

# Virtual Summer School: Python and parallel debugging

Erik Spence

SciNet HPC Consortium

1 September 2021

# Material for this class

The slides and code for this class can be found here:

<https://scinet.courses/584>

All the material for the 2021 Virtual Summer School can be found here:

<https://scinet.courses/573>

Make sure that you confirm your attendance before the end of class.

# Details about this course

To get credit for this course:

- You need to attend 2 out of 3 sessions.
  - ▶ Each session is about 1.5 hours,
  - ▶ Sessions are 12:30 - 2:00pm, Monday, Wednesday, Friday, August 30 - September 3.
  - ▶ To demonstrate attendance, you must take the attendance test each class.
- You must submit the assignment.

Ask questions!

# Today's class

The purpose of this second set of material is to introduce you to Python debugging and the basics of debugging parallel code. We will cover the following topics:

- the Python debugger, PDB,
- debugging R code,
- DDT,
- Setting up DDT's client-server mode,
- (Parallel) Debugging with DDT,
- Examples.

Ask questions!

# Getting set up on SciNet

Be sure to use your username, not ejspence, when you attempt to log into SciNet.

```
ejspence@mycomp ~>
ejspence@mycomp ~> ssh -Y ejspence@teach.scinet.utoronto.ca
ejspence@teach01 ~>
ejspence@teach01 ~> cd $SCRATCH
ejspence@teach01 scinet/ejspence>
ejspence@teach01 scinet/ejspence> cp -r /scinet/course/ss2021/8_debug .
ejspence@teach01 scinet/ejspence>
ejspence@teach01 scinet/ejspence> cd 8_debug
ejspence@teach01 ejspence/8_debug>
```

These steps are only necessary if you want to follow along with the examples, or if you didn't do this for the first class.

# Debugging Python code

If you need to debug Python code you CAN use GDB. GDB has python integration which can be useful. However, Python also comes with its own debugger, PDB (Python DeBugger) built right in. I find this to be a simpler debugging interface for Python code.

- It has most of the commands that regular GDB has, though with a few modifications.
- There are two ways to invoke it:
  - ▶ from the command line,
  - ▶ embedding it directly into the code.
- It behaves very similarly to GDB,
- but also allows you to run Python commands within the debugger, to assist with debugging.

We will go over a Python version of our bugexample code to demonstrate its functionality.

# Invoking PDB

The simplest way to invoke PDB is to use through the bash shell prompt:

```
ejspence@teach01 ejspence/8_debug> module load python
-----
ejspence@teach01 ejspence/8_debug>
-----
ejspence@teach01 ejspence/8_debug> cd bugexample/python
-----
ejspence@teach01 bugexample/python> python bugexample.py 6 7
6
7
Sum of integers is 13
-----
ejspence@teach01 bugexample/python>
-----
ejspence@teach01 bugexample/python> python -m pdb bugexample.py
/scratch/s/scinet/ejspence/8_debug/bugexample/python/bugexample.py(2)<module>()
-> import argparse
-----
(Pdb)
```

This will give you access to the interactive PDB prompt, and indicate what line you're on, in which file.

# Invoking PDB, continued

The previous approach will only work if you're running a script. If you're working interactively and you need to use PDB, add the following lines to your function:

```
def my_func(arg1, arg2):  
    import pdb           # Add this line  
    pdb.set_trace()     # and this line  
                        # to start PDB when the code is run.  
  
    :
```

When the function is now called the debugger will be launched.



# Python debugging final notes

There are a few problems with PDB:

- You can set the values of variables, but those variables CANNOT be a PDB command (setting the variable 's' is not possible in this code, since 's' is a PDB command).
- Several important commands, such as frame, are not available.
- The list command stops working after the use of other commands, such as 'where'.
- There appear to be problems with using PDB with functions defined at the interactive Python prompt, rather than within a file.
- These problems disappear if you use iPython, but you can't exit PDB cleanly, whether the function is defined interactively or in a file.

Nonetheless, if you're working with code stored in files, PDB is a useful tool.

Also note that there are other Python debuggers out there, in particular pyCharm is popular.



