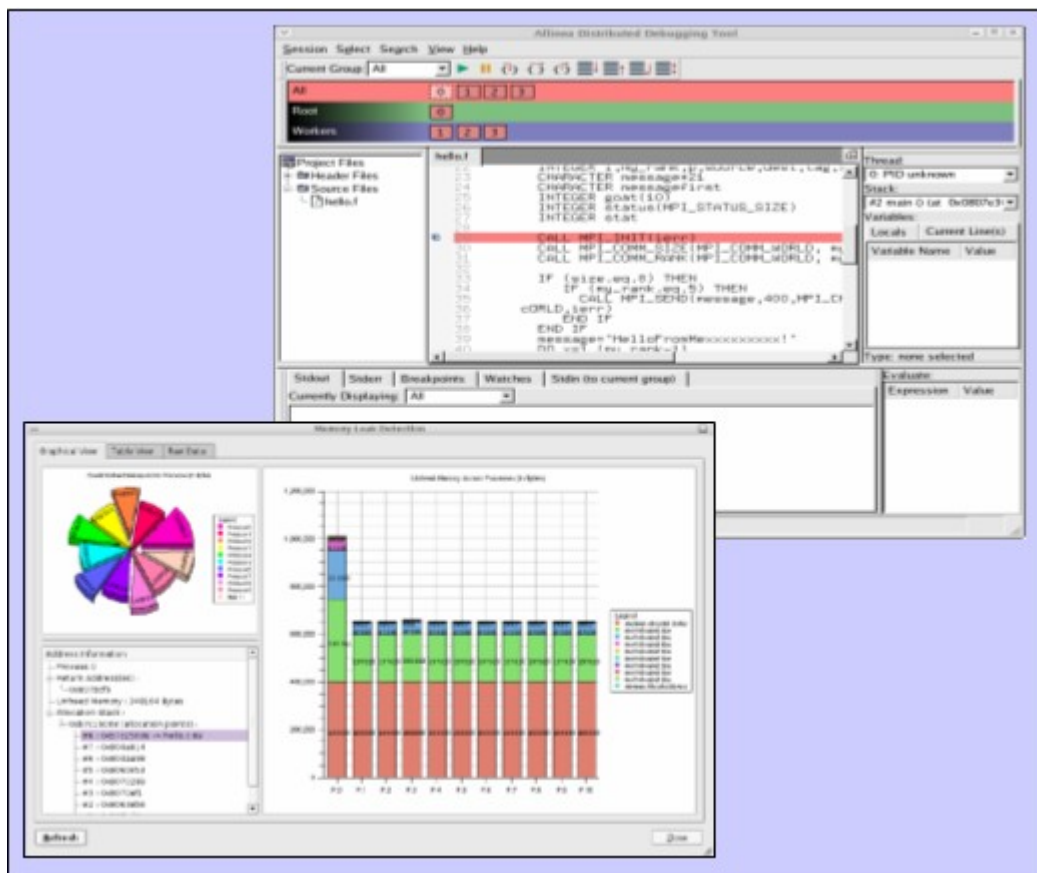


The Distributed Debugging Tool

User Guide



Contents

Contents.....	1
1. Introduction.....	5
2. Installation and Configuration.....	6
Installation.....	6
Licence Files.....	6
Floating Licences.....	7
Configuration.....	7
Site Wide Configuration.....	12
Advanced Configuration - Integrating DDT With Queuing Systems.....	13
The Template Script.....	13
Configuring Queue Commands.....	14
Getting Help.....	16
3. Starting DDT.....	17
Debugging Multi-Process Programs.....	17
Notes on the MPICH Standard Option.....	19
Debugging Single-Process Programs.....	20
Debugging A Core File.....	20
Attaching To Running Programs.....	21
Using DDT Command-Line Arguments.....	23
Starting A Job In A Queue.....	24
Using Custom MPI Scripts.....	24
4. DDT Overview.....	28
Setting The Font and Tab Sizes.....	29
Saving And Loading Sessions.....	29
Source Code.....	29
Finding Lost Source Files.....	29
Dynamic Libraries.....	30
Finding Code Or Variables.....	30
Jump To Line / Jump To Function.....	30
Editing Source Code.....	31
5. Controlling Program Execution.....	32
Process Control And Process Groups.....	32
Hotkeys.....	33
Starting, Stopping And Restarting A Program.....	33
Stepping Through A Program.....	34
Setting Breakpoints.....	34
Conditional Breakpoints.....	35
Suspending Breakpoints.....	35
Deleting A Breakpoint.....	35
Loading And Saving Breakpoints.....	35
Synchronizing Processes.....	36
Setting A Watch.....	36
Examining The Stack Frame.....	37
Align Stacks.....	37
Examining Threads.....	37
"Where are my processes?" - Viewing Stacks in Parallel.....	38
Overview.....	38
The Parallel Stack View in Detail.....	39
Browsing Source Code.....	40
Simultaneously Viewing Multiple Files.....	41
Signal Handling.....	42

6. Variables And Data.....	43
Current Line.....	43
Local Variables.....	43
Arbitrary Expressions And Global Variables.....	44
Help With Fortran Modules.....	44
Viewing Array Data.....	46
Changing Data Values.....	46
Examining Pointers.....	46
Multi-Dimensional Arrays in the Variable View.....	46
Examining Multi-Dimensional Arrays.....	47
Visualizing Data.....	48
Cross-Process Comparison.....	49
Assigning MPI Ranks.....	51
Viewing Registers.....	52
Interacting Directly With The Debugger.....	53
7. Program Input And Output.....	54
Viewing Standard Output And Error.....	54
Displaying Selected Processes.....	54
Saving Output.....	54
Sending Standard Input (DDT-MP).....	55
8. Message Queues.....	56
9. Memory Debugging.....	58
Configuration.....	58
Feature Overview.....	59
Stop on Error.....	59
Check Validity.....	60
View Pointer Details.....	60
Writing Beyond An Allocated Area.....	60
Memory Statistics.....	62
10. Advanced Data Display and C++ STL Support.....	64
Using The Sample Wizard Library.....	64
Writing A Custom Wizard Library.....	65
Theory Of Operation.....	66
Compiling A New Wizard Library.....	67
Using A New Wizard Library.....	68
The VirtualProcess Interface.....	68
The Expression Class.....	70
Final String-Enabled Wizard Library Example.....	71
11. The Licence Server.....	74
Running The Server.....	74
Running DDT Clients.....	74
Logging.....	74
Troubleshooting.....	75
Adding A New Licence.....	75
Examples.....	75
Example Of Access Via A Firewall.....	76
Querying Current Licence Server Status.....	76
Licence Server Handling Of Lost DDT Clients.....	77
A. Supported Platforms.....	79
B. Troubleshooting DDT.....	80
Problems Starting the DDT GUI.....	80
Problems Starting Scalar Programs.....	80
Problems Starting Multi-Process Programs.....	81
C. FAQs.....	82

Why does DDT say it can't find my hosts or the executable?.....	82
The progress bar doesn't move and DDT 'times out'.....	82
The progress bar gets close to half the processes connecting and then stops and DDT 'times out'.....	82
My program runs but I can't see any variables or line number information, why not?.....	82
My program doesn't start, and I can see a console error stating "QServerSocket: failed to bind or listen to the socket".....	83
Why can't I see any output on stderr?.....	83
DDT complains about being unable to execute malloc.....	83
Some features seem to be missing (e.g. The Fortran Module Browser) - what's wrong?.....	83
My code does not appear when I start DDT.....	83
When I use step out my program hangs.....	84
When viewing messages queues after attaching to a process I get a "Cannot find Message Queue DLL" error.....	84
I get the error `The mpi execution environment exited with an error, details follow: Error code: 1 Error Messages: "mprun:mpmd_assemble_rsrcs: Not enough resources available" when trying to start DDT.....	84
What do I do if I can't see my running processes in the attach window?.....	85
When trying to view my Message Queues using mpich I get no output but also see no errors.....	85
After I reload a session from a saved session file, the right-click menu options are not available in the file selector window.....	85
The View Pointer Details window says my pointer is valid but doesn't show me which line of code it was allocated on.....	85
Obtaining Support.....	85
D. Notes On MPI Distributions.....	87
Bproc.....	87
Bull MPI.....	87
HP MPI.....	87
Intel MPI.....	88
LAM/MPI.....	88
MPICH p4.....	88
MPICH p4 mpd.....	88
MPICH-GM.....	88
MPICH SHMEM.....	88
IBM PE.....	89
MVAPICH.....	89
NEC MPI.....	89
OpenMPI.....	89
Quadrics MPI.....	89
SCore.....	90
Scyld.....	91
SGI Altix.....	91
Sun Clustertools.....	91
E. Compiler Notes.....	92
Absoft.....	92
GNU.....	92
IBM XLC/XLF.....	92
Intel Compilers.....	93
Pathscale EKO compilers.....	93
Portland Group Compilers.....	93
Sun Forte Compilers and Solaris DBX.....	94

F. Architectures.....	95
IBM AIX Systems.....	95
AMD Opteron and Intel EM64T.....	95
Intel Itanium.....	95
Intel Xeon/Pentium 32 bit.....	95
Index.....	96

1. Introduction

The Distributed Debugging Tool is an intuitive, scalable, graphical debugger. This document introduces DDT and explains how to use it to its full potential. If you just wish to get started with DDT, you will find that the examples directory of your DDT installation contains a quick-start example.

DDT can be used as a single-process (non MPI) or a multi-process program debugger. The availability of these capabilities will depend on the licence that you have - although multi-process licences are always capable of supporting single-process debugging.

Both modes of DDT are capable of debugging multiple threads, including OpenMP codes. DDT provides all the standard debugging features (stack trace, breakpoints, watches, view variables, threads etc.) for every thread running as part of your program, or for every process - even if these processes are distributed across a cluster using an MPI implementation.

DDT can do many tasks beyond the normal capabilities of a debugger – for example the memory debugging feature detects some errors before they have caused a program crash by verifying usage of the system allocator, and the message queue integration with MPI can show the current state of communication between processes in the system.

Multi-process DDT encourages the construction of user-defined groups to manage and apply debugging actions to multiple processes. Once you are comfortable working with groups of processes, everything else becomes simple and intuitive. If you have used a visual debugger before you will find DDT's interface familiar. Using DDT to debug MPI code makes debugging parallel code as easy as debugging serial code.

C, C++, Fortran and Fortran 90/95/2003 are all supported by DDT, along with a large number of platforms, compilers and all known MPI libraries.

2. Installation and Configuration

This section describes the first steps necessary before you can begin to debug with DDT.

We begin by describing how to install the software, and then how to configure it.

The steps are simple and short – an ordinary – or root – user can do this.

Configuration can be done by any user for their personal settings, or by a root user who wishes to set the configuration for an entire site (see page 12). DDT's configuration wizard should explain things clearly enough to render reading all of this section unnecessary, if you just wish to get started quickly and if your system is relatively straightforward.

Installation

DDT may be downloaded from the Allinea website. The package is installed by untarring and running the install.sh script.

```
tar xvf ddt1.10-ARCH.tar
./install.sh
```

During this phase of installation, you will be given a choice of where to install DDT. DDT can be installed by a single user in their home directory or by an administrator in a common directory such as /opt or /usr/local.

If you are installing DDT on a cluster, make sure you choose a directory that is shared between the cluster frontend and the cluster nodes. Otherwise you must install or copy it to the same location on each node.

DDT assumes that the home directories on the cluster frontend (where the DDT GUI is run) are also available on the cluster nodes. If this is not true of your cluster then you will need to set the DDT_HOST environment variable to the hostname of the cluster frontend and make sure this variable is propagated to each node.

It is important to follow the instructions in the README file that is contained in the tar file. In particular, you will need a valid licence file – evaluation licences are available from <http://www.allinea.com>.

Due to the vast number of different site configurations and MPI distributions that are supported by DDT, it is inevitable that sometimes you may need to take further steps to get DDT working. For example, it may be necessary to ensure that environment variables are propagated to remote nodes, and that DDT's libraries and executables are available on the remote nodes.

Licence Files

Licence files should be stored as {installation-directory}/Licence, (e.g. /home/bob/ddt).

If this is inconvenient, the user can specify the location of a licence file using an environment variable, DDT_LICENCE_FILE. For example:

```
export DDT_LICENCE_FILE=$HOME/SomeOtherLicence
```

The user also has the choice of using DDT_LICENSE_FILE as the environment variable (American spelling).

The order of precedence when searching for licence files is:

- \$DDT_LICENCE_FILE
- \$DDT_LICENSE_FILE
- {installation-directory}/Licence

If you do not have a licence file, the DDT GUI will not start. A warning message will be presented. For remote MPI processes, you will also require the licence to be installed on the nodes. If this licence is not present, the remote nodes will be unable to connect to the GUI.

Time-limited evaluation licences are available from the Allinea website, www.allinea.com.

Floating Licences

For users with floating licences, the licensing daemon must be started prior to running DDT. It is recommended that this is done as a non-root user – such as “nobody” or a special unprivileged user created for this purpose.

```
{installation-directory}/bin/licenceserver &
```

This will start the daemon, it will serve all floating licences in the current working directory that match Licence* or License*.

The hostname, port and MAC address of the licence server will be agreed with you before issuing the licence, and you should ensure that the agreed host and port will be accessible by users.

DDT clients will use a separate client licence file which identifies the host, port, and licence number.

Log files can be generated for accounting purposes.

For more information on the Licence Server please see page 74 of this document.

Configuration

DDT has a configuration wizard to help simplify setting up DDT and choosing the correct options to start your programs. The first time you run DDT after installing it you may see this dialog:

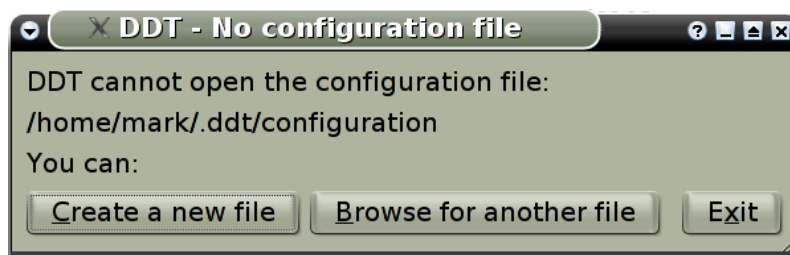


Fig.1 Configuration file finder

DDT has a Configuration Wizard that helps you to set it up to debug programs on your system, whether it is an individual workstation or a four thousand node super-cluster! Most settings will be automatically detected for you, so unless your system administrator has provided a configuration file for you to use, click on "Create a New File" and follow the simple instructions.

Note: this configuration wizard does not help you to set up DDT to submit jobs through a queue. Extensive documentation on the topic can be found on page 13 of this manual.

The first page of the configuration wizard welcomes you to the wizard and explains the process. Click on "Next" to start the process, or "Cancel" if you have changed your mind and would prefer to browse for a configuration file. It's both quick and easy to use the Configuration Wizard, so we recommend clicking on "Next". If you're in a hurry, stop reading this and just click on every "Next" until the wizard finishes! It'll probably do the right thing and you can always come back here through the "Session->Configuration Wizard" menu item if things don't work out.

After the welcome page, you will either see the MPI Configuration page or the 'Attach List' page. Only users with parallel DDT licences will see the MPI Configuration page. If you don't see this page, you can obtain a trial MPI licence from our website, www.allinea.com to see what you're missing.

The page itself looks like this:



Fig.2 Configuration wizard MPI selection

As you can see, there's a detailed description that we won't repeat here. Instead it's worth saying that if the selection box simply says "Click to select..." then DDT was not able to determine which MPI implementation was installed on your system. If you don't know yourself, your system administrator should be able to help you choose one of the options.

If you do know which MPI you have, and you're sure DDT has picked the wrong one, then go ahead and change it, but please contact support@allinea.com so that we can improve this feature in future releases!

Once you have chosen or accepted an MPI Implementation, click on 'Next' to configure DDT for attaching, using the page shown here:



Fig.3 Configuration wizard attach list

DDT is capable of finding and attaching to processes on remote systems, including parallel jobs running on your cluster if you have one. To do this, it needs to know which machines you want it to look on. This takes the form of a list of hostnames, one on each line, like this:

```
comp00
comp01
comp02
comp03
```

Or perhaps like this:

```
localhost
apple.mylab.com
pear.mylab.com
super.mylab.com
```

If you only use DDT on your own machine, you would create a file like this:

localhost

Hopefully your system administrator has a file like this for you and you can click on the '...' button next to "Choose an existing file". If not, you can click on the '...' button next to "Create a new file", choose a filename and then enter a list like the ones above into the big white box called "Nodes:". This is quick and easy, but if you really don't want to do this, you can always click on "Do not configure DDT for attaching at this time". And, next time you want to attach, you can come back to this wizard and set up attaching before you do.

When you're satisfied, click on 'Next'. DDT will now try to find a way to reach these machines without needing you to type in a password.

Important: If DDT is running in the background (e.g. "ddt &") then this process may get stuck (some ssh versions cause this behaviour when asking for a password). If this happens to you, go to the terminal and use the "fg" or similar command to make DDT a foreground process, or run DDT again, without using "&".

If DDT finds a suitable command, it will show a summary screen that tells you that you've finished and can now use DDT! If it fails, then DDT will explain what went wrong and how to correct it with a long explanation that looks something like this:

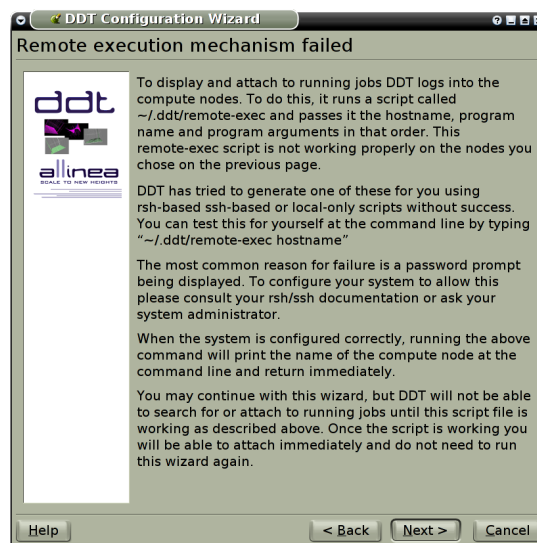


Fig.4 Remote execution error message

If you see this page, read it, understand it, try to perform the corrections it suggests. You may find the "Attaching To Running Programs" section of this manual helpful. Our support staff will be happy to assist you, drop them a mail at support@allinea.com! We all want you to see the page that comes after that, the "Congratulations!" page.

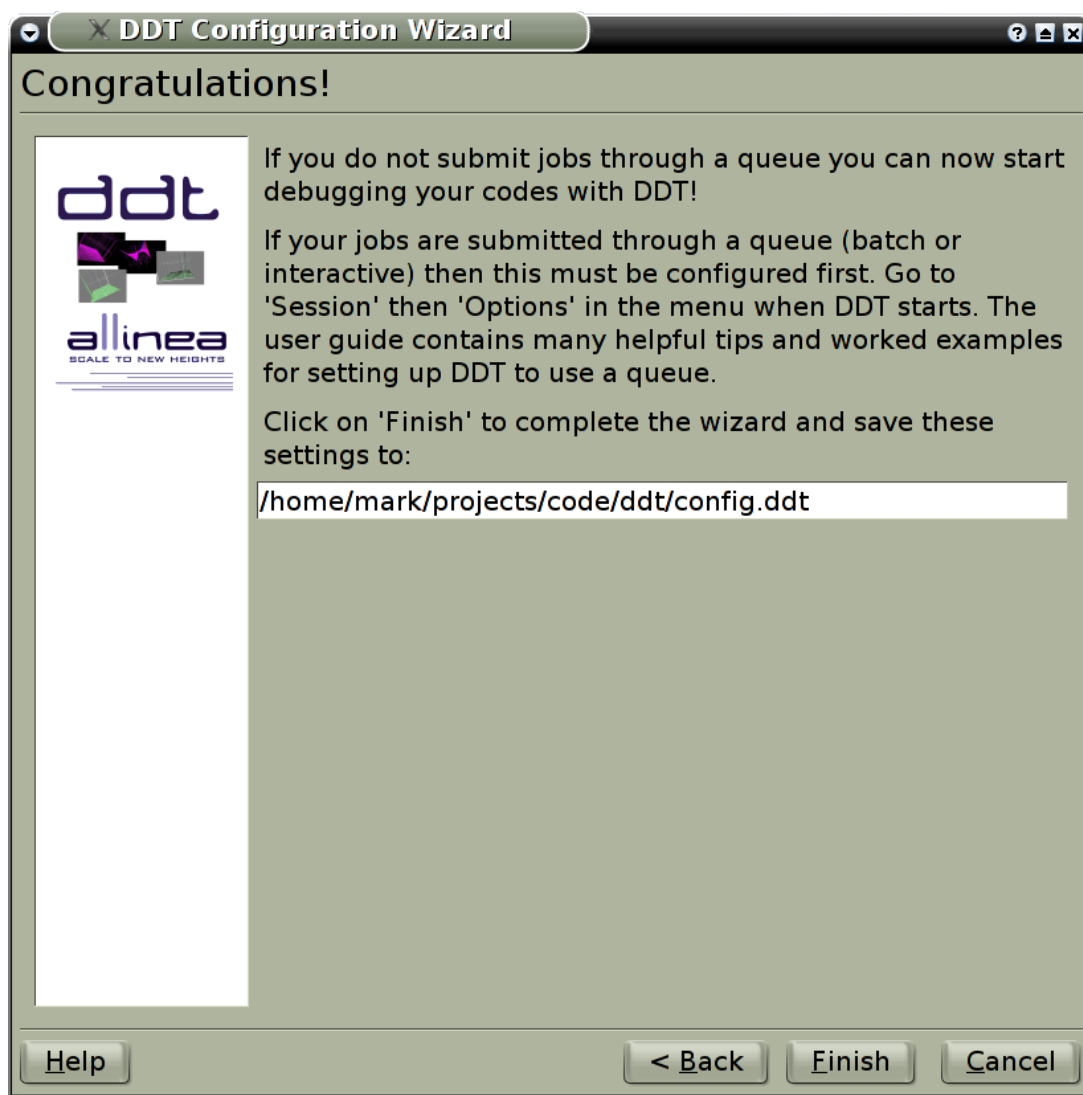


Fig.5 Configuration wizard complete

If you decided not to configure attaching, or if the remote-exec part failed then there will be a small red warning that you will not be able to attach to running programs, but all other features will be available. Note: this may also prevent the MPICH Standard startup method from working.

Click on 'Finish' to save these settings to the configuration file you selected, or 'Cancel' if you've changed your mind.

As stated before, this configuration wizard does not help you to set up DDT to submit jobs through a queue. If you are using a queuing system such as PBS on your cluster then you can find extensive documentation on setting DDT up to use these for debugging elsewhere in this manual.

Site Wide Configuration

If you are the system administrator, or have write-access to the installation directory, you can provide a DDT configuration file which other users will be given a copy of – automatically – the first time that they start DDT.

This can save other users from the configuration process – which can be quite involved if site-specific configuration such as queue templates and job submission have to be crafted for your location.

To provide a configuration file for this purpose, set the DDTCONFIG environment variable to {ddt-installation-path}/config.ddt and launch DDT. Make the appropriate settings for your configuration and then quit DDT. Unset the DDTCONFIG environment variable. DDT will have saved the configuration file in the file specified, and this will be picked automatically for all future first-time users.

