Debugging and Profiling

Nate Woody

Debugging

- **Debugging** is a methodical process of finding and reducing the number of bugs, or defects, in a computer program or a piece of electronic hardware thus making it behave as expected. Debugging tends to be harder when various subsystems are tightly coupled, as changes in one may cause bugs to emerge in another.
- A **debugger** is a computer program that is used to test and debug other programs.
- This can be hard enough with a single local process and but get's many times more complicated with many remote processes executing asynchronously. This is why **Parallel Debuggers** exist.

Debugging Requirements

- In general, while debugging you may need to:
  - Step through code
  - Set/Run to breakpoints
  - Examine variable values at different points during execution
  - Examine the memory profile/usage
  - Provide source-level information after a crash
- For MPI and OpenMP Code we have additional requirements
  - All of the above for remote processes
  - Examine MPI message status
  - Step individual processes independent of the rest

Profiling

- Software performance analysis
  - Profiling is examining where a given code is spending it's time so that you can understand the performance characteristics of a program or set of functions.
  - There are several levels of profiling, but we will talking about function level profiling which provides information on the frequency and duration of function calls.
  - A profile is a statistical summary of function calls, generally you get the number of times each function was called and the total amount of time spend in the function.
  - The goal of profiling is to identify "hot spots", which are functions that occupy an inordinate amount of the total time of a program, which means that optimization of these functions will provide the greatest benefit.

Profiling

- Flat profile – total time and number of calls of function

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name	
33.34	0.02		0.02	7208	0.00	0.00	open
16.67	0.03		0.01	244	0.04	0.12	offtime
16.67	0.04		0.01	8	1.25	1.25	memccpy
16.67	0.05		0.01	7	1.43	1.43	write

- Call graph – See how a function was called

index	% time	self	children	called	name	
[1]	100.	0 0.00	0.05	1/1	start [1]	
		0.00	0.05	1/1		main [2]
		0.00	0.00	1/2		on_exit [28]
		0.00	0.00	1/1		exit [59]

		0.00	0.05	1/1		start [1]
[2]	100.0	0.00	0.05	1	main [2]	
		0.00	0.05	1/1		report [3]

Tools

- Debugging requires a debugger, of which many are available.
  - Your development environment may well have a built-in debugger available. Eclipse is a good a good example, which provides a nice interface to a debugger.
- GDB – The GNU Project Debugger
  - Universally available debugger that can debug C, C++, and Fortran code (if you can compile it with GCC, you should be able to debug it with gdb).
  - GDB has a command line interface to walking through code that takes a little getting used to.
  - Your code must be compiled in debug mode before you can use GDB, you can't just start debugging a binary.

Debugging with GDB

- Step 1 – build with debugging symbols.
  - $ g++ -ggdb –Wall –o test main.cc
- Step 2 – launch the application inside the debugger
  - $ gdb test
- Step 3 – Run the application
  - $ (gdb) run
- Step 4 – Examine the backtrace
  - $(gdb) backtrace
- Step 5 – Examine the parameter values
  - $ (gdb) x 0x7fffa408c3d4

Breakpoints and stepping

- Previously, we just used the debugger to examine what happened after the program exploded. It may be more useful to examine the program before it blows up, which can be done by setting breakpoints and stepping.
- A breakpoint halts execution of the program at a specific source line.
  - (gdb) break LinkedList<int>::remove
- This can be made conditional by using the “condition” statement, so that the breakpoint only occurs when a specific condition is meant.
  - (gdb) condition 1 item_to_remove==1
- Re start using run and execution will be halted at the breakpoint. Execute one line of code by using step.
  - (gdb) step

GDB commands summary

- run – execute the program from beginning.
- backtrace – produce the backtrace from the last fault
- break <line number> or break <function-name> - break at the line number or at the use of the funciton
- delete <breakpoint number> - remove a breakpoint
- step – step to next line of code (step into function if possible)
- next – step to next line of code (do not step into function)
- list – print source list (list <function> to print a specific function)
- print <variable name> - print the value stored by the variable
- continue – run until next break point
- quit – quit
- help – get help on any command

Gprof for profiling

- Gprof is used for monitoring the performance of a FUNCTIONAL program to help guide optimization efforts.
  - It's not a debugger, make sure you're program is working the way you want before you think about profiling.
  - Optimization often results in less readable, modular, and maintainable code, the best optimization strategy may be to not optimize.
- In order to get profiling output, compile with the –pg option.
  - Generally, you'll want to use all the other compile flags that you are using, otherwise you may be profiling code that performs differently than it does. However, in *most* cases, this is not a huge issue. Try it both ways if you are concerned.

