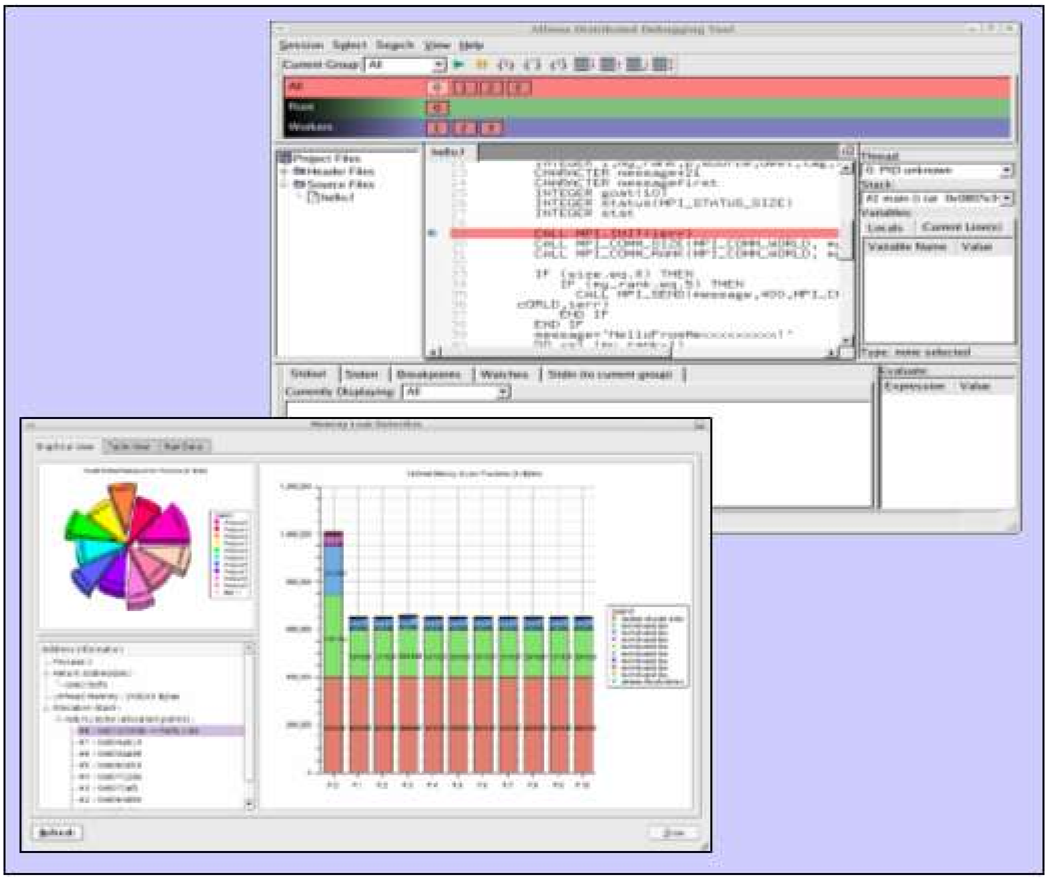
The Distributed Debugging Tool

Userguide



allinea
SCALE TO NEW HEIGHTS

Contents

1. Introduction

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

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

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

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

C, C++, Fortran and Fortran90/95/2003 are all supported by DDT, along with a large number of platforms, compilers and all known MPI libraries.

2. Installation

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

```
tar xvf ddt1.x-arch.tar

./install.sh
```

For the purposes of the rest of this document, the install directory that you choose will be referred to as $DDTPATH

During this phase of installation, you will be given a choice of where to install DDT. DDT can be installed by a single user in their home directory or by an administrator in a common directory where file permissions permit.

A number of environment variables are required for DDT to function correctly. Users can set these within their own accounts, or system administrators can provide appropriate universal set up scripts.

It is important to follow the instructions in the README file that is contained in the tar file.

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

Licence Files

Licence files should be stored as $DDTPATH/Licence, where $DDTPATH is the location you installed DDT (e.g. /home/mark/ddt). There is no need to set an environment variable called DDTPATH.

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

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

They may also use DDT_LICENSE_FILE as the environment variable (i.e. with American spelling).

The order of precedence when searching for licence files is:

- $DDT_LICENCE_FILE
- $DDT_LICENSE_FILE
- $DDTPATH/Licence.

If a licence file cannot be found in any of these places then the DDT frontend will not start and an error message will be displayed.
For remote MPI processes, we 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 frontend.

Time-limited evaluation licences are available from the Allinea website.

Floating Licences

For users with floating licences, the licensing daemon must be started prior to running DDT.

```
$DDTPATH/bin/licenceserver &
```

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

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

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

Log files can be generated for accounting purposes.

For more information on the Licence Server please see section 9 of this document.

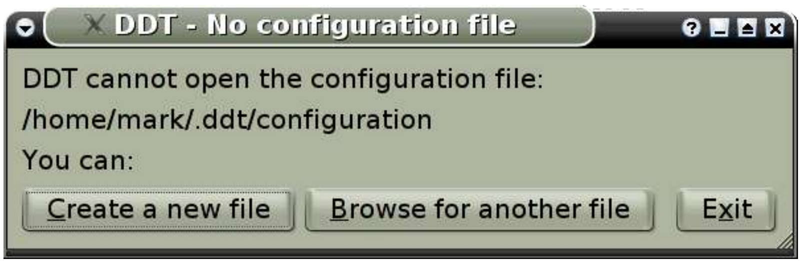
Getting Help

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

Support is also available from the support team – support@allinea.com

Configuration Wizard

In the past, setting DDT up for the first time might have required you to read and understand the documentation provided. As you're about to find out, things have changed. The first time you run DDT after installing it you may see this dialog:

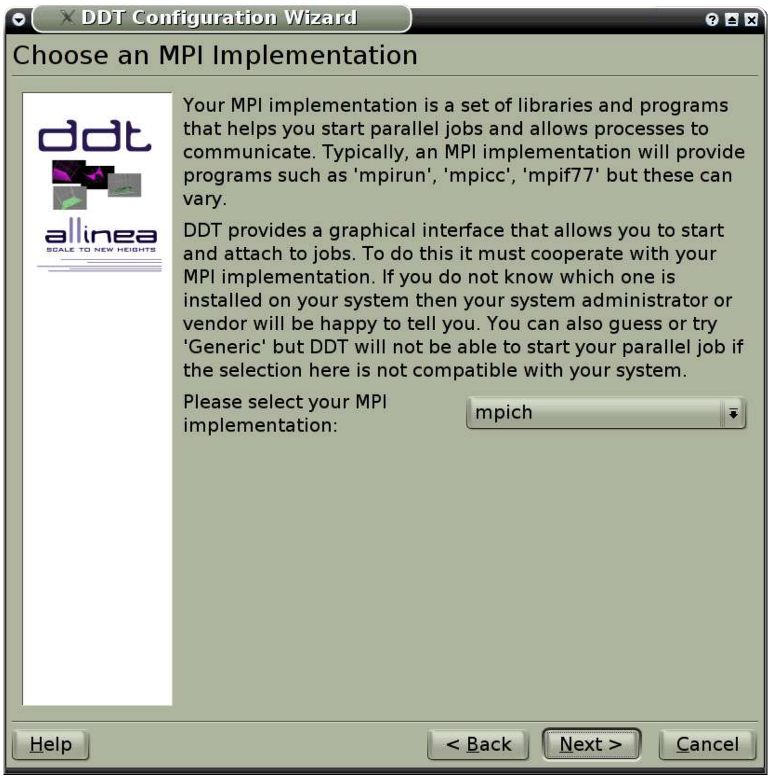

DDT - No configuration file

DDT cannot open the configuration file:
/home/mark/.ddt/configuration
You can:

Fig.1 Configuration file finder

DDT now has a Configuration Wizard that helps you to set it up to debug programs on your system, whether it is an individual workstation or a four thousand node super-cluster! Most settings will be automatically detected for you, so unless your system administrator has provided a configuration file for you to use, click on “Create a New File” and follow the simple instructions.
Note: this configuration wizard does not help you to set up DDT to submit jobs through a queue. Extensive documentation on the topic can be found elsewhere in this manual.
The first page welcomes you to the wizard and explains the process. Click on 'Next' to start the process, or 'Cancel' if you have changed your mind and would prefer to browse for a

configuration file. It's both quick and easy to use the Configuration Wizard, so we recommend clicking on 'Next'. If you're in a hurry, stop reading this and just click on every 'Next' until the wizard finishes! It'll probably do the right thing and you can always come back here through the 'Session->Configuration Wizard' menu item if things don't work out.
So, you'd rather take things step by step? Great! After the welcome page, you will either see the MPI Configuration page or the 'Attach List' page. Only users with parallel DDT licences will see the MPI Configuration page. If you don't see this page, you can obtain a trial MPI licence from our website, www.allinea.com to see what you're missing!
The page itself looks like this:



Choose an MPI Implementation

Your MPI implementation is a set of libraries and programs that helps you start parallel jobs and allows processes to communicate. Typically, an MPI implementation will provide programs such as 'mpirun', 'mpicc', 'mpif77' but these can vary.

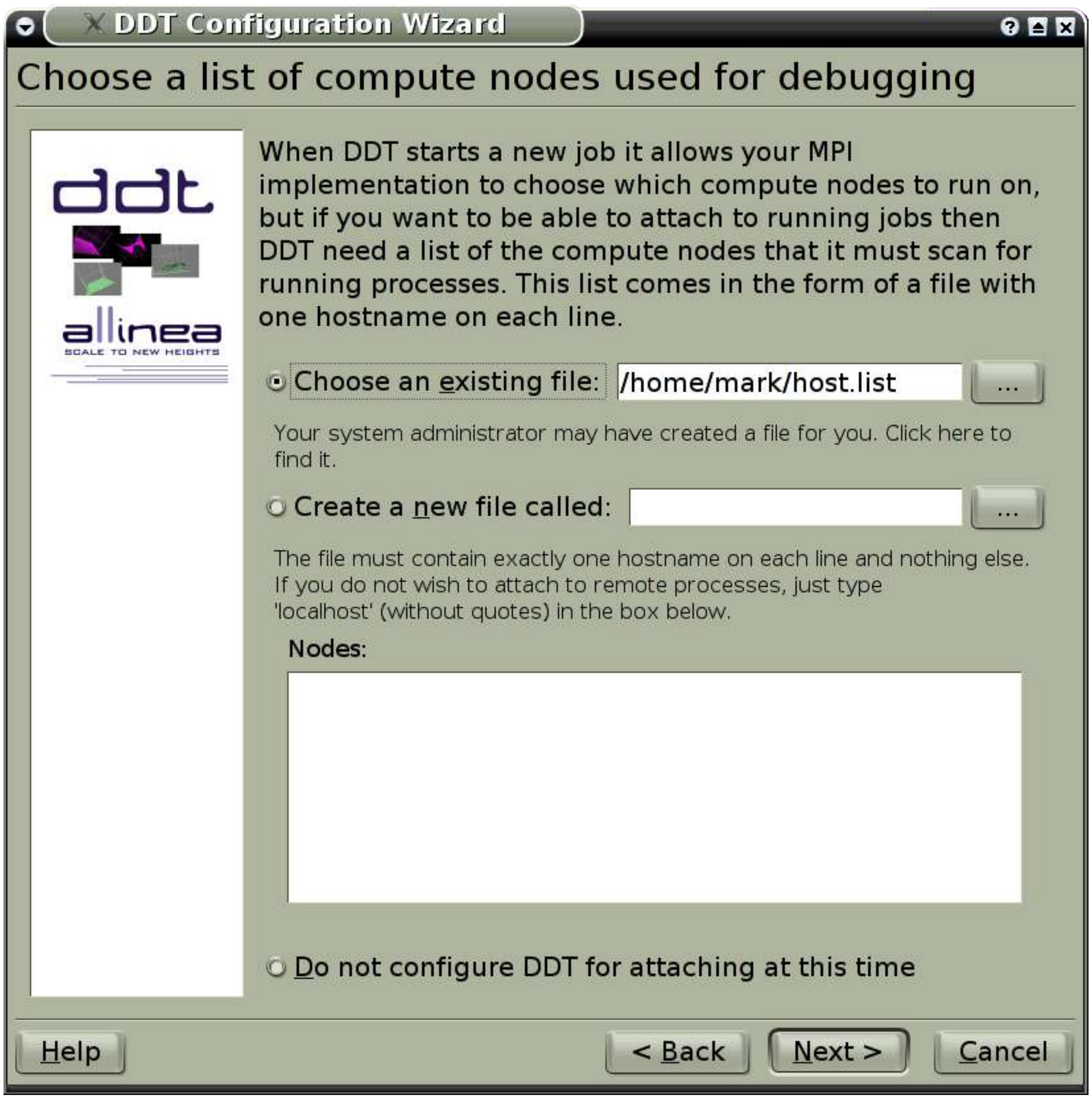
DDT provides a graphical interface that allows you to start and attach to jobs. To do this it must cooperate with your MPI implementation. If you do not know which one is installed on your system then your system administrator or vendor will be happy to tell you. You can also guess or try 'Generic' but DDT will not be able to start your parallel job if the selection here is not compatible with your system.

Please select your MPI implementation: mpich

Fig.2 Configuration wizard MPI selection

As you can see, there's a detailed description that we won't repeat here. Instead it's worth saying that if the selection box simply says "Click to select..." then DDT was not able to determine which MPI implementation was installed on your system. If you don't know yourself, your system administrator should be able to help you choose one of the options.
If you do know which MPI you have, and you're sure DDT has picked the wrong one, then go ahead and change it, but please contact bugs@allinea.com so that we can improve this feature in future releases! At the time of writing we are aware that MPICH-SMP is sometimes detected when in fact MPICH-SHMEM is the correct choice. If DDT selects MPICH-SMP for you and it doesn't work, try choosing the SHMEM variant instead.

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



Choose a list of compute nodes used for debugging

When DDT starts a new job it allows your MPI implementation to choose which compute nodes to run on, but if you want to be able to attach to running jobs then DDT need a list of the compute nodes that it must scan for running processes. This list comes in the form of a file with one hostname on each line.

[✓ Choose an existing file:] /home/mark/host.list

Your system administrator may have created a file for you. Click here to find it.

Create a new file called:

The file must contain exactly one hostname on each line and nothing else. If you do not wish to attach to remote processes, just type 'localhost' (without quotes) in the box below.

Nodes:

Do not configure DDT for attaching at this time

Fig.3 Configuration wizard attach list

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

```
comp00
comp01
comp02
comp03
```

Or perhaps like this:

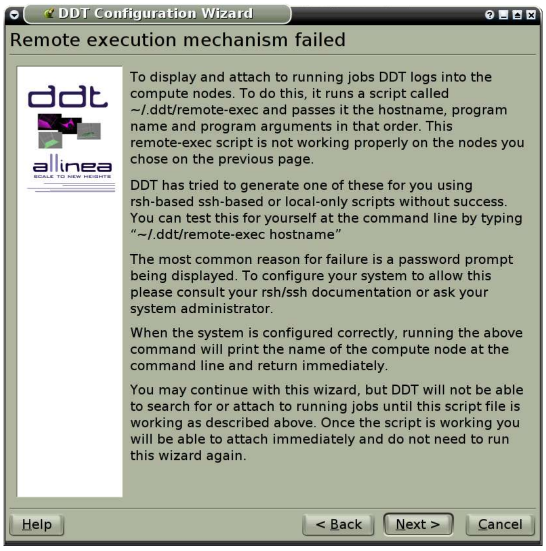
```
localhost
apple.mylab.com
pear.mylab.com
super.mylab.com
```

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

```
localhost
```

Hopefully your system administrator has a file like this for you and you can click on the '...' button next to “Choose an existing file”. If not, you can click on the '...' button next to “Create a new file”, choose a filname and then enter a list like the ones above into the big white box called “Nodes:”. This is quick and easy, but if you really don't want to do this, you can always

click on “Do not configure DDT for attaching at this time”. And, next time you want to attach, you can come back to this wizard and set up attaching before you do.
When you're satisfied, click on 'Next'. DDT will now try to find a way to reach these machines without needing you to type in a password. If it succeeds, you will go to a summary screen that tells you that you've finished and can now use DDT! If it fails, then DDT will explain what went wrong and how to correct it with a long explanation that looks something like this:



Remote execution mechanism failed

To display and attach to running jobs DDT logs into the compute nodes. To do this, it runs a script called ~/.ddt/remote-exec and passes it the hostname, program name and program arguments in that order. This remote-exec script is not working properly on the nodes you chose on the previous page.