Advanced Configuration - Integrating DDT With Queuing Systems

DDT can be configured to work with most job submission systems. In the DDT Options window, you should choose `Submit job through queue`. This displays extra options and switches DDT into queue submission mode.

The basic stages in configuring DDT to work with a queue are:

1. Making a template script, and
2. Setting the commands that DDT will use to submit, cancel, and list queue jobs.

Your system administrator may wish to provide a DDT config file containing the correct settings, removing the need for individual users to configure their own settings and scripts.

In this mode, DDT uses a template script to interact with your queue system. The “templates” subdirectory contains some example scripts that can be modified to meet your needs. {installation-directory}/templates/sample.qtf, demonstrates the process of creating a template file in some detail.

The Template Script

The template script is based on the file you would normally use to submit your job - typically a shell script that specifies the resources needed such as number of processes, output files, and executes `mpirun`, `vmirun`, `poe` or similar with your application. The most important difference is that job-specific variables, such as number of processes, number of nodes and program arguments, are replaced by capitalized keyword tags, such as NUM_PROCS_TAG. When DDT prepares your job, it replaces each of these keywords with its value and then submits the new file to your queue.

Each of the tags that will be replaced is listed in the following table – and an example of the text that will be generated when DDT submits your job is given for each.

Tag	Purpose	After Submission Example
PROGRAM_TAG	Target path and filename	/users/ned/a.out
PROGRAM_ARGUMENTS_TAG	Arguments to target program	-myarg myval
NUM_PROCS_TAG	Total number of processes	16

Tag	Purpose	After Submission Example
NUM_NODES_TAG	Number of compute nodes	8
PROCS_PER_NODE_TAG	Processes per node	2

Ordinarily, your queue script will probably end in a line that starts MPIRUN with your target executable. Your program name should be replaced in this line by {ddt-installation directory}/bin/ddt-debugger. For example, if your script currently has the line:

```
mpirun -np $num_procs $nodes.list program_name myarg1 myarg2
```

You would write:

```
mpirun -np NUM_PROCS_TAG /opt/ddt/bin/ddt-debugger PROGRAM_ARGUMENTS_TAG
```

Configuring Queue Commands

Once you have selected a queue template file, enter submit, display and cancel commands.

When you start the debug session DDT will generate a submission file and append its filename to the submit command you give.

For example, if you normally submit a job by typing `job_submit -u myusername -f myfile` then in DDT you should enter `job_submit -u myusername -f` as the submit command.

DDT Options

System Settings

MPI Implementation:

Select interface: ☒ Automatic (recommended)

☒ Submit job through queue

Queue template file: ...

Submit command:

Regex for job id:

Cancel command:

Display command:

Template uses:

☒ NUM_PROCS

☐ NUM_NODES and PROCS_PER_NODE

PROCS_PER_NODE:

Code Viewer Settings

Tab size:

Font name:

Font size:

Other Settings

Default groups file: ...

Attach hosts file: ...

Ok Cancel

Fig.11 Queuing systems

To cancel a job, DDT will use a regular expression you provide to get a value for JOB_ID_TAG. This tag is found by using regular expression matching on the output from your submit command.

This is substituted into the cancel command and executed to remove your job from the queue. The first bracketed expression in the regexp is used in the cancel command. The elements listed in the table are in addition to the conventional quantifiers, range and exclusion operators.

Element	Matches
C	A character represents itself
\t	A tab
.	Any character
\d	Any digit
\D	Any non-digit
\s	Whitespace
\S	Non-whitespace
\w	Letters or numbers (a word character)

\W	Non-word character
----	--------------------

For example, your submit program might return the output “job id j1128 has been submitted” - one regular expression for getting at the job id is “id\s(.+)\shas”. If you would normally remove the job from the queue by typing “job_remove j1128” then you should enter “job_remove JOB_ID_TAG” as DDT's cancel command.

Some queue systems allow you to specify the number of processes, others require you to select the number of nodes and the number of processes per node. DDT caters for both of these but it is important to know whether your template file and queue system expect to be told the number of processes (NUM_PROCS_TAG) or the number of nodes and processes per node (NUM_NODES_TAG and PROCS_PER_NODE_TAG). If these terms seem strange, see 'sample.qtf' for an explanation of DDT's queue template system.

Please note that on some rare platforms an extra environment variable may be needed whilst working with some queue systems: DDT_IGNORE_MPI_OUTPUT may need to be set to 1 prior to starting DDT.

Getting Help

In the event of difficulties - in either installing or using DDT - please consult the appendices to this document or the support and software updates section of our website. This document is also available from within DDT by pressing F1.

Support is also available from the support team – they may be contacted at support@allinea.com – and will be eager to help.

3. Starting DDT

As always, when compiling the program that you wish to debug, you must add the debug flag to your compile command. For the most compilers this is `-g`. It is also advisable to turn off compiler optimisations as these can make debugging appear strange and unpredictable. If your program is already compiled without debug information you will need to remake the files that you are interested in again.

To start DDT simply type one of the following into a shell window:

```
ddt
ddt program_name
ddt program_name arguments
```

Once DDT has started it will display the `Session Control` dialog used for configuring, starting and stopping debug sessions.

NB. You should not attempt to pipe input directly to DDT – for information about how to achieve the effect of sending input to your program, please read the section about program input in this userguide.

Debugging Multi-Process Programs



Fig.6 Session control dialog

If your licence supports multi-process debugging, you will usually see the above dialog.

If your previous DDT session was debugging non-MPI codes, the view will be more compact than the above, and in this case you will be able to choose an MPI implementation and this will restore the full complement of parameter boxes.

In the application box, enter the full pathname to your application. If you specified one on the command-line, this will already be filled in. You may alternatively select an application by clicking on the `...`.

The choice of MPI implementation is critical to correctly starting DDT. Your system will normally use one particular MPI implementation. If you are unsure as to which to pick, try `generic`, consult your system administrator or Alinea. A list of settings for common implementations is provided in an appendix.

Finally you should enter the number of processes that you wish to run. DDT supports over 1000 processes but this is limited by your licence.

If you wish to set more options, such as program and MPI arguments, memory debugging and so on, click the Advanced button. The dialog will expand, and look like this:

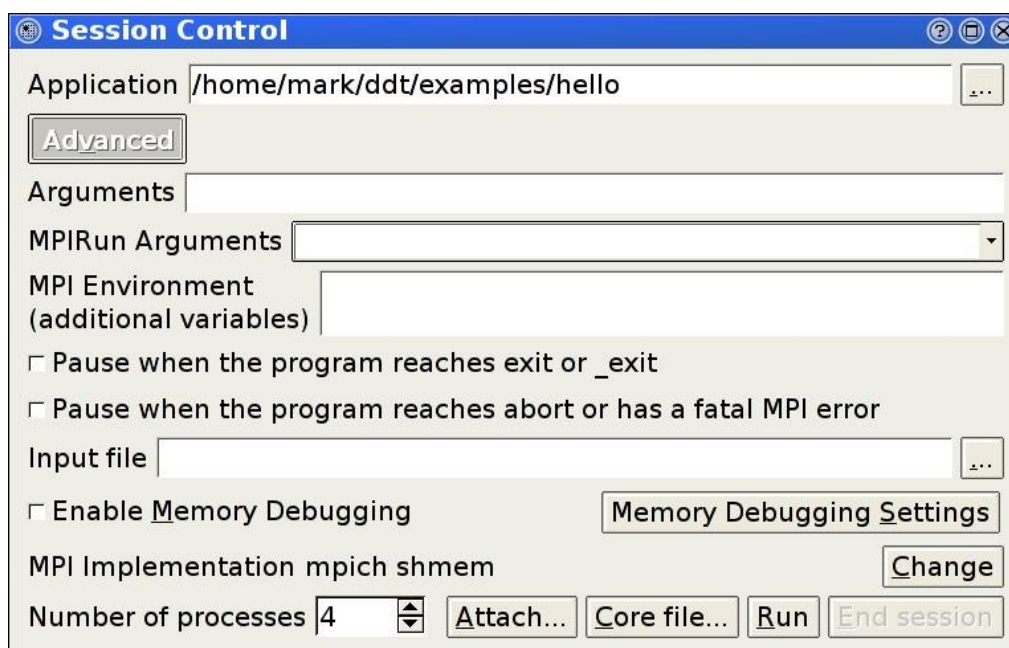


Fig.7 Advanced options

The next box is for arguments. These are the arguments passed to your application, and will be automatically filled if you entered some on the command-line.

The MPIRun arguments box is for arguments that are passed to `mpirun` or your equivalent (such as `scrun` on SCore, `mprun` on Solaris) – usually prior to your executable name in normal mpirun usage. You can place machine file arguments – if necessary – here. For most users this box can be left empty.

Please note that you should **not** enter the “-np” argument as DDT will do this for you.

The MPIRun environment should contain environment variables that should be passed to `mpirun` or its equivalent: some implementations allow you to set extra variables such as MPI_MAX_CLUSTER_SIZE=1 on MPICH. These environment variables may also be passed to your program, depending on which MPI implementation your system uses. Most users will not need to use this box.

If your desired MPI command is not in your PATH, or you wish to use an MPI run command that is not your default one, you can set the environment variable DDTMPIRUN before you start DDT, to run your desired command.

The next two checkboxes allow you to choose whether or not DDT will automatically pause your program when it looks like it is about to finish. If your program reaches one of the functions shown, DDT will ask you whether you want to pause the processes that have reached this point so you can see how they got there, or to let them carry on and (usually) finish. The default setting – do nothing on a normal exit but stop if there is an MPI error or if abort is called – is best for

most users. Otherwise, MPI might terminate your processes on certain errors and you would then be unable to discover what had happened.

The memory debugging options are described in detail in the Memory Debugging section of this document.

Select run to start your program – or submit if working through a queue (see Configuring DDT with Queuing Systems). This will run your program through the debug interface you selected and will allow your MPI implementation to determine which nodes to start which processes on.

When your program starts, DDT tries to work out which MPI world rank each process has. If this fails, you will see the following error message:



Of course you can continue to debug your program! It just means that the number DDT shows on each process will probably not be the MPI rank of the process, which can be confusing! Often there will be a variable in your program that holds the MPI world rank. You can tell DDT to use this variable as the rank for each process – see the “Assigning MPI ranks” section for details.

The End session button is provided to end the current session that you are running. This will close all processes and stop any running code. If any processes remain you may have to clean them up manually using the ``kill`` command or a command provided with your MPI implementation such as ``mpkill`` on Solaris.

Notes on the MPICH Standard Option

For most MPICH-based distributions, the “MPICH Standard” option should be chosen as DDT's MPI implementation. This allows MPICH to start all the processes, and then attaches to them while they're inside MPI_Init. To make this work, DDT must have a way to start programs on cluster nodes. This is usually chosen as part of the “attaching” configuration process.

If attaching is not already set up when you run the job, DDT will try to automatically configure attaching. You may see a small window appear while DDT searches for an appropriate command to use (ssh and rsh are both checked, for example).

Important: If DDT is running in the background (e.g. “`ddt &`”) then this process may get stuck (some ssh versions cause this behaviour when asking for a password). If this happens to you, go to the terminal and use the “`fg`” or similar command to make DDT a foreground process, or run DDT again, without using “`&`”.

If DDT can't find a password-free way to access the cluster nodes then you will not be able to use the MPICH Standard startup option. Instead, You can use “Generic”,

although startup may be slower for large numbers of processes, or you can use the Configuration Wizard to set up attaching - see the "Attaching To Running Programs" section, below, for more information on configuring DDT manually. If you have any problems, email support@allinea.com!

Debugging Single-Process Programs

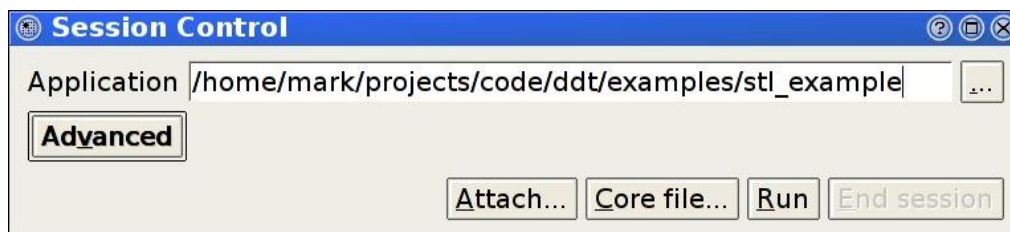


Fig.8 Single-process session control dialog

Users with single-process licences will immediately see the session control dialog that is appropriate for single-process applications.

Users with multi-process licences should set the MPI Implementation to "none" - this will then place DDT into single-process mode and a dialog similar to the above will be shown. DDT can be placed into multi-process mode by changing the MPI implementation to anything except "none".

Select the application – either by typing the file name in, or selecting using the browser by clicking the "..." button. Arguments can be typed into the supplied box.

If you wish, you can also click on the 'Advanced' button to access some of the features listed above (arguments, stop on exit and memory debugging).

Finally press run to start your program.

Note: If you have a program compiled with Intel ifort or gnu g77 you may not see your code and highlight line when DDT starts. This is because those compilers create a pseudo MAIN function, above the top level of your code. To fix this you can either open your Source Code window and add a breakpoint in your code – then run to that breakpoint, or you can use the 'Step into' function to step into your code.

Debugging A Core File

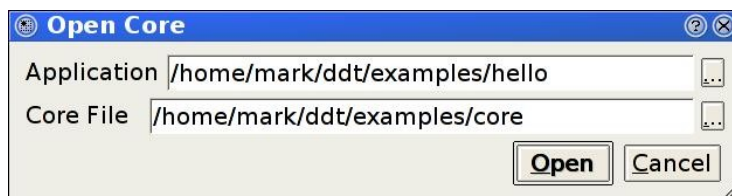


Fig.9 The open core dialog

DDT allows you to debug a single core file generated by your application.

To debug using a core file, click on the 'Core file...' button from the Session Control dialog. This switches to the 'Open Core' dialog, which allows you to select

an application, a core file and a debug interface. Clicking on `Open` will load the core file and start debugging it. While DDT is in this mode, you cannot play, pause or step because there is no process active - you are able to evaluate expressions and browse the variables and stack frames saved in the core file. Choosing to end the session and restart will return DDT to its normal mode of operation.

Attaching To Running Programs

DDT can attach to running processes on any machine you have access to, whether they are from MPI or scalar jobs, even if they have different executables and source pathnames.

The easiest way to set this up is by following the instructions in the Configuration Wizard, described elsewhere in this document. However, if you wish to set up attaching manually, this is how to go about it.

Firstly, either you or your system administrator should provide a script called `remote-exec` to put in your `~/.ddt/` directory. It will be automatically executed like this:

```
remote-exec HOSTNAME APPNAME [ARG1] [ARG2] ...
```

The script must start `APPNAME` on `HOSTNAME` with the arguments `ARG1 ARG2` and without further input (no password prompts). Standard output from `APPNAME` must appear on the standard output of `remote-exec`. On most systems the script can be implemented using `rsh`, as shown here:

```
#!/bin/sh  
rsh $*
```

This particular implementation depends on having an appropriate `.rhosts` file in your home directory as explained in the `rsh` manpage. DDT comes with a default `remote-exec` file, set up as in the above example.

Once the script is set up (we advise testing it at the command-line before using DDT) you must also provide a plain text file listing the nodes you want DDT to look for processes to attach to. If you ran the configuration wizard then you may have already made this file during that process, if not then you now need to create it manually. An example of the contents of this list is:

```
localhost  
comp00  
comp01  
comp02  
comp03
```

This file can be placed anywhere you have read access to, and may be provided by your system administrator. DDT must be given the location of this file - to do this just set the `Attach hosts file` in the options window (`Session -> Options`). Each

host name in this file will be sent to the remote-exec script as the `HOSTNAME` argument when DDT scans for and attaches to processes.

With the script and the node list configured, clicking on the `Attach...` button will show DDT's Attach dialog:

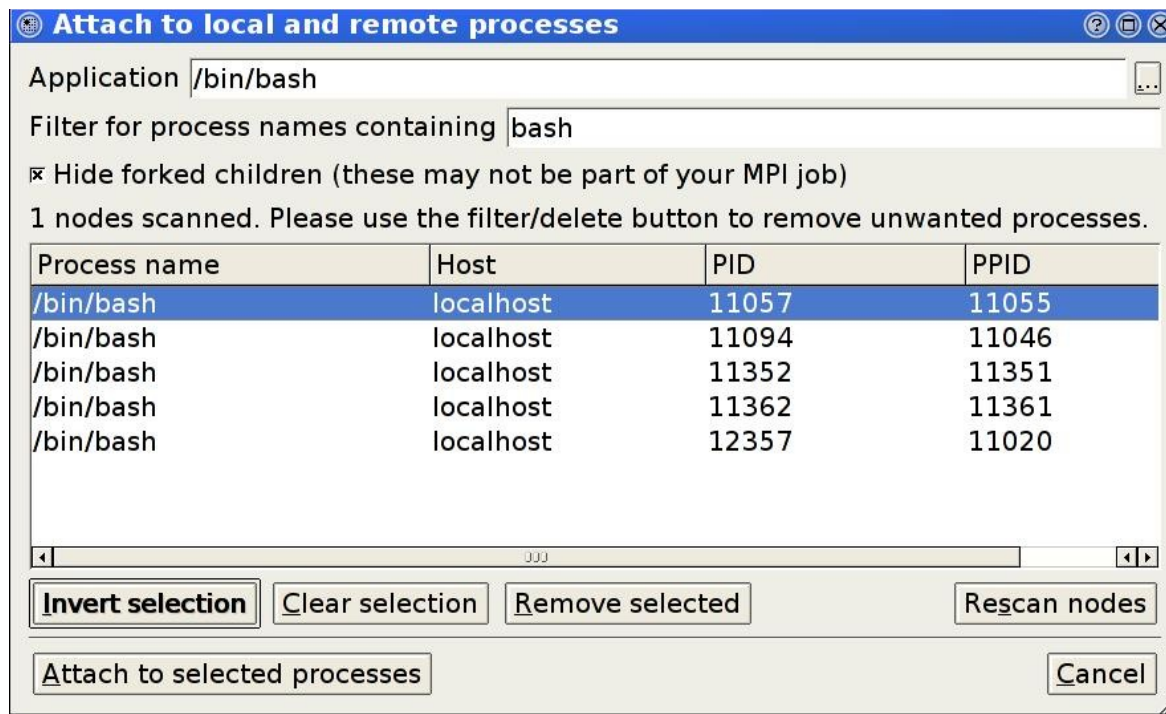


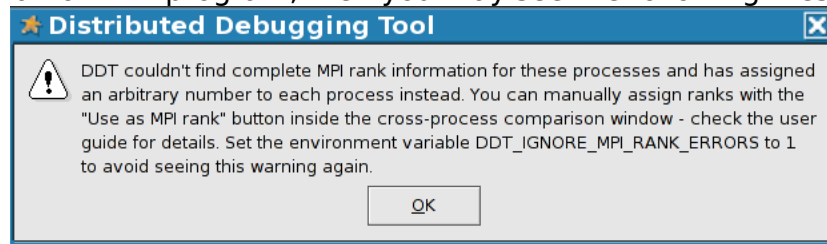
Fig.10 Attach dialog

Initially the list of processes will be blank while DDT scans the nodes, provided in your node list file, for running processes. When all the nodes have been scanned (or have timed out) the dialog will appear as shown in figure 5. If you have not already selected an application executable to debug, you must do so here. Ensure that the list shows all the processes you wish to debug in your job, and no extra/unnecessary processes. You may modify the list by selecting and removing unwanted processes, or alternatively selecting the processes you wish to attach to and clicking on `Attach to selected processes`. If no processes are selected, DDT uses the whole visible list.

Some MPI implementations (such as MPICH) create forked (child) processes that are used for communication, but are not part of your job. To avoid displaying and attaching to these, make sure the `Hide forked children` box is ticked. DDT's definition of a forked child is a child process that shares the parent's name. Other MPI implementations (such as MPICH SHMEM and the Scyld implementation) create your processes as children of each other. If you cannot see all the processes in your job, try clearing this checkbox and selecting specific processes from the list.

Once you click on the `Attach to selected/listed processes` button, DDT will use remote-exec to attach a debugger to each process you selected and will proceed to debug your application as if you had started it with DDT. When you end the debug session, DDT will detach from the processes rather than terminating them – this will allow you to attach again later if you wish.

DDT will examine the processes it attaches to and will try to discover the `MPI_COMM_WORLD` rank of each process. If you have attached to two MPI programs, or a non-MPI program, then you may see the following message:



If there is no rank (for example, if you've attached to a non-MPI program) then you can ignore this message and use DDT as normal. If there is, then you can easily tell DDT what the correct rank for each process via the “Use as MPI Rank” button in the Cross-Process Comparison dialog – see the Assigning MPI Ranks section later in this manual for details.

Note that the `stdin`, `stderr` and `stdout` (standard input, error and output) are not captured by DDT if used in attaching mode. Any input/output will continue to work as it did before DDT attached to the program (e.g. from the terminal or perhaps from a file).

Using DDT Command-Line Arguments

As an alternative to starting DDT and using the Session Control dialog, DDT can instead be instructed to attach to running processes from the command-line.

To do so, you will need to specify the pathname to the application executable as well as a list of hostnames and process identifiers (PIDs).

The list of hostnames and PIDs can be given on the command-line using the *-attach* option:

```
mark@holly:~$ ddt -attach /home/mark/ddt/examples/hello \  
localhost:11057 \  
localhost:11094 \  
localhost:11352 \  
localhost:11362 \  
localhost:12357
```

Another command-line possibility is to specify the list of hostnames and PIDs in a file and use the *-attach-file* option:

```
mark@holly:~$ cat /home/mark/ddt/examples/hello.list
localhost:11057
localhost:11094
localhost:11352
localhost:11362
localhost:12357

mark@holly:~$ ddt -attach-file /home/mark/ddt/examples/hello.list \
/home/mark/ddt/examples/hello
```

In both cases, if just a number is specified for a hostname:PID pair, then “localhost:” is assumed.

These command-line options work for both single- and multi-process attaching.

Starting A Job In A Queue

If DDT has been configured to be integrated with a queue/batch environment, as described on page 13 then you may use DDT to launch your job. In this case, a “submit” button is presented on the session control dialog, instead of the ordinary “run” button. Clicking submit from this session control dialog will display the queue status until your job starts. DDT will execute the display command every second and show you the standard output. If your queue display is graphical or interactive then you cannot use it here.

If your job does not start or you decide not to run it, click on “Cancel Job”. If the regular expression you entered for getting the job id is invalid or if an error is reported then DDT will not be able to remove your job from the queue – it is strongly recommend you check the job has been removed before submitting another as it is possible for a forgotten job to execute on the cluster and either waste resources or interfere with other debug sessions.

Once your job is running, it will connect to DDT and you will be able to debug it.

Using Custom MPI Scripts

On some systems a custom 'mpirun' replacement is used to start jobs, such as 'mpiexec'. DDT will normally use whatever the default for your MPI implementation is, so for mpich it would look for 'mpirun' and not 'mpiexec'. This section explains how to configure DDT to use a custom 'mpirun' command for job startup.