# PDB command summary

<b>help</b>	<b>h</b>	print description of command
<b>run</b>		run from the start (+args)
	<b>r</b>	run until the current function returns
<b>where</b>	<b>w</b>	print a stack trace
<b>list</b>	<b>l</b>	list code lines
<b>break</b>	<b>b</b>	set breakpoint
<b>down</b>	<b>d</b>	move one level down in the stack trace
<b>continue</b>	<b>c</b>	continue
<b>step</b>	<b>s</b>	step into function
<b>next</b>	<b>n</b>	continue until next line
	<b>p</b>	print variable
<b>down</b>	<b>do</b>	go to called function
<b>until</b>	<b>unt</b>	continue until line/function
<b>up</b>	<b>up</b>	go to caller
<b>quit</b>	<b>q</b>	quit pdb

# Debugging R code

The debugging capabilities of R are more limited than Python, but there are a few functions which can be helpful:

- `traceback()`: this function will print out the trace, which can be helpful if your code has failed at a specific location.
- `browser()`: launches an interactive environment inside the function where it is called. This environment also contains the 'next', 'step', 'finish', and 'continue' commands we know from `gdb`.
- If you are using Rstudio, you can add breakpoints to your code.
- Calling `'options(error = recover)'` before the failing function is called. If the failing function is then called, a different interactive prompt is launched, which displays the traceback.

There are many tutorials on debugging R code on the web. We won't be going into detail in this class.

# Parallel debugging, shared memory

You can use GDB for shared-memory debugging. You can

- track each thread's execution and variables,
- perform OpenMP serialization: `omp_set_num_threads(1)`,
- step into an OpenMP block: `break` at the first line!
- create a thread-specific breakpoint: `b <line> thread <n>`

Within Valgrind, you can use the "helgrind" tool for finding race conditions.

```
ejspence@teach01 ~> module load valgrind
-----
ejspence@teach01 ~> valgrind --tool=helgrind <exe> &> out
-----
ejspence@teach01 ~> grep <source> out
```

where `<source>` is the name of the source file where you suspect race conditions (valgrind reports a lot more).

# DDT

If you run an MPI code then you need a parallel debugger which can handle MPI. While GDB can debug threads, if you're using MPI you should use DDT ("Distributed Debugging Tool").

- Powerful GUI-based commercial debugger by *Arm* (one of the few pieces of software that SciNet purchases).
- Supports C, C++ and Fortran.
- Supports MPI, OpenMP, threads, CUDA and more.
- Available on all SciNet clusters (Niagara, Mist, teach).
- Of the General Purpose Compute Canada clusters, only available on Graham.
- Part of the "Arm Forge" suite, which also includes a 'profiler' called *MAP*.

Unlike GDB, DDT must be run through the GUI interface. This means it can be slow to run directly if your internet connection is slow. Better to use the client-server mode.

# DDT client-server mode

Fortunately, DDT supports a client-server mode, which allows you to run a DDT server on a SciNet cluster, but only send a minimal amount of graphical information over the internet to your local machine.

To download the DDT client go to this link:

```
https://developer.arm.com/tools-and-software/server-and-hpc/downloads/arm-forge/older-versions-of-remote-client-for-arm-forge
```

Be sure to download version 20.1.3. We need to use this version because that is the version installed on SciNet's clusters. Install the correct version for your computer.

As a note, **we can only all run this right now, during class, if there are few enough students doing so**. SciNet only possesses DDT licenses for 64 cores. Thus there are only enough licenses for 15 students to participate if we each take 4 licenses.

# Using 2FA with DDT

As an aside, if you use Two Factor Authentication (2FA) on Niagara or Mist you will need to disable it to use DDT in client-server mode.

There are several ways this can be accomplished.

- The easiest way is to move the seed file to a new name. The seed file is located at `/scinet/authenticator/$USER`.
- You can't read the `/scinet/authenticator` directory. Make sure that you don't forget what name you rename your directory.
- Be sure to put your directory back to your user name when you are finished.
- This only works because 2FA is optional on Niagara. When it becomes mandatory a different technique will be needed.

```
ejspence@teach01 ~> mv /scinet/authenticator/ejspence /scinet/authenticator/ejspence-old
```

# Client-server DDT, the server

How to run DDT in client-server mode?

- Log into the teach cluster.
- Start a debug session, giving you control over a set of teach-cluster compute nodes or cores (here I request only 4 cores).
- Note the node on which you are running (teach02, for example).
- Note also the location of the DDT setup script. This script loads the needed modules.

Again, we can all do this now if there are no more than 15 students participating today.

```
ejspence@teach01 ~>
-----
ejspence@teach01 ~> debugjob -n 4
SALLOC: Granted job allocation 89175
-----
ejspence@teach02 ~>
-----
ejspence@teach02 ~> cd $SCRATCH/8_debug
-----
ejspence@teach02 ejspence/8_debug>
-----
ejspence@teach02 ejspence/8_debug> ls
bugexample ddt_remote_setup.sh mpiexample
setup
-----
ejspence@teach02 ejspence/8_debug>
-----
ejspence@teach02 ejspence/8_debug> pwd
/scratch/s/scinet/ejspence/8_debug
-----
ejspence@teach02 ejspence/8_debug>
```

The path to your setup script will, of course, be different.