A Gprof session

- Compile with profiling on
  - $ gcc –pg –o prof_test prof_test.c
- Run the binary normally, which will generate a gmon.out file
  - $ ./prof_test
- Run gprof on the binary to generate the results
  - $ gprof prof_test >> profile_results.txt
- Examine the results
  - vi profile_results.txt

Profiling Caveats

- Basically, you're looking for functions that occupy a large amount of system time and/or are called a inordinate amount of times.
  - Functions that takes lots of time are candidates for optimization, particularly if they are called heavily. This will give you the best bang for the buck.
  - Functions that are called many times but don't occupy much system time are probably losers for optimization. You won't see much benefit from optimizing these even if you do!
- You should be careful about I/O!
  - I/O wait is not reported in profiling numbers, so examine timing information in I/O heavy functions carefully.
- Be cautious in interpretation of absolute time
- Don't shortchange the sample data when generating profile data

Results

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.34      0.02     0.02     7208     0.00     0.00  open
 16.67      0.03     0.01      244     0.04     0.12  offtime
 16.67      0.04     0.01        8     1.25     1.25  memccpy
 16.67      0.05     0.01        7     1.43     1.43  write
 16.67      0.06     0.01                             mcount
  0.00      0.06     0.00      236     0.00     0.00  tzset
  0.00      0.06     0.00      192     0.00     0.00  tolower
  0.00      0.06     0.00       47     0.00     0.00  strlen
  0.00      0.06     0.00       45     0.00     0.00  strchr
  0.00      0.06     0.00        1     0.00    50.00  main
  0.00      0.06     0.00        1     0.00     0.00  memcpy
  0.00      0.06     0.00        1     0.00    10.11  print
  0.00      0.06     0.00        1     0.00     0.00  profil
  0.00      0.06     0.00        1     0.00    50.00  report
...
```

- Only 5 functions take up any appreciative time in this report, so we would only start with these. A ton of open calls are made (30x more than anything else, I wonder why). 8 memcopy's take up 16% of the total execution time.

GDB/Gprof Lab

- The goal of this lab is to make sure everyone can successfully use and understand GDB and Gprof. Three source files are provided for practice and a Makefile is provided for help (try not to use it).
  - ~train200/profile_debug.tar.gz
- **Example1.c** – a simple profiling example. Use gcc to build a binary with profiling enabled (a Makefile is provided if you have trouble). Run gprof and examine the flat and call graph. Try and figure out the structure of the program from the call graph and verify by looking at the source. Alter the program to work without the increment function and verify the results using the call graph.
- **Linpack.c** – compile using the Makefile (make profile) to get an example2 binary. Run gprof on the resultant binary, what function is the target for optimization?
- **Example3.c** – a buggy mangled printer. Use the debugger to identify where the problem is and fix it.
- **Example4.c** – a buggy echo machine. This is actually a little tougher than you might expect as you get buried in c library functions and need to work your way back out to step.

Instrumenting code for logging and Debugging

Debugging and Logging

- GDB is a debugger that you would use when you have identified a problem in your code and you're trying to isolate and identify the source of the problem.
- Printf() debugging is the debugging style where you add all sorts of printf(), cout, print, System.out.println(), etc to dump information to stdout or stderr to track what the problem is.
  - Learned folks often disapprove of such nonsense and suggest that practical use of a debugger is vastly more efficient.
  - Practical folks will admit to it's utility and point to the fact that it allows continuous monitoring of the code outside of a debugger
  - Both are right, and with some simple setup, you can add debugging/logging statements to your code that will be useful, informative, and unintrisive.

Printf Debugging

- First, let's take a look at what the much feared printf debugging looks like.

```
int main (int argc, char** argv) {
        printf("Starting main…");
        int iterations = 5;
        int val = 0, val2=0;
        printf("Initialized val to %d and val2 to %d", val, val2);
        while (iterations --) {
                val = sometime();
                print("Sometime() returned %d\n", val);
                val2 = moretime();
                printf("moretime() returned %d\n", val);
        }
        printf("Exiting main, iterations ==%s\d", iterations);
```

Printf Debugging

- With this example, we have pretty much covered the code with 5 printf statements. This results in several problems that can occur.
  - You have drastically increased the number of lines of code, and it's quite easy to make an error in one of these new lines (mess up a format string and you're debugging breaks your program). The number of lines required to get debugging level information is very high.
  - There is no easy way to remove these lines from your code without potentially breaking something. If you insert these lines in the middle of a debugging session, if they aren't manually removed this function will forever emit all of this stuff on stdout. I hope no meaningful data goes to stdout anywhere.
  - Writing to stdout slows down your program significantly, having a printf in the middle of a tight for loop will have a big impact on performance.

Advantages of Printf Debugging

- There are some cases where printf-style debugging is useful.
  - Long running applications where erronous results are produced. Using a debugger is most useful when the identifying crashes or once the function/class/etc that has a bug is identified. Printf may help you identify the function or class where deviations occur.
  - It allows you to examine optimized code instead of code with debugging symbols added. It also let's you get output while running at full scale for parallel applications. This is occasionally useful.
  - Running multi-threaded or on remote machines. Connecting a debugger to a remote process can be difficult and tracking forks etc is non-trivial.
  - Help identify transient and/or timing related bugs.

What to do

- Printf debugging certainly has advantages, but it also creates ugliness in your code as well potentially the source of problems unrelated to the one you're trying to solve!
- These can be mitigated with a few easy steps
  - Don't ever use stdout, use stderr (unbuffered, seperation, etc)
  - Don't call printf directly, use a macro/function/class that handles the output safely.
  - Use "levels", which are the criticality of the problem and range from debug (the lines we showed earlier) to warning (possible erronous values). You can then control when and where these various levels are printed.

Logging Libraries

- What we're actually talking about is "logging".
  - Logging is the process of computer systems logging state changes and informational content to a central location where they can be recorded and examined later.
  - Here's a chunk of an up2date log (stashed in /var/log/up2date)