There are typically two ways you might want to start jobs using a custom script, and DDT supports them both. Firstly, you might pass all the arguments on the command-line, like this:

```
mpiexec -n 4 /home/mark/program/chains.exe /tmp/mydata
```

There are several key variables in this line that DDT can fill in for you:

1. The number of processes (4 in the above example)
2. The name of your program (/home/mark/program/chains.exe)
3. One or more arguments passed to your program (/tmp/mydata)

Everything else, like the name of the command and the format of it's own arguments remains constant. To use a command like this in DDT, we adapt the queue submission system described in the previous section. For this 'mpiexec' example, the settings would be as shown here:

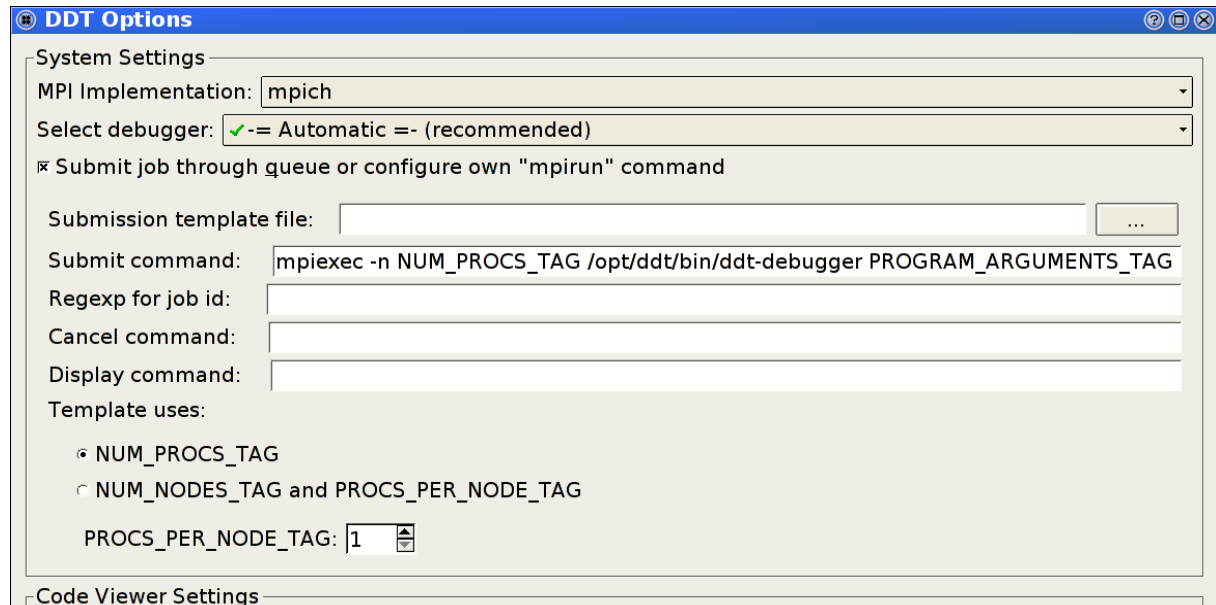


Fig.12 Using custom MPI scripts

As you can see, most of the settings are left blank. Let's look at the differences between the 'Submit command' in DDT and what you would type at the command-line:

1. The number of processes is replaced with NUM_PROCS_TAG
2. The name of the program is replaced by the full pathname to ddt-debugger
3. The program arguments are replaced by PROGRAM_ARGUMENTS_TAG

Note, it is NOT necessary to specify the program name here. DDT takes care of that during its own startup process. The important thing is to make sure your MPI implementation starts ddt-debugger instead of your program, but with the same options.

The second way you might start a job using a custom 'mpirun' replacement is with a settings file:

```
mpiexec -config /home/mark/myapp.nodespec
```

where 'myfile.nodespec' might contains something like this:

```
comp00 comp01 comp02 comp03 : /home/mark/program/chains.exe /tmp/mydata
```

DDT can automatically generate simple config files like this every time you run your program – you just need to specify a template file. For the above example, the template file 'myfile.ddt' would contain the following:

```
comp00 comp01 comp02 comp03 : /opt/ddt/bin/ddt-debugger PROGRAM_ARGUMENTS_TAG
```

This follows the same replacement rules described above and in detail in the section on queues. The options settings for this example might be:

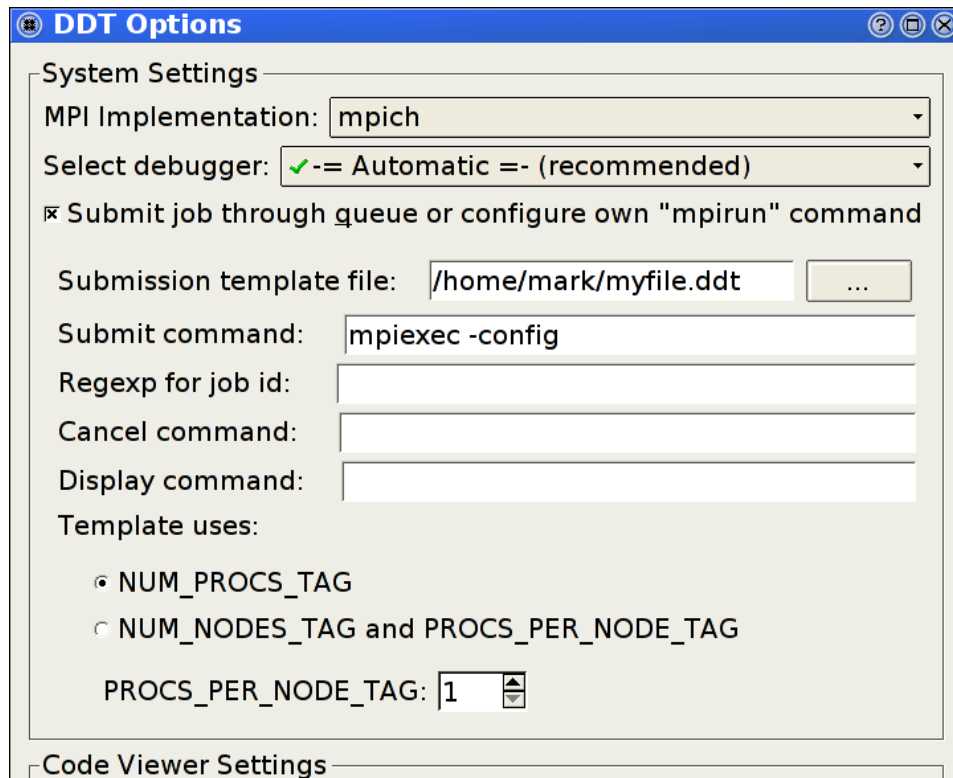


Fig.13 Using substitute MPI commands

Note the 'Submit command' and the 'Submission template file' in particular. DDT will create a new file and append it to the submit command before executing it. So, in this case what would actually be executed might be 'mpiexec -config /tmp/ddt-temp-0112' or similar. Therefore, any argument like '-config' must be last on the line, because DDT will add a filename to the end of the line. Other arguments, if there are any, can come first.

We recommend reading the section on queue submission, as there are many features described there that might be useful to you if your system uses a non-standard startup command. If you do use a non-standard command, please email us at support@allinea.com and let us know about it – you might find the next version supports it out-of-the-box!

Choosing The Right Debugger

DDT uses an enhanced version of GDB with complete F90 and F95 support – this will normally be chosen automatically – on all platforms except for Solaris where DBX is preferred. We recommend that you keep the debugger set to “Automatic” which will choose the correct debugger for your platform. Should you wish to use an alternative debugger, this can be configured from the session options menu but

please note this is not recommended: the other debuggers can vary greatly between minor versions in such a way as to render DDT support impossible.

4. DDT Overview

DDT uses a tabbed-document interface – a method of presenting multiple documents that is familiar from many present day applications. This allows you to have many source files open, and to view one (or two, if the Source Code window is “split”) in the full workspace area.

Each component of DDT (labeled and described in the key) is a dockable window, which may be dragged around by a handle (usually on the top or left-hand edge). Components can also be double-clicked, or dragged outside of DDT, to form a new window.

You can hide or show most of the components using the `View` menu. The screen shot shows the default DDT layout.

Key
(1) Menu bar
(2) Process controls
(3) Process group window (DDT-MP)
(4) Project Navigator window
(5) Code window
(6) Variable window
(7) Evaluate window
(8) Parallel Stack view and Output window
(9) Status bar

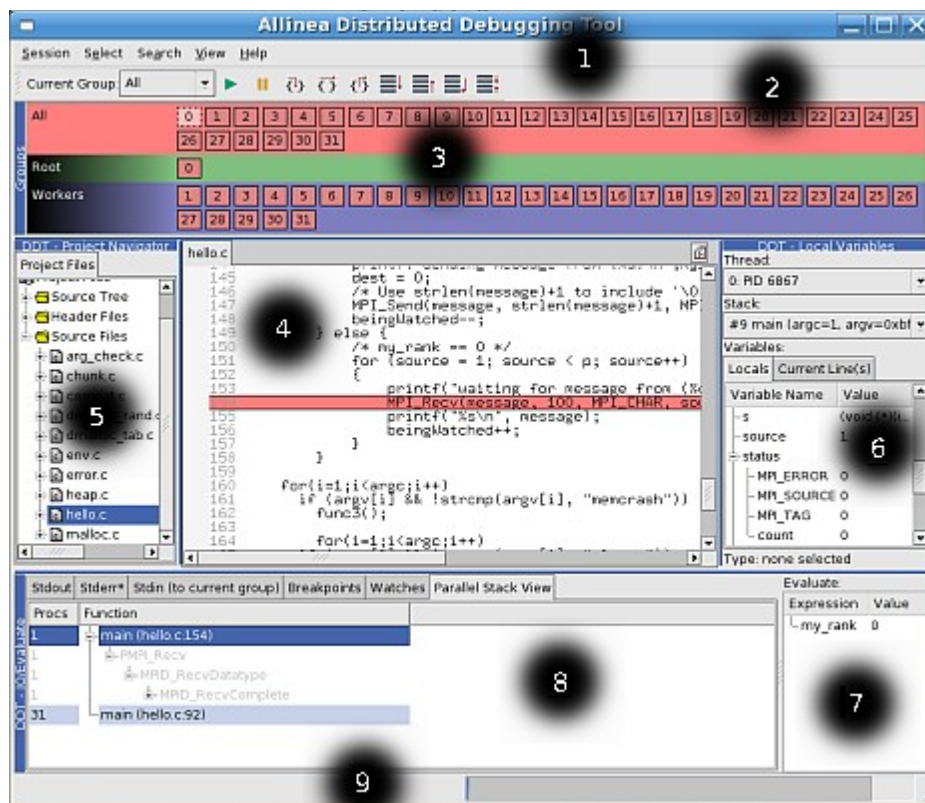


Fig.14 DDT main window

Please note that on some platforms, the default screen size can be insufficient to display the status bar – if this occurs, you should expand the DDT window until DDT is completely visible.

Setting The Font and Tab Sizes

You can change the main font settings, such as those used to display your source files, by choosing “Session” and then clicking “Options”. The font and font size is given as an option in this panel. The tab size (2, 4 or 8 spaces, typically) can be set on this panel also.

DDT inherits many of your font settings from your Desktop Manager (normally KDE or GNOME). To change these, you should use the Desktop Settings configuration tool that is provided by the Desktop Manager.

Saving And Loading Sessions

Most of the user-modified parameters and windows are saved by right-clicking and selecting a save option in the corresponding window.

However, DDT also has the ability to load and save all these options concurrently to minimize the inconvenience in restarting sessions. Saving the session stores such things as process groups, the contents of the Evaluate window and more. This ability makes it easy to debug code with the same parameters set time and time again.

To save a session simply use the “Save Session” option from the “Session” menu. Enter a filename (or select an existing file) for the save file and click OK. To load a session again simply choose the “Load Session” option from the “Session” menu, choose the correct file and click OK.

Source Code

When DDT begins a session, source code is automatically found from the information compiled in the executable.

Source and header files found in the executable are reconciled with the files present on the front-end server, and displayed in a simple tree view within the Project Files tab of the Project Navigator window. Source files can be loaded for viewing by clicking on the filename.

Whenever a selected process is stopped, the Source Code browser will automatically leap to the correct file and line, if the source is available.

Finding Lost Source Files

On some platforms, not all source files are found automatically. This can also occur, for example, if the executable or source files have been moved since compilation. Extra directories to search for source files can be added by right-clicking whilst in the Project Files tab, and selecting “Add/view source directory(s)”.

It is also possible to add an individual file – if, for example, this file has moved since compilation or is on a different (but visible) filesystem – by right-clicking in the Project Files tab and selecting the “Add file” option.

Any directories or files you have added are saved and restored when you use the “Save Session” and “Load Session” commands inside the Session menu. If DDT doesn't find the sources for your project, you might find these commands save you a lot of unnecessary clicking.

Dynamic Libraries

If a library is loaded dynamically, its source file may not be found at the time the program starts. The source can be added by right-clicking whilst in the Project Files tab, and selecting “Scan for more files”.

Finding Code Or Variables

The `Find` and `Find In Files` dialogs are found from the `Search` menu. The `Find` dialog will find occurrences of an expression in the currently visible source file. The `Find In Files` dialog searches all source and header files associated with your program and lists the matches in a result box. Click on a match to display the file in the main Source Code window and highlight the matching line; this can be of particular use for setting a breakpoint at a function. Note that both searches are regular expression based. The syntax of the regular expression is identical to that described in the batch system (queue) configuration in the preceding chapter.

Jump To Line / Jump To Function

DDT has a jump to line function which enables the user to go directly to a line of code. This is found in the `Search` menu. A dialog will appear in the center of your screen. Enter the line number you wish to see and click OK. This will take you to the correct line providing that you entered a line that exists. You can use the hotkey CTRL-G to access this function quickly.

DDT also allows you to jump directly to the implementation of a function. In the `Project Files` tab of the Project Navigator window on the left side of the main screen you should see small `+` symbols next to each file:



Fig.15 Function listing

Clicking on a the `+` will display a list of the functions in that file. Clicking on any function will display it in the Source Code viewer.

If your system has the program `etags` installed in your PATH, DDT will use this to provide faster and more accurate parsing of C/C++ files. If not then a default algorithm will be used that is less effective and, for example, pays no attention to preprocessor definitions. Regardless of which method is used, some language constructs (particularly templated functions) may not be shown in this view.

Editing Source Code

If, prior to starting DDT, you set the environment variable “DDT_EDITOR” to be the name of an editor/script that is an X application then on right-clicking in the Source Code window, DDT will offer to launch your editor and bring your code up at the line selected by the mouse. For example:

```
export DDT_EDITOR=emacs  
export DDT_EDITOR=xterm -e vi
```

5. Controlling Program Execution

Whether debugging a multi-process or a single process code, the mechanisms for controlling program execution are very similar.

In multi-process mode, most of the features described in this section are applied using process groups, which we describe now. For single process mode, the commands and behaviors are identical, but apply to only a single process – freeing the user from concerns about process groups.

Process Control And Process Groups



Fig.16 The process group viewer

MPI programs are designed to run as more than one process and can span many machines. DDT allows you to group these processes so that actions can be performed on more than one process at a time. The status of processes can be seen at a glance by looking at the process group viewer. The process group viewer is (by default) at the top of the screen with multi-coloured rows. Each row relates to a group of processes and operations can be performed on the currently highlighted group (e.g. playing, pausing and stepping) by clicking on the toolbar buttons. Switch between groups by clicking on them or their processes - the highlighted group is indicated by a lighter shade.

Each process is represented by a square containing its MPI rank (0 through n-1). The squares are colour-coded; red for a paused/stopped process and green for a running process. Any selected processes are highlighted with a lighter shade of their colour and the current process also has a dashed border. When a single process is selected the local variables are displayed in the variable viewer and displayed expressions are evaluated. You can make the code viewer jump to the file and line for the current stack frame (if available) by double-clicking on a process.

Groups can be created, deleted, or modified by the user at any time, with the exception of the `All` group, which cannot be modified. To copy processes from one group to another, simply click and drag the processes. To delete a process, press the delete key. When modifying groups it is useful to select more than one process by holding down one or more of the following:

Key	Description
Control	Click to add/remove process from selection
Shift	Click to select a range of processes
Alt	Click to select an area of processes

Note: Some window managers (such as KDE) use Alt and drag to move a window - you must disable this feature in your window manager if you wish to use the DDT's box select.

Groups are added and deleted from a context-sensitive menu that appears when you right-click on the process group widget. This menu can also be used to rename groups, delete individual processes from a group and jump to the current position of a process in the code viewer. You can load and save the current groups to a file, but be careful when using this feature as the number of processes might have changed since the file was saved. If you are on a process group already when you right-click, the further option of `add complement with` will be enabled. This allows a new group to be created using the set difference of the currently selected group and the one you choose from the submenu.

Hint: To communicate with a single process, create a new group and drag that process into it.

Hotkeys

DDT comes with a pre-defined set of hotkeys to enable easy control of your debugging. All the features you see on the toolbar and several of the more popular functions from the menus have hotkeys assigned to them. Using the hotkeys will speed up day to day use of DDT and it is a good idea to try to memorize these.

Key	Function
F9	Play
F10	Pause
F5	Step into
F8	Step over
F6	Step out
CTRL-D	Down stack frame
CTRL-U	Up stack frame
CTRL-B	Bottom stack frame
CTRL-A	Align stack frames with current
CTRL-G	Goto
CTRL-F	Find

Starting, Stopping And Restarting A Program

The Session Control dialog can be accessed at almost any time while DDT is running. If a program is running you can end it and run it again or run another program. When DDT's startup process is complete your program should automatically stop either at the main function for non-MPI codes, or at the MPI_Init function for MPI.

When a job has run to the end DDT will show a dialog box asking if you wish to restart the job. If you select yes then DDT will kill any remaining processes and clear up the temporary files and then restart the session from scratch with the same program settings.

When ending a job, DDT will attempt to ensure that all the processes are shutdown and clear up any temporary files. If this fails for any reason you may have to manually kill your processes using ``kill``, or a method provided by your MPI implementation such as ``lamclean`` for LAM/MPI.

Stepping Through A Program

To start the program running click ``Play/Continue`` and to stop it at any time click ``Pause``. For multi-process DDT these start/stop all the processes in the current group (see Process Control and Process Groups).

Like many other debuggers there are three different types of step available. The first is ``Step into`` that will move to the next line of source code unless there is a function call in which case it will step to the first line of that function. The second is ``Step over`` that moves to the next line of source code in the bottom stack frame. Finally, ``Step out`` will execute the rest of the function and then stop on the next line in the stack frame above.

When using step out be careful not to try and step out of the main function, as doing this will end your program.

Setting Breakpoints

First locate the position in your code that you want to place a breakpoint at. If you have a lot of source code and wish to search for a particular function you can use the ``Find`/`Find In Files`` dialog. Clicking the right mouse button in the Source Code window displays a menu showing several options, including one to add or remove a breakpoint. In multi-process mode this will set the breakpoint for every member of the current group.

Every breakpoint is listed under the breakpoints tab towards the bottom of DDT's window.

If you add a breakpoint at a location where there is no executable code, DDT will highlight the line you selected as having a breakpoint. However when hitting the breakpoint, DDT will stop at the next executable line of code.

You may wish to add a breakpoint in a function for which you do not have any source code: for example in `malloc`, `exit`, or `printf` from the standard system libraries. If this function is used inside your code, find the place where it is used position the mouse over the function name and right-click. The option of adding a breakpoint will be offered.

Conditional Breakpoints

Stdout	Stderr	Breakpoints	Watches	Stdin (to current group)		
Group	Filename	Line	Function	Condition	Full path	
<input checked="" type="checkbox"/>	All	hello.c	91		/home/david/projects/code/ddt/examples	
<input checked="" type="checkbox"/>	All		0 malloc			
<input checked="" type="checkbox"/>	All	hello.c	105	x > 0	/home/david/projects/code/ddt/examples	
<input checked="" type="checkbox"/>	Workers	hello.c	110		/home/david/projects/code/ddt/examples	

Fig.17 The breakpoints table

Select the breakpoints tab to view all the breakpoints in your program. You may add a condition to any of them by clicking on the condition cell in the breakpoint table and entering an expression that evaluates to true or false. Each time a process (in the group the breakpoint is set for) passes this breakpoint it will evaluate the condition and break only if it returns true (typically any non-zero value). You can drag an expression from the Evaluate window into the condition cell for the breakpoint and this will be set as the condition automatically.

Stdout	Stderr	Stdin (to current group)	Breakpoints	Watches		
Group	Filename	Line	Function	Condition	Full path	
<input checked="" type="checkbox"/>	All	trisol.f90	32	n_procs .LT. 0	/home/steff/trisol/trisol.f90	
<input checked="" type="checkbox"/>	All	trisol.f90	38	n_procs .GT. 0	/home/steff/trisol/trisol.f90	
<input checked="" type="checkbox"/>	All	trisol.f90	41	(n_procs .GT. 0) .AND. (me.LT.0)	/home/steff/trisol/trisol.f90	
<input checked="" type="checkbox"/>	All	trisol.f90	47	(n_procs.GT.0) .AND. (me.GE.0)	/home/steff/trisol/trisol.f90	

Conditional Breakpoints in Fortran

The expression should be in the same language as your program. Also, please the condition evaluator can be too pedantic with Fortran conditions, and to ensure the correct interpretation of compound boolean operations, it is advisable to bracket your expressions amply.

Suspending Breakpoints

A breakpoint can be temporarily deactivated and reactivated by checking/unchecking the activated column in the breakpoints panel.

Deleting A Breakpoint

Breakpoints are deleted by either right-clicking on the breakpoint in the breakpoints panel, or by right-clicking at the file/line of the breakpoint whilst in the correct process group and right-clicking and selecting delete breakpoint.

Loading And Saving Breakpoints

To load or save the breakpoints in a session right-click in the breakpoint panel and select the load/save option. Breakpoints will also be loaded and saved as part of the load/save session.

Synchronizing Processes

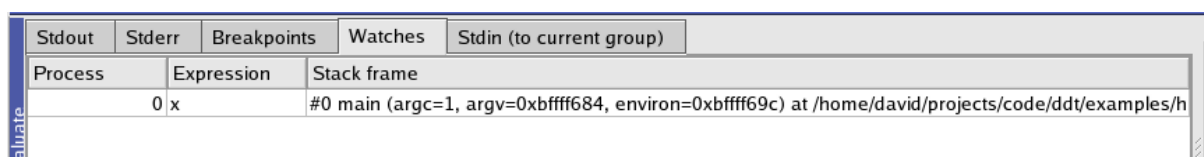
If the processes in a process group are stopped at different points in the code and you wish to re-synchronize them to a particular line of code this can be done by right-clicking on the line at which you wish to synchronize them to and selecting “run to here”. This effectively sets all the processes in the selected group running and puts a break point at the line at which you choose to synchronize the processes at, ignoring any breakpoints that the processes may encounter before they have synchronized at the specified line.

If you choose to synchronize your code at a point where all processes do not reach then the processes that cannot get to this point will run to the end.

Note: Though this ignores breakpoints while synchronizing the groups it will not actually remove the breakpoints.

Note: If a process is already at the line which you choose to synchronize at, the process will still be set to run. Be sure that your process will revisit the line, or alternatively synchronize to the line immediately after the current line.

