

## When Good Code Goes Bad - Workshop on Debugging December 2006

### Second Hack

#### Transcript

We looked at the first hack at debugging code using print statements. Now we want to look at how the code would function if we thought a little more about how to set it up. Basically what we want to do is have the code broken into bite-size chunks. Each chunk of the code should be responsible for doing one thing. So when I ask for something to do a matrix multiply, it shouldn't also readjust the matrix or determine how to distribute the matrix. It just should simply multiply two matrices together and give me a third matrix.

One of the things we also did in terms of decomposing the code is we separate out I/O from the actual computation. By having I/O and computation separated, having it properly decomposed, we can actually test each separate function of the code, verify that it's working properly, test the communication scheme is working properly, then recombine the code. We're not going to be guaranteed a success, we should be at least be a little more confident of success. We should also be much more confident in the quality of the code by doing this.

This is the sequential version of the code. This is the whole main program. I can look at this with almost with no effort, and tell exactly what's happening. The function's being named in a way that makes it pretty clear. I read a matrix; I set up B and C; I multiply the matrix; and then I output a matrix. We want to look at these pieces, in turn.

"READ\_MAT": Read the matrix. Here's my input file. You'll notice that I left all the errors in so we can treat them all over again. I can look at that mat size-- Knowing what matsize was supposed to be, I can tell that there's a problem there immediately. In knowing that Read\_MAT is something that will be done for all processes involved in this, I can see that since readmat is done by rank zero, and mat size is being done only within the context of reading the matrix, only rank zero will get it. Knowing what matsize is supposed to be, and having consistent names, I can see that this assignment isn't going to be correct.

We can look at SETUP\_BC. You can look at the code and you can see that you're looping through two indices. When the indices are equal, you're setting the value to two; otherwise, you're setting it to zero. C-Zero is set to zero.

We take a look at the matrix multiply. Well, we have this start and stop rank. Sequentially it doesn't make sense. Start and stop should be the beginning and ending. It's the carry-over for allowing the parallel version to be separable. But it's a small cost to pay.

We get to the output matrix. And here we see that we loop from MAX. Well, we shouldn't be looping on MAX, unless something's very odd. You look through the code and you see that you're looping on NMAX each time, and then all the sudden you're looping on MAX. It may be that we're

actually passing NMAX here, and that MAX is a perfectly legitimate number in this case. It may not be. All we have to do is go back to where it's being called, and you can see the signatures of the functions are different. That function's not going to work properly.

So, we've already debugged a big chunk of this code, just by simple inspection.

Now, you'll recall from the initial run of the first hack at this, we had a MPI Receive error. We shouldn't expect that here, because we're not using MPI. Right now this code is sequential; it's a serial code. That means that we're not going to have any communications issues to concern ourselves with. We can separate out the debugging of the computation from the debugging of the communications patterns. We can ensure that when we get on to debugging the communications that we're in fact debugging communications and not debugging a bunch of other things. We'll know what we're dealing with, because the code is separated in a way that allows us to deal with them separately.

So we go through the process of compiling the code and so on. All right, what we run into here is a segmentation violation. So what we want to do is go back and look at the matmul sequential. We want to start off with the highest level of the code, then start digging downward. And again, because it's structured this way, it's a very easy task.

We start off by printing statements before a call is made. Usually what you want to do is have something before and after each function call. We have actually bracketed the calls. Here's one function call. Before the next function call, we put in a new print statement. And we follow that pattern all the way through. We know when a call's being made, and we'll know whether or not we've returned from the call.

We ran for a very long time, and the code did eventually terminate. But we still have our segmentation violation. It died just before it called "BC Mul".

What we should have done, is try to include the state information. So just before it calls "BC mul", let's print out the values that are being passed to it. Now, I don't want to print out B and C just yet. But we can figure out these two scalar values. See what happens when we run the code.

There's our problem. Before, when we were doing it in a more brutish way, we didn't really have this information. And it's kind of hard to know what to print and where to print it. But because this code is already decomposed in a way, we know it died in this function. Let's start out by figuring out what we sent the function. And what we sent it was something that was guaranteed to kill it.

You can see that we linked the wrong file, so we remove it and link the appropriate file. We go ahead and run with the smaller matrix, and make, resubmit. Look at the error files, and what do we see? Well, there's our C matrix. That's not too bad. NMax and Max are pretty decent. When we look at the results--this is our output file--they're not really so good. The first value of the matrix should be a six, but the subsequent values aren't zero. We can look at the code, again, and start asking questions about the code.

