Arm Forge User Guide

Version 19.0.5

Contents

Part I

Arm Forge

Introduction to Arm Forge

Arm Forge combines Arm DDT, the leading parallel debugger for time-saving high-performance application debugging, and Arm MAP, the trusted performance profiler for invaluable optimization advice.

Arm Forge supports many parallel architectures and models, including MPI, UPC, CUDA and OpenMP. Arm Forge is a cross-platform tool, with support for the latest compilers and C++ 11 standards, and Intel, 64-bit Arm, AMD, OpenPOWER and Nvidia GPU hardware.

Arm Forge provides you with everything you need to debug, fix and profile programs at any scale. One common interface makes it easy to move between Arm DDT and Arm MAP during code development.

Arm Forge provides native remote clients for Windows, Mac OS X and Linux. Use a remote client to connect to your cluster, where you can run, debug, profile, edit and compile your application files.

Arm DDT

Arm DDT is a powerful graphical debugger suitable for many different development environments, including:

- Single process and multithreaded software.
- OpenMP.
- Parallel (MPI) software.
- Heterogeneous software, for example, GPU software.
- Hybrid codes mixing paradigms, for example, MPI with OpenMP, or MPI with CUDA.
- Multi-process software including client-server applications.

Arm DDT helps you to find and fix problems on a single thread or across hundreds of thousands of threads. It includes static analysis to highlight potential code problems, integrated memory debugging to identify reads and writes that are outside of array bounds, and integration with MPI message queues.

Arm DDT supports:

- C, C++, and all derivatives of Fortran, including Fortran 90.
- Limited support for Python (CPython 2.7).
- Parallel languages/models including MPI, UPC, and Fortran 2008 Co-arrays.
- GPU languages such as HMPP, OpenMP Accelerators, CUDA and CUDA Fortran.

Related information

- Chapter 5 provides details about getting started with Arm DDT.

Arm MAP

Arm MAP is a parallel profiler that shows you which lines of code took the most time to run, and why. Arm MAP does not require any complicated configuration, and you do not need to have experience with profiling tools to use it.

Arm MAP supports:

- MPI, OpenMP and single-threaded programs.
- Small data files. All data is aggregated on the cluster and only a few megabytes written to disk, regardless of the size or duration of the run.
- Sophisticated source code view, enabling you to analyze performance across individual functions.
- Both interactive and batch modes for gathering profile data.
- A rich set of metrics, that show memory usage, floating-point calculations and MPI usage across processes, including:
  - Percentage of vectorized instructions, including AVX extensions, used in each part of the code.
  - Time spent in memory operations, and how it varies over time and processes, to verify if there are any cache bottlenecks.
  - A visual overview across aggregated processes and cores that highlights any regions of imbalance in the code.

Related information

- Chapter 16 provides details about getting started with Arm MAP.

Online resources

You can find tutorials, webinars and white papers on the Arm developer website.

- Help and tutorials
- Known issues

If you have questions or require further support, please get in touch with our dedicated support team.

Get Arm Forge at Arm Forge downloads.

Installation

A release of Arm Forge, containing Arm DDT and Arm MAP may be downloaded from the Arm Developer website.

Both a graphical and text-based installer are provided. See the following sections for further details.

Linux installation

Graphical install

Untar the package and run the `installer` executable using these commands:

```
tar xf arm-forge-19.0.5-<distro>-<arch>.tar
cd arm-forge-19.0.5-<distro>-<arch>
./installer
```

replacing `<distro>` and `<arch>` with the OS distribution and architecture of your tar package, respectively. For example, the tarball package for Redhat 7.4 OS and Armv8-A (AArch64) architecture is: arm-forge-19.0.5-Redhat-7.4-aarch64.tar

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

The *Install Type* page lets you choose which user(s) to install Arm Forge for.

If you are an administrator (`root`) you may install Arm Forge for *All Users* in a common directory such as `/opt` or `/usr/local`, otherwise only the *Just For Me* option is enabled.

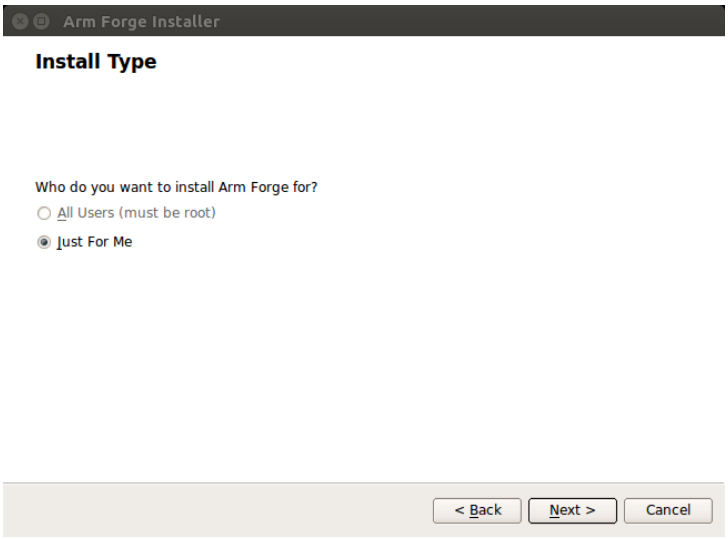

Figure 1: *Installer—Installation type*

Once you have selected the installation type, you are prompted to specify the directory you would like to install Arm Forge in. For a cluster installation, choose a directory that is shared between the cluster login or frontend node and the compute nodes. Alternatively, install it on or copy it to the same location on each node.

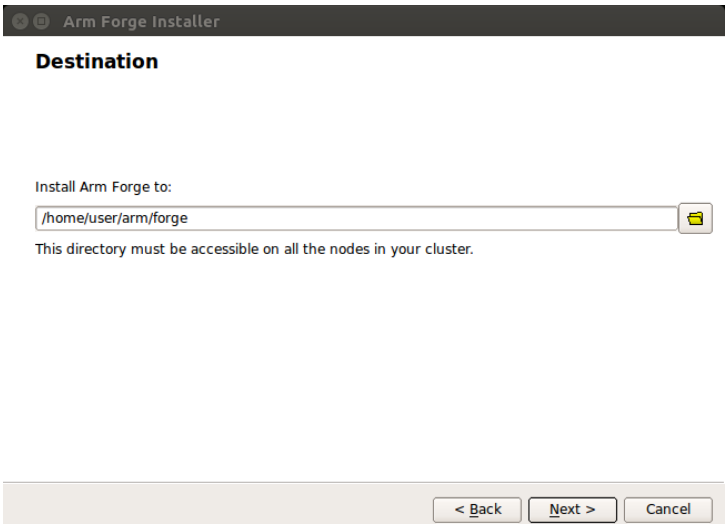

Figure 2: *Installer—Installation directory*

You are shown the progress of the installation on the *Install* page.

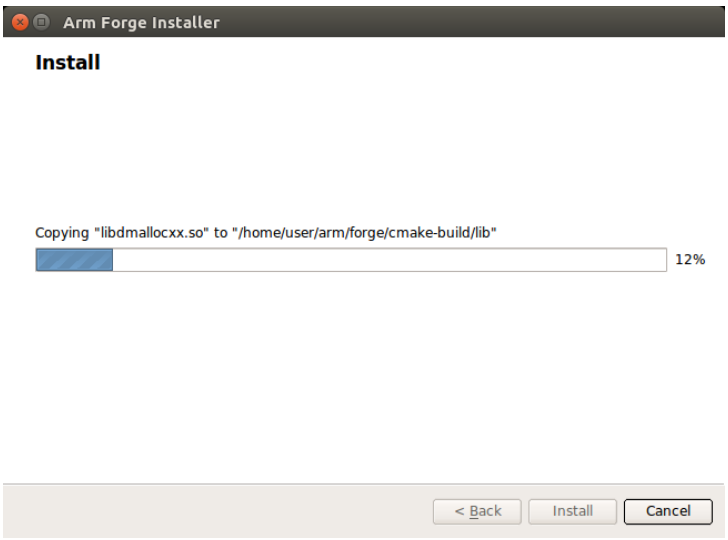

Figure 3: *Install in progress*

Icons for DDT and MAP will be added to your desktop environment's *Development* menu.

It is important to follow the instructions in the `README` file that is contained in the tar file. In particular, you need a valid license file. Use the following link to obtain an evaluation license: Get software.

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

Text-mode install

The text-mode install script `textinstall.sh` is useful if you are installing remotely. This can be used as follows:


```
tar xf arm-forge-19.0.5-<distro>-<arch>.tar
cd arm-forge-19.0.5-<distro>-<arch>
./textinstall.sh
```

replacing `<distro>` and `<arch>` with the OS distribution and architecture of your tar package, respectively. For example, the tarball package for Redhat 7.4 OS and Armv8-A (AArch64) architecture is: arm-forge-19.0.5-Redhat-7.4-aarch64.tar

Press *Return* to read the license when prompted and then input the directory where you would like to install Arm Forge. The directory must be accessible on all the nodes in your cluster.

Alternatively, to run the text-mode install script `textinstall.sh`, accept the license, and point to an installation directory in one step, pass the arguments `--accept-licence` and `<installation_directory>` when executing `textinstall.sh`. For example:

```
./textinstall.sh --accept-licence <installation_directory>
```

replacing the `<installation_directory>` with a directory of your choice.

Mac installation

The Arm Forge client for Mac OS X is supplied as an Apple Disk Image (*.dmg) file. This includes the documentation folder, which contains a copy of this user guide and the release notes. It also contains the Arm Forge client application bundle icon, which should be drag and dropped into the `Applications` directory.

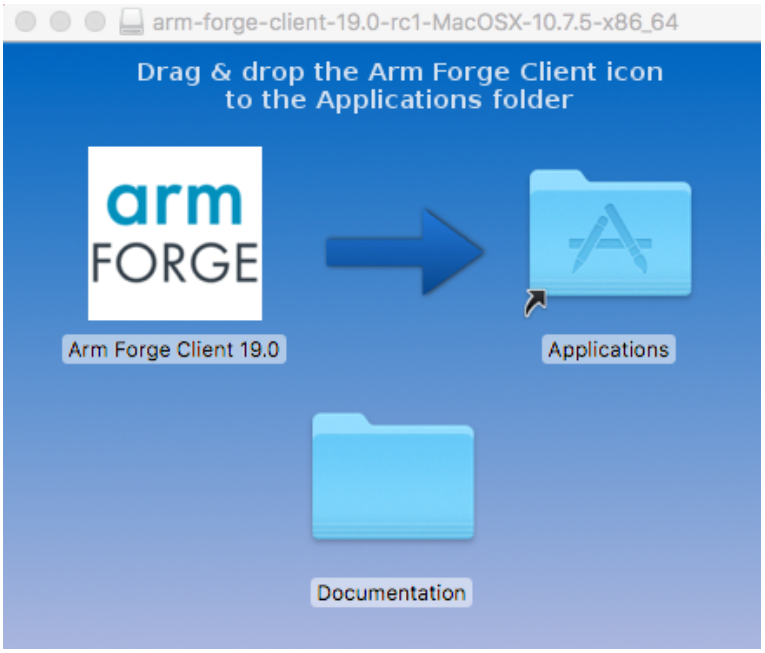

Figure 4: *Mac Installer—Installation Folder*

Windows installation

The Arm Forge client for Windows is installed using a graphical installer. This is a familiar Windows set-up executable, although care needs to be taken with the choice of a destination folder for the installation.

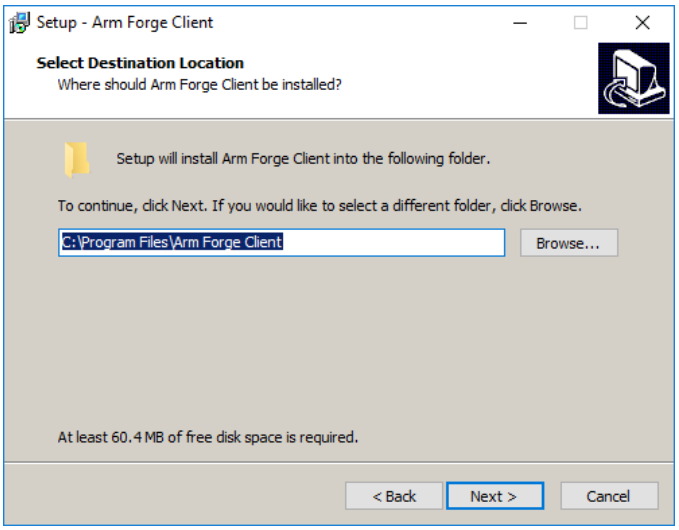

Figure 5: *Windows Installer—Installation Folder*

If the user performing the installation has administrative rights, then the default installation folder is `C:\Program Files\Arm Forge`. If administrative rights have not been granted, then the default is `C:\Users\<user>\AppData\Local`.

License files

Arm Forge products require a license file for their operation. If you are using the Remote Client you do not need a license file on the machine running the Remote Client, but on the machine you are connecting to instead.

If you do not have a license file, the GUI will show this in the lower-left corner and you will not be able to run, debug or profile new programs.

Time-limited evaluation licenses are available from the Arm website: Get software.

Workstation and evaluation licenses

Workstation and Evaluation license files for Arm Forge do not require Arm Licence Server and should be copied directly to *`{installation-directory}`*`/licences`. For example, `/home/user/arm/forge/licences/Licence.ddt`. Do not edit the files as this will prevent them from working.

If you have separate license files for Arm DDT and Arm MAP you do not need separate installations of Arm Forge. You may instead copy the individual license files to *`{installation-directory}`*`/licences`. When Arm Forge is started you may choose between Arm DDT and Arm MAP on the Welcome page. If you have multiple licenses for the same product the license with the most tokens is preferred.

You may specify an alternative location of the license directory using the environment variable `ALLINEA_LICENCE_DIR`. For example:

```
export ALLINEA_LICENCE_DIR=${HOME}/SomeOtherLicenceDir
```

`ALLINEA_LICENSE_DIR` is an alias for `ALLINEA_LICENCE_DIR`.

Supercomputing and other floating licenses

For users with Supercomputing and other floating licenses, the Arm Licence Server must be running on the designated license server machine prior to running Arm Forge.

The Arm Licence Server and instructions for its installation and usage may be downloaded from Arm Developer website.

The license server download is on the Arm Forge download page.

A floating license consists of two files: the server license (a file named `Licence.xxxx`) and a client license file named `Licence`. The client file should be copied to *`{installation-directory}`*`/licences`. For example, `/home/user/arm/forge/licences/Licence`. You will need to edit the `hostname` line to contain the host name or IP address of the machine running the Licence Server. See the Licence Server user guide for instructions on how to install the server license.

Architecture licensing

Licenses issued after the release of Arm Forge 6.1 specify the compute node architectures that they may be used with. Licenses issued prior to this release will enable the x86_64 architecture by default. Existing users for other architectures will be supplied with new licenses that will enable their architectures.

If there is any problem then contact Arm support at Arm support.

Using multiple architecture licenses

If you are using multiple license files to specify multiple architectures, it is recommended that you leave the default licenses directory empty. Instead, create a directory for each architecture, and when you target a specific architecture set `ALLINEA_LICENSE_DIR` to the relevant directory. Alternatively, you can set `ALLINEA_LICENSE_FILE` in order to specify the license file.

By way of example, consider a site where there are two target architectures, x86_64 and aarch64. Create two directories, `licenses_x86_64` and `licenses_aarch64`. Then, if you want to target aarch64, you would set the license directory as follows:

```
export ALLINEA_LICENSE_DIR=/path/to/licenses_aarch64
```


Connecting to a remote system

Often you will need to login to a remote system in order to run a job. For example you may use SSH to login from your desktop machine *mydesktop* to the login node *mycluster-login* and then start a job using the queue submission command *qsub*.

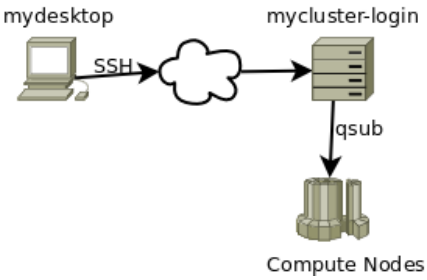

Figure 6: *Connecting to a Remote System*

The Arm Forge GUI can connect to remote systems using SSH, typically to a login node. It can also connect using Reverse Connect, typically to a batch compute node. See 3.3 Reverse Connect for more information on Reverse Connect. The remote client allows you to run the user interface on your local machine without the need for X forwarding. Native remote clients are available for Windows, Mac OS X and Linux.

No license file is required by a remote client. The license of the remote system will be used once connected.

Note: The same versions of Arm Forge must be installed on the local and remote systems in order to use DDT or MAP remotely.

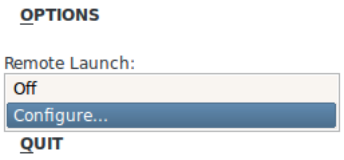

Figure 7: *Remote Launch—Configure*

To connect to a remote system click on the *Remote Launch* drop down list and select *Configure…* The Remote connections dialog will open where you can edit the necessary settings.

Remote connections dialog

The *Remote Connections Dialog* allows you to add, remove and edit connections to remote systems.

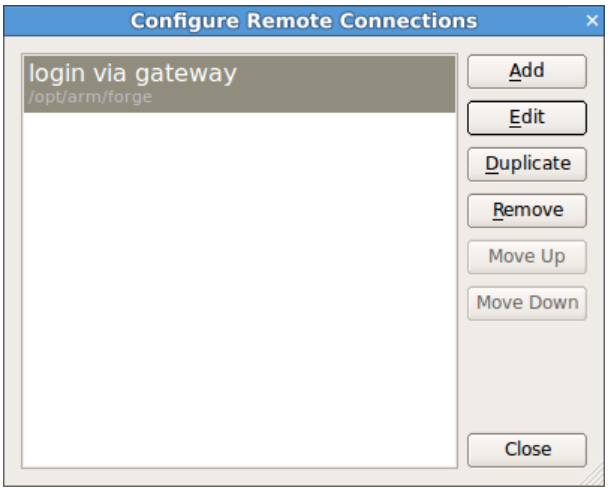

Figure 8: *Remote Connections Dialog*

When adding or editing a host, you are presented with the Remote launch settings for that host.

You may also remove a remote host from the list by clicking the *Remove* button, or duplicate an existing host using the *Duplicate* button.

You can also change the ordering of the hosts using the *Move Up* or *Move Down* buttons.

Remote launch settings

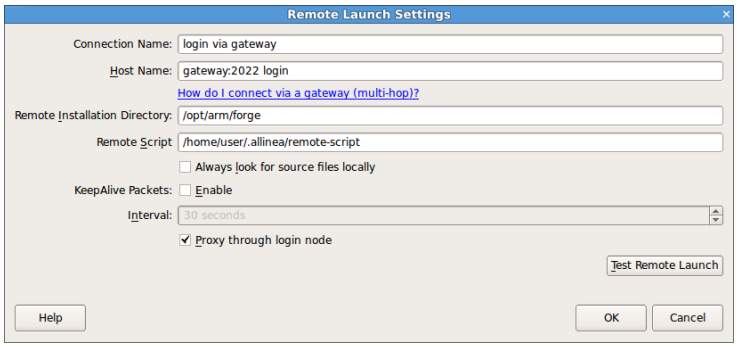

Figure 9: *Remote Launch Options*

Connection Name: An optional name for this connection. If no name is specified, the *Host Name* is used.

Host Name: The host name of the remote system you wish to connect to.

The syntax of the host name field is:

```
[username]@hostname[:port]...
```

`username` is an optional user name to use on the remote system. If not specified your local user name is used instead.

`hostname` is the host name of the remote system.

`port` is the optional port number that the remote host's SSH daemon is listening on. If not specified the default of 22 is used.

To login via one or more intermediate hosts (for example, a gateway) enter the host names in order, separated by spaces, for example, `gateway.arm.com cluster.lan`

Note: You must be able to login to the third and subsequent hosts without a password.

Additional SSH options may be specified in the `remote-exec` script covered in section A.4 Connecting to remote programs (remote-exec).

Remote Installation Directory: The full path to the Arm Forge installation on the remote system.

Remote Script: This optional script will be run before starting the remote daemon on the remote system. You may use this script to load the required modules for DDT and MAP, your MPI and compiler. See the following sections for more details. The script is usually not necessary when using *Reverse Connect*.

Always look for source files locally: Check this box to use the source files on the local system instead of the remote system.

KeepAlive Packets: Check this box to enable KeepAlive packets. These are dummy packets sent on regular intervals to keep some SSH connections from timing out. The interval can be configured from the spin box below.

Proxy through login node: Check this box to use the local RSA key to connect to both the login and the batch nodes. This is equivalent to **not** setting `ALLINEA_NO_SSH_PROXYCOMMAND`.

When this option is not set, DDT will first connect to the login node using your local RSA key and then use the key on the remote SSH configuration folder to connect to the batch node. This is equivalent to setting `ALLINEA_NO_SSH_PROXYCOMMAND=1`.

Remote script

The script may load modules using the `module` command or otherwise set environment variables. Arm Forge will source this script before running its remote daemon (your script does not need to start the remote daemon itself).

The script will be run using `/bin/sh` (usually a Bourne-compatible shell). If this is not your usual login shell, make allowances for the different syntax it might require.

You may install a site-wide script that will be sourced for all users at `/path/to/arm/forge/remote-init`.

You may also install a user-wide script that will be sourced for all of your connections at `$ALLINEA_CONFIG_DIR/remote-init`.

Note: `$ALLINEA_CONFIG_DIR` will default to `$HOME/.allinea` if not set.

Example Script

Note: This script file should be created on the remote system and the full path to the file entered in the Remote Script field box.

```
module load allinea-forge
module load mympi
module load mycompiler
```


Reverse Connect

Overview

The *Reverse Connect* feature allows you to submit your job from a shell terminal as you already do and with a small tweak to your `mpirun` (or equivalent) allow that job to connect back to Arm Forge GUI.

Reverse Connect makes it easy to debug and profile jobs with the right environment. You can easily load the required modules and prepare all setup steps necessary before launching your job.

Please note that node-locked licenses such as workstation or Arm DDT Cluster licenses do not include the Reverse Connect feature.

Usage

1. Start Arm Forge and let it connect to your remote system (typically a login node) with SSH.
2. Modify your current `mpirun` (or equivalent) command line inside your interactive queue allocation or queue submission script to enable Reverse Connect. In most of the cases it is sufficient to prefix it with `ddt/map --connect`. Almost all Arm Forge arguments beside `--offline` and `--profile` are supported by Reverse Connect.

   Example:

   ```
   $ mpirun -n 512 ./examples/wave_f
   ```

   To debug the job using Reverse Connect and 5.2 Express Launch run:

   ```
   $ ddt --connect mpirun -n 512 ./examples/wave_f
   ```

   To profile the job using Reverse Connect and 16.1 Express Launch run:

   ```
   $ map --connect mpirun -n 512 ./examples/wave_f
   ```

   If your MPI is not yet supported by Express Launch mode you can use *Compatibility Mode*.

   Debug:

   ```
   $ ddt --connect -n 512 ./examples/wave_f
   ```

   Profile:

   ```
   $ map --connect -n 512 ./examples/wave_f
   ```

3. After a short period of time the Arm Forge GUI will show the Reverse Connect request including the host (typically a batch compute node) from where the request was made and a command line summary.

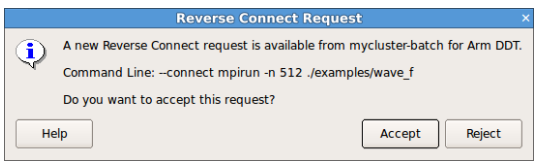

Figure 10: *Reverse Connect request*

4. You can accept the request with a click on *Accept*. Arm Forge will then connect to the specified host and execute what you specified with the command line. If you do not want to accept the request just click on *Reject*.

Connection details

If a Reverse Connect is initiated, for example with `ddt --connect`, Arm Forge starts a server listening on a port in the range between 4201 and 4240. If this port range is not suitable for some reason, such as ports are already taken by other services, you can override the port range with the environment variable `ALLINEA_REMOTED_PORTS`.

```
$ export ALLINEA_REMOTED_PORTS=4400-4500
$ ddt --connect
```

The server will now pick a free port between 4400 and 4500 (inclusive).

This connection is between the batch or submit node (where ddt –connect is run from) and the login node. This connection can also be to a compute node if for example, you are running `ddt --connect mpirun` on a single node.

Treeserver or general debugging ports

Connections are made in the following ways, depending on the use case:

Using a queue submission or using X-forwarding:

- A connection is made between the login node and the batch or submit node using ports 4242–4262.
- Connections are made between the batch or submit node and the compute nodes using ports 4242–4262.
- Connections are made from compute nodes to other compute nodes using ports 4242–4262.

Using reverse connect:

- See section 3.3.3 for details about login node to batch/submit node ports.
- Connections are made between the batch or submit node and the compute nodes using ports 4242–4262.
- Connections are made from compute nodes to other compute nodes using ports 4242–4262.

Using X forwarding or VNC

If you do not want to use the *Remote Launch* feature there are two other methods for running DDT or MAP on a remote system:

1. X forwarding is effective when the network connection is low latency, such as when the network spans a single physical site.
2. VNC (or similar Unix-supporting remote desktop software) is *strongly* recommended when the network connection is moderate or slow.

- Mac OS X users accessing a Linux or other Unix machine while using a single-button mouse should be advised that pressing the *Command* key and the single mouse button will have the same effect

as right clicking on a two button mouse. Right-clicking allows access to some important features in DDT and MAP.

You can use X forwarding to access the Arm Forge instance running on a remote Linux/Unix system from a Mac OS X system:

- Start the X11 server (available in the `X11User.pkg`).
- Set the display variable correctly to allow X applications to display by opening a terminal in Mac OS X and typing:

  ```
  export DISPLAY=:0
  ```

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

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

- Now start DDT or MAP on the remote system and the window will be displayed on your Mac.

- Windows users can use any one of a number of commercial and open source X servers, but may find VNC a viable alternative (http://www.realvnc.com/) which is available under free and commercial licensing options.
- VNC allows users to access a desktop running on a remote server (for example, a cluster login node or front end) and is more suitable than X forwarding for medium to high latency links. By setting up an SSH 'tunnel' users are usually able to securely access this remote desktop from anywhere.

  To use VNC and Arm Forge:

  - Log in to the remote system and set up a tunnel for port 5901 and 5801. On Mac OS X or any Linux/Unix systems use the `ssh` command. If you are using Putty on Windows use the GUI to setup the tunnel.

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

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

    ```
    vncserver
    ```

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

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

    ```
    krdc localhost:1
    ```

    or

    ```
    vncviewer localhost:1
    ```

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

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

Starting

To start Arm Forge simply type one of the following commands into a terminal window:

```
forge
forge program_name [arguments]
```

To start Arm Forge on Mac OS X , use the Arm Forge icon or type in the terminal window:

```
open /Applications/Arm\ Forge/Arm Forge.app [--args program_name [
   arguments]]
```

To launch additional instances of the Arm Forge application, right-click the Dock icon of a running instance of Arm Forge, and choose “Launch a new instance of Arm Forge”. Alternatively, you can use the following command in a terminal:

```
open -n /Applications/Arm\ Forge/Arm Forge.app [--args
   program_name [arguments]]
```

Note: Unless in Express Launch mode, you should not attempt to pipe input directly to the Arm Forge program. For information about how to achieve the effect of sending input to your program, please read section 9 Program input and output (DDT) or 28 Running MAP from the command line (MAP).

Once Arm Forge has started it will display the Welcome Page.

Note: In Express Launch mode (see 5.2 Express Launch (DDT) or 16.1 Express Launch (MAP)) the Welcome Page is not shown and the user is brought directly to the Run Dialog instead. If no valid license is found, the program is exited and the appropriate message is shown in the console output.

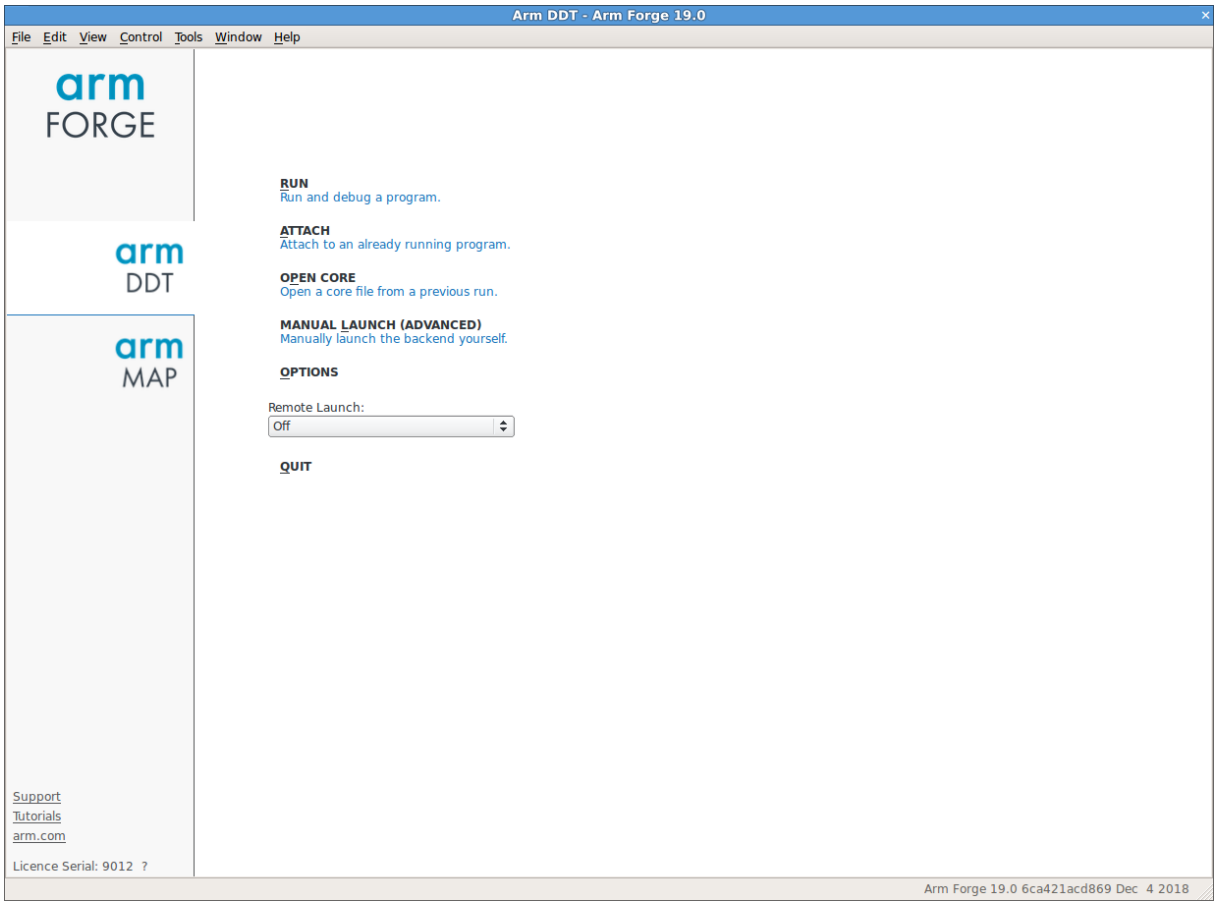

Figure 11: *DDT Welcome Page*

The *Welcome Page* allows you to choose what tool you would like to use (DDT or MAP). Click the icons on the left hand side to switch tools.

Once you have selected the tool you want to use, click the buttons in the menu to select a debugging or profiling activity.

Part II

DDT

Getting started

When compiling the program that you wish to debug, you must add the debug flag to your compile command. For most compilers this is `-g`.

It is also advisable to turn off compiler optimizations as these can make debugging appear strange and unpredictable. If your program is already compiled without debug information you will need to make the files that you are interested in again.

The *Welcome Page* allows you to choose what kind of debugging you want to do, for example you can:

- Run a program from DDT and debug it.
- Debug a program you launch manually (for example, on the command line).
- Attach to an already running program.
- Open core files generated by a program that crashed.
- Connect to a remote system and accept a Reverse Connect request.

Running a program

Figure 12: *Run Window*

If you click the *Run* button on the *Welcome Page* you see the window above. The settings are grouped into sections. Click the *Details…* button to expand a section. The settings in each section are described below.

Application

Application: The full path name to your application. If you specified one on the command line, this is filled in. You may browse for an application by clicking on the *Browse* button.

Note: Many MPIs have problems working with directory and program names containing spaces. You are advised to avoid the use of spaces in directory and file names.

Arguments: (optional) The arguments passed to your application. These are automatically filled if you entered some on the command line.

Note: Avoid using quote characters such as `'` and `"`, as these may be interpreted differently by DDT and your command shell. If you must use these and cannot get them to work as expected, please contact Arm support at Arm support.

stdin file: (optional) This allows you to choose a file to be used as the standard input (`stdin`) for your program. DDT automatically adds arguments to `mpirun` to ensure your input file is used.

Working Directory: (optional) The working directory to use when debugging your application. If this is blank then DDT's working directory is used instead.

MPI

*Note: If you only have a single process license or have selected **none** as your **MPI Implementation** the **MPI** options will be missing. The **MPI** options are not available when DDT is in single process mode. See section 5.4 Debugging single-process programs for more details about using DDT with a single process.*

Number of processes: The number of processes that you wish to debug. DDT supports hundreds of thousands of processes but this is limited by your license.

Number of nodes: This is the number of compute nodes that you wish to use to run your program.

Processes per node: This is the number of MPI processes to run on each compute node.

Implementation: The MPI implementation to use. If you are submitting a job to a queue the queue settings will also be summarized here. You may change the MPI implementation by clicking on the *Change…* button.

Note: 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 Arm support. A list of settings for common implementations is provided in Appendix E MPI distribution notes and known issues.

*Note: If your desired MPI command is not in your `PATH`, or you wish to use an MPI run command that is not your default one, you can configure this using the **Options** window (See section A.5.1 System).*

mpirun arguments: (optional): The arguments that are passed to `mpirun` or your equivalent, usually prior to your executable name in normal `mpirun` usage. You can place machine file arguments, if necessary, here. For most users this box can be left empty. You can also specify `mpirun` arguments on the command line (using the `--mpiargs` command line argument) or using the `ALLINEA_MPIRUN_ARGUMENTS` environment variable if this is more convenient.

*Note: You should **not** enter the `-np` argument as DDT will do this for you.*

*Note: You should **not** enter the `--task-nb` or `--process-nb` arguments as DDT will do this for you.*

OpenMP

Number of OpenMP threads: The number of OpenMP threads to run your application with. The `OMP_NUM_THREADS` environment variable is set to this value.

CUDA

If your license supports it, you may also debug GPU programs by enabling CUDA support. For more information on debugging CUDA programs, please see section 14 CUDA GPU debugging.

Track GPU Allocations: Tracks CUDA memory allocations made using `cudaMalloc`, and similar methods. See 12.2 CUDA memory debugging for more information.

Detect invalid accesses (memcheck): Turns on the CUDA-MEMCHECK error detection tool. See 12.2 CUDA memory debugging for more information.

Memory debugging

Clicking the *Details…* button will open the *Memory Debugging Settings* window.

See section 12.3 Configuration for full details of the available Memory Debugging settings.

Environment variables

The optional *Environment Variables* section should contain additional environment variables that should be passed to `mpirun` or its equivalent. 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.

Note: on some systems it may be necessary to set environment variables for the DDT backend itself. For example: if `/tmp` is unusable on the compute nodes you may wish to set `TMPDIR` to a different directory. You can specify such environment variables in `/path/to/ddt/lib/environment`. Enter one variable per line and separate the variable name and value with `=`, for example, `TMPDIR=/work/user`.

Plugins

The optional *Plugins* section allows you to enable plugins for various third-party libraries, such as the Intel Message Checker or Marmot. See section 13 Using and writing plugins for more information.

Click Run to start your program, or Submit if working through a queue. See section A.2 Integration with queuing systems. This runs your program through the debug interface you selected and allows your MPI implementation to determine which nodes to start which processes on.

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

When your program starts, DDT attempts to determine the MPI world rank of each process. If this fails, the following error message is displayed:

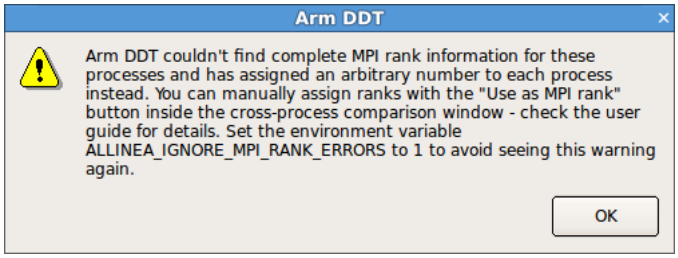

Figure 13: *MPI rank error*

This means that the number DDT shows for each process may not be the MPI rank of the process. To correct this you can tell DDT to use a variable from your program as the rank for each process.

See section 8.17 Assigning MPI ranks for details.

To end your current debugging session select the *End Session* menu option from the *File* menu. This closes all processes and stops 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.

Express Launch

Each of the Arm Forge products can be launched by typing its name in front of an existing `mpiexec` command:

```
$ ddt mpiexec -n 128 examples/hello memcrash
```

This startup method is called *Express Launch* and is the simplest way to get started. If your MPI is not yet supported in this mode, you will see an error message like this:

```
$ 'MPICH 1 standard' programs cannot be started using Express
   Launch syntax (launching with an mpirun command).

Try this instead:
    ddt --np=256 ./wave_c 20

Type ddt --help for more information.
```

This is referred to as *Compatibility Mode*, in which the `mpiexec` command is not included and the arguments to `mpiexec` are passed via a `--mpiargs="args here"` parameter.

One advantage of Express Launch mode is that it is easy to modify existing queue submission scripts to run your program under one of the Arm Forge products. This works best for Arm DDT with Reverse Connect, `ddt --connect`, for interactive debugging or in offline mode (`ddt --offline`). See 3.3 Reverse Connect for more details.

If you can not use Reverse Connect and wish to use interactive debugging from a queue you may need to configure DDT to generate job submission scripts for you. More details on this can be found in 5.10 Starting a job in a queue and A.2 Integration with queuing systems.