# Client-server DDT, the server, continued

A few more steps on the server side:

- Source the setup script, to load the modules.
- This is only necessary to find DDT's installation directory.
- Find DDT's installation directory. We will need this when we set up the client.

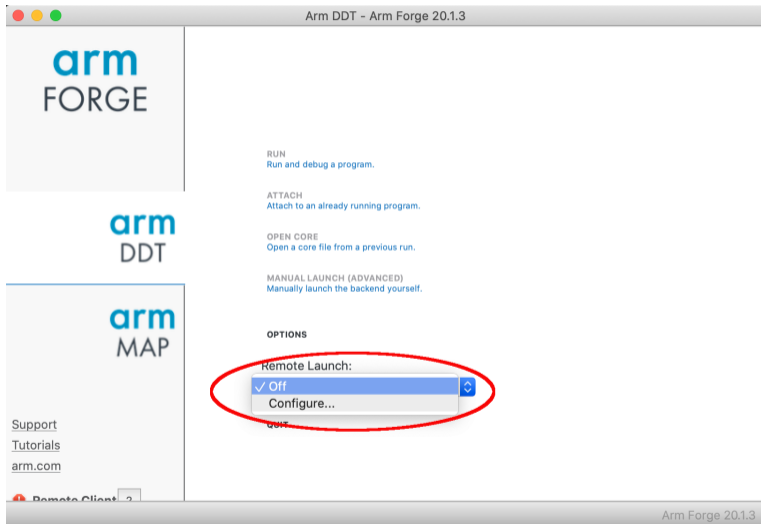
```
ejspence@teach02 ejspence/8_debug>  
-----  
ejspence@teach02 ejspence/8_debug> source setup  
-----  
ejspence@teach02 ejspence/8_debug>  
-----  
ejspence@teach02 ejspence/8_debug> echo $SCINET_DDT_ROOT  
/scinet/teach/software/2018a/opt/base/ddt/20.1.3  
-----  
ejspence@teach02 ejspence/8_debug>
```

Now we're done on the server side. The client itself is capable of launching the server.

# Client-server DDT, the client

Now we need to build our remote connection. On your local machine:

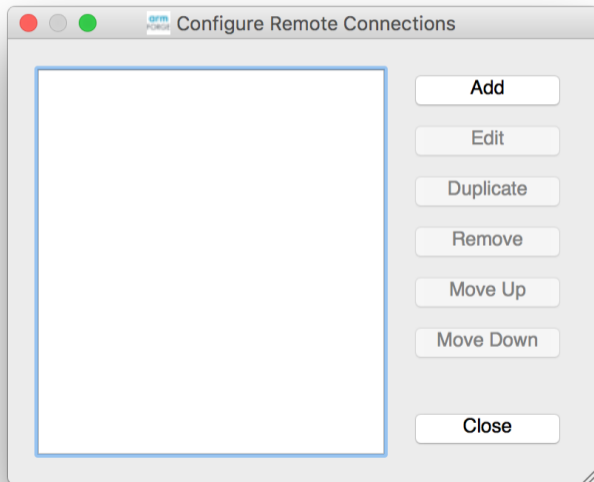
- Launch Arm-forge.
- Select "Remote Launch", "Configure..."



# Client-server DDT, the client, continued

We need to create a DDT connection session.

At this prompt, select "Add".



# Client-server DDT, the client, continued more

Give the connection a name, and fill in the settings:

- Host Name: your connection to the teach login node, and the node running the debug job.
- Remote Install Dir: the path on the teach cluster to the DDT installation.