```
[Mon May 18 09:53:49 2009] up2date logging into up2date server
[Mon May 18 09:53:50 2009] up2date succesfully retrieved authentication token
[Mon May 18 09:55:20 2009] up2date Updating pacakge profile
[Mon May 18 09:57:25 2009] up2date Updating package profile
```

- There are libraries that we can use to get safe, readable logging added to your code very easily.

Log4Blah

- Log4J is an Apache foundation project that provides logging utility for Java. The interface to this has now been copied to many different languages.
  - Log4Net – is for .NET and works with C++, C#, etc.
  - Log4CXX – is for C++ and works for most platforms.
  - Log4c – is for C
  - Log4py and log4p– is for python
  - Log4Ruby – you get the idea, yes?
- Other logging libraries exist, Log4J is the only one that crosses so many different languages, which makes it a little easier to use.
- As far as I know if you're using fortran, you'll need to implement the logging yourself (if someone knows differently, please let me know).

Log4J Features

- Automatic formatting of output with appending timestamps and who emitted the log.
- A library of “Appenders” which are objects that control *how* and *where* a log line is written.
  - RollingLogAppender – logs to a file which rolls when it reaches a certain size or date.
  - SocketAppender – logs over a socket to a log server
  - DatabaseAppenders – log information to a database
- It is a best-effort fail-stop system.
  - This means that it will not emit unexpected expectations causing your application to crash but will try really hard to actually log your info.
- It provides easy control of logging level at runtime

What does it look like

- Replacing printf lines with log lines doesn't significantly change the look of the program, some extra boilerplate and a logger object must be grabbed.

```
int main (int argc, char** argv) {
        log4c_init();
        mycat = log4c_category_get("sillyapp.main");
        int iterations = 5;
         log4c_category_log(mycat, LOG4C_PRIORITY_DEBUG,"Debugging app 1
- loop %d", iterations);
        int val = 0, val2=0;
        log4c_category_log(mycat, LOG4C_PRIORITY_ERROR, "Some error"
        printf("Initialized val to %d and val2 to %d", val, val2);
        …
```

Configuration

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="log4net" type="log4net.Config.Log4NetConfigurationSectionHandler,log4net" />
  </configSections>
  <log4net>
    <!--This is a rolling log file.  When the file exceeds 100kb, it's rool to a new file, keeping at most 10 files.
    <appender name="RollingLogFileAppender" type="log4net.Appender.RollingFileAppender">
      <file value="c:\NateIsRolling.txt" />
      <appendToFile value="true" />
      <maxSizeRollBackups value="10" />
      <maximumFileSize value="100" />
      <rollingStyle value="Size" />
      <staticLogFileName value="true" />
      <layout type="log4net.Layout.PatternLayout">
        <header value="[Header]&#13;&#10;" />
        <footer value="[Footer]&#13;&#10;" />
        <conversionPattern value="%date [%thread] %-5level %logger [%ndc] - %message%newline" />
      </layout>
    </appender>
    <!--This is where we specify what loggers to use and at what level they should log.-->
    <!--This says that anything DEBUG should be logged-->
    <root>
      <level value="DEBUG" />
      <appender-ref ref="RollingLogFileAppender" />
    </root>
  </log4net>
</configuration>
```

Results

- Log messages are then shunted to the appropriate location and formatted prior to putting them in the log.
- We can fancy things up and add headers and footers, as well as all sorts of other fanciness (log different levels to different files/appenders).

```
[Header]
2009-05-13 15:21:14,315 [11] WARN  Logger.Program Pretty sure I'm getting ready to die!
2009-05-13 15:21:14,331 [11] ERROR Logger.Program uh-oh, no I wasn't!
2009-05-13 15:21:14,331 [11] FATAL Logger.Program blech.  Out
[Footer]
```

- There are many programs out there designed for “log file analysis”, aka handling large nicely formatted log files.

Conclusion

- Ad hoc printf debugging probably causes as many problems as it solves
- Nonetheless, it can be highly useful in some cases.
- A few easy steps can make this style of debugging much less problematic and the early inclusion of a logging library will save you a lot of time down the line.
- The log4J line of loggers are a nice suite of tools that serve many different languages with a common interface and actions.

