

Allinea DDT User Guide

Version 3.1-21691



Contents

Contents	1
1 Introduction	6
1.1 Licence Options	6
1.2 Obtaining Help	7
2 Installation and Configuration	8
2.1 Installation	8
2.1.1 Graphical Install	8
2.1.2 Text-mode Install	9
2.1.3 Licence Files	10
2.1.4 Floating Licences	10
2.2 Configuration	11
2.2.1 Site Wide Configuration	13
2.3 Integrating DDT With Queuing Systems	14
2.3.1 The Template Script	14
2.3.2 OpenMPI, Altix, Blue Gene/P and Cray MPT	15
2.3.3 MPICH based MPI	16
2.3.4 Other MPIs	16
2.3.5 Scalar Programs	16
2.3.6 Defining New Tags	16
2.3.7 Configuring Queue Commands	18
2.4 Connecting to remote programs (remote-exec)	19
2.4.1 SSH based remote-exec	19
2.4.2 Testing	20
2.5 Optional Configuration	20
2.5.1 System	20
2.5.2 Job Submission	21
2.5.3 Remote Launch	21
2.5.4 Appearance	21
2.5.5 Look & Feel	21
2.5.6 Code Viewer Settings	21
2.5.7 Override System Font Settings	22
3 Starting DDT	23
3.1 Running a Program	23
3.2 Notes on the MPICH Standard and OpenMPI options	26
3.3 Debugging Single-Process Programs	26
3.4 Debugging OpenMP Programs	27
3.5 Debugging Multi-Process Non-MPI programs	27
3.6 Debugging OpenMPI MPMD Programs	28
3.7 Opening Core Files	29
3.8 Attaching To Running Programs	29
3.8.1 Automatically Detected MPI Jobs	30
3.8.2 Attaching To A Subset Of A n MPI Job	30
3.8.3 Manual Process Selection	30
3.8.4 Configuring Attaching to Remote Hosts	31
3.8.5 Using LaunchMON for high-speed attaching	31
3.8.6 Using DDT Command-Line Arguments	32
3.9 Starting A Job In A Queue	32
3.10 Using Custom MPI Scripts	33
3.11 Starting DDT From A Job Script	35
3.12 Notes on X Forwarding or VNC for remote users	35
4 DDT Overview	37
4.1 Saving And Loading Sessions	38
4.2 Source Code	38
4.3 Finding Lost Source Files	38

4.4 Finding Code Or Variables	38
4.4.1 Find File or Function	38
4.4.2 Find	39
4.4.3 Find in Files	39
4.5 Jump To Line / Jump To Function	40
4.6 Static Analysis	41
4.7 Editing Source Code	41
5 Controlling Program Execution	42
5.1 Process Control And Process Groups	42
5.1.1 Detailed View	42
5.1.2 Summary View	43
5.2 Focus Control	43
5.2.1 Overview of changing focus	44
5.2.2 Process Group Viewer	44
5.2.3 Breakpoints	44
5.2.4 Code Viewer	44
5.2.5 Parallel Stack View	44
5.2.6 Playing and Stepping	44
5.2.7 Step Threads Together	44
5.2.8 Stepping Threads Window	45
5.3 Hotkeys	46
5.4 Starting , Stopping and Restarting a Program	46
5.5 Stepping Through A Program	46
5.6 Stop Messages	46
5.7 Setting Breakpoints	47
5.7.1 Using the Source Code Viewer	47
5.7.2 Using the Add Breakpoint Window	47
5.7.3 Pending Breakpoints	48
5.8 Conditional Breakpoints	48
5.9 Suspending Breakpoints	48
5.10 Deleting A Breakpoint	49
5.11 Loading And Saving Breakpoints	49
5.12 Default Breakpoints	49
5.12.1 Stop at exit/ exit	49
5.12.2 Stop at abort/fatal MPI Error	49
5.12.3 Stop on throw (C++ exceptions)	49
5.12.4 Stop on catch (C++ exceptions)	49
5.12.5 Stop at fork	49
5.12.6 Stop at exec	49
5.12.7 Stop on CUDA kernel launch	49
5.13 Synchronizing Processes	49
5.14 Setting A Watchpoint	50
5.15 Tracepoints	51
5.15.1 Setting a tracepoint	51
5.15.2 Tracepoint Output	51
5.16 Examining The Stack Frame	52
5.17 Align Stacks	52
5.18 “Where are my processes?” - Viewing Stacks in Parallel	52
5.18.1 Overview	52
5.18.2 The Parallel Stack View in Detail	53
5.19 Browsing Source Code	54
5.20 Simultaneously Viewing Multiple Files	56
5.21 Signal Handling	56
5.21.1 Sending Signals	56
6 Variables And Data	57
6.1 Current Line	57
6.2 Local Variables	57

6.3 Arbitrary Expressions And Global Variables	58
6.3.1 Fortran Intrinsic	58
6.3.2 Changing the language of an Expression	58
6.3.3 Macros and #defined Constants	59
6.4 Help With Fortran Modules	59
6.5 Viewing Complex Numbers in Fortran	60
6.6 C++ STL Support	60
6.7 Viewing Array Data	60
6.8 UPC Support	61
6.9 Changing Data Values	61
6.10 Viewing Numbers In Different Bases	61
6.11 Examining Pointers	61
6.12 Multi-Dimensional Arrays in the Variable View	61
6.13 Multi Dimensional Array Viewer	62
6.13.1 Array Expression	63
6.13.2 Filtering by Value	64
6.13.3 Distributed Arrays	64
6.13.4 Advanced: How Arrays Are Laid Out in the Data Table	64
6.13.5 Auto Update	66
6.13.6 Statistics	66
6.13.7 Export	66
6.13.8 Visualisation	66
6.14 Cross-Process and Cross-Thread Comparison	68
6.15 Assigning MPI Ranks	69
6.16 Viewing Registers	69
6.17 Interacting Directly With The Debugger	70
7 Program Input And Output	71
7.1 Viewing Standard Output And Error	71
7.2 Displaying Selected Processes	71
7.3 Saving Output	71
7.4 Sending Standard Input (DDT-MP)	71
8 Message Queues	73
8.1 Viewing The Message Queues	73
8.2 Interpreting the Message Queues	74
8.3 Deadlock	75
9 Memory Debugging	76
9.1 Enabling Memory Debugging	76
9.2 Configuration	76
9.2.1 Changing Settings at Run Time	78
9.3 Pointer Error Detection and Validity Checking	78
9.3.1 Library Usage Errors	78
9.3.2 Check Pointer Validity	79
9.3.3 View Pointer Details	79
9.3.4 Writing Beyond An Allocated Area	80
9.4 Current Memory Usage	80
9.4.1 Detecting Leaks when using Custom Allocators/Memory Wrappers	82
9.5 Memory Statistics	82
10 Checkpointing	84
10.1 What Is Checkpointing?	84
10.2 Checkpoint Support In DDT	84
10.3 How To Checkpoint	84
10.4 Restoring A Run-time Checkpoint	85
10.5 Restoring A Persistent Checkpoint	85
11 The Licence Server	86
11.1 Running The Server	86
11.2 Running DDT Clients	86
11.3 Logging	86

11.4 Troubleshooting	87
11.5 Adding A New Licence	87
11.6 Examples	87
11.7 Example Of Access Via A Firewall	88
11.8 Querying Current Licence Server Status	88
11.9 Licence Server Handling Of Lost DDT Clients	89
12 Using and Writing Plugins for DDT	90
12.1 Supported Plugins	90
12.2 Installing a Plugin	90
12.3 Using a Plugin	90
12.4 Example Plugin: MPI History Library	91
12.5 Writing a Plugin	92
13 CUDA GPU Debugging	94
13.1 Licensing	94
13.2 Preparing to Debug GPU Code	94
13.3 Launching the Application	94
13.4 Controlling GPU threads	94
13.4.1 Breakpoints	95
13.4.2 Stepping	95
13.4.3 Running and Pausing	95
13.5 Examining GPU Threads and Data	96
13.5.1 Selecting GPU Threads	96
13.5.2 Viewing GPU Thread Locations	96
13.5.3 Understanding Kernel Progress	97
13.5.4 Source Code Viewer	97
13.6 GPU Devices Information	97
13.7 Known Issues / Limitations	98
13.7.1 Using Multiple GPUs	98
13.7.2 Thread control	98
13.7.3 General	98
13.7.4 Pre sm_20 GPUs	99
13.7.5 Toolkit 3.1	100
13.8 GPU Language Support	100
13.8.1 CAPS HMPP	100
13.8.2 Cray OpenMP Accelerator Extensions	100
13.8.3 PGI Accelerators and CUDA Fortran	101
14 Offline Debugging	102
14.1 Using Offline Debugging	102
14.2 Offline Report Output (HTML)	103
14.3 Offline Report Output (Plain Text)	104
A Supported Platforms	105
B MPI Distribution Notes and Known Issues	106
B.1 Bproc	106
B.2 Bull MPI	106
B.3 HP MPI	106
B.4 Intel MPI	106
B.5 MPICH p4	107
B.6 MPICH p4 mpd	107
B.7 IBM PE	107
B.8 MVAICH	108
B.9 OpenMPI	108
B.10 Scyld	108
B.11 SGI Altix	108
B.12 Cray XT/XE/XK	109
14.3.1 Using DDT with Cray ATP (the Abnormal Termination Process)	109
B.13 ScaleMP	110
C Compiler Notes and Known Issues	111

C.1 Absoft	111
C.2 Berkeley UPC Compiler	111
C.3 Cray Compiler Environment	111
C.4 GNU	111
C.5 IBM XLC/XLF	111
C.6 Intel Compilers	112
C.7 Pathscale EKO compilers	113
C.8 Portland Group Compilers	113
C.9 SGI Compilers	114
D Platform Notes and Known Issues	115
D.1 GNU/Linux Systems	115
D.2 IBM AIX Systems	115
D.3 Intel Itanium	116
D.4 IBM Blue Gene/P	116
E General Troubleshooting and Known Issues	117
E.1 General Troubleshooting	117
E.1.1 Problems Starting the DDT GUI	117
E.1.1 Problems Starting Scalar Programs	117
E.1.1 Problems Starting Multi-Process Programs	117
E.2 Starting a Program	118
E.2.1 DDT says it can't find your hosts or the executable	118
E.2.2 The progress bar doesn't move and DDT 'times out'	118
E.2.3 The progress bar gets close to half the processes connecting and then stops and DDT 'times out'	119
E.3 Attaching	119
E.3.1 Running processes don't show up in the attach window	119
E.4 Source Viewer	119
E.4.1 No variables or line number information	119
E.4.2 Source code does not appear when you start DDT	119
E.5 Input/Output	119
E.5.1 Output to stderr is not displayed	119
E.6 Controlling a Program	120
E.6.1 Program jumps forwards and backwards when stepping through it	120
E.6.2 DDT sometimes stop responding when using the Step Threads Together option	120
E.7 Evaluating Variables	120
E.7.1 Some variables cannot be viewed when the program is at the start of a function	120
E.7.2 Incorrect values printed for Fortran array	120
E.7.3 Evaluating an array of derived types, containing multiple-dimension arrays	120
E.7.4 C++ STL types are not pretty printed	120
E.8 Memory Debugging	121
E.8.1 The View Pointer Details window says a pointer is valid but doesn't show you which line of code it was allocated on	121
E.8.2 "mprotect fails" error when using memory debugging with guard pages	121
E.8.3 Allocations made before or during MPI_Init show up in Current Memory Usage but have no associated stack back trace	121
E.8.4 Deadlock when calling printf or malloc from a signal handler	121
E.8.5 Program runs more slowly with Memory Debugging enabled	121
E.9 Message Queues	122
E.9.1 When viewing messages queues after attaching to a process you get a "Cannot find Message Queue DLL" error	122
E.10 Miscellaneous	122
E.10.1 The Fortran Module Browser is missing	122
E.10.2 Application working directory	122
E.11 Obtaining Support	122
F Index	124

1 Introduction

Allinea DDT is an intuitive, scalable, graphical debugger capable of debugging a wide variety of scenarios found in today's development environments. With Allinea DDT, it is possible to debug:

- Single process and multithreaded software
- OpenMP
- Parallel (MPI) software
- Heterogeneous software such as that written to use GPUs
- Hybrid codes mixing paradigms such as MPI + OpenMP, or MPI + CUDA
- Multi-process software of any form, including client-server applications.

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

Allinea DDT supports all of the compiled languages that are found in mainstream and high-performance computing including:

- C, C++, and all derivatives of Fortran, including F90.
- Parallel languages/models including MPI, UPC, and Fortran 2008 Coarrays.
- GPU languages such as HMPP, OpenMP Accelerators, CUDA and CUDA Fortran

Whilst many users choose Allinea DDT for desktop development or for debugging on small departmental parallel machines, it is also scalable and fast to beyond Petascale and is used to debug hundreds of thousands of processes simultaneously at some sites.

1.1 Licence Options

There are a number of different licence types, and the type will determine the scenarios for debugging that your Allinea DDT licence will make available.

- Workstation Scalar – for single process or multi-threaded code, including unlimited thread counts. Locked to a single workstation.
- Workstation Parallel – for single process, multi-threaded, multi-process or parallel code up to 8 distinct processes and unlimited thread counts. Locked to a workstation.
- Cluster – for all types of software, up to a defined process count and maximum number of concurrent users. The user interface is locked to one machine (but may still be X-forwarded) but the parallel processes may run on other machines.
- Supercomputing – a more flexible licence for all types of software, floating up to a defined total number of concurrent processes in use by multiple users concurrently.
- Extreme – our most flexible licence, able to support multiple architectures and floating similar to the Supercomputing licence.

Additionally, CUDA support is an option which can be added to any licence to allow debugging of GPU software for NVIDIA CUDA devices.

Evaluation licences contain support for all the features of Allinea DDT, but are limited to 16 processes.

1.2 Obtaining Help

Whilst this document attempts to cover as many parts of the installation, the features and the usage of our tool as possible, there will be scenarios or configurations that are not covered, or are only briefly mentioned, or you may on occasion experience a problem using the product. In any event, the support team at Allinea will be able to help and will look forward to assist in ensuring that you can get the most out of Allinea DDT.

You can contact the team by sending an email directly to support@allinea.com.

Please provide as much detail as you can about the scenario in hand, such as:

- Version number of Allinea DDT and your operating system and the distribution (example Linux Redhat 5.2), this information is all available by using the “-v” option to DDT on the command line:

```
bash$ ddt -v
```

```
Allinea DDT
© Allinea Software 2002-2011
```

```
Version: 3.1
Build: Ubuntu 10.04 x86_64
Build Date: Nov 15 2011
```

```
Licence Serial Number: 3936
Optional Features: none
Expires: Sat Dec 31 00:00:00 2011
Support Expires: Sat Dec 31 00:00:00 2011
```

```
Host: Ubuntu 11.04 x86_64
Nodes: unknown
Last connected ddt-debugger: unknown
```

- The compiler used and version number
- The MPI library, and CUDA toolkit version as appropriate.

2 Installation and Configuration

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

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

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 section 2.2.1 *Site Wide Configuration*). 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.

2.1 Installation

Allinea DDT may be downloaded from the Allinea website <http://www.allinea.com>. Follow the instructions below to install DDT.

2.1.1 Graphical Install

Untar the package and run the `installer` executable using the commands below.

```
gunzip < ddt3.1-21691-ARCH.tar | tar xvf -
./installer
```

The installer consists of a number of pages where you can choose install options. Use the *Next* and *Back* buttons to move between pages or *Cancel* to cancel the installation.

The *Install Type* page lets you choose who you want to install DDT for. If you are an administrator (`root`) you may install DDT for *All Users* in a common directory such as `/opt` or `/usr/local`, otherwise only the *Just For Me* option is enabled.

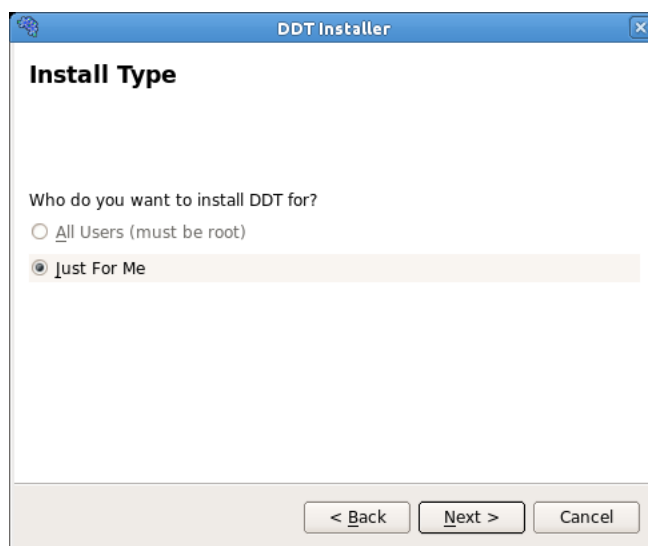


Fig 1: DDT Installer - Installation type

Once you have selected the installation type, you will be asked what directory you would like to install DDT in. 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.

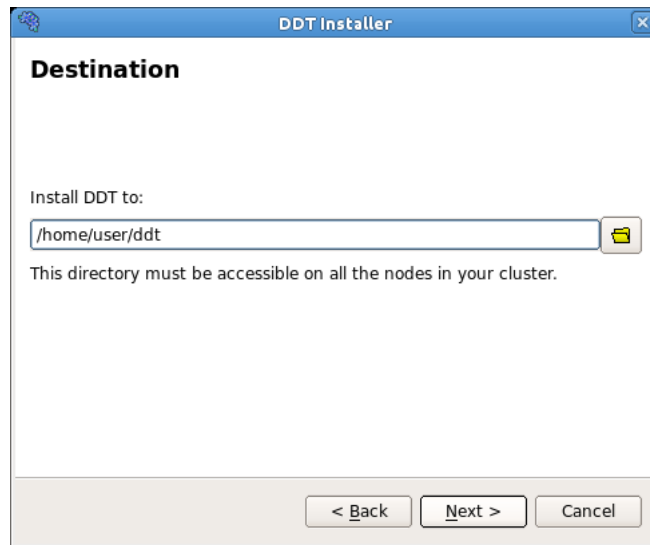


Fig 2: DDT Installer - Installation directory

You will be shown the progress of the installation on the *Install* page.

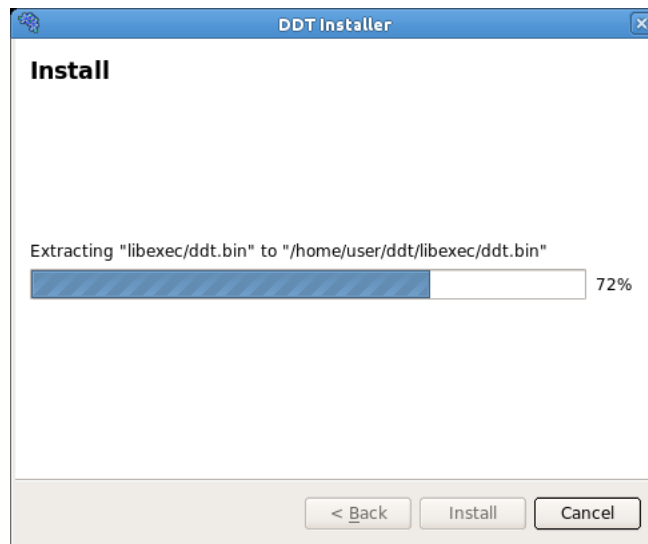


Fig 3: DDT Install in progress

An icon for DDT will be added to your desktop environment's *Development* menu or CDE's *Application Manager* .

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 www.allinea.com.

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

2.1.2 Text-mode Install

The text-mode install script `textinstall.sh` is useful if you are installing DDT remotely.

```
gunzip < ddt3.1-21691-ARCH.tar | tar xvf -
./textinstall.sh
```

Press Enter to read the licence when prompted and then enter the directory where you would like to install DDT. The directory must be accessible on all the nodes in your cluster.

2.1.3 Licence Files

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

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, <http://ww.allinea.com>.

2.1.4 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 host name, port and MAC (network) 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 section 11 *The Licence Server* .

2.2 Configuration

Allinea 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 will see the wizard.



Fig 4. Configuration Wizard

The Configuration Wizard helps you set DDT 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 *Next* and follow the simple instructions.

After the welcome page you will see the *MPI Implementation* page (this page is skipped if you only have a single process DDT licence - you can obtain a trial MPI licence from our website, www.allinea.com to see what you're missing).

The *System* page looks like this:

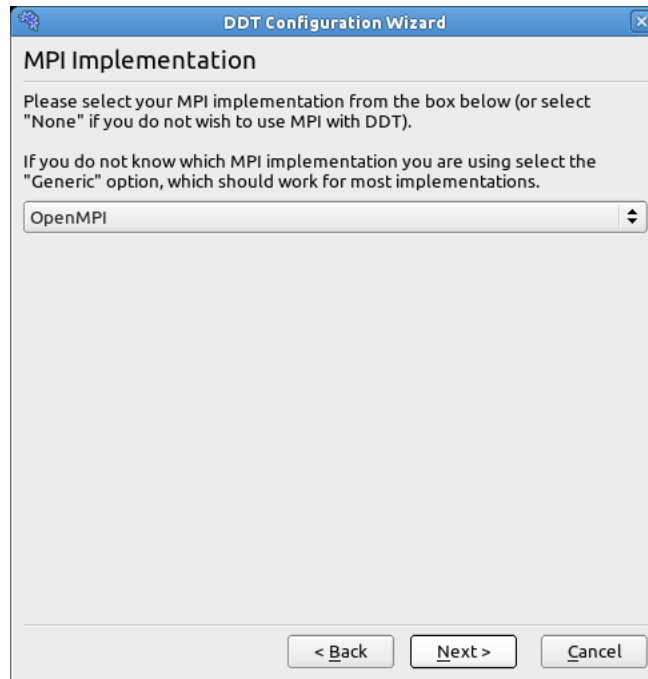


Fig 5: MPI Implementation Page

DDT will attempt to auto-detect and highlight your MPI implementation in the list, if this is not successful, please select your MPI implementation manually.

Once you have chosen or accepted an MPI Implementation, click on *Next*.

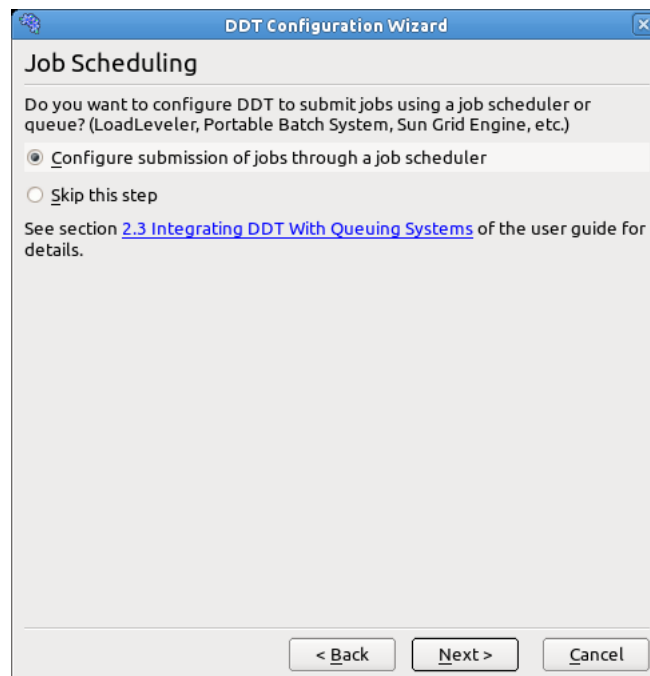


Fig 6 : Job Scheduling Page

The *Job Scheduling* page asks if you want to submit your jobs using a job scheduler or queue. If you are using a job scheduler such as LoadLeveler, Portable Batch System or Sun Grid Engine select the *Submit through a job scheduler* option, otherwise select *Run jobs directly* .

If you choose the *Submit through a job scheduler* option DDT will show the *Job Submission Settings* window. Click the folder icon next to the *Submission Template File* box to select a job submission template. If you use one of DDT's included templates the rest of the boxes (e.g. *Submit Command*) will be filled in automatically. The submission template may need modifying to match your environment. See section 2.3 *Integrating DDT With Queuing Systems* for details.

The final congratulatory page contains links to other optional configuration settings. You can click on one of the hyperlinks to open the relevant options page or help file.

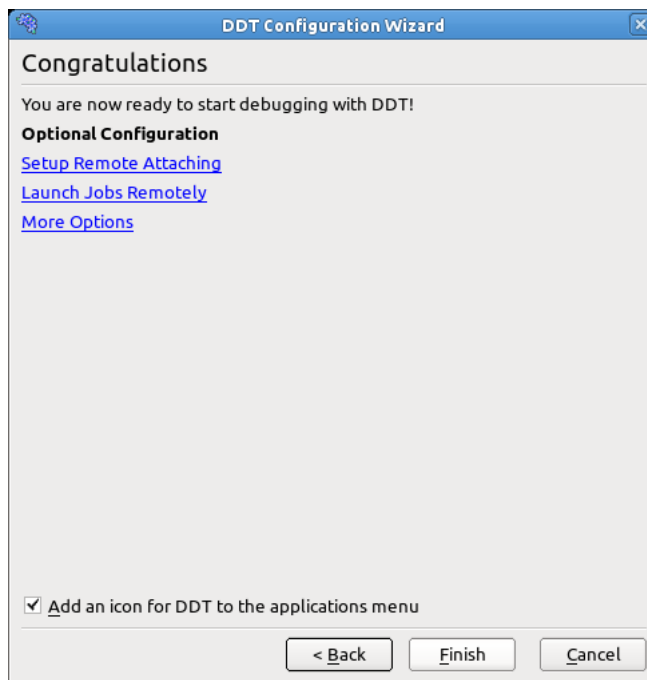


Fig 7: Configuration Wizard Complete

If you see a small red warning about attaching then you will not be able to attach to running programs or use the *Remote Launch* feature. Some MPI Implementations also require attaching to be configured (*MPICH 1 Standard* and *OpenMPI*) to start jobs.

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

Before you can attach to running programs you will also need to create a `nodes` file with a list of the compute nodes in your cluster. See section 3.8 *Attaching To Running Programs* for details.

2.2.1 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.

First configure DDT normally and run a test program to make sure all the settings are correct. When you are happy with your configuration execute the following command:

```
ddt -cleanconfig
```

This will remove any user-specific settings (such as the last program you ran) from your configuration file. Then copy the file to your DDT installation directory. This file will then be used as a template for all DDT users.

If you want to use DDT to attach to running jobs you will also need to create a file called `nodes` in the DDT installation directory with a list of compute nodes you want to attach to. See section 3.8 *Attaching To Running Programs* for details.

2.3 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 configuration 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.

2.3.1 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
MPIRUN_TAG	mpirun binary (can vary with MPI implementation)	/usr/bin/mpirun
AUTO_LAUNCH_TAG	Simple tag that can be used to generate the whole of the required line – ideal for scenarios where additional mpirun parameters are rarely set in script files	ddt-client /usr/bin/mpirun -np 4 myexample.bin
AUTO_MPI_ARGUMENTS_TAG	Required command line flags for mpirun (can vary with MPI implementation)	-np 4
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
NUM_PROCS_PLUS_ONE_TAG	Total number of processes + 1	17
NUM_NODES_TAG	Number of compute nodes	8
NUM_NODES_PLUS_ONE_TAG	Number of compute nodes + 1	9
PROCS_PER_NODE_TAG	Processes per node	2
PROCS_PER_NODE_PLUS_ONE_TAG	Processes per node + 1	3
NUM_THREADS_TAG	Number of OpenMP threads per node	4
OMP_NUM_THREADS_TAG	Number of OpenMP threads per node – or zero if OpenMP is “off”	4
EXTRA_MPI_ARGUMENTS_TAG	Extra mpirun arguments specified in the Run window	-partition DEBUG
WORKING_DIRECTORY_TAG	The working directory DDT was launched in	/users/ned
INPUT_FILE_TAG	The Input File specified in the Run window	/users/ned/input.dat
DDTPATH_TAG	The path to the DDT installation	/opt/allinea/ddt

Additionally, any environment variables in the GUI environment ending in _TAG are replaced throughout the script by the value of those variables.

2.3.2 OpenMPI, Altix, Blue Gene/P and Cray MPT

Ordinarily, your queue script will probably end in a line that starts `mpirun` with your target executable. You should prefix this line with `DDTPATH_TAG /bin/ddt-client`. For example, if your script currently has the line:

```
mpirun -np 16 program_name myarg1 myarg2
```

Then (for illustration only) the equivalent that DDT would need to use would be:

```
DDTPATH_TAG/bin/ddt-client mpirun -np 16 program_name myarg1 myarg2
```

Hence for the template script which will generate the line above when DDT submits your job, you use tags in place of the program name, arguments etc. so that they can be specified in the DDT GUI rather than editing the queue script each time. This leads to the following line – which should be used for these MPI versions:


```
DDTPATH_TAG/bin/ddt-client DDT_DEBUGGER_ARGUMENTS_TAG MPIRUN_TAG -np
NUM_PROCS_TAG EXTRA_MPI_ARGUMENTS_TAG PROGRAM_TAG PROGRAM_ARGUMENTS_TAG
```

Note: This advice is correct only for the Open MPI default mode, if you are using the “Open MPI (Compatibility)” mode – perhaps because SSH does not give access to the nodes – see the following “Other MPIs” subsection in this section and use the method given instead.

2.3.3 MPICH based MPI

Ordinarily, your queue script will probably end in a line that starts `mpirun` with your target executable.

```
mpirun -np 16 program_name myarg1 myarg2
```

To make this work with DDT you need to export the `TOTALVIEW` environment variable, and add the `-tv` parameter to `mpirun` . e.g.

```
export TOTALVIEW=DDTPATH_TAG/bin/ddt-debugger-mps
MPIRUN_TAG -np NUM_PROCS_TAG -tv PROGRAM_TAG PROGRAM_ARGUMENTS_TAG
```

2.3.4 Other MPIs

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 `DDTPATH_TAG/bin/ddt-debugger`. For example, if your script currently has the line:

```
mpirun -np 16 program_name myarg1 myarg2
```

You would write:

```
mpirun -np 16 DDTPATH_TAG/bin/ddt-debugger myarg1 myarg2
```

For a template script you use tags in place of the program name, arguments etc. so they can be specified in the DDT GUI rather than editing the queue script each time:

```
MPIRUN_TAG -np NUM_PROCS_TAG EXTRA_MPI_ARGUMENTS_TAG
DDTPATH_TAG/bin/ddt-debugger DDT_DEBUGGER_ARGUMENTS_TAG
PROGRAM_TAG
```

Note: don't include `PROGRAM_TAG` – `ddt-debugger` will launch your program for you.

