The Distributed Debugging Tool

Quickstart Guide
for C programs
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>2</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2. Starting A Session</td>
<td>4</td>
</tr>
<tr>
<td>Checking Your Program Status</td>
<td>4</td>
</tr>
<tr>
<td>3. Controlling Your Processes</td>
<td>6</td>
</tr>
<tr>
<td>Advancing Processes Forward By A Line</td>
<td>6</td>
</tr>
<tr>
<td>Synchronizing Processes At A Line</td>
<td>6</td>
</tr>
<tr>
<td>Stepping Into A Function</td>
<td>6</td>
</tr>
<tr>
<td>Stepping Out Of A Function</td>
<td>7</td>
</tr>
<tr>
<td>Following If Branches</td>
<td>7</td>
</tr>
<tr>
<td>Pausing Processes At A Breakpoint</td>
<td>7</td>
</tr>
<tr>
<td>Manually Pausing Processes</td>
<td>7</td>
</tr>
<tr>
<td>4. Viewing Data</td>
<td>9</td>
</tr>
<tr>
<td>Current Line</td>
<td>9</td>
</tr>
<tr>
<td>Local Data</td>
<td>9</td>
</tr>
<tr>
<td>Keeping An Expression In View</td>
<td>9</td>
</tr>
<tr>
<td>Finding A Segmentation Fault</td>
<td>10</td>
</tr>
<tr>
<td>6. Conclusion</td>
<td>12</td>
</tr>
</tbody>
</table>
1. Introduction

In this document you will use the example program provided with DDT to gain an overview of DDT and it's features. You will see how DDT can be used to debug a C program using the basic features found within DDT. After following this guide you should be able to start a program, control your processes, examine data, set breakpoints and backtrack from a segmentation fault.
2. Starting A Session

By now you should have installed DDT successfully. In your DDT directory you will find the examples subdirectory. We will now compile this:

```
[examples]$ make -f cmake.makefile
mpicc -c -g -O0 -o hello.o hello.c
mpicc -g -O0 -o hello hello.o
[examples]$
```

To verify that you have MPI working correctly, run the example without DDT first.

```
[examples]$ mpirun -np 4 ./hello
```

On termination, the hello program will, if your MPI supports I/O (input/output) from remote processes, have produced output that ends:

```
[....]
all done...(1)
all done...(0)
[examples]$
```

We now start DDT debugging this program:

```
[examples]$ ../bin/ddt ./hello
```

Note: if DDT is in your path, you can just type `ddt ./hello`. If this is the first time that you have used DDT you will be greeted with the configuration wizard. This will walk you through some of the steps necessary for setting up DDT to recognize your MPI implementation and the remote access to nodes.

Once configuration is complete, DDT will present a dialog box; you should now ensure that the right number of processes are selected. The advanced menu can be used for supplying extra parameters to your executable of your MPI, as well as to enable memory debugging.

```
You should now see a DDT session control window looking something like Fig.1. Select “Run” and DDT will connect to your processes, load your source files and take you into the main DDT window, ready to begin your debugging.
```

Checking Your Program Status

After loading the example program into DDT (as described above) your main DDT window should look like the one seen in Fig.2. The source files that have been found are shown, and the current
file, that from which MPI_Init is called, will be loaded and shown. You can examine any of your source files at any time by selecting it from the list. Note that for some compilers and MPI implementations the line after MPI_Init will be highlighted instead, as it is the next one to be executed.

At the top of the screen is a collection of coloured numbered boxes, and three lines: “All”, “Root” and “Workers”. These are process groups. Process groups are used to control your processes en-masse. Each numbered box represents a process. One of the lines will be brighter than the others, and one of the boxes may be brighter and have a dotted border. These are the current group and the current process (if one is selected) respectively.

The process boxes are either red or green. A green process is running, and a red process is stopped, paused or terminated.

You should see a red bar on the line containing MPI_Init. This indicates that some processes in the red process group are on that line, and the red process group is the current group. If the line is bright, this also means that the currently selected process is on that line. Hover your mouse pointer over this line and DDT will tell you which processes are at that point. Select a different process group by clicking on “Workers”. The red line will change to the colour of the Workers group.

Information about which line a process is on is updated every time the program is paused.

![DDT main window](image-url)
3. Controlling Your Processes

Process groups as described above are used to control most of the operations you will wish to perform on your processes. If you wish to make all processes stop at a point in your code, or all processes move a single step, the “All” process group would be selected. Alternatively, if none of the predefined groups are exactly right for you, by right clicking in the process group area, you can create your own using drag and drop moves.

The next step in our exercise will be to advance all the processes through the code. We will be using the process operation buttons found in the toolbar just above the process groups (see Fig.3). These buttons are (from left to right): Play, Pause, Step In, Step Over, and Step Out. The last four change the current stack frame and do not cause your processes to execute any code at all. From left to right, they move down the stack, up the stack, to the bottom of the stack and set all processes in the group to the current stack frame. Navigating the stack is described in more detail in the main userguide.

![Fig.3 Process operation buttons](image)

**Advancing Processes Forward By A Line**

Select the “All” process group using the mouse, and then press the `Step Over` button. The red line in your code will advance one line, your processes may also briefly turn green whilst they perform the step, returning to red when they have finished working.

**Synchronizing Processes At A Line**

We might want to make our processes run to a particular point in the code – to do things a bit quicker than repeated advancing a single line. To achieve this, find the line in your code, and right click – you will see a “Run to here” option. This will run the currently selected group, or the single process in single process mode, up to the selected line. Note that if your processes do not visit this line subsequently, that DDT will time out and stop waiting after a short period of time, and then you will be back in control of the processes again. It's also worth noting that any processes currently on that line will be executed until they return to the line! This is very useful inside a loop, but may catch you out if you expected them to wait there for the others to catch up.