Remote Launch Settings

Connection Name: DDT Test

Host Name: ejspence@teach.scinet.utoronto.ca ejspence@teach02

[How do I connect via a gateway \(multi-hop\)?](#)

Remote Installation Directory: /scinet/teach/software/2018a/opt/base/ddt/20.1.3

Remote Script: /scratch/s/scinet/ejspence/8\_debug/ddt\_remote\_setup.sh

Always look for source files locally

KeepAlive Packets:  Enable

Interval: 30 seconds

Proxy through login node

Test Remote Launch

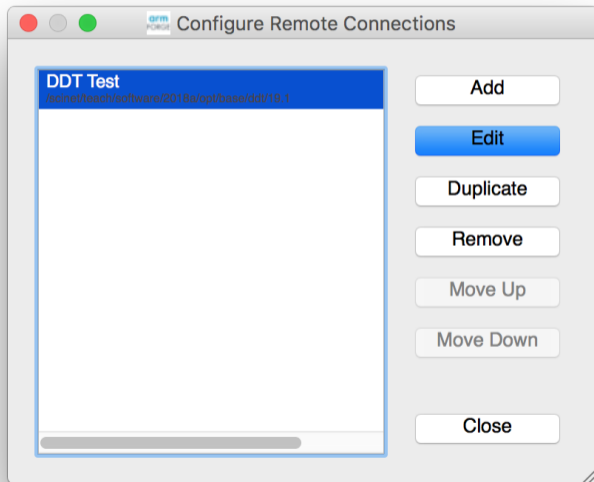
Help OK Cancel

- Remote Script: the path to the setup script.

# Client-server DDT, the client, continued even more

Now that our session has been built, we're ready to launch it.

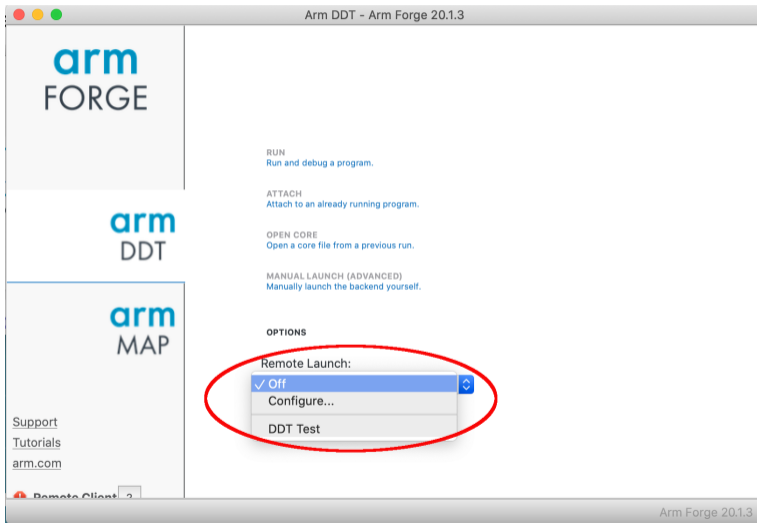
At this prompt, select "Close".



# Client-server DDT, the client, continued even morer

Now that the session ("DDT Test") is available, select it.

Once the connection is established, click on "RUN".



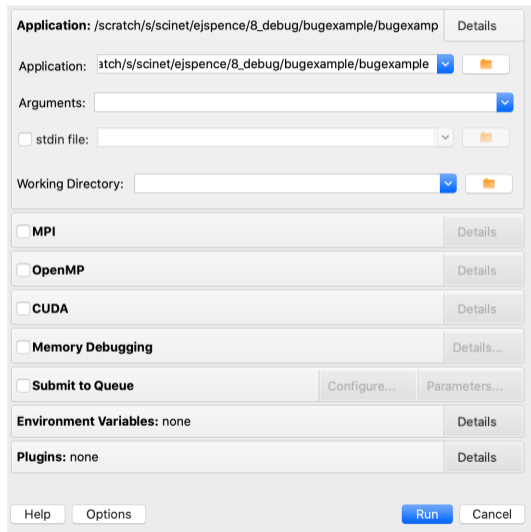
# Running DDT

Now click on "Run", which brings us to the window on the right.

From here you can specify all the details of the run:

- executable (note that this must be on the server side),
- arguments,
- number of MPI processes,
- number of OpenMP threads,
- and many other things.

When all the details have been specified, click on "Run".



The screenshot shows the DDT application configuration window. The 'Application' field is set to `/scratch/s/scinet/ejspence/8_debug/bugexample/bugexamp`. The 'Arguments' field is empty. The 'Working Directory' field is also empty. Below these fields are several checkboxes for enabling features: MPI, OpenMP, CUDA, and Memory Debugging. There are also buttons for 'Submit to Queue', 'Environment Variables', and 'Plugins'. At the bottom right, there are 'Run' and 'Cancel' buttons.

