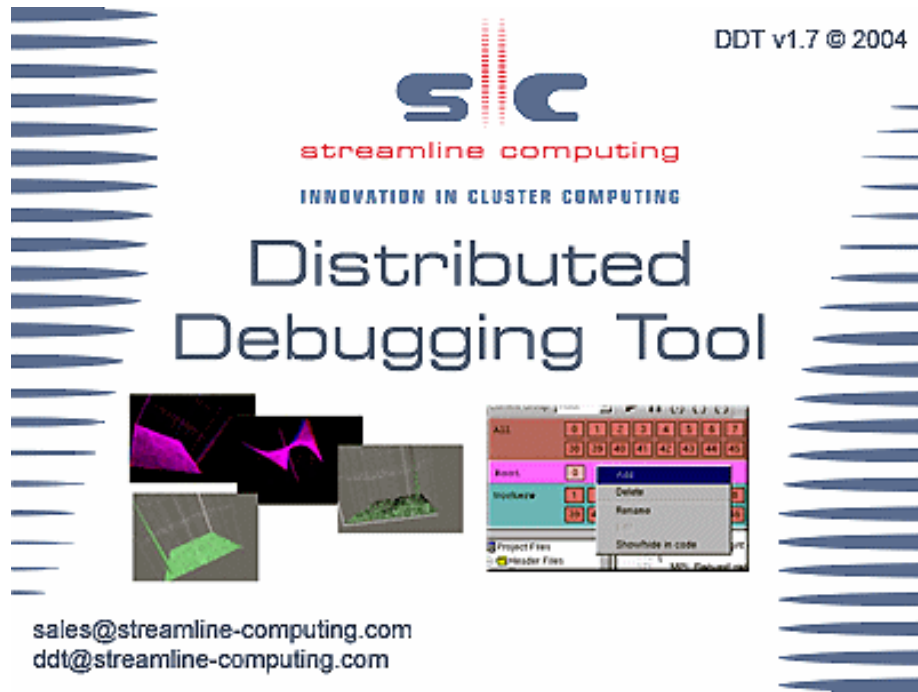


The Distributed Debugging Tool



Quickstart Guide (C programs)



Contents

Contents	2
1. Introduction.....	3
2. Starting A Session	4
Checking Your Program Status	5
3. Controlling Your Processes	7
Advancing Processes Forward By A Line.....	7
Stepping Into A Function	7
Stepping Out Of A Function.....	7
Following If Branches	8
Pausing Processes At A Breakpoint.....	8
Manually Pausing Processes	8
4. Viewing Data.....	9
Current Line.....	9
Local Data.....	10
Keeping An Expression In View.....	10
Finding A Segmentation Fault.....	10
5. Conclusion.....	12

1. Introduction

In this document you will use the example program provided with DDT to gain an overview of DDT and its 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 processors, have produced output that ends:

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

We now start DDT debugging this program:

```
[examples]$ $DDT ./hello &
```

DDT will present a dialog box; you should now ensure that the right number of processors are selected as well as the correct debugger interface for your compilers and the correct MPI implementation.

If you are unsure of which MPI implementation to choose, look in the FAQ for more help. The most common MPI is “generic”.

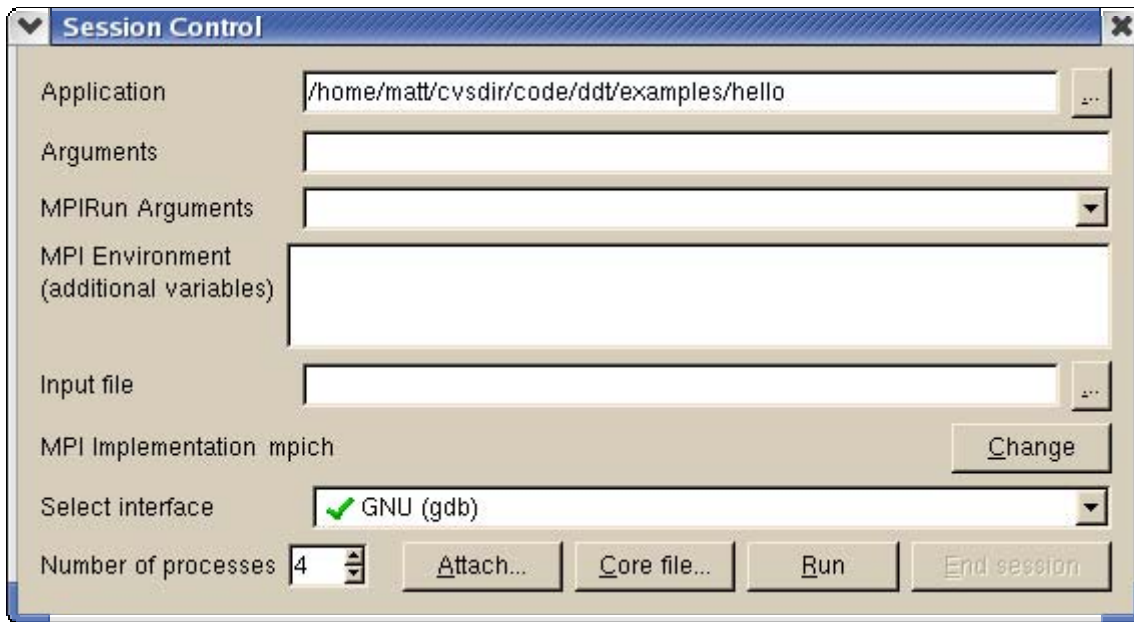


Fig.1 The session control window

You should now see the DDT session control window as in 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.