DDT has tried to generate one of these for you using rsh-based ssh-based or local-only scripts without success. You can test this for yourself at the command line by typing “~/.ddt/remote-exec hostname”

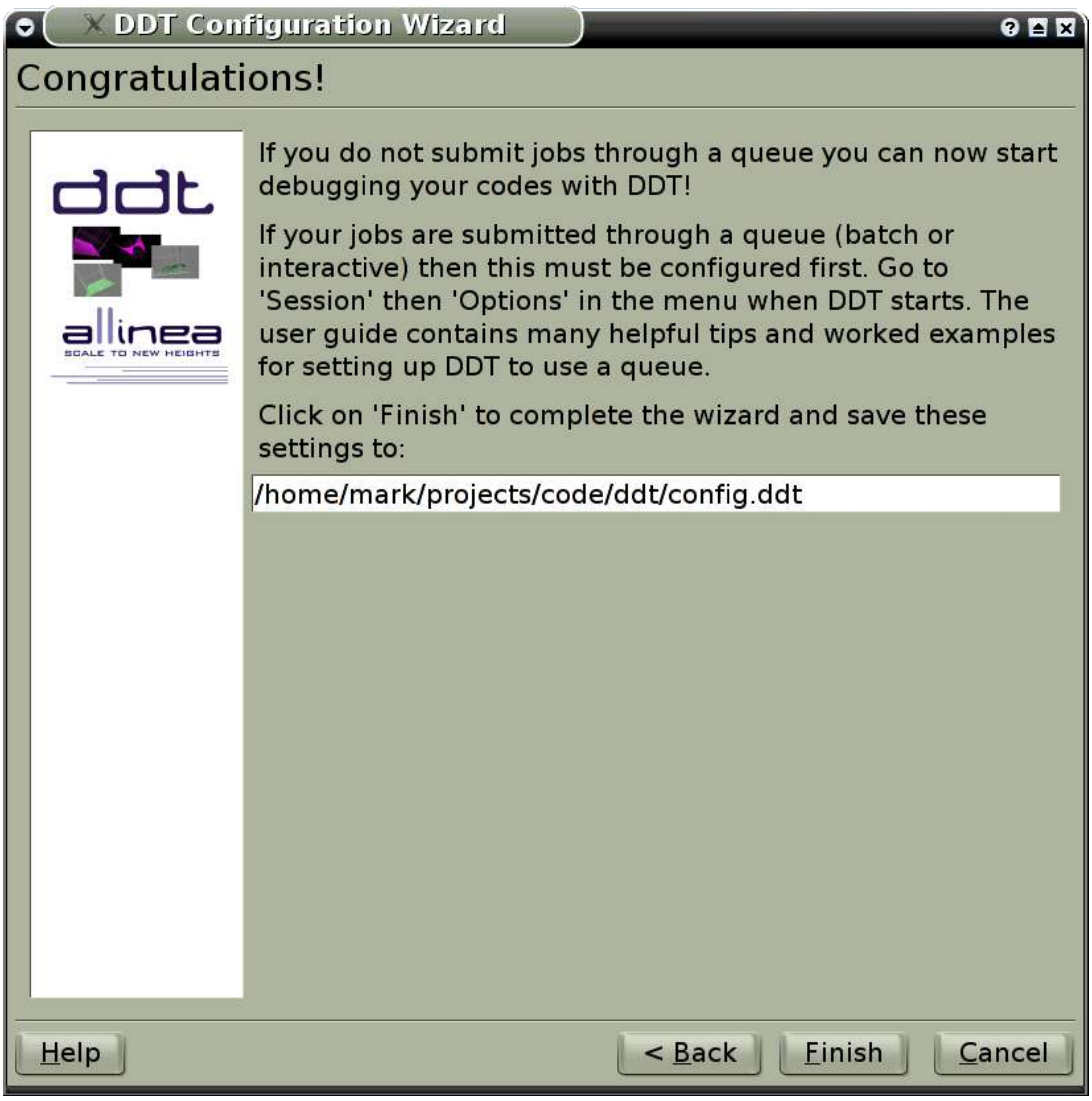
The most common reason for failure is a password prompt being displayed. To configure your system to allow this please consult your rsh/ssh documentation or ask your system administrator.

When the system is configured correctly, running the above command will print the name of the compute node at the command line and return immediately.

You may continue with this wizard, but DDT will not be able to search for or attach to running jobs until this script file is working as described above. Once the script is working you will be able to attach immediately and do not need to run this wizard again.

Fig.4 Remote execution error message

If you see this page, read it, understand it, try to perform the corrections it suggests. Our support staff will be happy to assist you, drop them a mail at support@allinea.com! We all want you to see the page that comes after that, the “Congratulations!” page. You'll know it when you see it, trust us! It'll look like this:



Congratulations!

If you do not submit jobs through a queue you can now start debugging your codes with DDT!

If your jobs are submitted through a queue (batch or interactive) then this must be configured first. Go to 'Session' then 'Options' in the menu when DDT starts. The user guide contains many helpful tips and worked examples for setting up DDT to use a queue.

Click on 'Finish' to complete the wizard and save these settings to:

/home/mark/projects/code/ddt/config.ddt

Fig.5 Configuration wizard complete

If you decided not to configure attaching, or if the remote-exec part failed then there will be a small red warning that you will not be able to attach to running programs, but all other features will be available. Click in 'Finish' to save these settings to the configuration file you selected, or 'Cancel' if you've changed your mind.
As stated before, this configuration wizard does not help you to set up DDT to submit jobs through a queue. If you are using a queuing system such as PBS on your cluster then you can find extensive documentation on setting DDT up to use these for debugging elsewhere in this manual.

3. Starting DDT

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

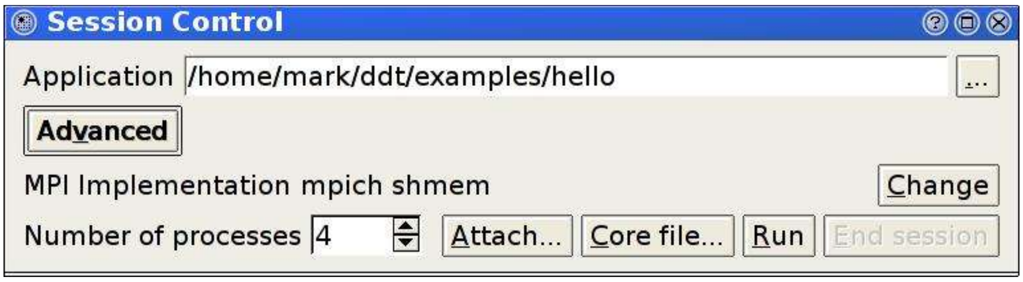
To start DDT simply type one of the following into a shell window:

```
ddt

ddt program_name

ddt program_name arguments
```

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

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

Debugging Multi-Process Programs



Application /home/mark/ddt/examples/hello

MPI Implementation mpich shmem

Number of processes 4

Fig.6 Session control dialog

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

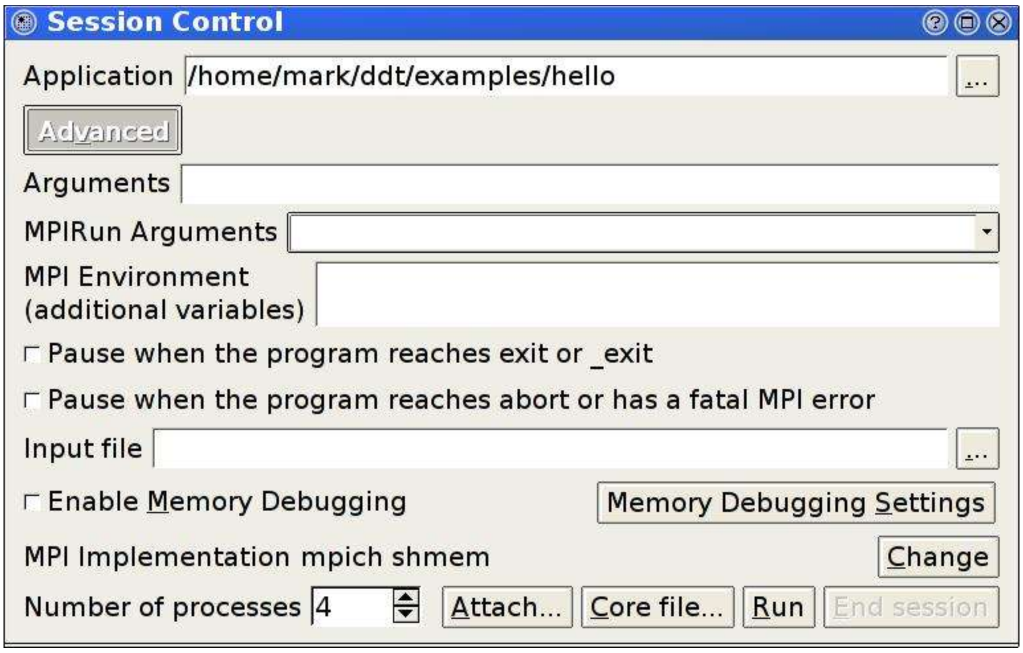
If your previous DDT session was debugging non-MPI codes, the view will be more compact than the above, and in this case you will be able to choose an MPI implementation and this will restore the full complement of parameter boxes.

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

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

Finally you should enter the number of processes that you wish to run. DDT supports over 256 processes but this is limited by your licence.

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


Session Control

Application /home/mark/ddt/examples/hello

Arguments

MPIRun Arguments

MPI Environment (additional variables)

Pause when the program reaches exit or _exit

Pause when the program reaches abort or has a fatal MPI error

Input file

Enable Memory Debugging

MPI Implementation mpich shmem

Number of processes 4

Fig.7 Advanced options

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

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

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

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

If your desired MPI command is not in the 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.

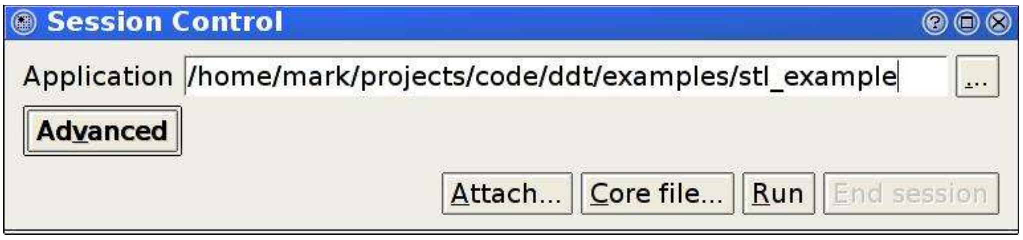
The two checkboxes for allow you to choose whether or not DDT will automatically pause your program when it looks like it is about to finish. If your program reaches one of the functions shown, DDT will ask you whether you want to pause the processes that have reached this point so you can see how they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

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

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

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

Debugging Single-Process Programs



Application: /home/mark/projects/code/ddt/examples/stl_example

Fig.8 Single-process session control dialog

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

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

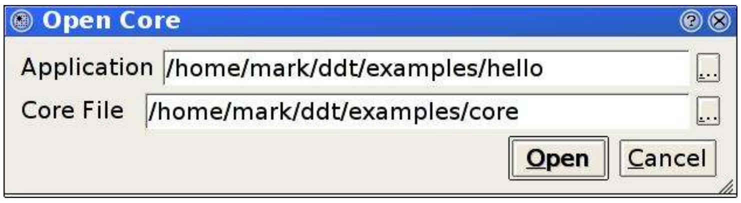
Select the application – either by typing the file name in, or selecting using the browser by clicking the “...” button. Arguments can be typed into the supplied box.

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

Finally press run to start your program.

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

Debugging A Core File



Application: /home/mark/ddt/examples/hello
Core File: /home/mark/ddt/examples/core

Fig.9 The open core dialog

DDT allows you to debug a single core file generated by your application. Core files are not supported on all platforms and debug interfaces. The supported platforms and debug interfaces are:

- Irix (SGI dbx)
- HP-UX (gdb)
- Linux (Intel idb, GNU gdb and Absoft Fx2)
- Solaris (Sun dbx)

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

Attaching To Running Programs

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

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

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

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

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

```
#!/bin/sh

rsh $*
```

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

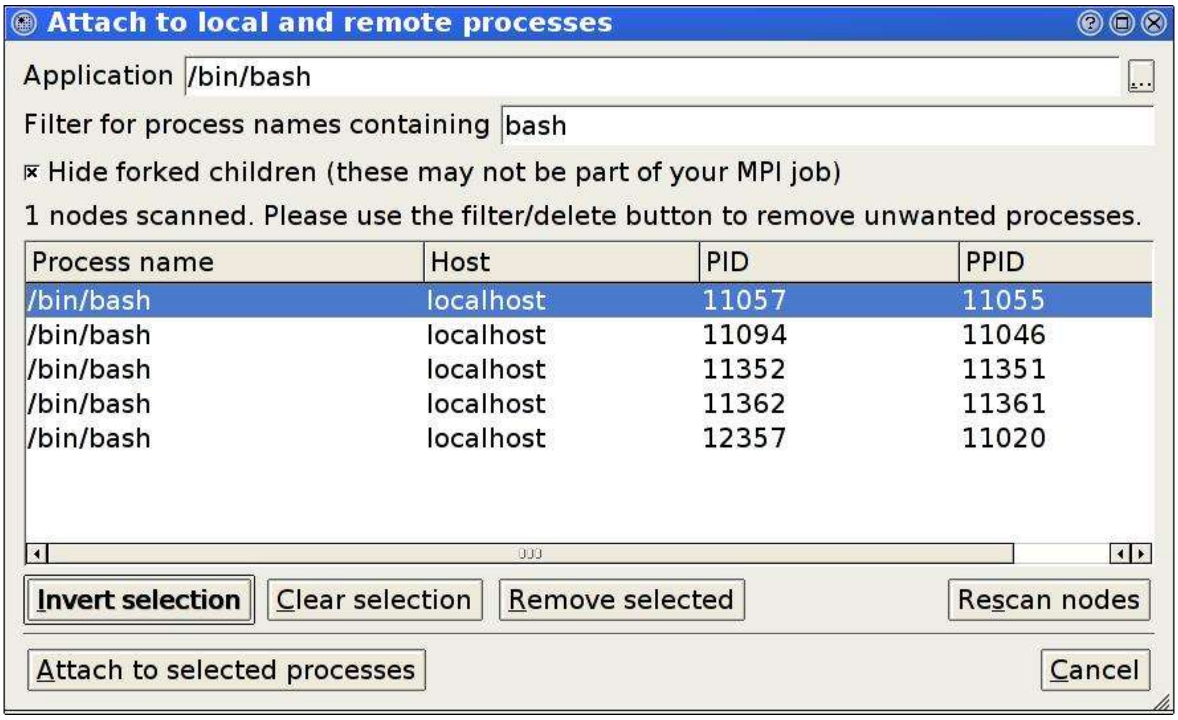
Once the script is set up (we advise testing it at the command line before using DDT) you must also provide a plain text file listing the nodes you want DDT to look for processes to attach to. If you ran the configuration wizard then you may have already made this file during that process, if not then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

```
localhost

comp00

comp01

comp02
```

```
comp03
```

This file can be placed anywhere you have read access to, and may be provided by your system administrator. DDT must be given the location of this file – to do this just set the `Attach hosts file` in the options window (`Session -> Configuration`). Each host name in this file will be sent to the remote-exec script as the `HOSTNAME` argument when DDT scans for and attaches to processes.