Setting A Watch



Stdout Stderr Breakpoints Watches Stdin (to current group)		
Process	Expression	Stack frame
0 x		#0 main (argc=1, argv=0xbffff684, environ=0xbffff69c) at /home/david/projects/code/ddt/examples/h

Fig.18 The watches table

A watchpoint is a type of breakpoint that monitors a variable's value and causes a break once the value is changed. Unlike breakpoints, it is not set on a line in the Source Code window. Instead you must drag a variable from either the Variables window or the Evaluate window into the Watches table. It is not generally useful to watch a variable that is allocated on the stack rather than the heap, and some debug interfaces (such as GDB) will remove a watchpoint when its variable goes out of scope. Variables on the heap do not go out of scope.

For multi-process debugging, watches can only be set on a single process and not a whole group.

Examining The Stack Frame

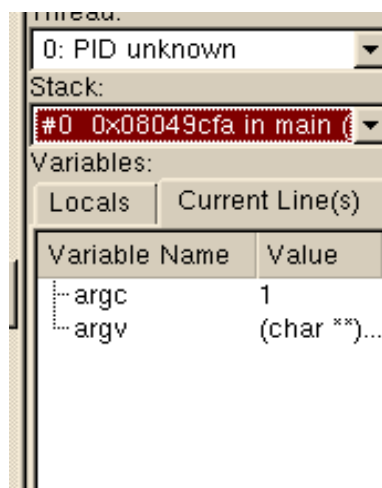


Fig.19 Selecting a stack frame

The stack frame (the current position in the stack) is displayed and changed using a drop-down list in the Variables window. When you select a stack frame DDT will jump to that position in the code (if it is available) and will display the local variables for that frame. The toolbar can also be used to step up or down the stack, or jump straight to the bottom-most frame.

Align Stacks

The align stacks button, or CTRL-A hotkey, sets the stack of the current thread on every process in a group to the same level – where possible – as the current process.

This feature is particularly useful where processes are interrupted – by the pause button – and are at different stages of computation. This enables tools such as the Cross-Process Comparison window to compare equivalent local variables, and also simplifies casual browsing of values.

Examining Threads

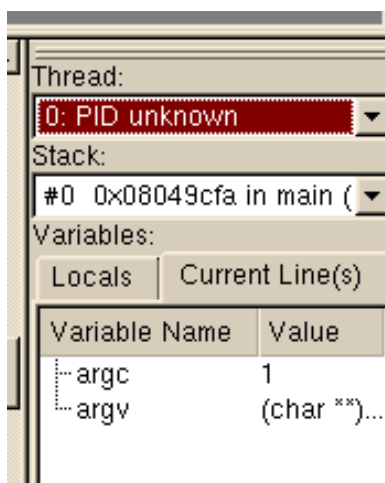


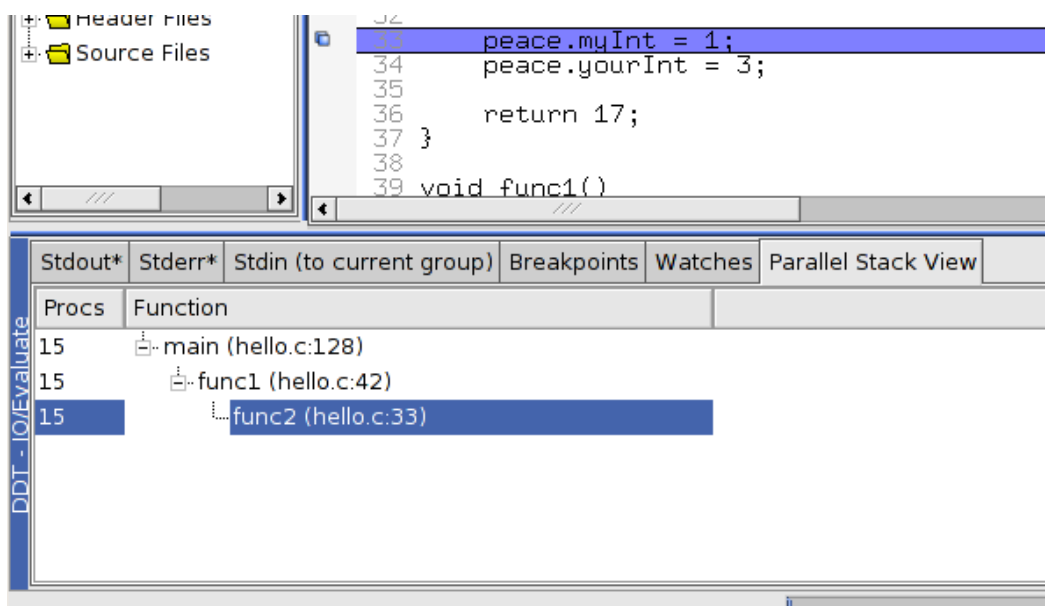
Fig.20 Selecting a thread

You can select a thread from the drop-down list in the Variables window. Changing the thread will update the stack frame and local variables. Multi-process DDT users should note that many MPI implementations are not thread safe so you must be very careful when using threads. DDT supports both OpenMP and native threading libraries such as pthreads. If your program uses the pthreads library (either natively or via OpenMP) you may see an extra handler thread - this is not part of your program but is used by the operating system to manage multiple threads and may be present even when your program is only using one thread.

“Where are my processes?” - Viewing Stacks in Parallel

Overview

To find out where your program is, in one single view, look no further than the Parallel Stack View. It's found in the bottom area of DDT's GUI, tabbed alongside the stdout, stderr, stdin, breakpoints and watches:



Do you want to know where a group's processes are? Click on the group and look at the Parallel Stack View – it shows a tree of functions, merged from every process in the group (by default). If there's only one branch in this tree – one list of functions – then all your processes are at the same place. If there are several different branches, then your group has split up and is in different parts of the code! Click on any branch to see its location in the Source Code viewer, or hover your mouse over it and a little popup will list the processes at that location. Right-click on any function in the list and select “New group” to automatically gather the processes at that function together in a new group, labelled by the function's own name.

The best way to learn about the Parallel Stack View is to simply use it to explore your program. Click on it and see what happens. Create groups with it, and watch what happens to it as you step processes through your code. The Parallel Stack View's ability to display and select large numbers of processes based on their location in your code is invaluable when dealing with moderate to large numbers of processes.

The Parallel Stack View in Detail

The Parallel Stack View takes over much of the work of the Stack Frame display, but instead of just showing the current process, this view combines the call trees (commonly called “stacks”) from many processes and displays them together. The call tree of a process is the list of functions (strictly speaking “frames” or locations within a function) that lead to the current position in the source code. For example, if `main()` calls `read_input()`, and `read_input()` calls `open_file()`, and you stop the program inside `open_file()`, then the call tree will look like this:

```
main()
  read_input()
    open_file()
```

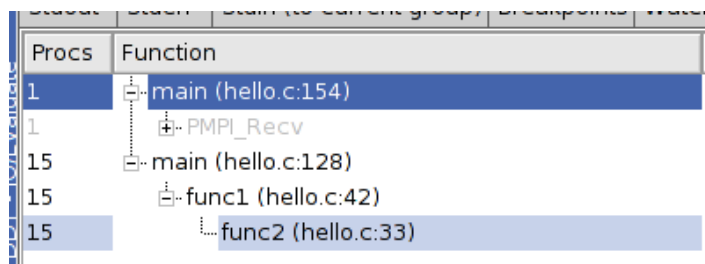
If a function was compiled with debug information (usually “-g”) then DDT adds extra information, telling you the exact source file and line number that your code is on. Any functions without debug information are greyed-out and are not shown by default. Functions without debug information are typically library calls or memory allocation subroutines and are not generally of interest. To see the entire list of functions, right-click on one and choose “Show children” from the pop-up menu.

You can click on any function to select it as the “current” function in DDT. If it was compiled with debug information, then DDT will also display its source code in the main window, and its local variables and so on in the other windows.

One of the most important features of the Parallel Stack View is its ability to show the position of many processes at once. Right-click on the view to toggle between:

1. Viewing all the processes in your program at once
2. Viewing all the processes in the current group at once (default)
3. Viewing only the current process

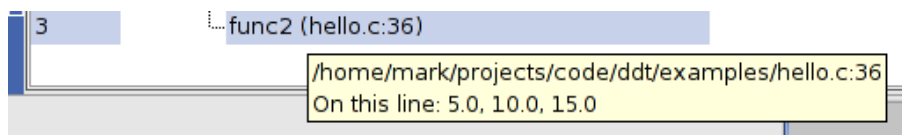
The function that DDT is currently displaying and using for the variable views is highlighted in dark blue. Clicking on another function in the Parallel Stack View will select another frame for the source code and variable views. It will also update the Stack Frame display, since these two controls are complementary. If the processes are at several different locations, then only the current process' location will be shown in dark blue. The other processes' locations will be shown in a light blue:



In the example above, the program's processes are at two different locations. One process is calling `PMPI_Recv` from the “main” function, at `hello.c` line 154, and the other 15 processes are all inside a function called “func2”, specifically at line 33 of `hello.c`. All these processes reached this line of “func2” in the same way – first “main” called “func1” on line 128 of `hello.c`, and then “func1” called “func2” on line 42 of `hello.c`. Clicking on any of these functions will show the exact line of source code, and will display the state of the variables in that function when the call was made.

There are two optional columns in the Parallel Stack View. The first, "Procs" shows the number of processes at each location. The second, "Threads", shows the number of threads at each location. By default, only the number of processes is shown. Right-click to turn these columns on and off. Note that in a normal, single-threaded MPI application, each process has one thread and these two columns will show identical information.

Hovering the mouse over any function in the Parallel Stack View displays the full path of the filename, and a list of the process ranks that are at that location in the code:



DDT is at its most intuitive when each process group is a collection of processes doing a similar task. The Parallel Stack View is invaluable in creating and managing these groups. Simply right-click on any function in the combined call tree and choose the "New group" option. This will create a new process group that contains only the processes sharing that location in code. By default DDT uses the name of the function for the group, or the name of the function with the file and line number if it's necessary to distinguish the group further.

Browsing Source Code

Source code will be automatically displayed – when a process is stopped, when you select a process or change position in the stack.

In addition to colour highlighting source lines to display which groups processes are at particular lines of the present source, it is also possible to hover the mouse over a line and get a summary tooltip. This tooltip will state whether the line is presently a breakpoint and whether any processes are stopped at the line.

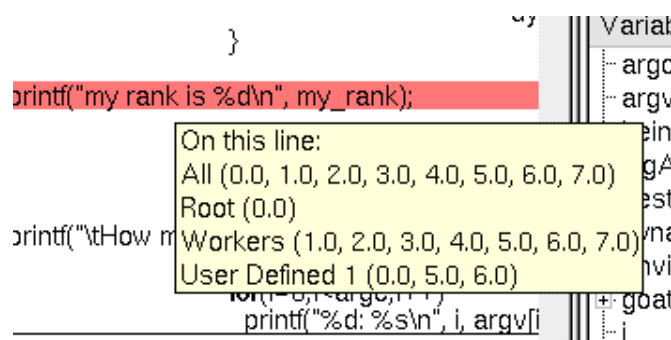


Fig.21 Process list tooltip

Right-clicking above a term in the Source Code viewer will cause DDT to probe that term and establish whether there is a matching variable or function.

In the case of a variable, the type and value are displayed, along with options to view the variable in the Cross-Process Comparison window (CPC) or the Multi-dimensional array viewer (MDA), or to drop the variable into the Evaluate window – each of which are described in the next chapter.

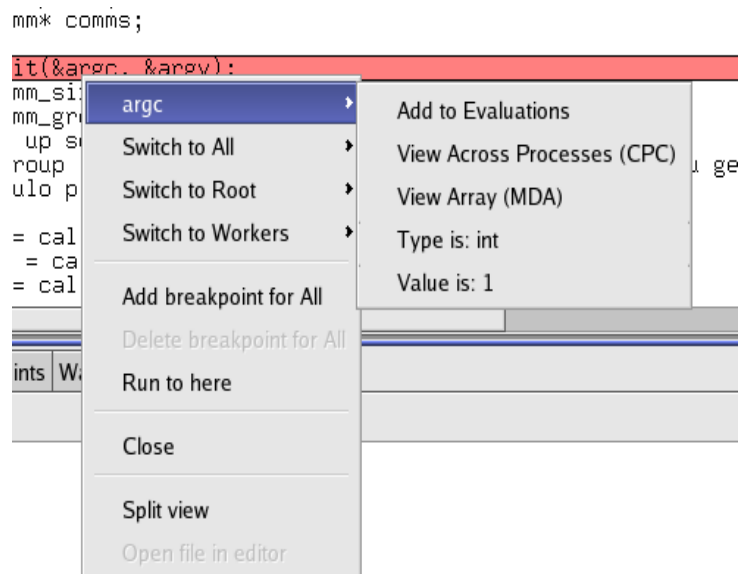


Fig.22 Right-click menu – variable options

In the case of a function, it is also possible to add a breakpoint in the function, or to the source code of the function when available.

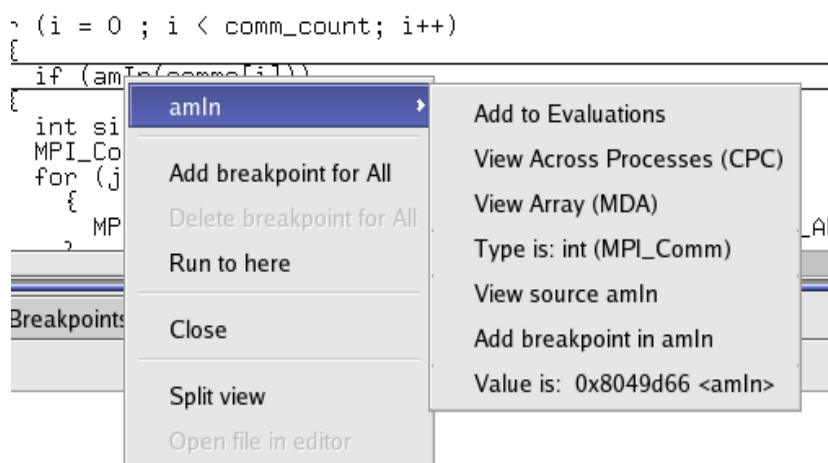


Fig.23 Right-click menu – function options

Simultaneously Viewing Multiple Files

DDT presents a tabbed pane view of source files, but occasionally it may be useful to view two files simultaneously – whilst tracking two different processes for example.

Inside the code viewing panel, right-click to split the view. This will bring a second tabbed pane which can be viewed beneath the first one. When viewing further files, the currently “active” panel will display the file. Click on one of the views to make it active.

The split view can be reset to a single view by right-clicking in the code panel and deselecting the split view option.

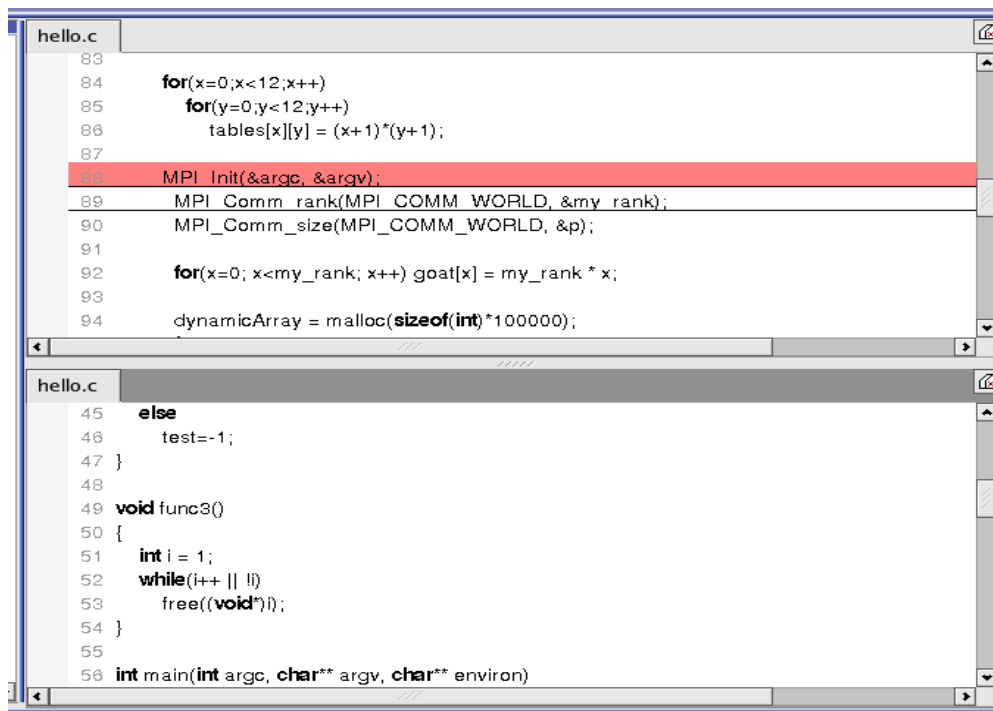


Fig.24 Horizontal alignment of multiple source files

Signal Handling

DDT will stop a process if it encounters one of the standard signals such as

- SIGSEGV – Segmentation fault
 - The process has attempted to access memory that is not valid for that process. Often this will be caused by reading beyond the bounds of an array, or from a pointer that has not been allocated yet. The DDT Memory Debugging feature may help to resolve this problem.
- SIGFPE – Floating Point Exception
 - This is raised typically for integer division by zero, or dividing the most negative number by -1. Whether or not this occurs is Operating System dependent, and not part of the POSIX standard. Linux platforms will raise this.
 - Note that floating point division by zero will not necessarily cause this exception to be raised, behaviour is compiler dependent. The special value “Inf” or “-Inf” may be generated for the data, and the process would not be stopped.
- SIGPIPE - Broken Pipe
 - A broken pipe has been detected whilst writing.
- SIGILL – Illegal Instruction

Note that SIGUSR1, SIGUSR2, SIG63 and SIG64 are passed directly through to the user process without being intercepted by DDT.

6. Variables And Data

The Variables window contains two tabs that provide different ways to list your variables. The `Locals` tab contains all the variables for the current stack frame, while the `Current Line(s)` tab displays all the variables referenced on the currently selected lines.

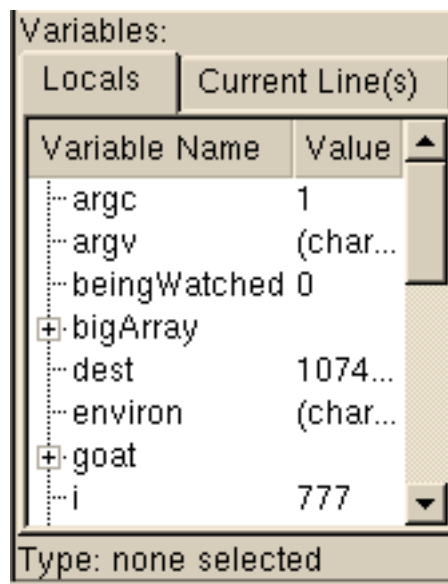


Fig.25 Displaying variables

Current Line

You can select a single line by clicking on it in the code viewer - or multiple lines by clicking and dragging. The variables are displayed in a tree view so that user-defined classes or structures can be expanded to view the variables contained within them. You can drag a variable from this window into the `Evaluate` window; it will then be evaluated in whichever stack frame, thread or process you select.

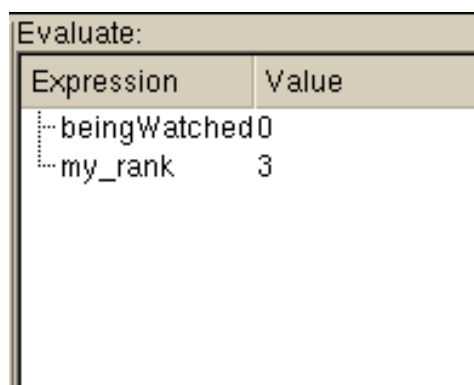
Local Variables

The locals tab contains local variables for the current process's currently active thread and stack frame.

For Fortran codes the amount of data reported as local can be substantial – as this can include many global or common block arrays. Should this prove problematic, it is best to conceal this tab underneath the `Current Line(s)` tab as this will not then update after every step.

It is worth noting that variables defined within common blocks may not appear in the local variables tab with some compilers, this is because they are considered to be global variables when defined in a common memory space.

Arbitrary Expressions And Global Variables



Evaluate:	
Expression	Value
beingWatched	0
my_rank	3

Fig.26 Evaluating expressions

Since the global variables and arbitrary expressions do not get displayed with the local variables, you may wish to use the `Current Line(s)` tab in the Variables window and click on the line in the Source Code window containing a reference to the global variable.

Alternatively, the Evaluate panel can be used to view the value of any arbitrary expression. Right-click on the Evaluate window, click on `Add Expression`, and type in the expression required in the current source file language. This value of the expression will be displayed for the current process and stack/thread, and is updated after every step.

Note: at the time of writing DDT does not apply the usual rules of precedence to logical Fortran expressions, such as "x .ge. 32 .and. x .le. 45". For now, please bracket such expressions thoroughly: "(x .ge. 32) .and. (x .le. 45)". Sorry for the inconvenience!

Help With Fortran Modules

An executable containing Fortran modules presents a special set of problems for developers:

- if there are many modules, each of which contains many procedures and variables (each of which can have the same name as something else in a separate Fortran module), keeping track of which name refers to which entity can become difficult
- when the `Locals` or `Current Line(s)` tabs (within the Variables window) display one of these variables, to which Fortran module does the variable belong?
- how do you refer to a particular module variable in the Evaluate window?
- how do you quickly jump to the source code for a particular Fortran module procedure?

To help with this, DDT provides a `Fortran Modules` tab in the Project Navigator window.

When DDT begins a session, Fortran module membership is automatically found from the information compiled into the executable.

A list of Fortran modules found is displayed in a simple tree view within the `Fortran Modules` tab of the Project Navigator window.

Each of these modules can be “expanded” (by clicking on the '+' symbol to the left of the module name) to display the list of member procedures, member variables and the current values of those member variables.

Clicking on one of the displayed procedure names will cause the Source Code viewer to jump to that procedure's location in the source code. In addition, the return-type of the procedure will be displayed at the bottom of the Fortran Modules tab – Fortran subroutines will have a return-type of 'VOID ()'.

Similarly, clicking on one of the displayed variable names will cause the type of that variable to be displayed at the bottom of the Fortran Modules tab.

A module variable can be “dragged” across and “dropped” into the Evaluate window. Here, all of the usual Evaluate window functionality applies to the module variable (see the section on “Variables and Data”, below). To help with variable identification in the Evaluate window, module variable names are prefixed with the Fortran module name and a '::'.

Right-clicking within the Fortran Modules tab will bring up a context menu. For variables, choices on this menu will include sending the variable to the Evaluate window, the Multi-Dimensional Array viewer and the Cross-Process Comparison viewer.

Some caveats apply to the information displayed within the Fortran Modules tab:

1. If the underlying debugger does **not** support the retrieval and manipulation of Fortran module data, the Fortran Modules tab will not be displayed.
2. If the underlying debugger **does** support the retrieval and manipulation of Fortran module data, but the executable does not contain any Fortran modules (or, the Fortran modules debug data is not in a format understood by DDT), the Fortran Modules tab will still be displayed. However, the list of Fortran modules itself will be empty. For example, the GNU Fortran compilers (g77, gfortran) do **not** provide Fortran modules debug data in a format understood by DDT.
3. The display and update of the contents of each of the Fortran modules comes at a small runtime cost to the performance of DDT (**not** to the cost of the executable's performance!). Expanding and displaying **only** the Fortran modules required, and contracting those Fortran modules **no longer required**, will have a direct impact on the performance of your DDT debugging session.