We know that the C matrix was getting an appropriate value. It looked appropriate at this point. It went to Output, and something went wrong there. Something had to have gone wrong there, because

this is what's printing the matrix, and the matrix wasn't printed properly. Well, something's wrong with output; let's take a look at it.

You can see, by looking at it, that there's something wrong. So all we have to do is fix it by adding in the appropriate signature, and changing the limits to the appropriate values. So, when we run this, now we get what we wanted to see.

The decomposed code, with small programming units, makes this process simple. And it's just easier to handle code when it's presented in chunks that you can actually process.

What we're going to do next is jump onto the parallel version. So, we switch from making sequential to making parallel, and we submit the code.

Well, something's clearly wrong with the MPI Receive. So, I'm going to go about the same process of figuring out where the receives and sends are, or if there are other interprocess communication. Now we're going to go through bracketing the various communications. So, in MPI Distribute, we see here's a send, and what we're going to do is check to see, from the file and the line, that we've actually sent it, and give me some information about what was actually sent. In this case, NMAX is being sent, and K is being sent. So we bracket this one call with all this ahead of it and after it, all describing the first send. We do the second send.

The next thing we want to do is take a look at the receive. Well, we have one receive in there. We very clearly are sending NMAX and the matrix. We see the matrix being received, but we see nothing about NMAX. We don't have to stop and think about complex details related to the code, because it's decomposed in a way that makes it easy to see.

So in any case, though... When you're doing the MPI sends and receives, regardless of the condition of your code, have something in front of it and after it to ensure that the send is done and has completed.

So, we're going to go through the process all over again. We can see we go through a series of sends and then we hang on this last receive. Well, that can certainly be the problem, but when we look at ifort we see that we have a terrible mat size and we've hung on that first receive. Something's clearly wrong. So, we go ahead and update the nmax receive.

This function was telling us start and finish value. Just looking at the code, we can see that nmax is required. We need to put a receive ahead of that function call. This function call requires nmax; this receive is giving us nmax. That way we don't induce an error.

We go through this process. You can see that we hit the receive, we hang on the second receive, and this value makes no sense. So we've got to go about fixing that. We take our mat size out of the read, and we put it into matmul. Go ahead and rerun the code, and we have a receive on the right, but now we have a send hanging. Well, let's look at the send and receive pair. And they're both being done in this file.

You'll notice that I've printed out values that are related to variables. I should have included this tag. So we go about fixing that, and there's my one hundred lines of output.

So, which one's easier? It's one of those things that's the choice is definitely the user's. Decide what you want to do, but you will save yourself grief by simply decomposing your code into appropriate units and working from there. Now, of course, we've seen five thousand line functions come through and that's not feasible to work with. It's very difficult to debug, it's very difficult to upgrade, it's very difficult to fix. And it's both time and money consuming to take care of these problems. But in the long run, if you've got the wherewithall to do it, it will save you more time, more on excessive headaches and the like to just simply take care of the code and put it in an appropriate condition.

Decomposing the code isn't just a matter of breaking the code into small chunks. You really want to be thoughtful about how you break up your code and, in particular, where you're placing function calls. Don't create a huge DO-Loop, or a nested DO-loop, and then put a function call in the middle of the DO-loop, for example. That's the sort of thing you want to do... you want to have the function contain the DO-loops. It is something that you've got to be judicious about, thoughtful about, but small programming units are easier to deal with. You can look at the code and understand it.

By having the communications infrastructure separated from the computation, you can actually look at your computation and ensure that it's correct--ensure that it's proper--so that when you go about debugging the more complex communications problems, you know that you're going to be dealing with communications. You know that you're not going to be dealing with computation. So you don't get confounding effects.

Another reason to use small programming units is, it allows you to unit-test your code. By cutting and pasting an eigenvalue computation, you have to ensure that not only you've cut and pasted it properly in every location, but that every time you update it or change it somewhat, you have to make sure that those changes propagate.

You can't really test that chunk of code in isolation when you don't know how it operates or what it's doing as a separate issue from the rest of the code. If you have it in a small chunk, you can isolate that piece of code, do whatever sort of testing you want to do and ensure that that piece of code works correctly.

You can also do this with the communications. There are several examples where we've had a missing receive and we had an incorrect tag. By having the code decomposed appropriately and having the communications separated out, it's really much easier to figure out what has gone wrong. So by breaking the code up, you can do unit testing and verification of the separate pieces of the code before you put it together, and you can be certain that each piece is doing its job appropriately because you can test those pieces in isolation.