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


Attach to local and remote processes
Application
...
Filter for process names containing
Invert selection
Clear selection
Remove selected
Rescan nodes
Attach to selected processes
Cancel

Application: /bin/bash

Filter for process names containing: bash

☒ Hide forked children (these may not be part of your MPI job)

1 nodes scanned. Please use the filter/delete button to remove unwanted processes.

Process name	Host	PID	PPID
/bin/bash	localhost	11057	11055
/bin/bash	localhost	11094	11046
/bin/bash	localhost	11352	11351
/bin/bash	localhost	11362	11361
/bin/bash	localhost	12357	11020

Fig.10 Attach dialog

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

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

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

Because DDT is capable of attaching to several MPI and non-MPI programs in the same session, the rank numbers displayed may not be accurate – particularly if you have attached to a subset of the processes in your job. Allinea is working on methods to automatically determine process rank and/or display PID for processes with no rank. If this is an issue for you, please contact Allinea for information on and access to our feature development programme.

Configuring DDT With Queuing Systems

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

The basic stages in configuring DDT to work with a queue are:
1. Making a template script, and
2. Setting the commands that DDT will use to submit, cancel, and list queue jobs.

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

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

The Template Script

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

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

Tag	Purpose	After Submission Example
PROGRAM_TAG	Target path and filename	/users/ned/a.out
PROGRAM_ARGUMENTS_TAG	Arguments to target program	-myarg myval
NUM_PROCS_TAG	Total number of processes	16
NUM_NODES_TAG	Number of compute nodes	8
PROCS_PER_NODE_TAG	Processes per node	2

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

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

You would write:

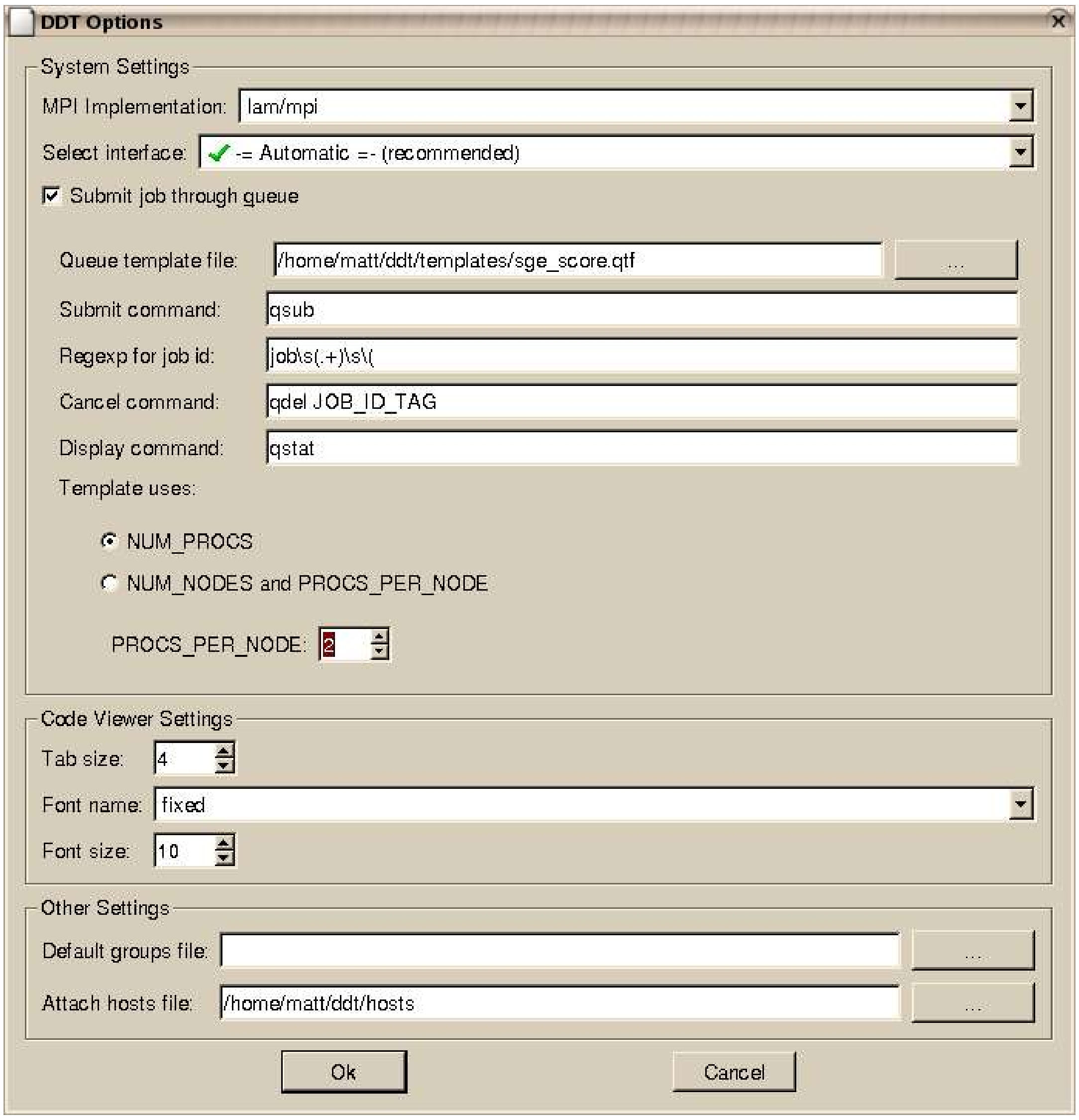
```
mpirun -np NUM_PROCS_TAG /opt/ddt/bin/ddt-debugger PROGRAM_ARGUMENTS_TAG
```

Configuring Queue Commands

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

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

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


DDT Options

System Settings

MPI Implementation: lam/mpi

Select interface: ✓ -= Automatic =- (recommended)

[✓] Submit job through queue

Queue template file: /home/matt/ddt/templates/sge_score.qtf

Submit command: qsub

Regexp for job id: job\s(.+)\s\(

Cancel command: qdel JOB_ID_TAG

Display command: qstat

Template uses:

[● NUM_PROCS]

○ NUM_NODES and PROCS_PER_NODE

PROCS_PER_NODE: 2

Code Viewer Settings

Tab size: 4

Font name: fixed

Font size: 10

Other Settings

Default groups file:

Attach hosts file: /home/matt/ddt/hosts

Fig.11 Queuing systems

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

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

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

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

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

Please note that for DDT v1.5 onwards an extra environment variable must be set whilst working with some queue systems: DDT_IGNORE_MPI_OUTPUT may need to be set to 1 prior to starting DDT.

Starting A Job In A Queue

Clicking submit from the usual session control dialog will display the queue status until your job starts. DDT will execute the display command every second and show you the standard output. If your queue display is graphical or interactive then you cannot use it here.

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

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

Using Custom MPI Scripts

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

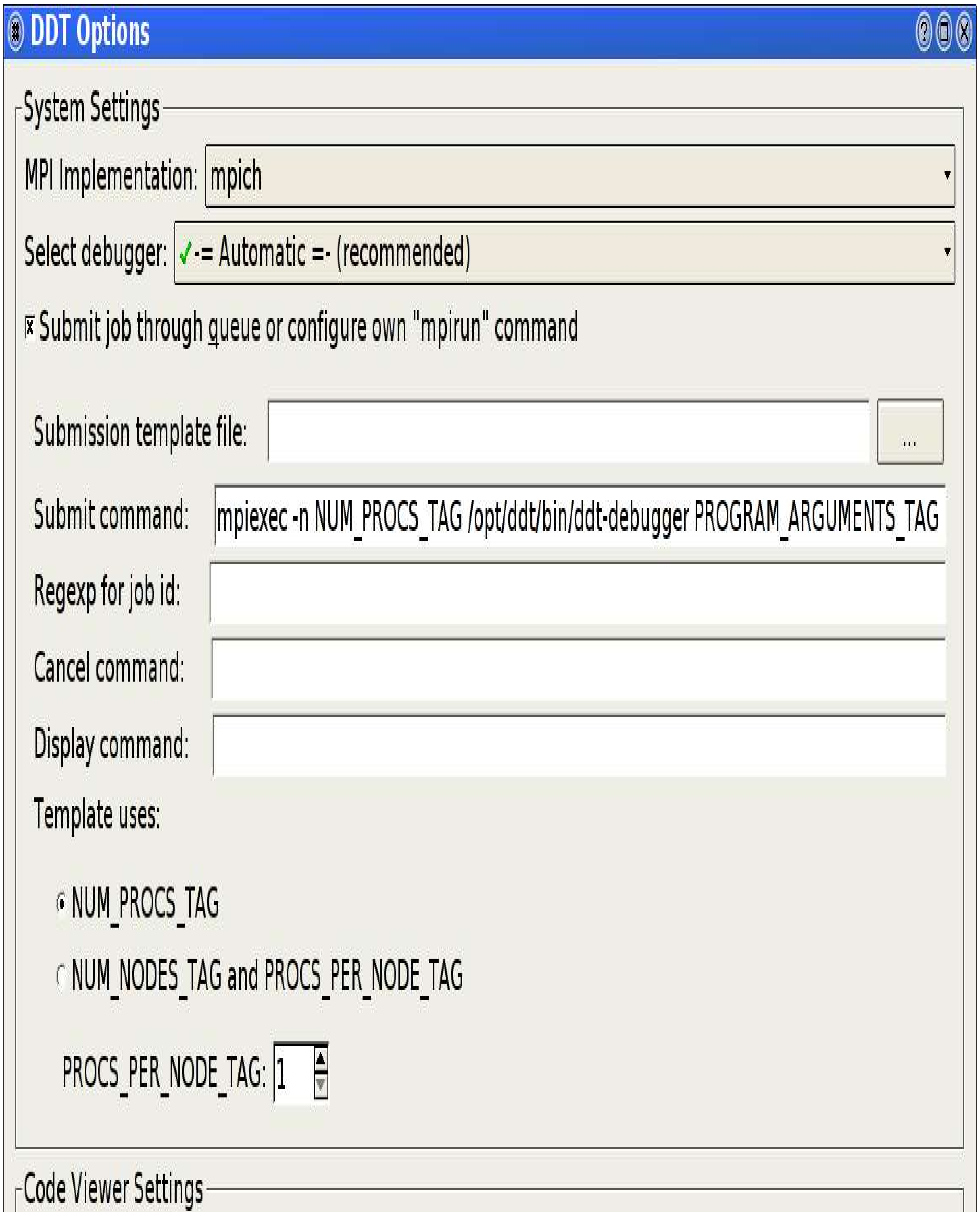
There are typically two ways you might want to start jobs using a custom script, and DDT supports them both. Firstly, you might pass all the arguments on the command line, like this:

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

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

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

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



DDT Options

System Settings

MPI Implementation: mpich

Select debugger: ✓-= Automatic =- (recommended)

☒ Submit job through queue or configure own "mpirun" command

Submission template file:

Submit command: mpiexec -n NUM_PROCS_TAG /opt/ddt/bin/ddt-debugger PROGRAM_ARGUMENTS_TAG

Regexp for job id:

Cancel command:

Display command:

Template uses:

● NUM_PROCS_TAG

○ NUM_NODES_TAG and PROCS_PER_NODE_TAG

PROCS_PER_NODE_TAG: 1

Code Viewer Settings

Fig.12 Using custom MPI scripts

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

1. The number of processes is replaced with NUM_PROCS_TAG
2. The name of the program is replaced by the full 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 config files like this every time you run your program – you just need to specify a template file. For the above example, the template file 'myfile.ddt' would contain the following:

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

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

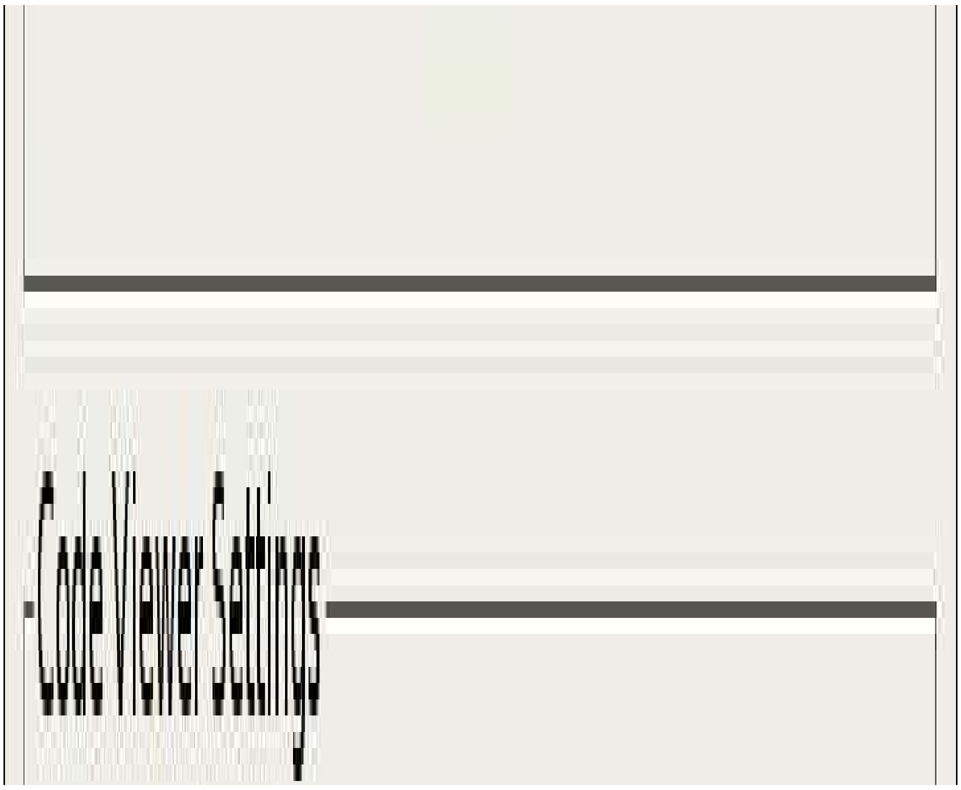


Fig.13 Using substitute MPI commands

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

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

Choosing The Right Debugger

DDT uses an enhanced version of GDB with complete F90 and F95 support, and this
We recommend that you choose the default debugger supplied by Allinea to obtain the best support on all platforms. Should you wish to use an alternative supported debugger, this can be configured from the session configuration menu.

Known issues with GDB can be found in the GDB release notes, but note that the GDB provided with DDT has resolved the Fortran issues noted in the GNU GDB bug tracking system.