DDT
Distributed Debugging Tool

Parallel Debugging on Ranger

DDT

- **DDT – D**istributed **D**ebugging **T**ool (www.allinea.com)
- A graphical debugger for scalar, multi-threaded and parallel applications for C, C++ and Fortran
- DDT's provides graphical process grouping functionality. DDT makes it really easy to assign arbitrary processes into groups which can be acted on separatly.
- Provides memory debugging features as well, things like checking pointers, array bounds, etc.
- Provides functionality to interact reasonable with STL components (ie you can see what a map actually contains) and create views for your own objects.
- Allows viewing of MPI message queues for running processes

DDT Demo

- By far the best way to show what DDT can do is to start it up and look at it and show some things with it. Once we do this, we'll have everybody log in and make sure they can DDT started.
- We'll talk about:
  - Creating and altering groups
  - Stepping groups and processes
  - Show Cross-group comparison
  - Show Memory Usage/Profiling
  - Show MPI Queues
  - Show multi-dimensional array viewer

Starting DDT

- Login to ranger with an X tunnel

```
$ ssh -X ranger.tacc.utexas.edu
```

- We need a binary compiled with debugging flags. If you don't have a binary already on ranger, you can get one from the train00 directory

```
login3% mkdir ~/ddt
login3$ cp ~train00/ddt_debug/debug_code.f .
```

- Ensure you have your preferred compiler loaded

```
login3% module list
login3% module unload mvapich
login3% module swap pgi intel
login3% module load mvapich
```

Starting DDT

- Compile with debugging flags

```
login3% cd ~/ddt
login3% mpif90 -g -O0 debug_code.f -o ddt_app
```

- Load the DDT module

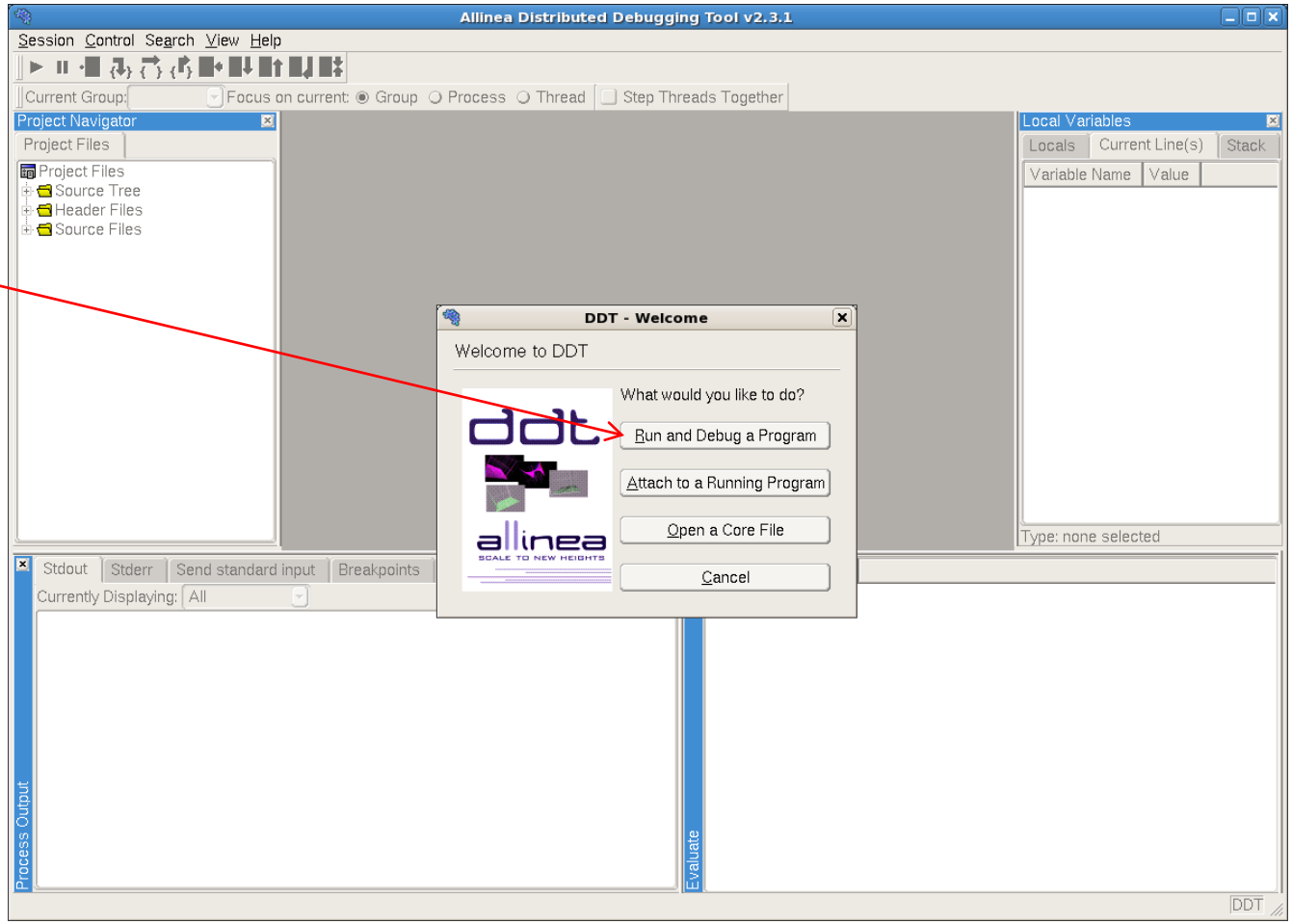
```
login3% module list
login3% module load ddt
login3% module list
login3% echo $DDTROOT
```