2.3.5 Scalar Programs

To make your template script work for scalar programs you can add something similar to the following:

```
if [ $NUM_PROCS_TAG = 1 ]; then
    DDTPATH_TAG/bin/ddt-client DDT_DEBUGGER_ARGUMENTS_TAG PROGRAM_TAG
    PROGRAM_ARGUMENTS_TAGC
else
    mpirun -np NUM_PROCS_TAG DDTPATH_TAG/bin/ddt-debugger
    DDT_DEBUGGER_ARGUMENTS_TAG PROGRAM_ARGUMENTS_TAG
fi
```

2.3.6 Defining New Tags

As well as the pre-defined tags listed in the table above you can also define new tags in your template script whose values can be specified in the DDT GUI.

Tag definitions have the following format:

```
EXAMPLE_TAG: { key1=value1, key2=value2, ... }
```

Where key1, key2, ... are attribute names and value1, value2, ... are the corresponding values.

The tag will be replaced wherever it occurs with the value specified in the DDT GUI, for example:

#PBS -option EXAMPLE_TAG

The following attributes are supported:

Attribute	Purpose	Example
type	text: general text input select: select from two or more options check: a boolean option	type=text
label	The label for the user interface widget.	label="Account"
default	Default value for this tag	default="interactive"
mask	Input mask 0: ASCII digit permitted but not required. 9: ASCII digit required. 0-9. N: ASCII alphanumeric character required. A-Z, a-z, 0-9. n: ASCII alphanumeric character permitted but not required.	mask="09:09:09"
options	Options to use with the select tag type, separated by the character	options="not_shared shared"
checked	Value of a check tag if checked.	checked="enabled"
unchecked	Value of a check tag if unchecked.	unchecked="enabled"

Examples

```
# JOB_TYPE_TAG: {type=select,options=parallel|serial,label="Job
Type",default=parallel}
# WALL_CLOCK_LIMIT_TAG: {type=text,label="Wall Clock
Limit",default="00:30:00",mask="09:09:09"}
# NODE_USAGE_TAG: {type=select,options=not_shared|shared,label="Node
Usage",default=not_shared}
# ACCOUNT_TAG: {type=text,label="Account",global}
```

See the template files in *{installation-directory} /templates* for more examples.

To specify values for these tags click the *Edit Template Variables* button on the *Job Submission Options* page (see Fig 9 below) or the *Run* window. You will see a window similar to the one below:

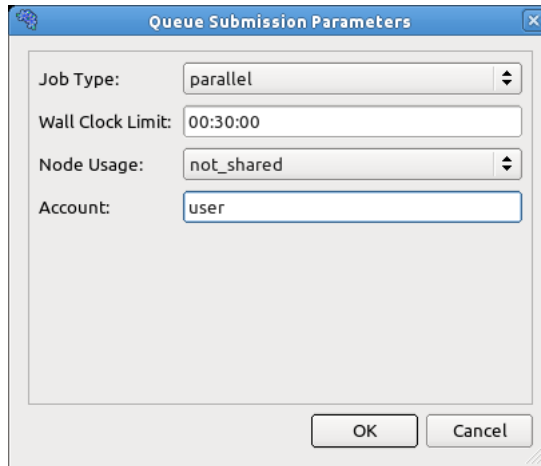


Fig 8: Queue Submission Parameters Window

The values you specify are substituted for the corresponding tags in the template file when you run a job.

2.3.7 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 file name 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.

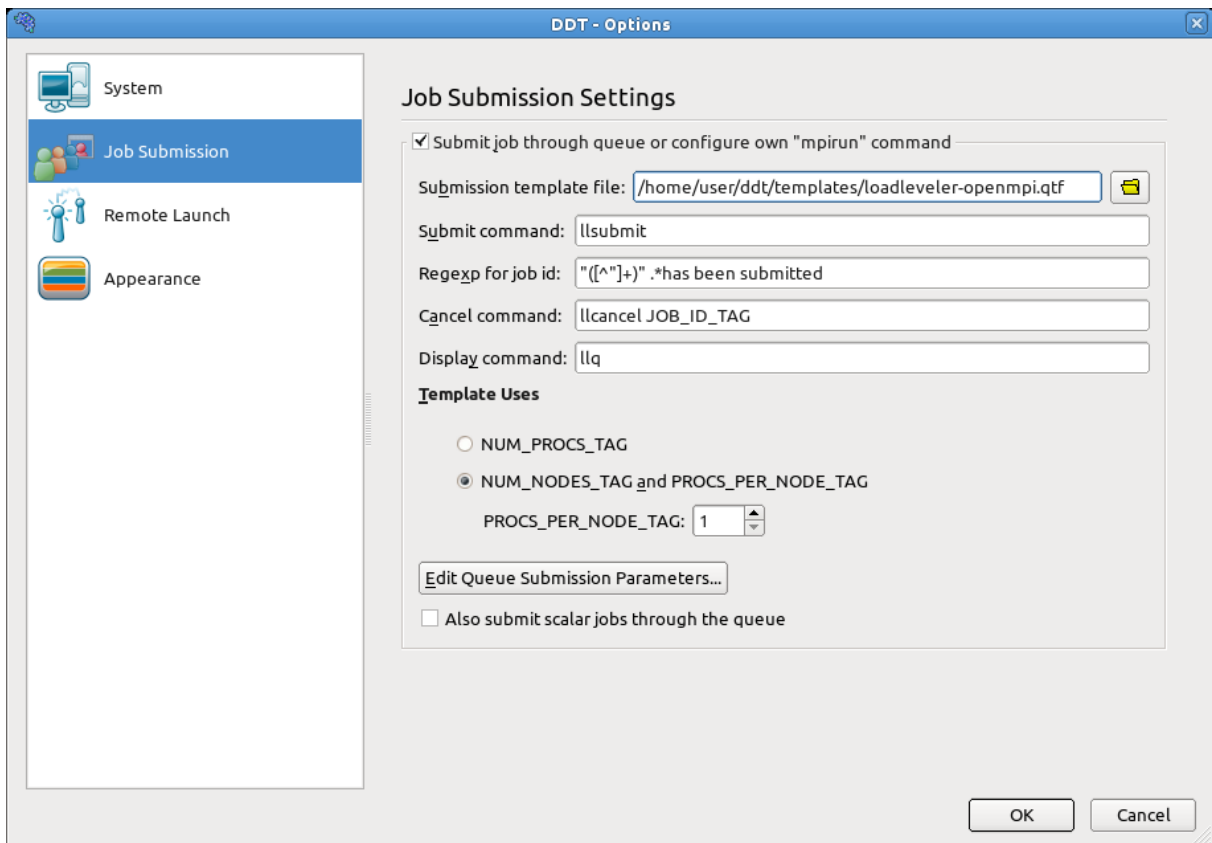


Fig 9: Queuing Systems

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 regular expression 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
<code>C</code>	A character represents itself
<code>\t</code>	A tab
<code>.</code>	Any character
<code>\d</code>	Any digit
<code>\D</code>	Any non-digit
<code>\s</code>	White space
<code>\S</code>	Non-white space
<code>\w</code>	Letters or numbers (a word character)
<code>\W</code>	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.

2.4 Connecting to remote programs (remote-exec)

When DDT needs to access another machine as part of starting some MPIs or of attaching to remote processes, it will attempt to use the secure shell, `ssh`, by default.

However, this may not always be appropriate, `ssh` may be disabled or be running on a different port to the normal port 22. In this case, you can create a file called "remote-exec" which is placed in your `~/ddt` directory and DDT will use this instead.

DDT will use look for the script at `~/ddt/remote-exec`, and it will be executed as follows:

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

The script should start `APPNAME` on `HOSTNAME` with the arguments `ARG1 ARG2` without further input (no password prompts). Standard output from `APPNAME` should appear on the standard output of `remote-exec`. An example is shown below:

2.4.1 SSH based remote-exec

A `remote-exec` script using `ssh` running on a non-standard port could look as follows:

```
#!/bin/sh
ssh -P {port-number} $*
```

In order for this to work without prompting for a password, you should generate a public and private SSH key, and ensure that the public key has been added to the `~/ .ssh/authorized_keys` file on machines you wish to use. See the `ssh-keygen` manual page for more information.

2.4.2 Testing

Once you have set up your `remote-exec` script, it is recommended that you test it from the command line. e.g.

```
~/ .ddt/remote-exec TESTHOST uname -n
```

Should return the output of `uname -n` on `TESTHOST`, without prompting for a password.

If you are having trouble setting up `remote-exec`, please contact support@allinea.com for assistance.

2.5 Optional Configuration

In addition to the configuration wizard, DDT also provides an options window, which allows you to quickly edit the settings in the configuration wizard, as well as other non-essential preferences. These options are outlined briefly below.

2.5.1 System

MPI Implementation: Allows you to tell DDT which MPI implementation you are using.

Note: If you are not using DDT to debug MPI programs select none.

Select Debugger: Tells DDT which underlying debugger DDT should use. This should almost always be left as *Automatic*.

Create Root and Workers groups automatically: If this option is checked DDT will automatically create a *Root* group for rank 0 and a *Workers* group for ranks 1... *n* when you start a new MPI session.

Use Shared Symbol Cache: The shared symbol cache is a file that contains all the symbols in your program in a format that can be used directly by the debugger. Rather than loading and converting the symbols itself, every debugger shares the same cache file. This significantly reduces the amount of memory used on each node by the debuggers. For large programs there may be a delay starting a job while the cache file is created as it may be quite large. The cache files are stored in `$HOME/ .ddt/symbols`. We recommend only turning this option on if you are running out of memory on compute nodes when debugging programs with DDT.

Default groups file: Entering a file here allows you to customise the groups displayed by DDT when starting an MPI job. If you do not specify a file DDT will create the default *Root* and *Workers* groups if the previous option is checked.

Note: A groups file can be created by right clicking the process groups panel and selecting Save groups... while running your program.

Attach hosts file: When attaching, DDT will fetch a list of processes for each of the hosts listed in this file. This option can also be configured using the *Configuration Wizard*. See section 3.8.4 *Configuring Attaching to Remote Hosts* for more details.

2.5.2 Job Submission

This section allows you to configure DDT to use a custom `mpirun` command, or submit your jobs to a queuing system. For more information on this, see section 2.3 *Integrating DDT With Queuing Systems* .

2.5.3 Remote Launch

This section allows you to configure DDT to launch scalar programs, or your `mpirun` command on a remote system. If this option is enabled, DDT will launch your scalar jobs on the specified remote system, rather than the local machine.

Configuring remote launch is not necessary if your `mpirun` command is run on the local machine (even if it launches remote processes itself).

Hostname: Hostname or IP address of the remote system to launch on.

Path to ddt-debugger: The full path to the `ddt-debugger` binary on the remote system. This

Once you have entered your Remote Launch settings, you can click the *Test Remote Launch* button to test your configuration.

Note: This feature requires that attaching is properly configured in DDT, and that your home directory is accessible by both the remote and local machines.

2.5.4 Appearance

This section allows you to configure the graphical style of DDT, as well fonts and tab settings for the code viewer.

2.5.5 Look & Feel

This determines the general graphical style of DDT. This includes the appearance of buttons, context menus.

2.5.6 Code Viewer Settings

This allows you to configure the appearance of the DDT code viewer (used to display your source code while debugging)

Tab size : Sets the width of a tab character in the source code display. (A width of 8 means that a tab character will have the same width as 8 space characters.)

Font name: The name of the font used to display your source code. It is recommended that you use a fixed width font.

Font size: The size of the font used to display your source code.

Editor : This is the program DDT will execute if you right click the code viewer and choose *Open file in editor*. This command should launch a graphical editor. If no editor is specified, DDT will attempt to launch a default editor (as configured in your desktop environment).

Warn about potential programming errors: This setting enables or disables the use of static analysis tools that are included with the Allinea DDT installation. These tools support F77, C and C+, and analyse the source code of viewed source files to discover common errors, but can cause heavy CPU usage on the system running the DDT user interface. You can disable this by unchecking this option.

2.5.7 Override System Font Settings

This setting can be used to change the font and size of all components in DDT (except the code viewer).

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
```

Note: 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 section 7 Program Input And Output .

Once DDT has started it will display the *Welcome Screen*.



Fig 10: Welcome Screen

The Welcome Screen allows you to choose what kind of debugging you want to do. You can:

- run a program from DDT and debug it
- debug a program you launch manually (e.g. on the command line)
- attach to an already running program
- open core files generated by a program that crashed
- restore a checkpoint of a program and continue debugging

3.1 Running a Program

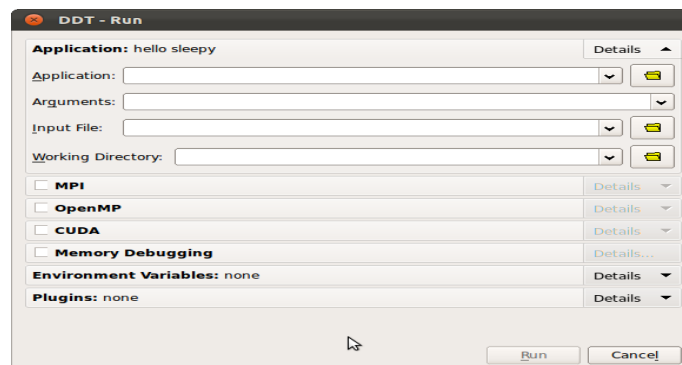



Fig 11: Run window

If you click the *Run* button on the Welcome Screen you will see the window above.

If you only have a single process licence or have selected *none* as your MPI Implementation the view will be more compact than shown above. The MPI options are not available when DDT is in single process mode. If you have a multiple process licence you can restore the full view by clicking the *Change...* button and selecting a different MPI implementation. See section 3.3 *Debugging Single-Process Programs* for more details about using DDT with a single process.

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 browse  button.

Note: Many MPIs have problems working with directory and program names containing spaces. We recommend avoiding the use of spaces in directory and file names.

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. Avoid using special characters such as ' and ", as these may be interpreted differently by DDT and your command shell. If you must use these and cannot get them to work as expected, please contact support@allinea.com.

If your licence supports it, you may also debug GPU programs by enabling CUDA support. For more information on debugging CUDA programs, please see section 13 *CUDA GPU Debugging*.

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 Allinea. A list of settings for common implementations is provided in Appendix B *MPI Distribution Notes and Known Issues*.

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 *details* button in the relevant section. The window will expand, and look like this:

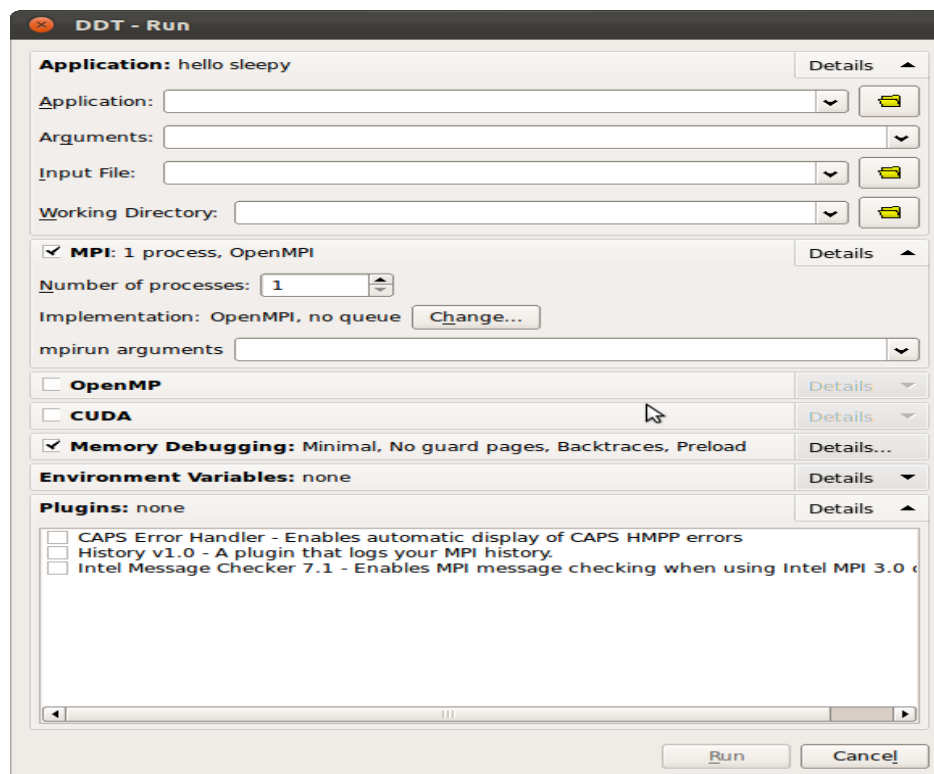


Fig 12: Advanced options

The MPIRun arguments box is for arguments that are passed to `mpirun` or your equivalent – 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 plugins box allows you to enable plugins for various third-party libraries, such as the Intel Message Checker or Marmot. See section 12 *Using and Writing Plugins for DDT* for more information.

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 give you the option to pause the processes so you can see how they got there. The default setting is to do nothing on a normal exit but stop if there is an MPI error or if `abort` is called. This allows you to inspect the program state after an error (but before MPI has terminated your job).

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 section 2.3 *Integrating 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.

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.

When your program starts, DDT will attempt to determine the MPI world rank of each process. If this fails, you will see the following error message:

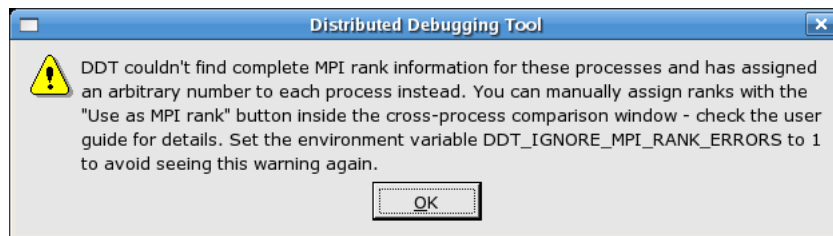


Fig 13: MPI rank error

This means that the number DDT shows for each process may not be the MPI rank of the process. To correct this you can tell DDT to use a variable from your program as the rank for each process – see section 6.15 *Assigning MPI Ranks* for details.

To end your current debugging session select the *End Session* menu option from the *Session* menu. 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).

3.2 Notes on the MPICH Standard and OpenMPI options

When using the *MPICH 1 Standard* or *OpenMPI* MPI implementations, DDT will allow mpirun to start all the processes, then attach to them while they're inside `MPI_Init`.

This method is often faster than the *generic* method, but requires the remote-exec facility in DDT to be correctly configured if processes are being launched on a remote machine. For more information on remote-exec, please see section 2.4 *Connecting to remote programs (remote-exec)* .

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.

3.3 Debugging Single-Process Programs

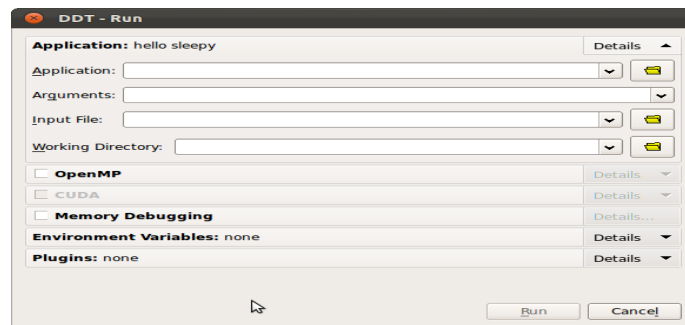



Fig 14: Single-Process Run Window

Users with single-process licences will immediately see the *Run Window* that is appropriate for single-process applications.

Users with multi-process licences can check the *Run Without MPI Support* checkbox to run a single process program.

Select the application – either by typing the file name in, or selecting using the browser by clicking the browse  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 (e.g. *Memory Debugging*).

Finally click *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 play to that breakpoint, or you can use the Step Into function to step into your code.

To end your current debugging session select the *End Session* menu option from the *Session* menu. This will close all processes and stop any running code.

3.4 Debugging OpenMP Programs

When running an OpenMP program, set the *Number of threads (OpenMP only)* value to the number of threads you require. DDT will run your program with the `OMP_NUM_THREADS` environment variable set to the appropriate value.

There are several important points to keep in mind while debugging OpenMP programs:

1. Some OpenMP libraries only create the threads when the first parallel region is reached. Don't worry if you can only see one thread at the start of the program.
2. You cannot step into a parallel region. Instead, tick the *Step threads together* box and use the *Run to here* command to synchronise the threads at a point inside the region – these controls are discussed in more detail in their own sections of this document.
3. You cannot step out of a parallel region. Instead, use *Run to here* to leave it. Most OpenMP libraries work best if you keep the *Step threads together* box ticked until you have left the parallel region. With the Intel OpenMP library, this means you will see the *Stepping Threads* window and will have to click *Skip All* once.
4. Leave *Step threads together* off when you are outside a parallel region (as OpenMP worker threads usually do not follow the same program flow as the main thread).
5. To control threads individually, use the *Focus on Thread* control. This allows you to step and play one thread without affecting the rest. This is helpful when you want to work through a locking situation or to bring a stray thread back to a common point. The Focus controls are discussed in more detail in their own section of this document.
6. Shared OpenMP variables may appear twice in the Locals window. This is one of the many unfortunate side-effects of the complex way OpenMP libraries interfere with your code to produce parallelism. One copy of the variable may have a nonsense value – this is usually easy to recognise. The correct values are shown in the Evaluate and Current Line windows.
7. Parallel regions may be displayed as a new function in the stack views. Many OpenMP libraries implement parallel regions as automatically-generated “outline” functions, and DDT shows you this. To view the value of variables that are not used in the parallel region, you may need to switch to thread 0 and change the stack frame to the function *you* wrote, rather than the outline function.
8. Stepping often behaves unexpectedly inside parallel regions. Reduction variables usually require some sort of locking between threads, and may even appear to make the current line jump back to the start of the parallel region! Don't worry about this – step over another couple of times and you'll see it comes back to where it belongs.
9. Some compilers optimise parallel loops regardless of the options you specified on the command line. This has many strange effects, including code that appears to move backwards as well as forwards, and variables that have nonsense values because they have been optimised out by the compiler.

If you are using DDT with OpenMP and would like to tell us about experiences, we would appreciate your feedback. Please email support@allinea.com with the subject title *OpenMP feedback*.

3.5 Debugging Multi-Process Non-MPI programs

DDT can only launch MPI programs and scalar (single process) programs itself. The *Debug a Multi-Process Non-MPI Program* button on the *Welcome Screen* allows you to debug multi-process and multi-executable programs. These programs don't necessarily need to be MPI programs. You can debug programs that use other parallel frameworks, or both the client and the server from a client/server application in the same DDT session, for example.

You must run each program you want to debug manually using the `ddt -client` command, similar to debugging with a scalar debugger like the GNU debugger (`gdb`). However, unlike a scalar debugger, you can debug more than one process at the same time in the same DDT session (licence permitting). Each program you run will show up as a new process in the DDT window.

For example to debug both client and server in the same DDT session:

1. Click on the Manual Launch button.
2. Select 2 processes.

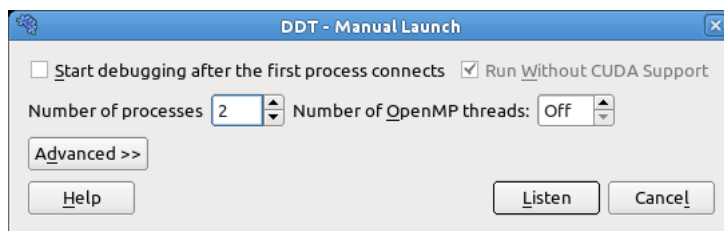


Fig 15: Manual Launch Window

3. Click the Listen button.
4. At the command line run:

```
ddt-client server &  
ddt-client client &
```

The server process will appear as process 0 and the client as process 1 in the DDT window.



Fig 16: Manual Launch Process Groups

After you have run the initial programs you may add extra processes to the DDT session (for example extra clients) using `ddt-client` in the same way.