One limitation of the Fortran Modules tab is that the modules debug data compiled into the executable does not include any indication of the module USE hierarchy (e.g. if module “A” USEs module “B”, the “inherited” members of module “B” are not shown under the data displayed for module “A”). Consequently, the Fortran Modules tab shows the module USE hierarchy in a flattened form, one level deep.

Viewing Array Data

Fortran users may find that it is not possible to view the upper bounds of an array. This is due to a lack of information from the compiler. In these circumstances DDT will display the array with a size of 0. It is still possible to view the contents of the array however using the Evaluate window to view array(1) array(2) etc. as separate entries.

Changing Data Values

In the Evaluate window, the value of an expression may be set by right-clicking and selecting "edit value". This will change the value of the expression in the currently selected process.

Note: This depends on the variable existing in the current stack frame and thread.

Examining Pointers

Pointer contents cannot normally be examined in the Variables window but after dragging them into the Evaluate window you can right-click and select any of the following new options: View as vector, Reference, or Dereference.

If a structure contains another pointer you must drag this pointer onto its own line in the Evaluate window before you can start Referencing/Dereferencing it.

Multi-Dimensional Arrays in the Variable View

When viewing a multi-dimensional array in either the `Locals`, `Current Line(s)` or `Evaluate` windows it is possible to expand the array to view the contents of each cell. In C/C++ the array will expand from left to right (x,y,z will be seen with the x column first, then under each x cell a y column etc.) whereas in Fortran the opposite will be seen with arrays being displayed from right to left as you read it (so x,y,z would have z as the first column with y under each z cell etc.)

C Example: type of twodee is int[5][3]

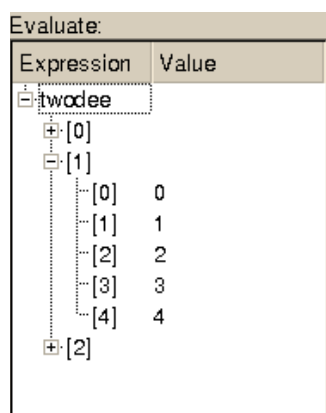


Fig.27 2D array in C

Fortran Example: type of twodee is integer(3,5)

Expression	Value
twodee	
[1]	
[2]	
[1]	2
[2]	4
[3]	6
[3]	
[4]	
[5]	

Fig.28 2D array in Fortran

Examining Multi-Dimensional Arrays

Large multi-dimensional arrays are not easy to view in a tree structure, so DDT provides a Multi-Dimensional Array viewer (the MDA). This allows you to view the results of evaluating expressions involving up to two variables (i and j) in a table, and also to visualize the results in 3D (see the next section 'Visualizing Data').

To bring up the Multi-Dimensional Array viewer, right-click on a variable inside the `Source Code`, `Locals`, `Current Line(s)` or `Evaluate` windows - and then choose the "View Array (MDA)" option. You can also bring up the MDA directly by selecting "View" from the menu bar and picking "Multi-Dimensional Array Viewer".

At the top of the MDA are controls allowing you to change the expression, and the range over which it is to be evaluated. You can view any two-dimensional slice of data by entering an expression involving i and j (e.g. `A(i+j,j,1)` in Fortran, or `myArray[1][j][i+j]` in C/C++).

If you brought up the MDA by right clicking on a variable, it will automatically set the expression and ranges based on the type of the variable or array in question, but these default values can be modified.

Hint: DDT will retrieve data **much faster** if the array section is contiguous (all one block in memory) or if it can be broken down into several contiguous blocks (such as rows of the array). When DDT automatically fills in the values for a variable, it always chooses such an arrangement. Changing the array expression will force DDT to evaluate each element individually. For maximum performance, always use the "Right-click on variable -> View in MDA" method to view your arrays, and only change the ranges of i and j, and not their positions in the array. If you really want to view a non-contiguous slice then we strongly recommend starting with a small subset of the array.

Clicking `Evaluate` will fill the table with all evaluations of i and j. This data can be exported to a csv (comma-separated values) file which can be plotted or analyzed in your favorite spreadsheet. If `Auto-update` is selected, the table will be re-evaluated automatically as you step through the code. The `Statistics` tab displays information which may be of interest, such as the range of the values in the table, and the number of special numerical values, such as nan or inf. Once

the data is loaded into DDT's table you can visualize the data as a surface in 3-D space (see next section).

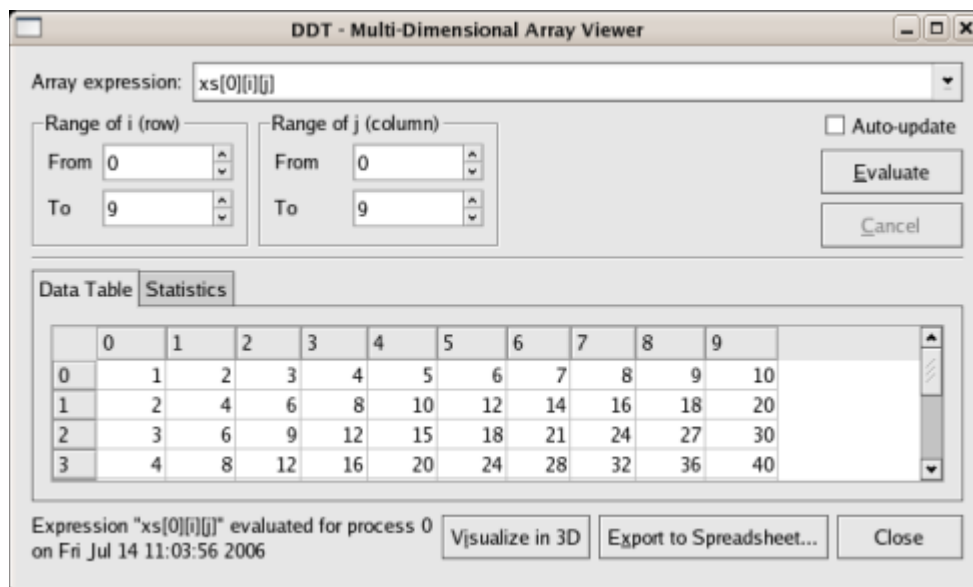


Fig.29 Multi-dimensional array viewer

Visualizing Data

If your system is OpenGL-capable then a 2-D slice of an array, or table of expressions, may be displayed as a surface in 3-D space through the multi-dimensional array (MDA) viewer. After filling the table of the MDA viewer with values (see previous section), click `Visualize` to open a 3-D view of the surface. To display surfaces from two or more different processes on the same plot simply select another process in the main process group window and click `Evaluate` in the MDA window, and when the values are ready, click `Visualize` again. The surfaces displayed on the graph may be hidden and shown using the checkboxes on the right-hand side of the window.

The graph may be moved and rotated using the mouse and a number of extra options are available from the window toolbar.

The mouse controls are:

- Hold down the left button and drag the mouse to rotate the graph.
- Hold down the right button to zoom - drag the mouse forwards to zoom in and backwards to zoom out.
- Hold the middle button and drag the mouse to move the graph.

Please note: visualization is not possible without OpenGL support. If you are unsure about whether or not you can run OpenGL applications on your system, try typing "glxinfo" at the command prompt, or contact your system vendor. For machines without hardware OpenGL support, software emulation libraries such as MesaGL are also supported.

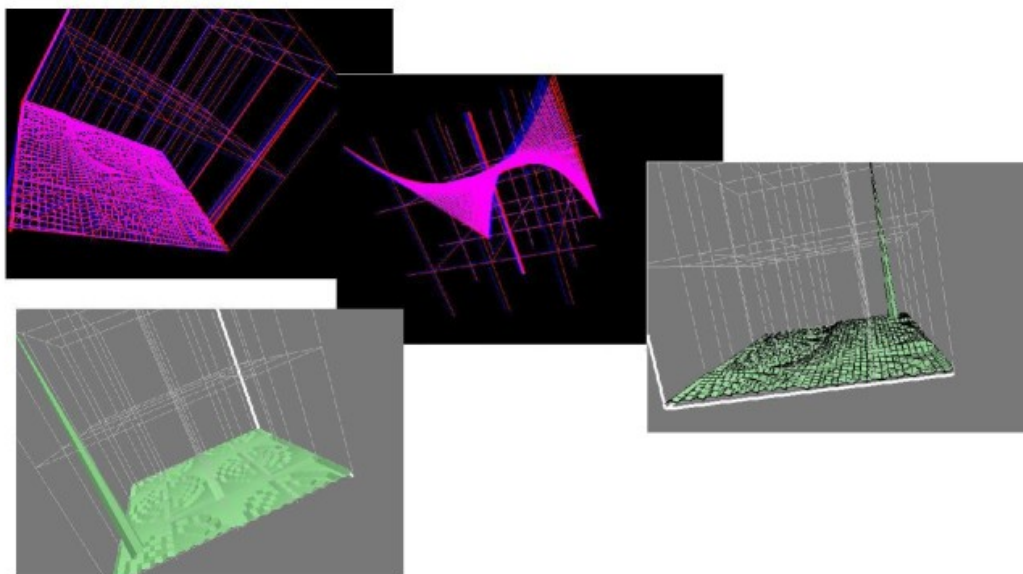


Fig.30 DDT visualization

The toolbar and menu offer options to configure lighting and other effects, including the ability to save an image of the surface as it currently appears. There is even a stereo vision mode that works with red-blue glasses to give a convincing impression of depth and form. Contact Allinea if you need to get hold of some 3D glasses.

Cross-Process Comparison

The Cross-Process Comparison window can be used to analyze expressions calculated on each of the processes in the current process group. This window displays information in three ways: raw comparison, statistically, and graphically.

To bring up the Cross-Process Comparison viewer, you may either right-click on a variable inside the `Source Code`, `Locals`, `Current Line(s)` or `Evaluate` windows - and then choose the "View Across Processes (CPC)" option. You can also bring up the CPC directly by selecting "View" from the menu bar and picking "Cross-Process Comparison".

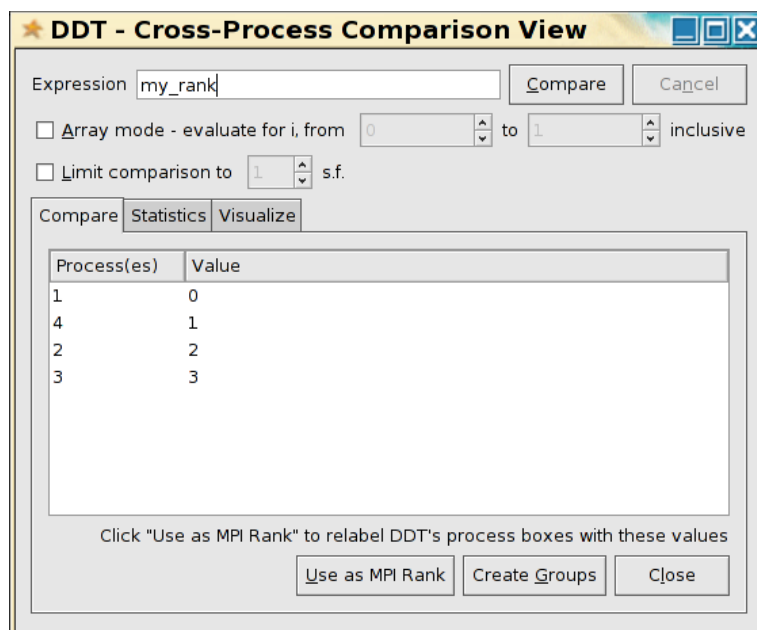


Fig.31 Cross-Process Comparison – compare view

Processes are grouped by expression value when using the raw comparison. The precision of this grouping can be specified (for floating point values) by filling the “limit” box. It is also practical to compare complete arrays, or sections of them, by parameterising the expression using “i” as the parameter.

You can turn each of these groupings of processes into a DDT process group by clicking the create groups button. This will create several process groups – one for each line in the panel. Using this capability large process groups can be managed with simple expressions to create groups. These expressions are any valid expression in the present language (i.e. C/C++/Fortran).

The “Use as MPI Rank” button is described in the next section, “Assigning MPI Ranks”.

The statistical view shows maximum, minimum, variance and similar with a box-and-whisker plot which displays max, min and interquartile range graphically.

The graphical panel shows a plot of values. In the case of a 1-D array expression the plot of values will display a line graph of values for all processes and these can be turned on and off individually by clicking the appropriate checkbox.

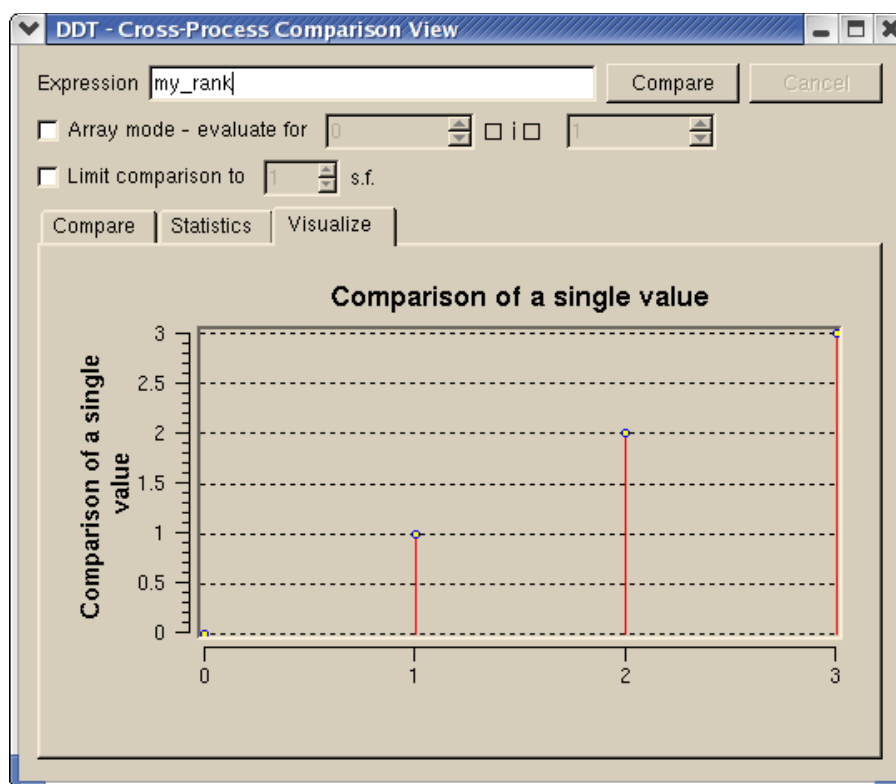


Fig.32 Cross-Process Comparison – visualize view

To use this window, select the Cross-Process Comparison window from the view menu. Type the expression that you wish to analyze. If you wish to compare an array, select the `Array Mode` option and type in the bounds that you require. Click the `Compare` button, and the current values will be evaluated on all the processes in the current group. If a process is still running in the current group, its value will not be found; press cancel and pause all the processes before trying again.

Assigning MPI Ranks

Sometimes, DDT cannot detect the MPI rank for each of your processes. This might be because you are using an experimental MPI version, or because you have attached to a running program, or only part of a running program. Whatever the reason, it is easy to tell DDT what each process should be called.

To begin, choose a variable that holds the MPI world rank for each process, or an expression that calculates it. Use the Cross-Process Comparison window to evaluate the expression across **all** the processes. If the variable is valid, the “Use as MPI Rank” button will be enabled. Click on it; DDT will immediately relabel all its processes with these new values.

What makes a variable or expression valid? These criteria must be met:

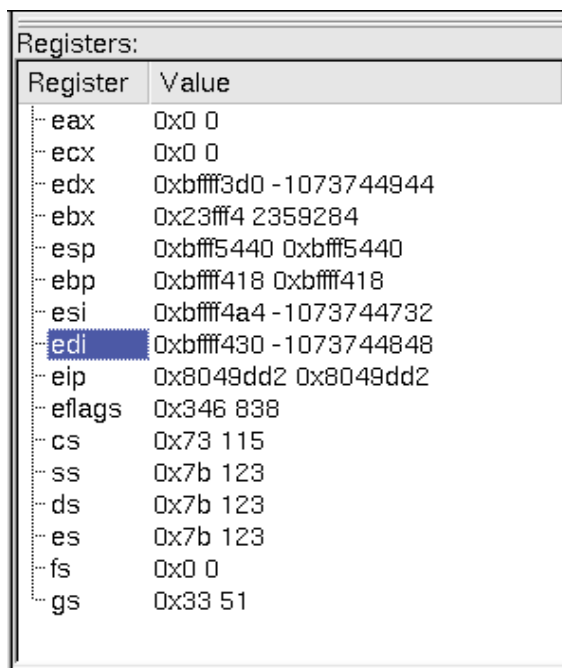
1. It must be an integer
2. Every process must have a unique number afterwards

These are the only restrictions. As you can see, there is no need to use the MPI rank if you have an alternate numbering scheme that makes more sense in your

application. In fact you can relabel only a few of the processes and not all, if you prefer, so long as afterwards **every** process still has a unique number.

Viewing Registers

To view the values of machine registers on the currently selected process, select the Registers window from the `View` pull-down menu. These values will be updated after each instruction, change in thread or change in stack frame.



Register	Value
eax	0x0 0
ecx	0x0 0
edx	0xbfff3d0 -1073744944
ebx	0x23fff4 2359284
esp	0xbfff5440 0xbfff5440
ebp	0xbfff418 0xbfff418
esi	0xbfff4a4 -1073744732
edi	0xbfff430 -1073744848
eip	0x8049dd2 0x8049dd2
eflags	0x346 838
cs	0x73 115
ss	0x7b 123
ds	0x7b 123
es	0x7b 123
fs	0x0 0
gs	0x33 51

Fig.33 Register view

Interacting Directly With The Debugger

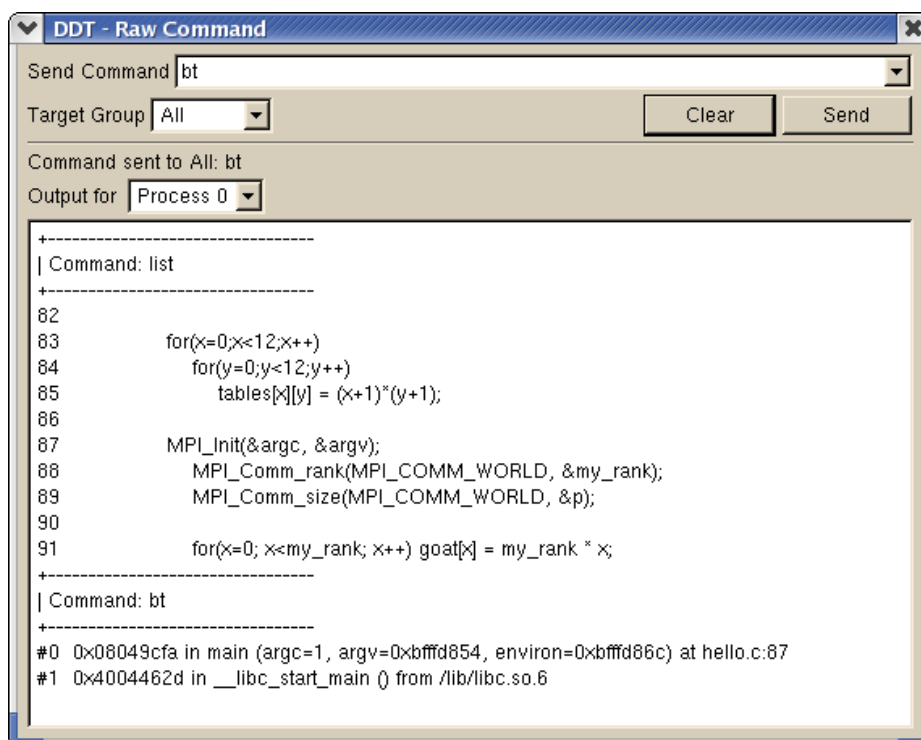


Fig.34 Debugger interaction window

DDT provides a Raw Command dialog that will allow you to send commands directly to the debugger interface. This dialog bypasses DDT and its book-keeping - if you set a breakpoint here, DDT will not list this in the breakpoint list, for example.

Be careful with this dialog; we recommend you only use it where the graphical interface does not provide the information or control you require. Sending commands such as `quit` or `kill` may cause the interface to stop responding to DDT.

Each command is sent to a group of processes (selected from within the dialog box - not necessarily the current group). To communicate with a single process, create a new group and drag that process into it.

The Raw Command window will not work with running processes and requires all processes in the chosen group to be paused.

7. Program Input And Output

DDT collects and displays output from all processes, placing output and error streams in separate panels that are controllable and allow output to be filtered by origin. Both standard output and error are handled identically, although on most MPI implementations, error is not buffered but output is and consequently can be delayed.

Viewing Standard Output And Error

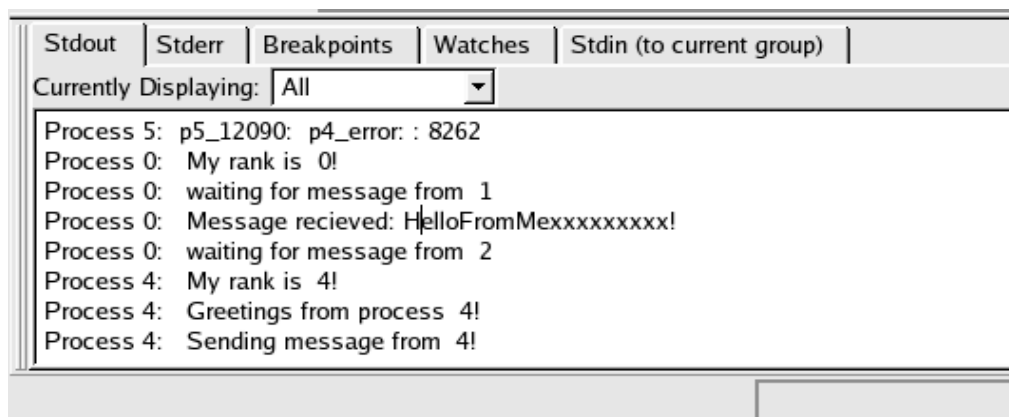


Fig.35 Standard output window

At the bottom of the screen (by default) there are tabs for displaying standard output and standard error.

The contents of these panels can be cut and pasted into the X-clipboard.

Displaying Selected Processes

By right-clicking the mouse you can choose whether to view output for the current process, the current process group if no process is selected, or for all processes. You also have the option to copy a selection to the clipboard.

MPI users should note that most MPI implementations place their own restrictions on program output. Some buffer it all until MPI_Finalize is called, others may ignore it or send it all through to one process. If your program needs to emit output as it runs, Allinea suggest writing to a file.

All users should note that many systems buffer stdout but not stderr. If you do not see your stdout appearing immediately, try adding an fflush(stdout) or equivalent to your code.

Saving Output

By right-clicking in an Output window, it is possible to save the contents of the window to a file.

Sending Standard Input (DDT-MP)

DDT provides an ``Input file`` option in the Session Control window. Using this window you may select the file you wish to use as the input file. DDT will automatically insert the correct arguments to your MPI implementation to cause your file to be “piped” into your MPI job.

Alternatively in DDT you may enter the arguments directly in the ``MPIRun Arguments`` box. For example if using MPI directly from the *command-line* you would normally use an option to the `mpirun` such as ``-stdin filename``, then you may add the same options to the ``MPIRun Arguments`` box when starting your DDT session in the Session Control Advanced dialog.