- Start DDT

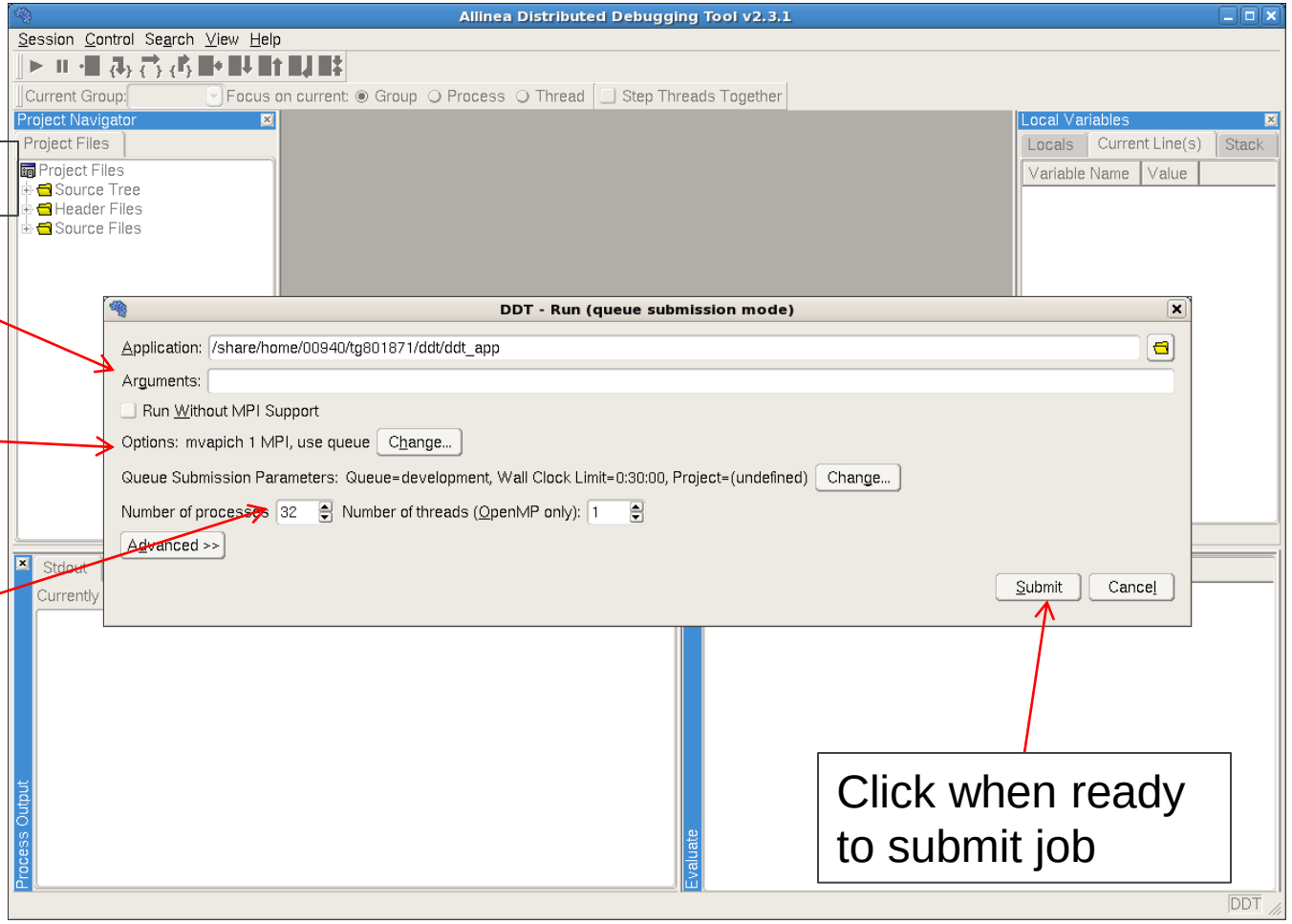
```
login3% ddt ddt_app
```

Starting DDT

Click!