The appendix details any known issues with the compilers.
http://www.gnu.org/software/gdb/download/ANNOUNCEMENT

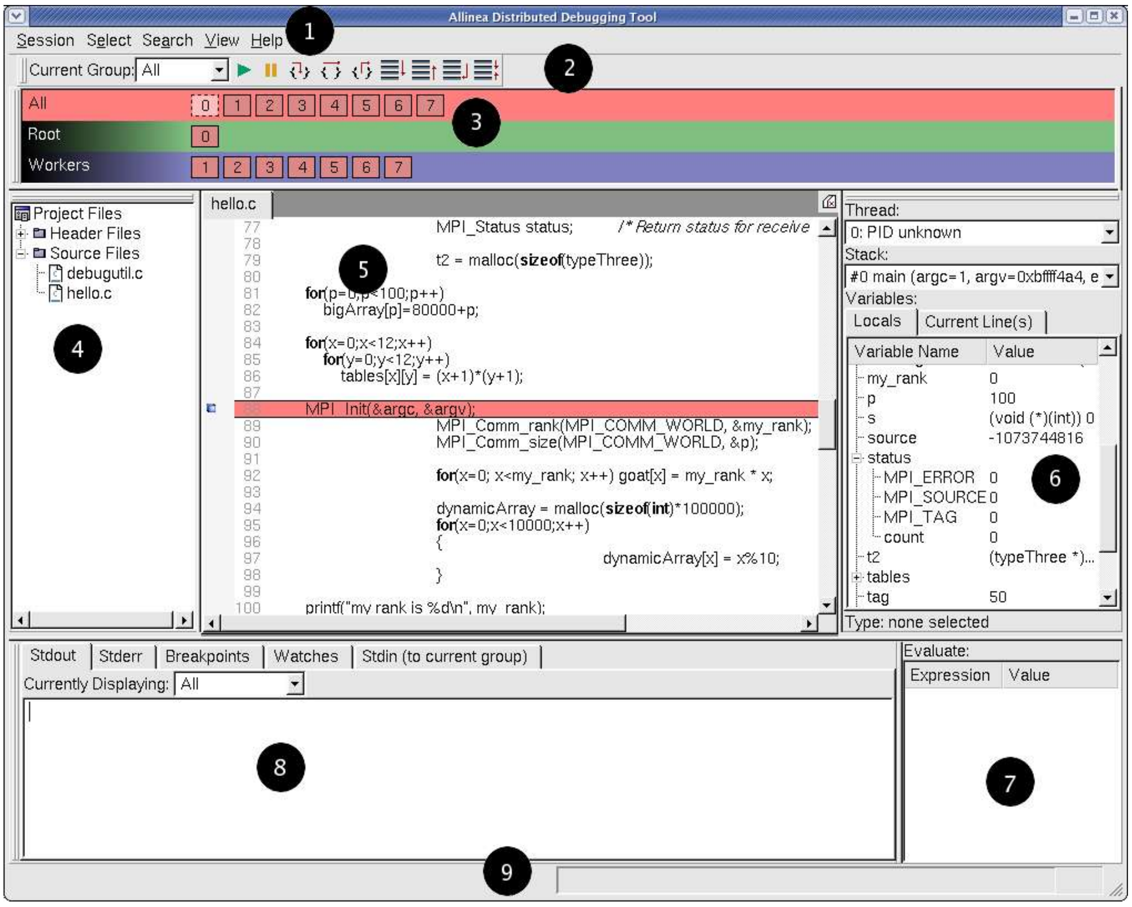
4. DDT Overview

DDT uses a multi-document interface which is common in many present day applications. This allows you to have many source files open, and to view one in the full workspace area, or to tile or cascade them. You may switch between these modes in the `Window` menu.

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

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

Key
(1) Menu bar
(2) Process controls
(3) Process group window (DDT-MP)
(4) File window
(5) Code window
(6) Variable window
(7) Evaluate window
(8) Output window
(9) Status bar



```
                MPI_Status status;      /* Return status for receive

                t2 = malloc(sizeof(typeThree));

    for(p=0;p<100;p++)
      bigArray[p]=80000+p;

    for(x=0;x<12;x++)
      for(y=0;y<12;y++)
        tables[x][y] = (x+1)*(y+1);

    MPI_Init(&argc, &argv);
                MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
                MPI_Comm_size(MPI_COMM_WORLD, &p);

                for(x=0; x<my_rank; x++) goat[x] = my_rank * x;

                dynamicArray = malloc(sizeof(int)*100000);
                for(x=0;x<10000;x++)
                {
                                dynamicArray[x] = x%10;
                }

    printf("my rank is %d\n", my_rank);
```

Variable Name	Value
my_rank	0
p	100
s	(void (*)(int)) 0
source	-1073744816
status	
MPI_ERROR	0
MPI_SOURCE	0
MPI_TAG	0
count	0
t2	(typeThree *)...
tables	
tag	50

Fig.14 DDT main window

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

Saving And Loading Sessions

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

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

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

Source Code

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

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

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

Finding Lost Source Files

On some platforms, not all source files are found automatically. This can also occur, for example, if the executable or source files have been moved since compilation. Extra directories to search for source files can be added by right clicking whilst in the project files window, and selecting “Add source directories”. After adding the directories necessary, right click again and select “Scan for more sources” - ensuring that there is a current process selected and paused before doing so.

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

Dynamic Libraries

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

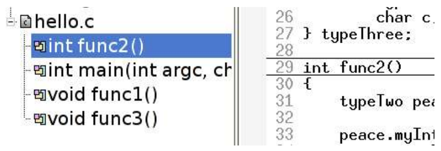
Finding Code Or Variables

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

Jump To Line

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

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



```
        char c
} typeThree;

int func2()
{
    typeTwo pea

    peace.myInt
```

Fig.15 Function listing

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

If your system has the program `etags` installed in the path, DDT will use this to provide faster, more accurate parsing of C++ files. If not, a default algorithm will be used that is less effective. Regardless of which method is used, some language constructs (particularly templated functions) may not be shown in this view.

Editing Source Code

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

```
export DDT_EDITOR=emacs

export DDT_EDITOR=xterm -e vi
```

5. Controlling Program Execution

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

In multi-process mode, most of the features described in this section are applied using process groups, which we describe now. For single process mode, the commands and behaviors are identical, but as there is only one process there is no need for groups and the group display itself will not be shown.

Process Control And Process Groups

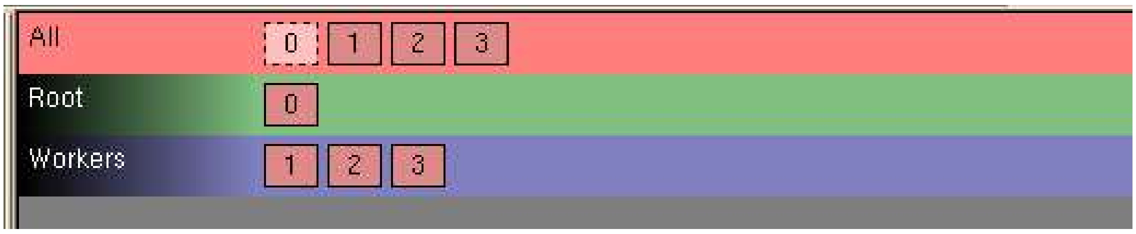


Fig.16 The process group viewer

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

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

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

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

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

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

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

Hotkeys

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

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

Starting, Stopping And Restarting A Program

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

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

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

Stepping Through A Program

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

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

When using step out be careful not to try and step out of the main function. Doing this will cause problems with the debugger most likely resulting in your program hanging. With some debuggers DDT will detect the defunct process and time out.

Setting Breakpoints

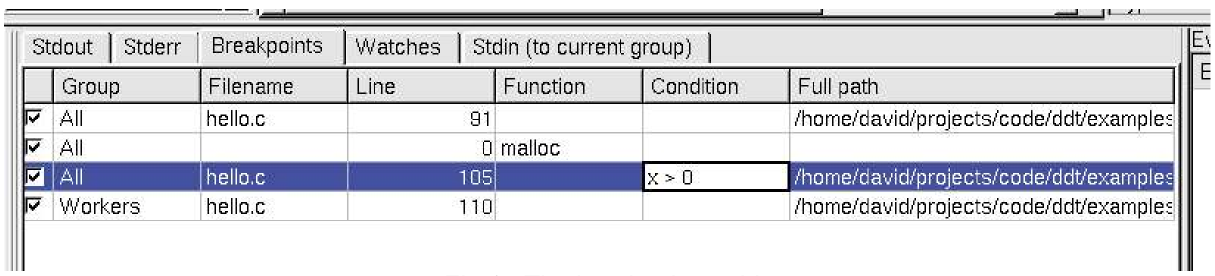
First locate the position in your code that you want to place a breakpoint at. If you have a lot of source code and wish to search for a particular function you can use the `Find`/`Find in files` dialog. Clicking the right mouse button in the code window displays a menu showing several options, including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

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

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

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

Conditional Breakpoints



	Group	Filename	Line	Function	Condition	Full path
✓	All	hello.c	91			/home/david/projects/code/ddt/examples
✓	All		0	malloc		
✓	All	hello.c	105		x > 0	/home/david/projects/code/ddt/examples
✓	Workers	hello.c	110			/home/david/projects/code/ddt/examples

Fig.17 The breakpoints table

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

Suspending Breakpoints

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

Deleting A Breakpoint

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

Loading And Saving Breakpoints

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

Synchronizing Processes

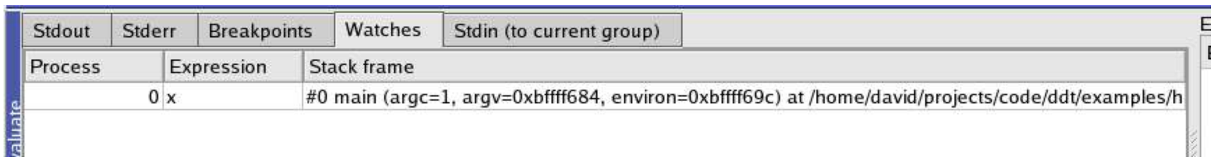
If the processes in a process group are stopped at different points in the code and you wish to re-synchronize them to a particular line of code this can be done by right clicking on the line at which you wish to synchronize them to and selecting “run to here”. This effectively sets all the processes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

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

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

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

Setting A Watch



Process	Expression	Stack frame
0	x	#0 main (argc=1, argv=0xbffff684, environ=0xbffff69c) at /home/david/projects/code/ddt/examples/h

Fig.18 The watches table

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

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

Examining The Stack Frame

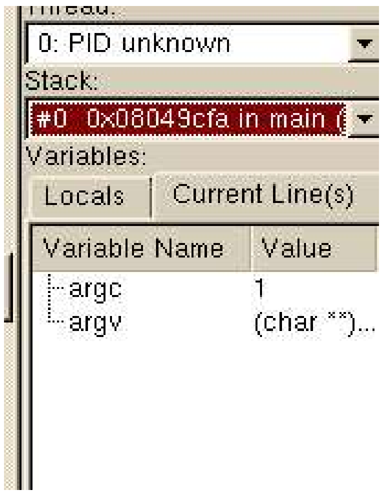


Fig.19 Selecting a stack frame

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

Align Stacks

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

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

Examining Threads

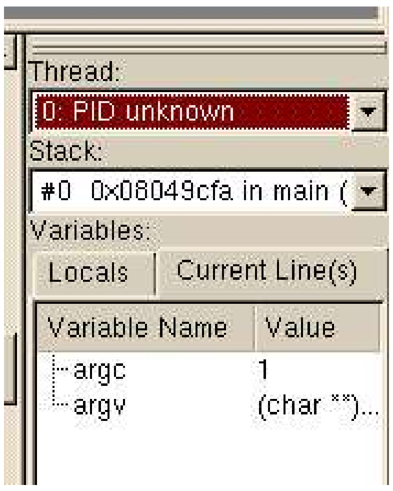


Fig.20 Selecting a thread

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

Browsing Source Code

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

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

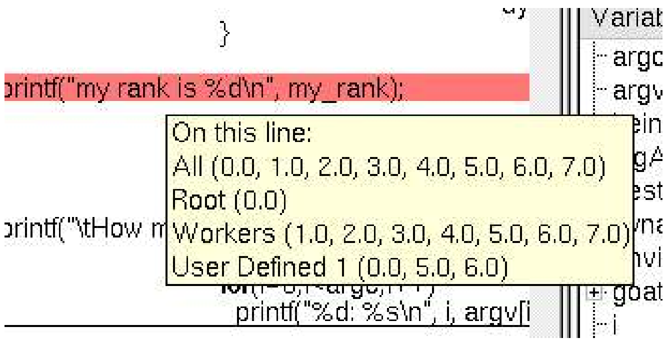
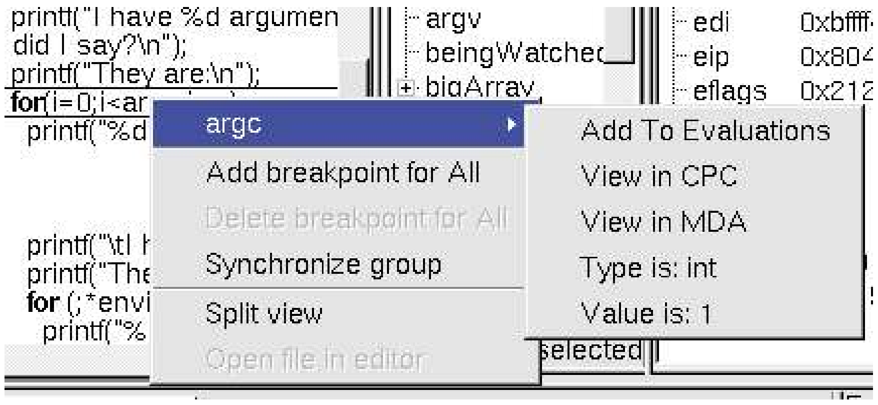

On this line:
All (0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0)
Root (0.0)
Workers (1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0)
User Defined 1 (0.0, 5.0, 6.0)

Fig.21 Process list tooltip

Right clicking above a term in the source code will cause DDT to probe that term and establish whether there is a matching variable or function.

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



Type is: int
Value is: 1

Fig.22 Right click menu – variable options

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

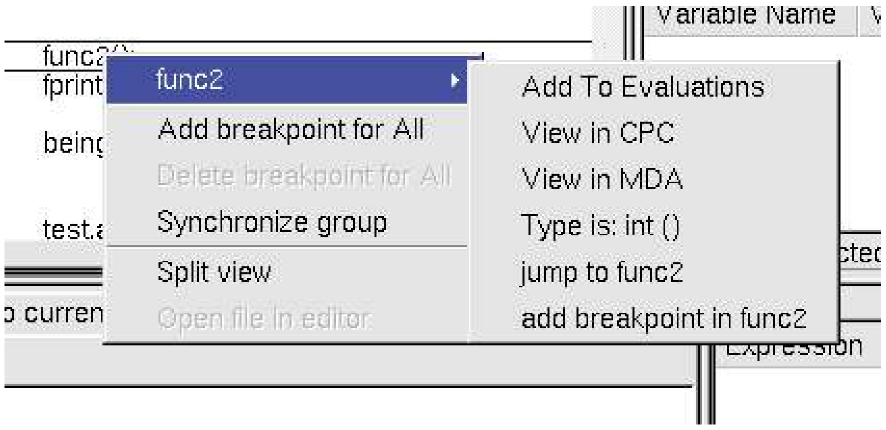


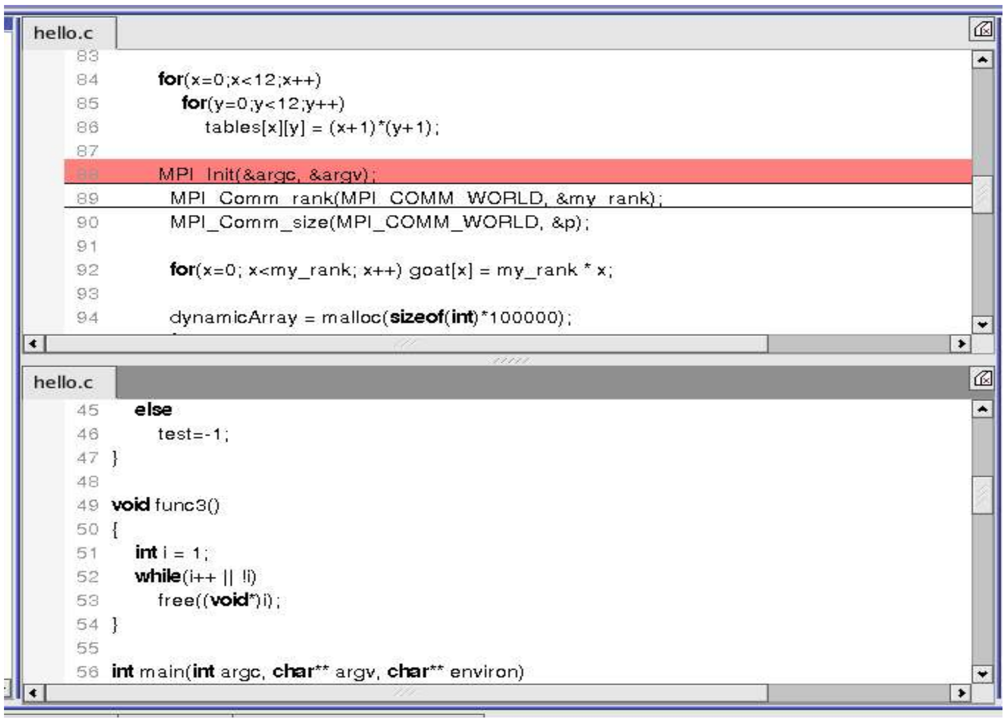
Fig.23 Right click menu – function options

Simultaneously Viewing Multiple Files

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

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

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



```
    for(x=0;x<12;x++)
      for(y=0;y<12;y++)
        tables[x][y] = (x+1)*(y+1);

    MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
     MPI_Comm_size(MPI_COMM_WORLD, &p);

     for(x=0; x<my_rank; x++) goat[x] = my_rank * x;

     dynamicArray = malloc(sizeof(int)*100000);
```

```
  else
    test=-1;
}

void func3()
{
  int i = 1;
  while(i++ || !i)
    free((void*)i);
}

int main(int argc, char** argv, char** environ)
```

Fig.24 Horizontal alignment of multiple source files

6. Variables And Data

The variable 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` tab displays all the variables referenced on the currently selected lines.