The following lists the MPI implementations currently supported by Express Launch:

- bullx MPI
- Cray X-Series (MPI/SHMEM/CAF)
- Intel MPI
- MPICH 2
- MPICH 3
- Open MPI (MPI/SHMEM)
- Oracle MPT
- Open MPI (Cray XT/XE/XK)
- Spectrum MPI
- Spectrum MPI (PMIx)
- Cray XT/XE/XK (UPC)

Run dialog box

In Express Launch mode, the Run dialog has a restricted number of options:

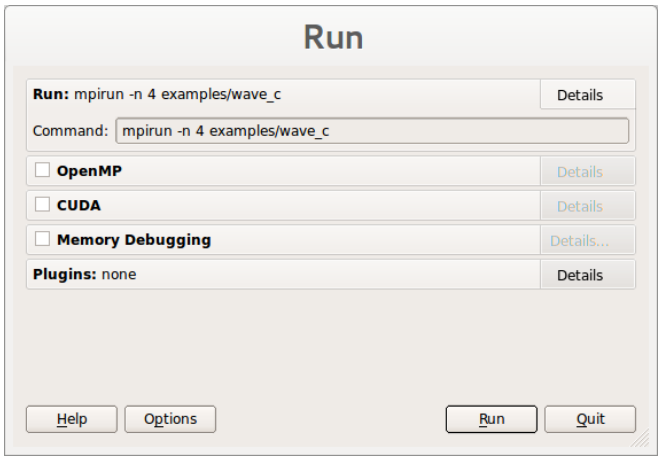

Figure 14: *Express Launch DDT Run dialog box*

`remote-exec` required by some MPIs

When using *SGI MPT*, *MPICH 1 Standard* or the MPMD variants of *MPICH 2*, *MPICH 3* or *Intel MPI*, DDT will allow `mpirun` to start all the processes, then attach to them while they're inside `MPI_Init`.

This method is often faster than the generic method, but requires the `remote-exec` facility in DDT to be correctly configured if processes are being launched on a remote machine. For more information on `remote-exec`, please see section A.4 Connecting to remote programs (remote-exec).

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

If DDT cannot find a password-free way to access the cluster nodes then you will not be able to use the specialized startup options. Instead, You can use *generic*, although startup may be slower for large numbers of processes.

In addition to the listed MPI implementations above, all MPI implementations except for *Cray MPT* DDT require password-free access to the compute nodes when explicitly starting by attaching.

Debugging single-process programs

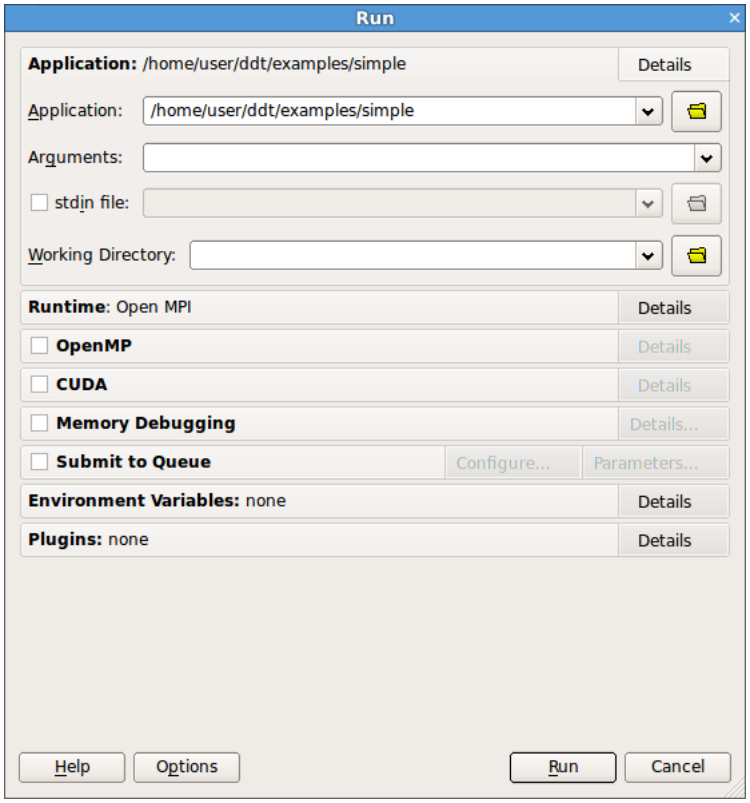

Figure 15: *Single-Process Run dialog*

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

Users with multi-process licenses can uncheck the *MPI* check box to run a single process program.

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

Click *Run* to start your program.

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

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

Debugging OpenMP programs

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

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

1. Parallel regions created with `#pragma omp parallel` (C) or `!$OMP PARALLEL` (Fortran) will usually not be nested in the *Parallel Stack View* under the function that contained the `#pragma`. Instead they will appear under a different top-level item. The top-level item is often in the OpenMP runtime code, and the parallel region appears several levels down in the tree.
2. Some OpenMP libraries only create the threads when the first parallel region is reached. It is possible you may only see one thread at the start of the program.
3. You cannot step into a parallel region. Instead, check the *Step threads together* box and use the *Run to here* command to synchronize the threads at a point inside the region. These controls are discussed in more detail in their own sections of this document.
4. You cannot step out of a parallel region. Instead, use *Run to here* to leave it. Most OpenMP libraries work best if you keep the *Step threads together* box ticked until you have left the parallel region. With the Intel OpenMP library, this means you will see the *Stepping Threads* window and will have to click *Skip All* once.
5. Leave *Step threads together* off when you are outside a parallel region, as OpenMP worker threads usually do not follow the same program flow as the main thread.
6. To control threads individually, use the *Focus on Thread* control. This allows you to step and play one thread without affecting the rest. This is helpful when you want to work through a locking situation or to bring a stray thread back to a common point. The Focus controls are discussed in more detail in their own section of this document.
7. Shared OpenMP variables may appear twice in the Locals window. This is one of the many unfortunate side-effects of the complex way OpenMP libraries interfere with your code to produce parallelism. One copy of the variable may have a nonsense value, this is usually easy to recognize. The correct values are shown in the Evaluate and Current Line windows.
8. Parallel regions may be displayed as a new function in the stack views. Many OpenMP libraries implement parallel regions as automatically-generated "outline" functions, and DDT shows you this. To view the value of variables that are not used in the parallel region, you may need to switch to thread 0 and change the stack frame to the function you wrote, rather than the outline function.
9. Stepping often behaves unexpectedly inside parallel regions. Reduction variables usually require some sort of locking between threads, and may even appear to make the current line jump back to the start of the parallel region. If this happens step over several times and you will see the current line comes back to the correct location.
10. Some compilers optimize parallel loops regardless of the options you specified on the command line. This has many strange effects, including code that appears to move backwards as well as forwards, and variables that are not displayed or have nonsense values because they have been optimized out by the compiler.
11. The thread IDs displayed in the Process Group Viewer and Cross-Thread Comparison window will match the value returned by `omp_get_thread_num()` for each thread, *but only if your OpenMP implementation exposes this data to DDT*. GCC's support for OpenMP (GOMP) needs to be built with TLS enabled with our thread IDs to match the return `omp_get_thread_num()`, whereas your system GCC most likely has this option disabled. The same thread IDs will be displayed as tooltips for the threads in the thread viewer, but only your OpenMP implementation exposes this data.

If you are using DDT with OpenMP and would like to tell us about your experiences, please contact Arm support at Arm support, with the subject title *OpenMP feedback*.

Manual launching of multi-process non-MPI programs

DDT can only launch MPI programs and scalar (single process) programs itself. The *Manual Launch (Advanced)* button on the *Welcome Page* allows you to debug multi-process and multi-executable programs. These programs do not necessarily need to be MPI programs. You can debug programs that use other parallel frameworks, or both the client and the server from a client/server application in the same DDT session.

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

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

1. Click the *Manual Launch (Advanced)* button.
2. Select 2 processes

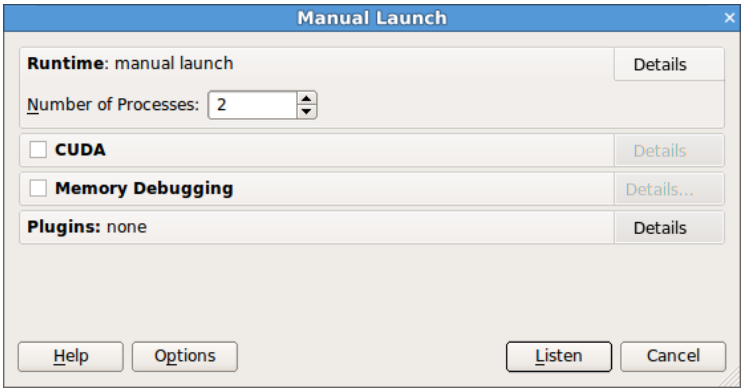

Figure 16: *Manual Launch Window*

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

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

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

Figure 17: *Manual Launch Process Groups*

After you have run the initial programs you may add extra processes to the DDT session, for example extra clients could be added, using ddt-client in the same way.

```
ddt-client client2 &
```


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

Debugging MPMD programs

The easiest way to debug MPMD programs is by using Express Launch to start your application.

To use Express Launch, simply prefix your normal MPMD launch line with ddt, for example:

```
ddt mpirun -n 1 ./master : -n 2 ./worker
```

For more information on Express Launch, and compatible MPI implementations, see section 5.2.

Debugging MPMD programs without Express Launch

If you are using Open MPI, MPICH 2, MPICH 3 or Intel MPI, DDT can be used to debug multiple program, multiple data (MPMD) programs. To start an MPMD program in DDT:

1. MPICH 2 and Intel MPI only: Select the MPMD variant of the MPI Implementation on the *System* page of the *Options* window, for example, for MPICH 2 select *MPICH 2 (MPMD)*.
2. Click the *Run* button on the *Welcome Page*.
3. Select one of the MPMD programs in the *Application* box, it does not matter what executable you choose.
4. Enter the total amount of processes for the MPMD job in the *Number of processes* box.
5. Enter an MPMD style command line in the *mpirun Arguments* box in the MPI section of the *Run* window, for example:

   ```
   -np 4 hello : -np 4 program2
   ```

   or:

   ```
   --app /path/to/my_app_file
   ```

6. Click the Run button.

Note: be sure that the sum of processes in step 5 is equal to the number of processes set in step 4.

Debugging MPMD programs in Compatibility mode

If you are using Open MPI in *Compatibility* mode, for example, because you do not have SSH access to the compute nodes, then replace:

```
-np 2 ./progc.exe : -np 4 ./progf90.exe
```

in the *mpirun Arguments* / appfile with this:

```
-np 2 /path/to/ddt/bin/ddt-client ./progc.exe : -np 4
/path/to/ddt/bin/ddt-client ./progf90.exe
```


Opening core files

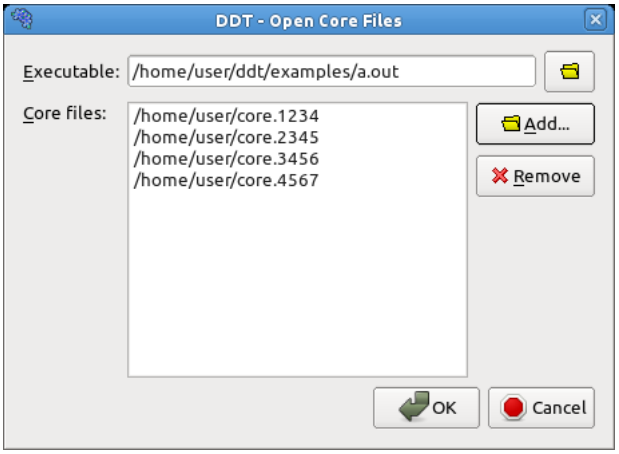

Figure 18: *The Open Core Files Window*

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

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

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

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

Attaching to running programs

DDT can attach to running processes on any machine you have access to, whether they are from MPI or scalar jobs, even if they have different executables and source pathnames. Clicking the *Attach to a Running Program* button on the *Welcome Page* shows DDT's *Attach Window*:

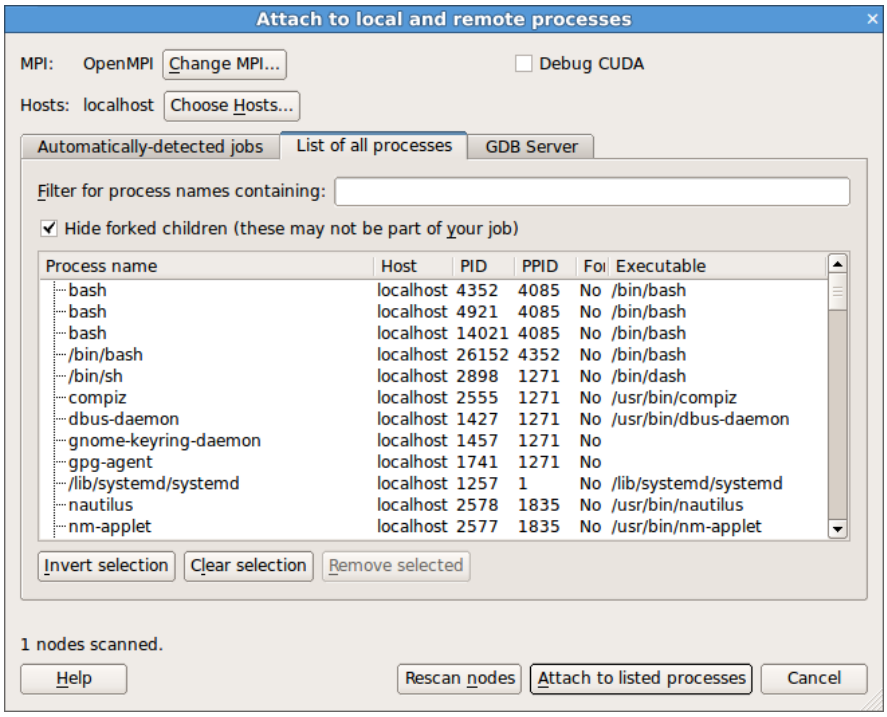

Figure 19: *Attach Window*

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

Automatically detected MPI jobs

DDT can automatically detect MPI jobs started on the local host for selected MPI implementations. This also applies to other hosts you have access to, if an *Attach Hosts File* is configured. See section A.5.1 System for more details.

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

Note: non-MPI programs that were started using MPI may not appear in this window. For example `mpirun -np 2 sleep 1000`

Attaching to a subset of an MPI job

You may want to attach only to a subset of ranks from your MPI job. You can choose this subset using the *Attach* to ranks box on the *Automatically-detected MPI jobs* tab of the Attach Window. You may change the subset later by selecting the *File* → *Change Attached Processes...* menu item. The menu item is only available for jobs that were attached to, and not for jobs that were launched using DDT.

Manual process selection

You can manually select which processes to attach to from a list of processes using the *List of all processes* tab of the *Attach Window*. If you want to attach to a process on a remote host see section A.4 Connecting

to remote programs (remote-exec) first.

Initially the list of processes is blank while DDT scans the nodes, provided in your node list file, for running processes. When all the nodes have been scanned (or have timed out) the window appears as shown above. Use the Filter box to find the processes you want to attach to. On non-Linux platforms you also need to select the application executable you want to attach to. Ensure that the list shows all the processes you wish to debug in your job, and no extra/unnecessary processes. You may modify the list by selecting and removing unwanted processes, or alternatively selecting the processes you wish to attach to and clicking on *Attach to Selected Processes*. If no processes are selected, DDT uses the whole visible list.

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

With some supported MPI implementations (for example, Open MPI) DDT shows MPI processes as children of the `mpirun` (or equivalent) command, as shown in the following figure. Clicking the `mpirun` command automatically selects all the MPI child processes.

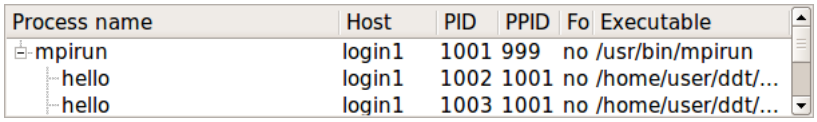

Figure 20: *Attaching with Open MPI*

Some MPI implementations (such as MPICH 1) create forked (child) processes that are used for communication, but are not part of your job. To avoid displaying and attaching to these, make sure the *Hide Forked Children* box is ticked. DDT's definition of a forked child is a child process that shares the parent's name. Some MPI implementations create your processes as children of each other. If you cannot see all the processes in your job, try clearing this checkbox and selecting specific processes from the list.

Once you click on the *Attach to Selected/Listed Processes* button, DDT uses `remote-exec` to attach a debugger to each process you selected and proceeds to debug your application as if you had started it with DDT. When you end the debug session, DDT detaches from the processes rather than terminating them, this allows you to attach again later if you wish.

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

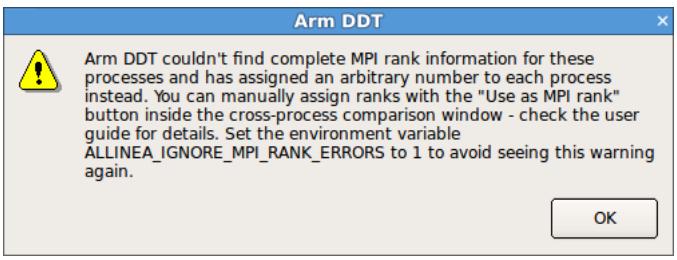

Figure 21: *MPI rank error*

If there is no rank, for example, if you have attached to a non-MPI program, then you can ignore this message and use DDT as normal. If there is, then you can easily tell DDT what the correct rank for each

process via the *Use as MPI Rank* button in the *Cross-Process Comparison Window*. See section 8.17 Assigning MPI ranks for details.

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

Configuring attaching to remote hosts

To attach to remote hosts in DDT, click the *Choose Hosts* button in the attach dialog. This displays the list of hosts to be used for attaching.

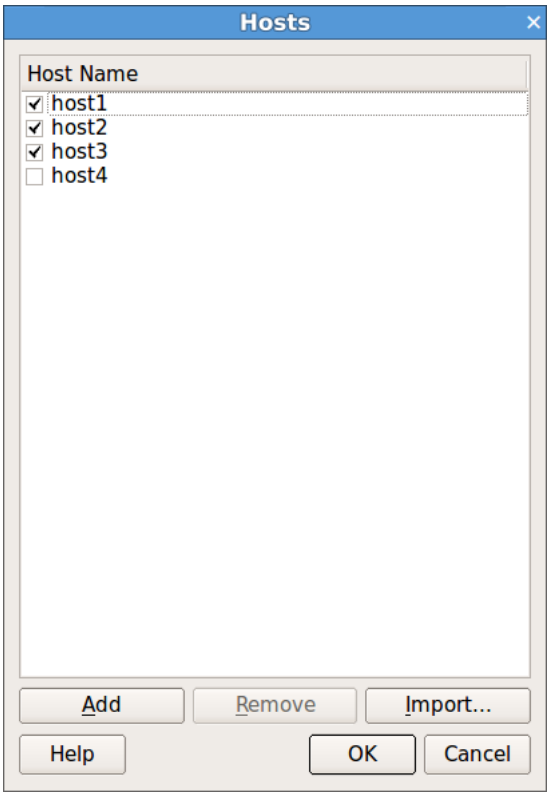

Figure 22: *Choose Hosts Window*

From here you can add and remove hosts, as well as unchecking hosts that you wish to temporarily exclude.

To import a list of hosts from a file, click the *Import* button.

The hosts list populates using the *attach Hosts File*. To configure the hosts, use the *Options window*: *File* → *Options* (*Arm Forge* → *Preferences* on Mac OS X) .

Each remote host is scanned for processes, and the result is displayed in the attach window. If you have trouble connecting to remote hosts, please see section A.4 Connecting to remote programs (remote-exec).

Using DDT command-line arguments

As an alternative to starting DDT and using the *Welcome Page*, DDT can instead be instructed to attach to running processes from the command-line.

To do so, you need to specify a list of hostnames and process identifiers (PIDs). If a hostname is omitted then localhost is assumed.

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

```
mark@holly:~$ ddt --attach=11057,node5:11352
```

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

```
mark@holly:~$ cat /home/mark/ddt/examples/hello.list

node1:11057
node1:11094
node2:11352
node2:11362
node3:12357

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

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

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

Starting a job in a queue

In most cases you can debug a job simply by putting `ddt --connect` in front of the existing `mpiexec` or equivalent command in your job script. If a GUI is running on the login node or it is connected to it via the remote client, then a message is displayed prompting you with the option to debug the job when it starts.

See 5.2 Express Launch and 3.3 Reverse Connect for more details.

If DDT has been configured to be integrated with a queue/batch environment, as described in section A.2 Integration with queuing systems then you may use DDT to submit your job directly from the GUI. In this case, a Submit button is presented on the *Run Window*, instead of the ordinary *Run* button. Clicking *Submit* from the *Run Window* will display the queue status until your job starts. DDT will execute the display command every second and show you the standard output. If your queue display is graphical or interactive then you cannot use it here.

If your job does not start or you decide not to run it, click on *Cancel Job*. If the regular expression you entered for getting the job id is invalid or if an error is reported then DDT will not be able to remove your job from the queue. In this case it is strongly recommended that 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 connects to DDT and you can debug it.

Using custom MPI scripts

On some systems a custom '`mpirun`' replacement is used to start jobs, such as `mpiexec`. DDT normally uses whatever the default for your MPI implementation is, so for MPICH 1 it would look for `mpirun` and not `mpiexec`. This section explains how to configure DDT to use a custom `mpirun` command for job start up.

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

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

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

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

Everything else, like the name of the command and the format of its arguments remains constant. To use a command like this in DDT, you adapt the queue submission system described in the previous section. For this `mpiexec` example, the settings are as shown here:

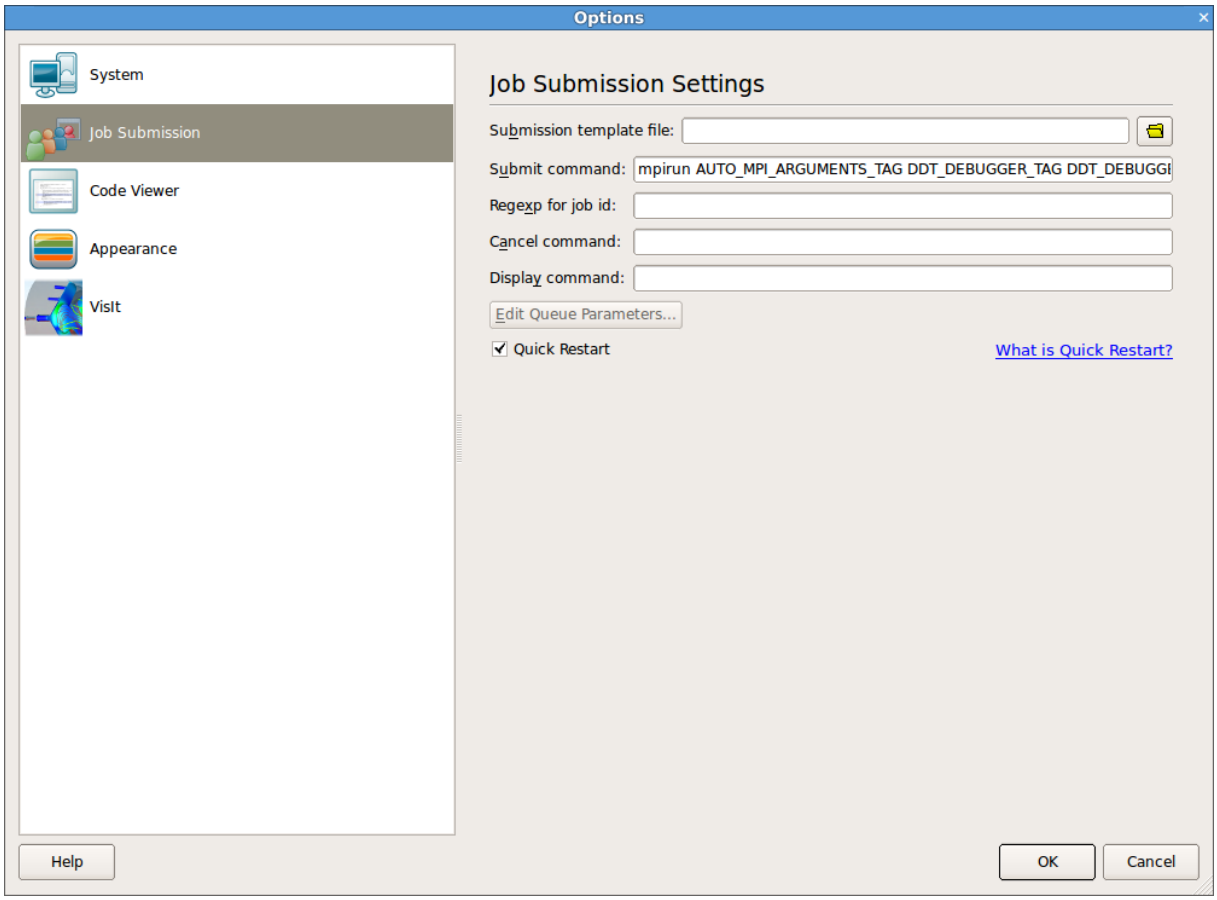

Figure 23: *DDT Using Custom MPI Scripts*

As you can see, most of the settings are left blank. There are some 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 similar to the following:

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

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

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

This follows the same replacement rules described above and in detail in section A.2 Integration with queuing systems. The options settings for this example might be:

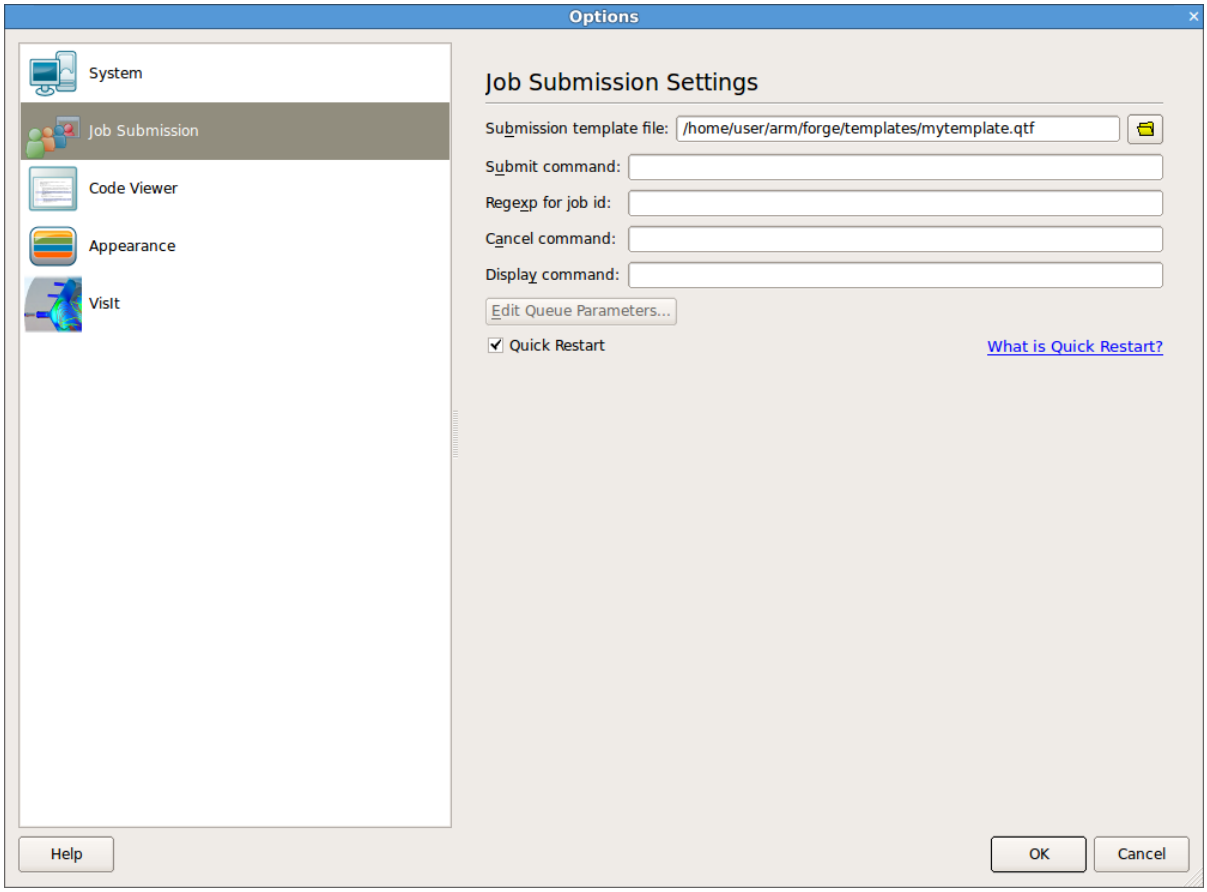

Figure 24: *DDT Using Substitute MPI Commands*

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. In this case what would actually be executed might be `mpiexec -config /tmp/ddt-temp-0112` or similar. Therefore, any argument like `-config` must be last on the line, because DDT will add a file name to the end of the line. Other arguments, if there are any, can come first.

It is recommended that you read the section on queue submission, as there are many features described there that might be useful to you if your system uses a non-standard start up command.

If you do use a non-standard command, please contact Arm support at Arm support.

Starting DDT from a job script

The usual way of debugging a program with Arm DDT in a queue/batch environment is with Reverse Connect and let it connect back from inside the queue to the GUI. See 3.3 Reverse Connect for more details on Reverse Connect.

To do this replace your usual program invocation with a Arm DDT `--connect` command such as the following:

```
ddt --connect --start MPIEXEC -n NPROCS PROGRAM [ARGUMENTS]
```

The following could also be used:

```
ddt --connect --start --once --np=NPROCS -- PROGRAM [ARGUMENTS]
```

In these examples `MPIEXEC` is the MPI launch command, `NPROCS` is the number of processes to start, `PROGRAM` is the program to run, and `ARGUMENTS` are the arguments to the program.

The `--once` argument tells Arm DDT to exit when the session ends.

The alternative to Reverse Connect for debugging a program in a queue/batch environment is to configure Arm DDT to submit the program to the queue for you. See section 5.10 Starting a job in a queue.

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

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

   ```
   ddt --start MPIEXEC -n NPROCS PROGRAM [ARGUMENTS]
   ```

   Or the following:

   ```
   ddt --start --no-queue --once --np=NPROCS -- PROGRAM [
      ARGUMENTS]
   ```

   In these examples `MPIEXEC` is the MPI launch command, `NPROCS` is the number of processes to start, `PROGRAM` is the program to run, and `ARGUMENTS` are the arguments to the program.
4. Submit the job script to the queue. The `--once` argument tells DDT to exit when the session ends.

Attaching via gdbserver

DDT can attach to debugging sessions that have been started by gdbserver.

This is typically used for debugging embedded devices only. This should be considered as an expert mode and would not normally be used to debug an application running on a server or workstation.

To prepare for using this mode, you must first start a gdbserver on the target device. Please see https://sourceware.org/gdb/onlinedocs/gdb/Server.html for further details as invocation may be system dependent.

You may then attach to a running application either via the command line or the user interface.

To attach via the command line use:


```
ddt --attach-gdbserver=host:port target-executable
```

Note that the arguments are not optional.

To attach via the user interface, select the Attach dialog on DDT's welcome page. Select the GDB Server tab and substitute the appropriate settings.

If the gdbserver has been used to launch an application, then it will have been stopped before starting the user code. In this case, add a breakpoint in the main function using the Add Breakpoint button, and then play until this is reached. After this point is reached, source code will be displayed.

UPC

The DDT configuration depends on the UPC compiler used.

GCC UPC

DDT can debug applications compiled with GCC UPC 4.8 with TLS disabled. See section F.5 GNU.

To run a UPC program in DDT you need to select the MPI implementation *"GCC libupc SMP (no TLS)"*

Berkeley UPC

To run a Berkeley UPC program in DDT you need to compile the program using `-tv` flag and then select the same MPI implementation used in the Berkeley compiler build configuration.

The Berkeley compiler must be build using the MPI transport.

See section F.3 Berkeley UPC compiler.

Numactl

DDT supports launching programs via `numactl`. DDT supports this feature for MPI programs but has limited support for non-MPI programs.

MPI and SLURM

DDT can attach to MPI programs launched via `numactl` with or without SLURM. The recommended way to launch via `numactl` is to use express launch mode (5.2 Express Launch).

```
$ ddt mpiexec -n 4 numactl -m 1 ./myMpiProgram.exe
$ ddt srun -n 4 numactl -m 1 ./myMpiProgram.exe
```

It is also possible to launch via `numactl` using compatibility mode (5.1 Running a program). When using compatibility mode, you must specify the full path to `numactl` in the Application box. You can find the full path by running:

```
which numactl
```


Enter the name of the required application in the Arguments field, after all arguments to be passed to `numactl`. It is not possible to pass any more arguments to the parallel job runner when using this mode for launching.

Note: When using memory debugging, with a program launched via `numactl`, the Memory Statistics view will report all memory as 'Default' memory type unless allocated with memkind. (12.6 Memory Statistics)

Non-MPI Programs

There is a minor caveat to launching non-MPI programs via `numactl`. If you are using SLURM, set `ALLINEA_STOP_AT_MAIN=1`, otherwise DDT will not be able to attach to the program. For example, the two following commands are examples of launching non-MPI programs via `numactl`:

```
$ ddt numactl -m 1 ./myNonMpiProgram.exe
$ ALLINEA_STOP_AT_MAIN=1 ddt srun \
      numactl -m 1 ./myNonMpiProgram.exe
```

Once launched, the program stops in `numactl` main. To resume debugging as normal, set a breakpoint in your code (optional), then use the play and pause buttons to progress and pause the debugging, respectively.

Python debugging

Overview

Python debugging in DDT has the following limited support:

- Debugging Python scripts running under the CPython interpreter (version 2.7, 3.5 and 3.6 only).
- Decoding the stack to show Python frames, function names and line numbers.
- Displaying Python local and global variables when a Python frame is selected.
- Stopping on breakpoints and exceptions in native libraries that were invoked from Python code.
- Debugging MPI programs written in Python using mpi4py.

This feature is useful when debugging a mixed C, C++, Fortran and Python program that crashes somewhere in native code. If this native code was invoked from a Python function, then you can examine the Python stack and local variables that led to the crash. The feature does not currently support breakpoints, stepping, evaulating Python variables, or the current line window.

Prerequisites

On your system, a debug version of Python must be available to DDT. We suggest doing this by compiling python yourself with debug information and disabling optimisation. Alternatively you can install just the debug symbols however this is dependant entirely on the distribution vendor if the correct information has been compiled into the binary for DDT to be able to operate normally. If you wish to take this approach however one solution is to install the Python debug symbols package. You may need to enable additional debug respositories in your package manager.

On Ubuntu:

```
$ sudo apt-get install python2.7-dbg
```


On Redhat:

```
$ sudo yum install python-debug
```

On SuSE:

```
$ sudo zypper install python-base-debug
```

Python debugging depends on GDB 7.12.1. If GDB 7.6.2 is the selected debugger, you need to change to GDB 7.12.1, using: *Go to File → Options → System* and set the Debugger field to *Automatic (recommended)*.

Running

To debug Python scripts, start the Python interpreter that will execute the script under DDT. To get line level resolution, rather than function level resolution, insert `%allinea_python_debug%` before your script when passing arguments to Python. To run the demo in the examples folder, change into the `examples` folder and run the following steps.

Note: The demo requires `mpi4py` to be installed.

1. **`$ make -f python.makefile`**
2. **`$ ../bin/ddt -np 4 python %allinea_python_debug% python-debugging.py`**

   Note: On loading into DDT you will be inside the C code. This is normal as you are debugging the python binary.
3. Click Run.
4. Click Play/Continue. If successful, you see a segfault. DDT can help you find the source of this segfault. DDT can help us find the source of this segfault.
5. To see the Python local variables, open the 'Stacks' view and select a Python frame.

Note: DDT does not search in your `PATH` when launching executables, so you must specify the full path to Python.

Please be aware DDT loads the correct python debugging information based on executable name. The supported names are **python2.7**, **python3.5** and **python3.6**, these cannot be symbolic links. Issues with symbolic links usually manifest when setting up a virtual environments as it takes the name you specify for the environment.

Overview

Arm DDT uses a tabbed-document interface as a method of presenting multiple documents. This allows you to have many source files open. You can view one file in the full workspace area, or two if the *Source Code Viewer* is 'split'.

Each component of Arm DDT is a dockable window, which may be dragged around by a handle, usually on the top or left-hand edge. Components can also be double-clicked, or dragged outside of Arm DDT, to form a new window. You can hide or show most of the components using the *View* menu. The screenshot shows the default Arm DDT layout.

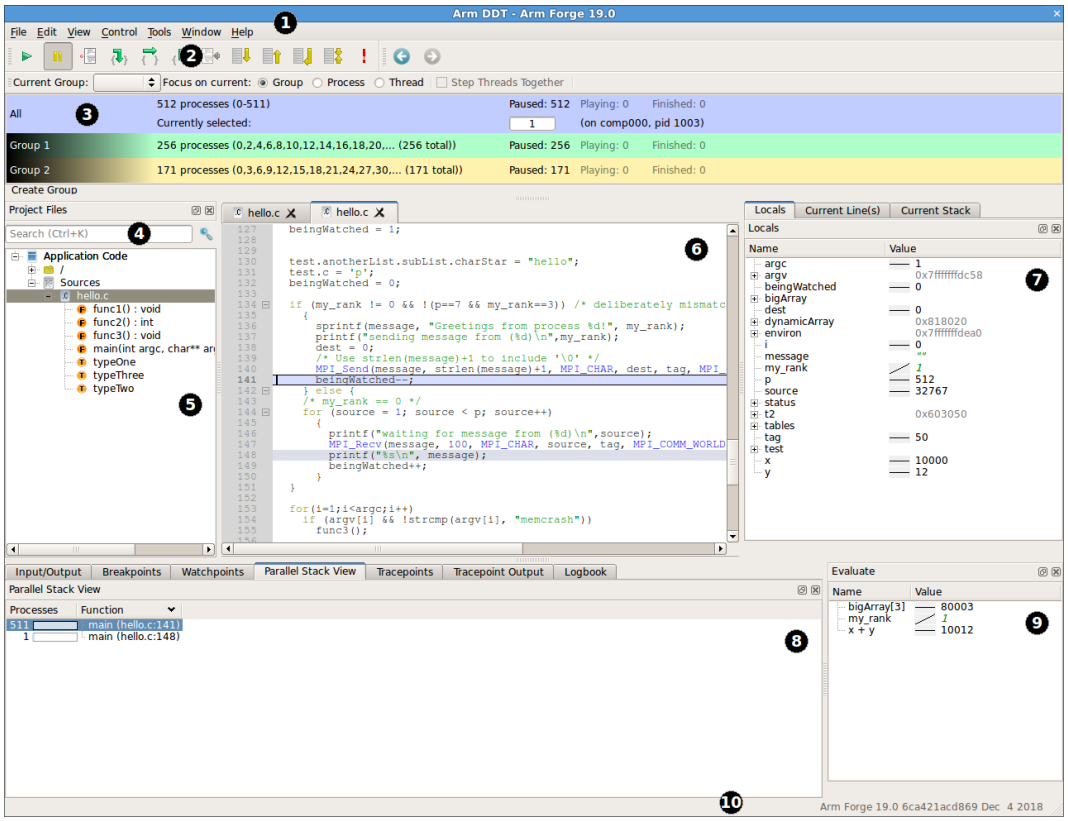

Figure 25: *DDT Main Window*

The following table shows the key components:

Key
(1) Menu Bar
(2) Process Controls
(3) Process Groups
(4) Find File or Function
(5) Project Files
(6) Source Code
(7) Variables and Stack of Current Process/Thread
(8) Parallel Stack, IO and Breakpoints
(9) Evaluate Window
(10) Status Bar

Note: On some platforms, the default screen size can be insufficient to display the status bar. If this occurs, you should expand the Arm DDT window until it 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, Arm 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 *File* menu. Enter a file name (or select an existing file) for the save file and click OK. To load a session again simply choose the *Load Session* option from the *File* menu, choose the correct file and click OK.

Source code

Arm Forge provides code viewing, editing and rebuilding features. It also integrates with the Git, Subversion and Mercurial version control systems and provides static analysis to automatically detect many classes of common errors.

The code editing and rebuilding capabilities are not designed for developing applications from scratch, but they are designed to fit into existing debugging or profiling sessions that are running on a current executable.

The same capabilities are available for source code whether running remotely (using the remote client) or whether connected directly to your system.

Viewing

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

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

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

The source code viewer supports automatic color syntax highlighting for C and Fortran.

You can hide functions or subroutines you are not interested in by clicking the ’-’ glyph next to the first line of the function. This will collapse the function. Simply click the ’+’ glyph to expand the function again.

Editing

Source code may be edited in the code viewer windows of DDT. The actions *Undo*, *Redo*, *Cut*, *Copy*, *Paste*, *Select all*, *Go to line*, *Find*, *Find next*, *Find previous*, and *Find in files* are available from the *Edit* menu. Files may be opened, saved, reverted and closed from the *File* menu.

Note: Information from Arm DDT will not match edited source files until the changes are saved, the binary is rebuilt, and the session restarted.

If the currently selected file has an associated header or source code file, it can be opened by right-clicking in the editor and choosing *Open <filename>.<extension>*. There is a global shortcut on function key F4, available in the *Edit* menu as *Switch Header/Source* option.

To edit a source file in an external editor, right-click the editor for the file and choose *Open in external editor*. To change the editor used, or if the file does not open with the default settings, open the Options window by selecting *File → Options* (*Arm Forge → Preferences* on Mac OS X) and enter the path to the preferred editor in the Editor box, for example `/usr/bin/gedit`.

If a file is edited the following warning will be displayed at the top of the editor:

Figure 26: *File Edited Warning*

This is to warn that the source code shown is not the source that was used to produce the currently executing binary. The source code and line numbers may not match the executing code.

Rebuilding and restarting

If source files are edited, the changes will not take effect until the binary is rebuilt and the session restarted. To configure the build command choose *File → Configure Build…*, enter a build command and a directory in which to run the command, and click *Apply*.

To issue the build command choose *File → Build*, or press Ctrl+B (Cmd+B on Mac OS X). When a build is issued the *Build Output* view is shown. Once a rebuild succeeds it is recommended to restart the session with the new build by choosing *File → Restart Session*.

Committing changes

Changes to source files may be committed using one of Git, Mercurial, and Subversion. To commit changes choose *File → Commit…*, enter a commit message to the resulting dialog and click the commit button.

Project Files

The *Project Files* tree shows a list of source files for your program. Click on a file in the tree to open it in the *Code Viewer*. You may also expand a source file to see a list of classes, functions, defined in that source file (C / C++ / Fortran only).

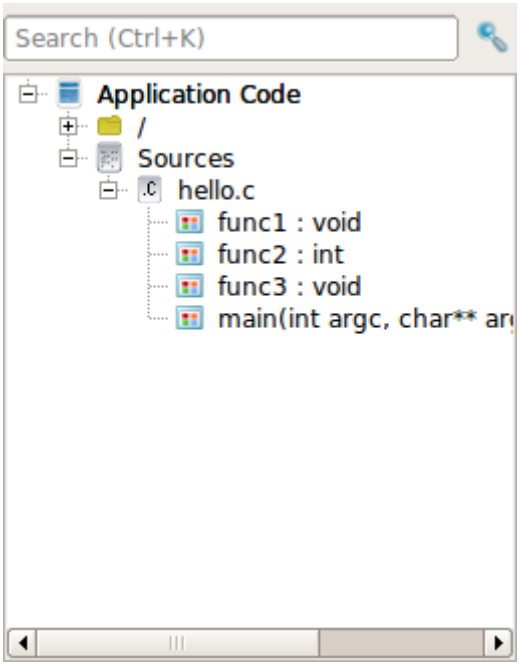

Figure 27: *Function Listing*

Clicking on any source code element (class, function, and so on) will display it in the Source Code viewer.

Application and external code

Arm DDT automatically splits your source code into *Application Code*, which is source code from your application and *External Code*, which is code from third party libraries. This allows you to quickly distinguish between your own code and third party libraries.

You can control exactly which directories are considered to contain Application Code using the *Application / External Directories* window. Right-click on the *Project Files* tree to open the window.

The checked directories are the directories containing Application Code. Once you have configured them to your satisfaction click *Ok* to update the *Project Files* tree.

Finding lost source files

In some situations, 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 while in the *Project Files* tab, and selecting *Add/view Source Directory(s)*. You can also specify extra source directories on the command line using the `--source-dirs` command line argument (separate each directory with a colon).

It is also possible to add an individual file, if this file has moved since compilation, or is on a different (but visible) file system. To do this right-click in the *Project Files* tab and select the *Add File* option.

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

Finding code or variables

Find Files or Functions

The *Find Files Or Functions* box appears above the source file tree in the *Project Files* view.

You can type the name of a file, function, or other source code element (such as classes, Fortran modules, and so on) in this box to search for that item in the source tree. You can also type just part of a name to see all the items whose name contains the text you typed.

Double-click on a result to jump to the corresponding source code location for that item.

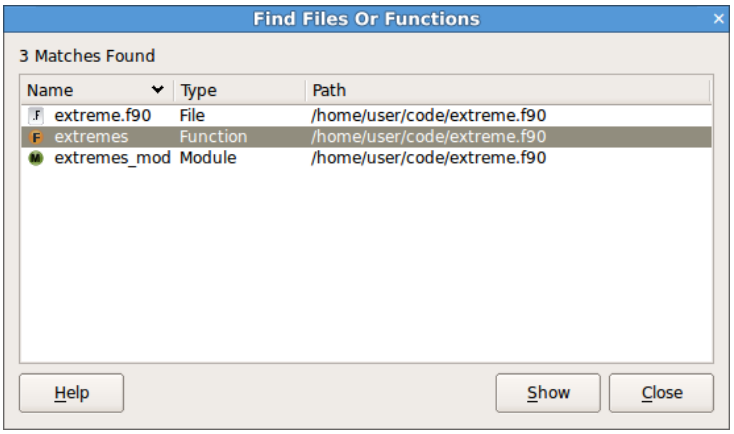

Figure 28: *Find Files Or Functions dialog*

Find

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

DDT will search from the current cursor position for the next or previous occurrence of the search term. Click on the magnifying glass icon for more search options.

Case Sensitive: When checked, DDT will perform a case sensitive search. For example, `Hello` will not match `hello`.

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

Use Regular Expressions: When this is checked, your search may use Perl-style regular expressions.

Find in Files

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

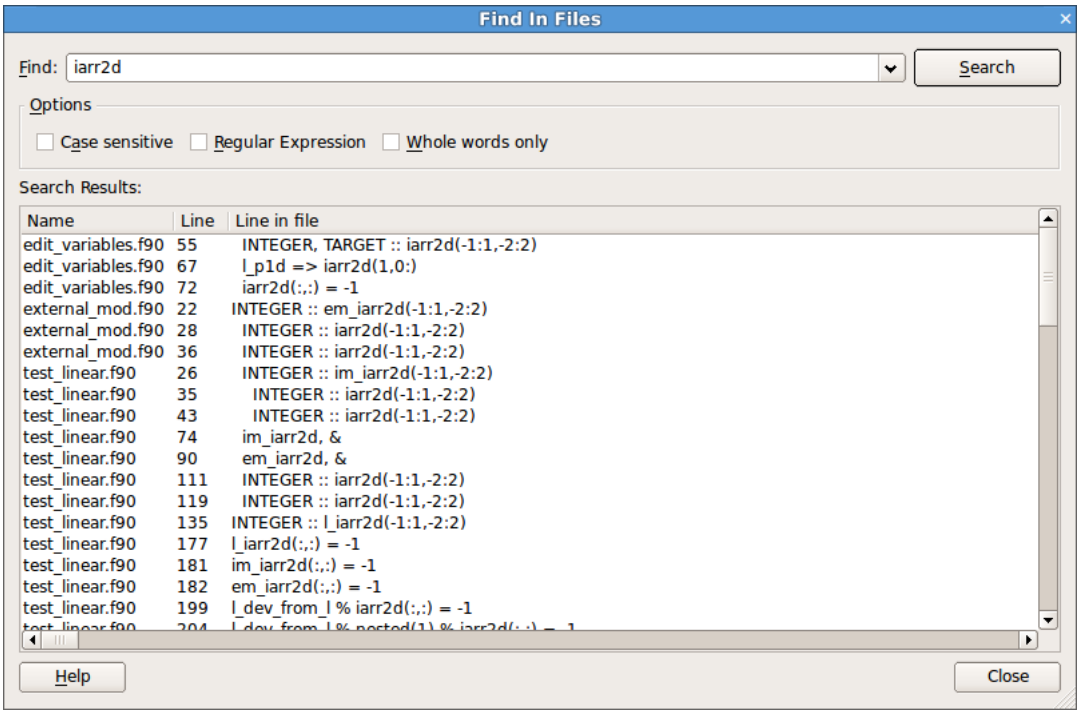

Figure 29: *Find in Files dialog*

Case sensitive: When checked, DDT will perform a case sensitive search. For example, `Hello` will not match `hello`.

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

Regular Expression: When checked, DDT will interpret the search term as a regular expression rather than a fixed string. The syntax of the regular expression is identical to that described in the appendix I.6 Job ID regular expression.

Go To Line

DDT has a go to line function which enables the user to go directly to a line of code. This is found in the *Edit* menu. A window will be displayed in the centre of your screen. Enter the line number you wish to see and click *OK*. This will take you to the correct line providing that you entered a line that exists. You can use the hotkey *CTRL-L* to access this function quickly.

Navigating through source code history

After jumping to a source code location or opening a new file, it is possible to return to the previous location using the ”Navigate backwards in source code history” button on the toolbar or item in the ”Edit” menu. This can be done several times to revisit previous locations in the source code.

After navigating backwards, you can also use the ”Navigate forwards in source code history” toolbar button or ”Edit” menu item to return to the previous location.

Static analysis

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

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

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

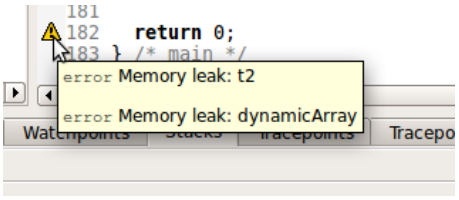

Figure 30: *Static Analysis Error Annotation*

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

Version control information

The version control integration in DDT and MAP allows users to see line-by-line information from Git, Mercurial or Subversion next to source files. Information is color-coded to indicate the age of the source line.

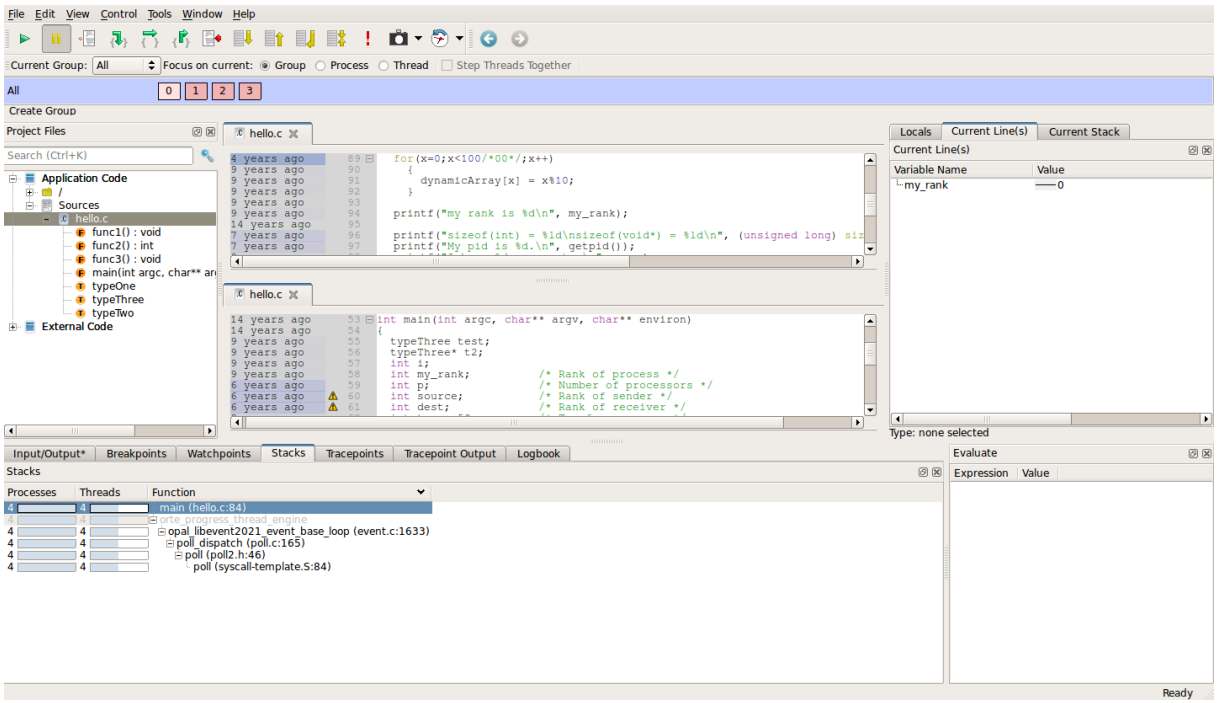

Figure 31: *DDT running with Version Control Information enabled*

To enable select the *Version Control Information* option from the *View* menu. When enabled columns to left of source code viewers are shown. In these columns are displayed how long ago the line was added/modified. Each line in the information column is highlighted in a color to indicate its age. The lines changed in the current revision are highlighted in red.

Where available lines with changes not committed are highlighted in purple. All other lines are highlighted with a blend of transparent blue and opaque green where blue indicates old and green young.

Currently uncommitted changes are only supported for Git. Arm Forge will not show *any* version control information for files with uncommitted changes when using Mercurial or Subversion.

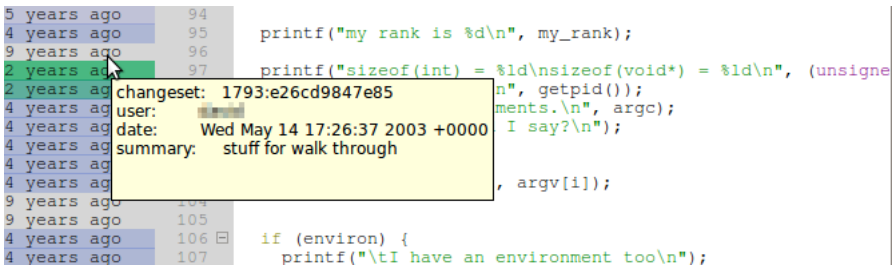

Figure 32: *Version Control Information—Tooltips*

A folded block of code displays the annotation for the most recently modified line in the block.

Hovering the cursor over the information column reveals a tool-tip containing a preview of the commit message for the commit that last changed the line.

Figure 33: *Version Control Information—Context Menu*

To copy the commit message right-click the column on the desired row and from the menu select Copy Commit Message.

See also Version control breakpoints and tracepoints.

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 are described in the following sections.

For single process mode, the commands and behaviors are identical, but apply to only a single process, freeing the user from concerns about process groups.

Process control and process groups

MPI programs are designed to run as more than one process and can span many machines. Arm 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-colored rows. Each row relates to a group of processes and operations can be performed on the currently highlighted group (for example, playing, pausing and stepping) by clicking on the toolbar buttons. Switch between groups by clicking on them or their processes. The highlighted group is indicated by a lighter shade. Groups can be created, deleted, or modified by the user at any time, with the exception of the *All* group, which cannot be modified.

Groups are added by clicking on the *Create Group* button or from a context-sensitive menu that appears when you right-click on the process group widget. This menu can also be used to rename groups, delete individual processes from a group and jump to the current position of a process in the code viewer. You can load and save the current groups to a file, and you can create sub-groups from the processes currently playing, paused or finished. You can even create a sub-group excluding the members of another group. For example, to take the complement of the *Workers* group, select the *All* group and choose *Copy, but without Workers*.

You can also use the context menu to switch between the two different methods of viewing the list of groups in Arm DDT. These methods are the detailed view and the summary view.

Detailed view

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

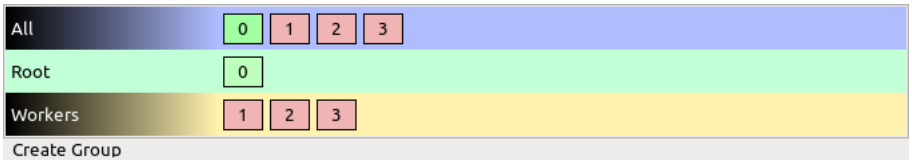

Figure 34: *The Detailed Process Group View*

In the detailed view, each process is represented by a square containing its MPI rank (0 through n-1). The squares are color-coded; red for a paused process, green for a playing process and gray for a finished/dead process. Selected processes are highlighted with a lighter shade of their color and the current process also has a dashed border.

When a single process is selected the local variables are displayed in the *Variable Viewer* and displayed expressions are evaluated. You can make the *Source Code Viewer* jump to the file and line for the current stack frame (if available) by double-clicking on a process.

To copy processes from one group to another, simply click and drag the processes. To delete a process, press the delete key. When modifying groups it is useful to select more than one process by holding down one or more of the following:

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

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

Summary view

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

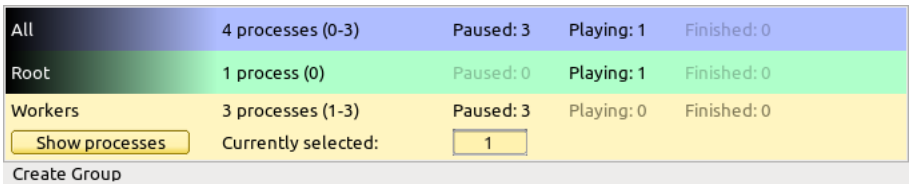

Figure 35: *The Summary Process Group View*

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

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

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

Focus control

The focus control allows you to focus on individual processes or threads as well as process groups. When focused on a particular process or thread, actions such as stepping, playing/pausing, adding breakpoints,

will only apply to that process or thread rather than the entire group.

In addition, the Arm DDT GUI will change depending on whether you are focused on group, process or thread. This allows Arm DDT to display more relevant information about your currently focused object.

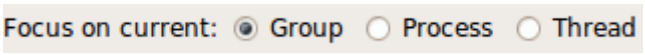

Figure 36: *Focus options*

Overview of changing focus

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

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

Process group viewer

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

Figure 37: *The Detailed Process Group View Focused on a Process*

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

During focus on process, a tooltip is shown that identifies the OpenMP thread ID of each thread, if the value exists.

Breakpoints

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

Code viewer

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

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

Parallel stack view

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

Playing and stepping

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

Step threads together

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

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

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

Stepping threads window

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

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

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

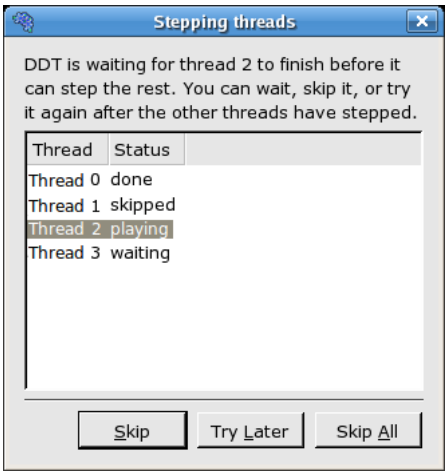

Figure 38: *The Stepping Threads Window*

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

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

The Stepping Threads Window also lets you interact with the threads with the following options:

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

Starting, stopping and restarting a program

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

When a job has run to the end, Arm DDT displays a window box asking if you wish to restart the job. If you select yes then Arm DDT kills 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, Arm DDT attempts to ensure that all the processes are shut down and any temporary files are cleared up. 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 continue the program playing click *Play/Continue* and to stop it at any time click *Pause* .

For multi-process Arm 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. These are enumerated here:

1. *Step Into* moves to the next line of source code unless there is a function call in which case it steps to the first line of that function.
2. *Step Over* moves to the next line of source code in the bottom stack frame.
3. *Step Out* executes the rest of the function and then stop on the next line in the stack frame above. The return value of the function is displayed in the *Locals* view. When using *Step Out* be careful not to try and step out of the main function, as doing this ends your program.

Stop messages

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

1. It hit one of Arm DDT's default breakpoints, for example, `exit` or `abort`. See section 7.10 Default breakpoints for more information on default breakpoints.
2. It hit a user-defined breakpoint, that is a breakpoint shown in the *Breakpoints* view.
3. The value of a watched variable changed.
4. It was sent a signal. See section 7.20 Signal handling for more information on signals.
5. It encountered a Memory Debugging error. See section 12.4 Pointer error detection and validity checking for more information on Memory Debugging errors.

Arm DDT displays a message telling you exactly why the program was paused. To copy the message text to the clipboard select it with the mouse cursor, then right-click and select *Copy*.

You may want to suppress these messages in certain circumstances, for example if you are playing from one breakpoint to another. Use the *Control* → *Messages* menu to enable or disable stop messages.

Setting breakpoints

Using the source code viewer

First locate the position in your code where you want to place a breakpoint. If you have numerous source code files and wish to search for a particular function you can use the *Find/Find In Files* window.

Right-clicking in the *Source Code Viewer* displays a menu showing several options, including one to add or remove a breakpoint.

In multi-process mode this sets the breakpoint for every member of the current group. Breakpoints may also be added by left-clicking the margin to the left of the line number.

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

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

Using the Add Breakpoint window

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

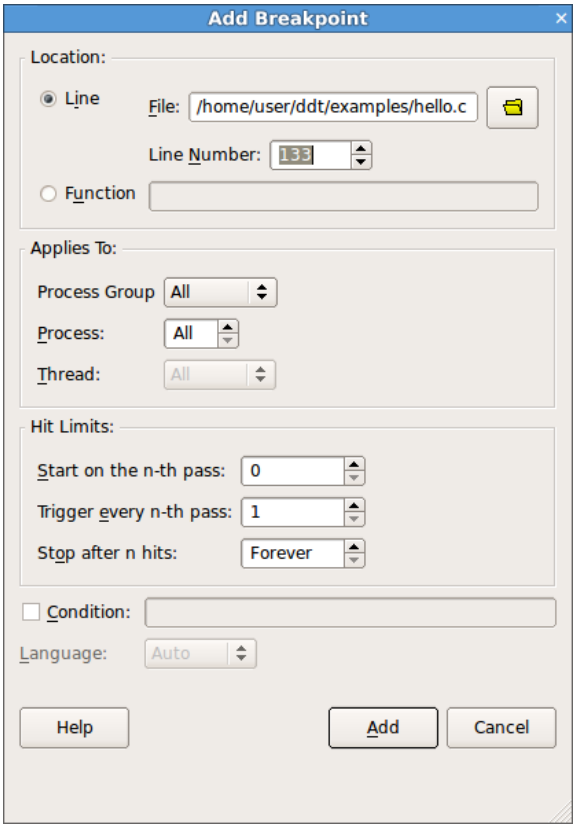

Figure 39: *The Add Breakpoint window*

You may wish to add a breakpoint in a function for which you do not have any source code: for example in `malloc`, `exit`, or `printf` from the standard system libraries. Select the *Function* radio button and enter the name of the function in the box next to it.

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

Pending breakpoints

Note: This feature is not supported on all platforms.

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

Conditional breakpoints

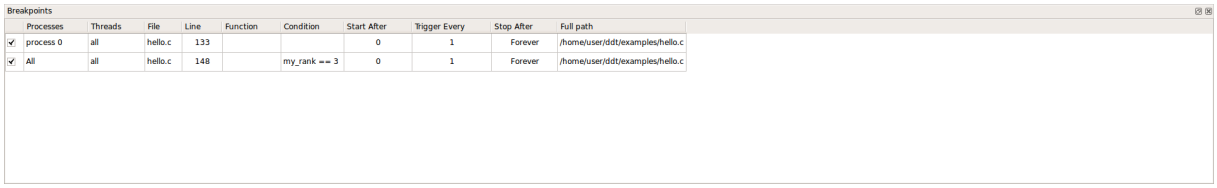

Figure 40: *The Breakpoints Table*

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 evaluates the condition and breaks 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 is set as the condition automatically.

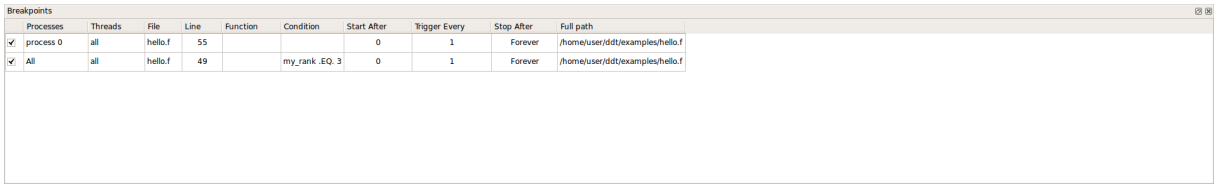

Figure 41: *Conditional Breakpoints In Fortran*

Conditions may be any valid expression for the language of the file containing the breakpoint. This includes other variables in your program and function calls.

You may want to avoid using functions with side effects as these will be executed every time the breakpoint is reached.

The expression evaluation may be more pedantic than your compiler. To ensure the correct interpretation of, for example, boolean operations, it is advisable to use brackets explicitly, to ensure correct evaluation.

Suspending breakpoints

To deactivate or reactivate a breakpoint, either:

- Check or clear the activated column in the breakpoints panel.
- Right-click the breakpoint icon in the code editor and choose **Enable**/**Disable**.
- Hold SHIFT and select a breakpoint icon in the code editor.

Breakpoints that are disabled are grayed out.

Deleting a breakpoint

Breakpoints may be deleted by right-clicking on the breakpoint in the breakpoints panel.

They can also be deleted by right-clicking in the file/line of the breakpoint, while in the correct process group, and right-clicking and selecting delete breakpoint.

They may also be deleted by left-clicking the breakpoint icon in the margin, situated to the left of the line number in the code viewer.

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 are also loaded and saved as part of the load/save session.

Default breakpoints

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

- *Stop at exit/_exit*

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

- *Stop at abort/fatal MPI Error*

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

- *Stop on throw (C++ exceptions)*

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

- *Stop on catch (C++ exceptions)*

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

- *Stop at fork*

  Arm DDT stops whenever your program forks (that is, calls the fork system call to create a copy of the current process). The new process is added to your existing Arm DDT session and can be debugged along with the original process.

- *Stop at exec*

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

- *Stop on CUDA kernel launch*

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

Synchronizing processes

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

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

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

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

Setting a watchpoint

Figure 42: *The Watchpoints Table*

A watchpoint is a variable or expression that will be monitored by the debugger such that when it is changed or accessed the debugger pauses the application.

Figure 43: *Program Stopped At Watchpoint being watched*

Unlike breakpoints, watchpoints are not displayed in the *Source Code Viewer*. Instead they are created by right-clicking on the *Watchpoints* view and selecting the *Add Watchpoint* menu item.

It is also possible to add watchpoints automatically dragging a variable to the *Watchpoints* view from the *Local Variables*, *Current Line* and *Evaluate* views, or right-clicking over the variable in the *Source Code Viewer* and then selecting *Add Watchpoint*.

The automatic watchpoints are write-only by default.

Upon adding a watchpoint the *Add Watchpoint* dialog appears allowing you to apply restrictions to the watchpoint:

- *Process Group* restricts the watch point to the chosen process group (see 7.1 Process control and process groups).
- *Process* restricts the watchpoint to the chosen process.
- *Expression* is the variable name in the program to be watched.
- *Language* is the language of the portion of the program containing the expression.
- *Trigger On* allows you to select whether the watchpoint will trigger when the expression is read, written or both.

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

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

Modern processors have hardware support for a handful of watchpoints that are set to watch the contents of a memory location. Consequently, watchpoints can normally be used with no performance penalty.

Where the number of watchpoints used is over this quantity, or the expression being watched is too complex to tie to a fixed memory address, the implementation is through software monitoring, which imposes significant performance slowdown on the application being debugged.

The number of hardware watchpoints available depends on the system. The read watchpoints are only available as hardware watchpoints.

Consequently, watchpoints should, where possible, be a single value that is stored in a single memory location. While it is possible to watch the whole contents of non-trivial user defined structures or an entire array simultaneously, or complex statements involving multiple addresses, these can cause extreme application slow down during debugging.

Tracepoints

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

Examples of situations in which this feature will prove invaluable include:

- Recording entry values in a function that is called many times, but crashes only occasionally. Setting a tracepoint makes it easier to correlate the circumstances that cause a crash.
- Recording entry to multiple functions in a library, enabling the user or library developer to check which functions are being called, and in which order. An example of this is the MPI History Plugin, which records MPI usage. See section 13.3 Using a plugin.
- Observing progress of an application and variation of values across processes without having to interrupt the application.

Setting a tracepoint

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

Tracepoints can lead to considerable resource consumption by the user interface if placed in areas likely to generate a lot of passing. For example, if a tracepoint is placed inside of a loop with N iterations, then N separate tracepoint passings will be recorded.

While Arm DDT attempts to merge such data in a scalable manner, when alike tracepoints are passed in order between processes, where process behavior is likely to be divergent and unmergeable then a considerable load would result.

If it is necessary to place a tracepoint inside a loop, set a condition on the tracepoint to ensure you only log what is of use to you. Conditions may be any valid expression in the language of the file the tracepoint is placed in and may include function calls, although you may want to be careful to avoid functions with side effects as these will be evaluated every time the tracepoint is reached.

Tracepoints also momentarily stop processes at the tracepoint location in order to evaluate the expressions and record their values. This means if they are placed inside (for example) a loop with a very large number of iterations, or a function executed many times per second, then a slowdown in your application will be noticed.

Tracepoint output

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

Tracepoint	Processes	Values logged
subdomain.f:...	1, rank 0	jend: 0 ny: 9
blts.f:74	16, ranks 0-15	jend: 8 ldmx: 9 j: 9 ldmy: 9 jst: 1-2 ldmz: 33
blts.f:74	16, ranks 0-15	jend: 8 ldmx: 9 j: 9 ldmy: 9 jst: 1-2 ldmz: 33
blts.f:74	16, ranks 0-15	jend: 8 ldmx: 9 j: 9 ldmy: 9 jst: 1-2 ldmz: 33
blts.f:74	16, ranks 0-15	jend: 8 ldmx: 9 j: 9 ldmy: 9 jst: 1-2 ldmz: 33

Figure 44: *Output from Tracepoints in a Fortran application*

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

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

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

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

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

Version control breakpoints and tracepoints

Version control breakpoint/tracepoint insertion allows you to quickly record the state of the parts of the target program that were last modified in a particular revision. The resulting tracepoint output may be viewed in the *Tracepoint Output* tab or the *Logbook* tab and may be exported or saved as part of a logbook or offline log.

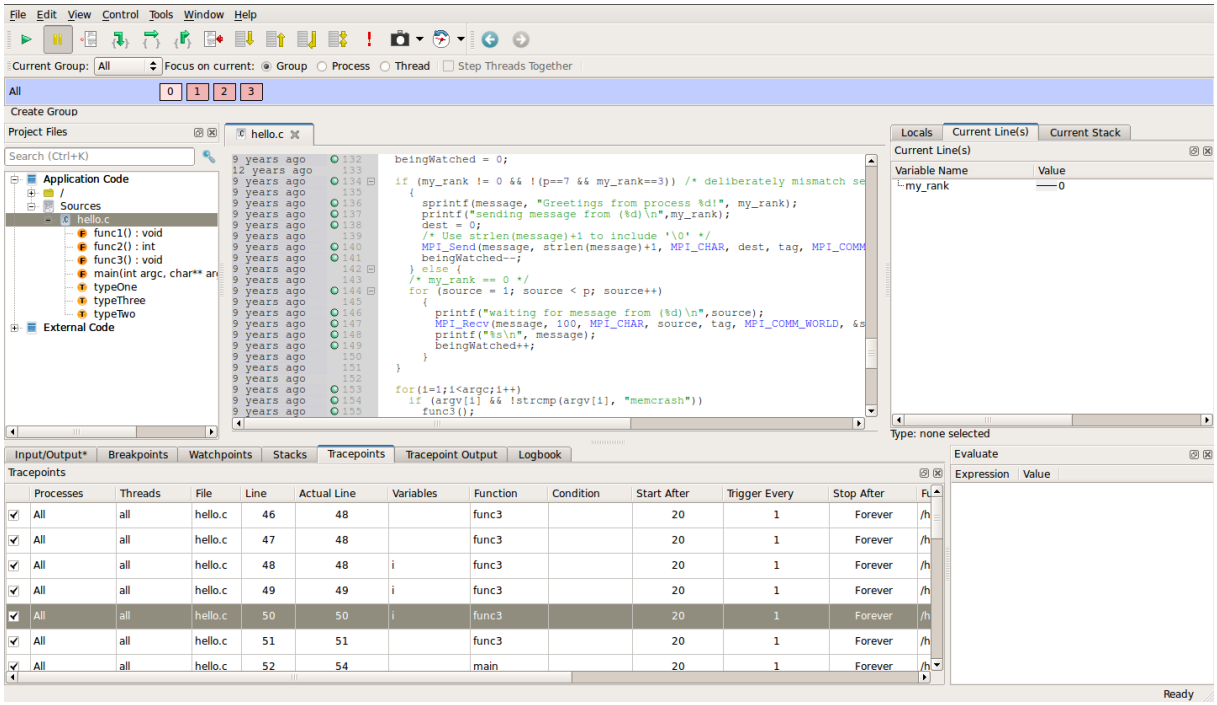

Figure 45: *DDT with version control tracepoints*

Version control tracepoints may be inserted either in the graphical interactive mode or in offline mode via a command line argument.

In interactive mode enable *Version Control Information* from the *View* menu and wait for the annotation column to appear in the code editor. This does not appear for files that are not tracked by a supported version control system.

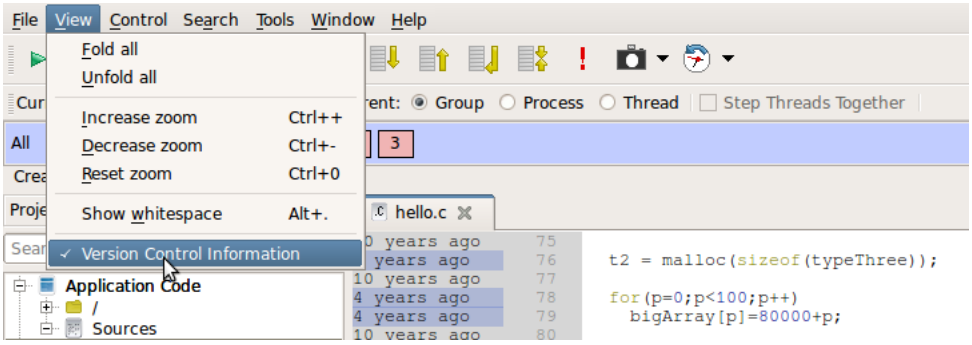

Figure 46: *Version Control—Enable from Menu*

Right-click a line last modified by the revision of interest and choose *Trace Variables At This Revision*.

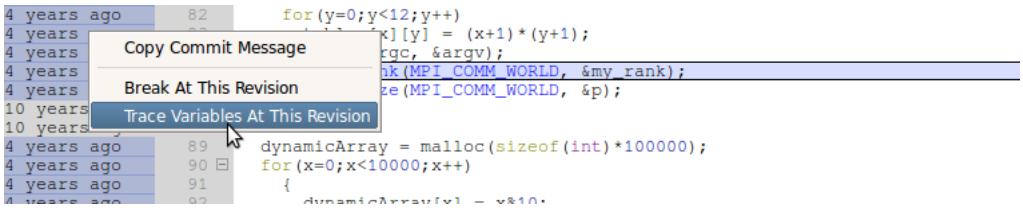

Figure 47: *Version Control—Trace at this revision*

Arm DDT will find all the source files modified in the revision, detect the variables on the lines modified in the revision and insert tracepoints (pending if necessary). A progress dialog may be shown for lengthy tasks.

Both the tracepoints and the tracepoint output in the *Tracepoints*, *Tracepoint Output*, and *Logbook* tabs may be double-clicked during a session to jump to the corresponding line of source in the code viewer.

In offline mode supply the additional argument `--trace-changes` and Arm DDT applies the same process as in interactive mode using the current revision of the repository.

By default version control tracepoints are removed after 20 hits. To change this hit limit set the environment variable `ALLINEA_VCS_TRACEPOINT_HIT_LIMIT` to an integer greater than or equal to `0`. To configure version control tracepoints to have no hit limit set this to `0`.

See also Version control information.

Examining the stack frame

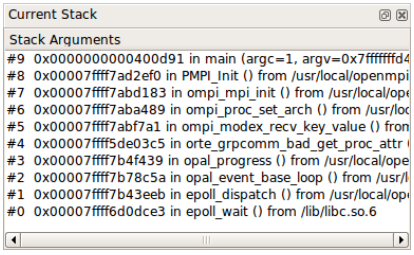

Figure 48: *The Stack Tab*

The stack back trace for the current process and thread are displayed under the *Stack* tab of the *Variables Window*. When you select a stack frame Arm DDT jumps 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 as the current process, where it is possible to do so.

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.

Viewing stacks in parallel

Overview

To find out where your program is, in one single view, you can use the *Parallel Stack View*. It is found in the bottom area of Arm DDT's GUI, tabbed alongside *Input/Output*, *Breakpoints* and *Watches*:

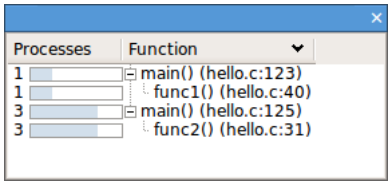

Figure 49: *DDT Parallel Stack View*

If you want to know where a group's processes are, click on the group and look at the *Parallel Stack View*. This shows a tree of functions, merged from every process in the group (by default). If there is only one branch in this tree, one list of functions, then all your processes are at the same place.

If there are several different branches, then your group has split up and is in different parts of the code. Click on any branch to see its location in the *Source Code Viewer*, or hover your mouse over it and a little popup will list the processes at that location. Right-click on any function in the list and select *New Group* to automatically gather the processes at that function together in a new group, labelled by the function's own name.

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

The Parallel Stack View in detail

The *Parallel Stack View* takes over much of the work of the *Stack* display, but instead of just showing the current process, this view combines the call trees (commonly called *stacks*) from many processes and displays them together. The call tree of a process is the list of functions (strictly speaking *frames* or locations within a function) that lead to the current position in the source code.

For example, if `main()` calls `read_input()`, and `read_input()` calls `open_file()`, and you stop the program inside `open_file()`, then the call tree looks like the following:

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

If a function was compiled with debug information (usually `-g`) then Arm DDT adds extra information, displaying the exact source file and line number that your code is on.

Any functions without debug information are grayed-out and are not shown by default. Functions without debug information are typically library calls or memory allocation subroutines and are not generally of interest. To see the entire list of functions, right-click on one and choose *Show Children* from the pop-up menu.

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

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

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

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

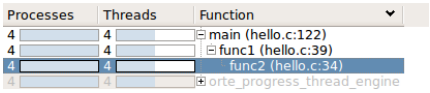

Figure 50: *Current Frame Highlighting in Parallel Stack View*

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

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

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

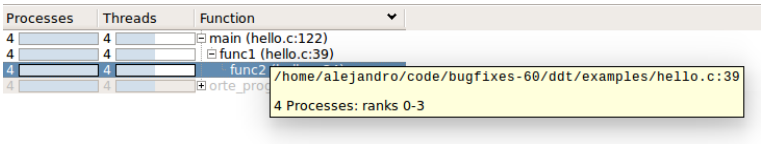

Figure 51: *Parallel Stack View tool tip*

Arm DDT is at its most intuitive when each process group is a collection of processes doing a similar task. The *Parallel Stack View* is invaluable in creating and managing these groups.

Right-click on any function in the combined call tree and choose the *New Group* option. This creates a new process group that contains only the processes sharing that location in code. By default Arm DDT uses the name of the function for the group, or the name of the function with the file and line number if it is necessary to distinguish the group further.

Browsing source code

Source code is automatically displayed when a process is stopped, when you select a process, or position in the stack changed. If the source file cannot be found you are prompted for its location.

Arm DDT highlights lines of the source code to show the current location of your program's execution. Lines that contain processes from the current group are shaded in that group's color. Lines only containing processes from other groups are shaded in gray.

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

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

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

Focus on Group

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

Focus on Process

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

Focus on Thread

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

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

Variables and Functions

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

In the case of a variable, the type and value are displayed, along with options to view the variable in the *Cross-Process Comparison Window* (CPC) or the *Multi-Dimensional Array Viewer* (MDA), or to drop the variable into the *Evaluate Window*, each of which are described in the next chapter.

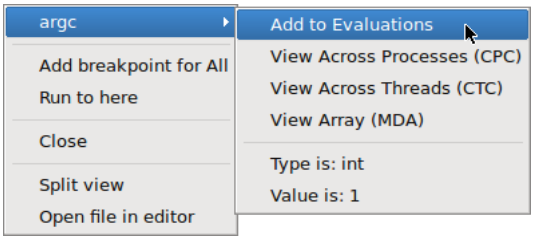

Figure 52: *Right-Click Menu—Variable Options*

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.

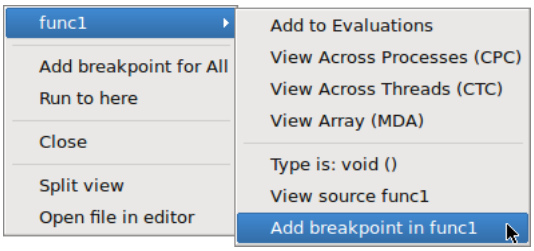

Figure 53: *Right-Click Menu—Function Options*

Simultaneously viewing multiple files

Arm DDT presents a tabbed pane view of source files. Occasionally it may be useful to view two files simultaneously, such as when tracking two different processes.

Inside the code viewing panel, right-click to split the view. This displays a second tabbed pane which can be viewed beneath the first one. When viewing additional files, the currently 'active' panel displays 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.

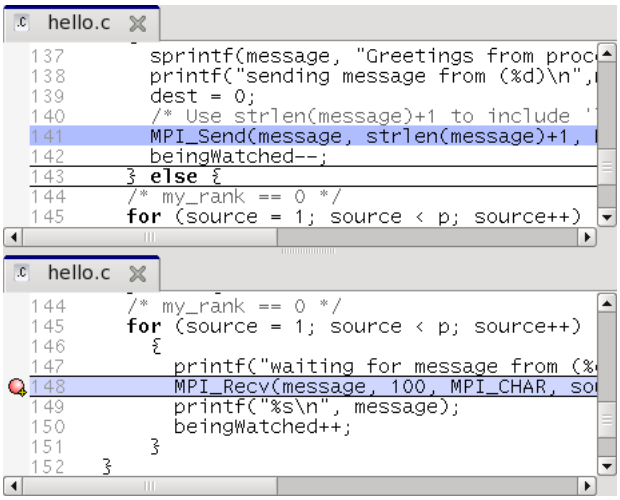

Figure 54: *Horizontal Alignment Of Multiple Source Files*

Signal handling

By default Arm DDT will stop a process if it encounters one of the standard signals. See section 7.20.1 Custom signal handling (signal dispositions). The standard signals include:

- `SIGSEGV` – Segmentation fault

  The process has attempted to access memory that is not valid for that process. Often this will be caused by reading beyond the bounds of an array, or from a pointer that has not been allocated yet. The DDT *Memory Debugging* feature may help to resolve this problem.

- `SIGFPE` – Floating Point Exception

  This is raised typically for integer division by zero, or dividing the most negative number by -1. Whether or not this occurs is Operating System dependent, and not part of the POSIX standard.

Linux platforms will raise this.

Note that floating point division by zero will not necessarily cause this exception to be raised, behavior is compiler dependent. The special value `Inf` or `-Inf` may be generated for the data, and the process would not be stopped.

- `SIGPIPE` – Broken Pipe

  A broken pipe has been detected while writing.

- `SIGILL` – Illegal Instruction

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

Custom signal handling (signal dispositions)

You can change the way individual signals are handled using the *Signal Handling* window. To open the window select the *Control* → *Signal Handling…* menu item.

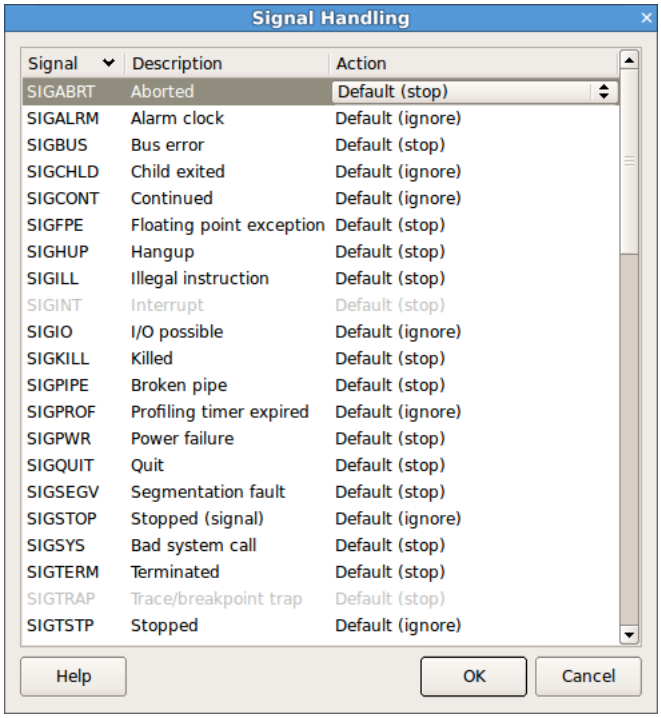

Figure 55: *Signal Handling dialog*

Set a signal's action to *Stop* to stop a process whenever it encounters the given signal, or *Ignore* to let the process receive the signal and continue playing without being stopped by the debugger.

Sending signals

The *Send Signal* window allows a signal to be sent to the debugged processes. Select the *Control* → *Send Signal…* menu item. Select the signal you want to send from the drop-down list and click the *Send to process* button.

Viewing variables and data

The *Variables Window* contains two tabs that provide different ways to list your variables. The *Locals* tab contains all the variables for the current stack frame, while the *Current Line(s)* tab displays all the variables referenced on the currently selected lines. Please note that several compilers and libraries (such as Cray Fortran, OpenMP and others) generate extra code, including variables that are visible in Arm DDT's windows.

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

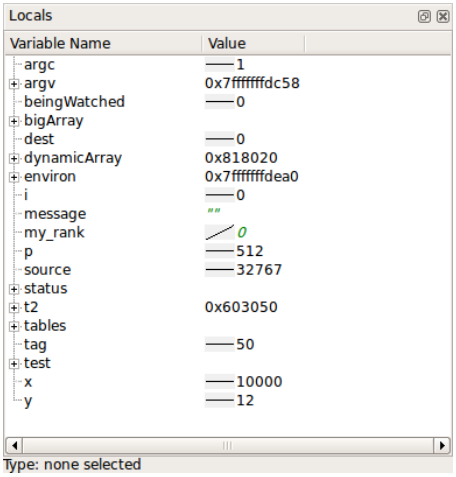

Figure 56: *Displaying Variables*

Sparklines

Numerical values may have sparklines displayed next to them. A sparkline is a line graph of process rank or thread index against value of the related expression. The exact behavior is determined by the focus control. See section 7.2 Focus control.

If focussed on process groups, then process ranks are used. Otherwise, thread indices are used. The graph is bound by the minimum and maximum values found, or in the case that all values are equal the line is drawn across the vertical center of the highlighted region. Erroneous values such as *Nan* and *Inf* are represented as red, vertical bars. If focus is on process groups, then clicking on a sparkline displays the Cross-Process Comparison window for closer analysis. Otherwise, clicking on a sparkline displays the Cross-Thread Comparison window.

Current line

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

Local variables

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

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

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

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

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

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

Arbitrary expressions and global variables

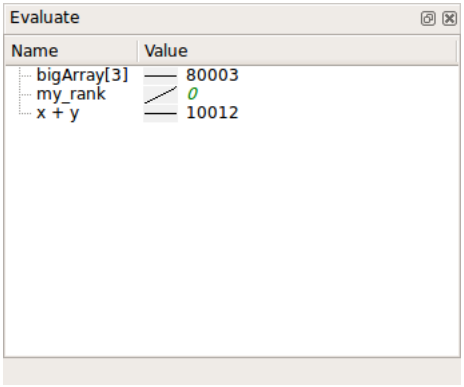

Figure 57: *Evaluating Expressions*

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

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

Note: At the time of writing Arm DDT does not apply the usual rules of precedence to logical Fortran expressions, such as `x .ge. 32 .and. x .le. 45.`*. For now, please bracket such expressions thoroughly:* `(x .ge. 32) .and. (x .le. 45)`*.*

Note: Although the Fortran syntax allows you to use keywords as variable names, Arm DDT is not be able to evaluate such variables on most platforms. Please contact Arm support at Arm supportif this is a problem for you.

Expressions containing function calls are only evaluated for the current process/thread and sparklines are not displayed for those expressions, because of possible side effects caused by calling functions. Use *Cross-Process or Cross-Thread Comparison* for functions instead. See section 8.16 Cross-process and cross-thread comparison.

Fortran intrinsics

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

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

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

Changing the language of an expression

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

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

Macros and #defined constants

By default, many compilers do not output sufficient information to allow the debugger to display the values of "#defined" constants or macros, as including this information can greatly increase executable sizes.

With the GNU compiler, adding the "`-g3`" option to the command line options generates extra definition information which Arm DDT will then be able to display.

Help with Fortran modules

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

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

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

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

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

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

Clicking on one of the displayed procedure names causes the *Source Code Viewer* to jump to that procedure's location in the source code. In addition, the return type of the procedure is displayed at the bottom of the *Fortran Modules* tab. Fortran subroutines will have a return type of `VOID ()`.

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

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

Right-clicking within the *Fortran Modules* tab brings up a context menu. For variables, choices on this menu includes sending the variable to the *Evaluate* window, the *Multi-Dimensional Array Viewer* and the *Cross-Process Comparison Viewer*.

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

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

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

Viewing complex numbers in Fortran

When working with complex numbers, you may wish to view only the real or imaginary elements of the number. This can be useful when evaluating expressions, or viewing an array in the Multi-Dimensional Array Viewer See section 8.15 Multi-dimensional array viewer (MDA).

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

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

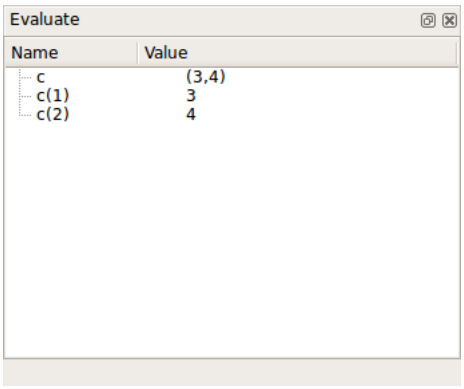

Figure 58: *Viewing the Fortran complex number 3+4i*

C++ STL support

Arm DDT uses pretty printers for the GNU C++ STL implementation (versions 4.7 and greater), Nokia's Qt library, and Boost, designed for use with the GNU Debugger. These are used automatically to present such C++ data in a more understandable format.

For some compilers, the STL pretty printing can be confused by non-standard implementations of STL types used by a compiler's own STL implementation. In this case, and in the case where you wish to see the underlying implementation of an STL type, you can disable pretty printing by running DDT with the environment variable setting ALLINEA_DISABLE_PRETTY_PRINT=1.

Expanding elements in `std::map`, including `unordered` and `multimap` variants, is not supported when using object keys or pointer values.

Custom pretty printers

In addition to the pre-installed pretty printers you may also use your own GDB pretty printers.

*Note: custom pretty printers are only supported when using the GDB 7.6.2 debugger. You must select this debugger on the **System Settings** page of the **Options** window.*

A GDB pretty printer consists of an `auto-load` script that is automatically loaded when a particular executable or shared object is loaded and the actual pretty printer Python classes themselves. To make a pretty printer available in DDT copy it to `~/.allinea/gdb`.

Example

An example pretty printer may be found in {installation-directory}/examples.

Compile the `fruit` example program using the GNU C++ compiler as follows:

```
cd {installation-directory}/examples
make -f fruit.makefile
```

Now start Arm DDT with the example program as follows:

```
ddt --start {installation-directory}/examples/fruit
```

After the program has started right-click on line 20 and click the *Run to here* menu item. Click on the *Locals* tab and notice that the internal variable of `myFruit` are displayed.

Now install the fruit pretty printer by copying the files to `~/.allinea/gdb` as follows:

```
cp -r {installation-directory}/examples/fruit-pretty-printer/* ~/.
   allinea/gdb/
```

Re-run the program in Arm DDT and run to line 20, as before. Click on the *Locals* tab and notice that now, instead of the internal variable of `myFruit`, the type of fruit is displayed instead.

Viewing array data

Fortran users may find that it is not possible to view the upper bounds of an array. This is due to a lack of information from the compiler. In these circumstances Arm DDT displays the array with a size of 0, or simply `<unknown_bounds>`. It is still possible to view the contents of the array using the Evaluate window to view `array(1)`, `array(2)`, and so on, as separate entries.

To tell Arm DDT the size of the array right-click on the array and select the *Edit Type…* menu option. This opens a window similar to the one below. Enter the real type of the array in the *New Type* box.

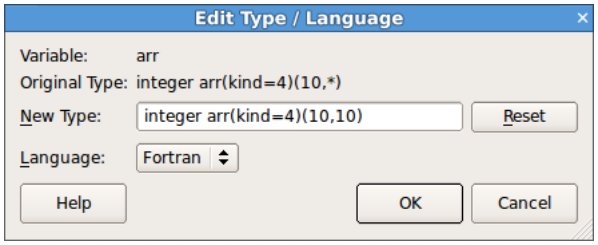

Figure 59: *Edit Type window*

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

UPC support

Arm DDT supports many different UPC compilers, including the GNU UPC compiler, the Berkeley UPC compiler and those provided by Cray.

Note: In order to enable UPC support, you may need to select the appropriate MPI/UPC implementation from DDT's Options/System menu. See Section 5.14 UPC

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

- Processes will be identified as UPC Threads, this is purely a terminology change for consistency with the UPC language terminology. UPC Threads will have behavior identical to that of separate processes: groups, process control and cross-process data comparison for example will apply across UPC Threads.
- The type qualifier `shared` is given for shared arrays or pointers to shared.
- Shared pointers are printed as a triple (address, thread, phase). For indefinitely blocked pointers the phase is omitted.
- Referencing shared items will yield a shared pointer and pointer arithmetic may be performed on shared pointers.
- Dereferencing a shared pointer (for example, dereferencing `*(&x[n] + 1])`) will correctly evaluate and fetch remote data where required.

- Values in shared arrays are not automatically compared across processes: the value of `x[i]` is by definition identical across all processes. It is not possible to identify pending read/write to remote data. Non-shared data types such as local data or local array elements will still be compared automatically.
- Distributed arrays are handled implicitly by the debugger. There is no need to use the explicit *distributed dimensions* feature in the MDA.

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

Changing data values

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

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

Viewing numbers in different bases

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

Examining pointers

You can examine pointer contents by clicking the `+` next to the variable or expression. This expands the item and dereference the pointer.

In the *Evaluate* window, you can also use the *View As Vector*, *Reference*, and *Dereference* menu items.

See also Multi-dimensional array viewer (MDA).

Multi-dimensional arrays in the Variable View

When viewing a multi-dimensional array in either the *Locals*, *Current Line(s)* or *Evaluate* windows it is possible to expand the array to view the contents of each cell.

In C/C++ the array expands from left to right, `x`, `y`, `z` will be seen with the x column first, then under each `x` cell a `y` column, whereas in Fortran the opposite will be seen with arrays being displayed from right to left as you read it so `x`, `y`, `z` would have `z` as the first column with `y` under each `z` cell.

The first thousand elements in an array are shown in the *Locals* or *Current Line(s)* view. Larger arrays are truncated, but elements after the first thousand can be viewed by evaluating an expression or using the multi-dimensional array viewer.

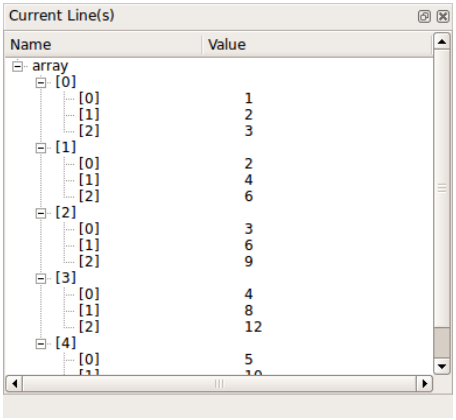

Figure 60: *2D Array in C: type of array is int[4][3]*

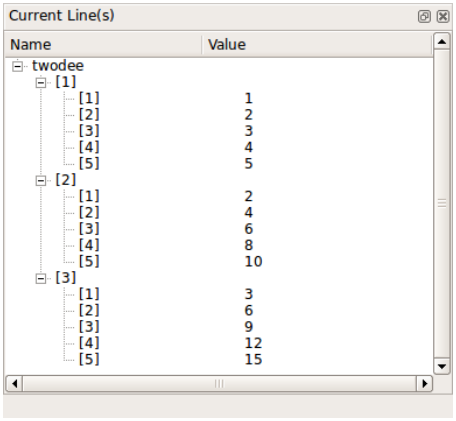

Figure 61: *2D Array in Fortran: type of twodee is integer(3,5)*

Multi-dimensional array viewer (MDA)

Arm DDT provides a *Multi-Dimensional Array (MDA) Viewer* (fig. 62) for viewing multi-dimensional arrays.

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

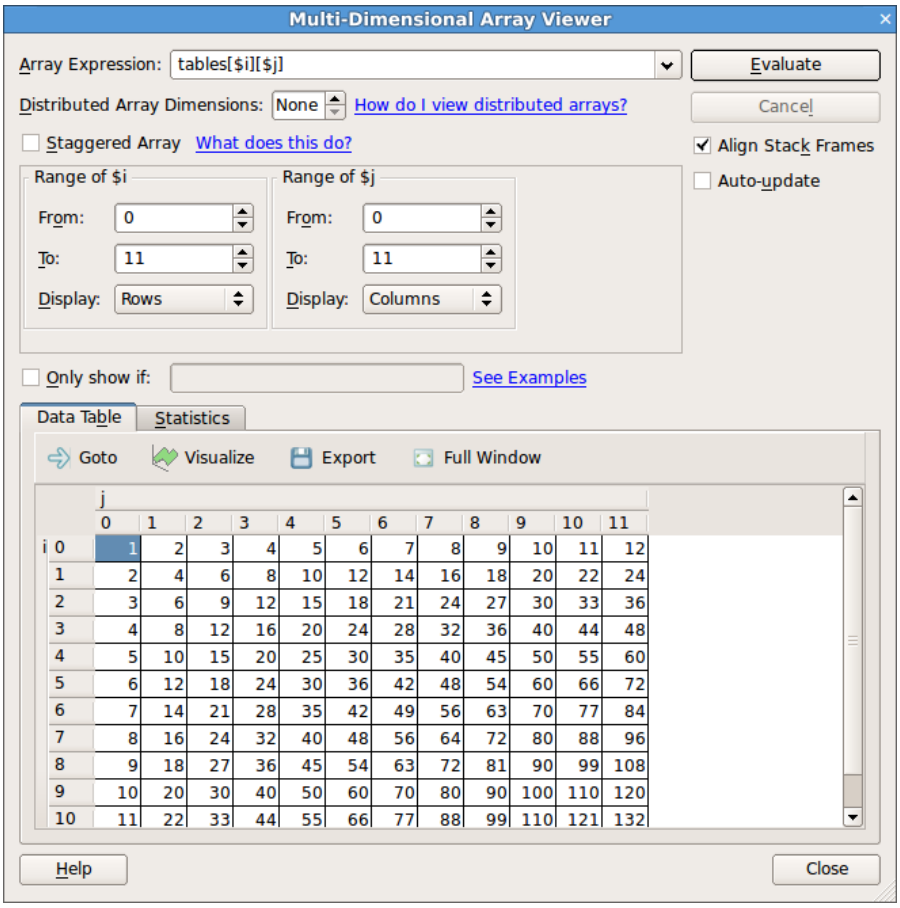

Figure 62: *Multi-Dimensional Array Viewer*

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

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

Array expression

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

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

By default, the ranges for these metavariables are integer constants entered using spin boxes. However, the MDA also supports specifying these ranges as *expressions* in terms of program variables. These expressions are then evaluated in the debugger. To allow the entry of these expressions, check the *Staggered Array* check box. This will convert all the range entry fields from spin boxes to line edits allowing the entry of freeform text.

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

Filtering by value

You may want the *Data Table* to only show elements that fit a certain criteria, for example elements that are zero.

If the *Only show if* box is checked then only elements that match the boolean expression in the box are displayed in the *Data Table*, for example, `$value == 0`. The special metavariable `$value` in the expression is replaced by the actual value of each element. The *Data Table* automatically hides rows or columns in the table where no elements match the expression.

Any valid expression for the current language may be used here, including references to variables in scope and function calls. You may want to be careful to avoid functions with side effects as these will be evaluated many times over.

Distributed arrays

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

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

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

To view a non-UPC distributed array first create a process group containing all the processes that the array is distributed over.

If the array is distributed over all processes in your job then you can simply select the *All* group instead. Right-click on the local array variable in the *Source Code*, *Locals*, *Current Line(s)* or *Evaluate* views.

The *Multi-Dimensional Array Viewer* window will open with the Array Expression already filled in.

Enter the number of distributed array dimensions in the corresponding box. A new subscript metavariable (such as `$p`, `$q`) will be automatically added for each distributed dimension.

Enter the ranges of the distributed dimensions so that the product is equal to the number of processes in the current process group, then click the *Evaluate* button.

Advanced: how arrays are laid out in the data table

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

Each subscript metavariable (such as `$i`, `$j`, `$p`, `$q`) maps to a separate dimension on a hypercube. Usually the number of metavariables is equal to the number of dimensions in a given array, but this

does not necessarily need to be the case. For example `myArray($i, $j) * $k` introduces an extra dimension, `$k`, as well as the two dimensions corresponding to the two dimensions of `myArray`.

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

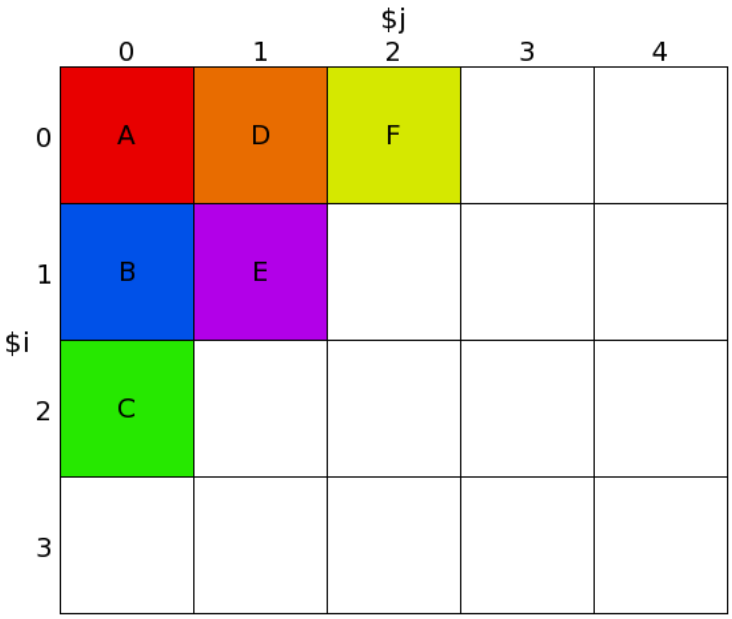

Figure 63: *myArray($i, $j) with $i = 0..3 and $j = 0..4.*

If, by way of example, imagine that `myArray` is part of a three dimensional array distributed across three processes. The figure below shows what the local arrays look like for each process.

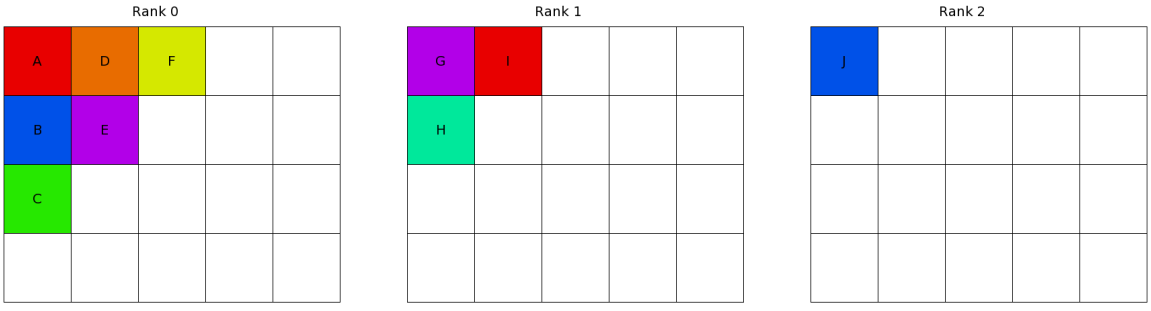

Figure 64: *The local array myArray($i, $j) with $i = 0..3 and $j = 0..4 on ranks 0–2*

And as a three dimensional distributed array with `$p` the distributed dimension:

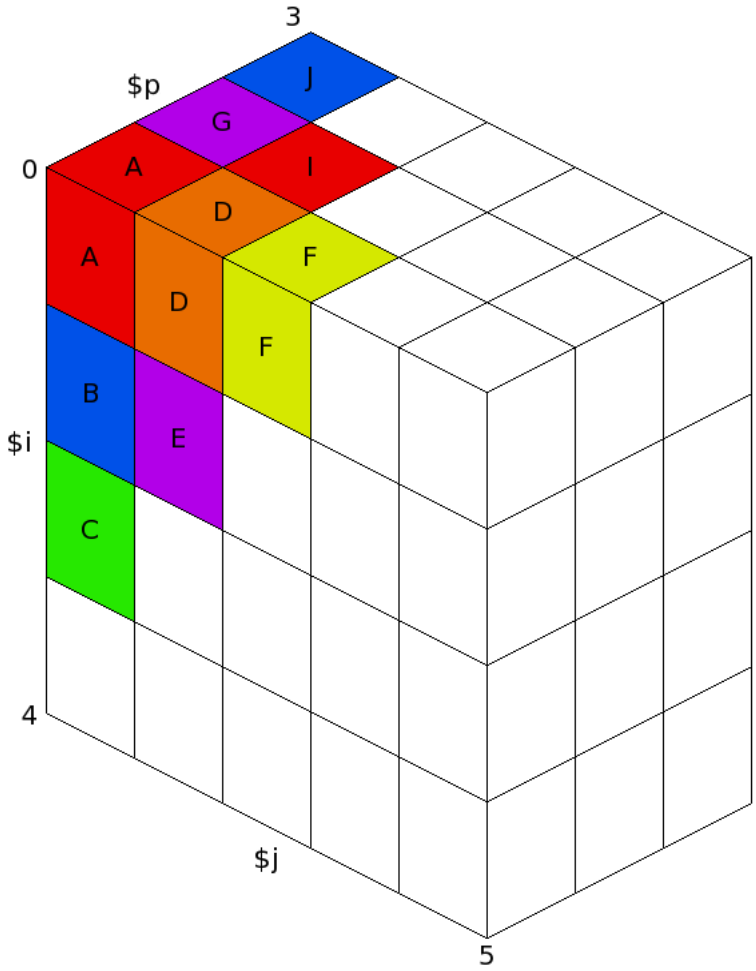

Figure 65: *A three dimensional distributed array comprised of the local array myArray($i, $j) with $i = 0..3 and $j = 0..4 on ranks 0–2 with $p the distributed dimension*

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

More than one dimension may viewed as Rows, or more than one dimension viewed as Columns.

The dimension that changes fastest depends on the language your program is written in. For C/C++ programs the leftmost metavariable (usually `$i` for local arrays or `$p` for distributed arrays) changes the most slowly (just like with C array subscripts). The rightmost dimension changes the most quickly. For Fortran programs the order is reversed, that is the rightmost is most major, the leftmost most minor.

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

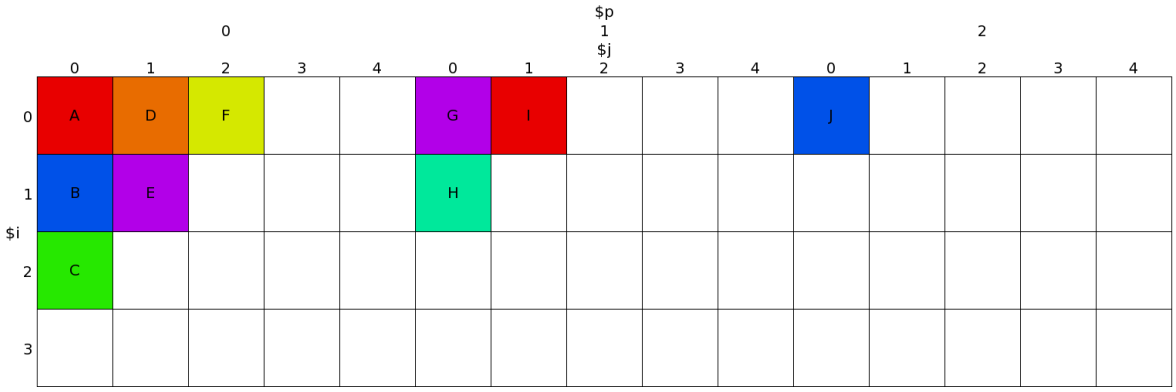

Figure 66: *A three dimensional distributed array comprised of the local array myArray($i, $j) with $i = 0..3 and $j = 0..4 on ranks 0–2 projected onto the Data Table with $p (the distributed dimension) and $j displayed as Columns and $i displayed as Rows.*

Auto Update

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

Comparing elements across processes

When viewing an array in the *Data Table*, you may double-click or choose *Compare Element Across Processes* from the context menu for a particular element.

This displays the *Cross-Process Comparison* dialog for the specified element.

See 8.16 Cross-process and cross-thread comparison for more information.

Statistics

The *Statistics* tab displays information which may be of interest, such as the range of the values in the table, and the number of special numerical values, such as `nan` or `inf`.

Export

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

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

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

Visualization

If your system is OpenGL-capable then a 2-D slice of an array, or table of expressions, may be displayed as a surface in 3-D space through the *Multi-Dimensional Array (MDA) Viewer*.

You can only plot one or two dimensions at a time. If your table has more than two dimensions the *Visualise* button will be disabled.

After filling the table of the *MDA Viewer* with values (see previous section), click *Visualise* to open a 3-D view of the surface.

To display surfaces from two or more different processes on the same plot simply select another process in the main process group window and click *Evaluate* in the MDA window, and when the values are ready, click *Visualise* again.

The surfaces displayed on the graph may be hidden and shown using the check boxes on the right-hand side of the window.

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

The mouse controls are:

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

Note: Arm DDT requires OpenGL to run. If your machine does not have hardware OpenGL support, software emulation libraries such as MesaGL are also supported.

Note: In some configurations OpenGL is known to crash. A work-around if the 3D visualization crashes is to set the environment variable LIBGL_ALWAYS_INDIRECT to 1. The precise configuration which triggers this problem is not known.

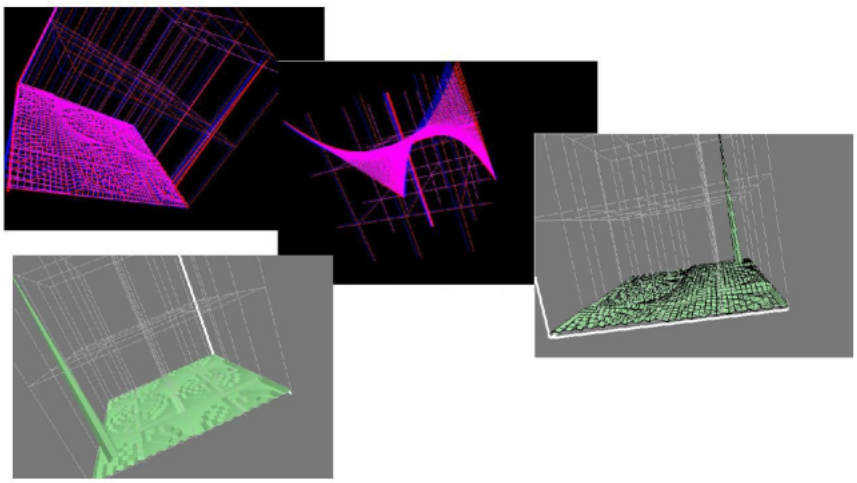

Figure 67: *DDT Visualization*

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 Arm support if you need to obtain some 3D glasses.

Cross-process and cross-thread comparison

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

This is a more detailed view than the sparklines that are automatically drawn against a variable in the evaluations and locals/current line windows for multi-process sessions.

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

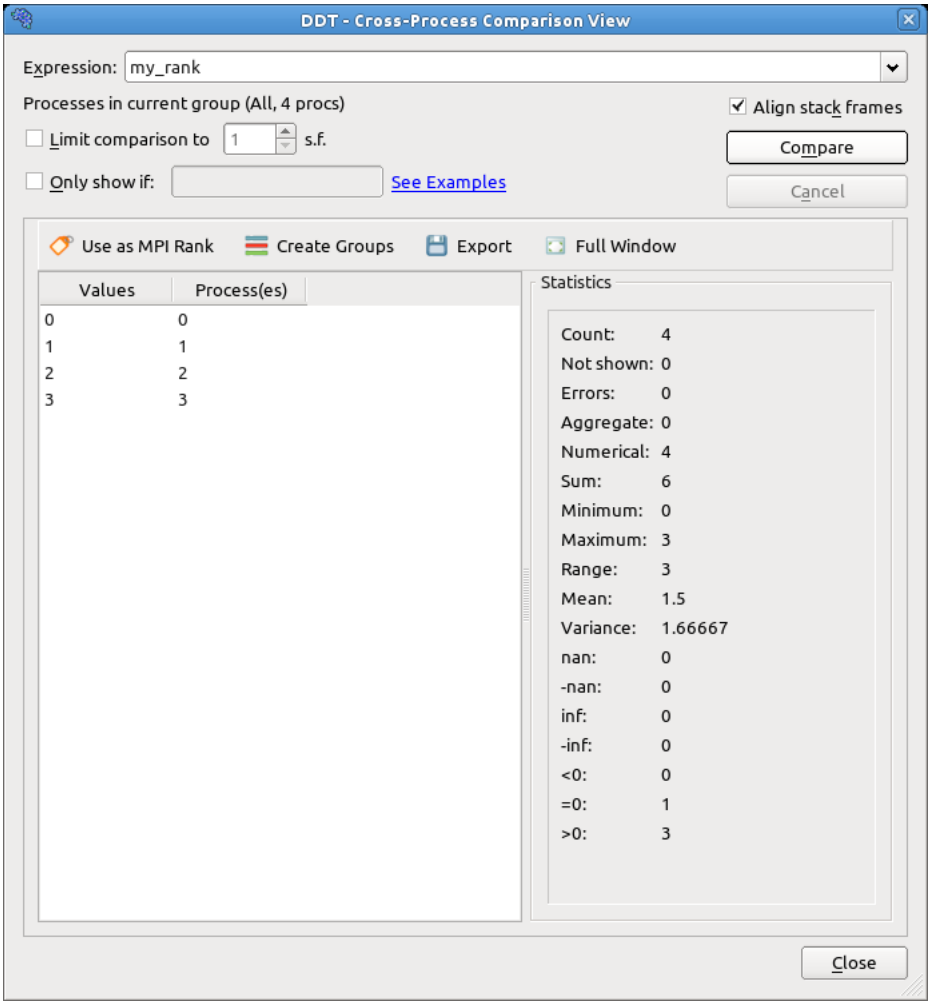

Figure 68: *Cross-Process Comparison—Compare View*

Processes and threads are grouped by expression value when using the raw comparison. The precision of this grouping can be specified (for floating point values) by filling the *Limit* box.

If you are comparing across processes, you can turn each of these groupings of processes into a Arm DDT process group by clicking the create groups button. This creates several process groups, one for each line in the panel. Using this capability large process groups can be managed with simple expressions to create groups. These expressions are any valid expression in the present language (that is, C/C++/Fortran).

For threaded applications, when using the CTC, if Arm DDT is able to identify OpenMP thread IDs, a third column will also display the corresponding OpenMP thread IDs for each thread that has each value. The value displayed in this third column for any non-OpenMP threads that are running depends on your compiler but is typically `-1` or `0`. OpenMP thread IDs should be available when using Intel and PGI compilers provided compiler optimisations have not removed the required information (recompile with `-O0` if necessary). OpenMP thread IDs can only be obtained from GCC compiled programs if the compiler itself was compiled with TLS enabled, unfortunately this is not the case for the packaged GCC installs on any of the major Linux distributions at time of writing (Redhat 7, SUSE 12 or Ubuntu 16.04). The display of OpenMP thread IDs is not currently supported when using the Cray compiler or the IBM XLC/XLF compilers.

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

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

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

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

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

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

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

Assigning MPI ranks

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

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

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

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

These are the only restrictions. As you can see, there is no need to use the MPI rank if you have an alternate numbering scheme that makes more sense in your application. In fact you can relabel only a few of the processes and not all, if you prefer, so long as afterwards **every** process still has a unique number.

Viewing registers

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

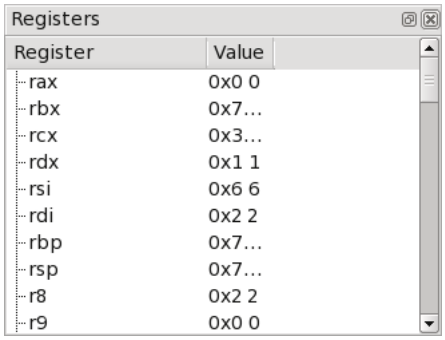

Figure 69: *Register View*

Process details

To view the process details dialog select the *Process Details* menu item from the *Tools* menu. Details can be sorted by any columns, in ascending or descending order.

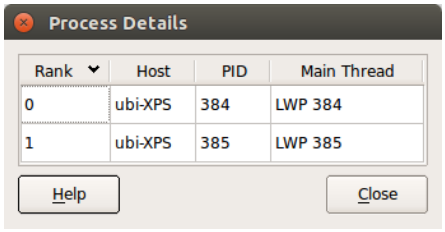

Figure 70: *Process Details*

Disassembler

To view the disassembly (assembly instructions) of a function select the *Disassemble* menu item from the *Tools* menu. By default you will see the disassembly of the current function, but you can view the disassembly of another function by entering the function name in the box at the top and clicking the *Disassemble* button.

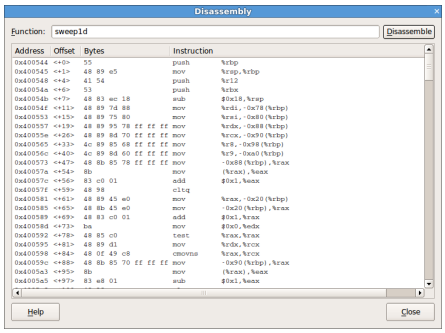

Figure 71: *Disassemble Tool*

Interacting directly with the debugger

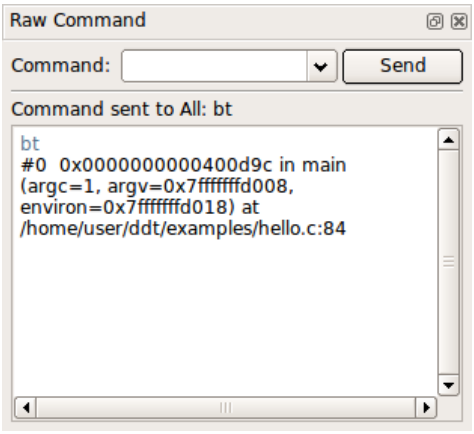

Figure 72: *Raw Command Window*

Arm DDT provides a *Raw Command* window that allows you to send commands directly to the debugger interface. This window bypasses DDT and its book-keeping. If you set a breakpoint here, Arm DDT will not list this in the breakpoint list.

Be careful with this window. It is recommended 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 Arm DDT.

Each command is sent to the current group or process depending on the current focus. If the current group or process is running, Arm DDT prompts you to pause the group or process first.

Program input and output

Arm DDT collects and displays output from all processes under the *Input/Output* tab. Both standard output and error are shown, although on most MPI implementations, error is not buffered but output is and consequently can be delayed.

Viewing standard output and error

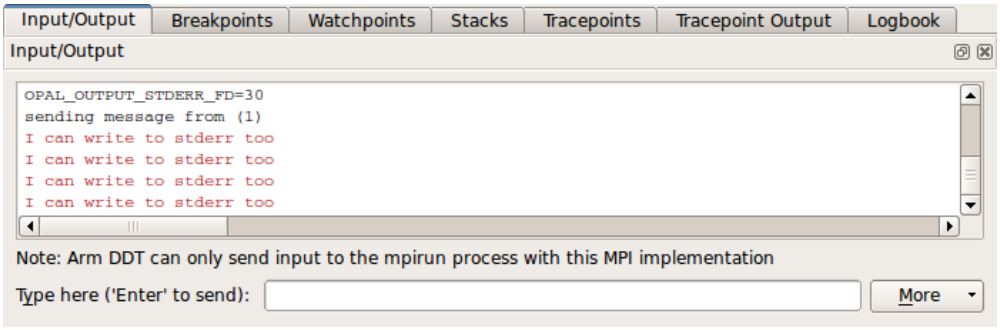

Figure 73: *DDT Standard Output Window*

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

The output may be selected and copied to the 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, and others may ignore it. If your program needs to emit output as it runs, try 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 `fflush(stdout)` or equivalent to your code.

Saving output

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

Sending standard input

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

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

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

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

The input you type will be sent to all processes.

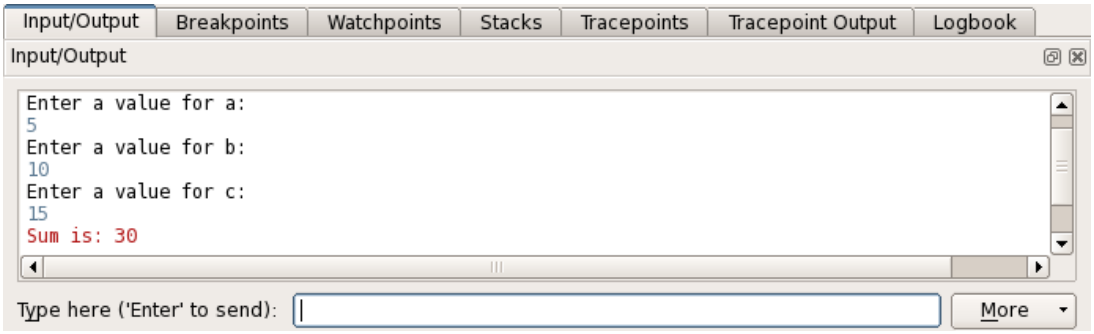

Figure 74: *DDT Sending Input*

Note: If Arm DDT is running on a fork-based system such as Scyld, or a `-comm=shared` *compiled MPICH 1, 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 H General troubleshooting and known issues or contact Arm support at Arm support for a list of possible fixes.*

Logbook

The logbook automatically generates a log of your interaction with Arm DDT, for example, setting a breakpoint or playing the program. For each stop of the program, the reason and location is recorded together with the parallel stacks and local variables for one process.

Tracepoint values and output are logged as well.

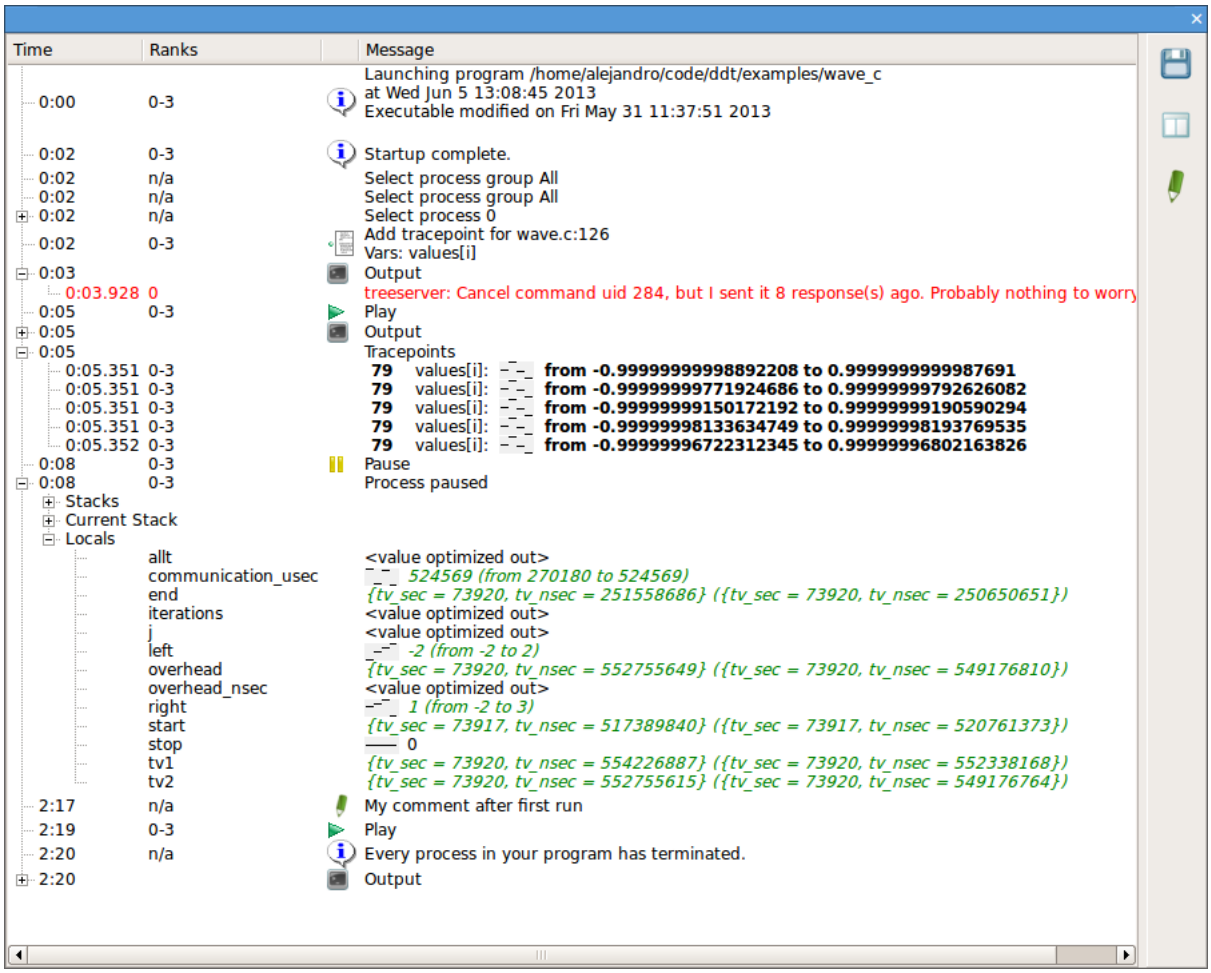

Figure 75: *Logbook example of a debug session*

The user can export the current logbook as HTML or compare it to a previously exported one.

This enables comparative debugging and repeatability. It is always clear how a certain situation in the debugger was caused as the previous steps are visible.

Usage

The logbook is always on and does not require any additional configuration. The Logbook tab is located at the bottom of the main window beside the Tracepoint Output tab.

To export the logbook click the disk icon on the right-hand side of the Logbook View and specify a filename. Open previously saved logbooks from the *Tools* menu.

Annotation

Add annotations to the logbook using either the pencil icon on the right-hand side of the logbook tab or by right-clicking the logbook and choosing *Add annotation*.

Comparison window

Two logbooks can be compared side by side with the Logbook Files Comparison window. To run a comparison, click the ‘compare’ icon on the right-hand side of the Logbook View. Compare the current logbook with another logbook file, or choose two different files to compare.

To easily find differences, align both logbook files to corresponding entries and choose the Lock icon. This fixes the vertical and horizontal scrollbars of the logbooks so that they scroll together.

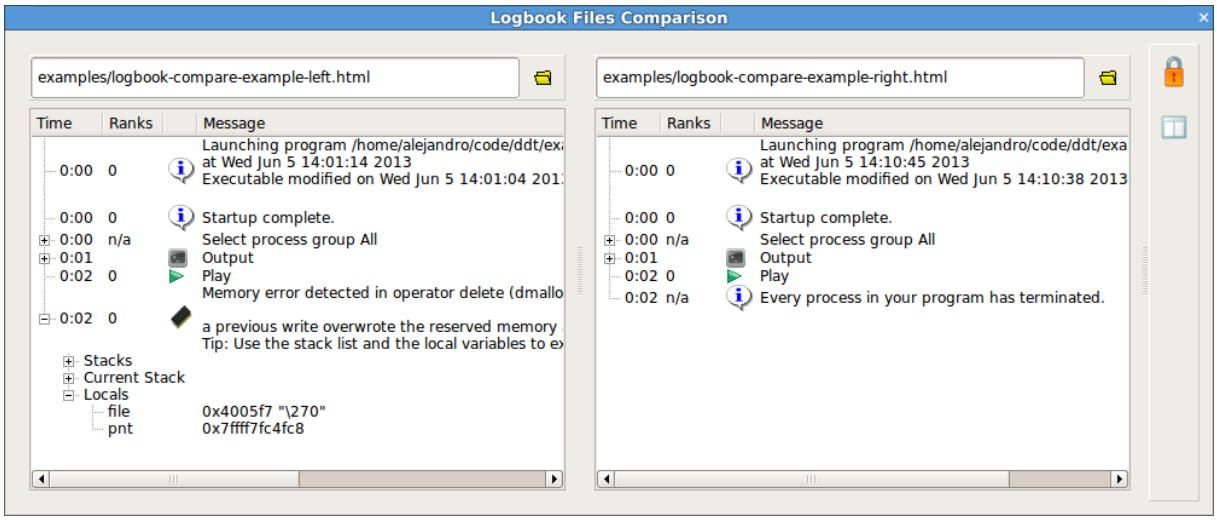

Figure 76: *Logbook comparison window with tracepoint difference selected*

Message queues

Arm DDT's Message Queue debugging feature shows the status of the message buffers of MPI. For example, it shows the messages that have been sent by a process but not yet received by the target.

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

This capability relies on the MPI implementation supporting this via a debugging support library: the majority of MPIs provide this. Furthermore, not all implementations support the capability to the same degree, and a variance between the information provided by each implementation is to be expected.

Viewing the message queues

Open the *Message Queues* window by selecting *Message Queues* from the *Tools* menu. The Message Queues window will query the MPI processes for information about the state of the queues.

While the window is open, click *Update* to refresh the current queue information. Note that this will stop all playing processes. While DDT is gathering the data a "Please Wait" dialog may be displayed and you can cancel the request at any time.

DDT will automatically load the message queue support library from your MPI implementation (provided one exists). If it fails, an error message will be shown. Common reasons for failure to load include:

- The support library does not exist, or its use must be explicitly enabled.

  Most MPIs will build the library by default, without additional configuration flags. MPICH 2 and MPICH 3 must be configured with the `--enable-debuginfo` argument. MPICH 1.2.x must be configured with the `--enable-debug` argument. MVAPICH 2 must be configured with the `--enable-debug` and `--enable-sharedlib` arguments. Some MPIs, notably Cray's MPI, do not support message queue debugging at all.

  Intel MPI includes the library, but debug mode must be enabled. See E.6 Intel MPI for details.

  LAM and Open MPI automatically compile the library.

- The support library is not available on the compute nodes where the MPI processes are running.

  Ensure the library is available, and set the environment variable `ALLINEA_QUEUE_DLL` if necessary to force using the library in its new location.

- The support library has moved from its original installation location.

  Ensure the proper procedure for the MPI configuration is used. This may require you to specify the installation directory as a configuration option.

  Alternatively, you can specifically include the path to the support library in the `LD_LIBRARY_PATH`, or if this is not convenient you can set the environment variable, `ALLINEA_QUEUE_DLL`, to the absolute path of the library itself (for example, `/usr/local/mpich-1.2.7/lib/libtvmpich.so`).

- The MPI is built to a different bit-size to the debugger.

  In the unlikely case that the MPI is not built to the bit-size of the operating system, then the debugger may not be able to find a support library that is the correct size. This is unsupported.

Interpreting the message queues

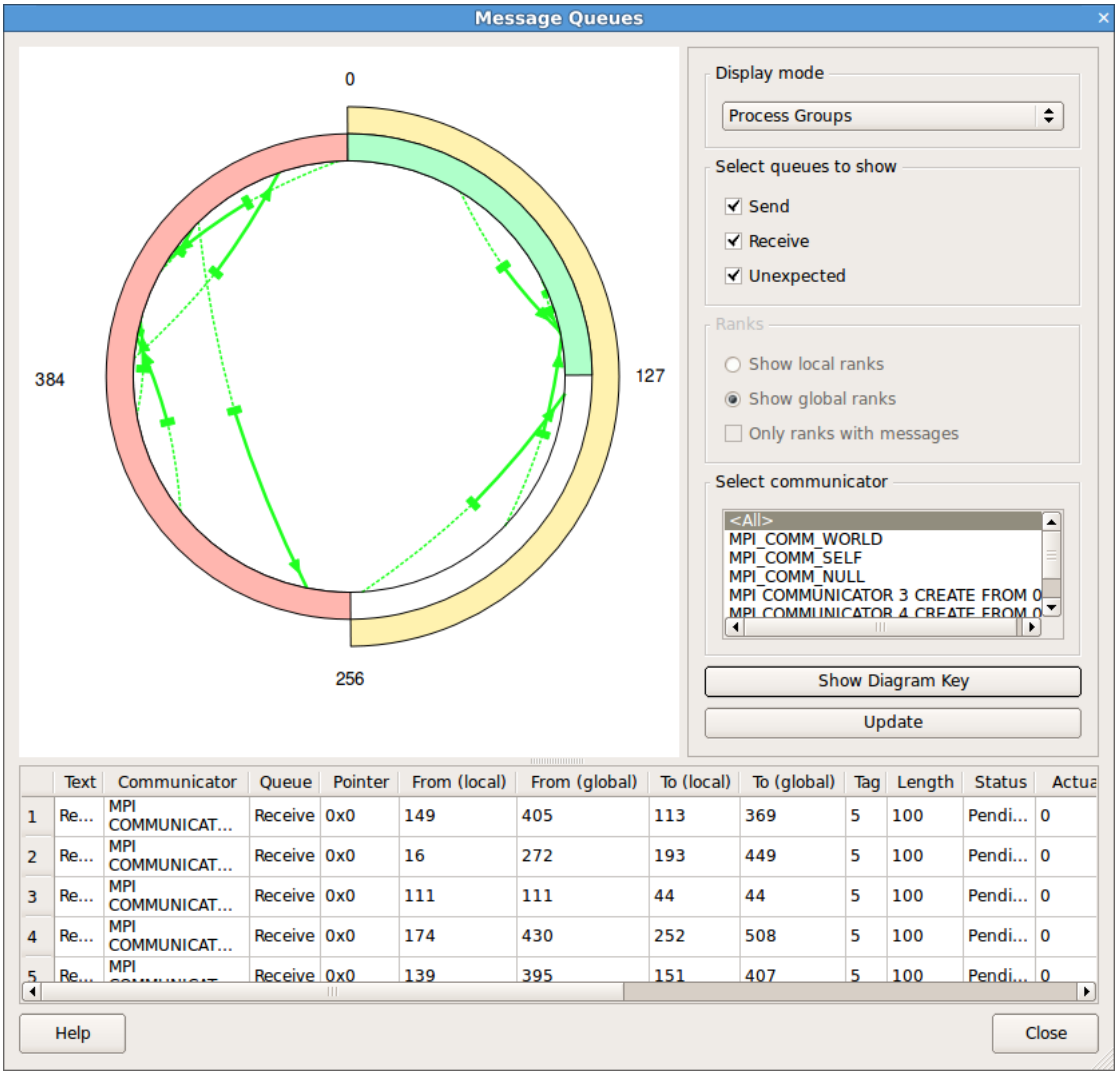

Figure 77: *Message Queue Window*

To see the messages, you must select a communicator to see the messages in that group. The ranks displayed in the diagram are the ranks within the communicator (not `MPI_COMM_WORLD`), if the *Show Local Ranks* option is selected. To see the 'usual' ranks, select *Show Global Ranks*. The messages displayed can be restricted to particular processes or groups of processes. To restrict the display in the grid to a single process, select *Individual Processes* in the *Display mode* selector, and select the rank of the process. To select a group of processes, select *Process Groups* in the *Display mode* selector and select the ring arc corresponding to the required group. Both of these display modes support multiple selections.