It is also possible to enter input from the keyboard instead of from a file. Using this mode you would start your program as normal, then run to the point where your program executes its ``scanf`` function or similar. You should then change to the input tab and type in the input you wish to send, the program will then continue executing.

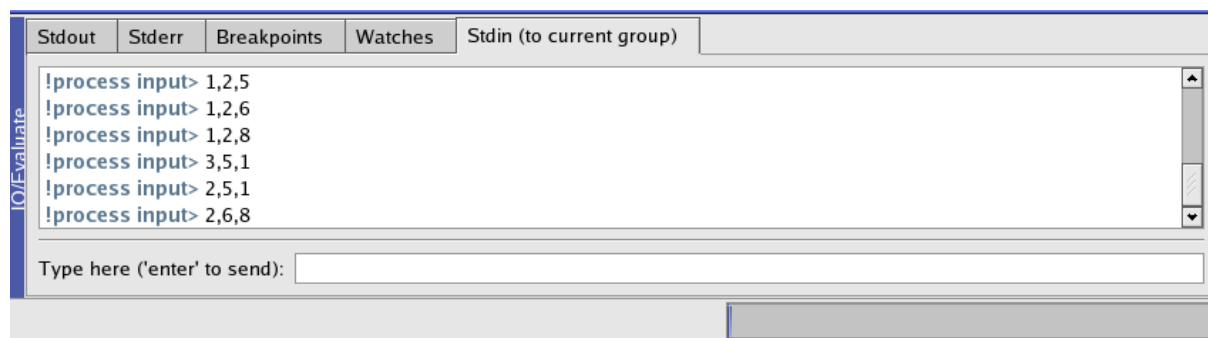


Fig.36 Sending Input

Note: If DDT is running on a fork-based system such as Scyld, or a ``-comm=shared`` compiled MPICH, your program may not receive an EOF correctly from the input file. If your program seems to hang while waiting for the last line or byte of input, this is likely to be the problem. See the FAQ or contact Allinea for a list of possible fixes.

8. Message Queues

Open the Message Queue view by selecting 'Message Queues' from the 'View' menu.

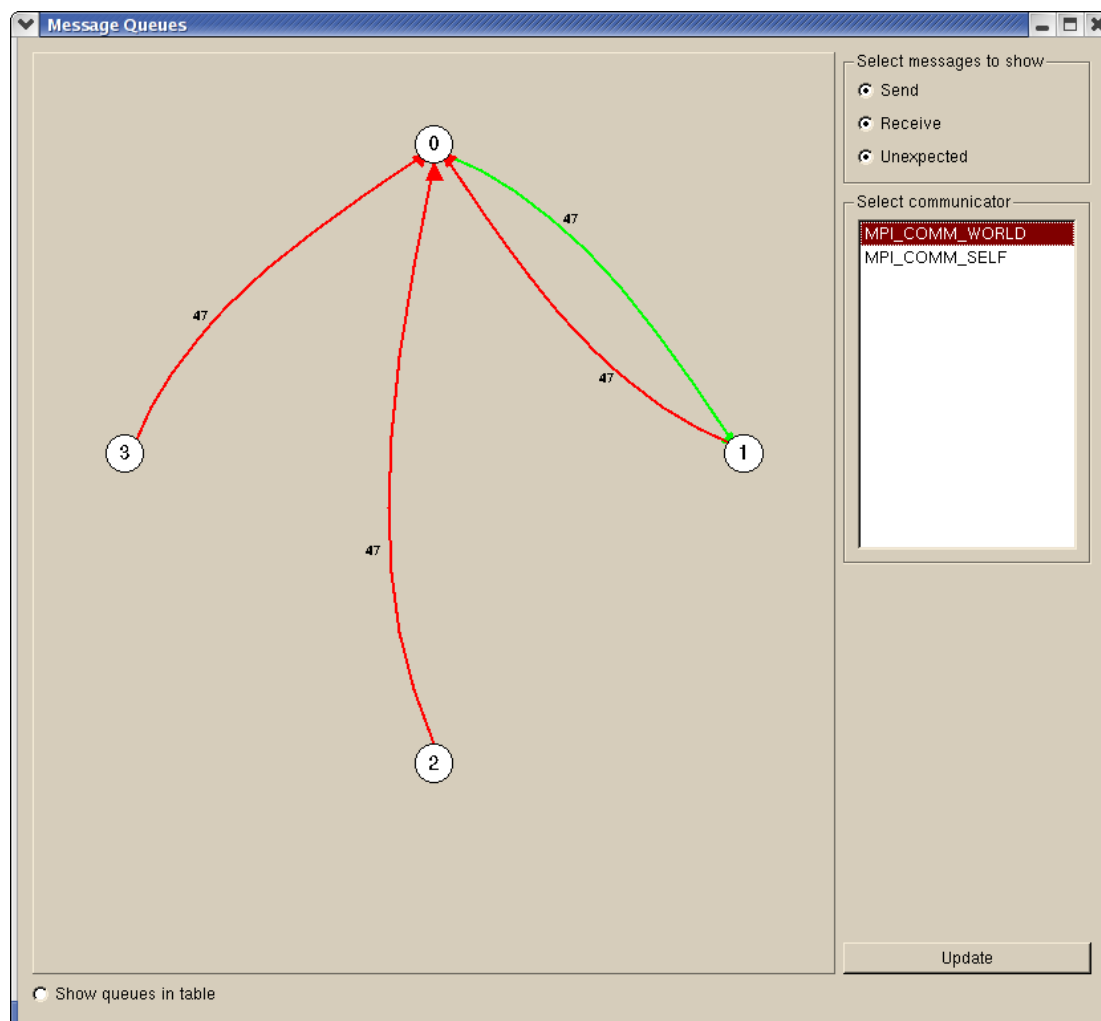


Fig.37 Message queue window

When the window appears you can click 'Update' to get the current queue information. Please note that this will stop all running processes. While DDT is gathering the data a dialog box will be displayed and you can cancel the request at any time.

You must select a communicator to see the messages in that group. The ranks displayed in the diagram are the ranks within the communicator (not MPI_COMM_WORLD). Different colours are used to display messages from each type of queue.

DDT uses the message queue debug interface to gather queue information. Within this interface each communicator has three distinct message queues:

Label	Description
Send Queue	Represents all the outstanding send operations
Receive Queue	Represents all the outstanding receive operations

Unexpected Message Queue	Represents messages that have been sent to this process but have not been received
--------------------------	--

In order to take advantage of the message queue view within DDT you need to compile the debug interface for your MPI implementation. In MPICH this is done by using '--enable-debug' when running configure. LAM automatically compiles the library.

DDT will try to load the default library for the MPI implementation (provided one exists) but for this to happen it must be in the LD_LIBRARY_PATH. If this is not convenient you can set the environment variable, DDT_QUEUE_DLL, to the absolute pathname of the library itself (e.g. /usr/local/mpich-1.2.7/lib/libtvmppich.so).

If you experience problems connecting to the message queue library when attaching to a process see the FAQ for possible solutions.

Please note that the quality of underlying implementations of the message queue debugging interface varies considerably – some of the data can therefore be incomplete.

9. Memory Debugging

DDT 1.9 introduced powerful parallel memory debugging capabilities. At the back end, DDT interfaces with a modified version of the dmalloc library. This powerful, cross-platform library intercepts memory allocation and deallocation calls and performs lots of complex heap- and bounds- checking. Fortunately, DDT makes it easy to use, even across tens or hundreds of parallel processes.

Configuration

Enabling memory debugging with DDT is easy. From the “Session Control” window that is displayed when DDT starts up, click on the 'Advanced' button to display the memory debugging controls. You can turn memory debugging on and off with the “Enable Memory Debugging” checkbox. Next to this box is a 'Memory Debugging Settings' button. Click on this to open the Memory Debugging Options window, shown here:

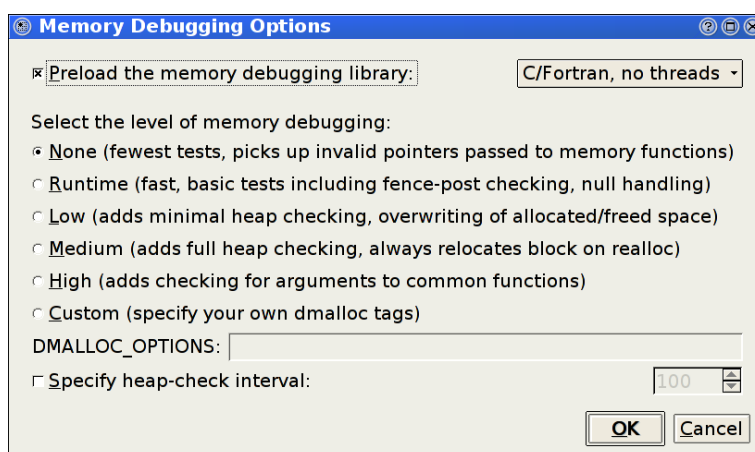


Fig.38 Memory debugging options

The array of options can be bewildering at first, but to use memory debugging with DDT you only need to understand the two controls at the top:

1. “Preload the memory debugging library” - when this is checked, DDT will automatically load the memory debugging library. This is great for most users. If you have linked against one of DDT's dmalloc libraries yourself then you can deselect this option. DDT can only preload the memory debugging library when you start a program through DDT **and if it uses shared libraries**. It is not possible to preload for statically-linked programs. You will have to re-link your program with the appropriate dmalloc library in ddt/lib/32 or ddt/lib/64 before running in DDT. If you attach to a running process, this setting has no effect. You can still use memory debugging when you attach, but you will have to manually link your program against one of the libdmalloc*.so versions in one of DDT's lib/32 and lib/64 directories and set the DMALLOC_OPTIONS environment variable before running your program.
2. The combo box, showing “C/Fortran, no threads” in the screenshot – click here and select an option that matches your program, be it C/Fortran, C++, single-threaded, multi-threaded, you should know what this means. Remember to come back here if you start/stop using multi-threading/OpenMP or change

between debugging C/Fortran and C++ programs. Many people find they can leave this set to C++/threaded for all their programs, rather than keep on changing the setting.

The rest of the window allows you to turn on/off specific dmalloc debugging features. The two most important things to remember are:

1. Even 'None' will catch trivial memory errors such as deallocating memory twice. Selecting this option does NOT turn off memory debugging!
2. The further down the list you go, the more slowly your program will execute. We find that for general use, anything up to "Low" is fast enough to use and will catch almost all errors. If you come across a memory error that's difficult to pin down, choosing a higher setting might expose the problem earlier, but you'll need to be very patient on large, memory intensive codes!

Almost all users can leave the heap check interval at its default setting. It determines how often dmalloc will check the entire heap for consistency. This is a slow operation, so is normally performed every 100 memory allocations. This figure can be changed manually – a higher setting (1000 or above) is recommended if your program allocates and deallocates memory very frequently (e.g. inside a computation loop).

Click on 'OK' to save these settings, or 'Cancel' to undo your changes.

NOTE: Choosing the wrong library to preload or the wrong number of bits may prevent DDT from starting your job, or may make memory debugging unreliable. These settings should be the first place you check if you experience problems when memory debugging is enabled.

Feature Overview

Once you have enabled memory debugging and started debugging, several new features become available. We will cover each of them in turn.

Stop on Error

As soon as dmalloc reports an error, DDT will stop the affected process and will display a message briefly reporting the type of error detected:

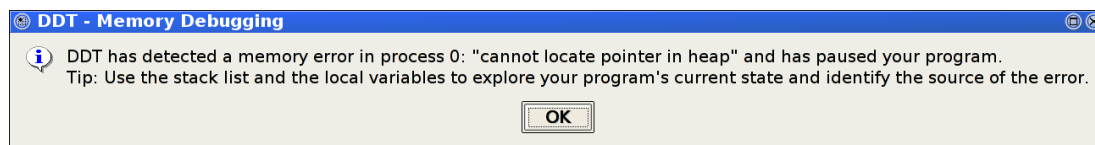


Fig.39 Memory error message

It will then highlight the line of your code that was being executed when the error was reported. Often this is enough to debug simple memory errors, such as freeing or dereferencing an unallocated variable, iterating past the end of an array and so on. If it is not clear, you may find yourself checking some of the variables and pointers referenced to see whether they're valid and which line they were allocated on, which brings us to:

Check Validity

Any of the variables or expressions in the 'Evaluate' window can be right-clicked on to bring up a menu. If memory debugging is enabled, one option will be “Check pointer is valid”. Clicking on this will pop up a message telling you whether or not that expression points to memory that was allocated on the heap or not¹:

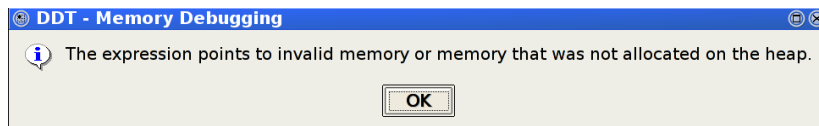


Fig.40 Invalid memory message

This is particularly useful for checking function arguments, and key variables when things seem to be going awry. Of course, just because memory is valid doesn't mean it is the same type as you were expecting, or of the same size, or the same dimensions and so on. To help you discover this we take you back to the place the memory was first allocated with our next feature:

View Pointer Details

Just next to the 'Check Validity' option on the menu is 'View Pointer Details'. DDT will show you the amount of memory allocated to the pointer and which part of your code originally allocated that memory:

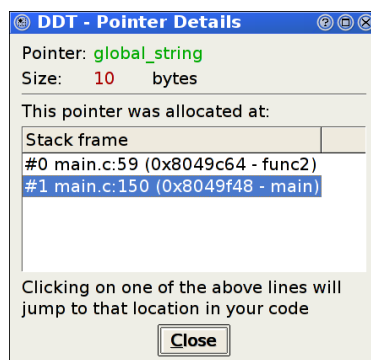


Fig.41 Pointer details

Clicking on any of the stack frames will display the relevant section of your code, so you can see where you allocated the variable in the first place. Should you want to check dynamic values at the time of allocation, you can now place a breakpoint here and use the new “Restart Session” function to re-run the program from the start while keeping your breakpoints, evaluated expressions and so on.

Writing Beyond An Allocated Area

If the Memory Debugging settings are “Runtime” or above, DDT will perform “Fence Post” checking. In this mode, an extra portion of memory is allocated at the start and end of your allocated block, and a pattern is written into this area. If you

¹ Memory allocated on the heap means something returned by “malloc”, “ALLOCATE”, “new” and so on. A pointer may also point to a local variable, in which case DDT will tell you it does not point to data on the heap. This can be useful, since a common error is taking a pointer to a local variable that later goes out of scope.

attempt to write beyond your data, say by a few elements, then this will be noticed by DDT.

Note that your program will not be stopped at the exact location at which your program wrote beyond the allocated data – it will only stop at the next heap consistency check. Heap consistency checks are only carried out after a defined number of allocation/deallocations – and the number of allocations or deallocations between these checks is set in the memory debugging settings, as described in the configuration section of this chapter.

Note also that it is not possible to detect reads beyond the bounds of allocation, except when such reads cause an actual segmentation violation.

Current Memory Usage

Of course, another problem of memory management is in memory leaks. If the size of your program grows faster than you expect, you may well be allocating memory and not freeing it when it's finished with. Such problems are typically difficult to diagnose, fiendishly so in a parallel environment, but DDT makes it trivial for us all. At any point in your program you can go to 'View->Current Memory Usage' and DDT will display a screenful of information about the currently allocated memory in your program for the currently selected group:

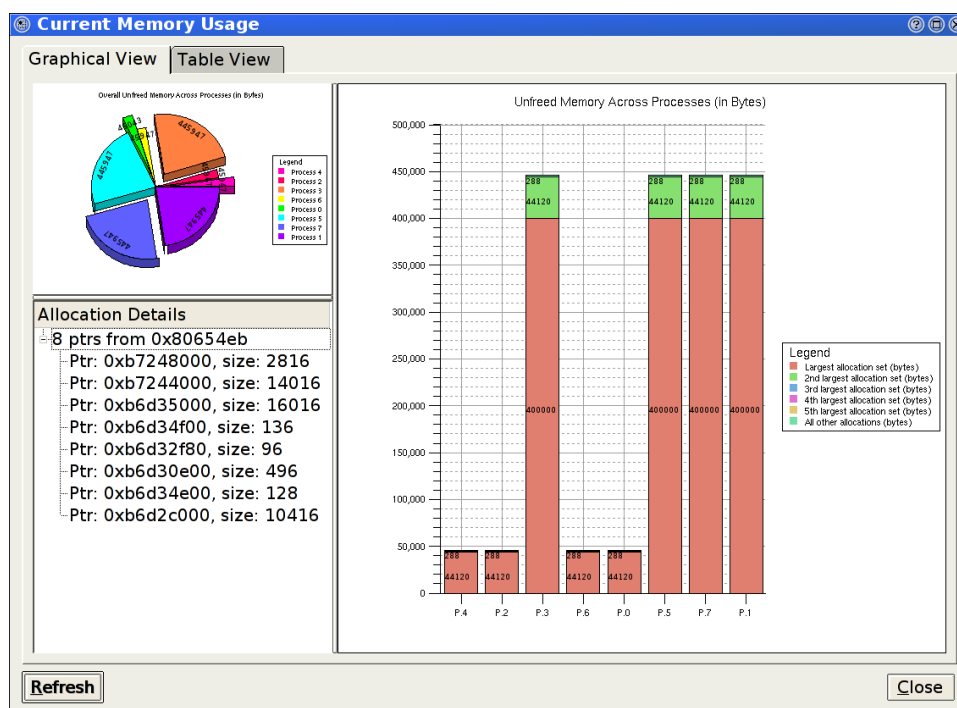


Fig.42 Memory usage graphs

The pie chart gives an at-a-glance comparison of the total memory allocated to each process. This gives a good indication of the balance of memory allocations; any one process taking an unusually large amount of memory is usually easily visible here.

The stacked bar chart on the right is where the most interesting information starts. Each process is represented by a bar, and each bar broken down into blocks of colour that represent the cumulative size of each allocation. Let's break that down a bit. Say your program contains a loop that allocates a hundred bytes that is

never freed. That's not a lot of memory. But if that loop is executed ten million times, you're looking at a gigabyte of memory being leaked! This chart groups memory allocations by the return address. For practical purposes, this means that each block of colour represents the total memory currently allocated from a particular line of code. There are 6 blocks in total. The first 5 represent the 5 lines of code with the most memory allocated, and the 6th (at the top) represents the rest of the allocated memory, wherever it is from.

As you can see, large allocations (if your program is close to the end, or these grow, then they are severe memory leaks) show up as large blocks of colour. Typically, if the memory leak does not make it into the top 5 allocations under any circumstances then it isn't that big a deal – although if you are still concerned you can view the data in the 'Table View' yourself.

If any block of colour interests you, click on it. This will display detailed information about the memory allocations that make it up in the bottom-left pane. Scanning down this list gives you a good idea of what size allocations were made, how many and where from. Clicking on any one of these will display the 'Pointer Details' view described above, showing you exactly where that pointer was allocated from in your code.

Some compilers wrap memory allocations inside many other functions. In this case DDT may find, for example, that all Fortran 90 allocations are inside the same routine. This can also happen if you have written your own wrapper for memory allocation functions. In these circumstances you will see one large block in the main view and will have to click on it to see the list of the memory allocations inside it. You can expect to see a smarter way around this in future versions!

Another valuable use of this window is to run the program for a while, refresh the window, run it for a bit longer, refresh the window and so on – if you pick the points at which to refresh (e.g. after units of work are complete) you can watch as the memory load of the different processes in your job fluctuates and will easily spot any areas that grow and grow – these are problematic leaks.

Naturally it occurs to you that some of this information is of interest for balance and other purposes. DDT goes one step further and provides you with our next feature:

Memory Statistics

The menu option 'View->Memory Statistics' displays a window like this one:



Fig.43 Memory statistics

Again, this is filtered by the currently-selected process group. The contents and location of the memory allocations themselves are not repeated here; instead this window displays the total amount of memory allocated/freed since the program began in the left-hand pane. This can help show if your application is unbalanced, if particular processes are allocating or failing to free memory and so on. The right hand pane shows the total number of calls to allocate/free functions by process. At the end of program execution you can usually expect the total number of calls per process to be similar (depending on how your program divides up work), and memory allocation calls should always be greater than deallocation calls - anything else indicates serious problems!

10. Advanced Data Display and C++ STL Support

With the DDT Wizard facility, DDT can take advantage of a user-provided shared-library (a.k.a. “Wizard Library”) to customise debugging behaviour. A Wizard Library can be used by DDT to improve the display of data structures (“Evaluate Expressions”) which are difficult to automatically navigate without specialist domain knowledge.

An example where this would be useful is for the display of C++ data structures that are templated or involve inheritance (e.g. the C++ Standard Library container classes, the Trolltech Qt library, etc.). The interface is not specific to C++ and as such other languages, such as C, can work with the provided API.

DDT comes with a sample Wizard Library for the following C++ Standard Library classes:

- vector
- deque
- list
- pair
- set
- multiset
- map
- multimap
- string

Please note that if your compiler version differs from that which is standard with the DDT installation, you may need to recompile the wizard library. Source and the makefile are provided for this purpose with the DDT installation.

Using The Sample Wizard Library

The sample Wizard Library works with target programs compiled with GNU GCC and using GNU gdb as the underlying debugger.

To use the sample Wizard Library, send a variable to the “Evaluate” window. Either:

- select a line in the “Source Code” window, right-click the mouse whilst the cursor is over the variable and choose the “Add To Evaluations” option, or
- select a line in the “Source Code” window, select the variable in the “Current Line(s)” window and drag it into the “Evaluate” window, or
- select the variable in the “Locals” window and drag it into the “Evaluate” window

Once in the “Evaluate” window, select the variable and right-click the mouse. From the pop-up menu that appears, select the “Evaluate with Wizard” option.

This should lead to a re-display of the value of the variable, this time evaluated by the Wizard Library rather than by DDT itself.

To re-display the value of the variable as it was before the Wizard Library was used, right-click the variable again and select “Evaluate without Wizard” from the pop-up menu.