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 processor is on is updated every time the program is paused.

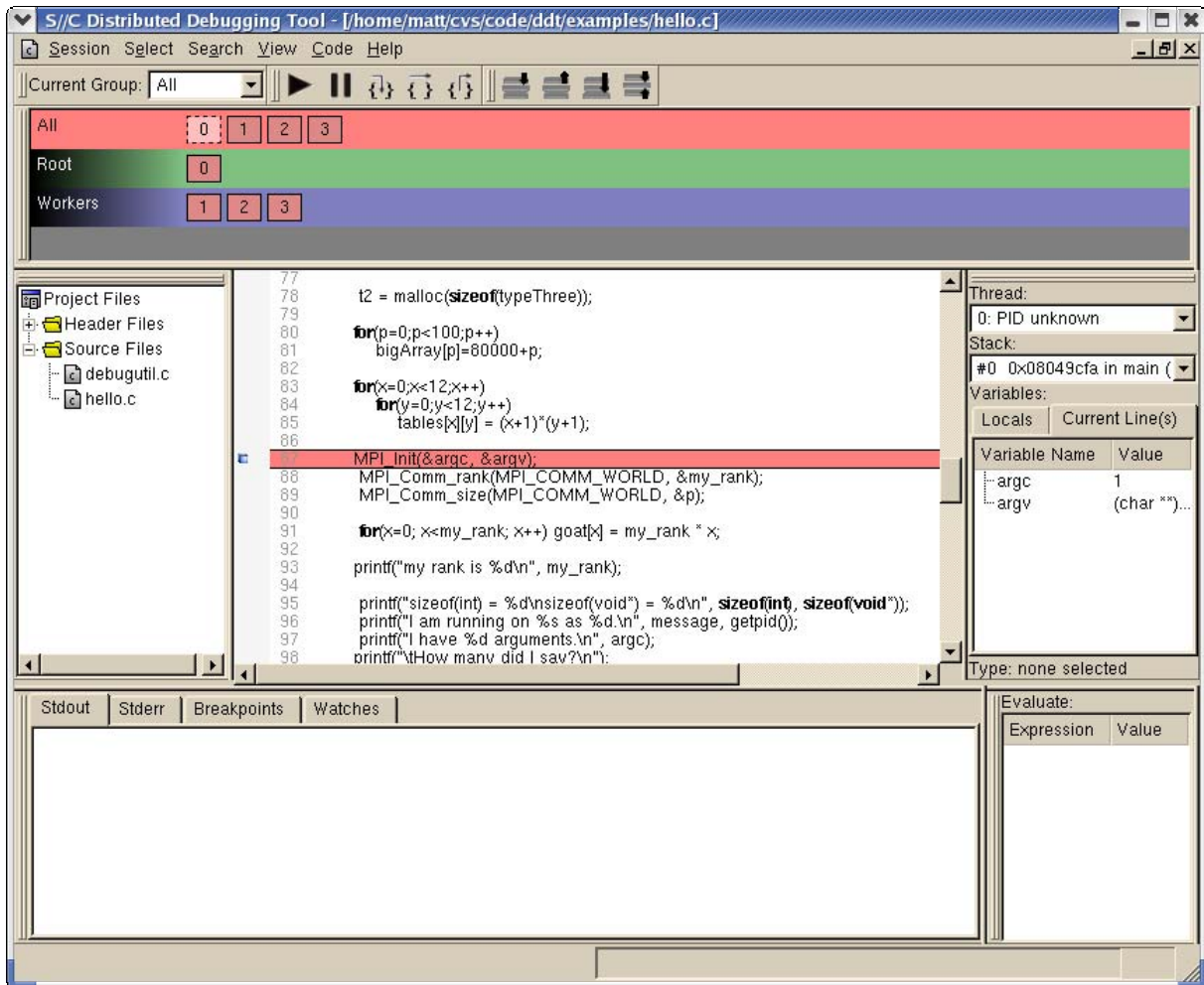


Fig.2 DDT main window

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 below the process groups (see Fig.3). These buttons are (from left to right): Play, Pause, Step In, Step Over, and Step Out.



Fig.3 Process operation buttons

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.

In the example, press step over again - until the processes are at the line which calls `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 139. 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 will bring the stack frame up to the correct stack to view the currently highlighted 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 any of the paused processes, we can examine the data on the current line of code. If we choose any of the processes that have just executed a send procedure we can view the message that is about to be sent to process 0. First select line 139 and then click on the “Current Line” tab, to the right of the window.