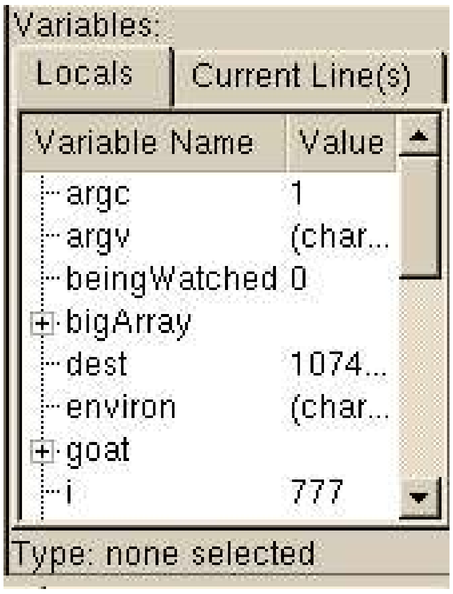


Fig.25 Displaying variables

Current Line

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

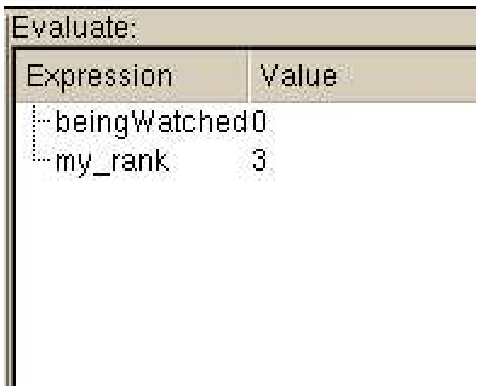
Local Variables

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

For Fortran codes the amount of data reported as local can be substantial – as this can include many global or common block arrays. Should this prove problematic, it is best to conceal this tab underneath the current line 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 debuggers, this is because they are considered to be global variables when defined in a common memory space.

Arbitrary Expressions And Global Variables



```
beingWatched 0
my_rank      3
```

Fig.26 Evaluating expressions

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

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

Viewing Data

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

Changing Data Values

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

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

Examining Pointers

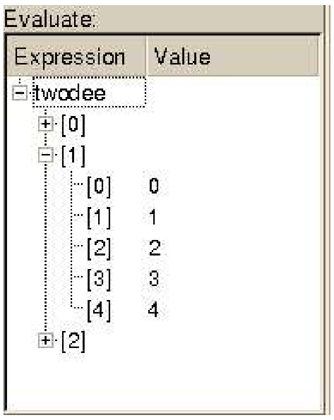
Pointer contents cannot normally be examined in the variables window but after dragging them into the evaluate window you can right click and select any of the following new options: view as vector, reference, or de-reference.

If a structure contains another pointer you must drag this pointer onto its own line in the evaluate window before you can start referencing/de-referencing it.

Multi-Dimensional Arrays in the Variable View

When viewing a multi-dimensional array in either the Local Variables, Current Line 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 fortan the opposite will be seen with arrays being displayed from right to left as you read it (so x,y,z would have z as the first column with y under each z cell etc.)

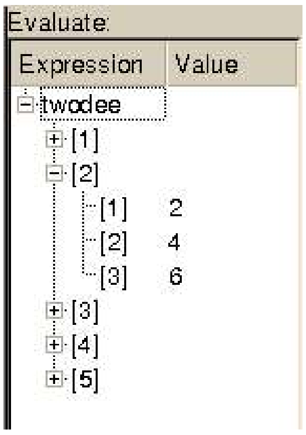
C Example: type of twodee is int[5][3]



```
twodee
  [0]
  [1]
    [0]   0
    [1]   1
    [2]   2
    [3]   3
    [4]   4
  [2]
```

Fig.27 2D array in C

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



```
twodee
  [1]
  [2]
    [1]   2
    [2]   4
    [3]   6
  [3]
  [4]
  [5]
```

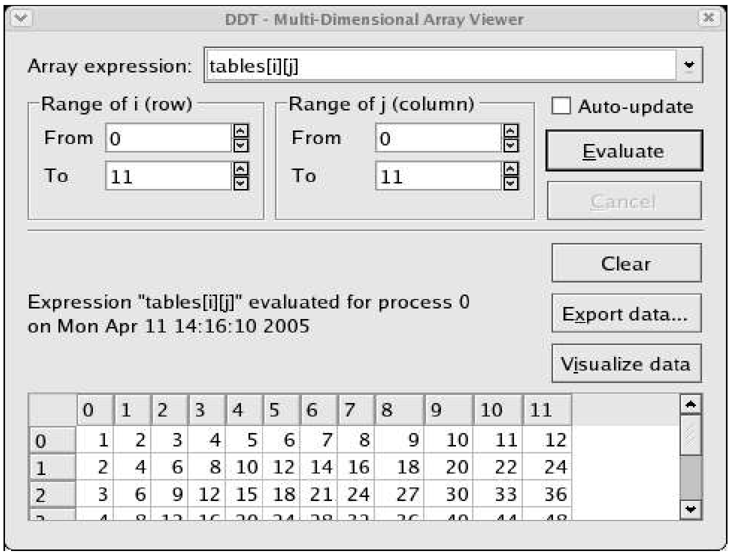
Fig.28 2D array in Fortran

Examining Multi-Dimensional Arrays

Large multi-dimensional arrays are not easy to view in a tree structure so DDT provides a multi-dimensional array viewer. This allows you to type in an expression optionally using up to two variables (i and j) and set the range for these variables.

Clicking `Evaluate` will fill a table with all evaluations of i and j. This data can be exported to a csv (comma-separated variables) file which can be plotted or analyzed in your favorite spreadsheet.

You can view any two-dimensional slice of your data by entering an expression based on i and j, such as A(i+j,j,1) in Fortran or myArray[1][j][i+j] in C/C++. Once the data is loaded into DDT's table you can visualize the data as a surface in 3-D space (see next section).



Array expression: tables[i][j]

Range of i (row): From 0, To 11

Range of j (column): From 0, To 11

☐ Auto-update

Expression "tables[i][j]" evaluated for process 0
on Mon Apr 11 14:16:10 2005

	0	1	2	3	4	5	6	7	8	9	10	11
0	1	2	3	4	5	6	7	8	9	10	11	12
1	2	4	6	8	10	12	14	16	18	20	22	24
2	3	6	9	12	15	18	21	24	27	30	33	36

Fig.29 Multi-dimensional array viewer

Visualizing Data

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

If OpenGL is enabled on your system, 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 in OpenGL 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.

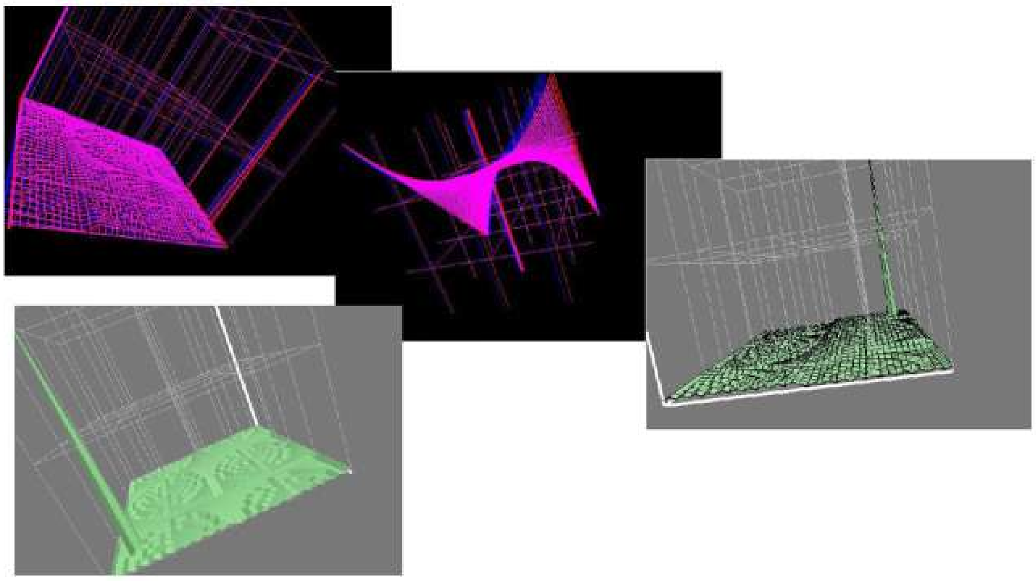


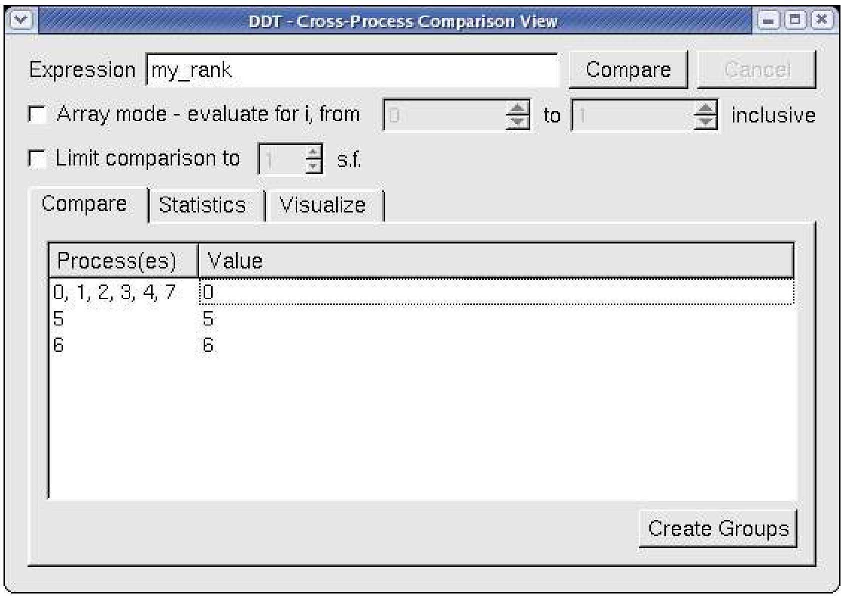
Fig.30 DDT visualization

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

If OpenGL is not enabled, you will still be able to visualize your data, but further manipulation is not possible.

Cross-Process Comparison

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



DDT - Cross-Process Comparison View

Expression: my_rank

☐ Array mode - evaluate for i, from 0 to 1 inclusive

☐ Limit comparison to 1 s.f.

Process(es)	Value
0, 1, 2, 3, 4, 7	0
5	5
6	6

Fig.31 Cross process comparison – compare view

When using the raw comparison – processes are displayed by expression value. The precision of the comparison can also be set by filling the "limit" box. It is also practical to compare complete – or sections of – arrays by parameterising the expression using "i" as the parameter.

It is also possible to automatically create process groups from this panel by clicking the create groups button. This will create 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 (ie. C/C++/Fortran).

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

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

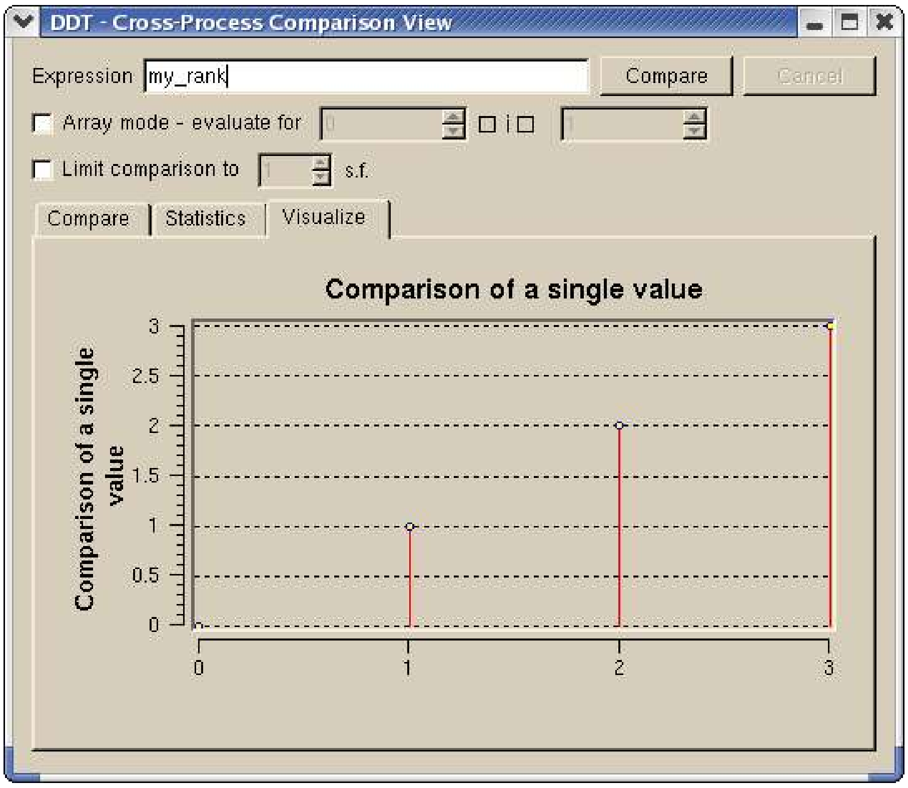
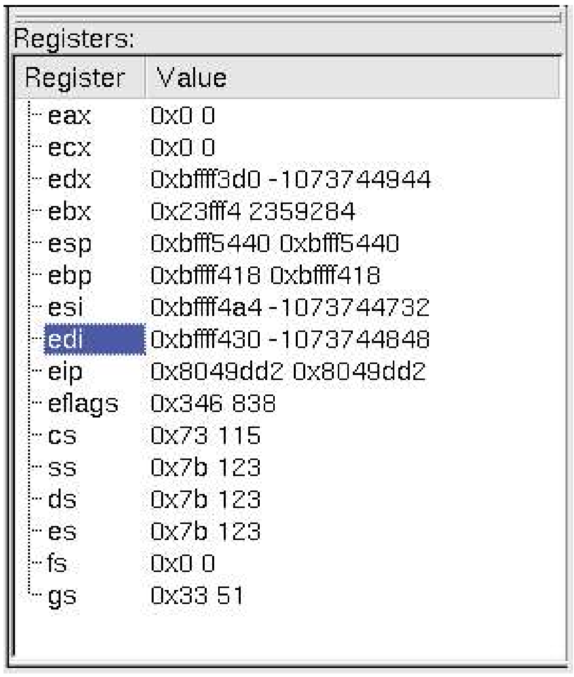

DDT - Cross-Process Comparison View
Expression
my_rank
Compare
Cancel
Array mode - evaluate for
i
Limit comparison to
s.f.
Compare
Statistics
Visualize
Comparison of a single value
Comparison of a single value
0
0.5
1
1.5
2
2.5
3
0
1
2
3