As an example, suppose a C++ program contained the following code:

```
string s = "hello world";
```

Without evaluation by the Wizard Library, the “Evaluate” window displays something like the following:

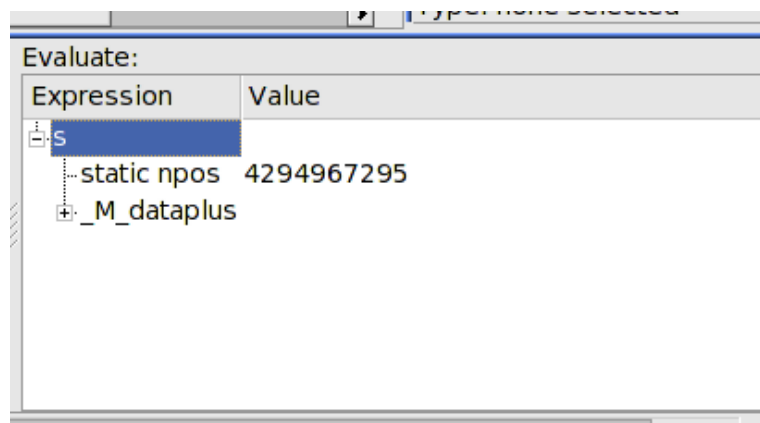


Fig.44 Evaluating without wizard

After evaluation with the Wizard Library, the display changes to the following:

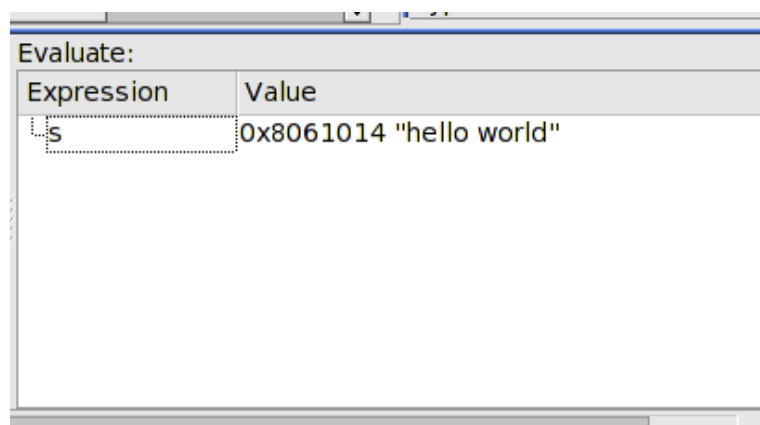


Fig.45 Evaluating with wizard

Writing A Custom Wizard Library

Full source code for the sample Wizard Library is included with the DDT distribution. However, that is in excess of 1400 lines of code and comments. To explain how to write your own, the following sections will cover writing a small example Wizard Library that works just for the C++ Standard Library string class.

Theory Of Operation

A Wizard Library must support the following interface:

```
typedef VirtualProcess*(*CurrentFun)();  
  
extern "C"  
{  
    void* initialize(CurrentFun);  
  
    Expression evaluate(const std::string&);  
}
```

When DDT has to evaluate an expression using the a Wizard Library, each evaluation causes a call to the Wizard Library's `initialize()` function followed by a call to the `evaluate()` function.

The `initialize()` function takes one argument, which is a pointer to a DDT function to call whenever the Wizard Library requires access to a handle for the current `VirtualProcess`.

The `VirtualProcess` handle is required by the Wizard Library's `evaluate()` function to allow it to make calls back into DDT in order to service the evaluation request.

The `evaluate()` function takes one argument, which is a constant-reference to a string containing the expression to be evaluated. The function must perform some calculations and record the results inside an `Expression` before returning that back to DDT.

DDT will examine the returned `Expression` and display the contents as the value of the expression in the "Evaluate" window.

So, an example of a minimal (and fairly useless) Wizard Library would look like:

```
#include <Interface.h>  
  
#include <iostream>  
#include <string>  
  
using namespace std;  
  
typedef VirtualProcess*(*CurrentFun)();  
  
static CurrentFun current = 0;  
  
extern "C"  
{  
    void* initialize(CurrentFun _current);  
  
    Expression evaluate(const string& e);  
}
```

```
void* initialize(CurrentFun _current)
{
    current = _current;
    return (void*) 1;
}

Expression evaluate(const string& e)
{
    cerr << "evaluate(" << e << ")" << endl;

    Expression r(e);

    try
    {
        VirtualProcess* v(current());
        if (v)
        {
            // Normally do some real work here, but just print
            // a diagnostic for now ...
            cerr << "evaluate(" << e << "): acquired handle" <<
out          endl;
        }
    }
    catch (...)
    {
        cerr << "evaluate(" << e <<
        "): unknown exception caught" << endl;
    }

    cerr << "evaluate(" << e << "): return" << endl;

    return r;
}
```

Compiling A New Wizard Library

Two steps are required to build a new Wizard Library:

- create an object file from the source code, and
- link the object file and another, Expression.o, to create the Wizard Library

Expression.o contains the required code to handle Expressions and is provided with the DDT distribution.

The following example builds a Wizard Library using GNU g++:

```
$ g++ -c -pipe -Wall -W -g -fPIC -I. -o example1.o example1.cpp
$ g++ -shared -Wl,-E -Wl,-soname,libwizardexample1.so.1 -o \
    libwizardexample1.so.1.0.0 Expression.o example1.o
```

Using A New Wizard Library

DDT uses an environment variable, `DDT_WIZARD_DLL`, to indicate the location of the Wizard Library (if any) to use. Assuming your Wizard Library has been compiled as above under `{installation-directory}/wizard`, the following Bourne-shell commands will DDT to run and use the new Wizard Library on an example `your_test_program` binary.

```
$ DDT_WIZARD_DLL={installation-directory}/wizard/libwizardexample1.so.1.0.0
$ export DDT_WIZARD_DLL
$ ddt your_test_program
```

The VirtualProcess Interface

DDT provides the Wizard Library with a `VirtualProcess` interface for *each* target process being debugged. Although there may be many of these `VirtualProcess` instances, only the one corresponding to the *current* target process is available when the Wizard Library is invoked by DDT.

As seen in the example above, the current `VirtualProcess` is retrieved using the “get current” function-pointer passed into the Wizard Library during DDT's call of the `initialize()` function:

```
...
VirtualProcess* v(current());
if (v)
...
```

The `VirtualProcess` interface contains the following methods:

```
virtual std::string readType(const std::string expression);
virtual Expression readExpression(const std::string expression);
```

`readType()` sends an expression back into DDT and returns a textual representation (according to the underlying debugger, that is) of the *type* associated with that expression. This can be used to determine how to process the original `evaluate()` request from DDT.

`readExpression()` send an expression back into DDT and returns an `Expression` representing the results of evaluating that expression.

The following code modifies a section of our first example, by reading the type of the expression sent in by DDT and then sending the expression back into DDT for evaluation. This is somewhat of a waste of time, in that it is acting as a “Loopback Wizard” and adding no extra value!

```

ostream& operator<< (ostream& strm, const Expression& e)
{
    strm << "{ name = '" << e.getDisplayName() <<
        "'", value = '" << e.getValue() <<
        "'" << " } ";
    return strm;
}

Expression evaluate(const string& e)
{
    cerr << "evaluate(" << e << ")" << endl;

    Expression r(e);

    try
    {
        VirtualProcess* v(current());
        if (v)
        {
            string eType(v->readType(e));

            cerr << "evaluate(" << e << "): type = '" <<
                eType << "'" << endl;

            const Expression eResult(v->readExpression(e));

            cerr << "evaluate(" << e << "): result = " <<
                eResult << endl;

            r = eResult;
        }
    }
    catch (...)
    {
        cerr << "evaluate(" << e <<
            "): unknown exception caught" << endl;
    }

    cerr << "evaluate(" << e << "): return" << endl;

    return r;
}

```

After building this example and running DDT on a test program containing the following code:

```

string s = "hello world";

int i = 42;

```

the following is output when both variables are evaluated with the “Null Wizard”:

```
evaluate(i)
evaluate(i): type = 'int'
evaluate(i): result = { name = 'i', value = '42' }
evaluate(i): return
evaluate(s)
evaluate(s): type = 'string'
evaluate(s): result = { name = 's', value = '' }
evaluate(s): return
```

DDT reports the *type* and *value* of the integer variable *i*, as you would expect. The *type* of the string variable *s* proves no problem either. However, the *value* of the string variable *s* can not be determined – which is why a Wizard Library is required!

The Expression Class

The results of an evaluation are returned using the Expression class. Simple Expressions consist of a couple of strings:

- `displayName` (the notional name of this expression)
- `value` (the result of evaluating the expression)

Where the “name” of the expression differs from the string sent to DDT for evaluation, another string field in Expression is used:

- `realName` (actual expression evaluated by underlying debugger, can be the same as the `displayName` for simple expressions)

An example of this will be coming up soon when we finally implement evaluation of strings.

Where more complicated expressions are involved, the `value` field is not used and is left blank. Instead, a list of “child” sub-Expressions are slung under this Expression using a fourth field:

- `children` (further sub-expressions)

The type of this fourth field is `std::vector<Expression>`.

An example where the `children` field would be used is during the evaluation of a variable of type `vector`. The `displayName` would be the name of the variable. The `value` would be empty. There would be a number of children, one for each entry in the vector. Each of these children would have their `displayName` set to a pretty-printed string representing their index (e.g. “[4]”). Whether each of the children would have a `value` or whether they would contain further children would depend upon how complex the data structure being evaluated was.

For instance, the following screen snapshot shows the evaluation of a variable of type `pair<deque<string>, list<list<string>>>`:

Evaluate:	
Expression	Value
pairOfDequeOfStringsAndListOfListOfStrings	
first	
[0]	0x8061744 "a deque-B string"
[1]	0x806176c "a deque-B string"
[2]	0x8061794 "a deque-B string"
second	
[0]	
[0]	0x8061e2c "a list-A string"
[1]	0x8061e4c "a list-A string"
[1]	
[0]	0x8061e7c "a list-B string"
[1]	0x8061eac "a list-B string"
[2]	0x8061edc "a list-B string"

Fig.46 Evaluation of strings

The Expression's displayName is set to pairOfDequeOfStringsAndListOfListOfStrings and it has no value and two children. The first child Expression has a displayName set to first, no value and has three children. The first grandchild Expression has a displayName set to [0], value set to "a deque-A string" and no further children.

Final String-Enabled Wizard Library Example

Update the operator<<() function to display more information:

```
ostream& operator<< (ostream& strm, const Expression& e)
{
    strm << "{ name = '" << e.getDisplayName() <<
        "' , realName = '" << e.getRealName() <<
        "' , value = '" << e.getValue() <<
        "' } ";

    return strm;
}
```

Examination of the contents of a string variable (in a test program compiled with GNU g++) shows that it has a `_M_dataplus` data member which is further composed of a `_M_p char*` pointer.

With this in mind, modify the `evaluate()` function to behave specially for variables of type string:

```
Expression evaluate(const string& e)
{
    cerr << "evaluate(" << e << ")" << endl;

    Expression r(e);
```

```

try
{
    VirtualProcess* v(current());
    if (v)
    {
        string eType(v->readType(e));

        cerr << "evaluate(" << e << "): type = '" <<
            eType << "'" << endl;

        if (eType == "string")
        {
            Expression
                eResult(v->readExpression(e +
                    "._M_dataplus._M_o
"));
            // Want the pretty-printed name to be the name
            // the string itself.
            eResult.setDisplayName(e);

            r = eResult;
        }
        else
        {
            const Expression eResult(v->readExpression(e));

            r = eResult;
        }

        cerr << "evaluate(" << e << "): result = " <<
            r << endl;
    }
}
catch (...)
{
    cerr << "evaluate(" << e <<
        "): unknown exception caught"<< endl;
}

cerr << "evaluate(" << e << "): return" << endl;

return r;
}

```

When DDT is run on the test program using a Wizard Library with this code, evaluating the integer and string variables leads to the following output:

```
evaluate(i)
evaluate(i): type = 'int'
evaluate(i): result = { name = 'i', realName = 'i', value =
'1' }
evaluate(i): return
evaluate(s)
evaluate(s): type = 'string'
evaluate(s): result = { name = 's', realName =
's._M_dataplus._M_p', value = '0x8061014 "hello world"' }
evaluate(s): return
```

Notice that the Expression for the string evaluation has different values for the `displayName` and `realName` fields.

Single-stepping the test program yields the following output:

```
evaluate(i)
evaluate(i): type = 'int'
evaluate(i): result = { name = 'i', realName = 'i', value =
'1' }
evaluate(i): return
evaluate(s._M_dataplus._M_p)
evaluate(s._M_dataplus._M_p): type = 'char *'
evaluate(s._M_dataplus._M_p): result = { name =
's._M_dataplus._M_p', realName = 's._M_dataplus._M_p', value =
'0x8061014 "hello world"' }
evaluate(s._M_dataplus._M_p): return
```

What is happening here is that DDT has decided to re-evaluate each variable after the single-step operation has occurred, and has recognised that the string variable's `displayName` and `realName` differ. The `realName` string is sent back to the Wizard Library for evaluation and matched up to the original variable in the "Evaluate" window upon reply.

11. The Licence Server

The licence server supplied with DDT is capable of serving clients for several different licences, enabling one common server to serve all Allinea software in an organization.

Running The Server

For security, the licence server should be run as an unprivileged user (e.g. nobody). If run without arguments, the server will use licences in the current directory (files matching Licence* and License*). An optional argument specifies the pathname to be used instead of the current.

System administrators will normally wish to add scripts to start the server automatically during booting.

Running DDT Clients

DDT will, as is also the case for fixed licences, use a licence file either specified via environment variables (DDT_LICENCE_FILE or DDT_LICENSE_FILE) or from the default location of {installation-directory}/Licence.

In the case of floating licences this file is unverified and in plain-text, it can therefore be changed by the user if settings need to be amended.

The fields are:

Name	Required	Description
hostname	Yes	The hostname, or IP address of the licence server
ports	No	A comma separated list of ports to be tried locally for GUI-backend communication in DDT, Defaults to 4242,4243,4244,4244,4245
serial_number	Yes	The serial number of the server licence to be used
serverport	Yes	The port the server listens on
type	Yes	Must have value 2 – this identifies the licence as needing a server to run properly

Note: The serial number of the server licence is specified as this enables a user to be tied to a particular licence.

Logging

Set the environment variable DDT_LICENCE_LOGFILE to the file that you wish to append log information to. Set DDT_LICENCE_LOGLEVEL to set the amount of information required. These steps must be done prior to starting the server.

Level 0: no logging.

Level 1: client licences issued are shown, served licences are listed.

Level 2: stale licences are shown when removed, licences still being served are listed if there is no spare licence.

Level 3: full request strings received are displayed
Level 6 is the maximum.

In level 1 and above, the MAC address, username, process ID, and IP address of the clients are logged.

Troubleshooting

Licences are plain-text which enables the user to see the parameters that are set; a checksum verifies the validity. If problems arise, the first step should be to check the parameters are consistent with the machine that is being used (MAC and IP address), and that, for example, the number of users is as expected.

Adding A New Licence

To add a new licence to be served, copy the file to the directory where the existing licences are served and restart the server. Existing clients should not experience disruption, if the restart is completed within a minute or two.

Examples

In this example, a dedicated licence server machine exists but uses the same filesystem as the client machines, and DDT is installed at `"/opt/software/ddt"`

To run the `licenceserver` as `nobody`, serving all licences in `"/opt/software/ddt"`, and logging most events to the `"/tmp/licence.ddt.log"`.

```
% su - nobody
Password:
% export DDT_LICENCE_LOGFILE=/tmp/licence.ddt.log
% export DDT_LICENCE_LOGLEVEL=2
% cd /opt/software/ddt
% ./bin/licenceserver /opt/software/ddt/ &
% exit
```

Serving the floating licences from the same directory as a normal DDT installation is possible as the licence server will ignore licences that are not server licences.

If the server licence is file `"/opt/software/Licence.server.physics"` and is served by the machine `server.physics.acme.edu`, at port 4252, the licence would look like:

```
type=3
serial_number=1014
max_processes=48
expires=2004-04-01 00:00:00
support_expires=2004-04-01 00:00:00
mac=00:E0:81:03:6C:DB
```

```
interface=eth0
debuggers=gdb
serverport=4252
max_users=2
beat=60
retry_limit=4
hash=P5I: ?L, FS=[ CCTB<IW4
hash2=c18101680ae9f8863266d4aa7544de58562ea858
```

Then the client licence could be stored at `"/opt/software/Licence"` and contain:

```
type=2
serial_number=1014
hostname=server.physics.acme.edu
serverport=4252
```

Example Of Access Via A Firewall

SSH forwarding can be used to reach machines that are beyond a firewall, for example the remote user would start:

```
ssh -C -L 4252:server.physics.acme.edu:4242 login.physics.acme.edu
```

And a local licence file should be created:

```
type=2
serial_number=1014
hostname=localhost
serverport=4252
```

Querying Current Licence Server Status

The licence server provides a simple HTML interface to allow for querying of the current state of the licences being served. Point your favorite web browser at a URL of the form:

```
http://<hostname>:<serverport>/status.html
```

For example, using the values from the licence file examples, above:

```
http://server.physics.acme.edu:4252/status.html
```

Initially, no licences will be being served, and the output in your browser window should look something like:

```
[Licences start]
  [Licence Serial Number: 1014]
    [No licences allocated - 2 available]
[Licences end]
```

As licences are served and released, this information will change. To update the licence server status display, simply refresh your web browser window. For example, after one DDT has been started:

```
[Licences start]
  [Licence Serial Number: 1014]
    [1 licences available]
    [Client 1]
      [mac=00:04:23:99:79:65; uname=gwh; pid=14007; licence=1014]
      [Latest heartbeat: 2004-04-13 11:59:15]
[Licences end]
```

Then, after another DDT is started and the web browser window is refreshed (notice the value for number of licences available):

```
[Licences start]
  [Licence Serial Number: 1014]
    [0 licences available]
    [Client 1]
      [mac=00:04:23:99:79:65; uname=gwh; pid=14007; licence=1014]
      [Latest heartbeat: 2004-04-13 12:04:15]
    [Client 2]
      [mac=00:40:F4:6C:4A:71; uname=graham; pid=3700; licence=1014]
      [Latest heartbeat: 2004-04-13 12:04:59]
[Licences end]
```

Finally, after the first DDT finishes:

```
[Licences start]
  [Licence Serial Number: 1014]
    [1 licences available]
    [Client 1]
      [mac=00:40:F4:6C:4A:71; uname=graham; pid=3700; licence=1014]
      [Latest heartbeat: 2004-04-13 12:07:59]
[Licences end]
```

Licence Server Handling Of Lost DDT Clients

Should the licence server lose communication with a particular instance of a DDT client, the licence allocated to that particular DDT client will be made unavailable for new DDT clients until a certain timeout period has expired. The length of this timeout period can be calculated from the licence server file values for beat and retry_limit:

$$\text{lost_client_timeout_period} = (\text{beat seconds}) * (\text{retry_limit} + 1)$$

So, for the example licence files above, the timeout period would be:

$$60 * (4 + 1) = 300 \text{ seconds}$$

During this timeout period, details of the “lost” DDT client will continue to be output by the licence server status display. As long as additional licences are available, new DDT clients can be started. However, once all of these additional licences have been allocated, new DDT clients will be refused a licence while this timeout period is active.

After this timeout period has expired, the licence server status will continue to display details of the “lost” DDT client until another DDT client is started. The licence server will grant a licence to the new DDT client and the licence server status display will then reflect the details of the new DDT client.

A. Supported Platforms

A full list of supported platforms and configurations is maintained on the Allinea website. It is likely that MPI distributions supported on one platform will work immediately on other platforms.

Platform	Operating Systems	MPI	Compilers
Intel/AMD x86 AMD Opteron (32+64) Intel Itanium 2 Intel EM64T IBM Power	Redhat 7,8,9 and above, SuSE 8,9 and similar	SGI Altix, Bproc, LAM-MPI, MPICH, Myricom MPICH-GM and MPICH-MX, Quadrics MPI, Scali MPI Connect, SCore, Scyld, Intel MPI, OpenMPI, Bull MPI, Slurm MVAPICH (IBGD 1.8.1 and above)	GNU, Absoft, Intel, Pathscale, and Portland
IBM Power	AIX 5.2 and above	IBM PE	Native, GNU
Sun Sparc	Solaris 9 and above	Sun Clustertools 4 and above	Native – Studio 11
Sun Solaris Opteron	Solaris 10 and above	Sun Clustertools and MPICH	Native – Studio 11, GNU

B. Troubleshooting DDT

If you have any problems with DDT, please take a look at the FAQ list in Appendix C – you might just find the answer you're looking for. Equally, it's worth checking the support pages of the www.allinea.com and making sure you have the latest version of DDT.

If your problem persists, contact us directly by emailing support@allinea.com.

Problems Starting the DDT GUI

If DDT is unable to start, this is usually one of three reasons:

- DDT cannot connect to an X server. If you are running on a remote machine, make sure that your DISPLAY variable is set appropriately and that you can run simple X applications such as 'xterm' from the same command-line.
- The licence file is invalid – in this case DDT will issue an error message. You should verify that you have a licence file, that it is in a file called Licence in DDT's directory and check that the date inside it is still valid. If DDT still refuses to start, please contact Allinea.
- You are using a licence server, but DDT cannot connect to it. See the section on licence servers for more information on troubleshooting these problems.

Problems Starting Scalar Programs

There are a number of possible sources for problems. The most common is – for users with a multi-process licence – that the MPI implementation has **not** been set to “none”. From click on the 'Change' button next to the MPI name and select “none” from the drop-down list. If DDT reports a problem with MPI and you know your program is not using MPI, then this is usually the cause. If you HAVE selected “none” and DDT still mentions MPI then we would very much like to hear from you!

Other potential problems are:

- A previous DDT session is still running, or has not released resources required for the new session. Usually this can be resolved by killing stale processes. The most obvious symptom of this is a delay of approximately 60 seconds and a message stating that not all processes connected. You may also see, in the terminal, a QServerSocket message
- The target program does not exist or is not executable
- The selected debugger cannot be found
- The selected debugger has crashed
- DDT's backend daemon – ddt-debugger – is missing from DDT's bin directory – in this case you should check your installation, and contact Allinea for further assistance.

Problems Starting Multi-Process Programs

If you encounter problems whilst starting an MPI program with DDT, the first step is to establish that it is possible to run a single-process (non-MPI) program such as a trivial “hello world” - and resolve such issues that may arise. After this, attempt to run a multi-process job – and the symptoms will often allow a reasonable diagnosis to be made.

In the first instance, verify that MPI is installed correctly by running a job outside of DDT, such as the example in DDT's examples directory.

```
mpirun -np 8 ./a.out
```

Verify that mpirun is in the PATH, or the environment variable DDTMPIRUN is set to the full pathname of mpirun.

If the progress bar does not report that at least process 0 has connected, then the remote ddt-debugger daemons cannot be started or cannot connect to the GUI.

The majority of such problems are caused by environment variables not propagating to the remote nodes whilst starting a job. To a large extent, the solution to these problems depend on the MPI implementation that is being used. In the simplest case, for rsh based systems such as a default MPICH installation, correct configuration can be verified by rsh-ing to a node and examining the environment. It is worthwhile rsh-ing with the env command to the node as this will not see any environment variables set inside the .profile command. For example if your nodes use a .profile instead of a .bashrc for each user then you may well see a different output when running “rsh node env” than when you run “rsh node” and then run “env” inside the new shell.

If only one, or very few, processes connect, it may be because you have not chosen the correct MPI implementation. Please examine the list and look carefully at the options. Should no other suitable MPI be found, please contact Allinea for advice.

If a large number of processes are reported by the status bar to have connected, then it is possible that some have failed to start due to resource exhaustion, timing out, or, unusually, an unexplained crash. You should verify again that MPI is still working, as some MPI distributions do not release all semaphore resources correctly (for example MPICH on Redhat with SMP support built in).

To check for time-out problems, set the DDT_NO_TIMEOUT environment variable to 1 before launching the GUI and see if further progress is made. This is not a solution, but aids the diagnosis. If all processes now start, please contact Allinea for further long-term advice.

C. FAQs

Why does DDT say it can't find my hosts or the executable?

This can happen when attempting to attach to a process running on other machines. Ensure that the hostname(s) that DDT complains about are reachable using ping. If DDT fails to find the executable, ensure that it is available in the same directory on every machine. If you haven't already, then try using the Configuration Wizard to set up DDT's attach feature.