Later in the code, all processes call func1(), we'd like to look in detail here. Find the line (either using the find function or by scrolling down to around line 128) and use 'Run to here' to move all the processes to the line calling func1().

**Stepping Into A Function**

Suppose we now wish to see what happens inside func1(). Press the `Step Into` button, and the code window leaps to the relevant part of the source code. The processes have now moved into func1().

Do this again to enter func2().
Stepping Out Of A Function

You should now be seeing the definition of func2() and it looks uninteresting so lets go back up to func1() and see what happens there. Click the ‘Step Out’ button. The processes will now continue until they reach the end of func2() and then return to func1().

After this the processes are still shown as being on the line of func1() that calls func2(); this is because it hasn’t finished executing the last part of this line, which is where the return value of func2() is assigned to test.

Following If Branches

Probably the most important part of func1() is the if statement and we’d like to watch what happens in there; which path is taken. By stepping over we can watch what happens as the program proceeds. Use the ‘Step Over’ button to do this until you leave func1().

Pausing Processes At A Breakpoint

You could keep using ‘Step Over’ until you have reached a point in your code that is important to you, but this would be time consuming. It’s also possible that some processes may take more instructions to reach a point of interest than others, leading to different processes being on different lines of code. A better idea is to cause your processes to stop at a breakpoint.

We’d like to see what is happening during an MPI operation - where process 0 is receiving data from all of the other processes. Select the process group “Workers” and then click on the line containing MPI_Send; line 146. Click the right mouse button and choose “Add breakpoint”, this will add a breakpoint for the currently selected group.

Now select the “All” process group, and press play to resume execution. After a short period of computation, you will see that process 0 is Green, still running, and processes 1, 2 and 3 have reached the breakpoint.

You can have as many breakpoints as you like, each breakpoint you set will be active for the currently selected process group. You can see all the breakpoints that are set by clicking on the breakpoints tab at the bottom of the window.

Manually Pausing Processes

Process 0 is still running, but we can guess that it will be waiting to receive data and is not actually “working”. By selecting either the “Root” or the “All” group if neither is selected, and then pressing ‘Pause’, process 0 can now be stopped.

Select process 0 and you will see the stack, located to the right side of the code. This can be viewed by clicking the mouse over the stack display window, which will currently be displaying the top of the stack.

If you double click on process 0 the code window will jump to the point in the code where process 0 has paused and highlight the line in the code, and it’s associated variables.
All of the process operation buttons, and the setting of breakpoints work on the currently selected process group. The ability to configure and save your own groups is a very powerful feature making it much easier to debug your programs using DDT.
4. Viewing Data

Now that you know how to control processes in DDT, we are now ready to look at the data or contents of the processes.

Current Line

By clicking on process 3, for example, we can examine the data on the current line of code. The process is about to send a message, and we can view the message. First select line 146 or process 3 and then click on the “Current Line(s)” tab, to the right of the window, if it is not already showing.

The current line panel shows the variables that used on the current line of code for the currently selected process. In this case we can see the “message” variable, which is a string. The contents of the string are displayed.

You can see more than the currently selected line by dragging the mouse around the lines of interest in the code window.

![Fig.4 Current line for process 3](image)

Local Data

The variables known as “locals”, those in scope in your current function, can be seen by clicking the locals tab.

Keeping An Expression In View

Perhaps you would like to keep an eye on the values in “status”, which can be seen by selecting the “locals” tab.

Status is the MPI structure containing success information regarding our MPI receive commands; we’d like to watch this as we proceed. We can do this by putting expressions into the evaluate window. This is found below the current line/locals window.
You can drag “status” from the locals window into the evaluate window using mouse drag and drop techniques, or you can type it in directly by right clicking in the evaluate window and selecting “Add expression”.

Status is a structure: its contents can be seen by clicking the “+” preceding the variable name. This will open up a hierarchical view of the structure. If you have structures within structures you can unfold these too. As each process is selected, everything in the evaluate expression window is re-evaluated on that process.

We'll now make processes 1, 2 and 3 send the data. Press `Step Over` whilst the “Workers” group is selected. On some MPIs the buffering may not be sufficient to let all three communications occur without first setting process 0 running, but that's ok. Now select the green group `Root`, right-click on line 154 (the line after MPI_Recv) and choose `Run to here`. You will see the values in status change, showing that data has been received from a particular process.

**Finding A Segmentation Fault**

The “hello” program was designed to crash; if the argument “crash” is supplied, it will cause a segmentation violation.

We can restart “hello” by selecting `Session` and then the `Session Control` option from the tool bar. Now click on `End Session`. We will now modify the parameters and start debugging again. Click on `Advanced` to display extra functionality. The advanced start up dialog allows you to specify arguments to your program. In the session control dialog, type the word “crash” in the arguments box. Now commence the program as before (click on `Run`). When DDT is ready, press play to start everything going.

![Fig.5 Segmentation fault in DDT](image-url)
One of the processes will bring up a SIGSEGV message (see Fig.5). If you clear this message by clicking `OK`, the process that has failed will be selected by default and you can examine exactly where it was when the segmentation violation occurred.

It is now straightforward to see that as argv[i] is 0, then dereferencing this has thrown the error. You can now go back, edit your program, remove the error, and recompile.
6. Conclusion

You have now mastered the basic operations of DDT for debugging a C program. For a more in
depth look at the possibilities with DDT you should consult our userrguide. Alternatively you can
take a look at the quick start guide and tutorials for Fortran to learn how to use DDT to debug
Fortran 77 or Fortran 90 code.