

When Good Code Goes Bad - Workshop on Debugging December 2006

Using GDB

Transcript

The next step we're going to go through is GDB. GDB is fine for sequential code. You compile your code with the dash G option; you start GDB giving it the name of the program you're going to execute. You get a prompt back. Usually what you want to do is to set a breakpoint, and then you can step through your code, change values in your code, do a variety of things.

The parallel version becomes something a little more dicey. You're actually not going to follow the same pattern on a sequential version of debugging with GDB; you're going to follow a slightly different road.

We're going to start of with everything being done sequentially. Go through the usual process of making the code. IFORT and GDB don't really quite get along very well, so you have to use IDB, which is Intel's version. And what were going to do is use IDB dash GDB. This allow you to use GDB commands.

The first thing I'm going to do is break right at the very first instruction. When the code starts executing, it will just automatically execute until it reaches that point, and then stop. Then I can control what it does next.

Okay, so we tell it to run, and in this case what happened was it ran. And it ran, and it ran, and it ran. And I finally hit a control-C. There's no way that it should have run for as long as it had. At that point it's kind of hard to know what to do, other than look at your initial assumptions. Did you put in an appropriate input value, and are you reading something reasonable? And we look at our input value, and we see that it's really not appropriate.

So we fix that. And now we're going to go ahead and start it up again. This is the first line that GDB encountered, line number fourteen. Start one equals one. Here you can list what's going on. The list command will just simply tell you five lines above and four lines below your current line. At this point, we've already passed the reads, so we've read the matrix in. So I just want to see, does the matrix make sense? I print the first value and the last value, and that looks right.

Right now we're at line fourteen. I can either step through this or I can use next. If I step through this, every time I encounter a function call, it's going to go into that function. We're going to go hit finish. What I'm doing here is just hitting return. It's going to redo the last command. SetupBC. If I type in step, I'm going to go into BC. The next takes us to call bc; click it again... call matrix.

What I'm doing here is just hitting return. It's going to redo the last command. Right now we're at line fourteen; if we step we're going to go hit finish, then hit setup bc. I can either step through this or I can use next. If I type in step, I'm going to go into BC. The next takes us to Call Matrix.

Now I'm going to jump into the matrix. And we get this set of input values to the function. We're going to do this loop starting with I being set to the start and finish indices of the loop. And print the start value.

The idea behind debugging is to collect what information you can, as much information as you can, and in as reasonable way as you can. And what's gone on here is that, I initialized the value; it doesn't complain about the one. But I'm passing in a zero. Zero has no meaning to it.

We go back, fix that error, recompile the code. So now what we're going to do is rerun the code, and we're going to run it to completion. And this time we decided to set a twelve, because that's where the actual first value is. And we're going to go through the same thing.

Now let's just verify that what we've got is okay. It looks just fine. So it looks like everything up to output is good. We end it, and now we want to see whether our results are correct. And they're not.

Knowing that things were fine up to output, and we're still not getting the correct result, we need to look at output. So let's step into this function, and here you're immediately told what's wrong. Or at least you're immediately told that there is a problem. You're told that there's a problem because you can see that the call is receiving two values; you're sending it three. By knowing what the code looks like, we know that max should not be ten; that should be nmax.

Okay, at this point we know that we're going to have to fix the code up, so here we see that you have the wrong parameters passed. When you look into output, you'll see that we're using max as nmax, we're allocating the wrong size array. We start marching through this array, and eventually ask for things in an inappropriate order.

So we compile the code. This time probably this will fix it. Not a guarantee, but it's a lot faster just to execute and to take a look. I don't want us to start GDB again. Now, if there were an error, back to GDB.

All right. Well, that's fine. We've got the sequential version working correctly. Now we want to jump into the parallel run.

The first thing you want to do when you have parallel code is you want to put in an infinite loop into the code immediately. The infinite loop will stop the code. So you're actually going to start the code in a normal fashion. You don't start it with GDB and then type in run. You start the code the way you normally start it. Then for every process you want to debug, open up a new window. And you're going to start GDB on each of those.

You have to figure out what the processes are. After you start GDB without putting in the function name--just GDB by itself--you then attach the process to GDB. And finally, you have to reset the value of your infinite loop. By the time you attach the processes, GDB will control the execution. Then you can release the code from its infinite loop.

So we're going to start with the same code we had with the second hack, only we add in an infinite loop. And you'll notice now I'm using two screens. The top screen is node zero; the bottom screen is

node one. I'm going to walk through this procedure several times, because it's the sort of thing you need to just see over and over and over again. It's very easy to mess this up.

So, clean everything up, make parallel, submit. Now, what I'm going to do is type in `bjobs` and it tells that I'm using "m570". That's where the code's actually running. So I log in to m570, and you want to find out where they're actually running, what the process IDs are. So you do a `P-S dash A-U-X`, and, rather than seeing a few hundred processes to take a look at, I `grep` for `matmul`, and I find my two process IDs.