Running a job



Add any arguments

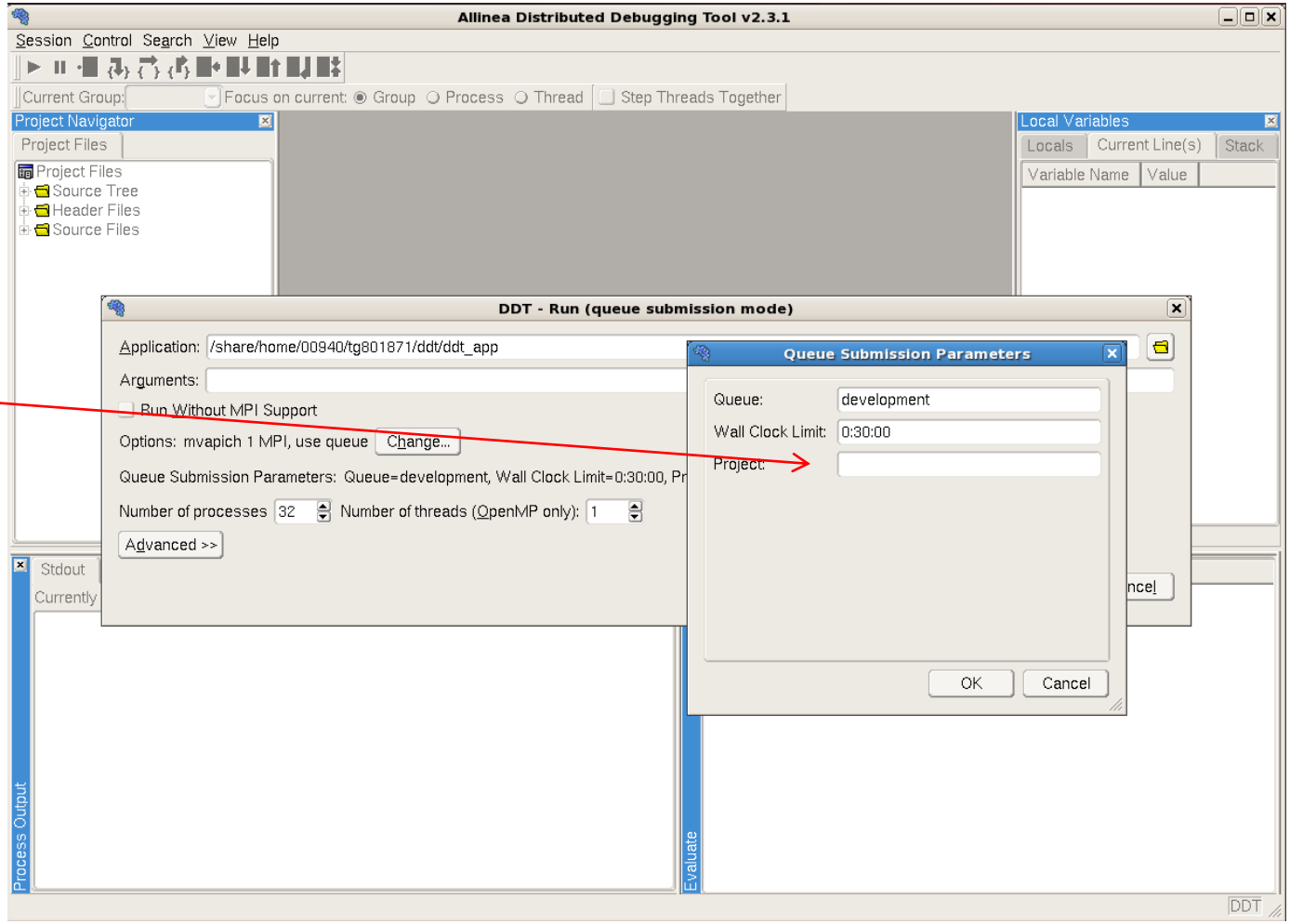
Ranger default

Sets number of nodes

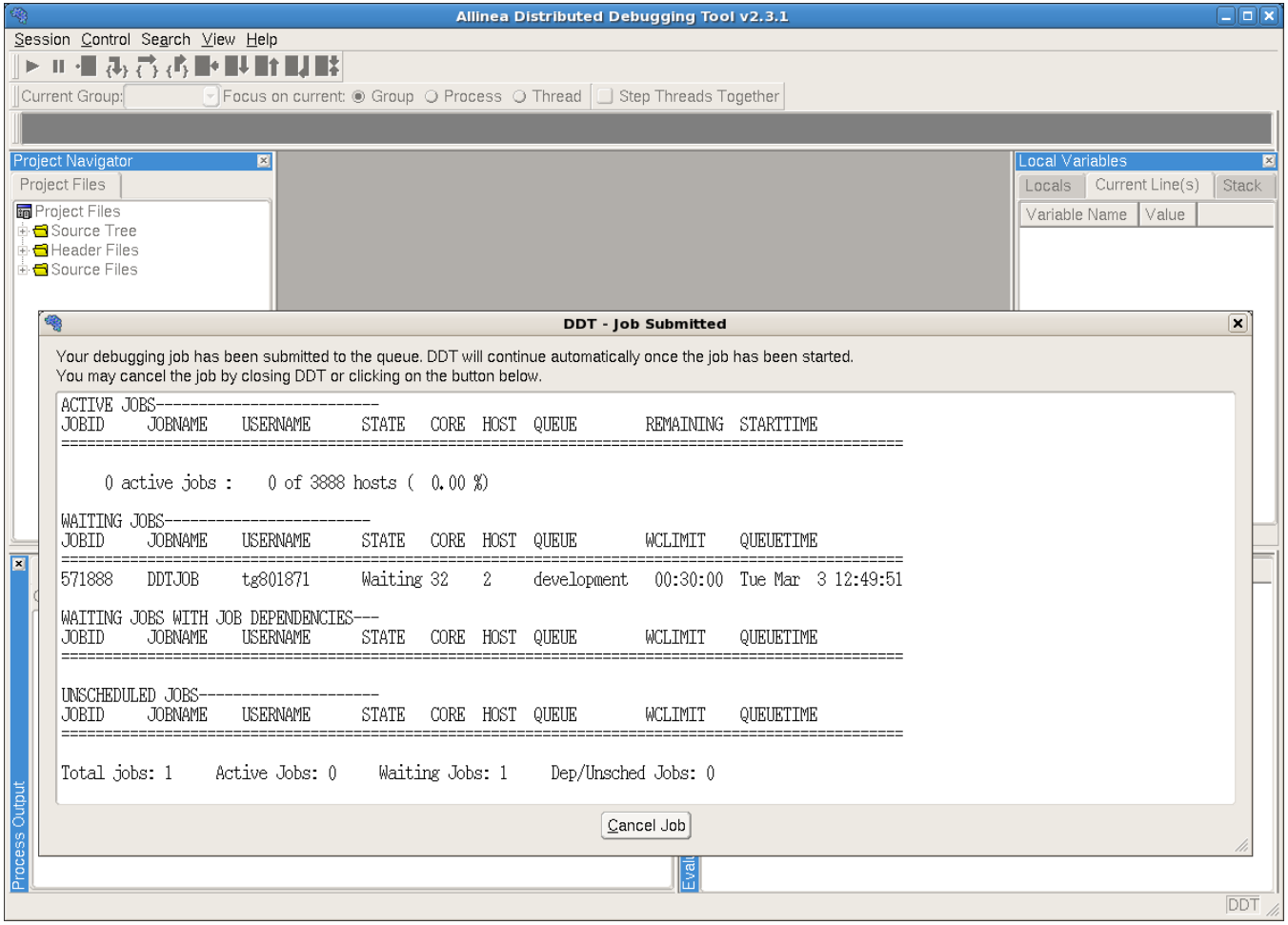