Fig.32 Cross process comparison – visualize view

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

Viewing Registers

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

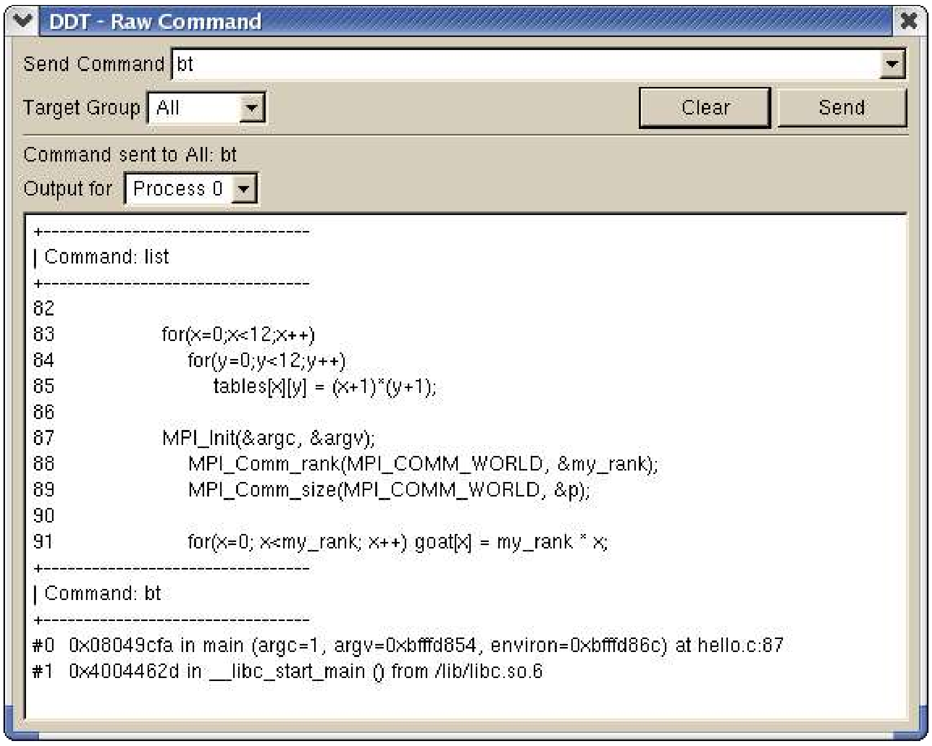


Registers:

Register	Value
eax	0x0 0
ecx	0x0 0
edx	0xbffff3d0 -1073744944
ebx	0x23fff4 2359284
esp	0xbfff5440 0xbfff5440
ebp	0xbffff418 0xbffff418
esi	0xbffff4a4 -1073744732
edi	0xbffff430 -1073744848
eip	0x8049dd2 0x8049dd2
eflags	0x346 838
cs	0x73 115
ss	0x7b 123
ds	0x7b 123
es	0x7b 123
fs	0x0 0
gs	0x33 51

Fig.33 Register view

Interacting Directly With The Debugger



Send Command: bt
Target Group: All

Command sent to All: bt
Output for: Process 0

```
+----------------------------------
| Command: list
+----------------------------------
82
83              for(x=0;x<12;x++)
84                  for(y=0;y<12;y++)
85                      tables[x][y] = (x+1)*(y+1);
86
87              MPI_Init(&argc, &argv);
88                  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
89                  MPI_Comm_size(MPI_COMM_WORLD, &p);
90
91                  for(x=0; x<my_rank; x++) goat[x] = my_rank * x;
+----------------------------------
| Command: bt
+----------------------------------
#0  0x08049cfa in main (argc=1, argv=0xbfffd854, environ=0xbfffd86c) at hello.c:87
#1  0x4004462d in __libc_start_main () from /lib/libc.so.6
```

Fig.34 Debugger interaction window

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

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

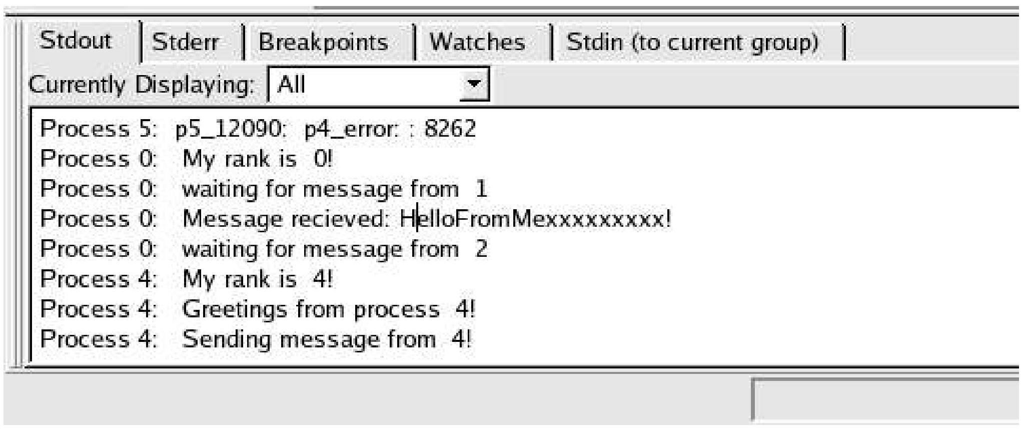
Each command is sent to a group of processes (selected from within the dialog box - not necessarily the current group). To communicate with a single process, create a new group and drag that process into it.

The raw process command window will not work with running processes and requires all processes in the chosen group to be paused.

7. Program Input And Output

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

Viewing Standard Output And Error



```
Process 5:  p5_12090:  p4_error: : 8262
Process 0:   My rank is  0!
Process 0:   waiting for message from  1
Process 0:   Message recieved: HelloFromMexxxxxxxxx!
Process 0:   waiting for message from  2
Process 4:   My rank is  4!
Process 4:   Greetings from process  4!
Process 4:   Sending message from  4!
```

Fig.35 Standard output window

At the bottom of the screen (by default) there are tabs for displaying standard output and standard error.
The contents of these panels can be cut and pasted into the X-clipboard.

Displaying Selected Processes

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

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

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

Saving Output

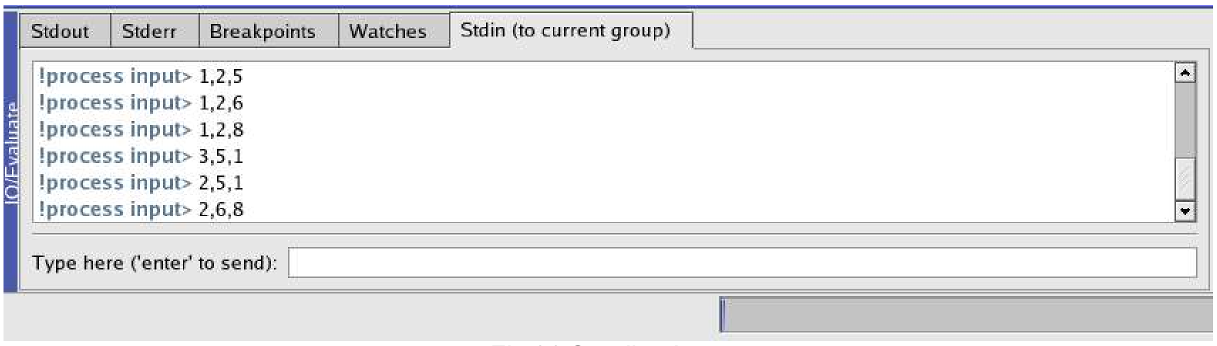
By right-clicking in an output window, it is possible to save the contents of the window to a file.

Sending Standard Input (DDT-MP)

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

Alternatively in DDT you may enter the arguments directly in the `MPI 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 `MPI Run Arguments` box when starting your DDT session in the advanced session configuration dialog.

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



```
!process input> 1,2,5
!process input> 1,2,6
!process input> 1,2,8
!process input> 3,5,1
!process input> 2,5,1
!process input> 2,6,8
```

Fig.36 Sending Input

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

8. Message Queues

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

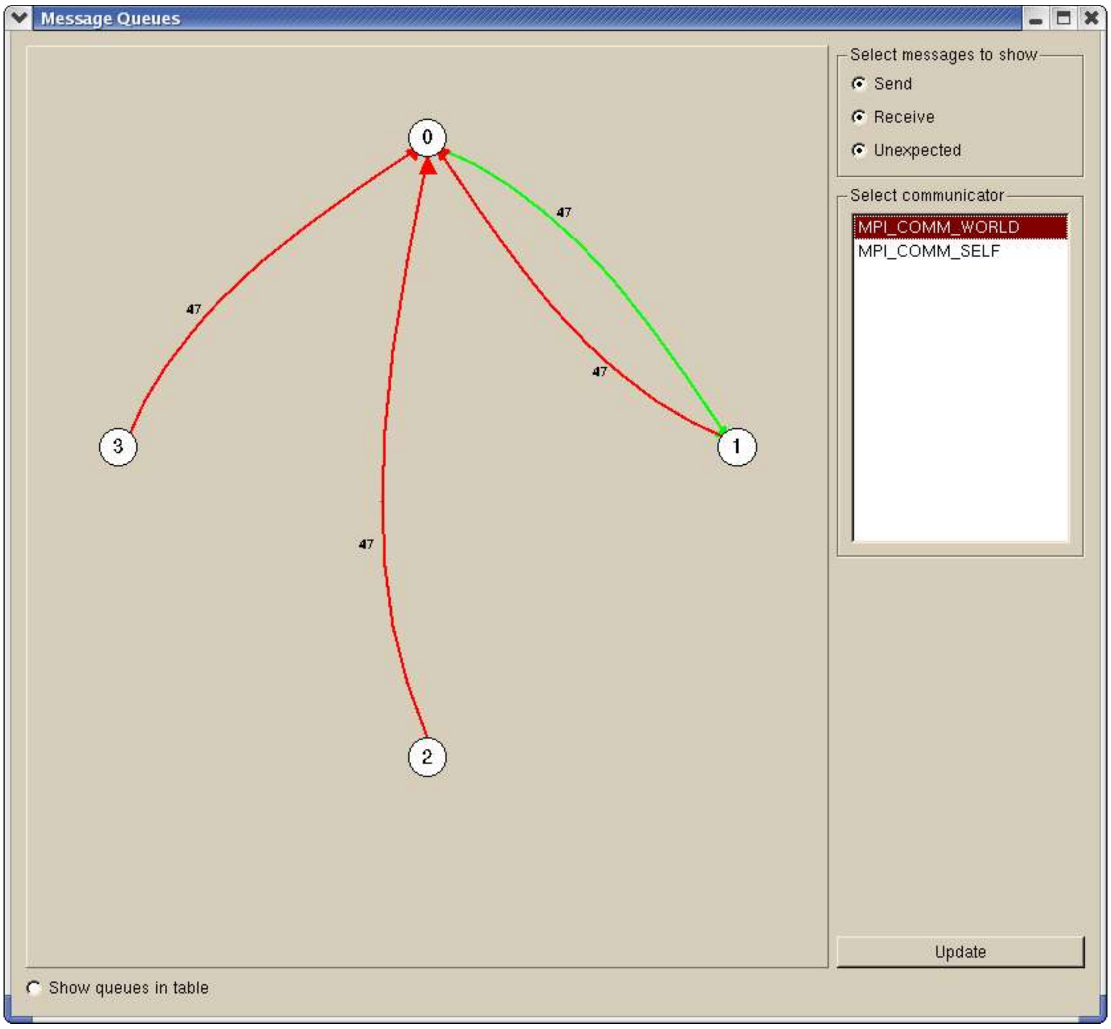


Fig.37 Message queue window

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

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

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

Label	Description
Send Queue	Represents all the outstanding send operations
Receive Queue	Represents all the outstanding receive operations
Unexpected Message Queue	Represents messages that have been sent to this process but have not been received

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

DDT will try to load the default library for the MPI implementation (provided one exists) but for this to happen it must be in the LD_LIBRARY_PATH. If this is not convenient you can set the environment variable, DDT_QUEUE_DLL, which can be the absolute path (or just in the LD_LIBRARY_PATH).

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

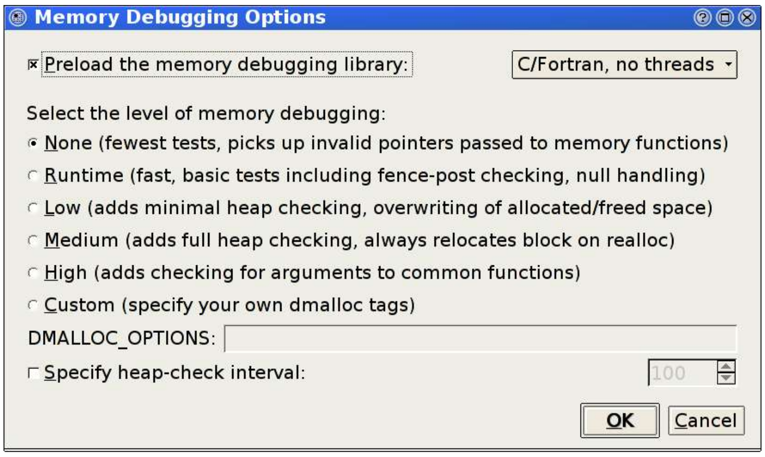
Please note that the quality of underlying implementations of the message queue debugging interface varies considerably – some of the data can therefore be incomplete.

9. Memory Debugging

DDT 1.9 introduces powerful parallel memory debugging capabilities. At the back end, DDT interfaces with a modified version of the dmalloc library[1]. This powerful, cross-platform library intercepts memory allocation and deallocation calls and performs lots of complex heap- and bounds- checking. Fortunately, DDT makes it easy to use, even across tens or hundreds of parallel processes.

Configuration

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


Memory Debugging Options

[✓ Preload the memory debugging library:] C/Fortran, no threads

Select the level of memory debugging:
- [✓ None (fewest tests, picks up invalid pointers passed to memory functions)]
- Runtime (fast, basic tests including fence-post checking, null handling)
- Low (adds minimal heap checking, overwriting of allocated/freed space)
- Medium (adds full heap checking, always relocates block on realloc)
- High (adds checking for arguments to common functions)
- Custom (specify your own dmalloc tags)

DMALLOC_OPTIONS:

Specify heap-check interval: 100

OK Cancel

Fig.38 Memory debugging options

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

1. “Preload the memory debugging library” - when this is checked, DDT will automatically load the memory debugging library. This is great for most users. If you have linked against one of DDT's dmalloc libraries yourself then you can deselect this option. DDT can only preload the memory debugging library when you start a program through DDT **and if it uses shared libraries**. It is not possible to preload for statically-linked programs. You will have to re-link your program with the appropriate dmalloc library in ddt/lib/32 or ddt/lib/64 before running in DDT. If you attach to a running process, this setting has no effect. You can still use memory debugging when you attach, but you will have to manually link your program against one of the libdmalloc*.so versions in one of DDT's lib/32 and lib/64 directories and generate your own DMALLOC_OPTIONS environment variable, as explained in the documentation available at www.dmalloc.com.

2. The combo box, showing “C/Fortran, no threads” in the screenshot – click here and select an option that matches your program, be it C/Fortran, C++, single-threaded, multi-threaded, you should know what this means. Remember to come back here if you start/stop using multi-threading/OpenMP or change between debugging C/Fortran and C++ programs. Many people find they can leave this set to C++/threaded for all their programs, rather than keep on changing the settting.

[1] Copyright 1992-2004 by Gray Watson, see www.dmalloc.com. Licence information is included in lib/dmalloc.txt

Some platforms, most notably AIX, do not support shared library preloading. On these systems the option to preload a memory debugging library will be disabled. To use memory debugging on these systems you must re-link your application with a dmalloc library from ddt/lib/32 or ddt/lib/64 as appropriate. For example:

```
bash$ pwd
/home/mark/ddt
bash$ xlc -q64 -g examples/simple.c -o examples/simple -Llib/64 -ldmalloc
```

would recompile simple.c in 64-bit, single-threaded mode for memory debugging with DDT. Alternatively:

```
bash$ xlc -q32 -g examples/simple.c -o examples-simple -Llib/32 -lpthread -ldmallocth
```

would compile simple.c in 32-bit, multi-threaded mode (notice we link against dmallocth and not dmalloc). The link flags needed for each version are:

	Single-threaded	***Multi-threaded***
C/Fortran	-ldmalloc	-ldmallocth
C++	-ldmallocxx	-ldmallocthcxx

Remember to add -L<ddt's directory>/lib/32 or -L<ddt's directory>/lib/64 first! It's important to link against an appropriate library. If you have trouble linking against the correct library or believe it is missing, just drop a mail to support@allinea.com and we'll be happy to help.