```
ddt-client client2 &
```

If you check *Start debugging after the first process connects* you do not need to specify how many processes you want to launch in advance. You can start debugging after the first process connects and add extra processes later as above.

3.6 Debugging OpenMPI MPMD Programs

If you are using OpenMPI, DDT can be used to debug multiple program, multiple data (MPMD) programs. To start an MPMD program in DDT:

1. Create an application context file (as taken by the `--app` parameter to `mpirun`, see the OpenMPI documentation for more information).
2. Click the *Run* button on the Welcome Screen.
3. Click the *Advanced >>* button.
4. Type `--app /path/to/my_app_file` in the *MPIRun Arguments* box.
5. Click the Run button.

Note: it doesn't matter what executable you select in the Application box.

For MPIs other than OpenMPI MPMD programs are supported through the *Manual Launch* feature (see previous section).

3.7 Opening Core Files

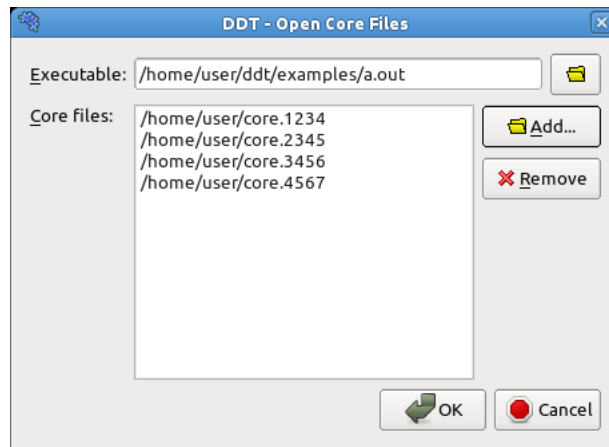


Fig 17: The Open Core Files Window

DDT allows you to open one or more core files generated by your application.

To debug using core files, click the *Open Core Files* button on the Welcome Screen. This opens the *Open Core Files* window, which allows you to select an executable and a set of core files. Click *Ok* to open the core files and start debugging them.

While DDT is in this mode, you cannot play, pause or step (because there is no process active). You are, however, able to evaluate expressions and browse the variables and stack frames saved in the core files.

The *End Session* menu option will return DDT to its normal mode of operation.

3.8 Attaching To Running Programs

DT 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. Clicking the *Attach to a Running Program* button on the Welcome Screen will show DDT's *Attach Window*:

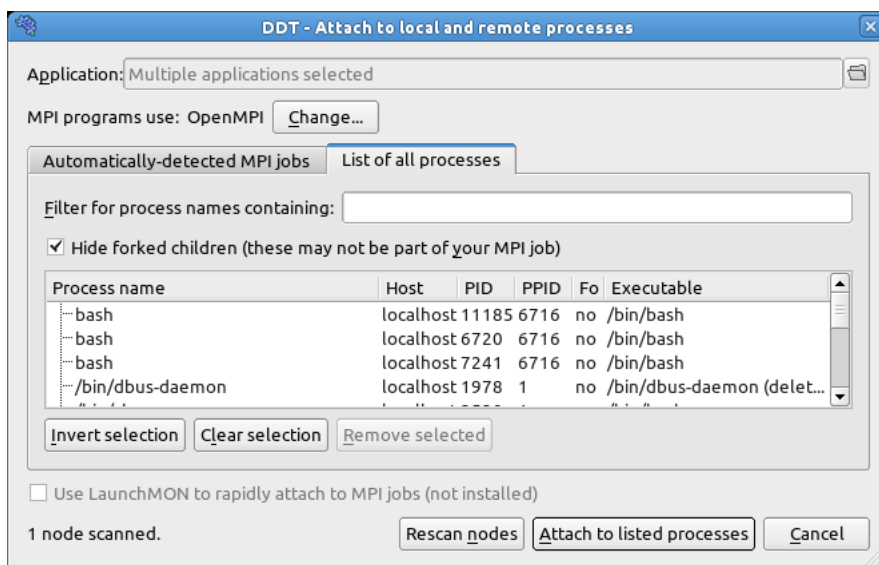


Fig 18 : Attach Window

There are two ways to select the processes you want to attach to: you can either choose from a list of automatically detected MPI jobs (for supported MPI implementations) or manually select from a list of processes.

3.8.1 Automatically Detected MPI Jobs

DDT can automatically detect MPI jobs started on the local host for selected MPI implementations (and other hosts you have access to if an *Attach Hosts File* is configured - see section 3.8.4 *Configuring Attaching to Remote Hosts* for more details).

The list of detected MPI jobs is shown on the *Automatically-detected MPI jobs* tab of the *Attach Window* . Click the header for a particular job to see more information about that job. Once you have found the job you want to attach to simply click the *Attach* button to attach to it.

3.8.2 Attaching To A Subset Of A n MPI Job

You may want to attach only to a subset of ranks from your MPI job. You can choose this subset using the *Attach to ranks* box on the *Automatically-detected MPI jobs* tab of the *Attach Window* . You may change the subset later by selecting the *Session* → *Change Attached Processes...* menu item.

3.8.3 Manual Process Selection

You can manually select which processes to attach to from a list of processes using the *List of all processes* tab of the *Attach Window* . If you want to attach to a process on a remote host see section 3.8.4 *Configuring Attaching to Remote Hosts* first.

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 window will appear as shown above. Use the Filter box to find the processes you want to attach to. On non-Linux platforms you will also need to select the application executable you want to attach to. 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.

On Linux you may use DDT to attach to multiple processes running different executables. When you select processes with different executables the application box will change to read *Multiple applications selected*. DDT will create a process group for each distinct executable

With some supported MPI implementations (e.g. OpenMPI) DDT will show MPI processes as children of the `mpirun` (or equivalent) command (see figure below). Clicking the `mpirun` command will automatically select all the MPI child processes.

Process name	Host	PID	PPID	Fo	Executable
/home/user/ddt/examples/hello	localhost	14779	14778	no	/home/user/ddt/exampl...
/home/user/ddt/examples/hello	localhost	14780	14778	no	/home/user/ddt/exampl...
/home/user/ddt/examples/hello	localhost	14781	14778	no	/home/user/ddt/exampl...
/home/user/ddt/examples/hello	localhost	14782	14778	no	/home/user/ddt/exampl...

Fig 19 : Attaching with OpenMP I

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. Some MPI implementations 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:

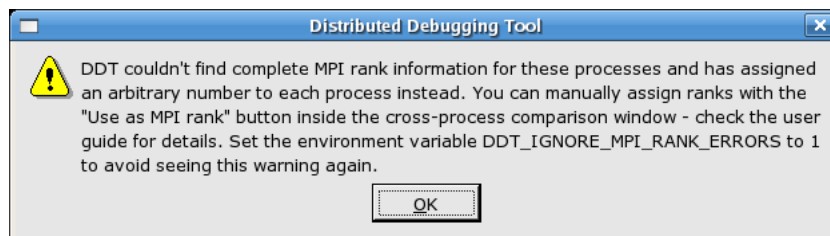


Fig 20: MPI rank error

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 Window* – see section 6.15 *Assigning MPI Ranks* 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).

3.8.4 Configuring Attaching to Remote Hosts

The easiest way to set up attaching for manual process selection is by following the instructions in the *Configuration Wizard*, described elsewhere in this document. However, if you wish to set up attaching manually, please see section 2.4 *Connecting to remote programs (remote-exec)* .

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* option 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.

3.8.5 Using LaunchMON for high-speed attaching

LaunchMON is an open-source library written by LLNL that enables HPC run-time tools to scalably place their daemons on the same nodes as a running job. If you have it installed, DDT can use LaunchMON to attach to large numbers of processes much more rapidly than by the default `rsync/ssh` method.

You can download LaunchMON from launchmon.sourceforge.net

Before using LaunchMON with DDT, please ensure that it is correctly configured and that its libraries are in your default `LD_LIBRARY_PATH`.

Once DDT detects the LaunchMON installation, the checkbox *Use LaunchMON to rapidly attach to MPI jobs* in the attach dialog becomes enabled. Simply tick it and then attach to an MPI job by selecting the 'mpirun' (often mpiexec or srun) process from the list. You can also attach to a subset of the processes by selecting them individually.

Non-MPI processes will not try to use the LaunchMON libraries even if this option is selected.

If you have problems using LaunchMON to attach to an MPI job you may wish to run the LaunchMON test programs to check that it is correctly configured for your system.

Please contact support@allinea.com if you are still experiencing trouble.

3.8.6 Using DDT Command-Line Arguments

As an alternative to starting DDT and using the Welcome Screen, 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.

3.9 Starting A Job In A Queue

If DDT has been configured to be integrated with a queue/batch environment, as described in section 2.3 *Integrating DDT With Queuing Systems* then you may use DDT to launch your job. In this case, a *Submit* button is presented on the *Run Window*, instead of the ordinary *Run* button. Clicking *Submit* from the *Run Window* 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.

3.10 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 start up.

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:

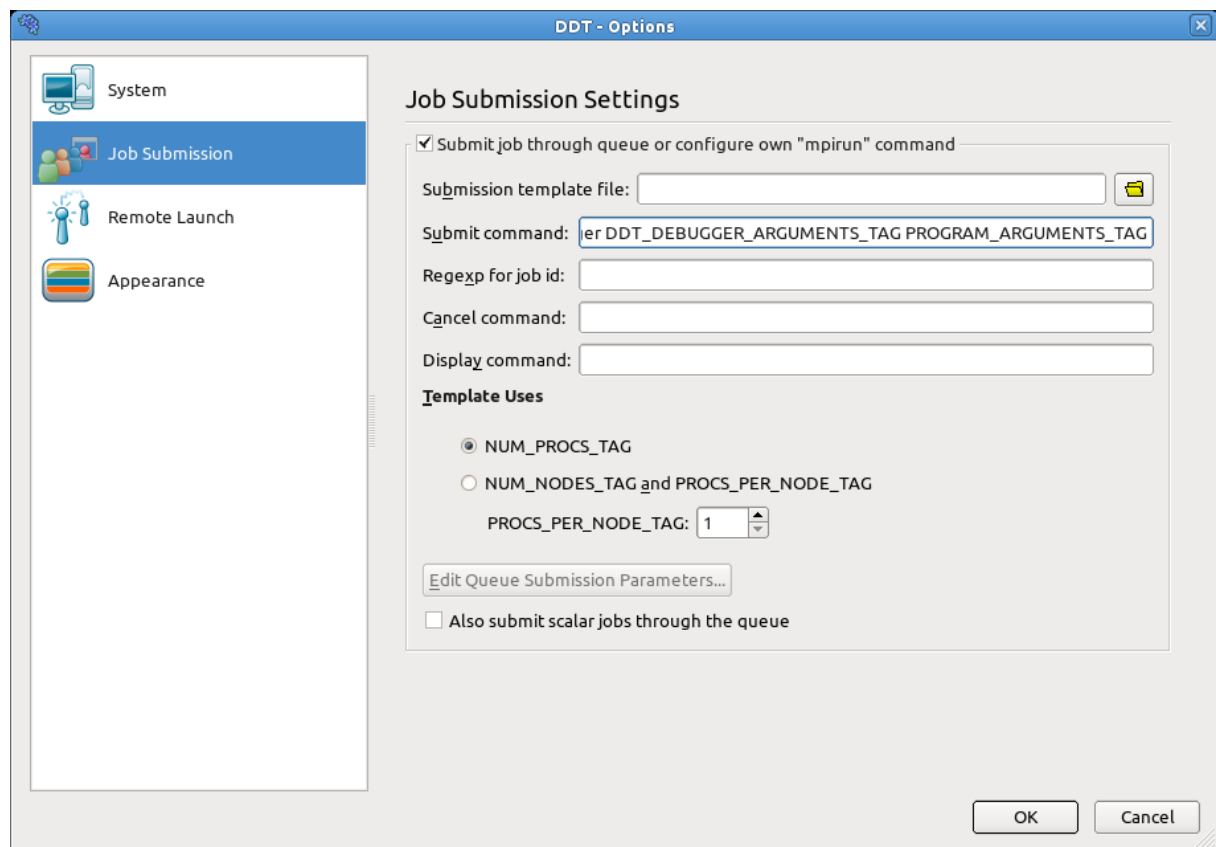


Fig 21: Using Custom MPI Scripts

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 path 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 configuration 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 : DDTPATH_TAG/bin/ddt-debugger  
DDT_DEBUGGER_ARGUMENTS_TAG PROGRAM_ARGUMENTS_TAG
```

This follows the same replacement rules described above and in detail in section 2.3 *Integrating DDT With Queuing Systems* . The options settings for this example might be:

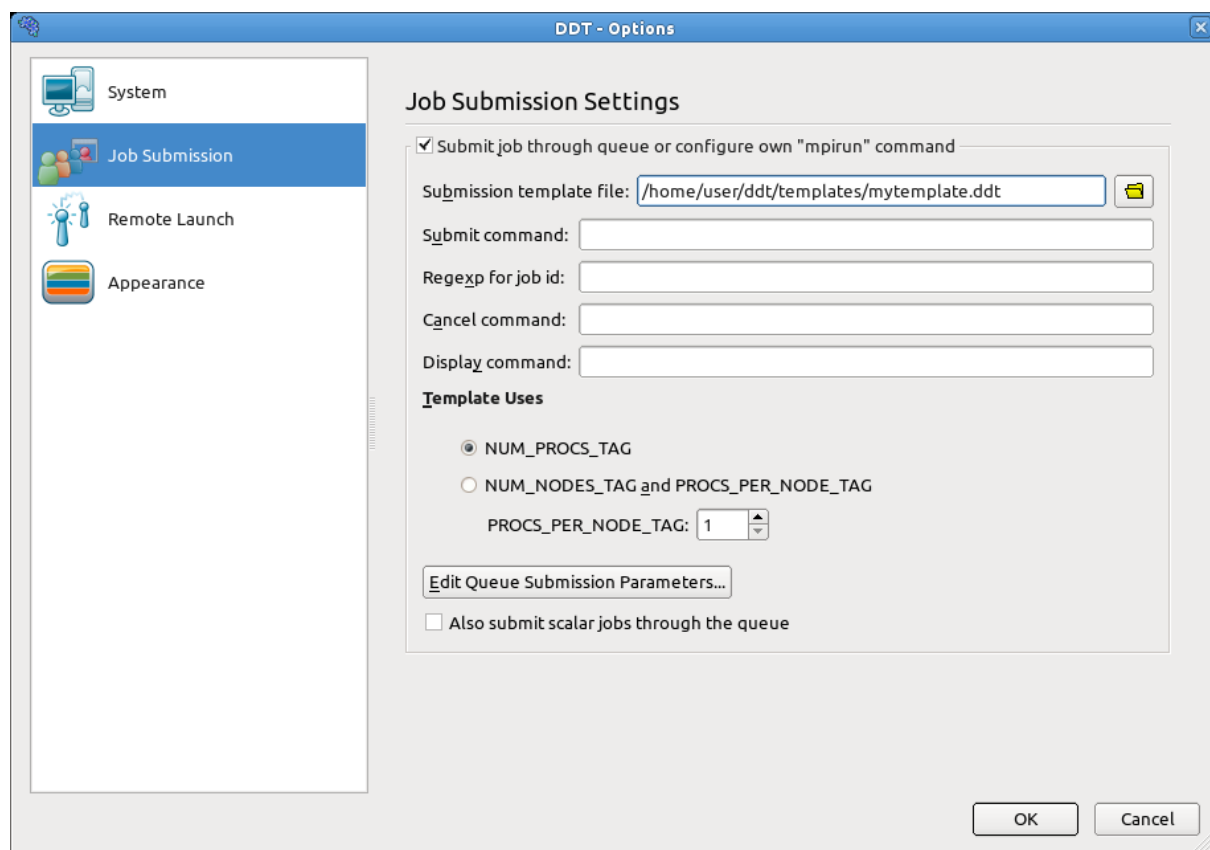


Fig 22: Using Substitute MPI Commands

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 file name 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 start up 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!

3.11 Starting DDT From A Job Script

The usual way of debugging a program with DDT in a queue/batch environment is to configure DDT to submit the program to the queue for you (see section 3.9 *Starting A Job In A Queue* above).

Some users may wish to start DDT itself from a job script that is submitted to the queue/batch environment. To do this:

1. Configure DDT with the correct MPI implementation.
2. Disable queue submission in the DDT options.
3. Create a job script that starts DDT using the command:

```
ddt -start -once -n NPROCS -- PROGRAM [ARGUMENTS]...
```

where NPROCS is the number of processes to start PROGRAM is the program to run and ARGUMENTS are the arguments to the program.

4. Submit the job script to the queue. The `-ONCE` argument tells DDT to exit when the session ends.

3.12 Notes on X Forwarding or VNC for remote users

Many users of DDT choose to display DDT on a machine that is not the actual cluster/machine that is running the DDT GUI, for example when accessing a departmental/facility cluster resource from their desktop/laptop. There are two methods for achieving this: X forwarding and VNC (or similar Unix-supporting remote desktop software).

X forwarding is effective when the network connection is very good. VNC is *strongly* recommended when the network connection is moderate or slow.

- Apple users accessing a Linux or other Unix machine whilst using a single-button mouse should be advised that pressing the *Command* key and the single mouse button will have the same effect as right clicking on a two button mouse. Right clicking is an important action in DDT: for example in each of the source code browser, the parallel stack view and variable views, right clicking will access some of the important features in DDT.

To use X forwarding and DDT with an Apple on a remote Linux/Unix system, start the X11 server (available in the `X11User . pkg`), then:

- Set the display variable correctly to allow X applications to display by opening a terminal in OS/X and typing

```
export DISPLAY=:0
```

- Then SSH to the remote system from that terminal, with SSH options `-X` and `-C` (X forwarding and compression). For example:

```
ssh -CX username@login.mybigcluster.com
```

- Now start DDT on the remote system and the display will appear on your Mac.
- Windows users can use any one of a number of commercial and open source X servers, but may find VNC a viable alternative (www.realvnc.com) which is available under free and commercial licensing options.

- VNC allows users to access a desktop running on a remote server (eg. a cluster frontend). By setting up an 'SSH tunnel' users are usually able to access this remote desktop from anywhere, securely, and, due to its own display protocols, will see a reduction in X11 traffic, which makes DDT very rapid to use remotely.

- To use VNC and DDT, log in to the remote system and set up a tunnel for port 5901 and 5801. On Apple or any Linux/Unix system this is set by options to SSH. For SSH using Putty on Windows, select options in the GUI to set tunnels.

```
ssh -L 5901:localhost:5901 -L 5801:localhost:5801  
username@login.mybigcluster.com
```

- At the remote prompt, start `vncserver`. If this is the first time you have used VNC it will ask you to set an access password.

vncserver

The output from `vncserver` will tell you which ports VNC has started on – $5800+n$ and $5900+n$, where n is the number given as `hostname:n` in the output. If this number, n , is not 1, then another user is already using VNC on that system, and you should set a new tunnel to these ports by logging in to the same host again and changing the settings to the new ports (or use SSH escape codes to add a tunnel, see the SSH manual pages for details).

- Now, on the local desktop/laptop, either use a browser and access the desktop within the browser by entering the URL <http://localhost:5801>, or (better) you may use a separate VNC client such as `krdc` or `vncviewer`.

```
krdc localhost:1 or vncviewer localhost:1
```

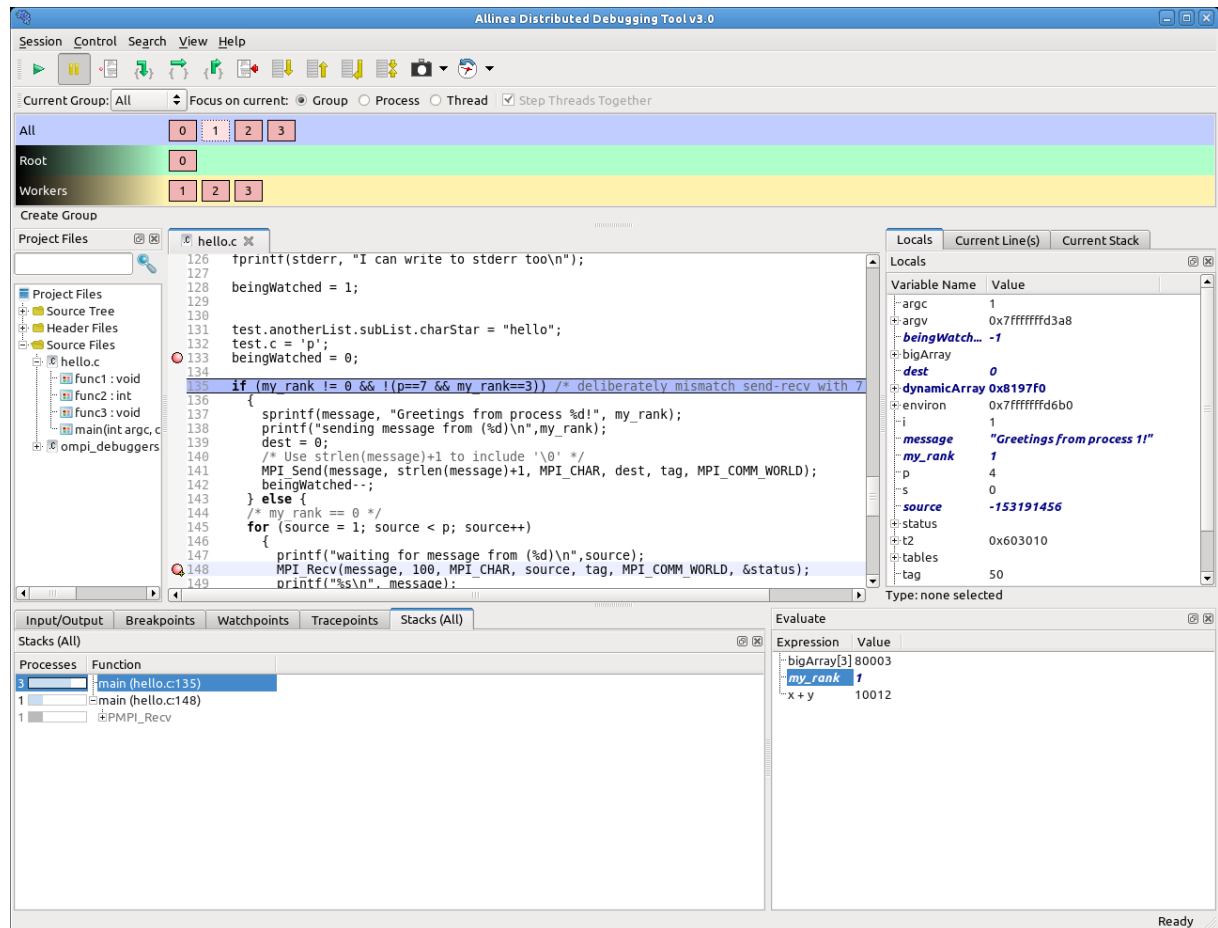
If n is not 1, as described above, use `:2`, `:3` etc. as appropriate instead.

- Note that a bug in the browser based access method means the `Tab` key does not work correctly in VNC. but `krdc` or `vncviewer` users are not affected by this problem.
- VNC also frequently defaults to an old X window manager (`twm`), this can be changed by editing the `~/ .vnc/xstartup` file to use KDE or GNOME and restarting the VNC server.

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 Viewer* is 'split') in the full workspace area.

Each component of DDT (labelled 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 Groups
(4) Find File or Function
(5) Project Navigator
(6) Source Code
(7) Variables and Stack of Current Process/Thread
(8) Parallel Stack, IO and Breakpoints
(9) Evaluate Window
(10) Status Bar

Fig 23: 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.

4.1 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 file name (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*.

4.2 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 file name.

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

4.3 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) file system – 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.

4.4 Finding Code Or Variables

4.4.1 Find File or Function

The *Find File Or Function Box* appears above the source file tree. You can type the name of a file or function in this box to search for that file/function in the source file tree. You can also type just part of a name to see all the files/functions whose name contains the text you typed. Double-click on a result to jump to the corresponding line for that file/function.

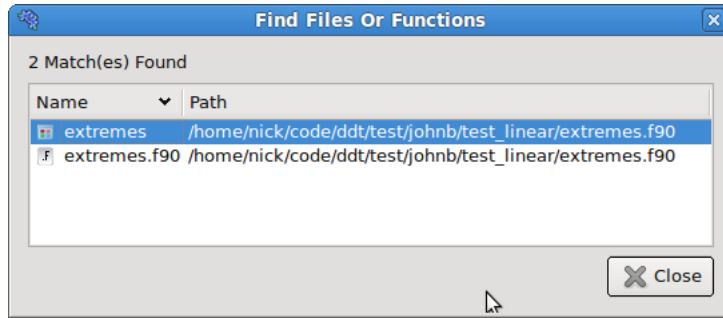


Fig 24: Find Files or Function box

4.4.2 Find

The *Find* window can be found in the *Search* menu, and can be used to find occurrences of an expression in the currently visible source file.

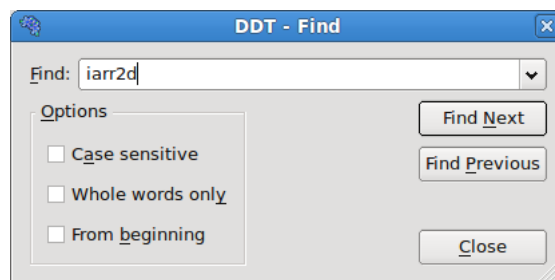


Fig 25: Find dialog

DDT will search from the current cursor position for the next or previous occurrence of the search term.

Case sensitive: When checked, DDT will perform a case sensitive search (e.g. Hello will not match hello).

Whole words only: When checked, DDT will only match your search term against whole 'words' in the source file. For example Hello would not match HelloWorld while searching for whole words only.

From beginning: When this is checked, your search will begin from the beginning of the document rather than the cursor.

4.4.3 Find in Files

The *Find In Files* window can be found in the *Search* menu, and can be used to search all source and header files associated with your program. The search results are listed and can be clicked to display the file and line number in the main *Source Code Viewer*; this can be of particular use for setting a breakpoint at a function.

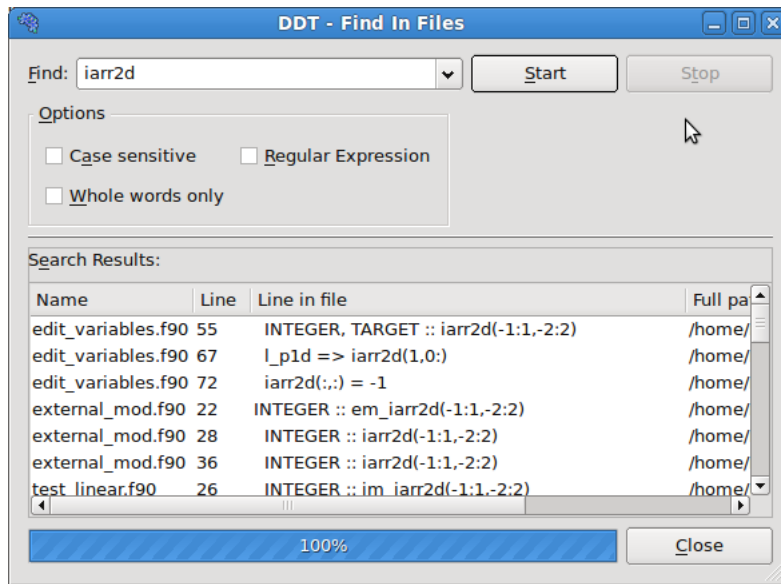


Fig 26: Find in Files dialog

Case sensitive: When checked, DDT will perform a case sensitive search (e.g. Hello will not match hello).

Whole words only: When checked, DDT will only match your search term against whole 'words' in the source file. For example Hello would not match HelloWorld while searching for whole words only.

Regular Expression : When checked, DDT will interpret the search term as a regular expression rather than a fixed string. The syntax of the regular expression is identical to that described in the section 2.3.7 *Configuring Queue Commands* .

4.5 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 window will appear in the centre 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:

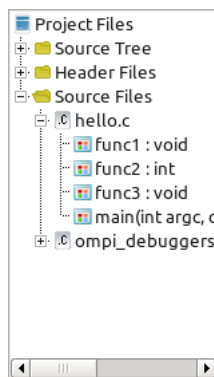


Fig 27: Function Listing

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.

4.6 Static Analysis

Static analysis is a powerful companion to debugging. Whilst Allinea DDT enables the user to discover errors by code and state inspection along with automatic error detection components such as memory debugging, static analysis inspects the source code and attempts to identify errors that can be detected from the source alone – independently of the compiler and actual process state.

Allinea DDT includes the static analysis tools `cppcheck` and `ftnchek`. These will by default automatically examine source files as they are loaded and display a warning symbol if errors are detected. Typical errors include:

- Buffer overflows – accessing beyond the bounds of heap or stack arrays
- Memory leaks – allocating memory within a function and there being a path through the function which does not deallocate the memory and the pointer is not assigned to any externally visible variable, nor returned.
- Unused variables, and also use of variables without initialization in some cases.

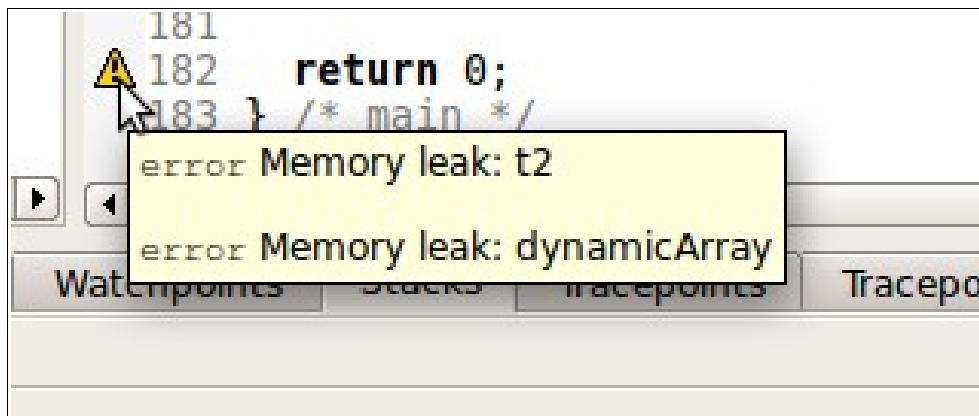


Fig 28: Static Analysis Error Annotation

Static analysis is not guaranteed to detect all, or any, errors, and an absence of warning triangles should not be considered to be an absence of bugs.

4.7 Editing Source Code

You can right click in the *Source Code Viewer* and select the *Open file in editor* option to open the current file in the default editor for your desktop environment. If you want to change the editor used, or the file doesn't open with the default settings, open the *Options* window by selecting the *Options* menu item from the *Session* menu and enter the path of your preferred editor in the *Editor* box, e.g. `/usr/bin/gedit`.

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 behaviours are identical, but apply to only a single process – freeing the user from concerns about process groups.

5.1 Process Control And Process Groups

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. Groups can be created, deleted, or modified by the user at any time, with the exception of the *All* group, which cannot be modified.

Groups are added by clicking on the *Create Group* button or 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, and you can create sub-groups from the processes currently playing, paused or finished. You can even create a sub-group excluding the members of another group – for example, to take the complement of the *Workers* group, select the *All* group and choose *Copy, but without Workers*.

You can also use the context menu to switch between the two different ways of viewing the list of groups in DDT – the detailed view and the summary view:

5.1.1 Detailed View

The detailed view is ideal for working with smaller numbers of processes. If your program has less than 32 processes, DDT will default to the detailed view. You can switch to this view using the context menu if you wish.

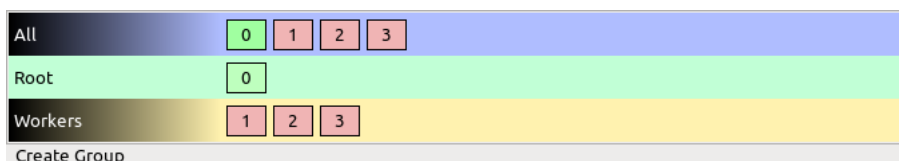


Fig 29 : The Detailed Process Group View

In the detailed view, each process is represented by a square containing its MPI rank (0 through n-1). The squares are colour-coded; red for a paused process, green for a playing process and grey for a finished/dead process. 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 *Source Code Viewer* jump to the file and line for the current stack frame (if available) by double-clicking on a process.

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 area select.

5.1.2 Summary View

The summary view is ideal for working with moderate to huge numbers of processes. If your program has 32 processes or more, DDT will default to this view. You can switch to this view using the context menu if you wish.

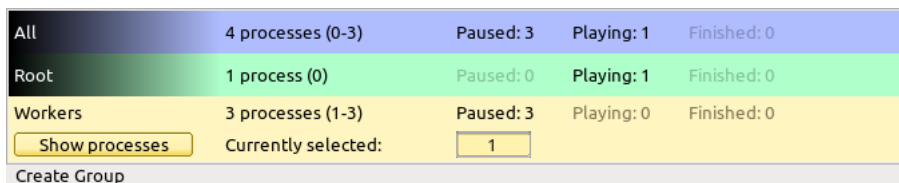


Fig 30 : The Summary Process Group View

In the summary view, individual processes are not shown. Instead, for each group, DDT shows:

- The number of processes in the group.
- The processes belonging that group – here *1-2048* means processes 1 through 2048 inclusive, and *1-10, 12-1024* means processes 1-10 and processes 12-1024 (but not process 11). If this list becomes too long, it will be truncated with a '...'. Hovering the mouse over the list will show more details.
- The number of processes in each state (playing, paused or finished). Hovering the mouse over each state will show a list of the processes currently in that state.
- The rank of the currently-selected process. You can change the current process by clicking here, typing a new rank and pressing Enter. Only ranks belonging to the current group will be accepted.

The *Show processes* toggle button allows you to switch a single group into the detailed view and back again – handy if you're debugging a 2048 process program but have narrowed the problem down to just 12 processes, which you've put in a group.

5.2 Focus Control



Fig 31: Focus options

The focus control allows you to focus on individual processes or threads as well as process groups. When focused on a particular process or thread, actions such as stepping, playing/pausing, adding breakpoints etc. will only apply to that process/thread rather than the entire group. In addition, the DDT GUI will change depending on whether you're focused on group, process or thread. This allows DDT to display more relevant information about your currently focused object.

5.2.1 Overview of changing focus

Focusing in DDT affects a number of different controls in the DDT main window. These are briefly described below:

Note: Focus controls do not affect DDT windows such as the Multi Dimensional Array Viewer, Memory Debugger, Cross Process Comparison etc.

5.2.2 Process Group Viewer

The changes to the process group viewer amongst the most obvious changes to the DDT GUI. When focus on current group is selected you will see your currently created process groups. When switching to focus on current process or thread you will see the view change to show the processes in the currently selected group, with their corresponding threads.

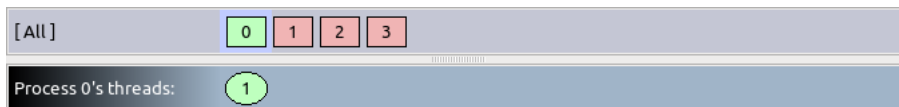


Fig 32: The Detailed Process Group View Focused on a Process

If there are 32 threads or more, DDT will default to showing the threads using a summary view (as in the Process Group View). The view mode can also be changed using the context menu.

5.2.3 Breakpoints

The breakpoints tab in DDT will be filtered to only display breakpoints relevant to your current group, process, thread. When focused on a process, The breakpoint tab will display which thread the breakpoint belongs to. If you are focused on a group, the tab will display both the process and the thread the breakpoint belongs to.