Click when ready to submit job

Account Name

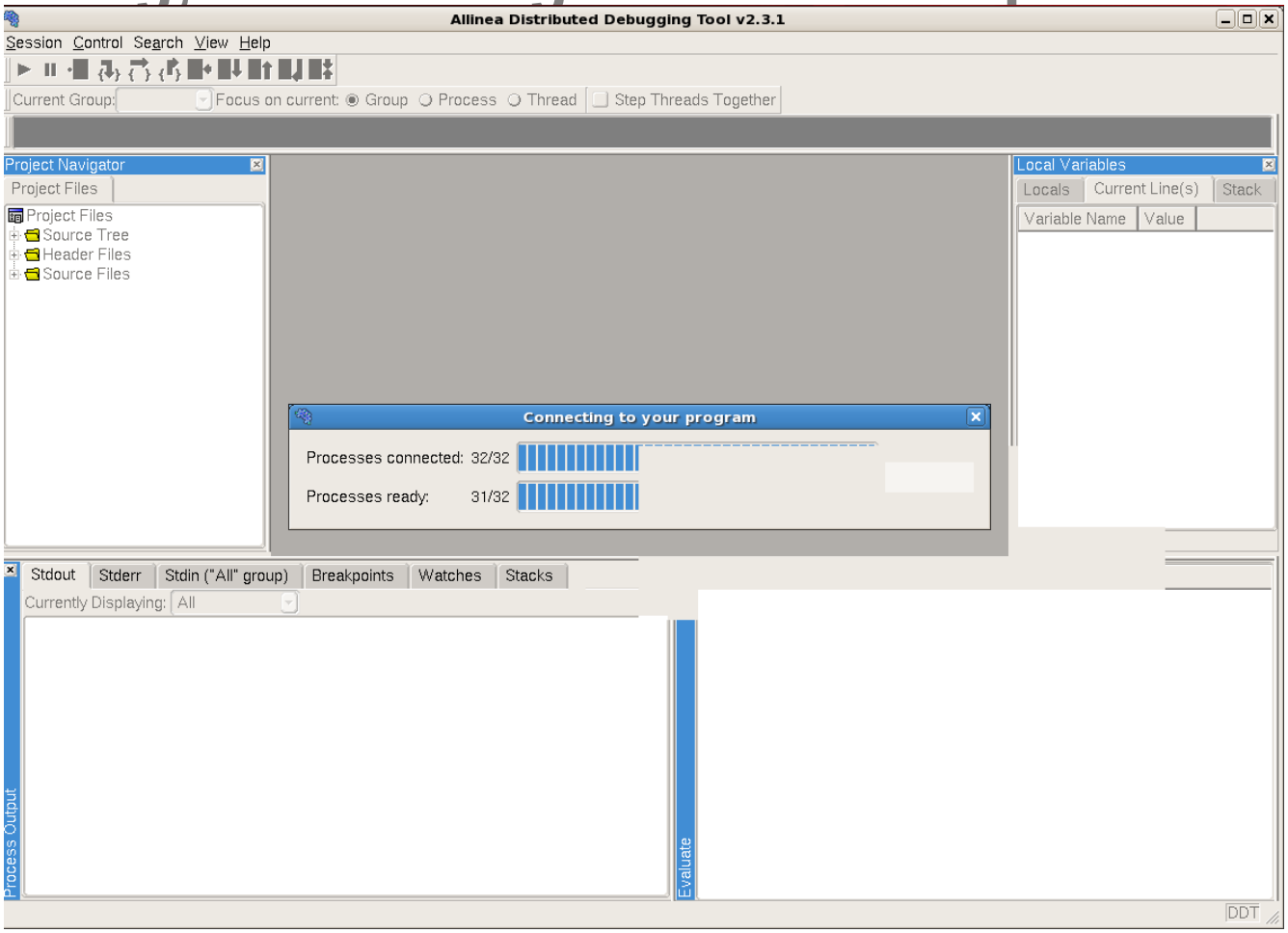
Provide allocation id (qsub -A value) then click OK