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

1. Even 'None' will catch trivial memory errors such as deallocating memory twice. Selecting this option does NOT turn off memory debugging!

2. The further down the list you go, the more slowly your program will execute. We find that for general use, anything up to “Low” is fast enough to use and will catch almost all errors. If you come across a memory error that's difficult to pin down, choosing a higher setting might expose the problem earlier, but you'll need to be very patient on large, memory intensive codes!

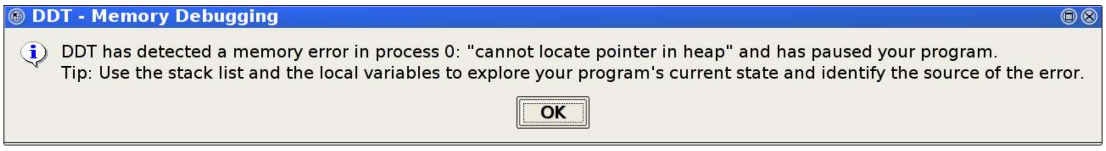
The other settings are really for advanced dmalloc users who know exactly what they want to do. Full documentation for these can be found at http://dmalloc.com/docs/5.4.2/online/dmalloc_toc.html.
Click on 'OK' to save these settings, or 'Cancel' to undo your changes.
NOTE: Choosing the wrong library 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, should you experience problems when memory debugging is enabled.

Feature Overview

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

Stop on Error

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



DDT - Memory Debugging

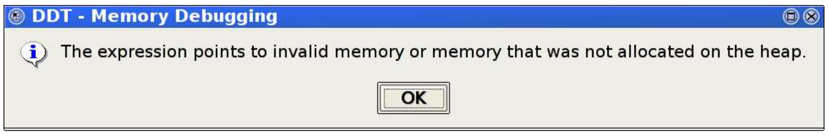
DDT has detected a memory error in process 0: "cannot locate pointer in heap" and has paused your program.
Tip: Use the stack list and the local variables to explore your program's current state and identify the source of the error.

Fig.39 Memory error message

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

Check Validity

Any of the variables or expressions in the 'Evaluate Expression' window can be right-clicked on to bring up a menu. If memory debugging is enabled, one option will be “Check pointer is valid”. Clicking on this will pop up a message telling you whether or not that expression points to memory that was allocated on the heap or not[2]:



DDT - Memory Debugging

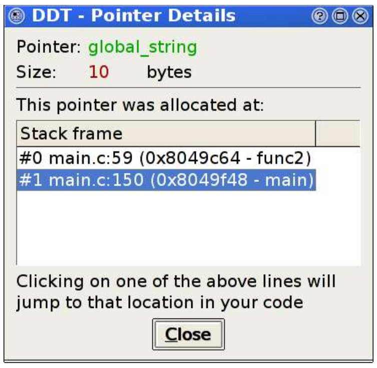
The expression points to invalid memory or memory that was not allocated on the heap.

Fig.40 Invalid memory message

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

View Pointer Details

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



DDT - Pointer Details

Pointer: global_string
Size: 10 bytes

This pointer was allocated at:

Stack frame
#0 main.c:59 (0x8049c64 - func2)
#1 main.c:150 (0x8049f48 - main)

Clicking on one of the above lines will jump to that location in your code

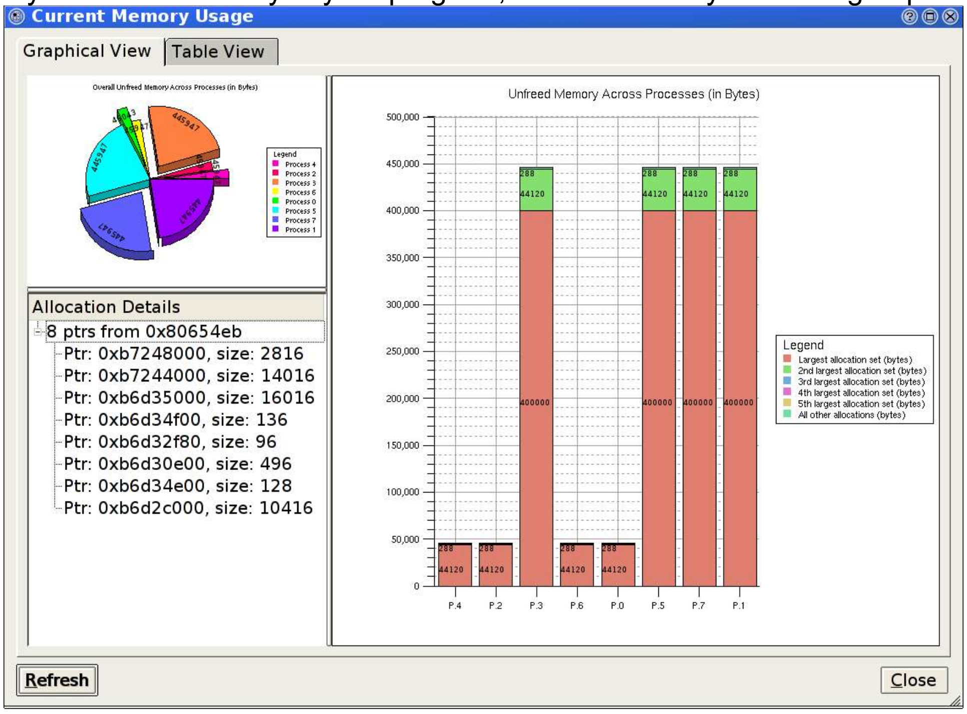
[2] Memory allocated on the heap means something returned by “malloc”, “ALLOCATE”, “new” and so on. A pointer may also point to a local variable, in which case DDT will tell you it does not point to data on the heap. This can be useful, since a common error is taking a pointer to a local variable that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

Fig.41 Pointer details

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

Current Memory Usage

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


Current Memory Usage
Graphical View
Table View
Overall Unfreed Memory Across Processes (in Bytes)
445947
445947
445947
445947
Legend
Process 4
Process 2
Process 3
Process 6
Process 0
Process 5
Process 7
Process 1
Unfreed Memory Across Processes (in Bytes)
500,000
450,000
400,000
350,000
300,000
250,000
200,000
150,000
100,000
50,000
0
288
44120
400000
P.4
P.2
P.3
P.6
P.0
P.5
P.7
P.1
Legend
Largest allocation set (bytes)
2nd largest allocation set (bytes)
3rd largest allocation set (bytes)
4th largest allocation set (bytes)
5th largest allocation set (bytes)
All other allocations (bytes)
Refresh
Close

Allocation Details

```
8 ptrs from 0x80654eb
  Ptr: 0xb7248000, size: 2816
  Ptr: 0xb7244000, size: 14016
  Ptr: 0xb6d35000, size: 16016
  Ptr: 0xb6d34f00, size: 136
  Ptr: 0xb6d32f80, size: 96
  Ptr: 0xb6d30e00, size: 496
  Ptr: 0xb6d34e00, size: 128
  Ptr: 0xb6d2c000, size: 10416
```

Fig.42 Memory usage graphs

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

The stacked bar chart on the right is where the most interesting information starts. Each process is represented by a bar, and each bar broken down into blocks of colour that represent the cumulative size of each allocation. Let's break that down a bit. Say your program contains a loop that allocates a hundred bytes that is never freed. That's not a lot of memory. But if that loop is executed ten million times, you're looking at a gigabyte of memory being leaked! This chart groups memory allocations by the return address. For practical purposes, this means that each block of colour represents the total memory currently allocated from a particular line of code. There are 6 blocks in total. The first 5 represent the 5 lines of code with the most memory allocated, and the 6th (at the top) represents the rest of the allocated memory, wherever it is from.

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

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

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

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

Memory Statistics

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

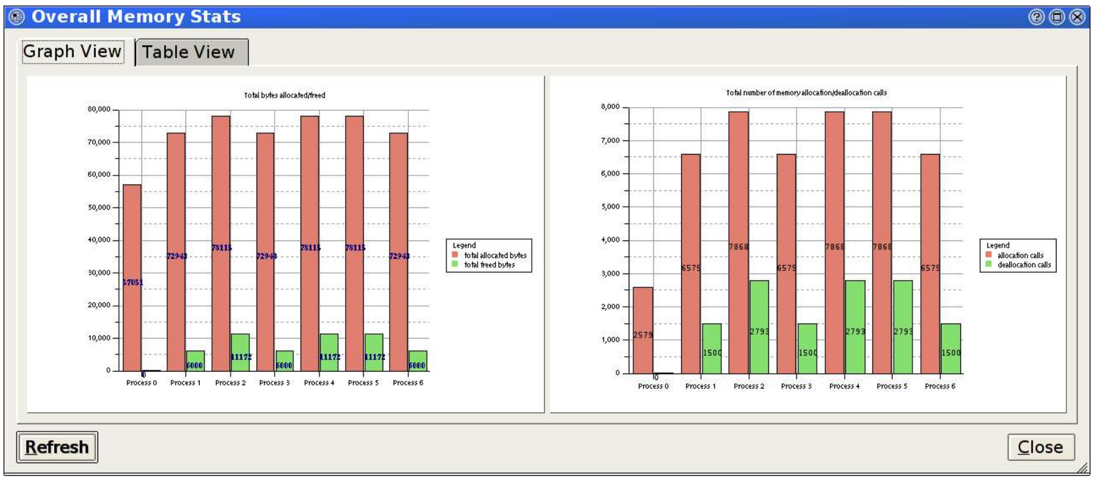


Fig.43 Memory statistics

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

10. Advanced Data Display and C++ STL Support

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

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

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

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

Using The Sample Wizard Library

The sample Wizard Library works with target programs compiled with GNU GCC and using GNU gdb as the underlying debugger.
To use the sample Wizard Library, send a variable to the “Evaluate Expressions” window. Either:

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

Once in the “Evaluate Expressions” window, select the variable and right-click the mouse. From the pop-up menu that appears, select the “Evaluate with Wizard” option.
This should lead to a re-display of the value of the variable, this time evaluated by the Wizard Library rather than by DDT itself.
To re-display the value of the variable as it was before the Wizard Library was used, right-click the variable again and select “Evaluate without Wizard” from the pop-up menu.
As an example, suppose a C++ program contained the following code:

```
string s = “hello world”;
```

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

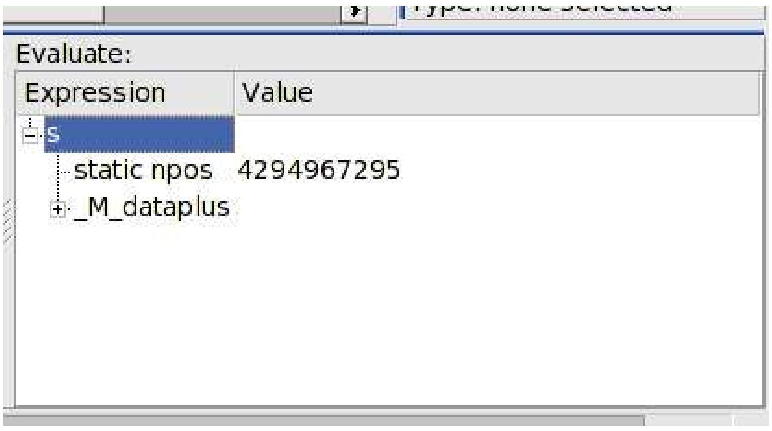


Fig.44 Evaluating without wizard

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

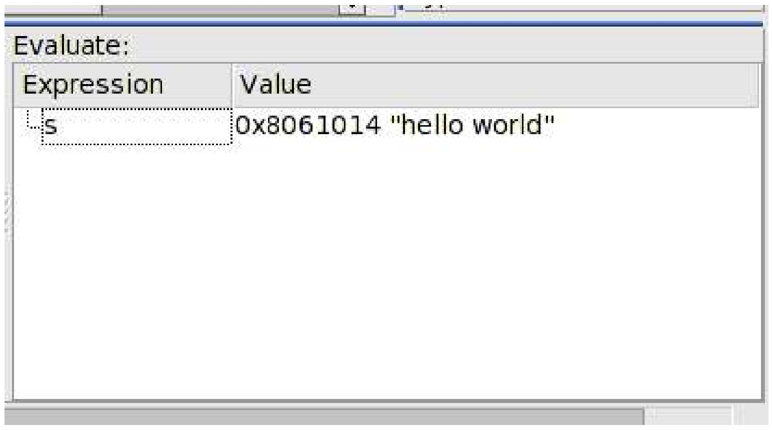


Fig.45 Evaluating with wizard

Writing A Custom Wizard Library

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

Theory Of Operation

A Wizard Library must support the following interface:

```
typedef VirtualProcess*(*CurrentFun)();

extern "C"
{
    void* initialize(CurrentFun);

    Expression evaluate(const std::string&);
}
```

When DDT has to evaluate an expression using the a Wizard Library, each evaluation causes a call to the Wizard Library's `initialize()` function followed by a call to the `evaluate()` function.
The `initialize()` function takes one argument, which is a pointer to a DDT function to call whenever the Wizard Library requires access to a handle for the current `VirtualProcess`.
The `VirtualProcess` handle is required by the Wizard Library's `evaluate()` function to allow it to make calls back into DDT in order to service the evaluation request.
The `evaluate()` function takes one argument, which is a constant-reference to a string containing the expression to be evaluated. The function must perform some calculations and record the results inside an `Expression` before returning that back to DDT.
DDT will examine the returned `Expression` and display the contents as the value of the expression in the “Evaluate Expressions” window.
So, an example of a minimal (and fairly useless) Wizard Library would look like:

```
#include <Interface.h>

#include <iostream>
#include <string>

using namespace std;

typedef VirtualProcess*(*CurrentFun)();

static CurrentFun current = 0;

extern "C"
{
    void* initialize(CurrentFun _current);

    Expression evaluate(const string& e);
}

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

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

    Expression r(e);

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

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

    return r;
}
```

Compiling A New Wizard Library

Two steps are required to build a new Wizard Library:

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

`Expression.o` contains the required code to handle `Expression`s and is provided with the DDT distribution.
The following example builds a Wizard Library using GNU g++:

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

Using A New Wizard Library

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

```
$ DDT_WIZARD_DLL=$DDTPATH/wizard/libwizardexample1.so.1.0.0
$ export DDT_WIZARD_DLL
$ ddt your_test_program
```

The VirtualProcess Interface

DDT provides the Wizard Library with a `VirtualProcess` interface for *each* target process being debugged. Although there may be many of these `VirtualProcess` instances, only the one corresponding to the *current* target process is available when the Wizard Library is invoked by DDT.
As seen in the example above, the current `VirtualProcess` is retrieved using the “get current” function-pointer passed into the Wizard Library during DDT's call of the `initialize()` function:

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

The `VirtualProcess` interface contains the following methods:

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

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

`readType()` sends an expression back into DDT and returns a textual representation (according to the underlying debugger, that is) of the *type* associated with that expression. This can be used to determine how to process the original `evaluate()` request from DDT.
`readExpression()` send an expression back into DDT and returns an `Expression` representing the results of evaluating that expression.
The following code modifies a section of our first example, by reading the type of the expression sent in by DDT and then sending the expression back into DDT for evaluation. This is somewhat of a waste of time, in that it is acting as a “Loopback Wizard” and adding no extra value!