The progress bar doesn't move and DDT 'times out'

It's possible that the program 'ddt-debugger' hasn't been started by mpirun or has aborted. You can log onto your nodes and confirm this by looking at the process list BEFORE clicking 'Ok' when DDT times out. Ensure ddt-debugger has all the libraries it needs and that it can run successfully on the nodes using mpirun.

Alternatively, there may be one or more processes ('ddt-debugger', 'mpirun', 'rsh') which could not be terminated. This can happen if DDT is killed during its startup or due to MPI implementation issues. You will have to kill the processes manually, using 'ps x' to get the process ids and then 'kill' or 'kill -9' to terminate them.

This issue can also arise for mpich-p4mpd, and the solution is explained in Appendix E.

If your intended mpirun command is not in your PATH, you may either add it to your PATH or set the environment variable DDTMPIRUN to contain the full pathname of the correct mpirun.

The progress bar gets close to half the processes connecting and then stops and DDT 'times out'

This is likely to be caused by a dual-processor configured MPI distribution. Make sure you have selected 'smp-mpich' or 'scyld' as your MPI implementation in the DDT Options window. If this doesn't help, see Appendix E for a workaround and email support@allinea.com for further assistance.

My program runs but I can't see any variables or line number information, why not?

You should compile your programs with debug information included, this flag is usually -g.

My program doesn't start, and I can see a console error stating "QServerSocket: failed to bind or listen to the socket"

Ordinarily this message is not a sign of a problem - it is emitted when another DDT session, is running and consequently the DDT master uses another socket instead. However, if you know this not to be the case and your program is not starting, it's likely that a previous run of DDT has been unable to terminate and release resources completely. This is known to occur occasionally for MPICH-GM. If this happens, run `/usr/bin/killall -9 ddt-debugger` on your nodes - you can actually use `mpirun` to do this for you.

Why can't I see any output on stderr?

DDT automatically captures anything written to stdout/stderr and displays it. Some shells (such as `csh`) and debuggers (such as `dbx` on Solaris) do not support this feature in which case you may see your stderr mixed with stdout, or you may not see it at all. In any case we strongly recommend writing program output to files instead, since the MPI specification does not cover stdout/stderr behaviour.

DDT complains about being unable to execute malloc

Should this error message occur, often due to backend-debugger failure, it is possible to bypass this step. Set the environment variable `DDT_DONT_GET_RANK` to 1 on the nodes and this will force DDT to guess ranks, which may resolve the problem.

Sometimes this error is shown when DDT could not start your program at all. Check the arguments, memory debugging settings, library paths and so on have the values you would expect and then contact support@allinea.com for more help.

Some features seem to be missing (e.g. The Fortran Module Browser) - what's wrong?

This is because not all debuggers support every feature that DDT does and so they are disabled by removing the window/tab by from DDT's interface.

My code does not appear when I start DDT

If you cannot see any text at all, perhaps the default selected font is not installed on your system. Go into the ``Session -> Options`` window and choose a font such as Times or Century Schoolbook and you should now be able to see the code.

If you see a screen of text telling you that DDT could not find your source files, follow the instructions given. If you still cannot get DDT to show your source code, check that the code is available on the same machine as you are running DDT on, and that the correct file and directory permissions are set. If some files are missing, and others found, try adding source directories and rescanning – see Chapter 4 for further instruction.

If the problem persists, drops us a mail at support@allinea.com!

When I use step out my program hangs

You should not use the step out feature from the main function of your program. This will cause the debugger to hang. With some debuggers DDT will return a time-out error. Make sure you only use step out inside functions.

Some backend debug interfaces have been known to behave unpredictably when told to step out. If you suspect this is the problem, check you can leave the function by putting a breakpoint on the next executable statement and pressing 'play' instead. If this works, and step out doesn't, support@allinea.com would love to hear from you!

When viewing messages queues after attaching to a process I get a “Cannot find Message Queue DLL” error

DDT can only detect the message queue library automatically when the program is started from within DDT. If you want to debug message queues when attaching, set the DDT_QUEUE_DLL environment variable explicitly before you start DDT. For example:

```
DDT_QUEUE_DLL=/usr/local/mpich/lib/libtvmppich.so ddt
```

Your MPI documentation should give you the filename for your implementation. The files needed for LAM and MPICH are:

- LAM – liblam_totalview.so
- MPICH – libtvmppich.so

Some MPIs will need to be configured with a specific command-line option to turn on message queue debugging. For example, MPICH must be configured with the --enable-debug argument.

I get the error `The mpi execution environment exited with an error, details follow: Error code: 1 Error Messages: “mprun:mpmd_assemble_rsrcs: Not enough resources available”` when trying to start DDT

This error occurs when running DDT on a Solaris machine. If you select more processes than you have processors in your machine then mprun is not able to allocate the resources needed. To fix this simply add the argument `-W` to the `MPI Arguments` box, this will tell mprun to wrap the processes and will enable you to start your desired number of processes in DDT.

What do I do if I can't see my running processes in the attach window?

This is usually a problem with either your remote-exec script or your node list file. First check that the entry in your node list file corresponds with either localhost (if you're running on your local machine) or with the output of `hostname` on the desired machine.

Secondly try running remote-exec manually ie. `remote-exec ls` and check the output of this. If this fails then there is a problem with your remote-exec script. If `rsh` is still being used in your script check that you can rsh to the desired machine. Otherwise check that you can attach to your machine in the way specified in the `remote-exec` script. If you still experience problems with your script then contact Allinea for assistance.

When trying to view my Message Queues using mpich I get no output but also see no errors

This is a known problem on the Opteron/EM64T system with 64/32bit compatibility. If the DDT binary is 64-bit, but your target application is 32-bit, then the MPI message queue support library needs to be 32-bit also – and most MPI implementations do not support this configuration. Contact Allinea for further advice to obtain a 32-bit binary of DDT.

After I reload a session from a saved session file, the right-click menu options are not available in the file selector window

When a session is reloaded by default it has no selected process. With no process selected it is not possible to use these options. Select a process from the process group window and the menu options will appear.

The View Pointer Details window says my pointer is valid but doesn't show me which line of code it was allocated on

The Pathscale and PGI compilers have known issues that can cause this – please see the compiler notes in section E of this appendix for more details. If this happens with another compiler, please contact support@allinea.com with the vendor and version number of your compiler.

Obtaining Support

If this guide hasn't helped you the most effective way to get support is to email us with a detailed report. If possible, you should obtain a log file for the problem and email this to support@allinea.com.

You can generate a log file by running DDT like this:

```
ddt -debug -log log.ddt
```

Then simply reproduce the problem using as few processors and commands as possible and then close DDT as usual. On some systems this file might be quite large; be prepared to gzip or bzip it before attaching it to your email.

D. Notes On MPI Distributions

This appendix has brief notes on many of the MPI distributions supported by DDT. Advice on settings and problems particular to a distribution are given here.

Bproc

By default, the p4 interface will be chosen. If you wish to use GM (Myrinet), place -gm in the MPIrun arguments, and this will be used instead. Select Generic as the MPI implementation.

Bull MPI

Select Bull MPI from the MPI implementations list. In the “Advanced” settings, you may also wish to specify the partition that you wish to use – by adding

`-p partition_name`

You should ensure that prun, the command used to launch jobs, is in your PATH before starting DDT.

HP MPI

Select HP MPI as the MPI implementation.

A number of HP MPI users have reported a preference to using “mpirun -f jobconfigfile” instead of “mpirun -np 10 a.out” for their particular system. It is possible to configure DDT to support this configuration – using the support for batch (queuing) systems (see page).

The role of the queue template file is analogous to the “-f jobconfigfile”.

If your job config file normally contains:

```
-h node01 -np 2 a.out
-h node02 -np 2 a.out
```

Then your template file should contain:

```
-h node01 -np PROCS_PER_NODE_TAG /usr/local/ddt/bin/ddt-debugger
-h node02 -np PROCS_PER_NODE_TAG /usr/local/ddt/bin/ddt-debugger
```

and the “submit command” box should be filled with

```
mpirun -f
```

Select the “Template uses NUM_NODES_TAG and PROCS_PER_NODE_TAG” radio button. After this has been configured by clicking “ok”, you will be able to start jobs. Note that the “run” button is replaced with “submit”, and that the number of processes box is replaced by “number of nodes”.

Intel MPI

Select Intel MPI from the MPI implementation list. DDT has been tested with Intel MPI 2.0.

LAM/MPI

No reported issues with this distribution. Select LAM-MPI as the MPI implementation.

MPICH p4

No reported issues with this distribution, choose MPICH Standard as the MPI implementation.

MPICH p4 mpd

This daemon based distribution passes a limited set of arguments and environments to the job programs. If the daemons do not start with the correct environment for DDT to start, then the environment passed to the ddt-debugger backend daemons will be insufficient to start.

It should be possible to avoid these problems if .bashrc or .tcshrc/.cshrc are correct. However, if unable to resolve these problems, you can pass HOME and LD_LIBRARY_PATH, plus any other environment variables that you need, such as LM_LICENSE_FILE if you're using the Portland debugger, manually. This is achieved by adding -MPDENV- HOME={homedir} LD_LIBRARY_PATH={ld-library-path} to the "program arguments" area of the run dialog. Alternatively from the command-line you may simply write:

```
$DDT {program-name} -MPDENV- HOME=$HOME LD_LIBRARY_PATH=$LD_LIBRARY_PATH
```

and your shell will fill in these values for you.

Choose MPICH Standard as the MPI implementation.

MPICH-GM

No known issues.

MPICH SHMEM

This distribution intercepts standard input and output and redirects it to process 0. DDT will show all output as coming from process 0.

Choose MPICH SHMEM as the MPI implementation.

IBM PE

If you are able to use poe outside of a queuing system, set the environment variable DDTMPIRUN to the full pathname of poe. If your poe does not take the standard mpirun arguments (e.g. -np xx), it is advisable to write a wrapper script called mpirun which will invoke poe with the arguments you want.

In the present release of DDT, it is sometimes necessary to set the DDT_DONT_GET_RANK variable to 1 for MPI debugging. Without this, processes will not be able to start.

A sample Loadleveller script, which starts debugging jobs on IBM AIX (POE) systems is included in the {installation-directory}/templates directory. When working with Loadleveller, it is necessary to set the environment variable DDT_IGNORE_MPI_OUTPUT to 1.

In order to view source files it is important to have bash and gdb in your PATH. GDB is provided with the DDT distribution. Bash can be installed locally if not on your system, and is available from <ftp.gnu.org>.

Select IBM PE as the MPI implementation.

MVAPICH

Versions of the Mellanox IBGD prior to 1.8.1 do not work with debuggers, due to a kernel bug with Infiniband drivers. DDT has been tested successfully with IBGD 1.8.1 running mvapich 0.9.5. Select "Generic" as the MPI interface. You will need to specify the hosts on which to launch your job to mvapich's mpirun by using the '-hostfile filename' or individually as per the mvapich documentation in the "MPIrun arguments" box, which is available by pressing the "Advanced" button on the session startup dialog.

NEC MPI

Select Generic as the MPI implementation.

OpenMPI

DDT has been tested with OpenMPI 1.0.1. Select OpenMPI from the list of MPI implementations.

Message queues are not presently supported by this MPI.

Quadrics MPI

Select Generic as the MPI implementation.

SCore

DDT is supported by SCore versions 5.8.2 and above, some earlier versions can also support DDT by applying a minor patch. DDT can be launched either within a scout session, or using a queue.

For versions up to and including 5.8.2 a glitch in SCore prevents arguments being passed to programs during debugging; a patch is available for this which is easy to apply. Contact Allinea if this issue affects you.

There are several methods to start DDT on an SCore system and your administrator should recommend one for use with your cluster. Allinea recommend using a Sun GridEngine and provide a queue template file for this system. However, we have found the following method to work on single-user mode clusters:

1. Make sure your home directory is mounted on each cluster node
2. Create a host file containing a list of the computer nodes your intend to run the job on
3. Start a scout session: `scout -F host.list`
4. Start DDT at the prompt: `ddt`
5. Make sure DDT is configured for SCore mode, with the correct number of processes. Use the MPI Argument ``nodes=MxN`` to specify the number of processes per node and number of nodes, as documented for `scrunch`. Make sure to multiply these numbers when selecting the number of processes for DDT! Both must be specified for single-user mode Score systems
6. Click on ``Start``

Note that the first release of Score 5.6.0 shipped with a flaw in `scrunch.exe` – this prevents DDT shutting down a job correctly. The scout session must be closed and reopened between DDT sessions on these systems. This only affects single-user mode Score 5.6.0 installs.

If environment variables are not being propagated to remote nodes, we suggest moving `{installation-directory}/bin/ddt-debugger` to `{installation-directory}/bin/ddt-debugger.bin`, and creating a replacement executable shell script which sets the correct environment variables before running `ddt-debugger.bin` – for example:

```
#!/bin/sh
. ./bashrc
{installation-directory}/bin/ddt-debugger.bin $*
```

Choose SCore as the MPI implementation.

Note also that the number of processors chosen must equal the number of processors declared in the Scout host file; if you choose fewer, or more, the job will not start in DDT.

To **attach** to an SCore job, DDT can in some circumstances be blocked by the signals (SIGSTOP) which SCore sends to the user job. If this occurs, the progress bar will stop at some point whilst attaching and make no further progress. To

resolve the problem, send a SIGCONT to all processes in the job. This can be achieved by the following command:

```
doall kill SIGCONT -1
```

where “doall” is a command which executes the “kill” on every node in the job.

SCore typically links applications statically – which means that **memory debugging** is not enabled by default. To enable memory debugging, use the “-nostatic” option to the compiler (mpicc/mpif90).

Scyld

When running under Scyld, DDT starts all its ddt-debugger processes on the local machine instead of on the nodes. This is because Scyld represents the cluster as a single system image. For all but the largest clusters this should not be a problem. If this is an issue for you (insufficient file handles etc.) then contact Allinea for additional assistance.

The process details window will not show any hostnames when running under Scyld. This should not matter because Scyld represents a cluster as a single system image.

Choose Scyld as the MPI implementation.

SGI Altix

Early versions SGI's MP Toolkit can cause GDB to crash due to a library problem. This is easily resolved if it occurs. Compile your application with the extra linking flag “-lrt” and try DDT again.

Choose SGI MP Toolkit as the MPI implementation.

Sun Clustertools

DDT has been tested with Clustertools 5 for Solaris on SPARC and Clustertools 6 for Solaris on Opteron.

To run with more processes than are physically available on the cluster, you'll need to add:

```
-W
```

to the MPIRun Arguments box (click on Advanced in the Session Control window).

Choose Sun HPC as the MPI implementation.

E. Compiler Notes

Always compile with a minimal amount of, or no, optimization - some compilers reorder instruction execution and omit debug information when compiled with optimization turned on.

Some MPI implementations such as MPICH 1.2.5, require you to compile your code with the same compiler family as the implementation was compiled with. For example, if your copy of MPICH was compiled up using the Intel compilers, you should also compile your programs using the Intel compilers.

Absoft

No known issues.

DDT can debug Absoft code using GDB. Tested platforms are x86, AMD64.

GNU

The compiler flag “-fomit-frame-pointer” should never be used in an application which you intend to debug. Doing so will mean DDT cannot properly discover your stack frames and you will be unable to see which lines of code your program has stopped at!

For GNU C++, large projects can often result in vast debug information size, which can lead to large memory usage by DDT's backend debuggers – for example each instance of an STL class used in different object files will result in the compiler generating the same information in each object file. This large memory usage can result in excessive thrashing of the disks due to using swap memory – and this becomes even more noticeable for SMP systems where two processors or more share the same memory resource. For SMP systems, if a severe performance issue is noted whilst debugging with DDT we suggest forcing the MPI to use only one CPU per node. Allinea are happy to advise users of how to achieve this with most common MPIs.

IBM XLC/XLF

It is advisable to use the -qfullpath option to the IBM compilers (XLC/XLF) in order for source files to be found automatically by DDT when they are in directories other than that containing the executable.

Module data items behave differently between 32 and 64 bit mode, with 32-bit mode generally enabling access to more module variables than 64-bit mode.

DDT has been tested against the C compiler xlc version 7.0 and Fortran/Fortran 90 version 9.1 – on both Linux and AIX. Note that xlc (C++) is not fully supported on AIX.

Intel Compilers

The recommended debugger for this compiler is “automatic”. DDT has been tested with versions 7, 8.0, 8.1, and 9.0.

Known issues: IFC can generate unexpected location information for variables when compiled with “-save” option, leading to incorrect data values. It is recommended that the “-save” option is not used.

Some compiler optimizations performed when “-ax” options are specified to IFC/ICC can result in programs which cannot be debugged. This is due to the reuse by the compiler of the frame-pointer, which makes DDT unable to obtain a stack trace.

Pathscale EKO compilers

The recommended debugger for this compiler is “automatic”; DDT has been tested with version 2.2.1 and 2.3 of this compiler suite.

Notes for version 1.4: Fortran pointers are reported as their values, not addresses, so do not need to be dereferenced. This may also occur with later versions.

The default Fortran compiler options do not generate enough information for DDT to show where memory was allocated from – “view pointer details” will not show which line of source code memory was allocated from. To enable this, please use version 2.3 of the compiler and compile and link with the following flags:

-WL,--export-dynamic -TENV:frame_pointer=ON -funwind-tables

For C programs, simply compiling with -g is sufficient.

Portland Group Compilers

DDT has been tested with Portland Tools 4, 5.1-3, 5.2 and 6.0-8.

Known issues with Portland 5.1: Bounds for allocated arrays are incorrectly generated by PGF90. This issue is resolved in Portland’s 5.2 compiler suite.

Known issues with Portland 5.2 and 6.0: Assumed shape arrays can have wrong dimensions (Portland reference TPR 301). The “view pointer details” dialog cannot show which line of code memory was allocated from in Fortran codes with the 64-bit version of this compiler.

Known issues with Portland 6.0: Included files in Fortran 90 generate incorrect debug information with respect to file and line information. The information gives line numbers which refer to line numbers from the *included* file but give the *including* file as the file.

Sun Forte Compilers and Solaris DBX

When using DDT on Solaris, DDT will use the native DBX to debug each process in a parallel job. Some versions of DBX may prevent redirection of stdout/stderr, which means that DDT may not display the program output. If your version of DBX is affected then Allinea recommend writing output to a file instead.

Watches are not supported by this debugger in multi-threaded codes - which includes all parallel codes using Sun Clustertools - consequently DDT cannot support watches when using DBX.

DDT has been tested with Solaris 10 for Opteron, using Sun Studio 11 (DBX 7.5). Earlier versions can exhibit some fatal problems and we strongly recommend users upgrade to the most recent version available from Sun. If DDT fails to start your debugging session, please contact Allinea for advice.

At the time of writing, the current DBX version (7.5) treats Fortran allocatable arrays inside derived types unusually and this affect DDT's operation in the following ways:

- Allocatable arrays inside derived types are displayed as a hex address instead of an array. Drag these into the Evaluate window as a single item (e.g. "nested%array") to view the contents of the array, or right-click and use the MDA viewer.
- Very large allocatable arrays inside derived types cannot be limited in the Evaluate window at all – instead of displaying the first 200 (or so) elements DDT will not display any. Right-click on the array and use the MDA viewer to inspect its contents.

F. Architectures

This page notes any particular issues affecting platforms. If a supported machine is not listed on this page, it is because there is no known issue.

IBM AIX Systems

A sample Loadleveller script, which starts debugging jobs on IBM AIX (POE) systems is included in the {installation-directory}/templates directory.

AMD Opteron and Intel EM64T

When using a 64-bit operating system please note that it is essential to use the 64-bit version of DDT on this platform. This applies regardless of whether the debugged program is 32-bit or 64-bit.

Intel Itanium

Ptrace, the kernel's debugging interface, in early 2.4.x kernels did not support rapid reading of all registers in one single system call – and the Itanium has a lot of registers! Moreover, a kernel lock prevents multiple processors in an SMP Itanium from entering this system call simultaneously – and so it is possible to experience very slow debugging on **early** systems. Affected systems include Redhat 2.1 and 3 AS. SuSE 9.0 does not suffer from this.

A second error, corrected in late 2005 by 2.6.15.x and higher kernels concerned reading memory – some reads for MPI codes can take 10 times longer than an ordinary single processor job – due to an erroneous memory page searching algorithm. It is unlikely, but still plausible that users could observe this behaviour whilst debugging large data chunks.

For SGI Altix Itanium clusters, early versions SGI's MP Toolkit can cause GDB to crash due to a library problem. This is easily resolved. Compile your application with the extra linking flag "-lrt" and try DDT again.

Intel Xeon/Pentium 32 bit

No known issues.

Index

AIX.....	79, 89, 95
Align Stacks.....	37
Altix.....	79
Bproc.....	87
Breakpoints.....	34
Deleting A Breakpoint.....	35
Saving.....	35
Suspending Breakpoints.....	35
C++ STL.....	64
Configuration.....	6
Consistency Checking.....	
Heap.....	59
Cross-Process Comparison.....	49
Data.....	
Changing.....	46
Debugger.....	24, 26
Division by zero.....	42
dmalloc.....	58
Dynamic Libraries.....	30
FAQ.....	82
Fence Post Checking.....	60
Finding Code Or Variables.....	30
Floating Point Exception.....	42
Font.....	29
Hotkeys.....	33
HP MPI.....	87
IBM PE.....	89
Inf.....	42
Input.....	55
Installation.....	6
Intel Compilers.....	93
Irix.....	91
Jobs.....	
Cancelling.....	24
Regular Expression.....	15, 16, 24
Starting.....	24
Jump To Line.....	30
Double Clicking.....	32
LAM/MPI.....	88
Licence Server.....	74
Log File.....	85
Memory Debugging.....	58
Check Validity.....	60
Configuration.....	58
Leak Detection.....	61
Memory Statistics.....	62
Stop on Error.....	59
Message Queues.....	56
MPI rank.....	32
MPICH.....	
GM.....	88

p4.....	88
p4 mpd.....	88
Multi-Dimensional Arrays.....	47
MVAPICH.....	89
NEC MPI.....	89
OpenMP.....	5, 38
Opteron.....	79, 85
Parallel Stack View.....	39
performance.....	92
Pointers.....	46
Portland Group.....	93
Process.....	
Groups.....	32
Process Group.....	
Deleting.....	32
Pthreads.....	38
Quadrics MPI.....	89
Queue.....	
Configuring.....	13
Queuing.....	13
Raw Command.....	53
Registers.....	
Viewing.....	52
Restarting.....	33
SCore.....	18, 79, 90
Scyld.....	91
Segmentation fault.....	42
Session.....	
Saving.....	29
SGL.....	91
SIGFPE.....	42
SIGILL.....	42
Signal Handling.....	42
SIGPIPE.....	42
SIGSEGV.....	42
SIGUSR1.....	42
SIGUSR2.....	42
Single-Process.....	
Multi Process Licence.....	20
Single Process Licence.....	20
Solaris.....	18, 19, 79, 83, 84, 94
Solaris DBX.....	94
Source Code.....	29
Source Files.....	
Find Missing.....	29
Stack Frame.....	37
Standard Error.....	54
Standard Output.....	54
Starting.....	33
Starting DDT.....	17
Stepping Through A Program.....	34
Stopping.....	33
Sun Clustertools.....	79, 94
Support.....	16

Synchronizing Processes.....	36
Tab Sizes.....	29
Threads.....	
Examining.....	37
Visualizing Data.....	48
Wizard.....	64
Creating a custom wizard library.....	65
Using a custom wizard library.....	68