in logic and other kinds of [relatively] obvious errors in the application. Ensnaring more subtle bugs such as race conditions or memory problems can be trickier. Indeed, the timing and resource perturbations introduced by running through a debugger can sometimes make bugs mysteriously disappear — applications that consistently fail under normal running conditions magically seem to run perfectly when run under a debugger.

The first step in troubleshooting such devious bugs is to run your application through a memory-checking debugger such as Valgrind. Consider the code in *Listing One*. Despite the several obvious problems with this code, it may actually run to completion without crashing (writing beyond the end of the `j` array is probably still within the allocated page on the heap and will likely not cause a segmentation violation).

Now consider that if code as obviously incorrect as *Listing One* can run seemingly without error, imagine applications that are much larger and more complex than this trivial example — there are bound to be errors similar to the ones shown in *Listing One* hidden within thousands (or millions) of lines of code.

LISTING ONE

Multiple Memory Maladies

```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3 #include <mpi.h>
 4 int main(int argc, char* argv[]) {
 5  int rank, size, i, *j;
 6  MPI_Init(&argc, &argv);
 7  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 8  MPI_Comm_size(MPI_COMM_WORLD, &size);
 9  j = malloc(sizeof(int));
10  MPI_Send(&i, 2,
              MPI_INT,
              (rank+1) % size,
              123,
              MPI_COMM_WORLD);
11  MPI_Recv(j, 2,
              MPI_INT,
              (rank+size-1) % size,
              123,
              MPI_COMM_WORLD,
              MPI_STATUS_IGNORE);
12  MPI_Finalize();
13  free(j);
14  free(j);
15  return 0;
16 }
```

Memory-checking debuggers are excellent tools in both parallel and serial applications. Compile and run *Listing One* through Valgrind:

```
$ gcc example.c -g -o example
$ valgrind -tool=memcheck \
   -logfile=valoutput example
```

Valgrind will show several distinct errors (one output per MPI process, named `valoutput.pid[pid]`:

1. Use of uninitialized variable on line 10.
2. Illegal read on line 10.
3. Illegal write on line 11, 4 bytes beyond the array allocated on line 9.
4. Duplicate free on line 14.

Postmortem Analysis

Postmortem analysis is a tool that is frequently overlooked. Those annoying core files that most people either ignore or remove can actually be loaded into a compiler to view a snapshot of the process just before it crashed. Even if race conditions disappear when run under debuggers, core files can still be examined from failed runs; depending on the nature of the error and the operating system's settings, it is not uncommon to get a core file for each failed process in the parallel job. Examining all the core files can provide insight into the cause(s) of a bug.

Intercepting Signals

If all else fails, it may be desirable

LISTING TWO

Sample Segmentation Fault Catcher

```
#include <stdio.h>
#include <signal.h>
#include <mpi.h>

static void handler(int);
static char str[MPI_MAX_PROCESSOR_NAME + 128];
static int len;

/* Setup a string to output */
void setup_catcher(void) {
  char hostname[MPI_MAX_PROCESSOR_NAME];

  MPI_Get_processor_name(hostname, &len);
  sprintf(str, "Seg fault: pid %d, host %s\n",
          getpid(), hostname);
  len = strlen(str);
  signal(SIGSEGV, handler);
}

/* write() the string to stderr
   then block forever waiting for
   a debugger to attach
 */
static void handler(int sig) {
  write(1, str, len);
  while (1 == 1);
}
```

Resources

- Etnus Totalview Parallel Debugger — www.etnus.com/index.html
- Streamline DDT Parallel Debugger — www.streamline-computing.com
- LAM/MPI FAQ — www.lam-mpi.org/faq (more information on debugging in parallel)
- Valgrind — 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. 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.

- The Tao of Programming by Geoffrey James. ISBN 0931137071.

to install a signal handler to catch segmentation faults (or whatever signal is killing your application) and print out the node's name and the process' PID. Be careful, however — very little can be safely executed in signal context. *Listing Two* shows an example of setting up a printable string ahead of time; the signal handler itself only invokes `write()` to output the string and then goes into an infinite loop to wait for a debugger to attach. This method potentially avoids the overhead and possible race condition timing changes caused by active checking in debuggers, increasing the chance of duplicating the bug, and therefore being able to catch it in a debugger.

Where to Go From Here?

Debugging in parallel is hard... but not impossible. Although it shares many of the characteristics of serial debugging, and although many of the same tools can be used (in creative ways), parallel debugging must be approached with a whole-system mindset. Remember that the bug(s) may span multiple processes — it is frequently not enough to examine a single process in a parallel job. And always always always use the right tool. `printf` is rarely the right tool.

Next month, we'll discuss some of the dynamic process models of MPI-2 — spawning new MPI processes.

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.