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

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

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

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

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

A message to self is indicated by a line with one end at the centre of the diagram.

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

Deadlock

A loop in the graph can indicate deadlock. This is where every process is waiting to receive from the preceding process in the loop. For synchronous communications, such as with `MPI_Send`, this is a common problem.

For other types of communication it can be the case, with `MPI_Send` that messages get stuck, for example in an O/S buffer, and the send part of the communication is complete but the receive has not started. If the loop persists after playing the processes and interrupting them again, this indicates a deadlock is likely.

Memory debugging

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

Typical problems which can be resolved by using Arm DDT with memory debugging enabled include:

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

Enabling memory debugging

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

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

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

If it is not possible to load the memory debugging library, a message will be displayed, and you should refer to the Configuration section in this chapter for possible solutions.

CUDA memory debugging

Arm DDT provides two options for debugging memory errors in CUDA programs, which are found in the CUDA section of the *Run* window. See section 14.2 Preparing to debug GPU code before debugging the memory of a CUDA application.

When the *Track GPU allocations* option is enabled Arm DDT tracks CUDA memory allocations *made by the host*, that is, allocations made using functions such as `cudaMalloc`). You can find out how much memory is allocated and where it was allocated from in the *Current Memory Usage* window.

Allocations are tracked separately for each GPU and the host (enabling *Track GPU allocations* will automatically track host-only memory allocations made using functions such as `malloc` as well). You can select between GPUs using the drop-down list in the top-right corner of the *Memory Usage* and *Memory Statistics* windows.

The *Detect invalid accesses (memcheck)* option turns on the CUDA-MEMCHECK error detection tool, which can detect problems such as out-of-bounds and misaligned global memory accesses, and syscall errors, such as calling `free()` in a kernel on an already free'd pointer.

The other CUDA hardware exceptions (such as a stack overflow) are detected regardless of whether this option is checked or not.

For further details about CUDA hardware exceptions, you should refer to NVIDIA's documentation.

Known issue: It is not possible to track GPU allocations created by the Cray OpenACC compiler as it does not directly call `cudaMalloc`.

Configuration

While manual configuration is often unnecessary, it can be used to adjust the memory checks and protection, or to alter the information which is gathered. A summary of the settings is displayed on the *Run* dialog in the *Memory Debugging* section.

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

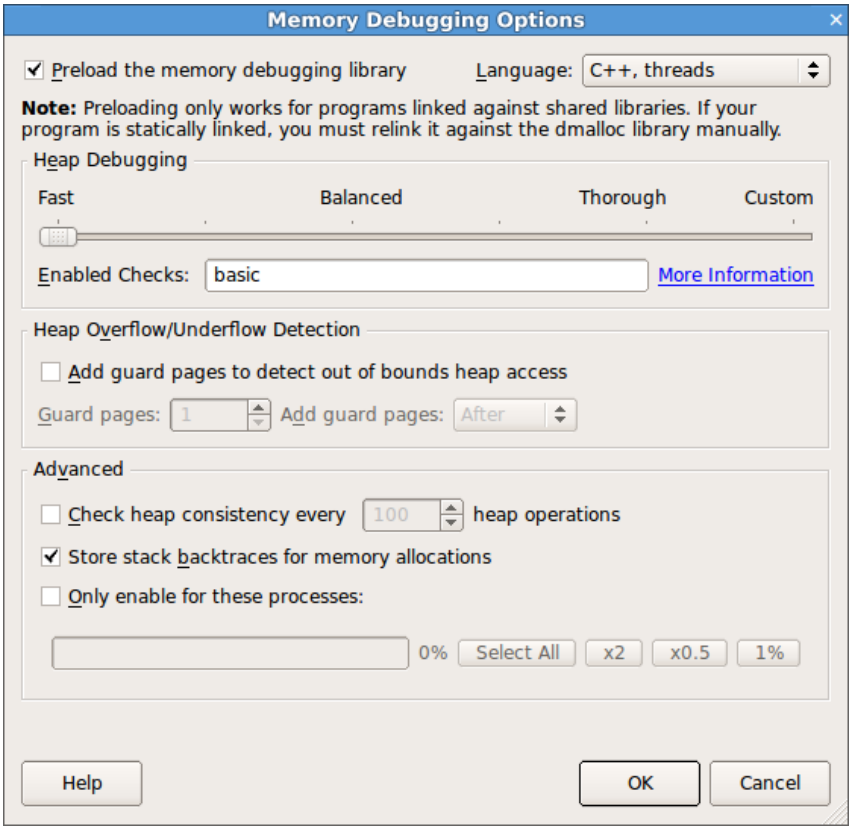

Figure 78: *Memory Debugging Options*

The two most significant options are:

1. *Preload the memory debugging library*. When this is checked, Arm DDT will automatically load the memory debugging library. Arm DDT can only preload the memory debugging library when you start a program in Arm DDT **and it uses shared libraries**.

   Preloading is not possible with statically-linked programs or when attaching to a running process. See section 12.3.1 Static linking for more information on static linking.

   When attaching, you can set the `DMALLOC_OPTIONS` environment variable before running your program, or see section 12.3.3 Changing settings at run time below.

2. The box showing *C/Fortran, No Threads* in the screen shot. You should choose the option that best matches your program. It is often sufficient to leave this set to *C++/Threaded* rather than continually changing this setting.

The *Heap Debugging* section allows you to trade speed for thoroughness. The two most important things to remember are:

1. Even the fastest (leftmost) setting will catch trivial memory errors such as deallocating memory twice.
2. The further right you go, the more slowly your program will execute. In practice, the *Balanced* setting is still fast enough to use and will catch almost all errors. If you come across a memory error that is difficult to pin down, choosing *Thorough* might expose the problem earlier, but you will need to be very patient for large, memory intensive programs. See also 12.3.3 Changing settings at run time.

You can see exactly which checks are enabled for each setting in the *Enabled Checks* box. See section 12.3.2 Available checks for a complete list of available checks.

You can turn on *Heap Overflow/Underflow Detection* to detect out-of-bounds heap access. See section 12.4.4 Writing beyond an allocated area for more details.

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

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

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

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

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

Note: Choosing the wrong library to preload or the wrong number of bits may prevent Arm DDT from starting your job, or may make memory debugging unreliable. You should check these settings if you experience problems when memory debugging is enabled.

Static linking

If your program is statically linked then you must explicitly link the memory debugging library with your program in order to use the *Memory Debugging* feature in Arm DDT.

To link with the memory debugging library, you must add the appropriate flags from the table below at the *very beginning* of the link command. This ensures that all instances of allocators, in both user code and libraries, are wrapped. Any *definition* of a memory allocator preceding the memory debugging link flags can cause partial wrapping, and unexpected runtime errors.

Note: if in doubt use `libdmallocthcxx.a`.

Multi-thread	C++	Bits	Linker Flags
no	no	64	`-Wl,--allow-multiple-definition,--undefined=malloc /path/to/ddt/lib/64/libdmalloc.a`
yes	no	64	`-Wl,--wrap=dlopen,--wrap=dlclose,--allow-multiple-definition,--undefined=malloc /path/to/ddt/lib/64/libdmallocth.a`
no	yes	64	`-Wl,--allow-multiple-definition,--undefined=malloc,--undefined=_ZdaPv /path/to/ddt/lib/64/libdmallocxx.a`
yes	yes	64	`-Wl,--wrap=dlopen,--wrap=dlclose,--allow-multiple-definition,--undefined=malloc,--undefined=_ZdaPv /path/to/ddt/lib/64/libdmallocthcxx.a`

`--undefined=malloc` has the side effect of pulling in all libc-style allocator symbols from the library. `--undefined` works on a per-object-file level, rather than a per-symbol level, and the c++ and c allocator symbols are in different object files within the library archive. Therefore, you may also need to specify a c++ style allocator such as `_ZdaPv` below.

`--undefined=_ZdaPv` has the side effect of pulling in all c++ style allocator symbols. It is the c++ mangled name of `operator delete[]`.

To link the correct library, use the full path to the static library. This is more reliable than using the `-l` argument of a compiler.

See section F.7 Intel compilers and section F.9 Portland Group compilers for compiler-specific information.

Available checks

The following heap checks are available and may be enabled in the *Enable Checks* box:

Name	Description
`basic`	Detect invalid pointers passed to memory functions (`malloc`, `free`, `ALLOCATE`, `DEALLOCATE`, etc.)
`check-funcs`	Check the arguments of addition functions (mostly string operations) for invalid pointers.
`check-heap`	Check for heap corruption, for example, due to writes to invalid memory addresses.
`check-fence`	Check the end of an allocation has not been overwritten when it is freed.
`alloc-blank`	Initialize the bytes of new allocations with a known value.
`free-blank`	Overwrite the bytes of freed memory with a known value.
`check-blank`	Check to see if space that was blanked when a pointer was allocated or when it was freed has been overwritten. Enables `alloc-blank` and `free-blank`.
`realloc-copy`	Always copy data to a new pointer when reallocating a memory allocation (for example, due to `realloc`).
`free-protect`	Protect freed memory where possible (using hardware memory protection) so subsequent read/writes cause a fatal error.

Changing settings at run time

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

Pointer error detection and validity checking

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

Library usage errors

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

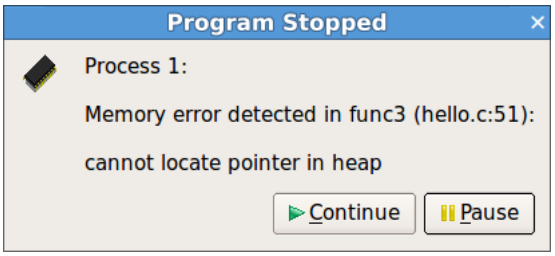

Figure 79: *Memory Error Message*

If you choose to pause the program then Arm DDT will highlight the line of your code which was being executed when the error was reported.

Often this is enough to debug simple memory errors, such as freeing or dereferencing an unallocated variable, iterating past the end of an array and so on, as the local variables and variables on the current line will provide insight into what is happening.

If the cause of the issue is still not clear, then it is possible to examine some of the pointers referenced to see whether they are valid and which line they were allocated on, as is explained in the following sections.

View pointer details

Any of the variables or expressions in the *Evaluate* window can be right-clicked on to bring up a menu. If memory debugging is enabled, *View Pointer Details* will be available. This will display the amount of memory allocated to the pointer and which part of your code originally allocated and deallocated that memory:

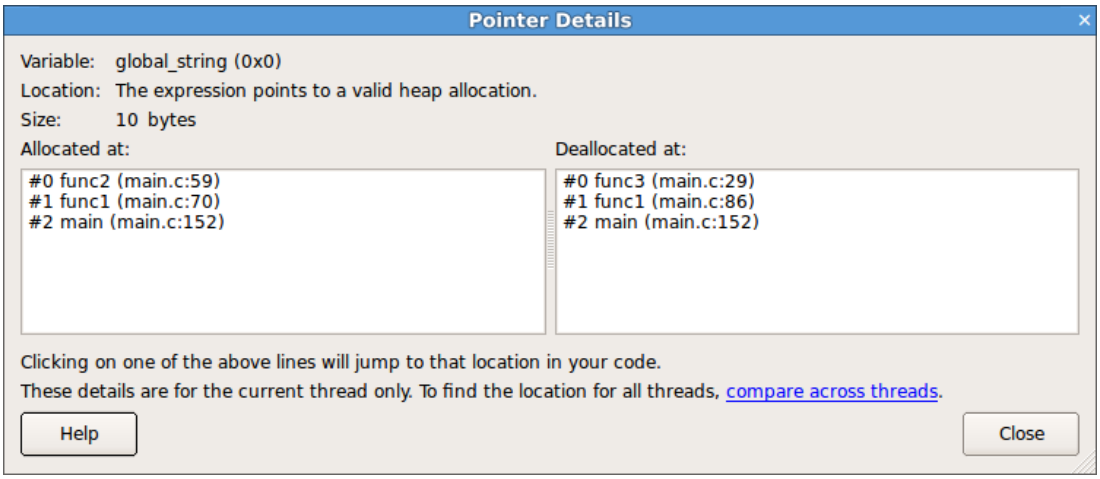

Figure 80: *Pointer details*

Clicking on any of the stack frames displays the relevant section of your code, so that you can see where the variable was allocated or deallocated.

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

This feature can also be used to check the validity of heap-allocated memory.

Note: Memory allocated on the heap refers to memory allocated by `malloc`, `ALLOCATE`, `new` *and so on. A pointer may also point to a local variable, in which case Arm 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.*

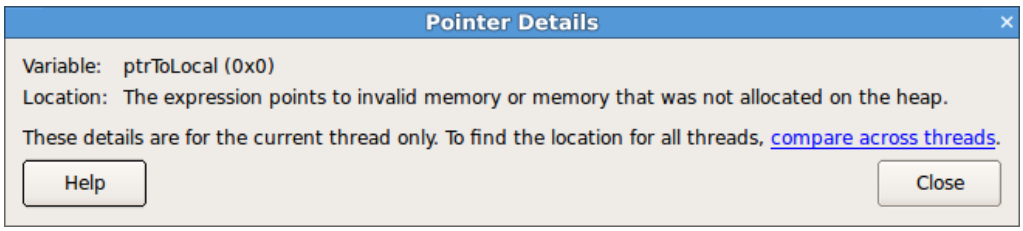

Figure 81: *Invalid memory message*

This is particularly useful for checking function arguments, and key variables when things seem to be going awry. Of course, just because memory is valid does not mean it is the same type as you were expecting, or of the same size and dimensions, and so on.

Memory Type/Location

As well as invalid addresses, Arm DDT can often indicate the type and location of the memory being pointed to. The different types are listed here:

- Null pointer.
- Valid heap allocation.
- Fence-post area before the beginning of an allocation.
- Fence-post area beyond the end of an allocation.
- Freed heap allocation.

- Fence-post area before the beginning of a freed allocation.
- Fence-post area beyond the end a freed allocation.
- A valid GPU heap allocation.
- An address on the stack.
- The program's code section (or a shared library).
- The program's data section (or a shared library).
- The program's bss section or Fortran COMMON block (or a shared library).
- The program's executable (or a shared library).
- A memory mapped file.
- High Bandwidth Memory.

Note: Arm DDT may only be able to identify certain memory types with higher levels of memory debugging enabled. See 12.3 Configuration for more information.

For more information on fence post checking, see 12.4.5 Fencepost checking

Cross-process comparison of pointers

Enabling memory debugging has an impact on the *Cross-Process Comparison* and *Cross-Thread Comparison* windows, see 8.16 Cross-process and cross-thread comparison.

If you are evaluating a pointer variable then the *Cross-Process Comparison* window shows a column with the location of the pointer.

Pointers to locations in heap memory are highlighted in green. Dangling pointers, that is pointers to locations in heap memory that have been deallocated, are shown in red.

The Cross-Process Comparison of pointers helps you to identify:

- Processes with different addresses for the same pointer.
- The location of a pointer (heap, stack, .bss, .data, .text or other locations).
- Processes that have freed a pointer while other processes have not, null pointers, and so on.

If the Cross-Process Comparison shows the value of what is being pointed at when the value of the pointer itself is wanted, then modify the pointer expression. For example, if you see the string that a `char*` pointer is pointing at when you actually want information concerning the pointer itself, then add `(void *)` to the beginning of the pointer expression.

Writing beyond an allocated area

Use the *Heap Overflow / Underflow Detection* option to detect reads and writes beyond or before an allocated block. Any attempts to read or write to the specified number of pages before or after the block will cause a segmentation violation which stops your program.

Add the guard pages after the block to detect heap overflows, or before to detect heap underflows. The default value of one page will catch most heap overflow errors, but if this does not work a good rule of thumb is to set the number of guard pages according to the size of a row in your largest array.

The exact size of a memory page depends on your operating system, but a typical size is 4 kilobytes. In this case, if a row of your largest array is 64 KiB, then set the number of pages to 64/4 = 16. Note that small overflows/underflows (for example, of less than 16 bytes) may not be detected. This is a result of maintaining correct memory alignment and without this vectorized code may crash or generate false positives. To detect small overflows or underflows, enable fencepost checking (see section 12.4.5 Fencepost checking) Note that your program will not be stopped at the exact location at which your program wrote beyond the allocated data, it only stops at the next heap consistency check.

On systems with larger page sizes (e.g. 2MB, 1GB) guard pages should be disabled or used with care as at least two pages will used per allocation. On most systems you can check the page size with `getconf PAGESIZE`.

Fencepost checking

DDT will also perform 'Fence Post' checking whenever the *Heap Debugging* setting is not set to *Fast*.

In this mode, an extra portion of memory is allocated at the start and/or end of your allocated block, and a pattern is written into this area.

If your program attempts to write beyond your data, say by a few elements, then this will be noticed by Arm DDT. However, your program will not be stopped at the exact location at which your program wrote beyond the allocated data, it will only be stopped at the next heap consistency check.

Suppressing an error

If Arm DDT stops at an error but you wish to ignore it (for example, it may be in a third party library which you cannot fix) then you may check *Suppress memory errors from this line in future*. This will open the *Suppress Memory Errors* window. Here you may select which function you want to suppress errors from.

Current memory usage

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

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

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

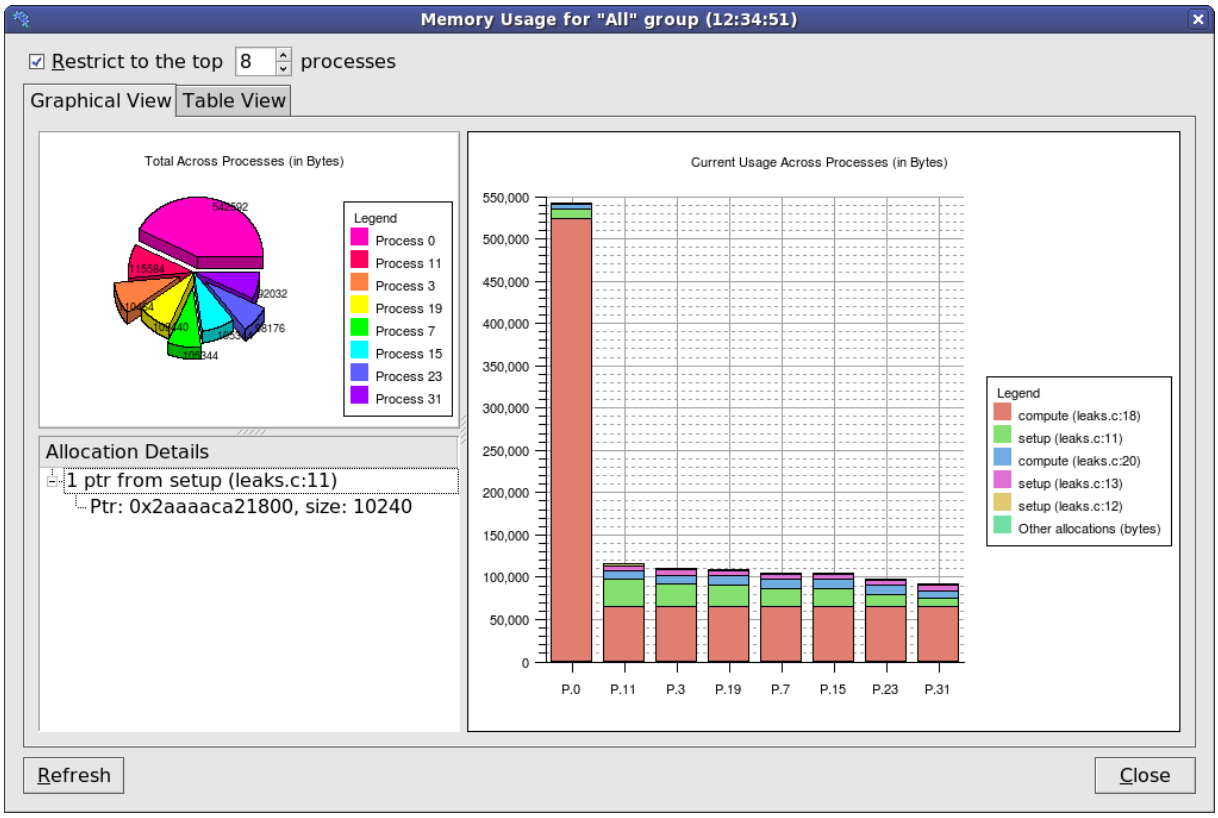

Figure 82: *Memory Usage Graphs*

To view graphical representations of memory usage, select the *Memory Usage* tab.

The pie chart gives an at-a-glance comparison of the total memory allocated to each process. This gives an indication of the balance of memory allocations. Any one process taking an unusually large amount of memory is identifiable here.

The stacked bar chart on the right is where the most interesting information starts. Each process is represented by a bar, and each bar broken down into blocks of color that represent the total amount of memory allocated by a particular function in your code. Say your program contains a loop that allocates a hundred bytes that is never freed. That is not a lot of memory. But if that loop is executed ten million times, you are looking at a gigabyte of memory being leaked! There are 6 blocks in total. The first 5 represent the 5 functions that allocated the most memory allocated, and the 6th (at the top) represents the rest of the allocated memory, wherever it is from.

As you can see, large allocations show up as large blocks of color. If your program is close to the end, or these grow, then they are severe memory leaks.

Typically, if the memory leak does not make it into the top 5 allocations under any circumstances then it may not be significant. If you are still concerned you can view the data in the *Table View* yourself.

For more information about a block of color, click on the block. This displays detailed information about the memory allocations comprising it in the bottom-left pane. Scanning down this list gives you a good idea of what size allocations were made, how many, where from and if the allocation resides in High Bandwidth Memory. Double-clicking on any one of these will display the *Pointer Details* view described above, showing you exactly where that pointer was allocated in your code.

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

To view the current memory usage in a tabular format, select the *Allocation Table* tab.

The table is split into five columns:

- **Allocated by:** Code location of the stack frame or function allocating memory in your program.
- **Count:** Number of allocations called directly from this location.
- **Total Size:** Total size (in bytes) of allocations directly from this location.
- **Count (including called functions):** Number of allocations from this location. This inludes any allocations called indirectly, for example, by calling other functions.
- **Total Size (including called functions):** Total size (in bytes) of allocations from this location, including indirect allocations.

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

Another valuable use of this feature is to play the program for a while, refresh the window, play it for a bit longer, refresh the window and so on. If you pick the points at which to refresh, for example, after units of work are complete, you can watch as the memory load of the different processes in your job fluctuates and you will see any areas which continue to grow. These are problematic leaks.

Detecting leaks when using custom allocators/memory wrappers

Some compilers wrap memory allocations inside many other functions. In this case Arm DDT may find, for example, that all Fortran 90 allocations are inside the same routine. This can also happen if you have written your own wrapper for memory allocation functions.

In these circumstances you will see one large block in the *Current Memory Usage* view. You can mark such functions as *Custom Allocators* to exclude them from the bar chart and table by right-clicking on the function and selecting the *Add Custom Allocator* menu item. Memory allocated by a custom allocator is recorded against its caller instead.

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

Memory Statistics

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

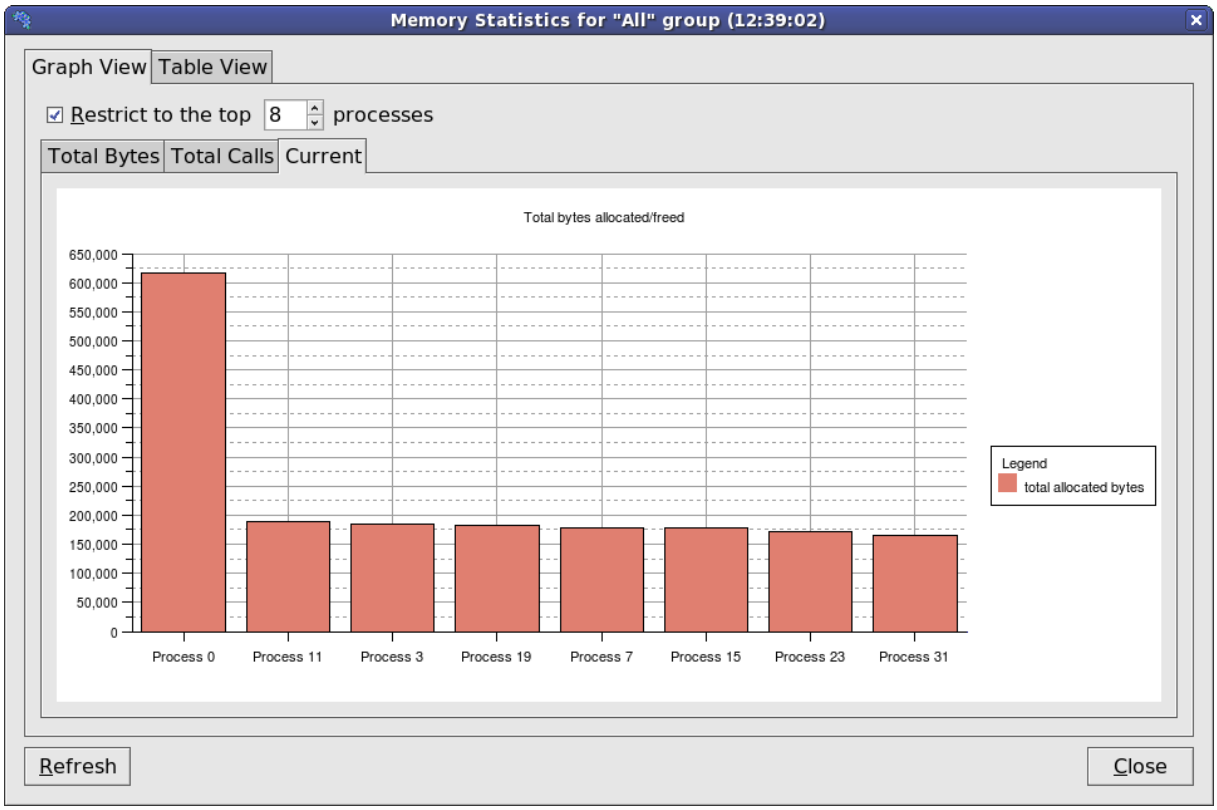

Figure 83: *Memory Statistics*

The contents and location of the memory allocations themselves are not repeated here. Instead this window displays the total amount of memory allocated and freed since the program began, the current number of allocated bytes and the number of calls to allocation and free routines.

These can help show if your application is unbalanced, if particular processes are allocating or failing to free memory and so on. 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.

If your application is using High Bandwidth Memory, the charts and tables in this dialog will be broken down into each type of memory in use.

Using and writing plugins

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

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

Supported plugins

Arm DDT supports plugins for two MPI correctness-checking libraries:

- Intel Message Checker, part of the Intel Trace Analyser and Collector (Commercial with free evaluation: http://software.intel.com/en-us/intel-trace-analyzer/) version 7.1
- Marmot (Open source: http://www.hlrs.de/organization/amt/projects/marmot), support expected in version 2.2 and above.

Arm DDT comes with two plugins for the GNU and LLVM compiler sanitizers.

- Address Sanitizer:

  The Address Sanitizer (also known as ASan) is a memory error detector for C/C++ code. It can be used to find various memory-related issues including use after free, buffer overflows, and use after return.

  To enable the Address Sanitizer:

  1. Compile your application whilst passing the `-fsanitize=address` compiler option to your compiler.
  2. Enable the Address Sanitizer plugin within Arm DDT. For more information on how to enable plugin withing Arm DDT , please refer to the 13.3 Using a plugin section.

  When compiling with GNU 7 you must disable leak detection due to a conflict with `ptrace` and this aspect of the plugin. To disable leak detection, either:

  1. Add the following piece of code into your application:

     ```
     extern "C" int __lsan_is_turned_off() { return 1; }
     ```

  2. Set the `LSAN_OPTIONS` environment variable at runtime, using:

     ```
     LSAN_OPTIONS=detect_leaks=0
     ```

  Note: ASan is not compatible with Arm DDT's memory debugging.

- Thread Sanitizer:

  The Thread Sanitizer (also known as TSan) is a data race detector for C/C++ code. A data race occurs when two different threads attempt to write to the same memory at the same time.

  To enable the Thread Sanitizer:

  1. Compile your application whilst passing the `-fsanitize=thread` compiler option to your compiler.

2. Enable the Thread Sanitizer plugin within Arm DDT. For more information on how to enable plugin within Arm DDT , please refer to the 13.3 Using a plugin section.

Note: TSan is not compatible with Arm DDT's memory debugging.

Installing a plugin

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