5.2.4 Code Viewer

The code viewer in DDT shows a stack back trace of where each thread is in the call stack. This will also be filtered by the currently focused item, for example when focused on a particular process, you will only see the back trace for the threads in that process.

Also, when adding breakpoints using the code viewer, they will be added for the group, process or thread that is currently focused.

5.2.5 Parallel Stack View

The parallel stack view can also be filtered by focusing on a particular process group, process or thread.

5.2.6 Playing and Stepping

The behaviour of playing, stepping and the *Run to here* feature are also affected by your currently focused item. When focused on a process group, the entire group will be affected, whereas focusing on a thread will mean that only current thread will be executed. The same goes for processes, but with an additional option which is explained below.

5.2.7 Step Threads Together

The step threads together feature in DDT is only available when focused on process. If this option is enabled then DDT will attempt to synchronise the threads in the current process when performing actions such as stepping, pausing and using *Run to here*.

For example, if you have a process with 2 threads and you choose *Run to here*, DDT will pause your program when either of the threads reaches the specified line. If *Step threads together* is selected DDT will attempt to play both of the threads to the specified line before pausing the program.

Important note: You should always use *Step threads together* and *Run to here* to enter or move within OpenMP parallel regions. With many compilers it is also advisable to use *Step threads together* when leaving a parallel region, otherwise threads can get 'left behind' inside system-specific locking libraries and may not enter the next parallel region on the first attempt.

5.2.8 Stepping Threads Window

When using the step threads together feature it is not always possible for all threads to synchronise at their target. There are two main reasons for this:

1. One or more threads may branch into a different section of code (and hence never reach the target). This is especially common in OpenMP codes, where worker threads are created and remain in holding functions during sequential regions.
2. As most of DDT's supported debug interfaces cannot play arbitrary groups of threads together, DDT simulates this behaviour by playing each thread in turn. This is usually not a problem, but can be if, for example, thread 1 is playing, but waiting for thread 2 (which is not currently playing). DDT will attempt to resolve this automatically but cannot always do so.

If either of these conditions occur, the Stepping Threads Window will appear, displaying the threads which have not yet reached their target.



Fig 33: The Stepping Threads Window

The stepping threads window also displays the status of threads, which may be one of the following:

- **Done:** The thread has reached its target (and has been paused).
- **Skipped:** The thread has been skipped (and paused). DDT will no longer wait for it to reach its target.
- **Playing:** This is the thread that is currently being executed. Only one thread may be playing at a time while the Stepping Threads Window is open.
- **Waiting:** The thread is currently awaiting execution. When the currently *playing* thread is *done* or has been skipped, the highest *waiting* thread in the list will be executed.

The stepping threads window also lets you interact with the threads with the following options:

- **Skip:** DDT will skip and pause the currently playing thread. If this is the last waiting thread the window will be closed.
- **Try Later:** The currently playing thread will be paused, and added to the bottom of the list of threads to be retried later. This is useful if you have threads which are waiting on each other.

- **Skip All:** This will skip (and pause) all of the threads and close the window.

5.3 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	Go to line number
CTRL-F	Find



5.4 Starting , Stopping and Restarting a Program

The *Session* menu 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 start up 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 window 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 shut down 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.

5.5 Stepping Through A Program

To continue the program playing 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. The return value of the function is displayed in the *Locals* view.

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

5.6 Stop Messages

In certain circumstances your program may be automatically paused by the debugger. There are five reasons your program may be paused in this way:

1. It hit one of DDT's default breakpoints (e.g. `exit` or `abort`). See section 5.12 *Default Breakpoints* for more information on default breakpoints.
2. It hit a user-defined breakpoint (a breakpoint shown in the *Breakpoints* view).
3. The value of a watched variable changed.
4. It was sent a signal. See section 5.21 *Signal Handling* for more information on signals.
5. It encountered a Memory Debugging error. See section Error: Reference source not found *Library Usage Errors* for more information on Memory Debugging errors.

DDT will display a message telling you exactly why the program was paused. The text may be copied to the clipboard by selecting it with the mouse, then right-clicking and selecting *Copy*. You may want to suppress these messages in certain circumstances, for example if you are playing from one breakpoint to another. Use the *Control* → *Messages* menu to enable/disable stop messages.

5.7 Setting Breakpoints

5.7.1 Using the Source Code Viewer

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* window. Clicking the right mouse button in the *Source Code Viewer* 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.

5.7.2 Using the Add Breakpoint Window

You can also add a breakpoint by clicking the *Add Breakpoint*  icon in the toolbar. This will open the *Add Breakpoint* window.

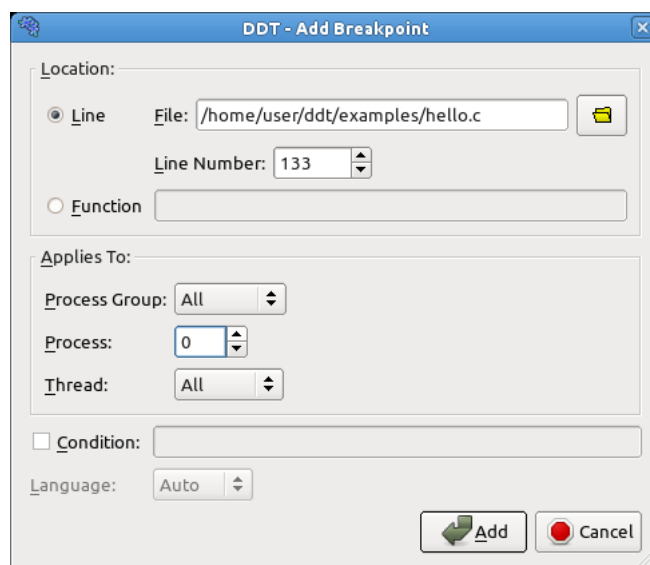


Fig 34: The Add Breakpoint window

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. Select the *Function* radio button and enter the name of the function in the box next to it.

You can specify what group/process/thread you want the breakpoint to apply in the *Applies To* section. You may also make the breakpoint conditional by checking the *Condition* check box and entering a condition in the box.

5.7.3 Pending Breakpoints

Note: This feature is not supported on all platforms.

If you try to add a breakpoint on a function that is not defined, DDT will ask if you want to add it anyway. If you click *Yes* the breakpoint will be applied to any shared objects that are loaded in the future.

5.8 Conditional Breakpoints

	Processes	Threads	File	Line	Function	Condition	Full path
<input checked="" type="checkbox"/>	process 0	all	hello.c	133			/home/user/ddt/examples/hello.c
<input checked="" type="checkbox"/>	All	all	hello.c	148		my_rank == 3	/home/user/ddt/examples/hello.c

Fig 35: 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.

	Processes	Threads	File	Line	Function	Condition	Full path
<input checked="" type="checkbox"/>	All	all	hello.f	55			/home/
<input checked="" type="checkbox"/>	All	all	hello.f	49		my_rank .EQ. 0	/home/

Fig 36: Conditional Breakpoints In Fortran

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

5.9 Suspending Breakpoints

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

5.10 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.

5.11 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.

5.12 Default Breakpoints

DDT has a number of default breakpoints that will stop your program under certain conditions which are described below. You may enable/disable these while your program is running using the *Control* → *Default Breakpoints* menu.

5.12.1 Stop at exit/_exit

When enabled, DDT will pause your program as it is about to end under normal exit conditions. DDT will pause both before and after any exit handlers have been executed. (Disabled by default.)

5.12.2 Stop at abort/fatal MPI Error

When enabled, DDT will pause your program as it about to end after an error has been triggered. This includes MPI and non-MPI errors. (Enabled by default.)

5.12.3 Stop on throw (C++ exceptions)

When enabled, DDT will pause your program whenever an exception is thrown (regardless of whether or not it will be caught). Due to the nature of C++ exception handling, you may not be able to step your program properly at this point. Instead, you should play your program or use the *Run to here* feature in DDT. (Disabled by default.)

5.12.4 Stop on catch (C++ exceptions)

As above, but triggered when your program catches a thrown exception. Again, you may have trouble stepping your program. (Disabled by default.)

5.12.5 Stop at fork

DDT will stop whenever your program forks (i.e. calls the `fork` system call to create a copy of the current process). The new process is added to your existing DDT session and can be debugged along with the original process.

5.12.6 Stop at exec

When your program calls the `exec` system call, DDT will stop at the main function (or program body for Fortran) of the new executable.

5.12.7 Stop on CUDA kernel launch

When debugging CUDA GPU code, this will pause your program at the entry point of each kernel launch.

5.13 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 plays all the processes in the selected group 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 play 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 play. Be sure that your process will revisit the line, or alternatively synchronize to the line immediately after the current line.

5.14 Setting A Watchpoint

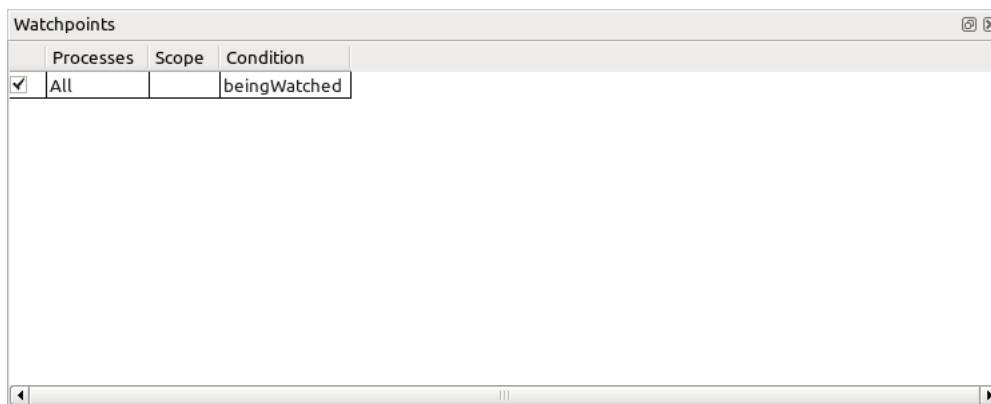


Fig 37: The Watches Table

You can set a watchpoint on a variable or expression that causes DDT to stop every time it changes.

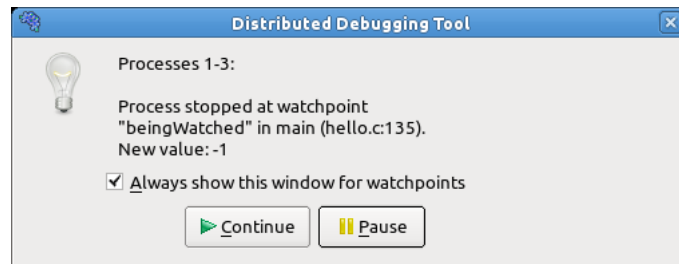


Fig 38: Program Stopped At Watchpoint beingWatched

Unlike breakpoints, watchpoints are not displayed in the *Source Code Viewer*. Instead they are created by right-clicking on the *Watchpoints* view and selecting the *Add Watchpoint* menu item, or dragging a variable to the *Watchpoints* view from the *Local Variables*, *Current Line* or *Evaluate* views.

You can set a watchpoint for either a single process, or every process in a process group.

DDT will automatically remove a watchpoint once the target variable goes out of scope. If you are watching the value pointed to by a variable, i.e. `*p` - you may want to continue watching the value at that address even after `p` goes out of scope. You can do this by right-clicking on `*p` in the *Watches* view and selecting the *Pin to address* menu item. This replaces the variable `p` with its address so the watch will not be removed when `p` goes out of scope.

5.15 Tracepoints

Tracepoints allow you to see what lines of code your program is executing – and the variables – without stopping it. Whenever a thread reaches a tracepoint it will print the file and line number of the tracepoint to the Input/Output view. You can also capture the value of any number of variables or expressions at that point.

Examples of situations in which this feature will prove invaluable include

- Recording entry values in a function that is called many times, but crashes only occasionally. Setting a tracepoint makes it easier to correlate the circumstances that cause a crash.
- Recording entry to multiple functions in a library – enabling the user (or library developer) to check which functions are being called, and in which order. An example of this is the MPI History Plugin (see Section Example Plugin: MPI History Library , Section 12.4 of this guide) which records MPI usage.
- Observing progress of an application and variation of values across processes without having to interrupt the application.

5.15.1 Setting a tracepoint

Tracepoints are added by either right-clicking on a line in the *Source Code Viewer* and selecting the *Add Tracepoint* menu item, or by right-clicking in the *Tracepoints* view and selecting *Add Tracepoint*. If you right-click in the *Source Code Viewer* a number of variables based on the current line of code will be captured by default.

Tracepoints can lead to considerable resource consumption by the user interface if placed in areas likely to generate a lot of passing. For example, if a tracepoint is placed inside of a loop with N iterations, then N separate tracepoint passings will be recorded. Whilst Allinea DDT will attempt to merge such data scalably, when alike tracepoints are passed in order between processes, where process behaviour is likely to be divergent and unmergeable then a considerable load would then be caused.

If it is necessary to place a tracepoint inside a loop, set a condition on the tracepoint to ensure you only log what is of use to you.

Tracepoints also momentarily stop processes at the tracepoint location in order to evaluate the expressions and record their values, and hence if placed inside (eg.) a loop with a very large number of iterations – or a function executed many times per second, then a slow down in the pace of your application will be noticed.

5.15.2 Tracepoint Output

The output from the tracepoints can be found in the *Tracepoint Output* view.

Tracepoint	Processes	Values logged
subdomain.f...	1, rank 0	jend: <input type="text"/> 0 ny: <input type="text"/> 9
blts.f:74	16, ranks 0-15	jend: <input type="text"/> 8 ldmx: <input type="text"/> 9 j: <input type="text"/> 9 ldmy: <input type="text"/> 9 jst: <input type="text"/> 1-2 ldmz: <input type="text"/> 33
blts.f:74	16, ranks 0-15	jend: <input type="text"/> 8 ldmx: <input type="text"/> 9 j: <input type="text"/> 9 ldmy: <input type="text"/> 9 jst: <input type="text"/> 1-2 ldmz: <input type="text"/> 33
blts.f:74	16, ranks 0-15	jend: <input type="text"/> 8 ldmx: <input type="text"/> 9 j: <input type="text"/> 9 ldmy: <input type="text"/> 9 jst: <input type="text"/> 1-2 ldmz: <input type="text"/> 33
blts.f:74	16, ranks 0-15	jend: <input type="text"/> 8 ldmx: <input type="text"/> 9 j: <input type="text"/> 9 ldmy: <input type="text"/> 9 jst: <input type="text"/> 1-2 ldmz: <input type="text"/> 33

Fig 39: Output from Tracepoints in an F90 application

Where tracepoints are passed by multiple processes within a short interval, the outputs will be merged. Sparklines of the values recorded are shown for numeric values – along with the range of values obtained – showing the variation across processes.

As alike tracepoints are merged then this can lose the order/causality between *different* processes in tracepoint output. For example, if process 0 passes a tracepoint at time T, and process 1 passes the tracepoint at T + 0.001, then this will be shown as one passing of both process 0 and process 1, with no ordering inferred.

Sequential consistency is preserved during merging, in that for any process, the sequence of tracepoints for that process will be in order.

To find particular values or interesting patterns, use the *Only show lines containing* box at the bottom of the panel. Tracepoint lines matching the text entered here will be shown, the rest will be hidden. To search for a particular value, for example, try “my_var: 34 “ - in this case the space at the end helps distinguish between my_var: 34 and my_var: 345.

For more detailed analysis you may wish to export the tracepoints – right-click and choose Export from the pop-up menu. A HTML tracepoint log will be written using the same format as DDT's offline mode.

5.16 Examining The Stack Frame

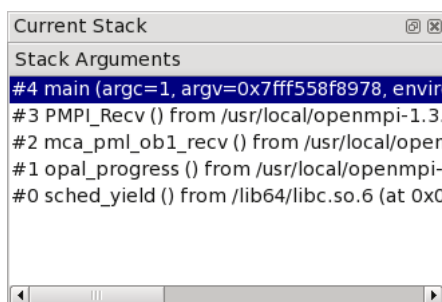


Fig 40: The Stack Tab

The stack back trace for the current process and thread are displayed under the *Stack* tab of 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.

5.17 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.

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

5.18.1 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 *Input/Output*, *Breakpoints* and *Watches*:

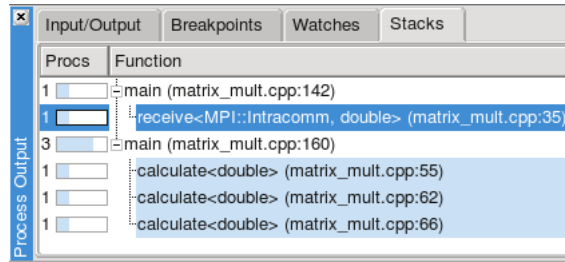


Fig 41: Parallel Stack View

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.

5.18.2 The Parallel Stack View in Detail

The *Parallel Stack View* takes over much of the work of the *Stack* 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* 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:

Procs	Function
15	main (hello.c:123)
15	func1 (hello.c:40)
15	func2 (hello.c:34)
1	main (hello.c:85)

Fig 42: Current Frame Highlighting in Parallel Stack View

In the example above, the program's processes are at two different locations. 1 process is in the `main` function, at line 85 of `hello.c`. The other 15 processes are inside a function called `func2`, at line 34 of `hello.c`. The 15 processes reached `func2` in the same way – `main` called `func1` on line 123 of `hello.c`, then `func1` called `func2` on line 40 of `hello.c`. Clicking on any of these functions will take you to the appropriate line of source code, and display any local variables in that stack frame.

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:

Procs	Function
2	main (matrix_mult.cpp:160)
2	calculate<double> (matrix_mult.cpp:55)
1	/home/cjanuary/code/ddt/examples/matrix_mult.cpp:55 On this line: 5.1, 6.1
6	main (matrix_mult.cpp:160)
6	broadcast<MPI::Intracomm, double> (matrix_mult.cpp:20)
6	MPI::Intracomm::Bcast
6	PMPI::Intracomm::Bcast (intracomm_inln.h:59)
6	PMPI_Bcast

Fig 43: Parallel Stack View tool tip

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.

5.19 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. If the source file cannot be found you will be prompted for its location.

DDT highlights lines of the source code to show where your program currently is. Lines that contain processes from the current group are shaded in that group's colour. Lines only containing processes from other groups are shaded in grey.

This pattern is repeated in the focus on process and thread modes. For example, when you focus on a process, DDT highlights lines containing that process in the group colour, and other processes from that group in grey.

DDT also highlights lines of code that are on the stack – functions that your program will return to when it has finished executing the current one. These are drawn with a faded look to distinguish them from the currently-executing lines.

You can hover the mouse over any highlighted line to see which processes/threads are currently on that line. This information is presented in a variety of ways, depending on the current focus setting:

Focus on Group

A list of groups that are on the selected line, along with the processes in them on this line, and a list of threads from the current process on the selected line.

Focus on Process

A list of the processes from the current group that are on this line, along with the threads from the current process on the selected line.

Focus on Thread

A list of threads from the current process on the selected line.

The tool tip distinguishes between processes and threads that are currently executing that line, and ones that are on the stack by grouping them under the headings *On the stack* and *On this line*.

Variables and Functions

Right-clicking on a variable or function name in the *Source Code Viewer* will make DDT check whether there is a matching variable or function, and then display extra information and options in a sub-menu.

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.

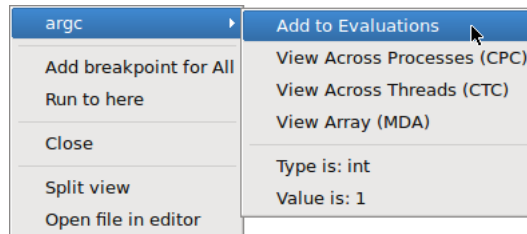


Fig 44: Right-Click Menu – Variable Options

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.

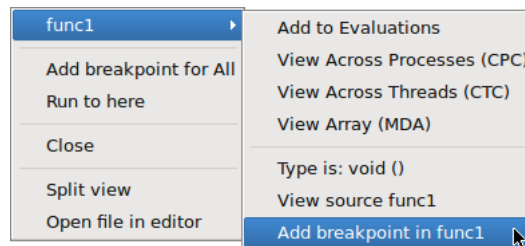


Fig 45: Right-Click Menu – Function Options

5.20 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.

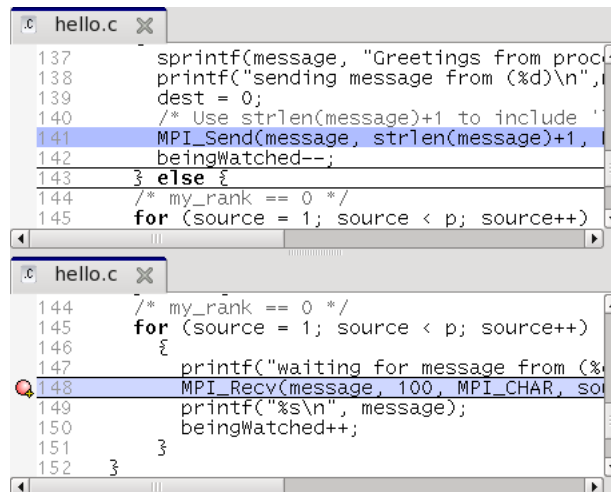


Fig 46: Horizontal Alignment Of Multiple Source Files

5.21 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, SIGCHLD, SIG63 and SIG64 are passed directly through to the user process without being intercepted by DDT.

5.21.1 Sending Signals

You can send a signal to your program using the *Send Signal* window (select the *Control* → *Send Signal...* menu item). Select the signal you want to send from the drop-down list and click the *Send to process* button.

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.

Right clicking in these windows brings up additional options – including the ability to edit values (in the *Evaluations window*), to change the display base, or to compare data across processes and threads. The right-click menu will also allow you to choose whether the fields in structures (classes or derived types) should be displayed alphabetically by element name or not – which is useful for when structures have very many different fields.

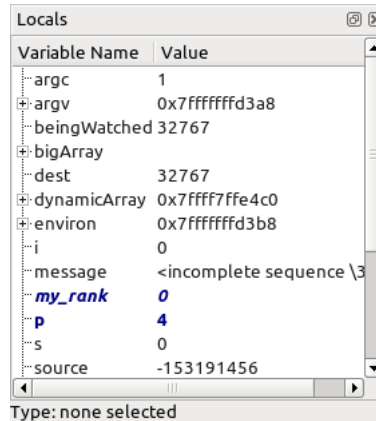


Fig 47: Displaying Variables

6.1 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.

6.2 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 ever 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.

The *Locals* view compares the value of scalar variables against other processes. If a value varies across processes in the current group the value is highlighted in green.

When stepping or switching processes if the value of a variable is different from the previous position or process it is highlighted in blue.

After stepping out of function the return value is displayed at the top of the *Locals* view (for selected debuggers).

6.3 Arbitrary Expressions And Global Variables

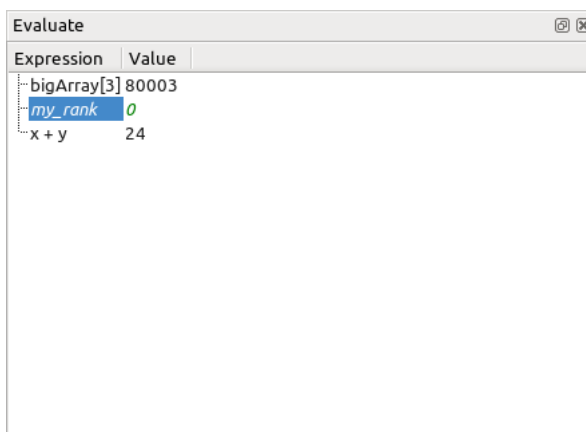


Fig 48: Evaluating Expressions

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 Viewer* 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)`. It is also worth noting that although the Fortran syntax allows you to use keywords as variable names, DDT will not be able to evaluate such variables on most platforms. Please contact support@allinea.com if this is a problem for you.

6.3.1 Fortran Intrinsics

The following Fortran intrinsics are supported by the default GNU debugger included with DDT:

ABS	AIMAG	CEILING	CMPLX
FLOOR	IEEE_IS_FINITE	IEEE_IS_INF	IEEE_IS_NAN
IEEE_IS_NORMAL	ISFINITE	ISINF	ISNAN
ISNORMAL	MOD	MODULO	REALPART

Support in other debuggers, including the CUDA and Cell GNU debugger variants, may vary.

6.3.2 Changing the language of an Expression

Ordinarily, expressions in the *Evaluate* window and *Locals/Current* windows are evaluated in the language of the current stack frame. This may not always be appropriate – for example a pointer to user defined structure may be passed as value within a Fortran section of code, and you may wish to view the fields of the C structure. Alternatively, you may wish to view a global value in a C++ class whilst your process is in a Fortran subroutine.

You can change the language that DDT uses for your expressions by right clicking on the expression, and clicking *Change Type/Language* - selecting the appropriate language for the expression. To restore the default behaviour, change this back to *Auto*.

6.3.3 Macros and #defined Constants

By default, many compilers will not output sufficient information to allow the debugger to display the values of “#defined” constants or macros – as including this information can greatly increase executable sizes. With the GNU compiler, adding the “-g3” option to the command line options will generate extra definition information which DDT will then be able to display.

6.4 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 and dropped into the *Evaluate* window. Here, all of the usual *Evaluate* window functionality applies to the module variable. To help with variable identification in the *Evaluate* window, module variable names are prefixed with the Fortran module name and two colons ::.

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. The *Fortran Modules* tab will not be displayed if the underlying debugger does not support the retrieval and manipulation of Fortran module data
2. The *Fortran Modules* tab will display an empty module list if the Fortran modules debug data is not present or in a format understood by DDT.

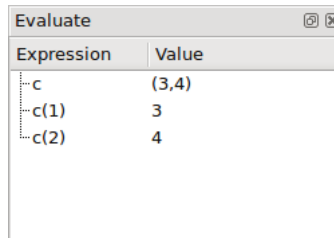
One limitation of the *Fortran Modules* tab is that the modules debug data compiled into the executable does not include any indication of the module USE hierarchy (e.g. if module A USEs module B, the inherited members of module B are not shown under the data displayed for module A). Consequently, the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

6.5 Viewing Complex Numbers in Fortran

When working with complex numbers, you may wish to view only the real or imaginary elements of the number. This can be useful when evaluating expressions, or viewing an array in the Multi Dimensional Array Viewer (see section 6.13 *Multi Dimensional Array Viewer*).

You can use the Fortran intrinsic functions REALPART and AIMAG to get the real or imaginary parts of a number, or their C99 counterparts creal and cimag .

Complex numbers in Fortran can also be accessed as an array, where element 1 is the real part, and element 2 is the imaginary part.



Expression	Value
c	(3,4)
c(1)	3
c(2)	4

Fig 49 : Viewing the Fortran complex number 3+4i

6.6 C++ STL Support

DDT uses pretty printers for the GNU C++ STL implementation and Nokia's Qt library, and Boost, designed for use with the GNU Debugger. These are used automatically to present such C++ data in a more understandable format.

For some compilers, the STL pretty printing can be confused by non-standard implementations of STL types used by a compiler's own STL implementation. In this case, and in the case where you wish to see the underlying implementation of an STL type, you can run DDT with the environment variable DDT_DISABLE_PRETTY_PRINT set which will disable the pretty printing.

6.7 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, or simply <unknown_bounds>. 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.

To tell DDT the size of the array right-click on the array and select the *Edit Type...* menu option. This will open a window like the one below. Enter the real type of the array in the *New Type* box.

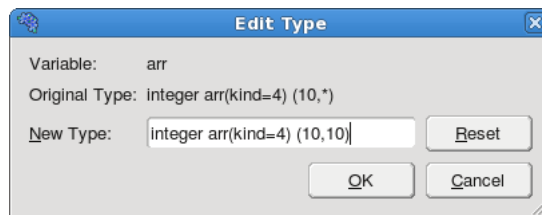


Fig 50 : Edit Type window

Alternatively the MDA can be used to view the entire array.

6.8 UPC Support

Allinea DDT supports many different UPC compilers – including the GNU UPC compiler, the Berkeley UPC compiler those provided by Cray and SGI. Note that in order to enable UPC support, you may need to select the appropriate MPI/UPC implementation from DDT's Options/System menu.

Debugging UPC applications introduces a small number of changes to the user interface.

- Processes will be identified as UPC Threads, this is purely a terminology change for consistency with the UPC language terminology. UPC Threads will have behaviour identical to that of separate processes: groups, process control and cross-process data comparison for example will apply across UPC Threads.
- The type qualifier “shared” is given for shared arrays or pointers to shared
- Dereferencing pointers to shared items will yield a triple (address, thread, phase) – and pointer arithmetic (eg. Referencing $*(&x[n] + 1)$) will correctly evaluate and fetch remote data where required.
- Values in shared arrays are not automatically compared across processes: the value of $x[i]$ is by definition identical across all processes. It is not possible to identify pending read/write to remote data. Non-shared data types such as local data or local array elements will still be compared automatically.
- When viewing shared arrays in the multi-dimensional array viewer (described later in this userguide) the “distributed dimensions” option is not available, again by definition of a shared array it is unnecessary as the distributed data is already shown by default.

All other components of DDT will be identical to debugging any multi-process code.

6.9 Changing Data Values

In the *Evaluate* window, the value of an expression may be set by right-clicking and selecting *Edit Value*. This will allow you to change the value of the expression for the current process, current group, or for all processes.

Note: The variable must exist in the current stack frame for each process you wish to assign the value to.

6.10 Viewing Numbers In Different Bases

When you are viewing an integer numerical expression you may right-click on the value and use the *View As* sub menu to change which base the value is displayed in. The *View As* → *Default* option displays the value in its original (default) base.

6.11 Examining Pointers

You can examine pointer contents by clicking the + next to the variable or expression. This will automatically dereference the pointer. You can also use the *View As Vector*, *Reference*, or *Dereference* menu items.

6.12 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.)