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

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

    Expression r(e);

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

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

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

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

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

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

    return r;
}
```

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

```
    string s = "hello world";

    int i = 42;
```

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

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

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

The Expression Class

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

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

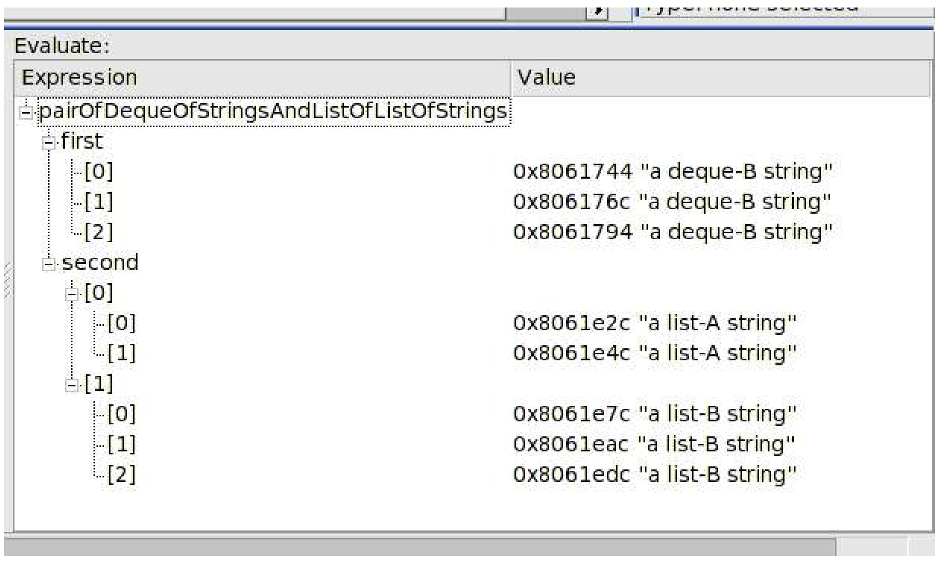
Where the "name" of the expression differs from the string sent to DDT for evaluation, another string field in `Expression` is used:

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

An example of this will be coming up soon when we finally implement evaluation of strings.
Where more complicated expressions are involved, the `value` field is not used and is left blank. Instead, a list of "child" sub-`Expression`s are slung under this `Expression` using a fourth field:

- `children` (further sub-expressions)

The type of this fourth field is `std::vector<Expression>`.
An example where the `children` field would be used is during the evaluation of a variable of type `vector`. The `displayName` would be the name of the variable. The `value` would be empty. There would be a number of `children`, one for each entry in the `vector`. Each of these `children` would have their `displayName` set to a pretty-printed string representing their index (e.g. "`[4]`"). Whether each of the `children` would have a `value` or whether they would contain further `children` would depend upon how complex the data structure being evaluated was.
For instance, the following screen snapshot shows the evaluation of a variable of type `pair<deque<string>, list<list<string> > >`:



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

Fig.46 Evaluation of strings

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

Final String-Enabled Wizard Library Example

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

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

    return strm;
}
```

Examination of the contents of a string variable (in a test program compiled with GNU g++) shows that it has a `_M_dataplus` data member which is further composed of a `_M_p char*` pointer.
With this in mind, modify the `evaluate()` function to behave specially for variables of type string:

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

    Expression r(e);

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

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

            if (eType == "string")
            {
                Expression
                    eResult(v->readExpression(e +
                                                   "._M_dataplus._M_p
"));

                  // Want the pretty-printed name to be the name
of
                // the string itself.
                eResult.setDisplayName(e);

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

                r = eResult;
            }

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

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

    return r;
}
```

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

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

Notice that the `Expression` for the string evaluation has different values for the `displayName` and `realName` fields.
Single-stepping the test program yields the following output:

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

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

11. The Licence Server

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

Running The Server

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

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

Running DDT Clients

DDT will, as is also the case for fixed licences, use a licence file either specified via environment variables (DDT_LICENCE_FILE or DDT_LICENSE_FILE) or from the default location of $DDTPATH/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 frontend-backend communication in DDT, Defaults to 4242,4243,4244,4244,4245
serial_number	Yes	The serial number of the server licence to be used
serverport	Yes	The port the server listens on
type	Yes	Must have value 2 – this identifies the licence as needing a server to run properly

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

Logging

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

Level 0: no logging.
Level 1: client licences issued are shown, served licences are listed.
Level 2: stale licences are shown when removed, licences still being served are listed if there is no spare licence.
Level 3: full request strings received are displayed
Level 6 is the maximum.

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

Troubleshooting

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

Adding A New Licence

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

Examples

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

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

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

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

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

```
type=3
serial_number=1014
max_processes=48
expires=2004-04-01 00:00:00
support_expires=2004-04-01 00:00:00
mac=00:E0:81:03:6C:DB
interface=eth0
debuggers=gdb
serverport=4252
max_users=2
beat=60
retry_limit=4
hash=P5I:?L,FS=[CCTB<IW4
```

```
hash2=c18101680ae9f8863266d4aa7544de58562ea858
```

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

```
type=2

serial_number=1014

hostname=server.physics.acme.edu

serverport=4252
```

Example Of Access Via A Firewall

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

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

And a local licence file should be created:

```
type=2

serial_number=1014

hostname=localhost

serverport=4252
```

Querying Current Licence Server Status

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

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

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

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

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

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

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

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

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

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

Finally, after the first DDT finishes:

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

Licence Server Handling Of Lost DDT Clients

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

```
lost_client_timeout_period = (beat seconds) * (retry_limit + 1)
```

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

```
60 * (4 + 1) = 300 seconds
```

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

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

A. Supported Platforms

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

Platform	Operating Systems	MPI	Compilers
Intel/AMD x86 AMD Opteron (32+64) Intel Itanium 2	Redhat 7 and above, SuSE, Debian, and similar	SGI Altix, Bproc, LAM-MPI, MPICH, Myricom MPICH-GM, Quadrics MPI, Scali MPI Connect, SCore, Scyld	GNU, Absoft, Intel and Portland
HP Itanium and PA-RISC	HP-UX 11.22 and 11.11	HP-MPI	Native
IBM Power	AIX 5.1 and above	IBM PE	Native, GNU
SGI MIPS	IRIX	SGI MP Toolkit	Native
Sun Ultrasparc	Solaris 8 and above	Sun Clustertools 4 and above	Native

B. Troubleshooting DDT

If you should encounter any difficulties in using DDT, you should, in the first instance, view the FAQ (Appendix C).

You may find a problem with your DDT that has already been fixed by Allinea, please check the support pages of the Allinea website for updates. If your problem persists, contact Allinea directly by emailing support@allinea.com.

Problems Starting DDT Frontend

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

- DDT is not in the PATH – the shell reports that DDT is not found. Change your path to include the $DDTPATH/bin directory
- The licence is invalid – in this case DDT will issue an error message. You should verify that you have a licence file, that it is stored as $DDTPATH/Licence, and check manually that it is valid by inspection. If DDT still refuses to start, please contact Allinea

Problems Starting Scalar Programs

There are a number of possible sources for problems. The most common is – for users with a multi-process licence – that the MPI implementation has not been set to “none”.

Please note that in some cases DDT reports a problem with MPI when encountering a problem, instead of suggesting that your program cannot start. If this happens to you, and you are sure that you have selected 'None' as the MPI implemenatation, we would like to hear from you!

Other potential problems are

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

Problems Starting Multi-Process Programs

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

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

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

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

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

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

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

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

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

C. FAQs

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

Ensure that the hostname(s) given are reachable using ping, if not try using the IP addresses. If DDT fails to find the executables, ensure the executable is available on every machine, and - for the common MPICH distribution - that you can "rsh" to each host without the password prompt.

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

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

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

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

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

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

This is likely to be caused by dual processor configuration for your MPI distribution. Make sure you have selected 'smp-mpich' or 'scyld' as your MPI implementation in DDT's configuration window. If this doesn't help, see Appendix E for a workaround and contact us for further assistance.

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

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

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

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

Why can't I see any output on stderr?

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

DDT complains about being unable to execute malloc

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

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

Some features seem to be missing (e.g. watch points) - what's wrong?

This is because not all debuggers support every feature that DDT does and so they are disabled by removing the window/tab by from DDT's interface. For example if you are using Intel's IDB debugger then the watches tab has been removed as this debugger doesn't support watches.

My code does not appear when I start DDT

This could be due to the default selected font not being installed on your system. Go into the Session – Configuration window and choose a font such as Times or Century Schoolbook and you should now be able to see the code.

When I use step out my program hangs

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

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

This is due to the fact that you have started your MPI process outside of DDT. The message queue process cannot then find which DLL to use for the version of MPI that you have started. The way to fix this is to set the variable DDT_QUEUE_DLL explictly before you start DDT.

Example: “export DDT_QUEUE_DLL=/usr/local/mpich/lib/libtvmpich.so”

The files needed for LAM and MPICH are listed here:

- Lam – liblam_totalview.so
- MPICH – libtvmpich.so

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

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

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

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

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

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

This is a known problem on the Opteron system with 64/32bit compatibility. If the DDT binary is 32-bit, but your target application is 64-bit, then the MPI message queue support library needs to be 32-bit also – and most MPI implementations do not support this configuration. An alternative is to copy in a library built on a compatible system such as Redhat 9 and setting the DDT_QUEUE_DLL argument to point to this library.

Alternatively, contact Allinea to obtain a 64-bit binary of DDT.

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

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

Stepping processes and changing between them seems very slow – what could be causing this?

If you have the local variables display showing it can sometimes take several seconds to retrieve all the data necessary – particularly for codes that have many local variables and arrays. Switching to current line view should speed things up considerably in this case.

Alternatively your program may have a very long stack trace, or a large number of arguments in to one or more functions within it. Click on the 'Stack' box in DDT to see a full list at any one time. Retrieving these can take a while if there are, say, 10 stack frames with 20 arrays of 5000 elements in each argument list! You can tell DDT not to read the arguments in the stack trace by setting the environment variable DDT_NO_STACK_ARGS to 1 before starting DDT.

We are continually looking for opportunities to optimise DDT's performance on very large codes. If these problems affect you, we'd like to work with you to improve the performance and scalability even further – drop us a mail at support@allinea.com and we'll be in touch!

My Itanium version of DDT seems to 'lose' one or more of my multi-threaded processes when I have memory debugging set to 'Runtime' or higher

It seems that dmalloc's fencepost-checking interacts very badly with the pthreads implementation on this platform. As a workaround you can select the 'Custom' option in the memory debugging settings, and remove “check-fence” from the string there. Other memory debugging functionality should be unaffected. If this problem occurs on your system, please contact support@allinea.com as we may have a patch available.

I'm using the “-stdin” mpirun argument and my program seems to get stuck after reading all the input from the file

We recommend using DDT's “input file” mechanism instead of MPI's – this instructs your program to read from the file directly instead of from a stream over the network.

If you really must use the MPI mechanism then you should be aware that no 'end of file' notification is sent to your program. The other alternative to the “input file” mechanism is to start your program from the command line as usual and then use DDT to attach to it once it is running.

Obtaining Support

If you are unable to resolve your problem, the most effective way to get support is to email Allinea with a detailed report of your problem. If possible, you should obtain a log file for the problem and email this to Allinea – support@allinea.com.

Generating a log file is simple. Simply run DDT as follows:

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

Once DDT has loaded simply reproduce the problem with as few processes as possible, then quit or kill DDT. These log files can be quite large, so to prevent email servers stripping the attachment we recommend using gzip or bzip2 to compress the file before sending it.

D. Notes On MPI Distributions

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

Bproc

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

HP MPI

Select HP MPI as the MPI implementation.

A number of HP MPI users have reported a preference to using “mpirun -f jobconfigfile” instead of “mpirun -np 10 a.out” for their particular system. It is possible to configure DDT to support this configuration – using the support for batch (queuing) systems (see page 16). 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”.

LAM/MPI

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

MPICH And SMP Nodes

This issue affects some distributions where shared memory is used to communicate between processors on a dual processor machine. For mpich-p4 this will only affect you if your configuration of mpich specified -comm=shared.

Under these circumstances the dual CPUs use a different starting mechanism for mpirun. We recommend selecting the `smp-mpich` or `scyld` implementation from DDT's configuration window as appropriate. If this does not solve your problem, or you are using an unsupported MPI implementation then you can try setting MPI_MAX_CLUSTER_SIZE=1. This will still allow you to use a large cluster, but it will fool MPI into using rsh/ssh instead of fork to start jobs. It will still use all available cpus, for MPICH you could do this by specifying a dual processor machine TWICE in the machines.LINUX file instead of specifying hostname:2.

MPICH p4

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

MPICH p4 mpd

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

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

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

and your shell will fill in these values for you.

Choose MPICH as the MPI implementation.

MPICH-GM

Some users have reported problems starting this using the MPICH-GM option in DDT. You may find that one of the other MPICH variants works better with your configuration. If you find one of the other variants works for you, or none do, please contact us at support@allinea.com.

IBM PE

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

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

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

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

Select IBM PE as the MPI implementation.

NEC MPI

Select Generic as the MPI implementation.

Quadrics MPI

Select Generic as the MPI implementation.

SCore

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

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

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

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

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

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

```
#!/bin/sh

. ./bashrc

$DDTPATH/bin/ddt-debugger.bin $*
```

Choose SCore as the MPI implementation.

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

Scyld

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

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

Choose Scyld as the MPI implementation.

SGI Altix/Irix

Early versions SGI's MP Toolkit can cause GDB to crash due to a library problem. This is easily resolved if it occurs. Compile your application with the extra linking flag "-lrt" and try DDT again. SGI MP Toolkit should be chosen from the MPI implementations list.

E. Compiler Notes

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

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

Absoft

No known issues.

DDT can debug Absoft code using either GDB or FX2. Tested platforms are x86, AMD64.

GNU

The compiler flag “-fomit-frame-pointer” should never be used in an application which you intend to debug. Doing so will mean DDT cannot properly discover your stack frames.

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.

For F90, allocatable arrays cannot be examined in any way, in either 32- or in 64-bit mode. 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.

Intel Compilers

DDT can debug Intel compiled code with either GDB or IDB. The recommended default debugger is “automatic”.

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

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

In some rare circumstances, IDB may be successful where the automatic debugger fails. If you do choose IDB, note that versions with a build date of 2003 are known to work.

On some systems IDB takes an unusually long time to return a list of source files to DDT, or consumes a large amount of memory in doing so. It is possible to work around the problem by setting the environment variable DDT_IDB_MANUAL_FILES to 1 – this will prevent IDB from building file lists automatically but when your job starts there will be no files in the file tree. You can add these manually by right-clicking on the file tree and choosing to add source directories. You may need to refresh DDT by double-clicking on the current process before the source file appears with correctly highlighted lines of code.

Some Intel-compiled MPI implementations can corrupt the stack, which forces DDT to start the job in a 'last resort' mode. If you find your first process has stopped at MPI_Init, but the others either finish running or are sitting in MPI calls (check the stack listing for each) then this is happening to you. We'd like to hear from you if this is the case (email support@allinea.com). As a workaround, you can put an MPI barrier immediately after MPI_Init. When DDT starts up, right-click on a line after that barrier and choose "Run to here". This will bring all of your processes to the same line at the start of your MPI code, and you can now debug as usual.

Pathscale EKO compilers

The recommended debugger for this compiler is "automatic"; DDT has been tested with version 1.2, 1.3 and 1.4 of this compiler suite.

Known issues with EKO 1.2: Allocated arrays incorrectly generate debug information pointing to the address of the array and not the contents.

Notes for version 1.4: Fortran pointers are reported as their values, not addresses, so do not need to be dereferenced. This may also occur with version 2.

Portland Group Compilers

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

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

Known issues with Portland 5.2: Assumed shape arrays can have wrong dimensions (Portland reference TPR 301).

When using PGDBG as the backend debugger it is important to ensure that the -Munroll and -fast options are **not** used to compile your MPI distribution (this is a known problem with PGDBG and the 1.2.x series of MPICH) as PGDBG is unable to execute a step-out which is necessary during MPI startup when this optimization is turned on.

Solaris DBX

Some versions of DBX may prevent redirection of stdout/stderr, which means that DDT may not display the program output. If your version of DBX is affected then Allinea recommend writing output to a file instead.

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

DDT has been tested with Solaris 10 for Opteron, using Sun's CC v5.7 and F90 v8.1. Some early versions of DBX which accompany these compilers exhibit some fatal problems. If DDT fails to start your debugging session, please contact Allinea for advice.

HP-UX And wdb

DDT has been tested to work with the gdb that is supplied with wdb v4.0. This is available from http://www.hp.com/go/wdb. Ensure that this gdb is in your path and select gdb as the interface.

F. Architectures

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

SGI Altix 3000

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

IBM AIX Systems

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

Index