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, and the first character of its contents. Let’s view the whole contents. To do this simply drag the “message” variable into the `Evaluate` window and drop it there. Now right click on the “message” variable inside the `Evaluate` window and choose the `View as vector` option. Enter the number of elements you wish to see, in this case choose 25, and click OK. You will see a “+” symbol appear next to the left of the variable name. Click on this and you will see the whole message appear in a list. Click on a different process and this message will update to reflect the one held by the current process.

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

Locals		Current Line(s)	
Variable Name		Value	
dest		0	
message		"G"	
tag		50	

Type: none selected

Evaluate:	
Expression	Value
+ message	
[0]	71 'G'
[1]	114 'r'
[2]	101 'e'
[3]	101 'e'
[4]	116 't'
[5]	105 'i'
[6]	110 'n'
[7]	103 'g'
[8]	115 's'
[9]	32 ' '
[10]	102 'r'
[11]	114 'r'
[12]	111 'o'
[13]	109 'm'
[14]	32 ' '
[15]	112 'p'
[16]	114 'r'
[17]	111 'o'
[18]	99 'c'
[19]	101 'e'
[20]	115 's'
[21]	115 's'
[22]	32 ' '
[23]	51 '3'
[24]	33 'I'

Fig.4 Current line for process 3

Local Data

The variables known as “locals”, those in scope in your current function, and in the case of Fortran most other global variables, can be seen by clicking the locals tab.

Keeping An Expression In View

It would be useful to keep a particular eye on the values in “status”, the MPI structure containing success information regarding our MPI send/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 expression window using mouse drag and drop techniques, or you can type it in directly by right clicking in the evaluate window and selecting “Evaluate expression”.

As status is a structure, it’s contents must be seen; do this 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.

We’ll now make processes 1, 2 and 3 send the data. Press `Step Over` whilst the “Workers” group is selected. Now examine the status structure and everything should be well. On some MPIs the buffering may not be sufficient to let all three communications occur without first setting process 0 running, you will be able to use the process operation buttons talked about in the previous section to find this out if your processes do not stop and instead are running but waiting on I/O.

Finding A Segmentation Fault

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

We can restart “hello” by selecting `Session` and then the `Restart` option from the tool bar. Then click on `End Session`.

In the session control dialog, type the word “crash” in the arguments box. Now commence the program as before. When DDT is ready, press play to start everything going.

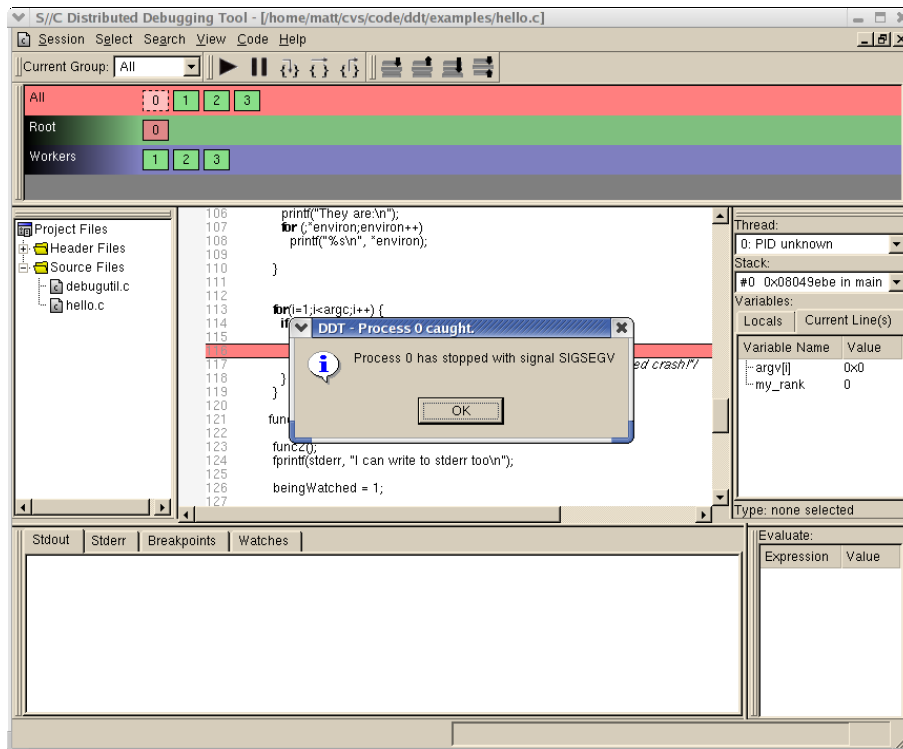


Fig.5 Segmentation fault in DDT

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.

5. 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 can attempt one of the two tutorials (tutorial-1 or tutorial-2) supplied with DDT.

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.

For more help on the features of DDT see the supplied Userguide.