<b>Application:</b> /scratch/s/scinet/ejspence/8_debug/bugexample/bugexamp	Details
Application: /scratch/s/scinet/ejspence/8_debug/bugexample/bugexample	
Arguments:	
<input type="checkbox"/> stdin file:	
Working Directory:	
<input type="checkbox"/> MPI	Details
<input type="checkbox"/> OpenMP	Details
<input type="checkbox"/> CUDA	Details
<input type="checkbox"/> Memory Debugging	Details...
<input type="checkbox"/> Submit to Queue	Configure... Parameters...
<b>Environment Variables:</b> none	Details
<b>Plugins:</b> none	Details
Help Options	Run Cancel

# Our previous example

Let us examine our previous example using DDT. If you haven't already compiled this code, please do so.

- The example code reads integers from the command line and sums them.
- The code is written in C.

When doing the below commands, make sure you've logged into the teach cluster, and have copied the class code into your SCRATCH directory (see the previous slide).

```
ejspence@teach02 ejspence/8_debug>  
-----  
ejspence@teach02 ejspence/8_debug> source setup  
-----  
ejspence@teach02 ejspence/8_debug>  
-----  
ejspence@teach02 ejspence/8_debug> cd bugexample/c  
-----  
ejspence@teach02 bugexample/c>  
-----  
ejspence@teach02 bugexample/c> make bugexample  
-----  
ejspence@teach02 bugexample/c>
```



# Other features of DDT

Of course, DDT comes packed with all manner of features.

- Some of the user-modified parameters and windows are saved by right-clicking and selecting a save option in the corresponding window (Groups; Evaluations)
- DDT can load and save sessions.
- *Find* and *Find in Files* in the Search menu.
- *Goto line* in Search menu (or Ctrl-G)
- Synchronize processes in group: Right-click, “Run to here”.
- View multiple source codes simultaneously: Right-click, “Split”
- Right-click power!

If there's a feature you're after it's likely already there.

# Other features of DDT, continued

DDT also supports several memory-debugging features.

- Select "memory debug" in the 'Run' window,
- DDT will stop on any error (before crash or corruption),
- You can then check the pointer (right click in evaluate),
- You can also view overall memory stats,

This allows you to do both regular code and memory debugging, in parallel, simultaneously.

# DDT, MPI debugging

With an MPI code you will have multiple simultaneously-running processes.

- Your code is running on different cores!
- Where should you run the debugger?
- Where do you send the debugger output?
- How do you debug the inter-process communication?
- Much is going on at same time.

A good approach:

- Write your code so it can run in serial: perfect that first.
- Deal with communication, synchronization and deadlock on *smaller* number of MPI processes/threads.
- Only then try full size.

# DDT MPI example

Let's do an MPI-debugging example using DDT.

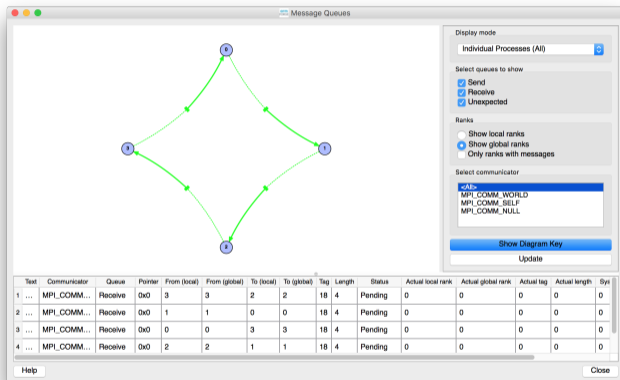
```
ejspence@teach02 bugexample/c>  
-----  
ejspence@teach02 bugexample/c> cd ../../mpiexample  
-----  
ejspence@teach02 8_debug/mpiexample>  
-----  
ejspence@teach02 8_debug/mpiexample> make  
-----  
ejspence@teach02 8_debug/mpiexample>
```

Once the code is compiled we can run it through our local DDT client.

# Detecting deadlock with DDT

If there are MPI messages which aren't being delivered, you can see them in the Message Queue:

- Tools → Message Queues.
- This produces both a graphical view and a table of active communications.
- This helps to find deadlocks and other communication problems.



# Useful references

Some debugging resources:

- *PDB*: <https://docs.python.org/3/library/pdb.html>
- N Matloff and PJ Salzman  
*The Art of Debugging with GDB, DDD and Eclipse*
- *GDB*: <http://sources.redhat.com/gdb>
- *DDT*: <http://www.allinea.com/knowledge-center/tutorials>
- *SciNet Wiki*: [https://docs.scinet.utoronto.ca/index.php/Performance\\_And\\_Debugging\\_Tools:\\_Niagara](https://docs.scinet.utoronto.ca/index.php/Performance_And_Debugging_Tools:_Niagara)