Waiting for job to start



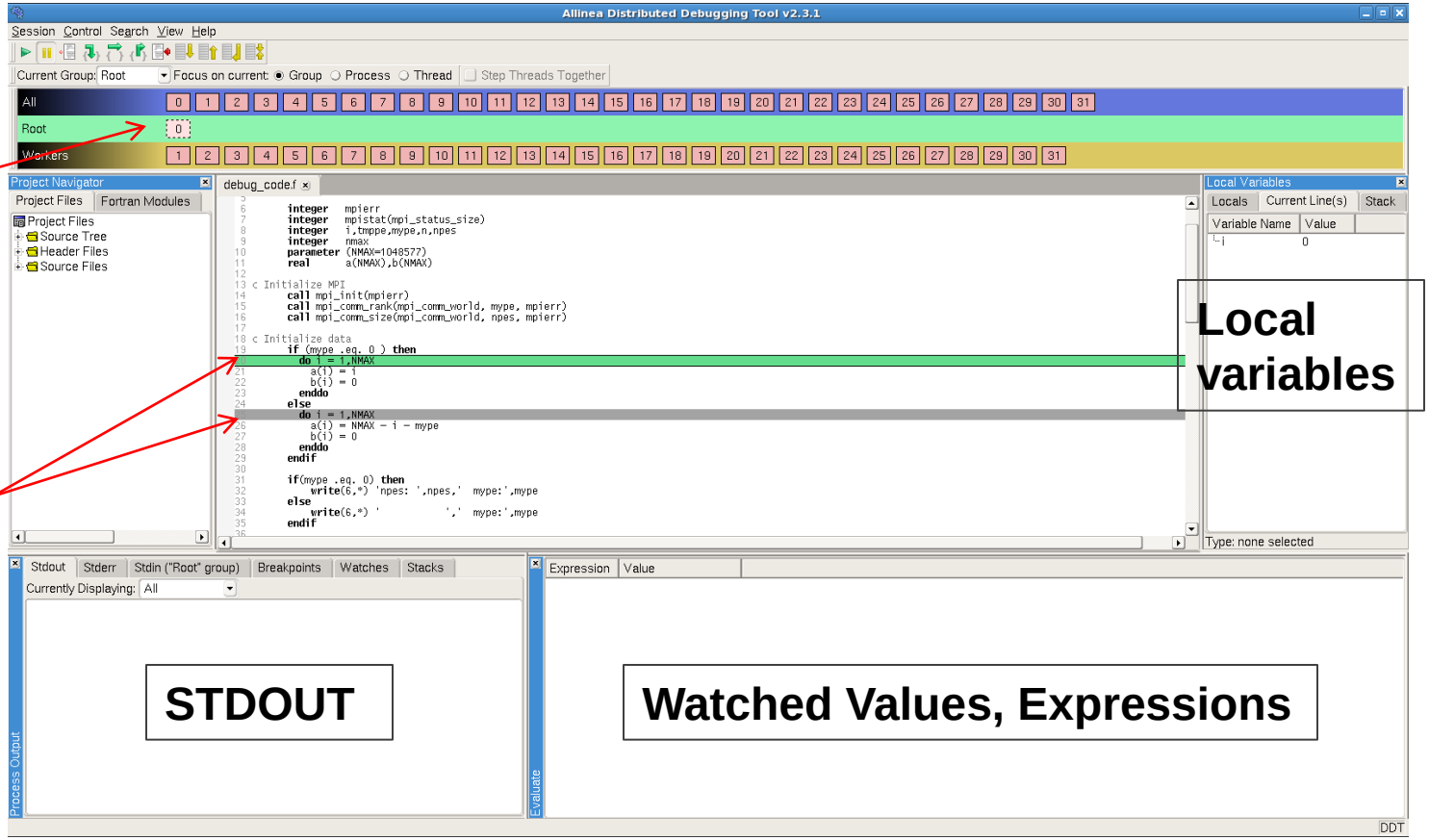
Job starting, connecting to all remote processes



Session started!

Root process is selected

Source locations of processes



DDT

- At this point, DDT should be up and running for you and you only need to load the DDT module and any configuration changes you made (ie Account name) will be saved for the next time you use it.
- It should feel very much like an IDE debugger, just with the added capabilities of viewing remote processes and MPI information.
- It wasn't shown, but this can be used just as well to debug OpenMP programs, though you may need to be careful when stepping through non-threaded sections. Check out the User Guide for any questions you have or request help through the TeraGrid help desk.
- UserGuide: http://www.allinea.com/downloads/userguide.pdf

  Or press F1 while running DDT to call up the help.

DDT Lab

- The DDT Lab is a free-form opportunity to get DDT running.
- Open an SSH session with an X-tunnel to ranger.tacc.utexas.edu and get the example code:

```
login3$ cp ~train00/ddt_debug/debug_code.f .
```

- Compile

```
login3% mpif90 -g -O0 debug_code.f -o ddt_app
```

- Load the DDT Module and run ddt

```
login3% module load ddt
login3% ddt ddt_app
```