Debugging MPI programs with the GNU debugger

Version 1.1.0

Tom Fogal

Originally written: December 27th, 2013.
Last update: February 19, 2014.

Abstract

We often see people attempting to debug MPI-parallel programs using '`printf` debugging'. While this is very simple to set up, it is a poor substitute for a real debugger.

Back when the author used to work on VisIt[1], parallel debugging was of paramount concern, as work was mostly focused within the domain of parallel rendering. Therefore we added some support for getting a debugger attached at the appropriate time, and used it quite profitably. While the technique is simple, it does not seem anybody else ever picked up on it. The author still sees people in other contexts trying to debug confusing parallel segmentation faults using traditional methods.

Here we expound a pattern that allows one to debug parallel jobs using a real debugger.

The tl;dr version of the solution is: pause job execution until the debugger is up and running.

[1] `http://www.visitusers.org/`

Chapter 1

GNU debugger and MPI programs

There are two major issues with debugging in a parallel environment such as MPI:

- **There are multiple processes**, and unlike with threads there is a **strong '1 debugger 1 process' mapping** with current tools.
- **The MPI system handles process creation and execution**, tasks which the debugger traditionally wants to apply. The **workaround**—attaching the debugger after process startup—suffers typical **race condition** issues.

There are some debuggers which can debug parallel programs as a group. Notably, TotalView boasts this kind of support. This feature is still lacking in open source debuggers.

We solve these issues by writing our code such that one *must* attach a debugger and twiddle a bit to get the program to continue. This may be obvious to anyone with working knowledge of the GNU debugger, your shell, and C, but the author finds that many people are missing one critical piece of that puzzle.

1.1 Attaching a debugger

An ill-used feature of the GNU debugger is the ability to `attach` to an already-running process[1]. The functionality takes a process ID as an argument to attach to:

```
(gdb) attach 12345
```

attaches to process '12345'.

The solution, then, is to start the MPI job and then 'quickly' attach the debugger from another terminal[2]. The trouble is that 'quickly' is rather difficult: frequently the program crashes before we can get it under debugger control. Some use a solution based on `sleep(3)` to ensure they will have enough time to get the debugger attached. This is undesirable: if we set the argument too small, we might 'miss' the process' bad behavior. If we set it too large, then we have to wait excessively after it attaches, which increases our iteration time.

What we really want is to just have the MPI job *automatically* wait until we have attached the debugger, at which point it should continue instantaneously.

1.2 Process modification

Another ill-used feature of the GNU (and many other) debuggers is the ability to *modify* arbitrary memory locations of a running process. The command in gdb is `set variable`:

```
(gdb) set variable v=42
```

of course, the variable must exist in the current scope of where gdb is stopped.

[1]Note that many Linux distributions are now disabling this functionality at the kernel-level by default. The error message you receive when attempting to attach to a running process explains how to re-enable the required support.

[2]**Note**: if you are running on a cluster, you may have to ssh to a backend node first. Identifying which nodes your job is running on is cluster-specific; ask your local IT people.

1.3 Wait for the debugger

We can use the '`attach`' and '`set variable`' features together to control the execution of the MPI job. We make our program wait on a variable which we set upon attaching the debugger. I do this with a method I call `wait_for_debugger`:

```
#define ROOTp() /* ... true iff rank == 0 */
static void
wait_for_debugger()
{
  if(getenv("TJF_MPI_DEBUG") != NULL && ROOTp()) {
    volatile int i=0;
    fprintf(stderr, "pid %ld waiting for debugger\n"
       , (long)getpid());
    while(i==0) { /* change 'i' in the debugger */ }
  }
  MPI_Barrier(MPI_COMM_WORLD);
}
```

We insert `wait_for_debugger` very early in our program's execution: perhaps immediately after `MPI_Init`. The debugging is controlled by an environment variable, in this case `TJF_MPI_DEBUG`. When the environment variable is not set, then we execute a quick barrier and continue on. This is just a single barrier that happens right at job start, and thus the method is lightweight enough that you can leave the call to '`wait_for_debugger`' in your code at all times.

If the environment variable is set[3], however, then our root process will enter the body of the `if` statement. Every other process will end up waiting at the `Barrier`. Inside the `if`, we produce what looks like an infinite loop, and indeed in some sense it is.

While the job is held up in this loop, we jump into another terminal and use the debugger to 'unstick' the process group:

```
$ pid=$(pgrep my-program|head -n 1) ; gdb -q \
  -ex "attach ${pid}" \
```

[3]Many MPI executors provide a way to set environment variables from the executor. Open MPI's `mpirun` includes an `-x` option, for example, so that one can quickly enable debugging by just saying `mpirun -x TJF_MPI_DEBUG=1 ./my-program`.

```
  -ex "set variable i=1" \
  -ex "finish"
...
Loaded symbols for /usr/lib/openmpi/lib/openmpi/
   mca_dpm_orte.so
0x0000000000400af2 in wait_for_debugger () at open.
   mpi.c:20
20          while(i == 0) { }
Run till exit from #0  0x0000000000400af2 in
   wait_for_debugger ()
main (argc=1, argv=0x7fff927d8998) at open.mpi.c:31
31         int rv =  MPI_File_open(MPI_COMM_WORLD, "
   atestfile", MPI_MODE_WRONLY,
(gdb) ...
```

In every MPI system I have used, MPI creates processes for the ranks in-order. This is convenient, as it means that rank 0 then has the lowest PID, except in pathological cases where the PID wraps around. Thus, `pgrep program | head -n 1` will always return the PID of rank 0. Note that the PID is also printed to standard error, so one could examine the program's output. That said, it is faster to maintain a terminal for such debugging, and use 'up-enter' to attach to the latest program instance.

If one throws `wait_for_debugger` directly after `MPI_Init`, as recommended earlier, than the program is essentially guaranteed to be stuck in that infinite loop when one attaches. `gdb` executes `-ex` commands in the order they are given, so we then change the loop variable to force the loop to exit, and `finish` to get us out of the `wait_for_debugger` function. Subsequently, we can debug as normal (often `continue` until the program segfaults).

1.3.1 A note on optimization

The `volatile` on the loop variable is critically important. Many people seem to be of the mistaken idea that `volatile` is for variables that one shares in threaded programs, but the actual use of `volatile` is to indicate to the compiler that the variable in question may change due to circumstances in the external environment. The initial use was in memory-mapping external devices: in this case, one would not want to cache the read of such a variable, because the external device will change the value at an unknown time.

This is, of course, exactly what is occurring here: the external environment (i.e. gdb) is changing the value of the variable, and so we don't want to cache the variable's value. Without the `volatile` qualifier, an optimizing compiler can (and often will) decide that the loop is infinite and therefore happily do things like 'forget' to update the frame pointer.

The author has found this works well and is quite portable in practice, and the author has AIX experience.

Chapter 2

Updates

The author *greatly* appreciate emails which correct even minor typos, improve upon this method, or just want to drop a note and say that this was useful for you. I tend to be pretty easy to contact; generally '`tfogal@my-current-affiliation`' is my email address, but you can also just google my name.