Expression	Value
tables	
[0]	1
[1]	2
[2]	3
[3]	4
[4]	5
[5]	6
[6]	7
[7]	8
[8]	9
[9]	10
[10]	11
[11]	12

Fig 51: 2D Array In C: type of tables `int [12][12]`

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

Fig 52: 2D Array In Fortran: type of twodee is `integer(3, 5)`

6.13 Multi Dimensional Array Viewer

DDT provides a *Multi-Dimensional Array (MDA) Viewer* (fig. 53) for viewing multi-dimensional arrays.

To open the *Multi-Dimensional Array Viewer*, right-click on a variable in the *Source Code*, *Locals*, *Current Line(s)* or *Evaluate* views and select the *View Array (MDA)* context menu option. You can also open the MDA directly by selecting the *Multi-Dimensional Array Viewer* menu item from the *View* menu.

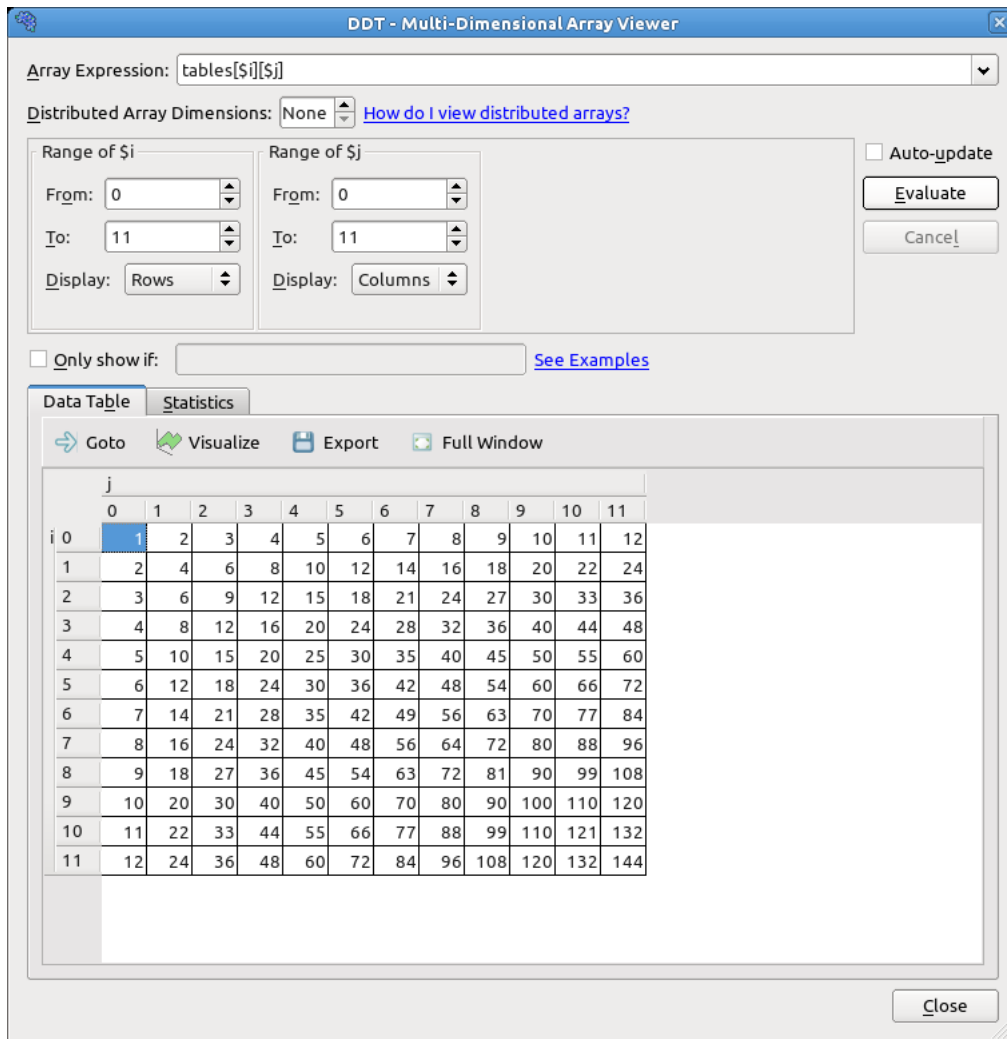


Fig 53: Multi-Dimensional Array Viewer

If you open the MDA by right clicking on a variable, DDT will automatically set the *Array Expression* and other parameters based on the type of the variable. Click the *Evaluate* button to see the contents of the array in the *Data Table* .

The *Full Window* button hides the settings at the top of the window so the table of values occupies the full window, allowing you to make full use of your screen space . Click the button again to rev e all the settings again.

6.13.1 Array Expression

The *Array Expression* is an expression containing a number of *subscript metavariables* that are substituted with the subscripts of the array. For example, the expression `myArray($i, $j)` has two metavariables, `$i` and `$j`. The metavariables are unrelated to the variables in your program.

The range of each metavariable is defined in the boxes below the expression, e.g. *Range of \$i* . The *Array Expression* is evaluated for each combination of `$i` , `$j` , etc. and the results shown in the *Data Table* . You can also control whether each metavariable is shown in the *Data Table* using Rows or Columns.

The metavariables may be re-ordered by dragging and dropping them. For C/C++ expressions the major dimension is on the left and the minor dimension on the right, for Fortran expressions the major dimension is on the right and the minor dimension on the left. Distributed dimensions may not be re-ordered – they must always be the most major dimensions.

6.13.2 Filtering by Value

You may want the *Data Table* to only show elements that fit a certain criteria, e.g. elements that are zero. If the *Only show if* box is checked then only elements that match the boolean expression in the box are displayed in the *Data Table*, e.g. `$value == 0`. The special metavariable `$value` in the expression is replaced by the actual value of each element. The *Data Table* automatically hides rows or columns in the table where no elements match the expression.

6.13.3 Distributed Arrays

A distributed array is an array that is distributed across one or more processes as local arrays.

The Multi-Dimensional Array Viewer can display certain types of distributed arrays, namely UPC shared arrays (for supported UPC implementations), and general arrays where the distributed dimensions are the most major (i.e. the distributed dimensions change the most slowly) and are independent from the non-distributed dimensions.

UPC shared arrays are treated the same as local arrays, simply right-click on the array variable and select View Array (MDA).

To view a non-UPC distributed array first create a process group containing all the processes that the array is distributed over. If the array is distributed over all processes in your job then you can simply select the *All* group instead. Right-click on the local array variable in the *Source Code*, *Locals*, *Current Line(s)* or *Evaluate* views. The *Multi-Dimensional Array Viewer* window will open with the Array Expression already filled in. Enter the number of distributed array dimensions in the corresponding box. A new subscript metavariable (`$x`, `$y`, etc.) will be automatically added for each distributed dimension. Enter the ranges of the distributed dimensions so that the product is equal to the number of processes in the current process group, then click the *Evaluate* button.

6.13.4 Advanced: How Arrays Are Laid Out in the Data Table

The Data Table is two dimensional, but the Multi-Dimensional Array Viewer may be used to view arrays with any number of dimensions, as the name implies. This section describes how multi-dimensional arrays are displayed in the two dimensional table.

Each subscript metavariable (`$i`, `$j`, `$x`, `$y`, etc.) maps to a separate dimension on a hypercube. Usually the number of metavariables is equal to the number of dimensions in a given array, but this does not necessarily need to be the case, e.g. `myArray($i, $j) * $k` introduces an extra dimension, `$k`, as well as the two dimensions corresponding to the two dimensions of `myArray`.

The figure below corresponds to the expression `myArray($i, $j)` with `$i = 0..3` and `$j = 0..4`.

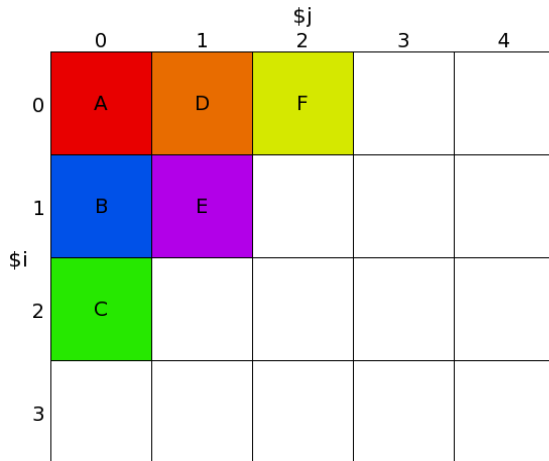


Fig 54: $myArray(\$i, \$j)$ with $\$i = 0..3$ and $\$j = 0..4$.

Now let's say $myArray$ is part of a three dimensional array distributed across three processes. The figure below shows what the local arrays look like for each process.

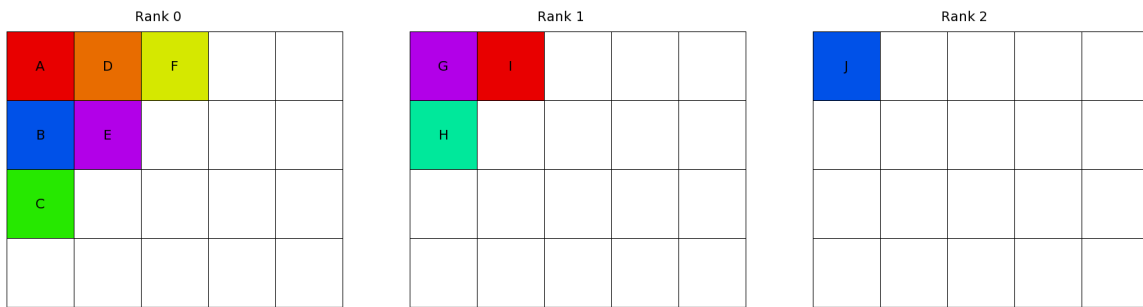


Fig 55: The local array $myArray(\$i, \$j)$ with $\$i = 0..3$ and $\$j = 0..4$ on ranks 0 – 2.

And as a three dimensional distributed array with $\$x$ the distributed dimension:

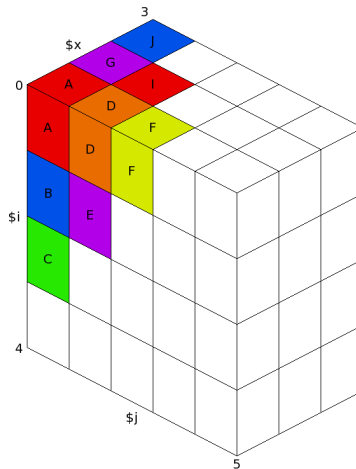


Fig 56: A three dimensional distributed array comprised of the local array $myArray(\$i, \$j)$ with $\$i = 0..3$ and $\$j = 0..4$ on ranks 0 – 2 with $\$x$ the distributed dimension.

This cube is projected (just like 3D projection) onto the two dimensional Data Table. Dimensions marked *Display as Rows* are shown in rows, and dimensions marked *Display as Columns* are shown in columns, as you would expect.

More than one dimension may viewed as Rows, or more than one dimension viewed as Columns. The dimension that changes fastest depends on the language your program is written in. For C/C++ programs the leftmost metavariable (usually $\$i$ for local arrays or $\$X$ for distributed arrays) changes the most slowly (just like with C array subscripts). The rightmost dimension changes the most quickly. For Fortran programs the order is reversed (the rightmost is most major, the leftmost most minor).

The figure below shows how the three dimensional distributed array above is projected onto the two dimensional Data Table:

			0						\$x 1 \$j 2					2				
	0	1	2	3	4	0	1							0	1	2	3	4
0	A	D	F			G	I							J				
1	B	E				H												
2	C																	
3																		

Fig 57 : A three dimensional distributed array comprised of the local array `myArray($i, $j)` with $\$i = 0..3$ and $\$j = 0..4$ on ranks 0 – 2 projected onto the Data Table with $\$X$ (the distributed dimension) and $\$j$ displayed as *Columns* and $\$i$ displayed as *Rows* .

6.13.5 Auto Update

If you check the *Auto Update* check box the *Data Table* will be automatically updated as you switch between processes/threads and step through the code.

6.13.6 Statistics

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`.

6.13.7 Export

You may export the contents of the results table to a file in the Comma Separated Values (CSV) or HDF5 format that can be plotted or analysed in your favourite spreadsheet or mathematics program.

There are two CSV export options: List (one row per value) and Table (same layout as the on screen table).

Note: If you export a Fortran array from DDT in HDF5 format the contents of the array are written in column major order. This is the order expected by most Fortran code, but the arrays will be transposed if read with the default settings by C-based HDF5 tools. Most HDF5 tools have an option to switch between row major and column major order.

6.13.8 Visualisation

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*. You can only plot one or two dimensions at a time – if your table has more than two dimensions the *Visualise*

button will be disabled. After filling the table of the *MDA Viewer* with values (see previous section), click *Visualise* 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 check boxes 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: DDT requires OpenGL to run. If your machine does not have hardware OpenGL support, software emulation libraries such as MesaGL are also supported.

In some configurations OpenGL is known to crash – a work around if the 3D visualization crashes is to set the environment variable `LIBGL_ALWAYS_INDIRECT` to 1 – the precise configuration which triggers this problem is not known.

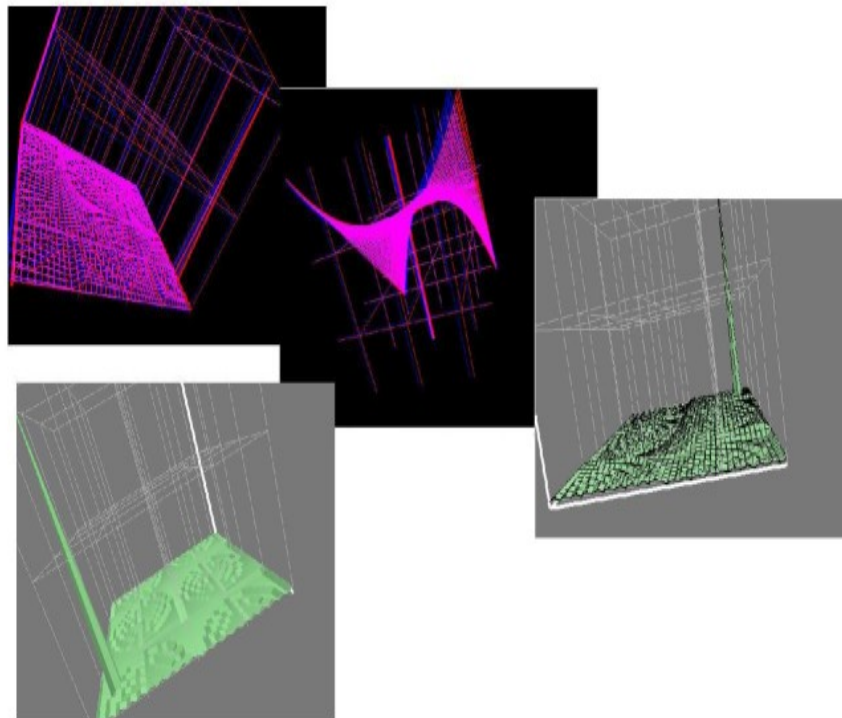


Fig 58 : DDT Visualization

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.

6.14 Cross-Process and Cross-Thread Comparison

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

To compare values across processes or threads, right-click on a variable inside the *Source Code*, *Locals*, *Current Line(s)* or *Evaluate* windows and then choose one of the *View Across Processes (CPC)* or *View Across Threads (CTC)* options. You can also bring up the CPC or CTC directly from the *View* menu in the main menu bar.

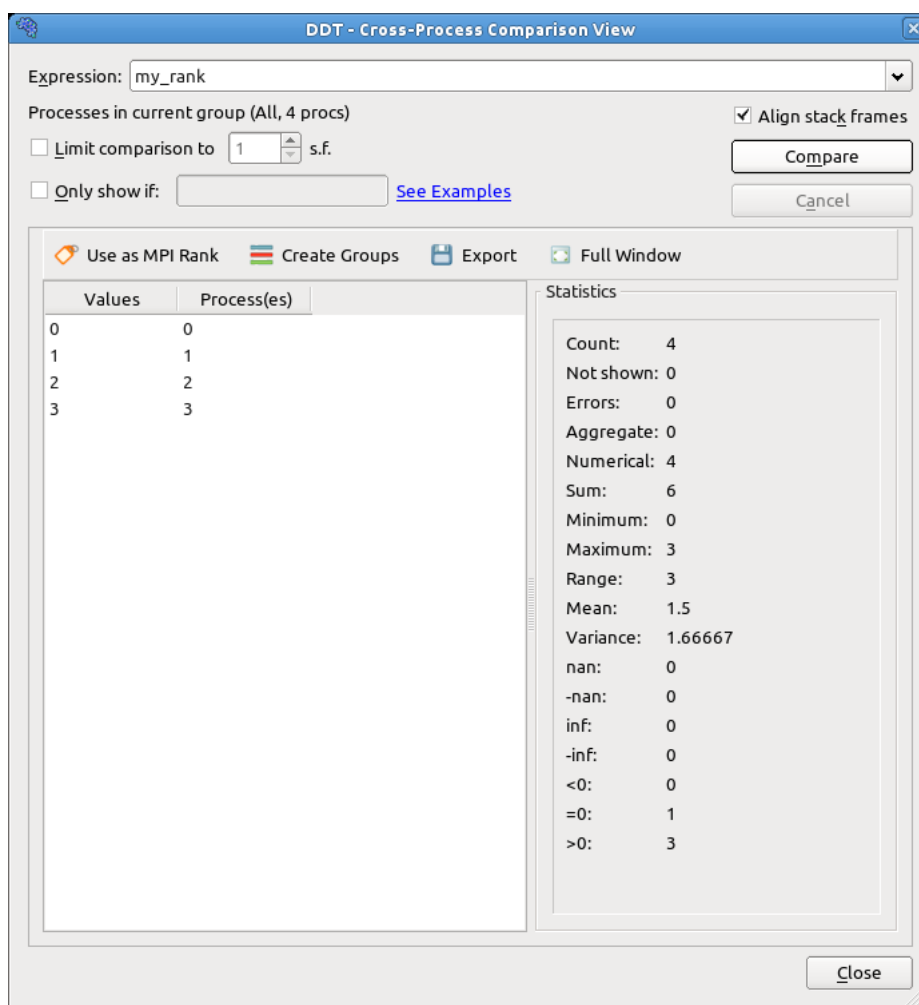


Fig 59: Cross-Process Comparison – Compare View

Processes and threads 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.

If you are comparing across processes, 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).

You can enter a second boolean expression in the *Only show if* box to control which values are displayed. Only values for which the boolean expression evaluates to `true` / `.TRUE.` are displayed in the results table. The special metavariable `$value` in the expression is replaced by the actual value. Click the *Show Examples* link to see examples.

The *Align Stack Frames* check box tries to automatically make sure all processes and threads are in the same stack frame when comparing the variable value. This is very helpful for most programs, but you may wish to disable it if different processes/threads run entirely different programs.

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

You can create a group for the ranks corresponding to each unique value by clicking the *Create Groups* button.

The *Export* button allows you to export the list of values and corresponding ranks as a Comma Separated Values (CSV) file.

The *Full Window* button hides the settings at the top of the window so the list of values occupies the full window, allowing you to make full use of your screen space. Click the button again to reveal the settings again.

The *Statistics* panel shows Maximum, Minimum, Variance and other statistics for numerical values.

6.15 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.

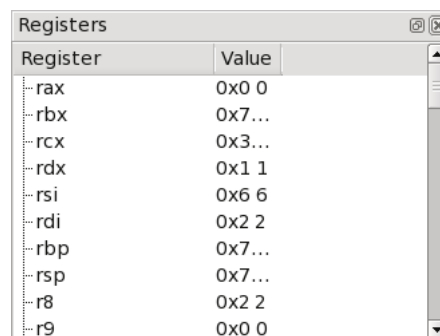
What makes a variable or expression valid? These criteria must be met:

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.

6.16 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
-rax	0x0 0
-rbx	0x7...
-rcx	0x3...
-rdx	0x1 1
-rsi	0x6 6
-rdi	0x2 2
-rbp	0x7...
-rsp	0x7...
-r8	0x2 2
-r9	0x0 0

Fig 60: Register View

6.17 Interacting Directly With The Debugger

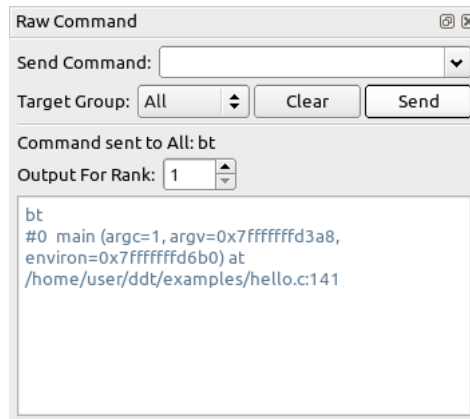


Fig 61: Raw Command Window

DDT provides a *Raw Command* window that will allow you to send commands directly to the debugger interface. This window 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 window; 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 window 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 playing 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 under the *Input/Output* tab. Both standard output and error are shown, although on most MPI implementations, error is not buffered but output is and consequently can be delayed.

7.1 Viewing Standard Output And Error

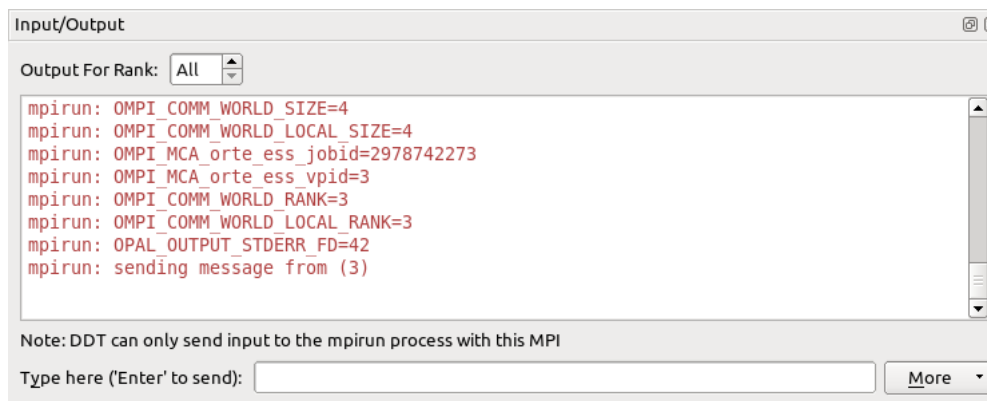


Fig 62: Standard Output Window

The Input/Output tab is at the bottom of the screen (by default).

The output may be selected and copied to the X-clipboard.

7.2 Displaying Selected Processes

You can choose whether to view the output for all processes, or just a single process.

Note: Some MPI implementations pipe stdin, stdout and stderr from every process through mpirun or rank 0.

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.

7.3 Saving Output

By right-clicking on the text it is possible to save it to a file. You also have the option to copy a selection to the clipboard.

7.4 Sending Standard Input (DDT-MP)

DDT provides an *Input File* box in the *Run* window. This allows you to choose a file to be used as the standard input (`stdin`) for your program. (DDT will automatically add arguments to `mpirun` to ensure your input file is used.)

Alternatively, 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 *Run* window *Advanced* options.

It is also possible to enter input during a session. Start your program as normal, then switch to the *Input/Output* panel. Here you can see the output from your program and type input you wish to send. You may also use the *More* button to send input from a file, or send an EOF character.

Remember: Although input can be sent while your program is paused, the program must then be played to read the input and act upon it.

If you are currently viewing output for all processes then the input you type will also be sent to all processes, similarly if you are currently viewing the output for a single process then the input will be sent to just that process.

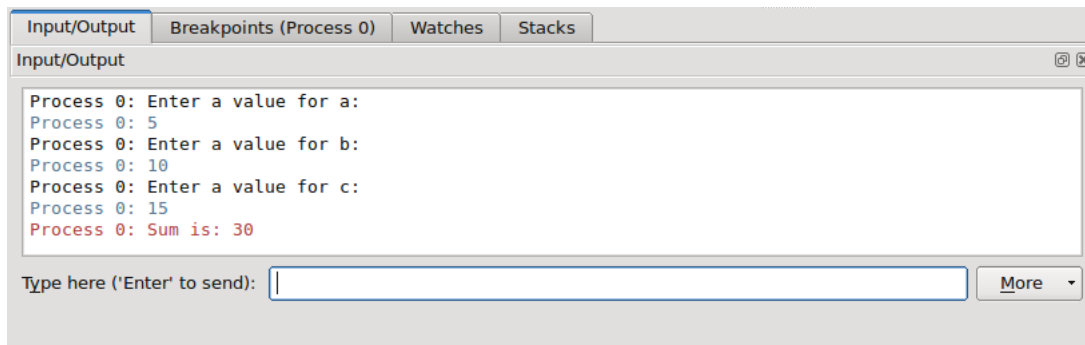


Fig 63: Sending Input

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

DDT's Message Queue debugging feature shows the status of the internal message buffers of MPI – for example showing the messages that have been sent by a process but not yet received by the target. This capability relies on the MPI implementation supporting this via a debugging support library: the majority of MPIs do this.

You can use DDT to detect common errors such as deadlock – where all processes are waiting for each other, or for detecting when messages are present that are unexpected, which can correspond to two processes disagreeing about the state of progress through a program.

8.1 Viewing The Message Queues

Open the *Message Queues* window by selecting *Message Queues* from the *View* menu.

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

DDT will automatically load the message queue support library from your MPI implementation (provided one exists). If it fails, an error message will be shown.

If an error is shown then the library may not exist, and you will need to compile the debug interface for your MPI implementation. In MPICH this is done by using `--enable-debug` when running `configure`. LAM and OpenMPI 1.2.4 automatically compile the library.

It may also be necessary to specifically include the path to the support library in the `LD_LIBRARY_PATH`, or 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/libtvmpich.so`).

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

8.2 Interpreting the Message Queues

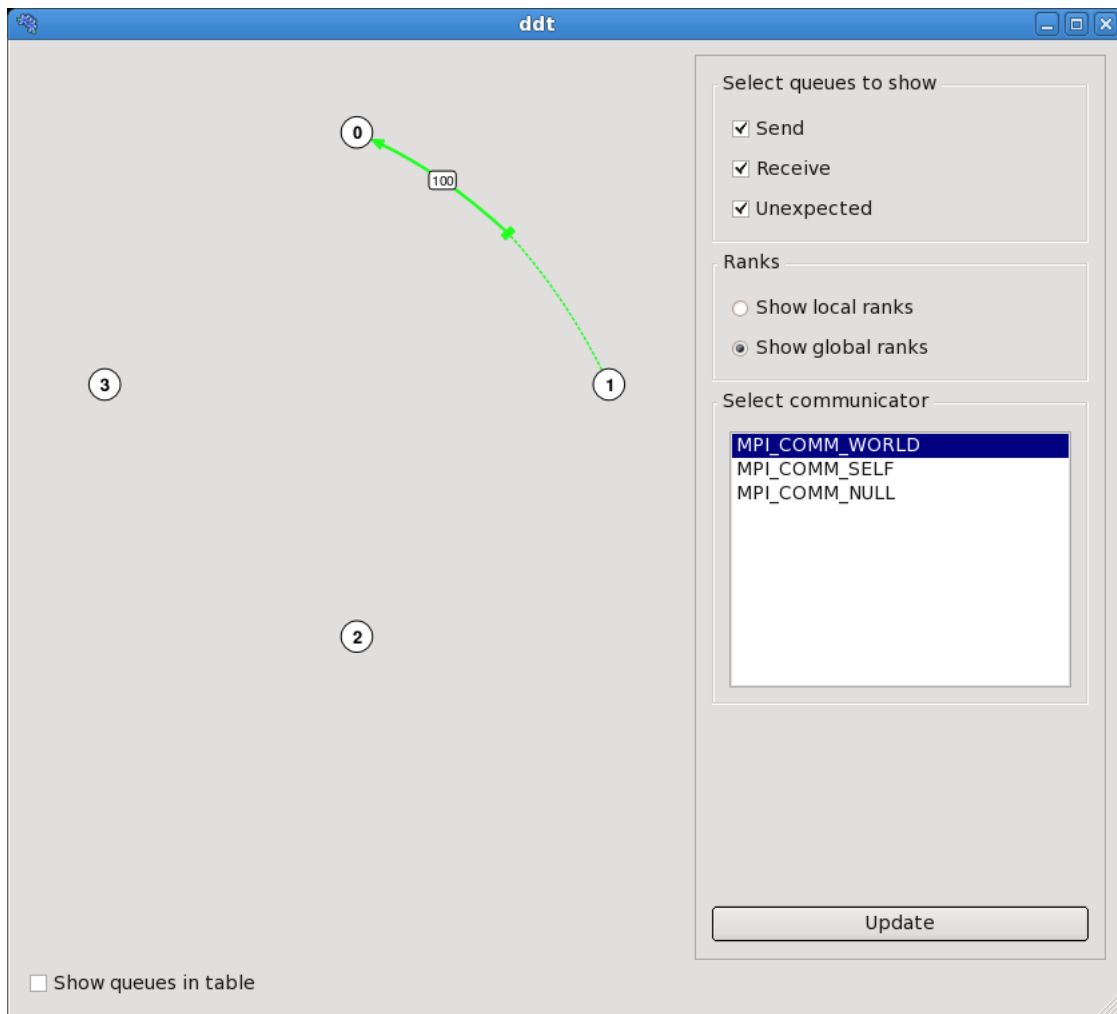


Fig 64: Message Queue Window

To see the messages, 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`), if the *Show Local Ranks* option is selected. To see the 'usual' ranks, select *Show Global Ranks*.

There are three different types of message queues about which there is information. Different colours are used to display messages from each type of queue.

Label	Description
Send Queue	Calls to MPI send functions that have not yet completed.
Receive Queue	Calls to MPI receive functions that have not yet completed.
Unexpected Message Queue	Represents messages received by the system but the corresponding receive function call has not yet been made.

Messages in the *Send* queue are represented by a red arrow, pointing from the sender to the recipient. The line is solid on the sender side, but dashed on the received side (to represent a message that has been *Sent* but not yet been *Received*).

Messages in the *Receive* queue are represented by a green arrow, pointing from the sender to the recipient. The line is dashed on the sender side, but solid on the recipient side (to represent the recipient being ready to receive a message that has not yet been sent).