At this point I start IDB, and I'm just going to start it without using the function name. But I'm going to start IDB on both processes. Now, I find it easier to do node one first and node zero second, especially if you're using GDB. GDB gives you a lot of information, and once you start getting things rolling, you'll lose this information, or at least it will scroll outside the view of your window if you start it on node zero. Since I always do PS on node zero, I always do the attachment on node one first, then do the attachment on node zero. That way I don't have to go scrolling through, finding the information.

What I'm going to do is attach PID-2722. It executes and it brings me to my DO-WHILE loop, my infinite loop. Then I'm going to attach, in node zero, 2721, and we're at the same point. I force Set equal to one. That breaks the infinite loop. And I do it here, and the infinite loop's broken.

What I'm going to do with this code in parallel is try to find out where my sends and receives are. And I'm going to put in break points that will cause GDB to stop processing at those points. I know my sequential version's okay; I know whatever I'm going to run here is going to be okay in terms of the computation. There may be other errors I haven't found yet. But I'm more likely to run into a communications problem.

Because this code is so small, I'm going to just start setting breakpoints at every one of the sends and receives. If you're dealing with thousands of lines of code, this isn't practical. By the time you're done setting all your breakpoints, your session will be up. Before I start doing anything on the code, I'm going to set the breakpoints on node one. So now when I start the code, it's just going to go merrily along until it runs into the first communications point on each node.

Now, it runs into this send and this receive. This is wrong. You have different types; you have different tag numbers. That doesn't work. So we quit. Back into the login node. So we add in this missing send and receive. We check to see that we've started running, and we check to see which node you're actually running on, and then you can go ahead and start running on that node. Now you get them both started.

At this point, we're going to do the exact same process, figure out where the sends and receives are. And this time you'll see that there are six breakpoints. So we start it running, and we get to `distribute_mat` and `receive_mat`. These look like the second sets of sends and receives. We want to look at these files, and once again we have something wrong. We see that `mat_size` is wrong.

We continue executing this thing, and finally hit a SEGV because we have a wrong value in there. We hit the SEGV, so we're going to use `backtrace`, and we get a whole lot of information. Now,

what BT does is it tells you where the code was in execution, and it tells you how it got to that point as a series of function calls. So each of these values are calls. These ones you're not going to worry too much about--starting and `libc_start`. `Main`, `matmul`, that's the starting point for the matrix multiplier. Then it called `send mat`, `distribute mat`, `mpi send`, and then the rest of this is just `elan` stuff. Nothing you're going to debug in that, so it's not worth taking a look at this, unless you really understand `elan`.

But that's certainly something that should leap out at you as an issue. We, again, know how to fix this from the previous discussion, so we go ahead and finish off. When I continued on... well, actually, eventually what happened was I killed it here, and then I killed it on node one. So we terminate; we get out of this altogether. Back to the login node.

You look through the code, you'll see that `mat_size` was set in the wrong place, so you put it in the right place. This was the case where it was put it in the read matrix, and that was clearly the wrong place to have it. So you just simply move it.

So we fix that, clean it up, run it, and get in the right place. Start the processes, start IDB, and do our attachment. And do the same thing on node one. Break the infinite loop, get the locations of the sends and receives. And in this case, it's pretty clear that the sends and receives are good. So it's already running. So click continue. Same thing here. Get all the breakpoints set. We hit continue. And we're just going to step through.

I got to this point, and what happened here was that it just hung. So I just hit a control-C this time and killed it. You can see why it hung. It failed again because of missing tags, and I know what's gone wrong here. Kill them both. Start again.

Getting our processes. Starting IDB. Go ahead, do the attachment, break, and continue. And so I get to this breakpoint and we finish.

What have we learned from this exercise with GDB? Well, one thing you need to do is get prepared. It's not as much preparation as you need for `print` and `flush` if you're using hand-coded debugging. But you still really want to be aware of what you're doing and where your code is.

GDB is wonderful for sequential code. It's okay for a couple processes. As you get into larger and larger numbers of processors, you will have a more and more difficult time figuring out what's going on. So, there's sort of a question of your ability to process versus the ability of GDB to function. GDB will do just fine. Your ability to deal with it is really going to be the limiting factor. On two processors, it's not that bad to work with. And it's nice in that you can set breakpoints at the sends and receives, and run it until it breaks and see whether it did what it was supposed to do. On the other hand, once you start getting four processes, eight processes, sixteen processes that you're looking at simultaneously, it starts to become more of a problem.

One good thing is it gives you control over the execution of your program. When you're using hand-coded debug statements, you could put in some interaction, depending on how you have it set up, but you really have to know what you're doing well in advance. GDB allows you to sort of fluidly change the starting and stopping points of the execution, which gives you a lot more control over

how you execute the code.

You don't really have to do much to the code. If you're going to do it in parallel, you do have to put in the infinite loop, and you have to remember to reset the infinite loop to get it going.