Messages in the *Unexpected* queue are represented by a dashed blue arrow, pointing from sender of the unexpected message to the recipient.

Please note that the quality and availability of message queue data can vary considerably between MPI implementations – some of the data can therefore be incomplete.

8.3 Deadlock

If you see a loop in the graph, it can be because of deadlock – every process waiting to receive from the preceding process in the loop. For synchronous communications (eg. `MPI_Ssend`) then this is invariably a problem. For other types of communication it can be the case (eg. with `MPI_Send`) that, for example, messages are 'in the ether' or in some O/S buffer and the send part of the communication is complete but the receive hasn't started. If the loop persists after playing the processes and interrupting them again, this indicates a likely deadlock.

9 Memory Debugging

Allinea DDT has a powerful parallel memory debugging capability. This feature intercepts calls to the system memory allocation library, recording memory usage and monitoring correct usage of the library by performing heap and bounds checking.

Typical problems that can be resolved by using Allinea DDT with memory debugging enabled include

- Memory exhaustion due to memory leaks can be prevented by examining the *Current Memory Usage* display which groups and quantifies memory according to the location at which blocks have been allocated.
- Persistent but random crashes caused by access to memory beyond the bounds of an allocation block – can be resolved by using the *Guard Pages* feature
- Crashing due to deallocation of the same memory block twice and other forms deallocation of an invalid pointers – for example deallocating a pointer that is not at the start of an allocation.

9.1 Enabling Memory Debugging

To enable memory debugging within Allinea DDT, from the *Run* window click on the *Memory Debugging* checkbox.

The default options are usually sufficient, but you may need to configure extra options (see below) if you have a multithreaded application or multithreaded MPI – such as that found on systems using Open MPI with Infiniband, or a Cray XE6 system.

With the Memory Debugging setting enabled, start your application as normal. Allinea DDT will take care of ensuring that the settings are propagated through your MPI or batch system when your application starts.

If a problem is detected and it was not possible to load the memory debugging library, a message will be displayed – and you should refer to the Configuration section in this chapter for suggested resolution steps.

9.2 Configuration

Whilst configuration is often not necessary, it can be used to increase or change the memory checks and protection or to alter the information that is available. A summary of the settings is displayed on the *Run* dialog in the *Memory Debugging* section.

To examine or change the options, select the *Details* button adjacent to the *Memory Debugging* checkbox on the *Run* dialog, which then displays the *Memory Debugging Options* window.

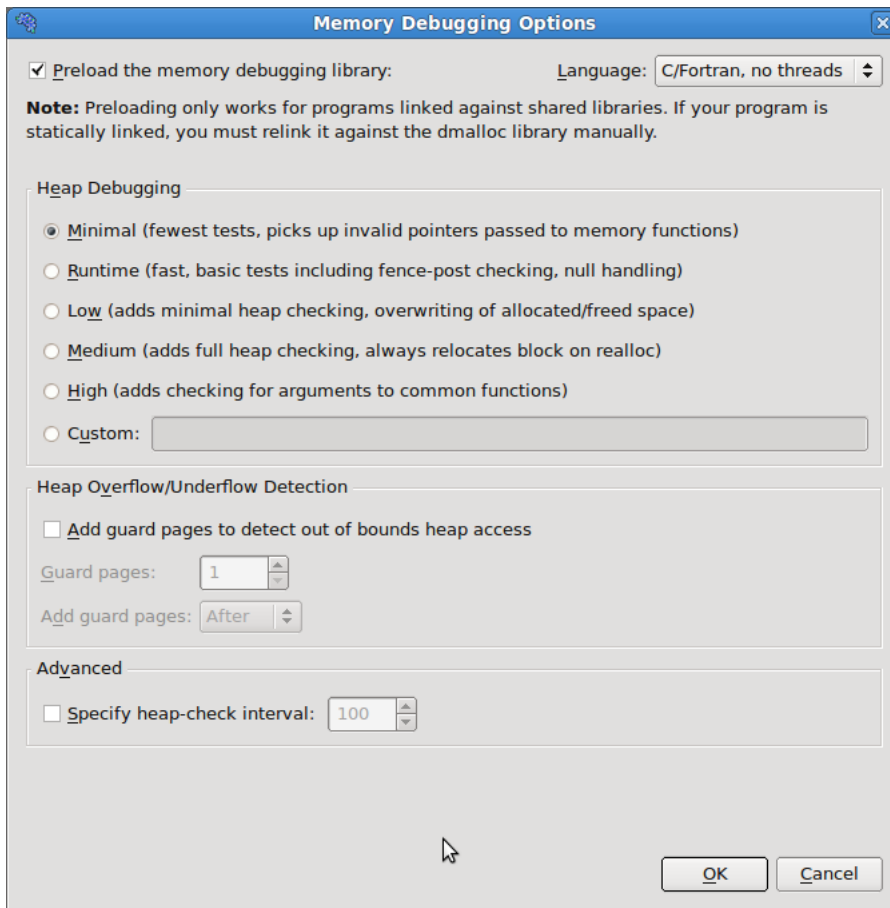


Fig 65: Memory Debugging Options

The two most significant options are:

1. *Preload the memory debugging library* - when this is checked, DDT will automatically load the memory debugging library. This is usually the most appropriate setting. If you have linked against one of DDT's memory debugging 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 box showing *C/Fortran, No Threads* in the screen shot – click here and select the option that best matches your program, be it *C/Fortran*, *C++*, *Single-Threaded*, *Multi-Threaded*. It is often sufficient to leave this set to *C++/Threaded* rather than continually changing this setting.

The *Heap Debugging* section allows you to turn on/off specific memory debugging features. The two most important things to remember are:

1. *Minimal* will catch trivial memory errors such as deallocating memory twice.
2. The further down the list you go, the more slowly your program will execute. In practice, anything up to *Low* is often 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.

You can turn on *Heap Overflow/Underflow Detection* to detect out of bounds heap access. See section 9.3.4 *Writing Beyond An Allocated Area* for more details.

Almost all users can leave the heap check interval at its default setting. It determines how often the memory debugging library 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).

If your program runs particularly slowly with Memory Debugging enabled you may be able to get a modest speed increase by disabling the *Store backtraces for memory allocations* option. This disables stack back traces in the *View Pointer Details* and *Current Memory Usage* windows, support for custom allocators and cumulative allocation totals.

It is possible to enable Memory Debugging for only selected MPI ranks by checking the *Only enable for these processes* option and entering the ranks you want to it for.

Note: The Memory Debugging library will still be loaded into the other processes, but no errors will be reported.

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.

9.2.1 Changing Settings at Run Time

You can change most Memory Debugging settings while your program is running by selecting the *Control → Memory Debugging Options* menu item. In this way you can enable Memory Debugging with a minimal set of options when your program starts, set a breakpoint at a place you want to investigate for memory errors, then turn on more options when the breakpoint is hit.

9.3 Pointer Error Detection and Validity Checking

Once you have enabled memory debugging and started debugging, all calls to the allocation and deallocation routines of heap memory will be intercepted and monitored. This allows both for automatic monitoring for errors, and for user driven inspection of pointers.

9.3.1 Library Usage Errors

If the memory debugging library reports an error Allinea DDT will display a window similar to the one shown below. This briefly reports the type of error detected and gives the option of continuing to play the program, or pausing execution.

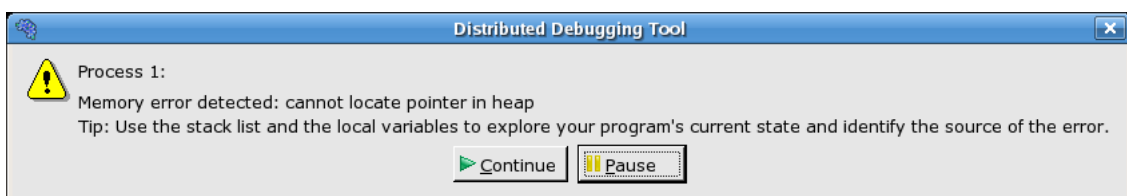


Fig 66: Memory Error Message

If you choose to pause the program then Allinea DDT will 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 – as the local variables and variables on the current line will provide insight into what is happening.

If the cause of the issue is still not clear, then it is possible to examine some of the pointers referenced to see whether they're valid and which line they were allocated on, as we now explain.

9.3.2 Check Pointer 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, *Check pointer Is Valid* will be available. Clicking on this will display a message telling you whether or the expression points to valid memory allocated on the heap.

Note: Memory allocated on the heap refers to memory allocated 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.

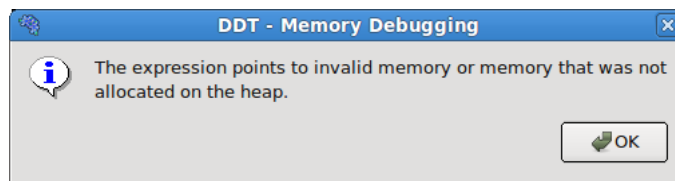


Fig 67: Invalid memory message

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:

9.3.3 View Pointer Details

Along side *Check Pointer Validity* in the expression context menu is *View Pointer Details*. This will display the amount of memory allocated to the pointer and which part of your code originally allocated that memory:

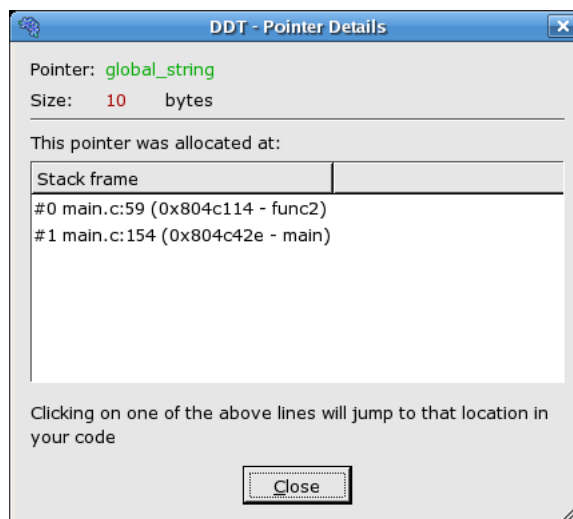


Fig 68: Pointer details

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

Note: Only a single stack frame will be displayed if the Store stack backtraces for memory allocations option is disabled.

9.3.4 Writing Beyond An Allocated Area

Use *Heap Overflow /Underflow Detection* option to detect read or writes beyond or before an allocated block. Any attempts to read or write to the specified number of pages before or after the block will cause a segmentation violation that stops your program. Add the guard pages after the block to detect heap overflows, or before to detect heap underflows. The default value of 1 page will catch most heap overflow errors but if this doesn't work a good rule of thumb is to set the number of guard pages according to the size of a row in your largest array. The exact size of a memory page depends on your operating system but a typical size is 4Kb. So if a row of your largest array is 64Kb then set the number of pages to $64/4 = 16$.

Some system architectures do not allow unaligned memory accesses. If you are running Allinea DDT on an Itanium- or SPARC-based machine, you may find that your program stops with SIGBUS errors when *Heap Overflow Detection* is turned on. Unfortunately, *Heap Overflow Detection* cannot be supported on these platforms. However, 'Fence Post' checking (see below) is still available.

DDT will also perform 'Fence Post' checking even if this option is disabled, as long as the *Heap Debugging* settings are *Runtime* or above. In this mode, an extra portion of memory is allocated at the start and/or end of your allocated block, and a pattern is written into this area. If you attempt to write beyond your data, say by a few elements, then this will be noticed by DDT, however 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.

9.4 Current Memory Usage

Memory leaks can be a significant problem for software developers. If your application's memory usage grows faster than expected or continues to grow through its execution then it is possible that memory is being allocated which is not being returned when it is no longer required.

This type of problem is typically difficult to diagnose, and particularly so in a parallel environment, but Allinea DDT is able to make this task simple.

At any point in your program you can go to View → *Current Memory Usage* and DDT will then display the currently allocated memory in your program for the currently selected group. For larger process groups, the processes displayed will be the ones that are using the most memory across that process group.

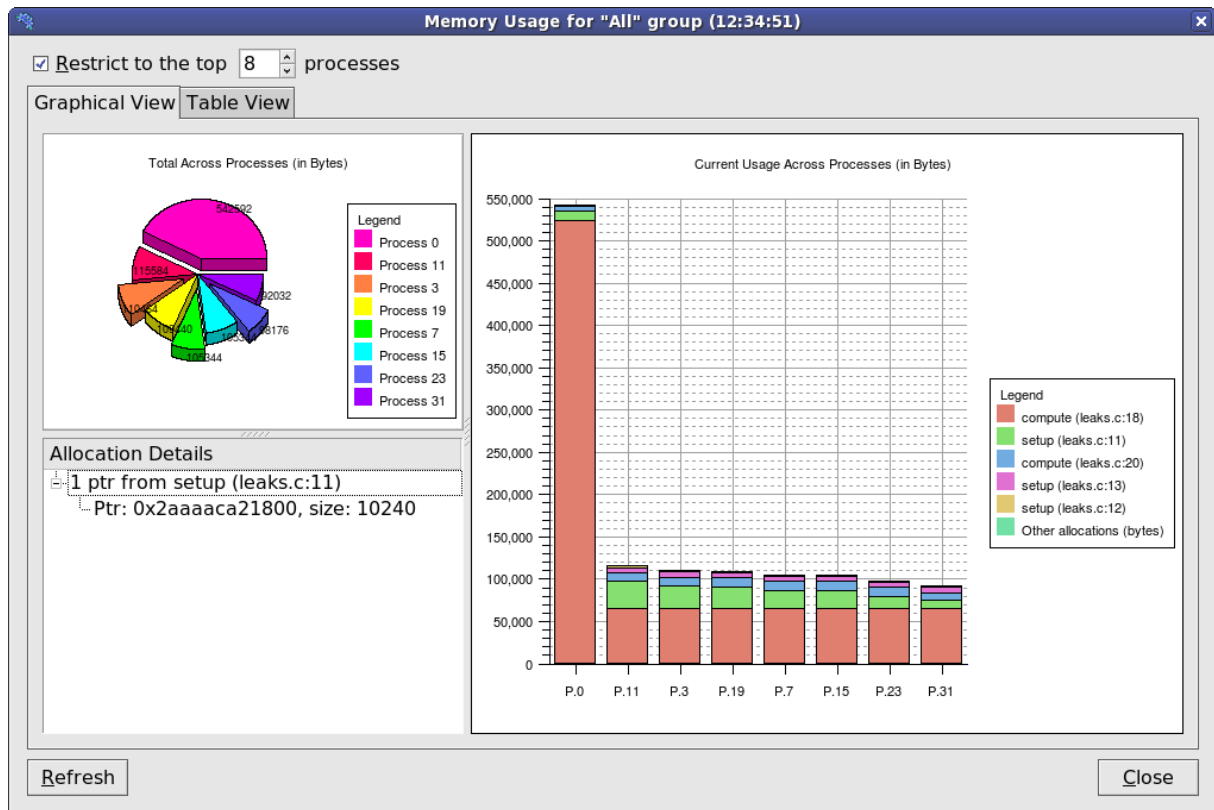


Fig 69: Memory Usage Graphs

The pie chart gives an at-a-glance comparison of the total memory allocated to each process. This gives an indication of the balance of memory allocations; any one process taking an unusually large amount of memory should be easily identifiable 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 total amount of memory allocated by a particular function in your code. 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! There are 6 blocks in total. The first 5 represent the 5 functions that allocated 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.

Note: Only a single stack frame will be displayed if the Store stack backtraces for memory allocations option is disabled.

The *Table View* shows all the functions that allocated memory in your program alongside the number of allocations (*Count*) and the total number of bytes allocated (*Size*). It also shows the total memory allocated by each function's callees in the *Cumulative Count* and *Cumulative Size* columns.

For example: `func1` calls `func2` which calls `malloc` to allocate 50 bytes. DDT will report an allocation of 50 bytes against `func2` in the *Size* column of the *Current Memory Usage* table. DDT will also record a cumulative allocation of 50 bytes against both functions `func1` and `func2` in the *Cumulative Size* column of the table.

Another valuable use of this feature is to play the program for a while, refresh the window, play 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.

9.4.1 Detecting Leaks when using Custom Allocators/Memory Wrappers

Some compilers wrap memory allocations inside many other functions. In this case Allinea 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 *Current Memory Usage* view. You can mark such functions as *Custom Allocators* to exclude them from the bar chart and table by right-clicking on the function and selecting the *Add Custom Allocator* menu item. Memory allocated by a custom allocator is recorded against its caller instead.

For example, if `myfunc` calls `mymalloc` and `mymalloc` is marked as a custom allocator the allocation will be recorded against `myfunc` instead. You can edit the list of custom allocators by clicking the “*Edit Custom Allocators...*” button at the bottom of the window.

9.5 Memory Statistics

The *Memory Statistics* view (*View → Memory Statistics*) shows a total of memory usage across the processes in an application. The processes using the most memory are displayed, along with the mean across all processes in the current group, which is useful for larger process counts.

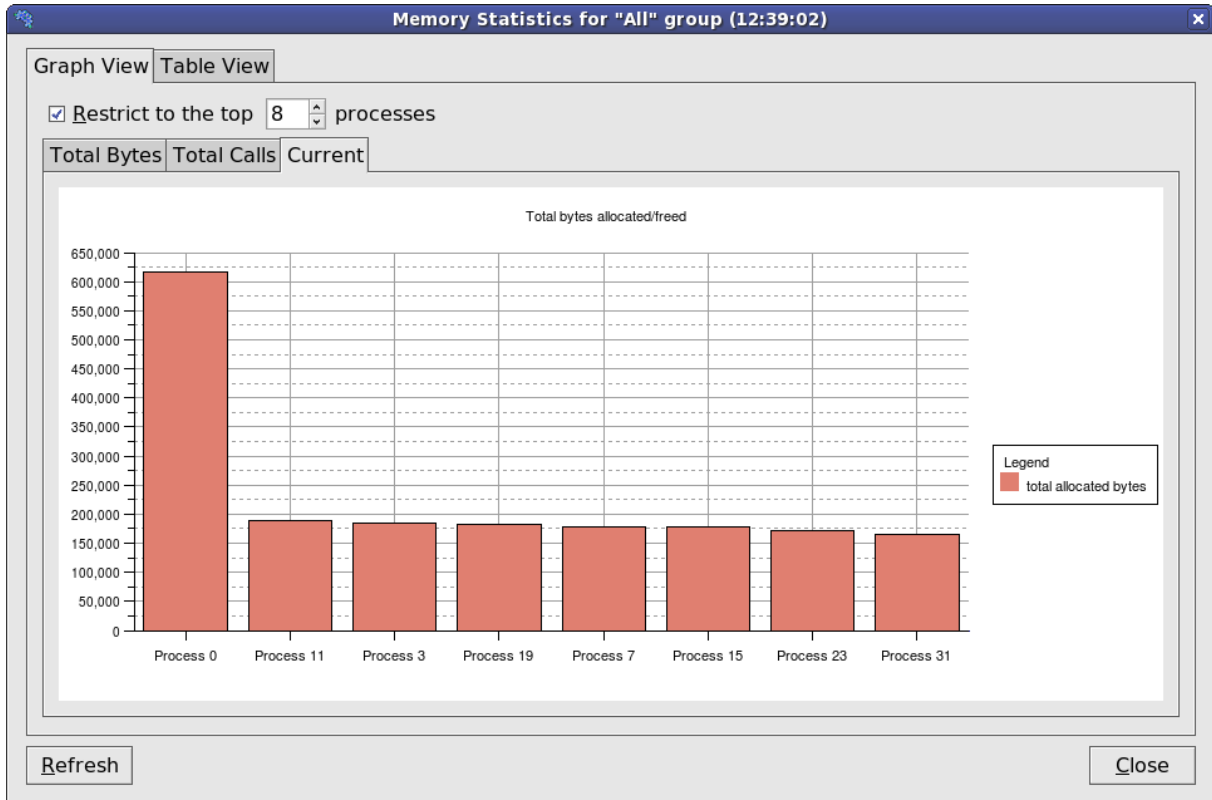


Fig 70: Memory Statistics

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 Checkpointing

10.1 What Is Checkpointing?

A program's entire state (or a practical subset thereof) may be recorded to disk/memory as a checkpoint. The program may later be restored from the checkpoint and will resume execution from the recorded state.

Sometimes you are not sure what information you need to diagnose a bug until it is too late to get it. For example, a program may crash because a variable has a particular unexpected value. You want to know where the variable was set to that value but it is too late to set a watch on it. However if you have an earlier checkpoint of the program you can restore the checkpoint, set the watch, and then let it fail again.

10.2 Checkpoint Support In DDT


DDT supports two types of checkpointing:

- Run-time checkpoints are stored in memory. They are valid for the life time of a session but are lost when the session is ended.
- Persistent checkpoints are stored on disk. When you restore a persistent checkpoint DDT will start a new session.

DDT includes three checkpoint providers:

- The *gdb* checkpoint provider is available on all Linux platforms. It supports run-time checkpoints only. It has no special support for MPI programs so restoring a checkpoint across an MPI function call may fail. It does not support threads and will fail if your program is linked with a threading library (e.g. libpthread), even if your program is only using one thread at the time of the checkpoint.
- The BLCR checkpoint provider support persistent checkpoints for single process programs using the Berkley Lab Checkpoint/Restart library. Your program must be linked with the BLCR library. See section 4.2 Making an application checkpointable of the BLCR user guide. Note: if you restart a BLCR checkpoint made in DDT at the command line the process(es) will be stopped (as if they had been sent SIGSTOP). Pass the `--cont` option to `cr_restart` to avoid this.
- The experimental OpenMPI checkpoint provider supports persistent checkpoints and has explicit MPI support. To work with DDT the `ompi-checkpoint` and `ompi-restart` commands must support the `-prd` argument. You must pass some arguments to `mpirun` before you can checkpoint your OpenMPI program in DDT. Click the *Advanced >>* button in the *Run* window then add the arguments below to the *MPIRun Arguments* box:
`-am ft-enable-cr -gmca opal_cr_enable_prd 1`

10.3 How To Checkpoint

To checkpoint your program, click the *Checkpoint* button on the tool bar . The first time you click the button you will be asked to select a checkpoint provider. If no checkpoint providers support the current MPI and debugger an error message will be displayed instead. See the previous section for a list of available checkpoint providers in DDT.

When the checkpoint has completed a new window will open displaying the name of the new checkpoint.

10.4 Restoring A Run-time Checkpoint

To restore a run-time checkpoint, click the *Restore Checkpoint* button on the tool bar . A new window will open with a list of available checkpoints. Select a checkpoint then click the *Ok* button.

The program state will be restored to the checkpoint. The Parallel Stack View, Locals View, etc. will all be updated with the new program state.

You may also restore a permanent checkpoint in the same way, however this will require a session restart and you will be warned accordingly.

10.5 Restoring A Persistent Checkpoint

To restore a persistent checkpoint, click the *Restore Checkpoint* button on the *Welcome* screen or select the *Restore Checkpoint* menu item from the *Session* menu. The *Restore Checkpoint* window will open.

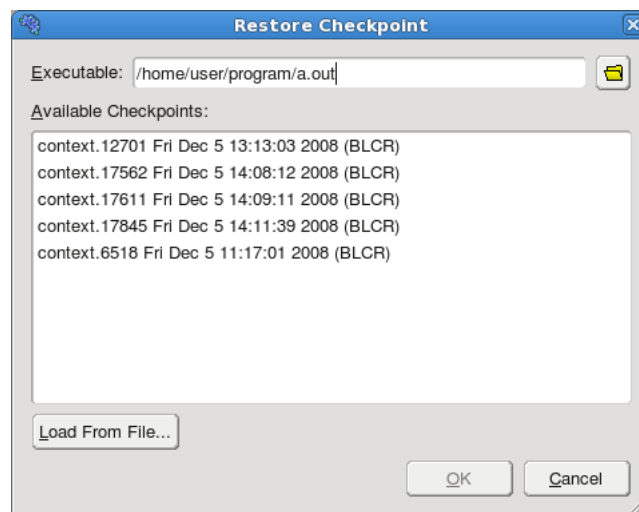


Fig 71: Restore Checkpoint Window

You can select from a list of checkpoints automatically discovered by DDT. You may also choose to load a checkpoint from a file if the checkpoint you want to restore is not listed.

Once you have selected a checkpoint click the *OK* button. DDT will start a new session, restoring your program from your selected checkpoint.

11 The Licence Server

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

11.1 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.

11.2 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.

11.3 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, user name, process ID, and IP address of the clients are logged.

11.4 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.

11.5 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.

11.6 Examples

In this example, a dedicated licence server machine exists but uses the same file system 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`, on 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
```


11.7 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
```

11.8 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. This can be accessed in a web browser at the following URL:

```
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]

11.9 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 time out period has expired. The length of this time out 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 time out period would be:

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

During this time out 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 time out period is active.

After this time out 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.

12 Using and Writing Plugins for DDT

Plugins are a quick and easy way to preload a library into your application and define some breakpoints and tracepoints during its use. They consist of an XML file which instructs DDT what to do and where to set breakpoints or tracepoints.

Examples are MPI correctness checking libraries, or you could also define a library that is preloaded with your application that could perform your own monitoring of the application. It also enables a message to be displayed to the user when breakpoints are hit, displaying, for example, an error message where the message is provided by the library in a variable.

12.1 Supported Plugins

At the time of release, DDT supports plugins for two MPI correctness-checking libraries:

- Intel Message Checker, part of the Intel Trace Analyser and Collector (Commercial with free evaluation: <http://www.intel.com/cd/software/products/asm-na/eng/306321.htm>) version 7.1
- Marmot (Open source: <http://www.hlrs.de/organization/amt/projects/marmot>), support expected in version 2.2 and above.

12.2 Installing a Plugin

To install a plugin, locate the XML DDT plugin file provided by your application vendor and copy it to:

```
{ddt-installation directory} /plugins/
```

It will then appear in DDT's list of available plugins on the DDT - Run dialog.

Each plugin takes the form of an XML file in this directory. These files are usually provided by third-party vendors to enable their application to integrate with DDT. A plugin for the Intel Message Checker (part of the Intel Trace Analyser and Collector) is included with the DDT distribution.

12.3 Using a Plugin

To activate a plugin in DDT, simply click on the checkbox next to it in the window, then run your application. Plugins may automatically perform one or more of the following actions:

- Load a particular dynamic library into your program
- Pause your program and show a message when a certain event such as a warning or error occurs
- Start extra, optionally hidden MPI processes (see the Writing Plugins section for more details on this).
- Set tracepoints – which log the variables during an execution.

If DDT says it cannot load one of the plugins you have selected, check that the application is correctly installed, and that the paths inside the XML plugin file match the installation path of the application.

12.4 Example Plugin: MPI History Library

DDT's plugin directory contains a small set of files that make a plugin to log MPI communication.

- Makefile – Builds the library and the configuration file for the plugin.
- README.wrapper – Details the installation, usage and limitations
- wrapper-config: Used to create the plugin XML config file - used by DDT to preload the library and set tracepoints which will log the correct variables.
- wrapper-source: Used to automatically generate the source code for the library which will wrap the original MPI calls.

The plugin is designed to wrap around many of the core MPI functions and seamlessly intercept calls to log information which is then displayed in DDT. It is targeted at MPI implementations which use dynamic linking, as this can be supported without relinking the debugged application.

Static MPI implementations can be made to work also, but this is outside the scope of this version.

This package must be compiled before first use - in order to be compatible with your MPI version. It will not appear in DDT's GUI until this is done.

To install - as a non-root user - in your local ~/.ddt/plugins directory.

make local

To install - as root in the DDT plugins directory

make

Once you have run the above, start DDT and to enable the plugin, click "Advanced" on the "Run and Debug" dialog. Select "History v1.0", and start your job as normal. DDT will take care of preloading the library and setting default tracepoints.

This plugin records call counts, total sent byte counts, and the arguments used in MPI function calls. Function calls and arguments are displayed (in blue) in the Input/Output panel.

The function counts are available -- in the form of a variable

`_MPIHistoryCount_{function}`.

The sent bytes counters are accumulated for most functions - but specifically they are not added for the vector operations such as MPI_Gatherv. These count variables within the processes are available for use within DDT - in components such as the cross-process comparison window, enabling a check that - say - the count of MPI_Barriers is consistent, or primitive MPI bytes sent profiling information to be discovered.

The library does not record the received bytes - as most MPI receive calls in isolation only contain a maximum number of bytes allowed, rather than bytes received. The MPI status is logged, the the SOURCE tag therein enables the sending process to be identified.

There is no per-communicator logging in this version.

This version is for demonstration purposes for the tracepoints and plugin features. It could generate excessive logged information, or cause your application to run slowly if it is a heavy communicator. This library can be easily extended – or its logging can be reduced by removing the tracepoints from the generated “history.xml” file (stored in DDTPATH or ~/.ddt/plugins) – which would make execution considerably faster, but still retain the byte and function counts for the MPI functions.

12.5 Writing a Plugin

Writing a plugin for DDT is easy. All that is needed is an XML plugin file that looks something like this:

```
<plugin name="Sample v1.0" description = "A sample plugin that
demonstrates DDT's plugin interface.">
  <preload name="samplelib1" />
  <preload name="samplelib2" />
  <environment name="SUPPRESS_LOG" value="1" />
  <environment name="ANOTHER_VAR" value="some value" />
  <breakpoint location="sample_log" action="log"
message_variable="message" />
  <breakpoint location="sample_err" action="message_box"
message_variable="message" />
  <extra_control_process hide="last" />
</plugin>
```

Only the surrounding `plugin` tag is required – all the other tags are entirely optional. A complete description of each appears in the table below. If you are interested in providing a plugin for DDT as part of your application bundle, we will be happy to provide you with any assistance you need getting up and running. Contact support@allinea.com for more information.

Tag	Attribute	Description
plugin	name	The plugin's unique name. This should include the application/library the plugin is for, and its version. This is shown in the <i>DDT – Run</i> dialog.
plugin	description	A short snippet of text to describe the purpose of the plugin/application to the user. This is also shown in the <i>DDT – Run</i> dialog.
preload	name	Instructs DDT to preload a shared library of this name into the user's application. The shared library must be locatable using <code>LD_LIBRARY_PATH</code> , or the OS will not be able to load it.
environment	name	Instructs DDT to set a particular environment variable before running the user's application.
environment	value	The value that this environment variable should be set to.
breakpoint	location	Instructs DDT to add a breakpoint at this location in the code. The location may be in a preloaded shared library (see above). Typically this will be a function name, or a fully-qualified C++ namespace and class name. C++ class members must include their signature and be enclosed in single quotes, e.g. <code>'MyNamespace::DebugServer::breakpointOnError(char*)'</code>
breakpoint	action	Only <code>message_box</code> is supported in this release. Other settings will cause DDT to stop at the breakpoint but take no action.
breakpoint	message_variable	A <code>char*</code> or <code>const char*</code> variable that contains a message to be shown to the user. DDT will group identical messages from different processes together before displaying them to the user in a message box.
extra_control_process	hide	Instructs DDT to start one more MPI process than the user requested. The optional <code>hide</code> attribute can be <code>first</code> or <code>last</code> , and will cause DDT to hide the first or last process in <code>MPI_COMM_WORLD</code> from the user. This process will be allowed to execute whenever at least one other MPI process is executing, and messages or breakpoints (see above) occurring in this process will appear to come from all processes at once. This is only necessary for tools such as Marmot that use an extra MPI process to perform various runtime checks on the rest of the MPI program.
tracepoint	location	See breakpoint location.
tracepoint	variables	A comma-separated list of variables to log on every passing of the tracepoint location.

13 CUDA GPU Debugging

Allinea DDT is able to debug applications that use NVIDIA CUDA, with actual debugging of the code running on the GPU, simultaneously whilst debugging the host CPU code.

In addition to the native NVIDIA nvcc compiler for CUDA, a number of other compilers with GPU targets are supported, including CAPS HMPP and Cray Open MP Accelerators, with on-device debugging. Other compilers are also supported, notably PGI CUDA Fortran and the PGI Accelerator Model, with device code actually running on the host CPU instead of the GPU. Please see the Compilers section of the Appendix for further details.

13.1 Licensing

In order to debug CUDA programs with Allinea DDT, you will require a CUDA-enabled licence key, which is an additional option to default licences. If CUDA is not included with a licence, the CUDA options will be greyed-out on the run dialog of DDT.

Whilst debugging a CUDA program, an additional process from your licence is used for each GPU. An exception to this is that single process licences will still allow the debugging of a single GPU.

Please note that in order to serve a *floating* CUDA licence, you will need to use the licence server shipped with DDT 2.6 or later.

13.2 Preparing to Debug GPU Code

In order to debug your GPU program, you may need to add additional compiler command line options to enable GPU debugging.

For NVIDIA's nvcc compiler, kernels must be compiled with the “-g -G” flags. This enables generation of information for debuggers in the the kernels, and will also disable some optimisations that would hinder debugging.

For other compilers, please refer to Page 100 of this guide (GPU Language Support) and Compiler Notes and Known Issues (Appendix C) and your vendor's own documentation.

Note: At this point OpenCL debugging of GPUs is not supported.

13.3 Launching the Application

To launch a CUDA job, tick the CUDA box on the run dialog before clicking run/submit.

Attaching to running CUDA applications is not possible if the application has already initialized the driver, for example having executed any kernel or called any functions from the CUDA library. For MPI applications it is essential to place all CUDA initialization after the MPI_Init call.

13.4 Controlling GPU threads

Controlling GPU threads has been integrated with the standard DDT controls – so that the usual play, pause, and breakpoints are all applicable to GPU kernels, for example.

As GPUs have different execution models to CPUs, there are some behavioural differences that we now detail.

13.4.1 Breakpoints

CUDA Breakpoints can be set in the same manner as other breakpoints in DDT (See section 5.7 *Setting Breakpoints*).

Breakpoints affect all GPU threads, and cause the application to stop whenever a thread reaches the breakpoint. Where kernels have similar workload across blocks and grids, then threads will tend to reach the breakpoint together and the kernel will pause once per set of blocks that are scheduled (ie. set of threads that fit on the GPU at any one time). Where kernels have divergent distributions of work across threads, then timing may be such that threads within a running kernel will hit a breakpoint and pause the kernel – and after subsequently continuing, more threads within the currently scheduled set of blocks will hit the breakpoint and pause the application again.

In order to apply breakpoints to individual blocks, warps or threads, conditional breakpoints can be used – for example using the built-in variables `threadIdx.x` (and `threadIdx.y` or `threadIdx.z` as appropriate) for thread indexes and setting the condition appropriately.

Where a kernel pauses at a breakpoint, the currently selected GPU thread will be changed if the previously selected thread is no longer “alive”.

13.4.2 Stepping

The GPU execution model is noticeably different from that of the host CPU. In the context of stepping operations – ie. step in, step over or step out – there are critical differences to note.

The smallest execution unit on a GPU is a warp – which, on current NVIDIA GPUs is 32 threads. Step operations operate on warps but nothing smaller.

Allinea DDT also makes it possible to step whole blocks, whole kernels or whole devices. The stepping mode is selected using the drop down list in the CUDA Thread Selector.

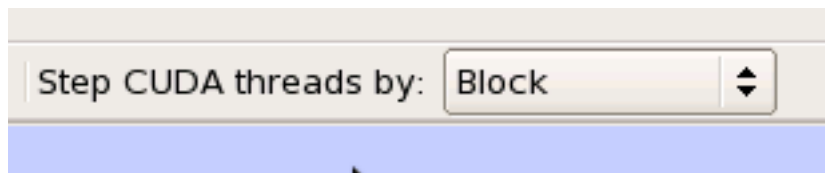


Fig 72: Selection of GPU Stepping Mode

*Note: GPU execution under the control of a debugger is not as fast as running without a debugger. When **stepping** blocks and kernels these are sequentialized into warps and hence stepping of units larger than a warp may be slow. It is not unusual for a step operation to take 60 seconds on a large kernel, particularly on newer devices where a step could involve stepping over a function call.*

It is not currently possible to “step over” or “step out” of inlined GPU functions.

Note: GPU functions are often inlined by the compiler. This can be avoided (dependent on hardware) by specifying the `__noinline__` keyword in your function declaration, and by compiling your code for a later GPU profile. e.g. by adding `-arch=sm_20` to your compile line.

13.4.3 Running and Pausing

Clicking the “Play/Continue” button in DDT will run all GPU threads. It is not possible to run individual blocks, warps or threads.

The pause button will pause a running kernel, although it should be noted that the pause operation is not as instantaneous for GPUs as would be experienced when using regular CPUs.

13.5 Examining GPU Threads and Data

Much of the user interface when working with GPUs is unchanged from regular MPI or multithreaded debugging. However, there are a number of enhancements and additional features that have been added to help understand the state of GPU applications.

These changes are summarised in this section.

13.5.1 Selecting GPU Threads



Fig 73: GPU Thread Selector

The Thread Selector allows you to select your current GPU thread. The current thread is used for the variable evaluation windows in DDT, along with the various GPU stepping operations.

The first entries represent the block index, and the subsequent entries represent the 3D thread index inside that block.

Changing the current thread will update the local variables, the evaluations, and the current line displays and source code displays to reflect the change.

The thread selector is also updated to display the current GPU thread if it changes as a result of any other operation – for example if

- The user changes threads by selecting an item in the Parallel Stack View
- A memory error is detected and is attributed to a particular thread
- The kernel has progressed, and the previously selected thread is no longer present in the device

The GPU Thread Selector also displays the dimensions of the grid, and blocks in your program.

It is only possible to inspect/control threads in the set of blocks that are actually loaded in to the GPU. If you try to select a thread that is not currently loaded, a message will be displayed.

Note: The thread selector is only displayed when there is a GPU kernel active.

13.5.2 Viewing GPU Thread Locations

The Parallel Stack View has been updated to display the location and number of GPU threads.

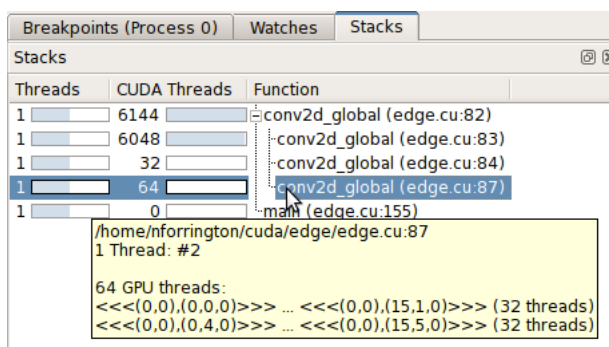


Fig 74: CUDA threads in the parallel stack view

Clicking an item in the Parallel Stack View will select the appropriate GPU thread – updating the variable displaying components accordingly and jump the source code viewer to the appropriate location.

Hovering over an item in the Parallel Stack view will also allow you to see which individual GPU thread ranges are at a location, as well as the size of each range.

13.5.3 Understanding Kernel Progress

Given a simple kernel that is to calculate an output value for each index in an array, it is not easy to check whether the value at position x in an array has been calculated, or whether the calculating thread has yet to be scheduled.

This contrasts sharply with scalar programming, where if the counter of a (up-)loop exceeds x then the value of index x can be taken as being the final value. If it is difficult to decide whether array data is fresh or stale, then clearly this will be a major issue during debugging.

Allinea DDT includes a component that makes this easy – the Kernel Progress display, which will appear at the bottom of the user interface by default when a kernel is in progress.

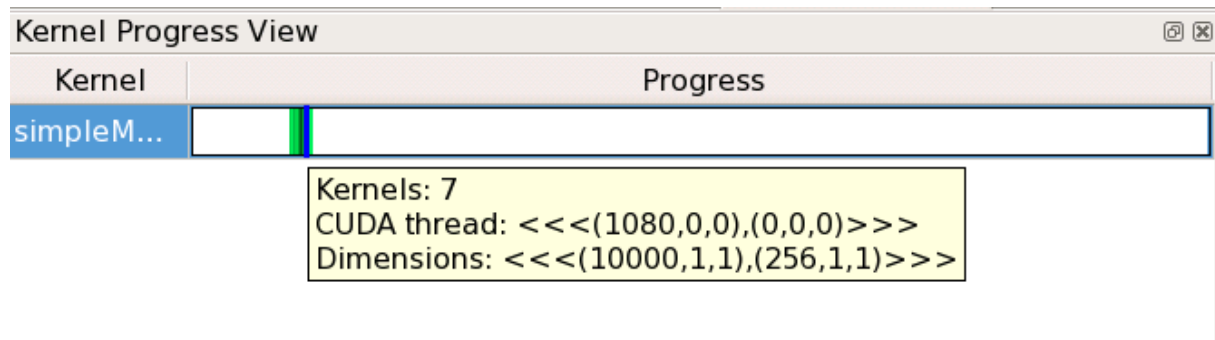


Fig 75: Kernel Progress Display

This view identifies the kernels that are in progress – with the number kernels identified and with grouping by different kernel identifiers (ie. kernel name) across processes – and uses a coloured progress bar to identify where GPU threads are in progress. The progress bar is a projection into a straight line of the (potentially) 6-dimension GPU block and thread indexing system and is tailored to the sizes of the kernels operating in the application.

By clicking within the colour highlighted sections of this progress bar, a GPU thread will be selected that matches the click location as closely as possible.

13.5.4 Source Code Viewer

The source code viewer allows you to visualise the program flow through your source code by highlighting lines in the current stack trace. When debugging GPU kernels, it will colour highlight lines with GPU threads present and display the GPU threads in a similar manner to that of regular CPU threads and processes. Hovering over a highlighted line in the code viewer will display a summary of the GPU threads on that line.

13.6 GPU Devices Information

One of the challenges of GPU programming is in discovering device parameters, such as the number of registers or the device type, and whether a device is present.

In order to assist in this, Alinea DDT includes a GPU Devices display. This display examines the GPUs that are present and in use across an application, and groups the information together scalably for multi-process systems.

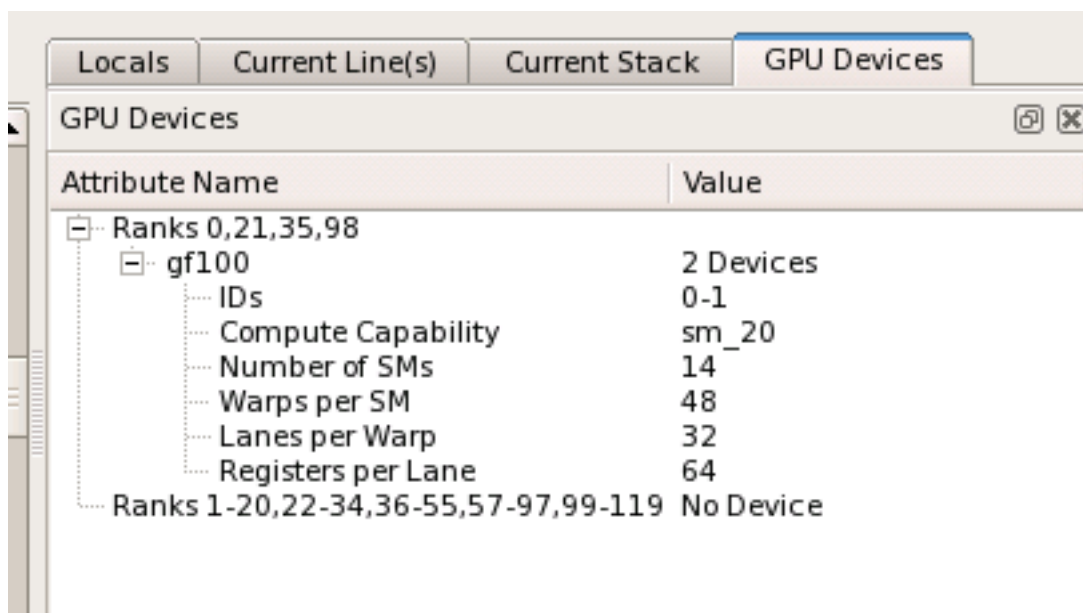


Fig 76: GPU Devices

Note: For CUDA 4.0, devices are only listed after initialization.

13.7 Known Issues / Limitations

13.7.1 Using Multiple GPUs

Single Process

- Debugging multiple GPUs requires NVIDIA CUDA Toolkit 3.2 or higher

Multiple Processes

- For CUDA 4.0, Alinea DDT can only debug a single GPU process per host. When trying to debug multiple processes with CUDA support enabled, DDT will disable GPU debugging for all but one process.

In order to debug multiple processes with GPU support you must configure your job launch mechanism to launch each process on a separate host.

- When debugging multiple GPU processes on the same machine, pausing the process with GPU debugging enabled may also pause kernels launched by other processes.

13.7.2 Thread control

- The focus on thread feature in DDT isn't supported, as it can lock up the GPU. This means that it is not currently possible to control multiple GPUs in the same process individually.

13.7.3 General

- DDT supports only versions 3.2, 4.0 and 4.1 of the NVIDIA CUDA toolkit.

- X11 cannot be running on any GPU used for debugging. (Any GPU running X11 will be excluded from device enumeration.)
- Attaching to an already running CUDA job is not currently supported if it has already commenced using the GPU.
- You must compile with "-g -G" to enable GPU debugging - if you do not your program will run through the contents of kernels without stopping.
- Debugging 32-bit CUDA code on a 64-bit host system is not supported.
- The debugger enforces blocking kernel launches.
- It is not yet possible to spot unsuccessful kernel launches or failures. An error code is provided by `getCudaLastError()` in the SDK which you can call in your code to detect this. Currently the debugger cannot check this without resetting it, which is not desirable behaviour.
- Device memory allocated via `cudaMalloc()` is not visible outside of the kernel function.
- Not all illegal program behaviour can be caught in the debugger - e.g. divide-by-zero.
- Device allocations larger than 100 MB on Tesla GPUs, and larger than 32 MB on Fermi GPUs, may not be accessible.
- Breakpoints in divergent code may not behave as expected.
- Debugging applications with multiple CUDA contexts running on the same GPU is not supported
- If CUDA environment variable `CUDA_VISIBLE_DEVICES <index>` is used to target a particular GPU, then make sure X server is not running on any of the GPUs. If X is running then reduce the `<index>` count by one more since the GPU running X is not visible to the application when running under the debugger.
- Known Issue: When debugging CUDA, gcc 4.5 is not supported in CUDA toolkit 4.0 or earlier (default for Ubuntu 11.04 and above). A workaround is to use older GNU packages such as gcc 4.4, which is available for Ubuntu, and then specify this compiler to `nvcc`:

```
nvcc -compiler-bindir=/usr/local/gcc-44
```

13.7.4 Pre sm_20 GPUs

For GPUs that have SM type less than `sm_20` (or when code is compiled targeting SM type less than `sm_20`), the following issues may apply.

- GPU code targeting less than SM type `sm_20` will inline all function calls. This can lead to behaviour such as not being able to step over/out of subroutines .
- Debugging applications using textures is not supported on GPUs with SM type less than `sm_20`.
- If you are debugging code in device functions that get called by multiple kernels, then setting a breakpoint in the device function will insert the breakpoint in only one of the kernels.

13.7.5 Toolkit 3.1

In addition to the issues above, the following issues are also present in version 3.1 of the CUDA toolkit.

- NVIDIA CUDA Toolkit 3.1 does not support debugging of multiple GPUs. (To debug multiple GPUs, please upgrade to toolkit 3.2).
- Host memory allocated with `cudaMallocHost()` is not visible to the debugger.
- Multi-threaded applications may not work in toolkit 3.1.
- CUDA 3.1 does not fully support sm_20 GPUs and above, so you will experience the same issues as with pre sm_20 GPUs.
- Memory debugging is not supported for fermi cards while using CUDA 3.1.

13.8 GPU Language Support

Presently DDT supports CUDA toolkits 3.1, 3.2, 4.0 and 4.1.

In addition to the native `nvcc` compiler, a number of other compilers are supported.

At this point in time, debugging of OpenCL is not supported on the device.

13.8.1 CAPS HMPP

CAPS HMPP 3.0 code can be debugged in both the host and the GPU. The “stop on kernel” launch feature can be used to identify the launch points of HMPP codelets, and breakpoints inside CAPS HMPP codelets can also be directly inserted.

The required compilation options and environment variables to the compiler are:

```
hmpg g -f -k ifort basic.f90 -o basic
```

Stepping inside a GPU codelet or pausing is supported with CAPS HMPP 3.0 and above.

The host code can be debugged as normal, subject to providing “-g” flags to the compilers as normal.

DDT will syntax highlight HMPP pragmas detected in the source.

F90 arrays and expressions are correctly displayed by DDT, including multi-dimensional arrays and arrays with (eg.) negative lower bounds.

Known issue: Local loop index variables may be incorrect, depending on the loop structure – notably nested loops may be flattened to a single loop and the mapping to a code's original index variables is not currently available.

13.8.2 Cray OpenMP Accelerator Extensions

Cray OpenMP Accelerator Extensions are fully supported by Alinea DDT. Code pragmas are highlighted, most variables are visible within the device, and stepping and breakpoints in the GPU code is supported. The compiler flag “-g -Gomp” is required for enabling device (GPU-based) debugging – “-O0” should **not** be used, as this disables using of the device and runs the accelerated regions on the CPU.

Known issue: Pointers in accelerator code cannot be dereferenced in CCE 8.0.

13.8.3 PGI Accelerators and CUDA Fortran

PGI Accelerator applications can be debugged when running on the host processor. To debug accelerator code it is recommended to target execution on the host process only – with the “-ta=host” compiler flag.

PGI CUDA Fortran can be debugged using Allinea DDT, by adding the “-Mcuda=emu” flag to the compiler. In this case, the CUDA will also run on the host CPU, and will use as many threads as there are host cores, allowing many thread issues to be detected.

Known issue: debugging inside the GPU is not supported – for both models, debugging is performed on CPU versions of the GPU code.

14 Offline Debugging

Offline debugging is a mode of running Allinea DDT in which an application is run, under the control of the debugger, but without user intervention and without a user interface.

There are many situations where running under this scenario will be useful, for example when access to a machine is not immediately available and may not be available during the working day. The application can run with features such as tracepoints and memory debugging enabled, and will produce a report at the end of the execution.

14.1 Using Offline Debugging

To launch DDT in this mode, the “-offline” argument and a filename is specified. A filename with a “.html” extension will cause a HTML version of the output to be produced, in other cases a plain text report is generated.

```
ddt -offline myjob.html -n 4 myapplication arg1 arg2
```

or

```
ddt -offline myjob.txt -n 4 myapplication arg1 arg2
```

Additional arguments can be used to set breakpoints (at which the stack of the stopping processes will be recorded before they are continued), or to set tracepoints at which variable values will be recorded.

Settings from your current DDT config file will be taken, unless over-ridden on the command line.

Command line options that are of the most significance for this mode of running are:

- “-ddtsession sessionfile” - run in offline mode using settings saved using the “Save Session” option from DDT’s session menu.
- “-n nnn” - run with “nnn” processes
- “-memory” enable memory debugging
- “-trace-at LOCATION[,N:M:P],VAR1,VAR2,...” - set a tracepoint at location, beginning recording after the N visit of each process to the location, and recording every M-th subsequent pass until it has been triggered P times. Record the value of variable VAR1, VAR2.
 - Example:
 - **main.c:22, -:2:-, x**
will record x every 2nd passage of line 22.
- “-break-at LOCATION[,N:M:P] ” - set a breakpoint at LOCATION (either file:line or function), optionally starting after the N-th pass, triggering every M passes and stopping after it has been triggered P times.
 - The stack traces of pausing processes will be recorded, before the processes are then made to continue, and will contain the variables of one of the processes in the set of processes that have paused.
 - Examples:
 - **-break-at main**
 - **-break-at main.c:22**

The application will run to completion, or to the end of the job.

When errors occur, for example an application crash, the stacktrace of crashing processes will be recorded to the offline output file. In offline mode, DDT will always act as if the user had clicked “Continue” if the continue option was available in an equivalent “online” debugging session.

14.2 Offline Report Output (HTML)

The output file is broken into three sections, Messages, Tracepoints, and Output. At the end of a job, DDT stitches together the three sections of the log output (tracepoint data, error messages, and program output) into one file – if DDT’s process is terminated abruptly, for example by the job scheduler, then these separate files will remain and the final single html report may not be created.

Allinea DDT Offline Log

Messages Tracepoints Output

Messages

[+] Expand All [-] Collapse All

#	Type	Time	Processes	Message																																										
1		00:00.010	n/a	Launching program /home/dschubert/code/examples/hello.																																										
2		00:03.675	0-3	Startup complete. ▶ Stacks																																										
3		00:08.067	0-3	Process stopped at breakpoint in main (hello.c:170). ▶ Stacks ▼ Stack for process 0 #0 main (argc=1, argv=0x7ffffffcf68, environ=0x7ffffffd290) at /home/dschubert/code/ddt/examples/hello.c:170 ▼ Local variables for process 0 (ranges shown for 0-3) <table border="1"> <thead> <tr> <th>Name</th> <th>Value</th> </tr> </thead> <tbody> <tr><td>argc</td><td>1</td></tr> <tr><td>argv</td><td>0x7ffffffcf68</td></tr> <tr><td>beingWatched</td><td>3 (-1-3)</td></tr> <tr><td>bigArray</td><td></td></tr> <tr><td>dest</td><td>32767 (0-32767)</td></tr> <tr><td>dynamicArray</td><td>0x80ba00</td></tr> <tr><td>environ</td><td>0x7ffffffd290</td></tr> <tr><td>i</td><td>1</td></tr> <tr><td>message</td><td>"Greetings from process 3!"</td></tr> <tr><td>my_rank</td><td>0 (0-3)</td></tr> <tr><td>p</td><td>4</td></tr> <tr><td>s</td><td>0</td></tr> <tr><td>source</td><td>4 (-153326624-4)</td></tr> <tr><td>status</td><td></td></tr> <tr><td>t2</td><td>0x603010</td></tr> <tr><td>tables</td><td></td></tr> <tr><td>tag</td><td>50</td></tr> <tr><td>test</td><td></td></tr> <tr><td>x</td><td>10000</td></tr> <tr><td>y</td><td>12</td></tr> </tbody> </table>	Name	Value	argc	1	argv	0x7ffffffcf68	beingWatched	3 (-1-3)	bigArray		dest	32767 (0-32767)	dynamicArray	0x80ba00	environ	0x7ffffffd290	i	1	message	"Greetings from process 3!"	my_rank	0 (0-3)	p	4	s	0	source	4 (-153326624-4)	status		t2	0x603010	tables		tag	50	test		x	10000	y	12
Name	Value																																													
argc	1																																													
argv	0x7ffffffcf68																																													
beingWatched	3 (-1-3)																																													
bigArray																																														
dest	32767 (0-32767)																																													
dynamicArray	0x80ba00																																													
environ	0x7ffffffd290																																													
i	1																																													
message	"Greetings from process 3!"																																													
my_rank	0 (0-3)																																													
p	4																																													
s	0																																													
source	4 (-153326624-4)																																													
status																																														
t2	0x603010																																													
tables																																														
tag	50																																													
test																																														
x	10000																																													
y	12																																													
4		00:08.299	n/a	Every process in your program has terminated.																																										

Messages Tracepoints Output

Tracepoints

#	Time	Tracepoint	Processes	Values
1	00:04.512	hello.c:92	0-3	x: 0
2	00:04.513	hello.c:92	0-3	x: 1000
3	00:04.515	hello.c:92	0-3	x: 2000
4	00:05.409	hello.c:92	0-3	x: 3000
5	00:05.411	hello.c:92	0-3	x: 4000
6	00:05.413	hello.c:92	0-3	x: 5000
7	00:06.313	hello.c:92	0-3	x: 6000
8	00:06.315	hello.c:92	0-3	x: 7000
9	00:06.316	hello.c:92	0-3	x: 8000
10	00:07.215	hello.c:92	0-3	x: 9000

Messages Tracepoints Output

Output

```

(00:07.527) 2: my rank is 2
(00:07.528) 0: my rank is 0
(00:07.528) 3: my rank is 3
(00:07.529) 1: my rank is 1
(00:07.529) 0-3: sizeof(int) = 4
(00:07.529) 0-3: sizeof(void*) = 8
(00:07.530) 2: My pid is 16549.
(00:07.530) 0: My pid is 16548.
(00:07.531) 1: My pid is 16547.
(00:07.531) 3: My pid is 16550.
(00:07.532) 0-2: I have 1 arguments.
(00:07.532) 0-3: How many did I say?
(00:07.533) 0-3: They are:
(00:07.533) 0-3: /home/dschubert/code/ddt/examples/hello
(00:07.572) 2-3: I can write to stderr too
(00:07.577) 2: sending message from (2)
(00:07.577) 3: sending message from (3)
(00:07.577) 1: sending message from (1)
(00:07.578) 0: waiting for message from (1)
(00:07.578) 0-1: I can write to stderr too
(00:07.578) 0: Greetings from process 1!
(00:07.578) 2: waiting for message from (2)
(00:07.578) 0: Greetings from process 2!
(00:07.579) 0: waiting for message from (3)
(00:07.579) 0: Greetings from process 3!
(00:09.020) 3: all done...(3)
(00:09.021) 2: all done...(2)
(00:09.022) 0: all done...(0)
(00:09.022) 1: all done...(1)
    
```

Fig 77: Offline Mode HTML output

Timestamps are recorded with the contents in the offline log, and even though the file is neatly organized into three sections, it remains possible to identify ordering of events from the timestamp.

The Messages section contains

- Error messages – for example if DDT’s Memory Debugging detects an error then the message and the stack trace at the time of the error will be recorded from each offending processes. Where multiple processes error within a short interval, one message is generated with the parallel stack view including the affected processes.
- Recorded breakpoints – the stack traces, local variables and stack arguments of a pausing process. Where multiple processes pause within a short interval, then variables across the pausing processes will be compared and a sparkline drawn, and the stacks displayed will be the merged parallel stack output from the pausing processes.

The Tracepoints section contains the output from tracepoints, similar to that shown in the tracepoints window in an online debugging session – including sparklines displaying the variable distribution.

Output from the application will be written to the Output section. For most MPIs this will not be identifiable to a particular process, but on those MPIs that do support it, DDT will report which processes have generated the output.

Identical output from the Output and Tracepoints section will, if received in close proximity and order, be merged in the output, where this is possible.

14.3 Offline Report Output (Plain Text)

Unlike the offline report in HTML mode, the plaintext mode does not separate the tracepoint, breakpoint and application output into separate sections.

Lines in the offline plain text report are identified as messages, standard output, error output, and tracepoints, as detailed in the Offline Report Output (HTML) section previously.

For example, a simple report could look like:

```

message (0-3): Process stopped at breakpoint in main (hello.c:97).
message (0-3): Stacks
message (0-3): Processes Function
message (0-3): 0-3      main (hello.c:97)
message (0-3): Stack for process 0
message (0-3): #0 main (argc=1, argv=0x7fffffff378,
environ=0x7fffffff388) at
  /home/ddt/examples/hello.c:97
message (0-3): Local variables for process 0 (ranges shown for 0-3)
message (0-3): argc: 1 argv: 0x7fffffff378 beingWatched: 0 dest: 7
environ: 0x7fffffff388 i: 0 message: ",!\312\t" my_rank: 0 (0-3) p: 4
source: 0 status: t2: 0x7ffff7ff7fc0 tables: tag: 50 test: x: 10000 y:
12

```

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
x86 and x86_64	RHEL 4, 5, 6; SLES 9, 10, 11; Fedora 4 and above, Ubuntu 8.04 and higher	<i>All known MPIs – including but not limited to:</i> SGI Altix , Bproc, Cray, Bull MPI 1 and 2, LAM-MPI, MPICH, Myricom MPICH-GM and MPICH-MX, Open MPI, Quadrics MPI, Platform (Scali) MPI, Scyld, Intel MPI, Slurm, MVAPICH	GNU, Absoft, Cray, Intel, Pathscale, PGI, Sun
IA64 (Itanium)	Redhat 4, SLES 10	<i>All</i>	GNU and Intel
IBM Power	AIX 5.3, 6.0 and 6.1 Redhat 6 Linux SLES 10 Linux	<i>All</i> , including IBM PE for AIX	GNU and IBM
Blue Gene/P	SLES 10 (frontend)	Native	GNU and IBM
NVIDIA CUDA Toolkit 3.2/4.0	Linux	<i>All</i>	CAPS HMPP, Cray OpenMP Accelerators, NVCC, PGI Accelerators, PGI CUDA Fortran
ARM v7	Ubuntu 11.04	<i>All</i>	GNU

B MPI Distribution Notes and Known Issues

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.

B.1 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.

B.2 Bull MPI

Bull MPI 1, MPI 2 and Bull X-MPI are supported. For Bull X-MPI select the Open MPI or Open MPI (Compatibility) MPIs, depending on whether ssh is allowed (in which case choose Open MPI) or not (choose Open MPI Compatibility mode).

Select *Bull MPI* or *Bull MPI 1* for Bull MPI 1, or *Bull MPI 2* for Bull MPI 2 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.

B.3 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.

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*.

B.4 Intel MPI

Select *Intel MPI* from the MPI implementation list. DDT has been tested with Intel MPI 2.0 and 3.0.

DDT also supports the Intel Message Checker tool that is included in the Intel Trace Analyser and Collector software. A plugin for the Intel Trace Analyser and Collector version 7.1 is provided in

DDT's plugins directory. Once you have installed the Intel Trace Analyser and Collector, you should make sure that the following directories are in your LD_LIBRARY_PATH:

```
{path to intel install directory}/itac/7.1/lib
{path to intel install directory}/itac/7.1/slib
```

The Intel Message Checker only works if you are using the Intel MPI. Make sure Intel's mpiexec is in your path, and that your application was compiled against Intel's MPI, then launch DDT, check the plugin checkbox and debug your application as usual. If one of the above steps has been missed out, DDT may report an error and say that the plugin could not be loaded.

Once you are debugging with the plugin loaded, DDT will automatically pause the application whenever Intel Message Checker detects an error. The Intel Message Checker log can be seen in the standard error (stderr) window.

Note that the Intel Message Checker will abort the job after 1 error by default. You can modify this by adding `-genv VT_CHECK_MAX_ERRORS 0` to the MPIRun arguments in the advanced section of DDT's Run dialog – see Intel's documentation for more details on this and other environment variable modifiers.

B.5 MPICH p4

Choose *MPICH Standard* as the MPI implementation.

B.6 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. This is achieved by adding `-MPDENV -HOME={homedir} LD_LIBRARY_PATH={ld-library-path}` to the *Arguments* area of the *Run* window. 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.

B.7 IBM PE

Ensure that poe is in your path when using DDT.

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

On AIX 5.3 TL12 when working via loadleveler, some users have experienced a POE imposed process count limit and been unable to debug above 5 MPI processes per node. This a known IBM issue and the default queue script for DDT and the “`ddt-client`” script used by it contains a workaround.

Select *IBM PE* as the MPI implementation.

B.8 Mvapich

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 *Run* window.

Mvapich 2 now offers “`mpirun_rsh`” instead of `mpirun` as a scalable launcher binary – to use this with DDT, on the *Session/System Options* page, select “Override default `mpirun` path” and enter “`mpirun_rsh`”.

B.9 OpenMPI

DDT has been tested with Open MPI 1.2.x, 1.3.x, 1.4.x, 1.5.x. Select *OpenMPI* from the list of MPI implementations.

There are three different Open MPI choices in the list of MPI implementations to choose from in DDT when debugging for Open MPI.

- *Default Open MPI* - this is the 'standard' way of debugging Open MPI, which conforms to the established MPI standards.
- *Open MPI (Compatibility)* - will let Open MPI launch DDT's daemons – can be used if the 'default' above does not work but does not use DDT's scalable tree – see known issue on Ubuntu.
- *Open MPI for Cray XT* - for Open MPI running on Cray XT systems. This method is fully able to use DDT's scalable tree infrastructure for large scale debugging.

Known issue: The version of OpenMPI packaged with Ubuntu has the OpenMPI debug libraries stripped. This prevents the *Message Queues* feature of DDT from working – and also requires the use of “Open MPI (Compatibility)” instead of “Open MPI”.

Known issue: With Open MPI 1.3.4 and Intel Compiler v11 – the default build will optimize away a vital call during the startup protocol which means the default Open MPI start up will not work. If this is your combination, either update your Open MPI, or select “Open MPI (Compatibility)” instead as the DDT MPI Implementation.

B.10 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.

B.11 SGI Altix

No known issues. Choose *SGI MP Toolkit* as the MPI implementation.

B.12 Cray XT/XE/XK

DDT has been tested with Cray XT4/5/6, XE6 and XK6 systems – with DDT submitting via the queue and also from within an interactive shell. DDT is able to launch and support debugging jobs in excess of 200,000 cores.

A number of template files for launching applications from within the queue (using DDT's job submission interface) are included in the distribution of DDT – these may require some minor editing to cope with local differences on your batch system.

Note that the default mode for compilers on this platform is to link statically. Section C.8 *Portland Group Compilers* describes how to ensure that DDT's memory debugging capabilities will work with the PGI compilers in this mode.

In order to statically link memory debugging for other compilers, such as GNU, Intel or Cray, on Suse 10/11 based systems, please use the `-Z muldefs` option to the linker, or the equivalent `-Wl, --allow-multiple-definition`.

Message queue debugging is not provided by the XT/XE/XK environment.

Known Issue: On the XK6 it may not be possible to automatically detect the CUDA toolkit/driver version – set the “DDT_FORCE_CUDA_VERSION” environment variable to 4.0 (or 4.1) as appropriate for your system to be able to debug inside GPU kernels.

14.3.1 Using DDT with Cray ATP (the Abnormal Termination Process)

DDT is compatible with the Cray ATP system, which will be default on some XE systems. This runtime addition to applications automatically gathers crashing process stacks, and can be used to let DDT attach to a job before it is cleaned up during a crash.

To be able to debug after a crash when an application is run with ATP but without a debugger, the `ATP_HOLD_TIME` environment variable should be initialized before launching the job – a value of 5 is (very) ample, even on a large Petscale system, giving 5 minutes for the attach to complete.

The following example shows the typical output of an ATP session.

```
n10888@kaibab:~> aprun -n 1200 ./atploop
Application 1110443 is crashing. ATP analysis proceeding...
Stack walkback for Rank 23 starting:
  _start@start.S:113
  __libc_start_main@libc-start.c:220
  main@atploop.c:48
  __kill@0x4b5be7
Stack walkback for Rank 23 done
Process died with signal 11: 'Segmentation fault'
View application merged backtrace tree file 'atpMergedBT.dot' with
'statview'
You may need to 'module load stat'.

atpFrontend: Waiting 5 minutes for debugger to attach...
```

At this point, DDT can be launched to debug the application.

DDT can attach using the Attaching dialogs described in Section 3.8.2, or given the PID of the aprun process, the debugging set can be specified from the command line.

For example, to attach to the entire job:

```
ddt --attach-mpi 12772
```

If a particular subset of processes are required, then the subset notation could also be used to select particular ranks.

```
ddt -attach-mpi 12772 -subset 23,100-112,782,1199
```

B.13 ScaleMP

Tested with VSMPICH2 (in DDT, choose MPICH2). No known issues.

C Compiler Notes and Known Issues

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.

C.1 Absoft

If you are using C-style include statements in your program the line numbers reported by DDT may be incorrect. To fix this problem replace the `#include` directives with an `INCLUDE` statement.

C.2 Berkeley UPC Compiler

The Berkeley UPC compiler is fully support by Allinea DDT.

C.3 Cray Compiler Environment

The Cray compiler is supported by DDT, but there are known issues with F90 array bounds.

Arrays with negative bounds may be incorrect – this has been fixed (in Feb 2011). Allocatable arrays may be incorrectly shown as allocated or unallocated – this has been fixed in (Feb 2011).

Call-frame information can also be incorrectly recorded, which can sometimes lead to DDT stepping into a function instead of stepping over it.

C++ pretty printing of the STL is not supported by DDT for the Cray compiler.

See CUDA/GPU debugging notes for details of Cray OpenMP Accelerator support.

Allinea DDT fully supports the Cray UPC compiler.

C.4 GNU

The compiler flag `-fomit-frame-pointer` should never be used in an application which you intend to debug. Doing so can 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 back end 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.

Allinea DDT also supports the GCC-UPC compiler.

C.5 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. This flag has been known to fail for `mpxlf95`, and so there may be circumstances when you must right click in the project navigator and add additional paths to scan for source files.

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.

Missing debug information in the binaries produced by XLF can prevent DDT from showing the values in Fortran pointers and allocatable arrays correctly, and assumed-size arrays cannot be shown at all. Please update to the latest compiler version before reporting this to support@allinea.com.

Sometimes, when a process is paused inside a system or library call, DDT will be unable to display the stack, or the position of the program in the Code view. To get around this, it is sometimes necessary to select a known line of code and choose *Run to here*. If this bug affects you, please contact support@allinea.com.

OpenMP loop variables are often optimized away and not present when debugging.

DDT has been tested against the C compiler xlc version 10.0 and Fortran/Fortran 90 version 12.1 – on both Linux and AIX. Note that x1C (C++) is not fully supported on AIX.

To view Fortran assumed size arrays in DDT you must first right click on the variable, select *Edit Type..*, and enter the type of the variable with its bounds (e.g. `integer arr(5)`).

C.6 Intel Compilers

DDT has been tested with versions 10, 11 and 12.

If you do not see stack traces for allocations in the *View Pointer Details* window try re-compiling your program with the `-fpp` argument (this enables frame pointers).

Some 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.

The Intel compiler doesn't always provide enough information to correctly determine the bounds of some Fortran arrays when they are passed as parameters, in particular the lower-bound of assumed-shape arrays.

The Intel OpenMP compiler will *always* optimise parallel regions, regardless of any `-O0` settings. This means that your code may jump around unexpectedly while stepping inside such regions, and that any variables which may have been optimised out by the compiler may be shown with nonsense values. There have also been problems reported in viewing thread-private data structures and arrays. If these affect you, please contact support@allinea.com.

Files with a `.F` or `.F90` extension are automatically preprocessed by the Intel compiler. This can also be turned on with the `-fpp` command-line option. Unfortunately, the Intel compiler does not include the correct location of the source file in the executable produced when preprocessing is used. If your Fortran file does not make use of macros and doesn't need preprocessing, you can simply rename its extension to `.f` or `.f90` and/or remove the `-fpp` flag from the compile line instead. Alternatively, you can help DDT discover the source file by right clicking in the *Project Files* window and then selecting *Add/view source directory* and adding the correct directory.

Some versions of the compiler emit incorrect debug information for OpenMP programs which may cause some OpenMP variables to show as `<not allocated>`.

By default Fortran `PARAMETERS` are not included in the debug information output by the Intel compiler. You can force them to be included by passing the `-debug-parameters all` option to the compiler.

Known Issue: If compiling static binaries (for example on a Cray XT/XE machine) then linking in the DDT memory debugging library is not straight forward for F90 applications. You will need to manually re-run the last `"ld"` command (as seen with `"ifort -v"`) to include `"-L{ddt-path}/lib/64 -ldmalloc"` in two locations – both immediately prior to where `"-lc"` is located, and also include the `"-zmuldefs"` option at the start of the `"ld"` line.

Pretty printing of STL types is not supported for the Intel 10 compiler.

Pretty printing of STL types for the Intel 11 and 12 compiler is almost complete – STL sets, maps and multi-maps cannot be fully explored – only the total number of items is displayed; other data types are unaffected.

To disable pretty printing set the environment variable `DDT_DISABLE_PRETTY_PRINTING` to 1 before starting DDT. This will enable – in the case of, for example, the incomplete `std::set` implementations – you to manually inspect the variable.

C.7 Pathscale EKO compilers

Known issues: The default Fortran compiler options may 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 compile and link with the following flags:

`-w1, --export-dynamic -TENV:frame_pointer=ON -funwind-tables`

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

When using the Fortran compiler, you may have to place breakpoints in `myfile.i` instead of `myfile.f90` or `myfile.F90`. We are currently investigating this; please let us know if it applies to your code.

Procedure names in modules often have extra information appended to them. This does not otherwise affect the operation of DDT with the Pathscale compiler.

The Pathscale 3.1 OpenMP library has an issue which makes it incompatible with programs that call the `fork` system call on some machines.

Some versions of the Pathscale compiler (e.g. 3.1) do not emit complete DWARF debugging information for typedef'ed structures. These may show up in DDT with a void type instead of the expected type.

Multi-dimensional allocatable arrays can also be given incorrect dimension upper/lower bounds – this has only been reproduced for large arrays, small arrays seem to be unaffected. This has been observed with v3.2 of the compiler, newer/older versions may also exhibit the same issue.

C.8 Portland Group Compilers

DDT has been tested with Portland Tools 9, 10 and 11.

Known issues: 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.

The PGI compiler may emit incorrect line number information for templated C++ functions or omit it entirely. This may cause DDT to show your program on a different line to the one expected, and also mean that breakpoints may not function as expected.

The PGI compiler does not emit the correct debugging tags for proper support of inheritance in C++, which prevents viewing of base class members.

When using memory debugging with statically linked PGI executables (`-Bstatic`) because of the in-built ordering of library linkage for F77/F90, you will need to add a `localrc` file to your PGI installation which defines the correct linkage when using DDT and (static) memory debugging. To your `{pgi-path}/bin/localrc` append the following:

**`switch -Bstaticddt is
help(Link for DDT memory debugging with static binding)`**

```

helpgroup(linker)
append(LDARGS=--eh-frame-hdr -z muldefs)
append(LDARGS=-Bstatic)
append(LDARGS=-L{DDT-Install-Path}/lib/64)
set(CRTL=$if(-Bstaticddt, -ldmalloc -lc -lns$(PREFIX)c
-l$(PREFIX)c, -lc -lns$(PREFIX)c -l$(PREFIX)c))
  set(LC=$if(-Bstaticddt, -ldmalloc -lgcc -lgcc_eh -lc -lgcc
-lgcc_eh -lc, -lgcc -lc -lgcc));

```

pgf90 -help will now list -Bstaticddt as a compilation flag. You should now use that flag for memory debugging with static linking.

This does not affect the default method of using PGI and memory debugging, which is to use dynamic libraries.

Note that some versions of ld (notably in SLES 9 and 10) silently ignore the --eh-frame-hdr argument in the above configuration, and a full stack for F90 allocated memory will not be shown in DDT. You can work around this limitation by replacing the system ld, or by including a more recent ld earlier in your path. This does not affect memory debugging in C/C++.

When you pass an array splice as an argument to a subroutine that has an assumed shape array argument, the offset of the array splice is currently ignored by DDT. Please contact support@allinea.com if this affects you.

DDT may show extra symbols for pointers to arrays and some other types. For example if your program uses the variable ialloc2d then the symbol ialloc2d\$sd may also be displayed. The extra symbols are added by the compiler and may be ignored.

The Portland compiler also wraps F90 allocations in a compiler-handled allocation area, rather than directly using the systems memory allocation libraries directly for each allocate statement. This means that bounds protection (Guard Pages) cannot function correctly with this compiler.

For information concerning the Portland Accelerator model and debugging this with DDT, please see the Chapter 13 (CUDA Debugging) of this userguide.

C.9 SGI Compilers

Allinea DDT supports the SGI UPC compiler.

D Platform Notes and Known Issues

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.

D.1 GNU/Linux Systems

When using a 64-bit Linux 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.

Some 64-bit GNU/Linux systems which have a bug in the GNU C library (specifically `libthread_db.so.1`) which can crash the debugger when debugging multi-threaded programs. Check with your Linux distribution for a fix. As a workaround you can try compiling your program as a statically linked executable using the `-static` compiler flag.

Some GNU/Linux distributions (e.g. RHEL 4) that use the Native POSIX Threads Library (NPTL) do not provide a static NPTL library. Instead the static threads library uses LinuxThreads instead. The thread debugging library (`libthread_db.so.1`) may not include support for LinuxThreads so you will be unable to debug statically linked multi-threaded programs on these distributions. We recommend you do not compile your programs as statically linked executables unless unavoidable.

Attaching to a stopped process does not work with some Linux kernel versions (e.g. the one shipped with Fedora 7) due to a kernel bug. This also stops DDT automatically attaching to forked processes when Stop on fork is enabled.

DDT may hang when restoring a GDB checkpoint on RHEL 4. This is due to a deadlock in the GNU C Library. At the time of writing there is no available fix.

Single stepping with the Redhat 4 kernel 2.6.9-89 is not 100% reliable – occasionally a kernel bug will cause at least two steps to be taken instead of one.

For the ARM architecture – breakpoints can be unreliable and will randomly be passed without stopping for some multicore processors (including the NVIDIA Tegra 2) unless a kernel option (fix) is built-in. The required kernel option is:

```
CONFIG_ARM_ERRATA_720789=y
```

This option is not present by default in many kernel builds.

D.2 IBM AIX Systems

The *Step Threads Together* and *Focus on Thread* features are unavailable on AIX due to a lack of operating system support.

When stepping through certain system library calls the program may run freely instead. This is presently known to occur with the `memset` library call, for example. This is due to a limitation of AIX.

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

The stop on fork and stop on exec feature is not supported on this platform.

D.3 Intel Itanium

Heap Overflow Detection is not supported on Itanium-based machines. You may find that your program stops with SIGBUS errors when this option is turned on. However, 'Fence Post' checking is still available.

D.4 IBM Blue Gene/P

To run DDT for Blue Gene we recommend you install the V1R2 update from IBM available from <https://www-304.ibm.com/systems/support/bluegene/index.html>

DDT must be installed in a directory that is visible from the front end node(s), the service nodes and the I/O nodes.

If your Blue Gene does not allow the I/O nodes to connect directly to ports on the front end then DDT must be run from a service node rather than the front end. Login to the service node using:

```
ssh -Y bgsn
```

then run DDT from the command line:

```
bgsn$ /gps/fs2/frontend-2/ddt-3.1-21691/bin/ddt
```

In order to allow many compute node processes per IO node to be debugged, it is recommended to switch on the “Shared symbol cache”. In the “Session/Options/System Settings” page, tick the “Shared symbol cache” button. This allows multiple debuggers (one per process) to share internal memory pages that contain debug information of the processes, and saves memory on the IO nodes. This is a sensible default for BG/P users.

DDT supports no more than 256 compute cores per IO node – in shared symbol cache mode – and considerably fewer without this setting. To debug more than 256 cores, spread the compute cores over more nodes by using DUAL or SMP mode.

Message queue debugging is not supported, as IBM do not provide a 32-bit library version to support this.

Attaching to running processes is only supported on the Blue Gene architecture if LaunchMON is installed.

Watchpoints are not presently supported on this platform.

E General Troubleshooting and Known Issues

If you have any problems with DDT, please take a look at the topics in this section – 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.

E.1 General Troubleshooting

E.1.1 Problems Starting the DDT GUI

If DDT is unable to start this is usually due to 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.

E.1.1 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 *Run Without MPI Support* check box has not been checked. 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 checked this box 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
- 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.

E.1.1 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 `DDTMPDIRUN` 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.

Sometimes 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.

E.2 Starting a Program

E.2.1 DDT says it can't find your hosts or the executable

This can happen when attempting to attach to a process running on other machines. Ensure that the host name(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.

E.2.2 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 B *MPI Distribution Notes and Known Issues*.

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`.

If your home directory is not accessible by all the nodes in your cluster then your jobs may fail to start in this fashion. To resolve the problem open the file `~/ . ddt / config . ddt` in a text editor. Change the shared directory option in the `[startup]` section so it points to a directory that is

available and shared by all the nodes. If no such directory exists, change the `use session cookies` option to `no` instead.

E.2.3 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 B *MPI Distribution Notes and Known Issues* for a workaround and email support@allinea.com for further assistance.

E.3 Attaching

E.3.1 Running processes don't show up 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.

E.4 Source Viewer

E.4.1 No variables or line number information

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

E.4.2 Source code does not appear when you 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 fixed width font such as *Courier* 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 for further instruction.

If the problem persists, contact support@allinea.com.

E.5 Input/Output

E.5.1 Output to `stderr` is not displayed

DDT automatically captures anything written to `stdout` / `stderr` and displays it. Some shells (such as `csh`) 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.

E.6 Controlling a Program

E.6.1 Program jumps forwards and backwards when stepping through it

If you have compiled with any sort of optimisations, the compiler will shuffle your programs instructions into a more efficient order. This is what you are seeing. We always recommend compiling with `-O0` when debugging, which disables this behaviour and other optimisations.

If you are using the Intel OpenMP compiler, then the compiler will generate code that appears to jump in and out of the parallel blocks regardless of your `-O0` setting. Stepping inside parallel blocks is therefore not recommended for the faint-hearted!

E.6.2 DDT sometimes stop responding when using the *Step Threads Together* option

DDT may stop responding if a thread exits when the *Step Threads Together* option is enabled. This is most likely to occur on Linux platforms using NPTL threads. This might happen if you tried to *Play to here* to a line that was never reached – in which case your program ran all the way to the end and then exited.

A workaround is to set a breakpoint at the last statement executed by the thread and turn off *Step Threads Together* when the thread stops at the breakpoint. If this problem affects you please contact support@allinea.com.

E.7 Evaluating Variables

E.7.1 Some variables cannot be viewed when the program is at the start of a function

Some compilers produce faulty debug information, forcing DDT to enter a function during the *prologue* or the variable may not yet be in scope. In this region, which appears to be the first line of the function, some variables have not been initialised yet. To view all the variables with their correct values, it may be necessary to play or step to the next line of the function.

E.7.2 Incorrect values printed for Fortran array

Pointers to non-contiguous array blocks (allocatable arrays using strides) are not supported. If this issue affects you, please email support@allinea.com for a workaround or fix. There are also many compiler limitations that can cause this. See Appendix C for details.

E.7.3 Evaluating an array of derived types, containing multiple-dimension arrays

The *Locals*, *Current Line* and *Evaluate* views may not show the contents of these multi-dimensional arrays inside an array of derived types. However, you can view the contents of the array by clicking on its name and dragging it into the evaluate window as an item on its own, or by using the MDA

E.7.4 C++ STL types are not pretty printed

The pretty printers provided with Allinea DDT are compatible with GNU compilers, and Intel C++ version 12 and above.

E.8 Memory Debugging

E.8.1 The View Pointer Details window says a pointer is valid but doesn't show you 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 C of this appendix for more details.

The Intel compiler may need the `-fp` argument to allow you to see stack traces on some machines.

If this happens with another compiler, please contact support@allinea.com with the vendor and version number of your compiler.

E.8.2 “mprotect fails” error when using memory debugging with guard pages

This can happen if your program makes more than 32768 allocations – a limit in the kernel prevents DDT from allocating more protected regions than this. You can set this limit manually by logging in as root and executing `echo 1048576 >/proc/sys/vm/max_map_count`, or another limit of your choice.

E.8.3 Allocations made before or during MPI_Init show up in Current Memory Usage but have no associated stack back trace

Memory allocations that are made before or during `MPI_Init` appear in Current Memory Usage along with any allocations made afterwards. However the call stack at the time of the allocation is not recorded for these allocations and will not show up in the Current Memory Usage window.

E.8.4 Deadlock when calling printf or malloc from a signal handler

The memory allocation library calls (e.g. `malloc`) provided by the memory debugging library are not async-signal-safe unlike the implementations in recent versions of the GNU C library.

POSIX does not require `malloc` to be async-signal-safe but some programs may expect this behaviour. For example a program that calls `printf` from a signal handler may deadlock when memory debugging is enabled in DDT since the C library implementation of `printf` may call `malloc`.

The web page below has a table of the functions that may be safely called from an asynchronous signal handler:

http://www.opengroup.org/onlinepubs/009695399/functions/xsh_chap02_04.html#tag_02_04_03

E.8.5 Program runs more slowly with Memory Debugging enabled

The Memory Debugging library performs more checks than the normal runtime's memory allocation routines – that's what makes it a debugging library! However those checks also makes it slower. If your program is running too slow when Memory Debugging is enabled there are a number of options you can change to speed it up.

Firstly try reducing the *Heap Debugging* option to a lower setting (e.g. if it is currently on *High*, try changing it to *Medium* or *Low*).

You can increase the heap check interval from the default of 100 to a higher value. The heap check interval controls how many allocations may occur between full checks of the heap (which may take some time). A higher setting (1000 or above) is recommended if your program allocates and deallocates memory very frequently (e.g. inside a computation loop).

You can disable the *Store backtraces for memory allocations* option, at the expense of losing backtraces in the *View Pointer Details* and *Current Memory Usage* windows.

E.9 Message Queues

E.9.1 When viewing messages queues after attaching to a process you 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/libtvmmpich.so ddt
```

Your MPI documentation should give you the file name for your implementation.

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.

Any message queue problems specific to a single MPI are listed in Appendix B .

E.10 Miscellaneous

E.10.1 The Fortran Module Browser is missing

This is because not all platforms support every feature that DDT has and so they are disabled by removing the window/tab by from DDT's interface. The Fortran modules browser is not supported on AIX.

E.10.2 Application working directory

For scalar programs, DDT will launch your application from the directory that you launched DDT from. For parallel programs, this is also – mostly – true, but with one important difference: DDT launches your program by running the `mpirun` command (or equivalent) from the directory DDT started from. Thus, if your `mpirun` command is fairly conventional, your program will start from the directory DDT started, but if `mpirun` sets a different launch directory, your program would start from that directory.

E.11 Obtaining Support

If this guide hasn't helped you, then 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 (for < 100 processes)
```

or this:

```
ddt -log log.ddt (for > 100 processes)
```

Then simply reproduce the problem using as few processors and commands as possible and close DDT as usual. On some systems this file might be quite large; if this is the case, please compress it using a program such as `gzip` or `bzip2` before attaching it to your email.

If your problem can only be replicated on large process counts, then please do not use the `-debug` option as this will generate very large log files – it will usually be sufficient to just use the `-log` option.

F Index

AIX.....	105, 107, 115
Align Stacks.....	52
Altix.....	105
Apple.....	35
Arguments.....	24
ARM.....	115
Attaching.....	29
Command Line.....	32
Hosts File.....	31
Backtrace.....	52
Blue Gene/P	116
Bounds Checking.....	76
Bproc.....	106
Breakpoints.....	47
Conditional.....	48
Deleting.....	49
Saving.....	49
Buffer overflow.....	41
Bull MPI	106
CAPS HMPP	100
Complex Numbers.....	60
Configuration.....	11, 26
Configuration.....	
Site Wide.....	13
Consistency Checking.....	
Heap.....	78
Core Files.....	29
Cray OpenMP Accelerator Extensions.....	100
Cray XT/XE/XK	109
Cross-Process Comparison.....	68
CUDA.....	94
CUDA Fortran	101
Data.....	
Changing.....	61
Editor	21
End Session.....	25
Fence Post Checking.....	80, 116
Floating Licences.....	10
Font.....	22
Font	21
Fortran Modules.....	59
Function Listing.....	40
GPU.....	94
GPU Language Support	100
Heap Overflow	80, 116
HMPP.....	100
Hotkeys.....	46
HP MPI.....	106
IBM PE.....	107
Inf.....	56
Input.....	71
Installation.....	8
Installation.....	
Graphical.....	8

Text-Mode.....	9
Intel Compiler	108
Intel Compilers.....	25, 112
Intel Message Checker.....	106
Intel MPI	106
Irix.....	108
Job Submission.....	32
Cancelling.....	32
Configuration.....	14
Regular Expression.....	19, 32
Jump To Line.....	40
Double Clicking.....	42
Licensing.....	
Licence Files.....	10
Licence Server.....	86
Multi Process Licence.....	24
Single Process Licence.....	26
Log File.....	122
Macros.....	59
Main Window.....	
Overview.....	37
Memory Debugging.....	76
Check Validity.....	79
Configuration.....	76
Libraries.....	77
Memory Statistics.....	82
mprotect fails.....	121
Memory leaks.....	41
Memory Usage.....	80
Message Queues.....	73
MPI.....	
Configuration.....	11
Function Counters.....	91
History/Logging.....	91
MPI Rank.....	42
MPI Ranks.....	69
mpirun.....	25
Troubleshooting.....	117
MPICH.....	26
p4.....	107
p4 mpd.....	107
MVAPICH.....	108
nvcc.....	94
NVIDIA.....	94
NVIDIA Tegra 2.....	115
OpenCL.....	94
OpenGL.....	66, 67
OpenMP.....	
OMP_NUM_THREADS.....	27
OpenMP Accelerator.....	100
OpenMPI	108
Parallel Stack View.....	53
PGI Accelerators.....	101
Pointer to shared (PTS).....	61
Pointers.....	61
Portland Group.....	113
Process Groups.....	42

Deleting.....	42
Programming errors.....	21
Raw Command.....	70
Registers.....	
Viewing.....	69
Remote Users.....	35
Restarting.....	46
Running a Program.....	23
Scyld.....	108
Search.....	39
Session.....	
Saving.....	38
Session Menu.....	46
SGI.....	108
Shared arrays.....	61
Signal Handling.....	56
Division by zero.....	56
Floating Point Exception.....	56
Segmentation fault.....	56
SIGFPE.....	56
SIGILL.....	56
SIGPIPE.....	56
SIGSEGV.....	56
SIGUSR1.....	56
SIGUSR2.....	56
Single Stepping.....	46
SMP.....	
Performance.....	117
Source Code.....	38, 54
Source Code.....	
Editing.....	41
Missing Files.....	38
Searching.....	39
Stack Frame.....	52
Standard Error.....	71
Standard Output.....	71
Starting.....	46
Starting DDT.....	23
Static Analysis.....	41
Static checking.....	21
Step Threads Together.....	44
Stopping.....	46
Synchronizing Processes.....	49
Tab size	21
Tracepoints.....	51
Unused variables.....	41
UPC.....	61
Variables.....	41, 57
Variables.....	
Searching.....	39
VNC.....	36
Warning symbols.....	41
Watchpoints.....	50
Welcome Screen.....	23
X forwarding.....	35
X11.....	117