```
{arm-forge installation directory}/plugins/
```

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

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

Using a plugin

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

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

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

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

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

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

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

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

To install as a non-root user in your local `~/.allinea/plugins` directory, type the following command:

```
make local
```

To install as root in the DDT `plugins` directory, type the following command:

```
make
```

Once you have run the above, start Arm DDT and to enable the plugin, click the *Details…* button to expand the *Plugins* section of the *Run* window. Select `History v1.0`, and start your job as normal. DDT will take care of preloading the library and setting default tracepoints.

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

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

`_MPIHistoryCount_{function}`

The sent bytes counters are accumulated for most functions, but specifically they are not added for the vector operations such as `MPI_Gatherv`.

These count variables within the processes are available for use within Arm DDT, in components such as the cross-process comparison window, enabling a check that, for example, the count of MPI_Barriers is consistent, or primitive MPI bytes sent profiling information to be discovered.

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

There is no per-communicator logging in this version.

This version is for demonstration purposes for the tracepoints and plugin features. It could generate excessive logged information, or cause your application to run slowly if it is a heavy communicator.

This library can be easily extended, or its logging can be reduced, by removing the tracepoints from the generated `history.xml` file (stored in `ALLINEA_FORGE_PATH` or `~/.allinea/plugins`). This would make execution considerably faster, but still retain the byte and function counts for the MPI functions.

Writing a plugin

Writing a plugin for Arm DDT is described here. An XML plugin file is required that is structured similar to the following example:

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


Only the surrounding `plugin` tag is required. All the other tags are entirely optional.

A complete description of each tag appears in the following table.

Note:If you are interested in providing a plugin for DDT as part of your application bundle, Arm can provide you with any assistance you need to get up and running. Contact Arm support at Arm support for more information.

Plugin reference

Tag	Attribute	Description
`plugin`	`name`	The plugin's unique name. This should include the application/library the plugin is for, and its version. This is shown in the *DDT—Run* dialog.
`plugin`	`description`	A short snippet of text to describe the purpose of the plugin/application to the user. This is also shown in the *DDT—Run* dialog.
`preload`	`name`	Instructs DDT to preload a shared library of this name into the user's application. The shared library must be locatable using `LD_LIBRARY_PATH`, or the OS will not be able to load it.
`environment`	`name`	Instructs DDT to set a particular environment variable before running the user's application.
`environment`	`value`	The value that this environment variable should be set to.
`breakpoint`	`location`	Instructs DDT to add a breakpoint at this location in the code. The location may be in a preloaded shared library (see above). Typically this will be a function name, or a fully-qualified C++ namespace and class name. C++ class members must include their signature and be enclosed in single quotes, for example, '`MyNamespace::DebugServer::breakpointOnError(char*)`'
`breakpoint`	`action`	Only `message_box` is supported in this release. Other settings will cause DDT to stop at the breakpoint but take no action.
`breakpoint`	`message_variable`	A `char*` or `const char*` variable that contains a message to be shown to the user. DDT will group identical messages from different processes together before displaying them to the user in a message box.

`extra_control_process`	`hide`	Instructs Arm DDT to start one more MPI process than the user requested. The optional `hide` attribute can be first or last, and will cause Arm DDT to hide the `first` or `last` process in `MPI_COMM_WORLD` from the user. This process will be allowed to execute whenever at least one other MPI process is executing, and messages or breakpoints (see above) occurring in this process will appear to come from all processes at once. This is only necessary for tools such as Marmot that use an extra MPI process to perform various runtime checks on the rest of the MPI program.
`tracepoint`	`location`	See breakpoint location.
`tracepoint`	`variables`	A comma-separated list of variables to log on every passing of the tracepoint location.

CUDA GPU debugging

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

Arm supports a number of GPU compilers that target CUDA devices.

- NVCC—the NVIDIA CUDA compilers
- Cray OpenACC
- PGI CUDA Fortran and the PGI Accelerator Model
- IBM XLC/XLF with offloading support

The CUDA toolkits and their drivers for toolkits version 7.0 and above are supported by Arm DDT.

Licensing

In order to debug CUDA programs with Arm DDT, a CUDA-enabled license key is required, which is an additional option to default licenses. If CUDA is not included with a license, the CUDA options will be grayed-out on the run dialog of Arm DDT.

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

Note: In order to serve a floating CUDA license you will need to use the Licence Servershipped with Arm DDT 2.6 or later.

Preparing to debug GPU code

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

For NVIDIA's `nvcc` compiler, kernels must be compiled with the "`-g -G`" flags. This enables generation of information for debuggers in the kernels, and also disables some optimisations that would hinder debugging. To use memory debugging in DDT with CUDA "`--cudart shared`" must also be passed to `nvcc`.

For other compilers, please refer to 14.10 GPU language support of this guide and F Compiler notes and known issues and your vendor's own documentation.

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

Launching the application

To launch a CUDA job, tick the CUDA box on the run dialog before clicking run/submit. You may also enable memory debugging for CUDA programs from the CUDA section. See section 12.2 CUDA memory debugging for details.

Attaching to running CUDA applications is not possible if the application has already initialized the driver in some way, for example through having executed any kernel or called any functions from the CUDA library.

For MPI applications it is essential to place all CUDA initialization after the `MPI_Init` call.

Controlling GPU threads

Controlling GPU threads is integrated with the standard Arm DDT controls, so that the usual play, pause, and breakpoints are all applicable to GPU kernels.

As GPUs have different execution models to CPUs, there are some behavioral differences that are described in the following sections.

Breakpoints

CUDA Breakpoints can be set in the same manner as other breakpoints in Arm DDT. See section 7.6 Setting breakpoints.

Breakpoints affect all GPU threads, and cause the application to stop whenever a thread reaches the breakpoint. Where kernels have similar workload across blocks and grids, then threads tend to reach the breakpoint together and the kernel pauses once per set of blocks that are scheduled, that is, the set of threads that fit on the GPU at any one time.

Where kernels have divergent distributions of work across threads, timing may be such that threads within a running kernel hit a breakpoint and pause the kernel. After continuing, more threads within the currently scheduled set of blocks will hit the breakpoint and pause the application again.

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

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

Stepping

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

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

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

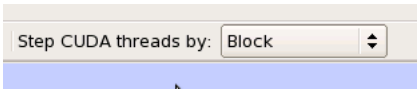

Figure 84: *Selection of GPU Stepping Mode*

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

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

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

Running and pausing

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

The pause button pauses a running kernel, although it should be noted that the pause operation is not as quick for GPUs as for regular CPUs.

Examining GPU threads and data

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

These changes are summarized in the following section.

Selecting GPU threads

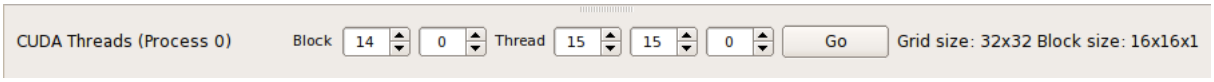

Figure 85: *GPU Thread Selector*

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

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

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

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

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

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

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

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

Viewing GPU thread locations

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

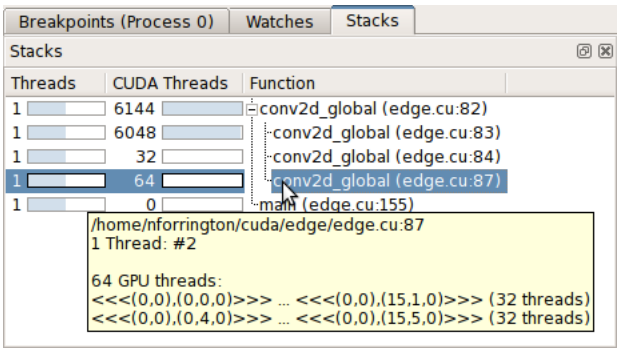

Figure 86: *CUDA threads in the parallel stack view*

Clicking an item in the Parallel Stack View selects the appropriate GPU thread, updating the variable display components accordingly and moving the source code viewer to the appropriate location.

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

Understanding kernel progress

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

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

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

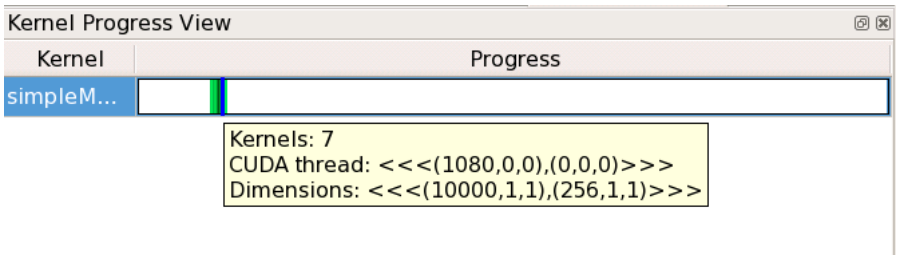

Figure 87: *Kernel Progress Display*

This view identifies the kernels that are in progress. The number of kernels are identified and grouped by different kernel identifiers across processes. The identifier is the kernel name.

A colored progress bar is used to identify which GPU threads are in progress. The progress bar is a projection onto a straight line of the (potentially) 6-dimensional GPU block and thread indexing system and is tailored to the sizes of the kernels operating in the application.

By clicking within the color highlighted sections of this progress bar, a GPU thread will be selected that matches the click location as closely *as* possible. Selected GPU threads are colored blue. For de-selected GPU threads, the ones that are scheduled are colored green whereas the unscheduled ones are white.

Source code viewer

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

GPU devices information

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

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

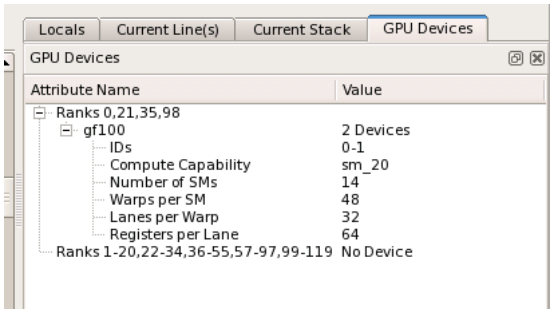

Figure 88: *GPU Devices*

Note: GPU devices are only listed after initialization.

Attaching to running GPU applications

Attaching to a running GPU application and then debugging the GPU threads is only supported for Fermi class cards and their successors. This includes Tesla C2050/2070, K10, and K20.

To attach to a running job, please see the section 5.9 Attaching to running programs and select the *Debug CUDA* button on the attach window.

Opening GPU core files

In CUDA 7.0, NVIDIA introduced support for GPU code to generate core files. These can be opened in DDT in exactly the same way as core files generated by CPU code. See 5.8 Opening core files for details.

Known issues / limitations

Debugging multiple GPU processes

CUDA allows debugging of multiple CUDA processes on the same node. However, each process will still attempt to reserve all of the available GPUs for debugging.

This works for the case where a single process debugs all GPUs on a node, but not for multiple processes debugging a single GPU.

A temporary workaround when using Open MPI is to export the following environment variable before starting DDT:

```
ALLINEA_CUDA_DEVICE_VAR=OMPI_COMM_WORLD_LOCAL_RANK
```

This will assign a single device (based on local rank) to each process. In addition:

- You must have *Open MPI (Compatibility)* selected in the *File* → *Options* (*Arm Forge* → *Preferences* on Mac OS X) . (Not Open MPI).
- The device selected for each process will be the only device visible when enumerating GPUs. This cause manual GPU selection code to stop working (due to changing device IDs, and so on).

Thread control

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

General

- DDT supports versions 7.0 onwards of the NVIDIA CUDA toolkit. In all cases, the most recent CUDA toolkit and driver versions is recommended.
- X11 cannot be running on any GPU used for debugging. (Any GPU running X11 will be excluded from device enumeration.)
- You must compile with `-g -G` to enable GPU debugging otherwise your program will run through the contents of kernels without stopping.
- Debugging 32-bit CUDA code on a 64-bit host system is not supported.
- It is not yet possible to spot unsuccessful kernel launches or failures. An error code is provided by `getCudaLastError()` in the SDK which you can call in your code to detect this. Currently the debugger cannot check this without resetting it, which is not desirable behavior.
- Device memory allocated via `cudaMalloc()` is not visible outside of the kernel function.
- Not all illegal program behavior can be caught in the debugger, for example, divide-by-zero.
- Device allocations larger than 100 MB on Tesla GPUs, and larger than 32 MB on Fermi GPUs, may not be accessible.
- Breakpoints in divergent code may not behave as expected.
- Debugging applications with multiple CUDA contexts running on the same GPU is not supported.

- If CUDA environment variable `CUDA_VISIBLE_DEVICES` <index> is used to target a particular GPU, then make sure no X server is running on any of the GPUs. Also note that any GPU running X will be excluded from enumeration, with may affect the device Ids.
- CUDA drivers requires that applications be debugged in a mode matching their version. If your system is running with a toolkit version lower than the CUDA driver version, you should force DDT to use the correct CUDA version by setting the `ALLINEA_FORCE_CUDA_VERSION` enviroment variable. For example, if you are using the CUDA 7.5 driver, set `ALLINEA_FORCE_CUDA_VERSION=7.5`. Alternatively, you should consider upgrading your CUDA toolkit to match the CUDA driver.
- If memory debugging and CUDA support are enabled in Arm DDT then only threaded memory preloads are available.

Pre sm_20 GPUs

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

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

Debugging multiple GPU processes on Cray limitations

It is not possible to debug multiple CUDA processes on a single node on a Cray machine, you must run with 1 process per node.

GPU language support

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

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

Cray OpenACC

Cray OpenACC is fully supported by Arm DDT. Code pragmas are highlighted, most variables are visible within the device, and stepping and breakpoints in the GPU code is supported. The compiler flag `-g` is required for enabling device (GPU-based) debugging; `-O0` should not be used, as this disables use of the GPU and runs the accelerated regions on the CPU.

You should be aware of the following known issues:

- It is not possible to track GPU allocations created by the Cray OpenACC compiler as it does not directly call `cudaMalloc`.
- Pointers in accelerator code cannot be dereferenced in CCE 8.0.

- Memory consumption in debugging mode can be considerably higher than regular mode, if issues with memory exhaustion arise, consider using the environment variable `CRAY_ACC_MALLOC_HEAPSIZE` to set total heap size (bytes) used on the device, which can make more memory available to the application.

PGI Accelerators and CUDA Fortran

Arm DDT supports debugging both the host and CUDA parts of PGI Accelerator and CUDA Fortran programs compiled with version 14.4 or later of the PGI compiler. Older versions of the PGI compiler support debugging only on the host.

IBM XLC/XLF with offloading OpenMP

Arm DDT supports debugging both the host and CUDA parts of OpenMP programs making use of offloading when compiled with version 13.1.7 or later of the IBM XLC/XLF compilers.

For the best debugging experience of offloading OpenMP regions, the following compiler flags are recommended `-g -O0 -qsmp=omp:noopt -qoffload -qfullpath -qnoinline -Xptxas -O0 -Xllvm2ptx -nvvm-compile-options=-opt=0`.

Offline debugging

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

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

Using offline debugging

To launch Arm DDT in this mode, the `--offline` argument is specified. Optionally, an output filename can be supplied with the –output=<filename> argument. A filename with a `.html` or `.htm` extension will cause an HTML version of the output to be produced, in other cases a plain text report is generated. If the –output argument is not used, DDT generates an HTML output file in the current working directory and reports the name of that file upon completion.

```
ddt --offline mpiexec -n 4 myprog arg1 arg2
ddt --offline -o myjob.html mpiexec -n 4 myprog arg1 arg2
ddt --offline -o myjob.txt mpiexec -n 4 myprog arg1 arg2
ddt --offline -o myjob.html --np=4 myprog arg1 arg2
ddt --offline -o myjob.txt --np=4 myprog arg1 arg2
```

Additional arguments can be used to set breakpoints, at which the stack of the stopping processes will be recorded before they are continued. You can also set tracepoints at which variable values will be recorded. Additionally, expressions can be set to be evaluated on every program pause.

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

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

- `--session=`*`SESSIONFILE`* – run in offline mode using settings saved using the Save Session option from the Arm DDT File menu.
- `--processes=`*`NUMPROCS`* or `-n` *`NUMPROCS`* – run with *`NUMPROCS`* processes
- `--mem-debug[=(fast|balanced|thorough|off)]` – enable and configure memory debugging
- `--snapshot-interval=`*`MINUTES`* – write a snapshot of the program's stack and variables to the offline log file every *MINUTES* minutes.

  See section 15.4 below.
- `--trace-at=`*`LOCATION`*`[,`*`N`*`:`*`M`*`:`*`P`*`],`*`VAR1`*`,`*`VAR2`*`,...] [if` *`CONDITION`*`]` – set a tracepoint at location, beginning recording after the *`N`*'th visit of each process to the location, and recording every *`M`*'th subsequent pass until it has been triggered *`P`* times. Record the value of variable *`VAR1`*, *`VAR2`*. The if clause allows you to specify a boolean *CONDITION* that must be satisfied for the tracepoint to trigger.

  Example:

  ```
  main.c:22,-:2:-,x
  ```

  This will record x every 2nd passage of line 22.

- `--break-at=`*`LOCATION`*`[,`*`N`*`:`*`M`*`:`*`P`*`][if `*`CONDITION`*`]` – set a breakpoint at *`LOCATION`* (either *`file`*:*`line`* or *`function`*), optionally starting after the *`N`*'th pass, triggering every *`M`* passes and stopping after it has been triggered *`P`* times. The if clause allows you to specify a boolean *CONDITION* that must be satisfied for the breakpoint to trigger. When using the if clause the value of this argument should be quoted.

  The stack traces of paused processes will be recorded, before the processes are then made to continue, and will contain the variables of one of the processes in the set of processes that have paused.

  Examples:

  ```
  --break-at=main
  --break-at=main.c:22
  --break-at=main.c:22 --break-at=main.c:34
  ```

- `--evaluate=`*`EXPRESSION`*`[;`*`EXPRESSION2`*`][;...]` – set one or more expressions to be evaluated on every program pause. Multiple expressions should be separated by a semicolon and enclosed in quotes. If shell special characters are present the value of this argument should also be quoted.

  Examples:

  ```
  --evaluate=i
  --evaluate="i; (*addr) / x"
  --evaluate=i --evaluate="i * x"
  ```

- `--offline-frames=(all|none|n)` – specify how many frames to collect variables for, where `n` is a positive integer. The default value is all.

  Examples:

  ```
  --offline-frames=all
  --offline-frames=none
  --offline-frames=1337
  ```

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

When errors occur, for example an application crash, the stack back trace of crashing processes is recorded to the offline output file. In offline mode, Arm DDT always acts as if the user had clicked Continue if the continue option was available in an equivalent "online" debugging session.

Reading a file for standard input

In offline mode, normal redirection syntax can be used to read data from a file as a source for the executable's standard input.

Examples:

```
cat <input-file> | ddt --offline -o myjob.html ...
ddt --offline -o myjob.html ... < <input-file>
```

Writing a file from standard output

Normal redirection can also be used to write data to a file from the executable's standard output:

```
ddt --offline -o myjob.html ... > <output-file>
```


Offline report output (HTML)

The output file is broken into four sections, *Messages*, *Tracepoints*, *Memory Leak Report*, and *Output*. At the end of a job, Arm DDT merges the four sections of the log output (tracepoint data, error messages, memory leak data, and program output) into one file. If the Arm DDT process is terminated abruptly, for example by the job scheduler, then these separate files will remain and the final single HTML report may not be created. Note that a memory leak report section is only created when memory debugging is enabled.

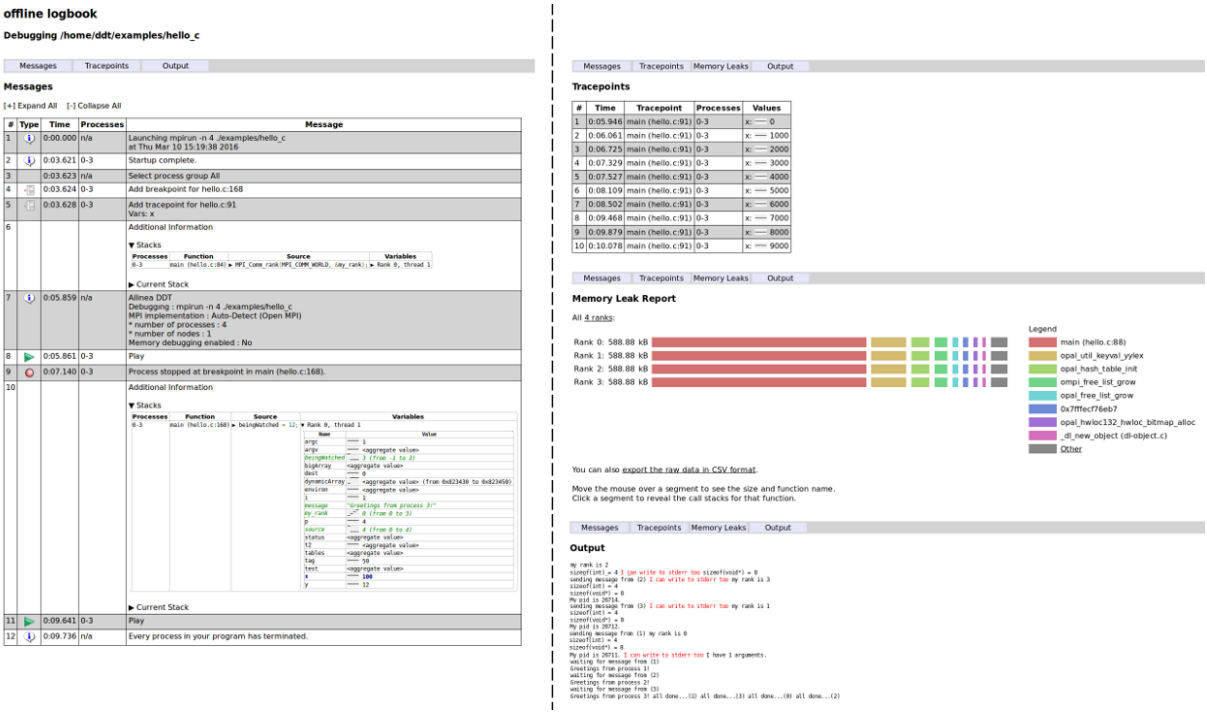

Figure 89: *Offline Mode HTML output*

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

The Messages section contains the following:

- Error messages: for example if Arm DDT's Memory Debugging detects an error then the message and the stack trace at the time of the error will be recorded from each offending processes.
- Breakpoints: a message with the stopped processes and another one with the Stacks, Current Stack and Locals at this point.
- Additional Information: after an error or a breakpoint has stopped execution, then an additional information message is recorded. This message could contain the stacks, current stack and local variables for the stopped processes at this point of the execution.
  - The Stacks table displays the parallel stacks from all paused processes. Additionally, for every top-most frame the variables (locals and arguments) will be displayed by default. You can use the `--offline-frames` command line option to display the variables for more frames or none. If `--offline-frames=none` is specified no variables at all will be displayed, instead a Locals table will show the variables for the current process. Clicking on a function expands the code snippet and variables in one go. If the stop was caused by an error or crash, the stack of the responsible thread or process is listed first.

- The Current Stacks table shows the stack of the current process.
- The Locals table (if `--offline-frames=none`) and the Variables column of the Stacks table shows the variables across the paused processes. The text highlighting scheme is the same as for the Local variables in the GUI. The Locals table shows the local variables of the current process, whereas the Variables column shows the locals for a representative process that triggered the stop in that frame. In either case a sparkline for each variable shows the distribution of values across the processes.

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

The *Memory Leak Report* section displays a graphical representation of the largest memory allocations that were not freed by the end of the program:

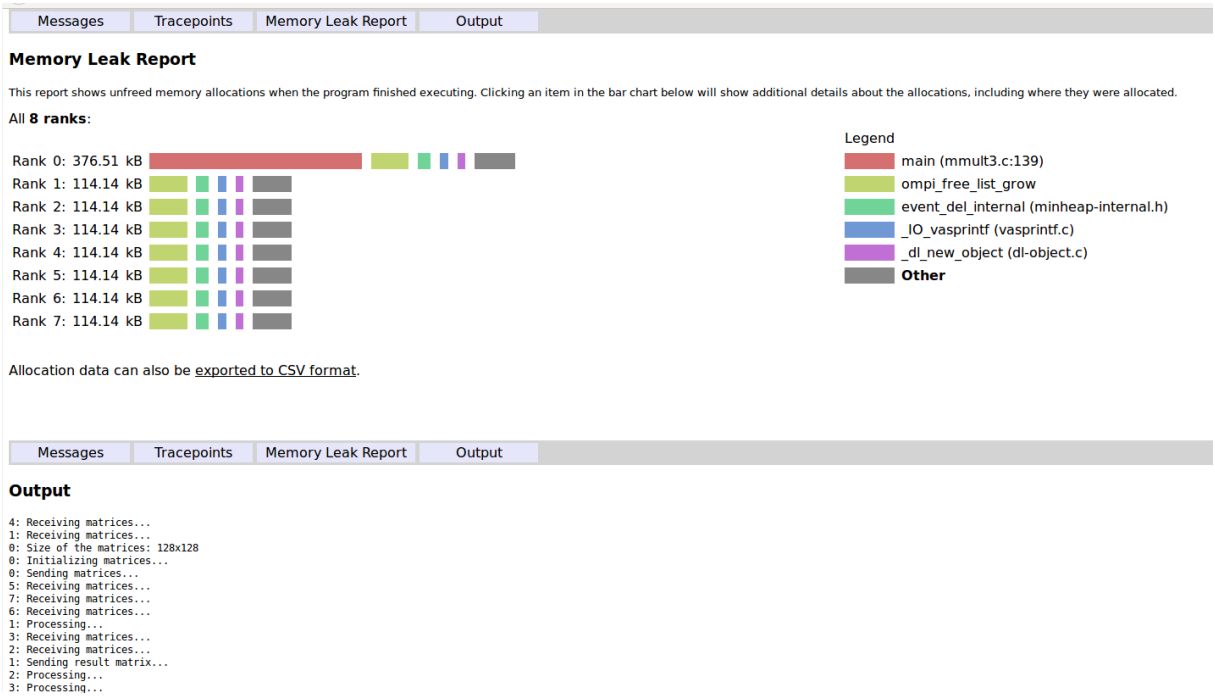

Figure 90: *Memory leak report*

Each row corresponds to the memory still allocated at the end of a job on a single rank. If multiple MPI ranks are being debugged, only those with the largest number of memory allocations are shown. You can configure the number of MPI ranks shown with `--leak-report-top-ranks=X`.

The memory allocations on each rank are grouped by the source location that allocated them. Each colored segment corresponds to one location, identified in the legend. Clicking on a segment reveals a table of all call paths leading to that location along with detailed information about the individual memory allocations:

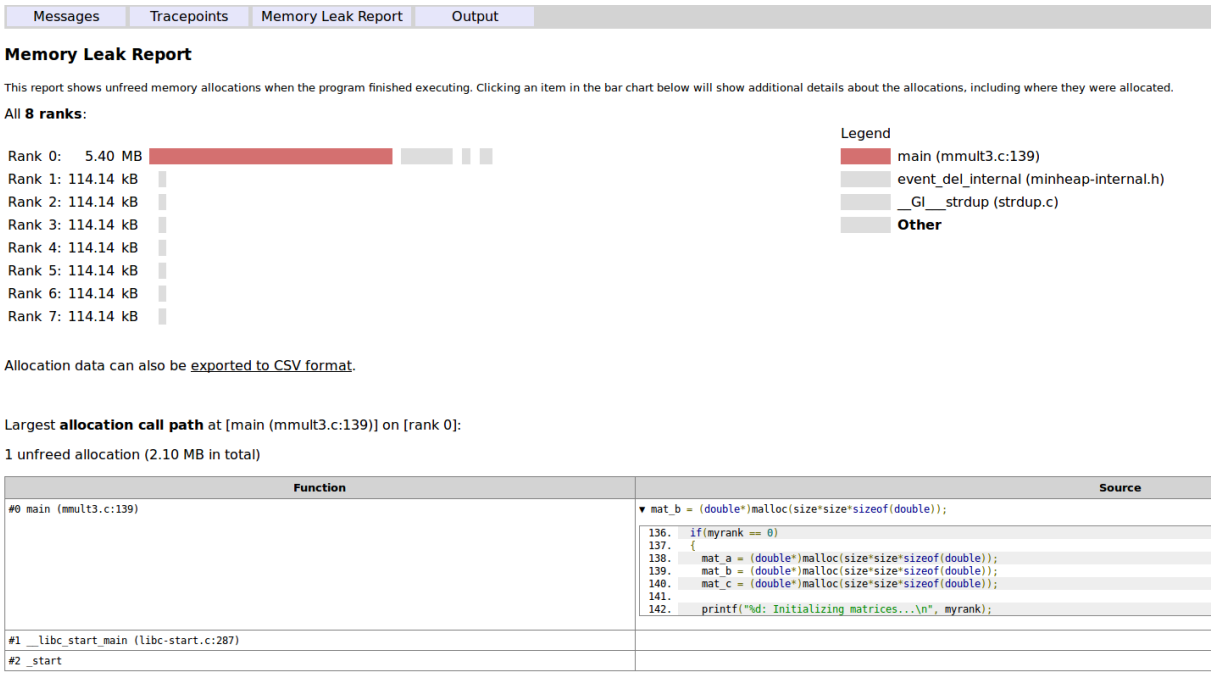

Figure 91: *Memory leak report detail*

By default all locations that contribute less than 1% of the total allocations are grouped together into the "Other" item in the legend.

This limit can be configured by setting the `ALLINEA_LEAK_REPORT_MIN_SEGMENT` environment variable to a percentage. For example, `ALLINEA_LEAK_REPORT_MIN_SEGMENT=0.5` will only group locations with less than 0.5% of the total allocated bytes together.

In addition, only the eight largest locations are shown by default. This can be configured with the `--leak-report-top-locations=Y` command-line option.

The raw data may also be exported by clicking the export link.

You may find the following command line options useful:

Option	Description
`--leak-report-top-ranks=X`	Limit the memory leak report to the top X ranks (default 8, implies `--mem-debug`)
`--leak-report-top-locations=Y`	Limit the memory leak report to the top Y locations in each rank (default 8, implies `--mem-debug`)
`--leak-report-top-call-paths=Z`	Limit the memory leak report to the top Z call paths to each allocating function (default 8, implies `--mem-debug`)

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

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

Offline report output (plain text)

Unlike the offline report in HTML mode, the plain text mode does not separate the tracepoint, breakpoint, memory leak, and application output into separate sections.

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

For example, a simple report could look like the following:

```
message (0-3): Process stopped at breakpoint in main (hello.c:97).
message (0-3): Stacks
message (0-3): Processes Function
message (0-3): 0-3    main (hello.c:97)
message (0-3): Stack for process 0
message (0-3): #0 main (argc=1, argv=0x7fffffffd378, \
        environ=0x7fffffffd388) at /home/ddt/examples/hello.c:97
message (0-3): Local variables for process 0 \
        (ranges shown for 0-3)
message (0-3): argc: 1 argv: 0x7fffffffd378 beingWatched: 0 \
        dest: 7 environ: 0x7fffffffd388 i: 0 message: ",!\312\t" \
        my_r ank: 0 (0-3) p: 4 source: 0 status: t2: 0x7ffff7ff7fc0 \
        tables: tag: 50 test: x: 10000 y: 12
```

Run-time job progress reporting

In offline mode, Arm DDT can be instructed to compile a snapshot of a job, including its stacks and variables, and update the session log with that information. This includes writing the HTML log file, which otherwise is only written once the session has completed.

Snapshots can be triggered periodically via a command-line option, or at any point in the session by sending a signal to the Arm DDT front-end.

Periodic snapshots

Snapshots can be triggered periodically throughout a debugging session with the command-line option `--snapshot-interval=`*`MINUTES`*. For example, to log a snapshot every three minutes:

```
ddt --offline -o log.html --snapshot-interval=3 \
    mpiexec -n 8 ./myprog
```

Signal-triggered snapshots

Snapshots can also be triggered by sending a SIGUSR1 signal to the DDT front-end process (called `ddt.bin` in process lists), regardless of whether or not the `--snapshot-interval` command-line option was specified. For example, after running the following:

```
ddt --offline -o log.html mpiexec -n 8 ./myprog
```

A snapshot can be triggered by running (in another terminal):


```
# Find PID of DDT front-end:
pgrep ddt.bin
> 18032
> 18039

# Use pstree to identify the parent if there are multiple PIDs:
pstree -p

# Trigger the snapshot:
kill -SIGUSR1 18032
```


Part III

MAP

Getting started

Arm MAP is a source-level profiler and can show how much time was spent on each line of code. To see the source code in MAP, compile your program with the debug flag, which for the most compilers this is `-g`. Do not use a debug build as you should always keep optimization flags turned on when profiling.

You can also use MAP on programs without debug information. In this case inlined functions are not shown and the source code cannot be shown but other features should work as expected.

To start MAP simply type one of the following shell commands into a terminal window:

```
map
map program_name [arguments]
map <profile-file>
```

Where `<profile-file>` is a profile file generated by a MAP profiling run. It contains the program name and has a '.map' extension.

Note: When starting MAP for examining an existing profile file, a valid license is not needed.

Note: Unless you are using Express Launch mode (see 16.1 Express Launch), you should not attempt to pipe input directly to MAP. For information about how to achieve the effect of sending input to your program, please read section 9 Program input and output.

It is also recommended you add the `--profile` argument to MAP. This runs without the interactive GUI and saves a `.map` file to the current directory and is ideal for profiling jobs submitted to a queue.

Once started in interactive mode, MAP displays the Welcome Page:

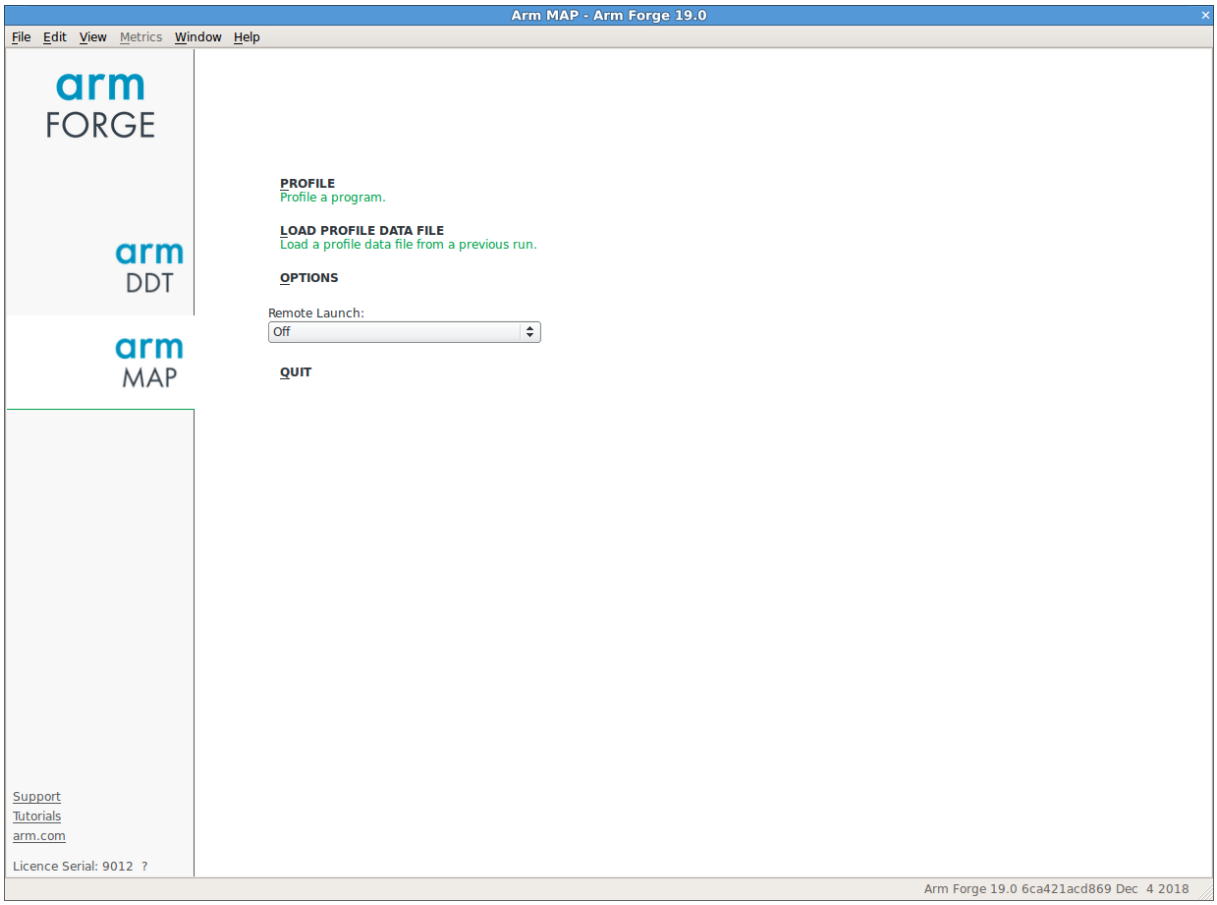

Figure 92: *MAP Welcome Page*

Note: In Express Launch mode (see 16.1 Express Launch) the Welcome Page is not shown and the user is brought directly to the Run Dialog instead. If no valid license is found, the program is exited and the appropriate message is shown in the console output.

The *Welcome Page* allows you to choose what kind of profiling you want to do. You can choose from the following:

- Profile a program.
- Load a Profile from a previous run.
- Connect to a remote system and accept a Reverse Connect request.

Express Launch

Each of the Arm Forge products can be launched by typing its name in front of an existing `mpiexec` command:

```
$ map mpiexec -n 256 examples/wave_c 30
```

This startup method is called *Express Launch* and is the simplest way to get started. If your MPI is not yet supported in this mode, you will see a error message similar to the following:

```
$ 'MPICH 1 standard' programs cannot be started using Express
   Launch syntax (launching with an mpirun command).
```



```
Try this instead:
    map --np=256 ./wave_c 20

Type map --help for more information.
```

This is referred to as *Compatibility Mode*, in which the `mpiexec` command is not included and the arguments to `mpiexec` are passed via a `--mpiargs="args here"` parameter.

One advantage of Express Launch mode is that it is easy to modify existing queue submission scripts to run your program under one of the Arm Forge products. This works best for MAP, which gathers data without an interactive GUI (`map --profile`) or Reverse Connect (`map --connect`, see 3.3 Reverse Connect for more details) for interactive profiling.

If you can not use Reverse Connect and wish to use interactive profiling from a queue you may need to configure MAPto generate job submission scripts for you. More details on this can be found in 16.7 Starting a job in a queue and A.2 Integration with queuing systems.

The following lists the MPI implementations supported by Express Launch:

- bullx MPI
- Cray X-Series (MPI/SHMEM/CAF)
- Intel MPI
- MPICH 2
- MPICH 3
- Open MPI (MPI/SHMEM)
- Oracle MPT
- Open MPI (Cray XT/XE/XK)
- Spectrum MPI
- Spectrum MPI (PMIx)
- Cray XT/XE/XK (UPC)

Run dialog box

In Express Launch mode, the Run dialog has a restricted number of options:

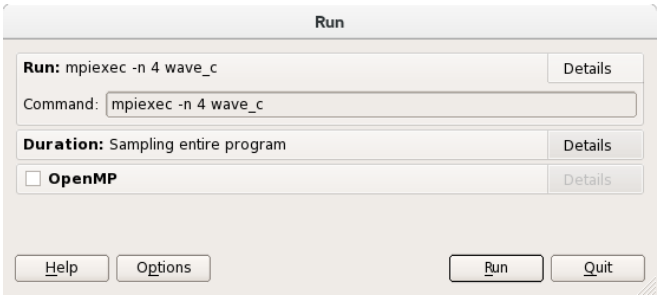

Figure 93: *Express Launch MAP Run dialog box*

Preparing a program for profiling

In most cases, if your program is already compiled with debugging symbols then you do not need to re-compile your program to use it with MAP, although in some cases it may need to be relinked, as explained in section 16.2.2 Linking.

Note: Ensure that the program does not set or unset the `SIGPROF` signal handler because this interferes with the MAP profiling function and can cause it to fail.

Note: It is not recommended to use MAP to profile programs that contain instructions to perform MPI profiling using MPI wrappers and the MPI standard profiling interface, PMPI. This is because MAP's own MPI wrappers may conflict with those contained in the program, producing incorrect metrics.

Debugging symbols

Arm MAP is a source-level profiler and can show how much time was spent on each line of code. To see the source code in MAP, compile your program with the debug flag, for example:

```
mpicc hello.c -o hello -g -O3
```

Do not just use a debug build. You should always keep optimization flags turned on when profiling.

You can also use MAP on programs without debug information. In this case inlined functions are not shown and the source code cannot be shown but other features will work as expected.

Cray compiler

For the Cray compiler Arm recommends using the `-G2` option with MAP.

CUDA programs

When compiling CUDA kernels do not generate debug information for device code (the `-G` or `--device-debug` flag) as this can significantly impair runtime performance. Use `-lineinfo` instead, for example:

```
nvcc device.cu -c -o device.o -g -lineinfo -O3
```

Linking

To collect data from your program, MAP uses two small profiler libraries, `map-sampler` and `map-sampler-pmpi`. These must be linked with your program. On most systems MAP can do this automatically without any action by you. This is done via the system's `LD_PRELOAD` mechanism, which allows an extra library into your program when starting it.

Note: Although these libraries contain the word 'map' they are used for both Arm Performance Reports and Arm MAP.

This automatic linking when starting your program only works if your program is dynamically-linked. Programs may be dynamically-linked or statically-linked, and for MPI programs this is normally determined by your MPI library. Most MPI libraries are configured with `--enable-dynamic` by default, and `mpicc/mpif90` produce dynamically-linked executables that MAP can automatically collect data from.

The `map-sampler-pmpi` library is a temporary file that is precompiled and copied or compiled at runtime in the directory `~/.allinea/wrapper`.

If your home directory will not be accessible by all nodes in your cluster you can change where the `map-sampler-pmpi` library will be created by altering the `shared directory` as described in H.2.5 No shared home directory.

The temporary library will be created in the `.allinea/wrapper` subdirectory to this `shared directory`.

For Cray X-Series Systems the `shared directory` is not applicable, instead `map-sampler-pmpi` is copied into a hidden `.allinea` sub-directory of the current working directory.

If MAP warns you that it could not pre-load the sampler libraries, this often means that your MPI library was not configured with `--enable-dynamic`, or that the `LD_PRELOAD` mechanism is not supported on your platform. You now have three options:

1. Try compiling and linking your code dynamically. On most platforms this allows MAP to use the `LD_PRELOAD` mechanism to automatically insert its libraries into your application at runtime.
2. Link MAP's `map-sampler` and `map-sampler-pmpi` libraries with your program at link time manually.

   See 16.2.3 Dynamic linking on Cray X-Series systems, or 16.2.4 Static linking and 16.2.5 Static linking on Cray X-Series systems.
3. Finally, it may be that your system supports dynamic linking but you have a statically-linked MPI. You can try to recompile the MPI implementation with `--enable-dynamic`, or find a dynamically-linked version on your system and recompile your program using that version. This will produce a dynamically-linked program that MAP can automatically collect data from.

Dynamic linking on Cray X-Series systems

If the `LD_PRELOAD` mechanism is not supported on your Cray X-Series system, you can try to dynamically link your program explicitly with the MAP sampling libraries.

Compiling the Arm MPI Wrapper Library

First you must compile the Arm MPI wrapper library for your system using the `make-profiler-libraries --platform=cray --lib-type=shared` command.

Note: Performance Reports also uses this library.

```
user@login:~/myprogram$ make-profiler-libraries --platform=cray
   --lib-type=shared

Created the libraries in /home/user/myprogram:
libmap-sampler.so       (and .so.1, .so.1.0, .so.1.0.0)
libmap-sampler-pmpi.so  (and .so.1, .so.1.0, .so.1.0.0)

To instrument a program, add these compiler options:
compilation for use with MAP - not required for Performance
   Reports:
    -g (or '-G2' for native Cray Fortran) (and -O3 etc.)
linking (both MAP and Performance Reports):
    -dynamic -L/home/user/myprogram -lmap-sampler-pmpi -lmap-
       sampler -Wl,--eh-frame-hdr
```



```
Note: These libraries must be on the same NFS/Lustre/GPFS
   filesystem as your
program.

Before running your program (interactively or from a queue), set
LD_LIBRARY_PATH:
export LD_LIBRARY_PATH=/home/user/myprogram:$LD_LIBRARY_PATH
map  ...
or add -Wl,-rpath=/home/user/myprogram when linking your program.
```

Linking with the Arm MPI Wrapper Library

```
mpicc -G2 -o hello hello.c -dynamic -L/home/user/myprogram \
      -lmap-sampler-pmpi -lmap-sampler -Wl,--eh-frame-hdr
```

PGI Compiler

When linking OpenMP programs you must pass the `-Bdynamic` command line argument to the compiler when linking dynamically.

When linking C++ programs you must pass the `-pgc++libs` command line argument to the compiler when linking.

Static linking

If you compile your program statically, that is your MPI uses a static library or you pass the `-static` option to the compiler, then you must explicitly link your program with the Arm sampler and MPI wrapper libraries.

Compiling the Arm MPI Wrapper Library

First you must compile the Arm MPI wrapper library for your system using the `make-profiler-libraries --lib-type=static` command.

Note: Performance Reports also uses this library.

```
user@login:~/myprogram$ make-profiler-libraries --lib-type=static

Created the libraries in /home/user/myprogram:
   libmap-sampler.a
   libmap-sampler-pmpi.a

To instrument a program, add these compiler options:
   compilation for use with MAP - not required for Performance
      Reports:
      -g (and -O3 etc.)
   linking (both MAP and Performance Reports):
      -Wl,@/home/user/myprogram/allinea-profiler.ld ...
         EXISTING_MPI_LIBRARIES
   If your link line specifies EXISTING_MPI_LIBRARIES (e.g. -lmpi)
      , then
   these must appear *after* the Arm sampler and MPI wrapper
      libraries in
   the link line.  There's a comprehensive description of the link
       ordering
```



```
requirements in the 'Preparing a Program for Profiling' section
    of either
userguide-forge.pdf or userguide-reports.pdf, located in
/opt/arm/forge/doc/.
```

Linking with the Arm MPI Wrapper Library

The `-Wl,@/home/user/myprogram/allinea-profiler.ld` syntax tells the compiler to look in `/home/user/myprogram/allinea-profiler.ld` for instructions on how to link with the Arm sampler. Usually this is sufficient, but not in all cases. The rest of this section explains how to manually add the Arm sampler to your link line.

PGI Compiler

When linking C++ programs you must pass the `-pgc++libs` command line argument to the compiler when linking.

The PGI C runtime static library contains an undefined reference to `__kmpc_fork_call`, which will cause compilation to fail when linking `allinea-profiler.ld`. Add `--undefined __wrap_-__kmpc_fork_call` to your link line before linking to the Arm sampler to resolve this.

The PGI compiler must be 14.9 or later. Using earlier versions of the PGI compiler will fail with an error such as "`Error: symbol 'MPI_F_MPI_IN_PLACE' can not be both weak and common`" due to a bug in the PGI compiler's weak object support.

If you do not have access to PGI compiler 14.9 or later try compiling and the linking Arm MPI wrapper as a shared library as described in 16.2.3 Dynamic linking on Cray X-Series systems Ommit the option `--platform=cray` if you are not on a Cray.

Cray

When linking C++ programs you may encounter a conflict between the Cray C++ runtime and the GNU C++ runtime used by the MAP libraries with an error similar to the one below:

```
/opt/cray/cce/8.2.5/CC/x86-64/lib/x86-64/libcray-c++-rts.a(rtti.o)
   : In function `__cxa_bad_typeid':
/ptmp/ulib/buildslaves/cfe-82-edition-build/tbs/cfe/lib_src/rtti.c
   :1062: multiple definition of `__cxa_bad_typeid'
/opt/gcc/4.4.4/snos/lib64/libstdc++.a(eh_aux_runtime.o):/tmp/peint
   /gcc/repackage/4.4.4c/BUILD/snos_objdir/x86_64-suse-linux/
   libstdc++-v3/libsupc++/../../../../xt-gcc-4.4.4/libstdc++-v3/
   libsupc++/eh_aux_runtime.cc:46: first defined here
```

You can resolve this conflict by removing `-lstdc++` and `-lgcc_eh` from `allinea-profiler.ld`.

-lpthread

When linking `-Wl,@allinea-profiler.ld` must go before the `-lpthread` command line argument if present.

Manual Linking

When linking your program you must add the path to the profiler libraries (`-L/path/to/profiler-libraries`), and the libraries themselves (`-lmap-sampler-pmpi`, `-lmap-sampler`).

The MPI wrapper library (`-lmap-sampler-pmpi`) must go:

1. *After* your program's object (`.o`) files.
2. *After* your program's own static libraries, for example `-lmylibrary`.

3. *After* the path to the profiler libraries (`-L/path/to/profiler-libraries`).
4. *Before* the MPI's Fortran wrapper library, if any. For example `-lmpichf`.
5. *Before* the MPI's implementation library usually `-lmpi`.
6. *Before* the Arm sampler library `-lmap-sampler`.

The sampler library `-lmap-sampler` must go:

1. *After* the Arm MPI wrapper library.
2. *After* your program's object (`.o`) files.
3. *After* your program's own static libraries, for example `-lmylibrary`.
4. *After* `-Wl,--undefined,allinea_init_sampler_now`.
5. *After* the path to the profiler libraries `-L/path/to/profiler-libraries`.
6. *Before* `-lstdc++`, `-lgcc_eh`, `-lrt`, `-lpthread`, `-ldl`, `-lm` and `-lc`.

For example:

```
mpicc hello.c -o hello -g -L/users/ddt/arm \
   -lmap-sampler-pmpi \
   -Wl,--undefined,allinea_init_sampler_now \
   -lmap-sampler -lstdc++ -lgcc_eh -lrt \
   -Wl,--whole-archive -lpthread \
   -Wl,--no-whole-archive \
   -Wl,--eh-frame-hdr \
   -ldl \
   -lm

mpif90 hello.f90 -o hello -g -L/users/ddt/arm \
   -lmap-sampler-pmpi \
   -Wl,--undefined,allinea_init_sampler_now \
   -lmap-sampler -lstdc++ -lgcc_eh -lrt \
   -Wl,--whole-archive -lpthread \
   -Wl,--no-whole-archive \
   -Wl,--eh-frame-hdr \
   -ldl \
   -lm
```

Static linking on Cray X-Series systems

Compiling the MPI Wrapper Library

On Cray X-Series systems use `make-profiler-libraries --platform=cray --lib-type=static` instead:

```
Created the libraries in /home/user/myprogram:
   libmap-sampler.a
   libmap-sampler-pmpi.a

To instrument a program, add these compiler options:
   compilation for use with MAP - not required for Performance
      Reports:
```



```
   -g (or -G2 for native Cray Fortran) (and -O3 etc.)
linking (both MAP and Performance Reports):
   -Wl,@/home/user/myprogram/allinea-profiler.ld ...
      EXISTING_MPI_LIBRARIES
If your link line specifies EXISTING_MPI_LIBRARIES (e.g. -lmpi)
   , then
these must appear *after* the Arm sampler and MPI wrapper
   libraries in
the link line.  There's a comprehensive description of the link
    ordering
requirements in the 'Preparing a Program for Profiling' section
    of either
userguide-forge.pdf or userguide-reports.pdf, located in
/opt/arm/forge/doc/.
```

Linking with the MPI Wrapper Library

```
cc hello.c -o hello -g -Wl,@allinea-profiler.ld

ftn hello.f90 -o hello -g -Wl,@allinea-profiler.ld
```

Dynamic and static linking on Cray X-Series systems using the modules environment

If your system has the Arm module files installed, you can load them and build your application as usual. See section 16.2.7.

1. `module load forge` or ensure that `make-profiler-libraries` is in your PATH.
2. `module load map-link-static` or `module load map-link-dynamic`.
3. Recompile your program.

map-link modules installation on Cray X-Series

To facilitate dynamic and static linking of user programs with the Arm MPI Wrapper and Sampler libraries Cray X-Series System Administrators can integrate the map-link-dynamic and map-link-static modules into their module system. Templates for these modules are supplied as part of the Arm Forge package.

Copy files `share/modules/cray/map-link-*` into a dedicated directory on the system.

For each of the two module files copied:

1. Find the line starting with **conflict** and correct the prefix to refer to the location the module files were installed, for example, `arm/map-link-static`. The correct prefix depends on the sub-directory (if any) under the module search path the `map-link-*` modules were installed.
2. Find the line starting with **set MAP_LIBRARIES_DIRECTORY "NONE"** and replace "NONE" with a user writable directory accessible from the login and compute nodes.

After installed you can verify whether or not the prefix has been set correctly with 'module avail', the prefix shown by this command for the map-link-* modules should match the prefix set in the 'conflict' line of the module sources.

Profiling a program

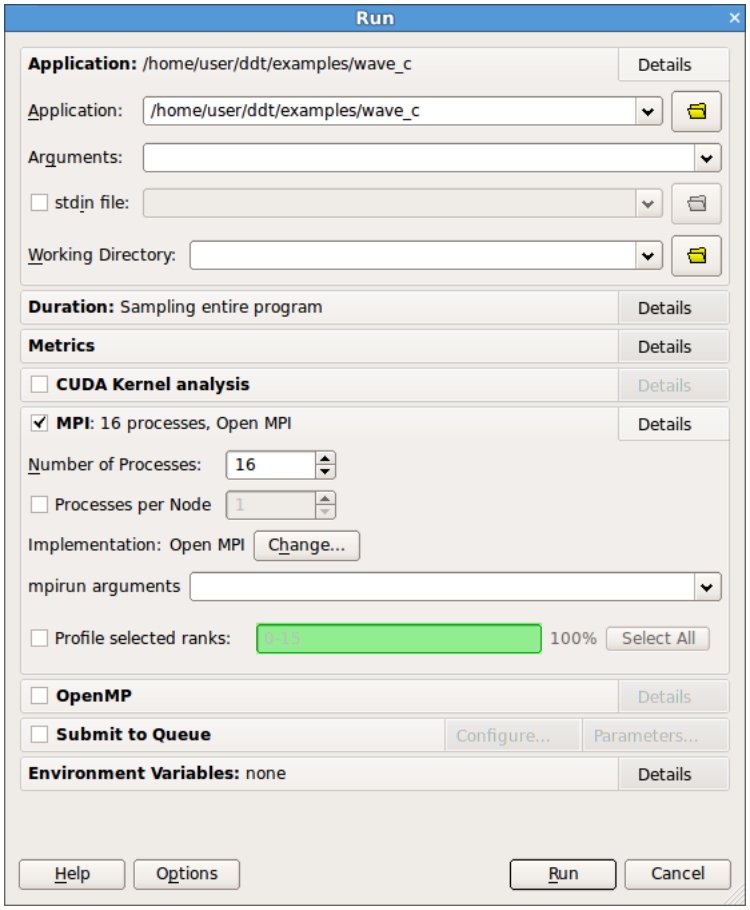

Figure 94: *Run window*

If you click the *Profile* button on the MAP *Welcome Page* you will see the window above. The settings are grouped into sections. Click the *Details…* button to expand a section. The settings in each section are described below.

Application

Application: The full path name to your application. If you specified one on the command line, this will already be filled in. You may browse for an application by clicking on the *Browse* button.

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

Arguments: (optional) The arguments passed to your application. These will be automatically filled if you entered some on the command line.

Note: Avoid using quote characters such as `'` and `"`, as these may be interpreted differently by MAP and your command shell. If you must use these and cannot get them to work as expected, please contact Arm support at Arm support.

stdin file: (optional) This allows you to choose a file to be used as the standard input (`stdin`) for your program. MAP will automatically add arguments to `mpirun` to ensure your input file is used.

Working Directory: (optional) The working directory to use when running your application. If this is blank then MAP's working directory will be used instead.

Duration

Start profiling after: (optional) This allows you to delay profiling by a number of seconds into the run of your program.

Stop profiling after: (optional) This allows you to specify a number of seconds after which the profiler will terminate your program.

Metrics

This section allows you to explicitly enable and disable metrics for which data is collected. Metrics are listed alphabetically with their display name and unique metric ID under their associated metric group. Select a metric to see a more detailed description, including the metric's default enabled/disabled state.

Only metrics that can be displayed in MAP's metrics view are listed. Metrics that are unlicensed, unsupported or always disabled are not listed. Additionally, you cannot disable metrics that are always enabled.

The initial state of enabled/disabled metrics are the combined settings given by the metric XML defintions, the previous GUI session, and those specified with the `--enabled-metrics` and `--disable-metrics` command line options. The command line options take preference over the previous GUI session settings, and both take preference over the metric XML defintions settings. Of course, metrics that are always enabled or always disabled cannot be toggled.

All PAPI metrics displays if installed, and available for enabling/disabling. However, only metrics specified in the `PAPI.config` file are affected.

All Perf metrics displays if available for enabling/disabling.

MPI

Note: If you only have a single process license or have selected none as your MPI Implementation *the* MPI *options will be missing. The* MPI *options are not available when in single process mode. See section 16.5 Profiling a single-process program for more details about using a single process.*

Number of processes: The number of processes that you wish to profile. MAP supports hundreds of thousands of processes but this is limited by your license. This option may not be displayed if disabled on the *Job Submission* options page.

Number of nodes: This is the number of compute nodes that you wish to use to run your program. This option is only displayed for certain MPI implementations or if it is enabled on the *Job Submission* options page.

Processes per node: This is the number of MPI processes to run on each compute node. This option is only displayed for certain MPI implementations or if it is enabled on the *Job Submission* options page.

Implementation: The MPI implementation to use, for example, Open MPI, MPICH 2. Normally the Auto setting will detect the currently loaded MPI module correctly. If you are submitting a job to a queue

the queue settings will also be summarized here. You may change the MPI implementation by clicking on the *Change…* button.

Note: The choice of MPI implementation is critical to correctly starting MAP. 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 Arm support. A list of settings for common implementations is provided in E MPI distribution notes and known issues.

Note: If your desired MPI command is not in your `PATH`, or you wish to use an MPI run command that is not your default one, you can configure this using the Options window. See section A.5.1 System.

mpirun arguments: (optional) The arguments that are passed to `mpirun` or your equivalent, usually prior to your executable name in normal `mpirun` usage. You can place machine file arguments, if necessary, here. For most users this box can be left empty.

Note: You should not enter the `-np` argument as MAP will do this for you.

Profile selected ranks: (optional) If you do not want to profile all the ranks, you can specify a set of ranks to profile. The ranks should be separated by commas and intervals are accepted. Example: 5,6-10.

OpenMP

Number of OpenMP threads: The number of OpenMP threads to run your program with. This ensures the `OMP_NUM_THREADS` environment variable is set, but your program may override this by calling OpenMP-specific functions.

Environment variables

The optional Environment Variables section should contain additional environment variables that should be passed to `mpirun` or its equivalent. 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.

Profiling

Click *Run* to start your program, or *Submit* if working through a queue. See section A.2 Integration with queuing systems. This will compile up a MPI wrapper library on the fly that can intercept the `MPI_INIT` call and gather statistics about MPI use in your program. If this has problems see H.9.3 MPI wrapper libraries. Then MAP brings up the Running window and starts to connect to your processes.

The program runs inside MAP which starts collecting stats on your program through the MPI interface you selected and will allow your MPI implementation to determine which nodes to start which processes on.

MAP collects data for the entire program run by default. Arm's sampling algorithms ensure only a few tens of megabytes are collected even for very long-running jobs. You can stop your program at any time by using the *Stop and Analyze* button. MAP will then collect the data recorded so far, stop your program and end the MPI session before showing you the results. If any processes remain you may have to clean them up manually using the `kill` command, or a command provided with your MPI implementation, but this should not be necessary.

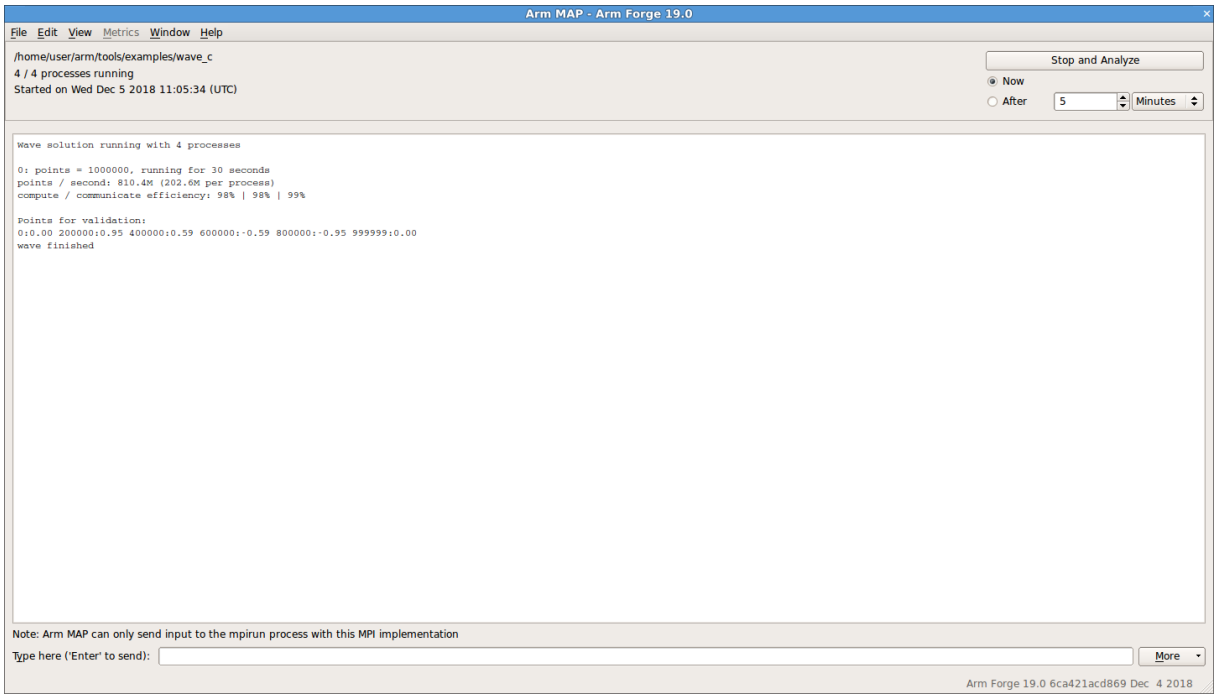

Figure 95: *Running window*

Profiling only part of a program

The easiest way to profile only part of a program in MAP is to use the ”Start profiling after” and ”Stop profiling after” settings in the Run dialog, or the equivalent `--start-after=TIME` and `--stop-after=TIME` command-line options. These allow you to specify a range of wall-clock time (the job starts at 0 seconds) during which the job should be profiled. Once the `--stop-after` time is reached, the job is terminated rather than letting it run to the end.

Alternatively, for more fine-grained control you may choose to start profiling programmatically at a later point by instrumenting your code. To do this you must set the `ALLINEA_SAMPLER_DELAY_START=1` environment variable before starting your program. For MPI programs it is important that this variable is set in the environment of all the MPI processes. It is not necessarily sufficient to simply set the variable in the environment of the MPI command itself. You must arrange for the variable to be set or exported by your MPI command for all the MPI processes.

You may call `allinea_start_sampling` and `allinea_stop_sampling` once each. That is to say there must be one and only one contiguous sampling region. It is not possible to start, stop, start, stop. You cannot pause or resume sampling using the `allinea_suspend_traces` and `allinea_resume_traces` functions. This will not have the desired effect. You may only delay the start of sampling and stop sampling early.

16.3.8.1 C

To start sampling programmatically you should `#include "mapsampler_api.h"` and call the `allinea_start_sampling` function. You will need to point your C compiler at the MAP include directory, by passing the arguments `-I <install root>/map/wrapper` and also link with the MAP sampler library, by passing the arguments `-L <install root>/lib/64 -lmap-sampler`. To stop sampling progammatically call the `allinea_stop_sampling` function.

16.3.8.2 Fortran

To start sampling programmatically you should call the `ALLINEA_START_SAMPLING` subroutine. You will also need to link with the MAP sampler library, for example by passing the arguments `-L <install root>/lib/64 -lmap-sampler`. To stop sampling programmatically call the `ALLINEA_STOP_SAMPLING` subroutine.

`remote-exec` required by some MPIs

When using SGI MPT, MPICH 1 Standard or the MPMD variants of MPICH 2, MPICH 3 or Intel MPI, MAP will allow `mpirun` to start all the processes, then attach to them while they're inside `MPI_Init`.

This method is often faster than the generic method, but requires the `remote-exec` facility in MAP to be correctly configured if processes are being launched on a remote machine. For more information on `remote-exec`, please see section A.4 Connecting to remote programs (remote-exec).

Note: If MAP is running in the background, for example using `map` &, then this process may get stuck. Some SSH versions cause this behavior when asking for a password. If this happens to you, go to the terminal and use the `fg` or similar command to make MAP a foreground process, or run MAP again, without using "&".

If MAP cannot find a password-free way to access the cluster nodes then you will not be able to use the specialized startup options. Instead, You can use *generic*, although startup may be slower for large numbers of processes.

Profiling a single-process program

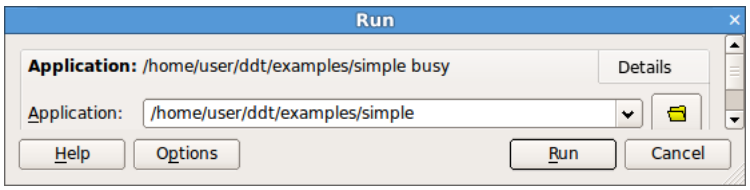

Figure 96: *Single-Process Run Window*

1. If you have a single-process license you will immediately see the *Run Window* that is appropriate for single-process applications. If your license supports multiple processes you can simply clear the *MPI* checkbox to run a single-process program.
2. Select the application, either by typing the file name in, or selecting it using the browser displayed by clicking the browse button.
3. Arguments can be typed into the supplied box.
4. If appropriate, tick the *OpenMP* box and select the *Number of OpenMP threads* to start your program with.
5. Click *Run* to start your program.

Sending standard input

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

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

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

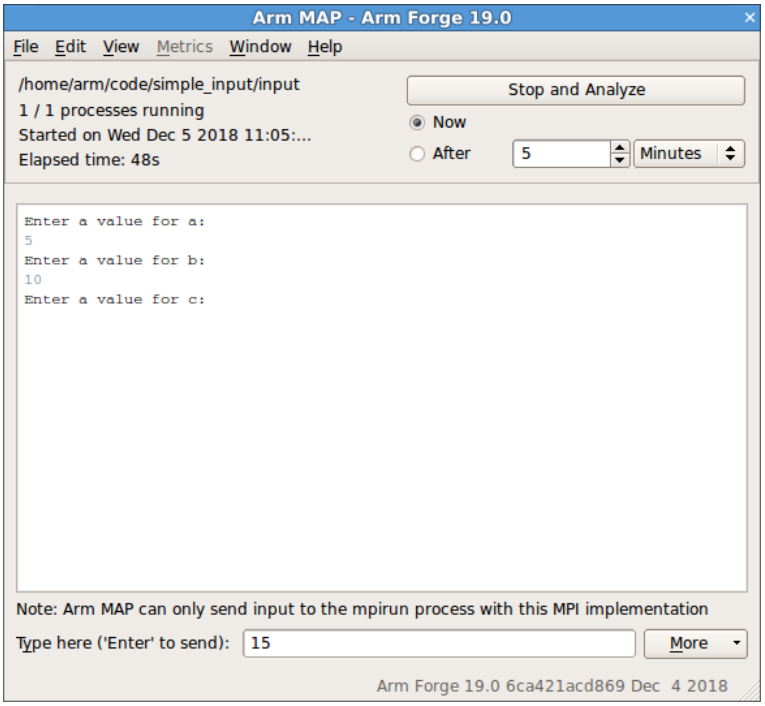

Figure 97: *MAP Sending Input*

Note: If MAP is running on a fork-based system such as Scyld, or a `-comm=shared` compiled MPICH 1, 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 H General troubleshooting and known issues or contact Arm support at Arm support for a list of possible fixes.

Starting a job in a queue

If MAP has been configured to be integrated with a queue/batch environment, as described in section A.2 Integration with queuing systems then you may use it to launch your job.

In this case, a *Submit* button is presented on the Run Window, instead of the ordinary *Run* button. Clicking *Submit* from the Run Window will display the queue status until your job starts. MAP 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 MAP will not be able to remove your job from the queue.

It is strongly recommended 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 profiling sessions.

After the sampling (program run) phase is complete, MAP will start the analysis phase, collecting and processing the distinct samples. This could be a lengthy process depending on the size of the program. For very large programs it could be as much as 10 or 20 minutes.

You should ensure that your job does not hit its queue limits during the analysis process, setting the job time large enough to cover both the sampling and the analysis phases.

MAP will also require extra memory both in the sampling and in the analysis phases. If these fail and your application alone approaches one of these limits then you may need to run with fewer processes per node or a smaller data set in order to generate a complete set of data.

Once your job is running, it will connect to MAP and you will be able to profile it.

Using custom MPI scripts

On some systems a custom `mpirun` replacement is used to start jobs, such as `mpiexec`. MAP will normally use whatever the default for your MPI implementation is, so for MPICH 1 it would look for `mpirun` and not `mpiexec`, for SLURM it would use `srun` and so on. This section explains how to configure MAP to use a custom `mpirun` command for job start up.

There are typically two ways you might want to start jobs using a custom script, and MAP supports them both.

The first way is to pass all the arguments on the command-line, as in the following example:

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

There are several key variables in this line that MAP 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 its arguments remains constant.

To use a command like this in MAP, the queue submission system is adpated as described in the previous section. For this `mpiexec` example, the settings would be as shown here:

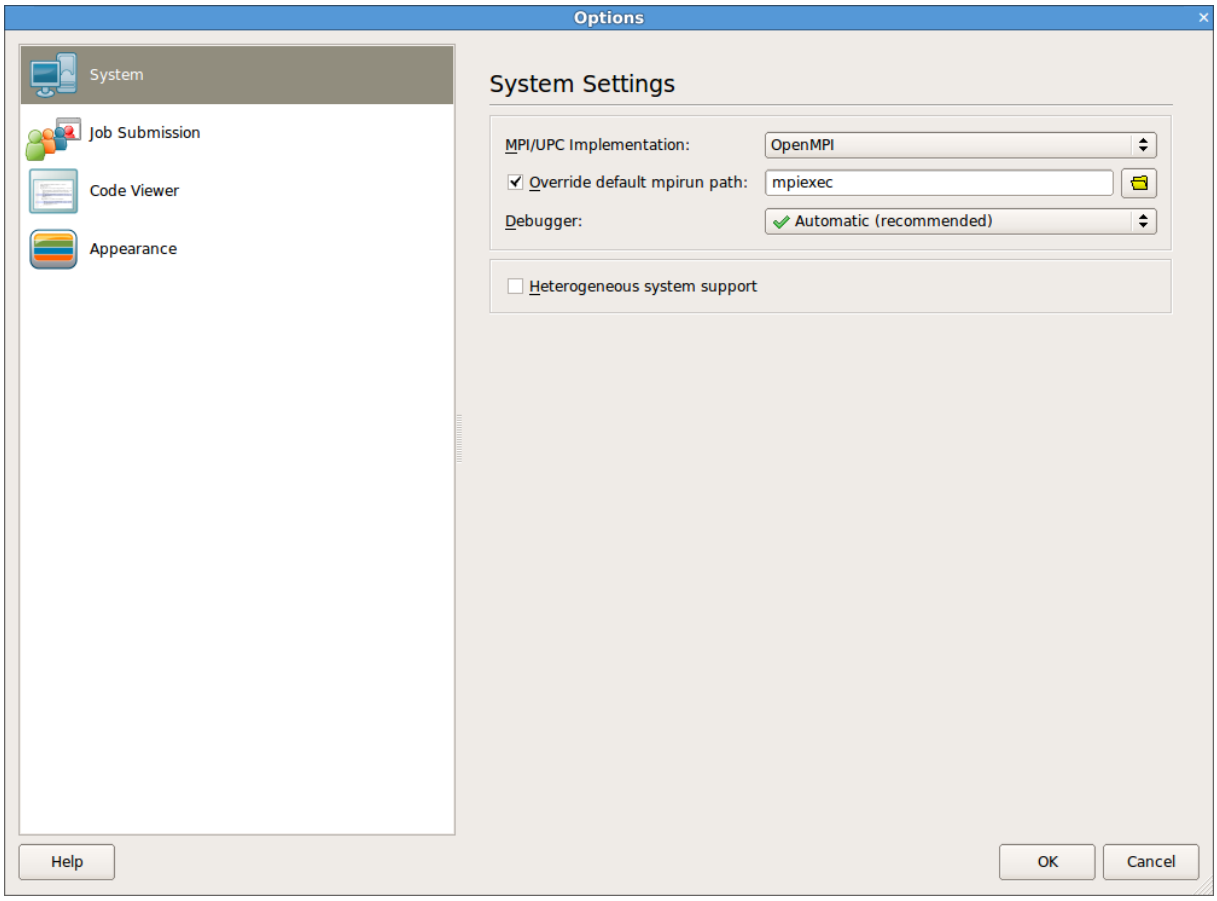

Figure 98: *MAP Using Custom MPI Scripts*

As you can see, most of the settings are left blank.

There are some differences between the *Submit Command* in MAP 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`, used by both DDT and MAP.
3. The program arguments are replaced by `PROGRAM_ARGUMENTS_TAG`.

Note: it is not necessary to specify the program name here. MAP 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 the following:

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

MAP can automatically generate simple configuration files like this every time you run your program if you specify a template file. For the above example, the template file `myfile.template` would contain the following:


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

This follows the same replacement rules described above and in detail in section A.2 Integration with queuing systems.

The options settings for this example might be:

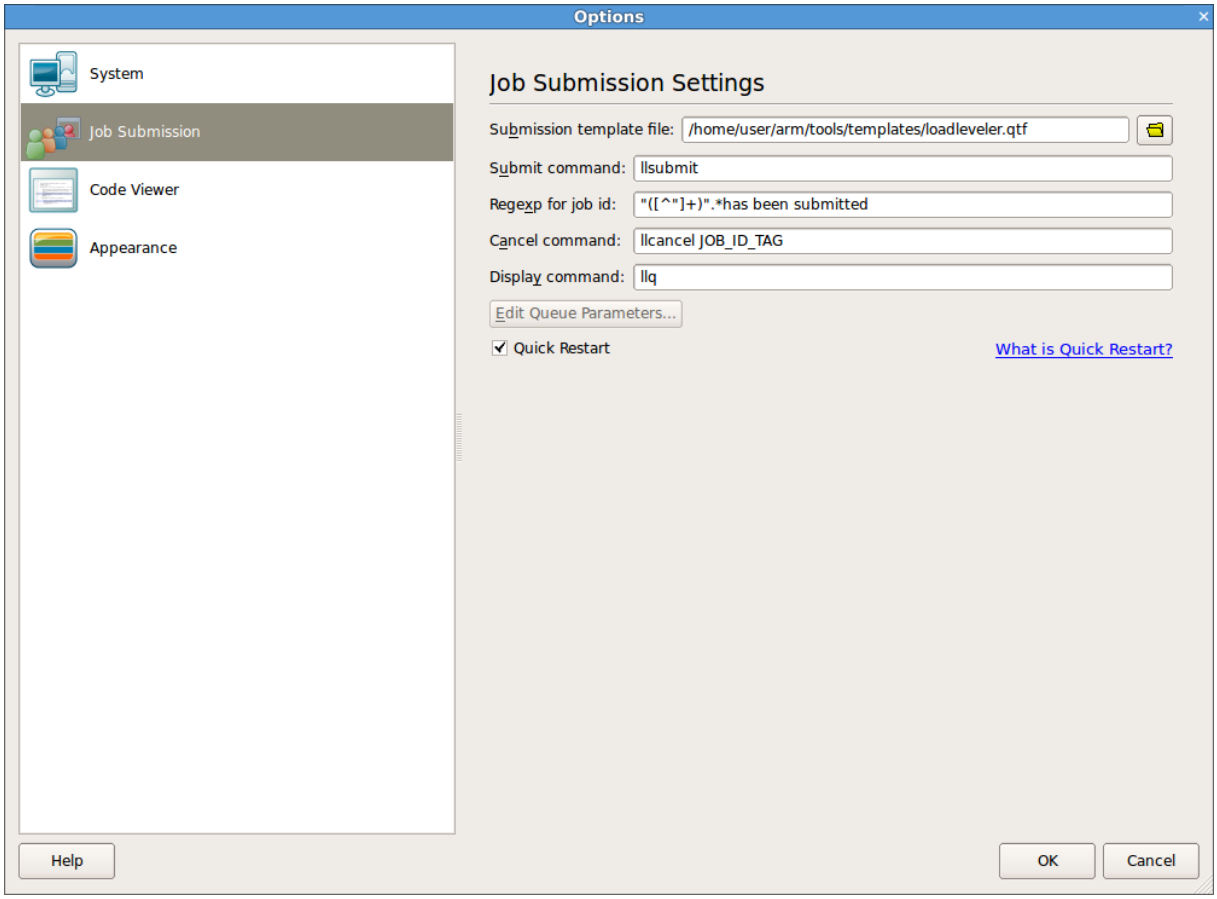

Figure 99: *MAP Using Substitute MPI Commands*

Note the Submit Command and the Submission Template File in particular. MAP 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/arm-temp-0112` or similar. Therefore, any argument like `-config` must be last on the line, because MAP will add a file name to the end of the line. Other arguments, if there are any, can come first.

Arm recommends reading the section on queue submission, as there are many features described there that might be useful to you if your system uses a non-standard start up command.

If you do use a non-standard command, please contact Arm at Arm support.

Starting MAP from a job script

While it is common when debugging to submit runs from inside a debugger, for profiling the usual approach would be to run the program offline, producing a profile file that can be inspected later.

To do this replace your usual program invocation with a MAP command such as:


```
mpirun -n 4 PROGRAM [ARGUMENTS]...
```

With either of the following examples:

```
map --profile mpirun -n 4 PROGRAM [ARGUMENTS]...
```

```
map --profile --np=4 PROGRAM [ARGUMENTS]...
```

MAP will run without a GUI, gathering data to a `.map` profile file. Its filename is based on a combination of program name, process count and timestamp, like `program_2p_2012-12-19_10-51.map`.

If using OpenMP, the value of `OMP_NUM_THREADS` is also included in the name after the process count, like `program_2p_8t_2014-10-21_12-45.map`.

This default name may be changed with the `--output` argument. To examine this file, either run MAP and select the *Load Profile Data File* option, or access it directly with the command:

```
map program_2p_2012-12-19_10-51.map
```

Note: When starting MAP for examining an existing profile file, a valid license is not needed.

When running without a GUI, MAP prints a short header and footer to stderr with your program's output in between. The `--silent` argument suppresses this additional output so that your program's output is intact.

As an alternative to `--profile` you can use Reverse Connect (see 3.3 Reverse Connect) to connect back to the GUI if you wish to use interactive profiling from inside the queue. So the above example becomes either:

```
map --connect mpirun -n 4 PROGRAM [ARGUMENTS]...
```

Or:

```
map --connect --np=4 PROGRAM [ARGUMENTS]...
```

Numactl

MAP supports launching programs via `numactl` for MPI programs. It works with or without SLURM. The recommended way to launch via `numactl` is to use express launch mode.

```
map mpiexec -n 4 numactl -m 1 ./myprogram.exe
map srun -n 4 numactl -m 1 ./myprogram.exe
```

It is also possible to launch via `numactl` using compatibility mode. If using compatibility mode, you need to put the full path to `numactl` in the Application box. If you do not know the full path to `numactl`, you can find it by running:

```
which numactl
```

Enter the name of the required application in the Arguments field, after all arguments to be passed to `numactl`. It is not possible to pass any more arguments to the parallel job runner when using this mode for launching.

MAP environment variables

ALLINEA_SAMPLER_INTERVAL

MAP takes a sample in each 20ms period, giving it a default sampling rate of 50Hz. This will be automatically decreased as the run proceeds to ensure a constant number of samples are taken. See `ALLINEA_SAMPLER_NUM_SAMPLES`.

If your program runs for a very short period of time, you may benefit by decreasing the initial sampling interval. For example, `ALLINEA_SAMPLER_INTERVAL=1` sets an initial sampling rate of 1000Hz, or once per millisecond. Higher sampling rates are not supported.

Increasing the sampling frequency from the default is not recommended if there are lots of threads and/or very deep stacks in the target program as this may not leave sufficient time to complete one sample before the next sample is started.

Note: Custom values for `ALLINEA_SAMPLER_INTERVAL` may be overwritten by values set from the combination of `ALLINEA_SAMPLER_INTERVAL_PER_THREAD` and the *expected* number of threads (from `OMP_NUM_THREADS`). For more information, see `ALLINEA_SAMPLER_INTERVAL_PER_THREAD`.

ALLINEA_SAMPLER_INTERVAL_PER_THREAD

To keep overhead low, MAP imposes a minimum sampling interval based on the number of threads. By default this is 2ms per thread, thus for eleven or more threads MAP will increase the initial sampling interval to more than 20ms.

You can adjust this behavior by setting `ALLINEA_SAMPLER_INTERVAL_PER_THREAD` to the minimum per-thread sample time in milliseconds.

Lowering this value from the default is not recommended if there are lots of threads as this may not leave sufficient time to complete one sample before the next sample is started.

Note: Whether OpenMP is enabled or disabled in MAP, the final script or scheduler values set for `OMP_NUM_THREADS` will be used to calculate the sampling interval per thread (`ALLINEA_SAMPLER_INTERVAL_PER_THREAD`). When configuring your job for submission, check whether your final submission script, scheduler or the MAP GUI has a default value for `OMP_NUM_THREADS`.

Note: Custom values for `ALLINEA_SAMPLER_INTERVAL` will be overwritten by values set from the combination of `ALLINEA_SAMPLER_INTERVAL_PER_THREAD` and the *expected* number of threads (from `OMP_NUM_THREADS`).

ALLINEA_MPI_WRAPPER

To direct MAP to use a specific wrapper library set `ALLINEA_MPI_WRAPPER=<pathofsharedobject>`.

MAP ships with a number of precompiled wrappers, when your MPI is supported MAP will automatically select and use the appropriate wrapper.

To manually compile a wrapper specifically for your system, set `ALLINEA_WRAPPER_COMPILE=1` and `MPICC` and run `<path to MAP installation>/map/wrapper/build_wrapper`.

This will generate the wrapper library `~/.allinea/wrapper/libmap-sampler-pmpi-<hostname>.so` with symlinks to the following files:

- `~/.allinea/wrapper/libmap-sampler-pmpi-<hostname>.so.1`
- `~/.allinea/wrapper/libmap-sampler-pmpi-<hostname>.so.1.0`
- `~/.allinea/wrapper/libmap-sampler-pmpi-<hostname>.so.1.0.0.`

ALLINEA_WRAPPER_COMPILE

To direct MAP to fall back to creating and compiling a just-in-time wrapper, set `ALLINEA_WRAPPER_COMPILE=1`.

In order to be able to generate a just-in-time wrapper an appropriate compiler must be available on the machine where MAP is running, or on the remote host when using remote connect.

MAP will attempt to auto detect your MPI compiler, however, setting the `MPICC` environment variable to the path to the correct compiler is recommended.

ALLINEA_MPIRUN

The path of `mpirun`, `mpiexec` or equivalent.

If this is set it has higher priority than that set in the GUI and the `mpirun` found in PATH.

ALLINEA_SAMPLER_NUM_SAMPLES

MAP collects 1000 samples per process by default. To avoid generating too much data on long runs, the sampling rate will be automatically decreased as the run progresses to ensure only 1000 evenly spaced samples are stored.

You may adjust this by setting `ALLINEA_SAMPLER_NUM_SAMPLES=<positiveinteger>`.

Note: It is strongly recommended that you leave this value at the default setting. Higher values are not generally beneficial and add extra memory overheads while running your code. Bear in mind that with 512 processes, the default setting already collects half a million samples over the job, the effective sampling rate can be very high indeed.

ALLINEA_KEEP_OUTPUT_LINES

Specifies the number of lines of program output to record in `.map` files. Setting to `0` will remove the line limit restriction, although this is not recommended as it may result in very large `.map` files if the profiled program produces lots of output.

See 17.3 Restricting output.

ALLINEA_KEEP_OUTPUT_LINE_LENGTH

The maximum line length for program output that will be recorded in `.map` files - lines containing more characters than this limit will be truncated. Setting to `0` will remove the line length restriction, although this is not recommended as it may result in very large `.map` files if the profiled program produces lots of output per line.

See 17.3 Restricting output.

ALLINEA_PRESERVE_WRAPPER

To gather data from MPI calls MAP generates a wrapper to the chosen MPI implementation. See 16.2 Preparing a program for profiling.

By default the generated code and shared objects are deleted when MAP no longer needs them.

To prevent MAP from deleting these files set `ALLINEA_PRESERVE_WRAPPER=1`.

Please note that if you are using remote launch then this variable must be exported in the remote script. See 3.2.1 Remote script.

ALLINEA_SAMPLER_NO_TIME_MPI_CALLS

Set this to prevent MAP from timing the time spent in MPI calls.

ALLINEA_SAMPLER_TRY_USE_SMAPS

Set this to allow MAP to use `/proc/[pid]/smaps` to gather memory usage data. This is not recommended since it slows down sampling significantly.

MPICC

To create the MPI wrapper MAP will try to use `MPICC`, then if that fails search for a suitable MPI compiler command in `PATH`. If the MPI compiler used to compile the target binary is not in `PATH` (or if there are multiple MPI compilers in `PATH`) then `MPICC` should be set.

Program output

MAP collects and displays output from all processes under the *Input/Output* tab. Both standard output and error are shown. As the output is shown after the program has completed, there are not the problems with buffering that occur with DDT.

Viewing standard output and error

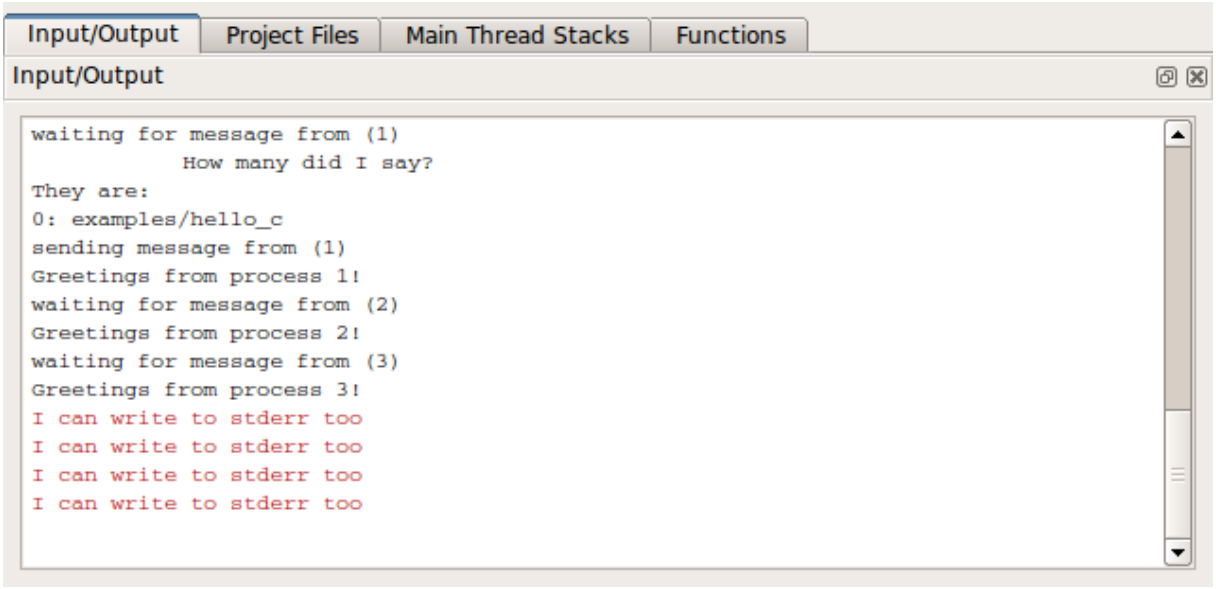

Figure 100: *MAP Standard Output Window*

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

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

Displaying selected processes

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

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

Restricting output

To keep file sizes within reasonable limits `.map` files will contain a summary of the program output limited to the first and last 500 lines (by default).

To change this number, profile with the environment variable `ALLINEA_KEEP_OUTPUT_LINES` set to the preferred total line limit (`ALLINEA_KEEP_OUTPUT_LINES=20` will restrict recorded output to the first 10 lines and last 10 lines).

Setting this to `0` will remove the line limit restriction, although this is not recommended as it may result in very large `.map` files if the profiled program produces lots of output.

The length of each line is similarly restricted to 2048 characters. This can be changed with the environment variable `ALLINEA_KEEP_OUTPUT_LINE_LENGTH`.

As before setting this to a value of `0` will remove the restriction, although this is not recommended as it risks a large `.map` file if the profiled program emits binary data or very long lines.

Saving output

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

Source code

Arm MAP provides code viewing, editing and rebuilding features. It also integrates with most major version control systems and provides static analysis to automatically detect many classes of common errors.

The code editing and rebuilding capabilities are not designed for developing applications from scratch, but they are designed to fit into existing profiling sessions that are running on a current executable.

The same capabilities are available for source code whether running remotely (using the remote client) or whether connected directly to your system.

Viewing

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

The source code viewer supports automatic color syntax highlighting for C and Fortran.

You can hide functions or subroutines you are not interested in by clicking the '−' glyph next to the first line of the function. This will collapse the function. Simply click the '+' glyph to expand the function again.

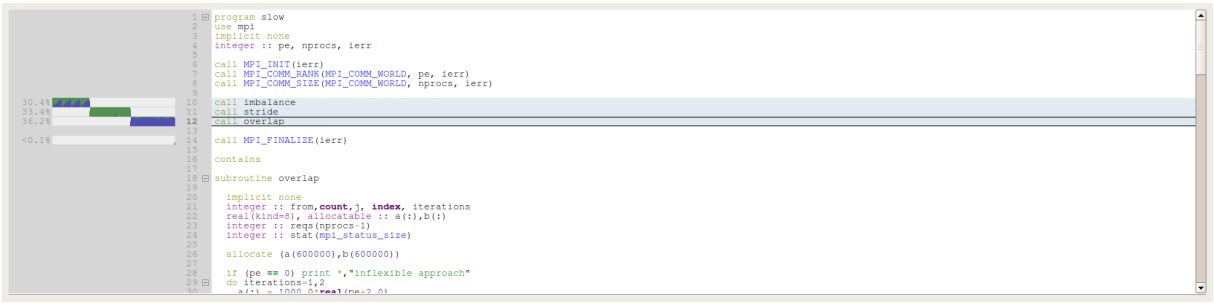

Figure 101: *Source Code View*

The centre pane shows your source code, annotated with performance information. All the charts you will see in MAP share a common horizontal time axis. The start of your job is at the left and the end at the right. The sparkline charts next to each line of source code shows how the number of cores executing that line of code varies over time.

What does it mean to say a core is executing a particular line of code? In the source code view, MAP uses *inclusive* time, that is time spent on this line of code or inside functions called by this line. So the `main()` function of a single-threaded C or MPI program is typically at 100% for the entire run.

Only 'interesting' lines get charts, that is, lines in which at least 0.1% of the selected time range was spent. In the previous figure you can see three different lines meet this criteria. The other lines were executed as well, but a negligible amount of time was spent on them.

The first line is a function call to `imbalance`, which was running for 18.1% of the wall-clock time. If you look closely, you will see that as well as a large block of green there is a sawtooth pattern in blue. Color is used to identify different kinds of time. In this single-threaded MPI code there are three colors:

- **Dark green** Single-threaded computation time. For an MPI program, this is all computation time. For an OpenMP or multi-threaded program, this is the time the main thread was active and no worker threads were active.
- **Blue** MPI communication and waiting time. All time spent inside MPI calls is blue, regardless of whether that is in `MPI_Send` or `MPI_Barrier`. Typically you want to minimize this, because the purpose of most codes is parallel *computation*, not communication for its own sake.
- **Orange** I/O time. All time spent inside known I/O functions such as reading and writing to the local or networked filesystem is shown in orange. You definitely want to minimize time spent in I/O and on many systems the complex data storage hierarchy can cause unexpected bottlenecks to occur when scaling a code up. MAP always shows the time from the *application's* point of view, so all the underlying complexity is captured and represented as simply as possible.
- **Dark purple** Accelerator. All the time the CPU is waiting the accelerator to return the control to the CPU. Typically you want to minimize this, making the CPU work in parallel with the accelerator using accelerator asynchronous calls.

In the above screenshot you can see the following:

- First a function called `imbalance` is called. This function spends most of its time in computation (dark green) and around 15–20% of it in MPI calls (blue). Hovering the mouse over any graph shows an exact breakdown of the time spent in it. There is a sawtooth pattern to the time spent in MPI calls that will be investigated later.
- Next the application moves on to a function called `stride`, which spends all of its time computing. You will see how to tell whether this time is well spent or not. You can also see an MPI synchronization at the end. The triangle shape is typical of ranks finishing their work at different times and spending varying amounts of time waiting at a barrier. Where you see triangles in these charts that indicates imbalance.
- Finally, a function called `overlap` is called, which spends almost all of its time in MPI calls.
- The other functions in this snippet of source code were active for <0.1% of the total runtime and can be ignored from a profiling point of view.

As this was an MPI program, the height of each block of color represents the percentage of MPI processes that were running each particular line at any moment in time. So the sawtooth pattern of MPI usage actually tells us that:

- The `imbalance` function goes through several *iterations*.
- In each iteration all processes start out computing, there is more green than blue.
- As execution continues more and more processes finish computing and transition to waiting in an MPI call, causing the distinctive triangular pattern showing workload imbalance.
- As each triangle ends all ranks finish communicating and the pattern begins again with the next iteration.

This is a classic sign of MPI imbalance. In fact, any triangular patterns in MAP's graphs show that first a few processes are changing to a different state of execution, then more, then more until they all synchronize and move on to another state together. These areas should be investigated.

You can explore this situation in more detail by opening the `examples/slow.map` file and looking at the `imbalance` function yourself. Can you see why some processes take longer to finish computing than others?

OpenMP programs

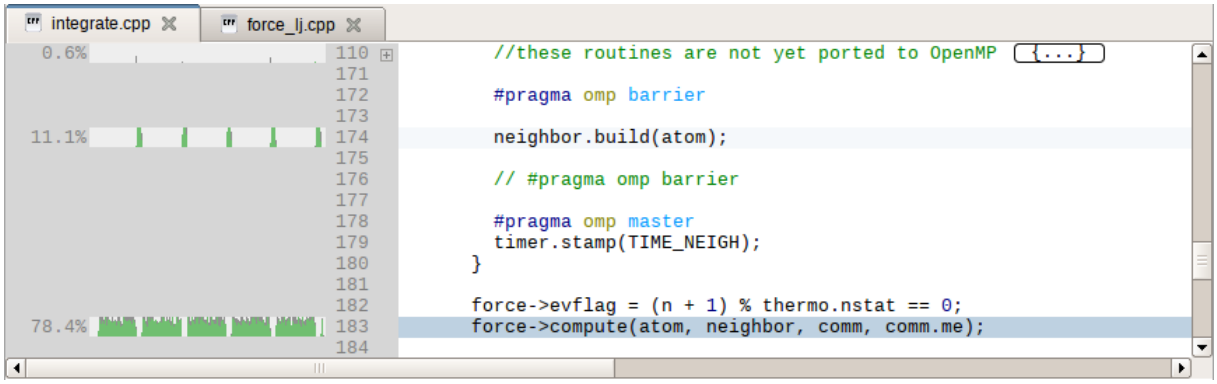

Figure 102: *OpenMP Source Code View*

In an OpenMP or multi-threaded program (or a mixed-mode MPI+OpenMP program) you will also see these colors used:

- **Light green** Multi-threaded computation time. For an OpenMP program this is time inside OpenMP regions. When profiling an OpenMP program you want to see as much light green as possible, because that is the only time you are using all available cores. Time spent in dark green is a potential bottleneck because it is serial code outside an OpenMP region.
- **Light blue** Multi-threaded MPI communication time. This is MPI time spent waiting for MPI communication while inside an OpenMP region or on a pthread. As with the normal blue MPI time you will want to minimize this, but also maximize the amount of multi-threaded computation (light green) that is occurring on the other threads while this MPI communication is taking place.
- **Dark Gray** Time inside an OpenMP region in which a core is idle or waiting to synchronize with the other OpenMP threads. In theory, during an OpenMP region all threads are active all of the time. In practice there are significant synchronization overheads involved in setting up parallel regions and synchronizing at barriers. These will be seen as dark gray holes in the otherwise happy light green of optimal parallel computation. If you see these there may be an opportunity to improve performance with better loop scheduling or division of the work to be done.
- **Pale blue** Thread synchronization time. Time spent waiting for synchronization between non-OpenMP threads (for example, a `pthread_join`). Whether this time can be reduced depends on the purpose of the threads in question.

In the screenshot above you can see that 11.1% of the time is spent calling `neighbor.build(atom)` and 78.4% of the time is spent calling `force->compute(atom, neighbor, comm, comm.me)`. The graphs show a mixture of **light green** indicating an OpenMP region and **dark gray** indicating OpenMP overhead. OpenMP overhead is the time spent in OpenMP that is not the contents of an OpenMP region (user code). Hovering the mouse over a line will show the exact percentage of time spent in overhead, but visually you can already see that it is significant but not dominant here.

Increasingly, programs use both MPI and OpenMP to parallelize their workloads efficiently. MAP fully and transparently supports this model of working. It is important to note that the graphs are a reflection of the `application` activity over time:

- A large section of blue in a mixed-mode MPI code means that all the processes in the application were inside MPI calls during this period. Try to reduce these, especially if they have a triangular shape suggesting that some processes were waiting inside MPI while others were still computing.

- A large section of dark green means that all the processes were running single-threaded computations during that period. Avoid this in an MPI+OpenMP code, or you might as well leave out the OpenMP sections altogether.
- Ideally you want to achieve large sections of light green, showing OpenMP regions being effectively used across all processes simultaneously.
- It is possible to call MPI functions *from within* an OpenMP region. MAP only supports this if the main thread (the OpenMP master thread) is the one that makes the MPI calls. In this case, the blue block of MPI time will be smaller, reflecting that *one* OpenMP thread is in an MPI function while the rest are doing something else such as useful computation.

GPU programs

In a program using NVIDIA CUDA CPU, time spent waiting for GPU kernels to complete is shown in **Purple**.

When CUDA kernel analysis mode is enabled (see Section 30) MAP will display also display data for lines inside CUDA kernels. These graphs show when GPU kernels were active, and for each kernel a breakdown of the different types of warp stalls that occurred on that line. The different types of warp stalls are listed in Section 30.1. Refer to the tooltip or selected line display (Section 19.2) to get the exact breakdown, but in general:

- **Purple** Selected. Instructions on this line were being executed on the GPU.
- **Dark Purple** Not selected. This means warps on this line were ready to execute but that there was no available SM to do the executing.
- **Red** (various shades) Memory operations. Warps on this line were stalled waiting for some memory dependency to be satisfied. Shade of red indicates the type of memory operation.
- **Blue** (various shades) Execution dependency. Warps on this line were stalled until some other action completes. Shade of blue indicates the type of execution dependency.

Note that warp stalls are only reported per-kernel, so it is not possible to obtain the times within a kernel invocation at which different categories of warp stalls occurred. As function calls in CUDA kernels are also automatically fully inlined it is not possible to see warp stalls for 'time spent inside function(s) on line' for GPU kernel code.

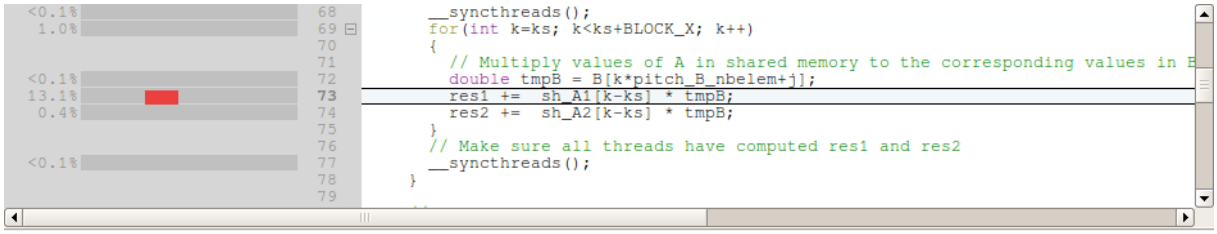

Figure 103: *Source Code View (GPU Kernel)*

In this screenshot a CUDA kernel involving this line was running on this line 13.1% of the time, with most of the warps waiting for a memory access to complete. The colored horizontal range indicates when any kernel observed to be using this source line was on the GPU. The height of the colored region indicates the proportion of sampled warps that were observed to be on this line. See the NVIDIA CUPTI documentation at http://docs.nvidia.com/cuda/cupti/r_main.html#r_pc_sampling for more information on how warps are sampling.

Dealing with complexity: code folding

Real-world scientific codes do not look much like the examples above. They tend to look more like the following:

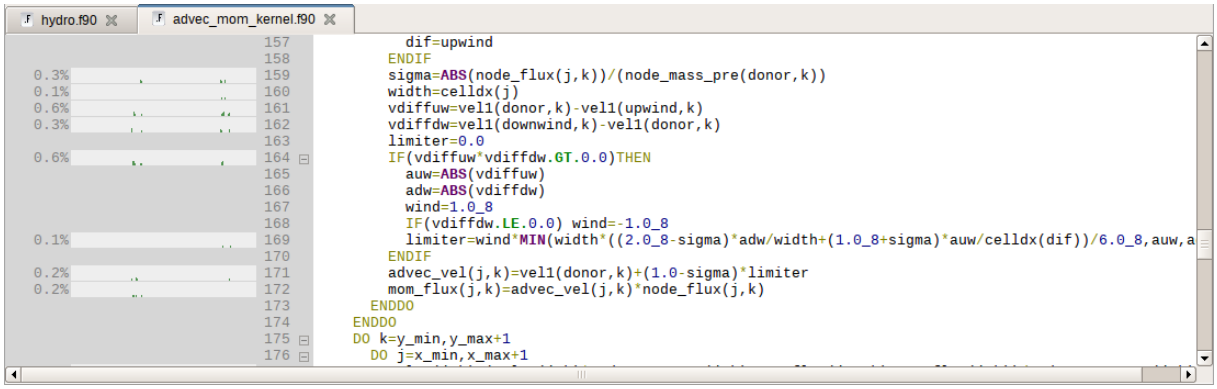

Figure 104: *Typical Fortran Code in MAP*

Here, small amounts of processing are distributed over many lines, and it is difficult to see which parts of the program are responsible for the majority of the resource usage.

To understand the performance of complex blocks of code like this, MAP allows supports *code folding*. Each logical block of code such as an if-statement or a function call has a small [–] next to it. Clicking this *folds* those lines of code into one and shows one single sparkline for the entire block:

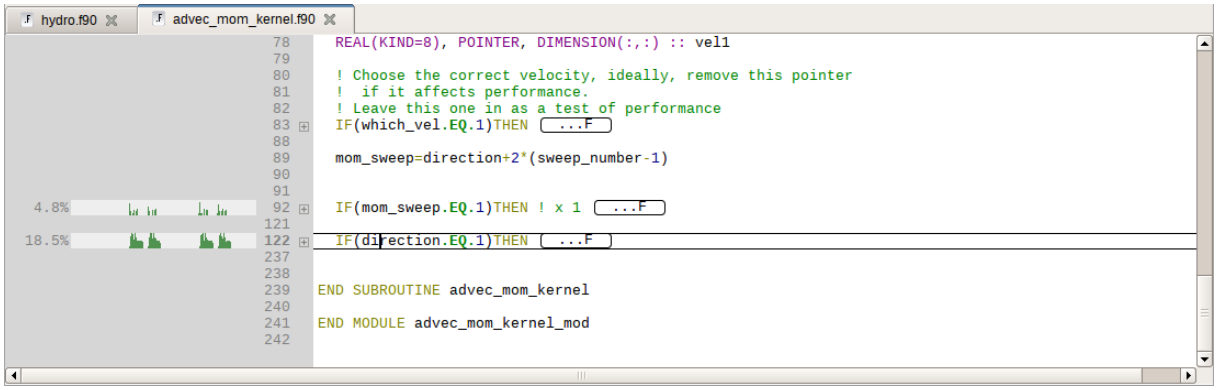

Figure 105: *Folded Fortran Code in MAP*

Now you can clearly see that most of the processing occurs within the conditional block starting on line 122.

When exploring a new source file, a good way to understand its performance is to use the `View->Fold All` menu item to collapse all the functions in the file to single lines, then scroll through it looking for functions that take an unusual amount of time or show an unusual pattern of I/O or MPI overhead. These can then be expanded to show their most basic blocks, and the largest of these can be expanded again and so on.

Editing

Source code may be edited in the code viewer windows of MAP. The actions *Undo*, *Redo*, *Cut*, *Copy*, *Paste*, *Select all*, *Go to line*, *Find*, *Find next*, *Find previous*, and *Find in files* are available from the *Edit*

menu.

Files may be opened, saved, reverted and closed from the *File* menu.

Note that information from MAP will not match edited source files until the changes are saved, the binary is rebuilt, and a new profile is recreated.

If the currently selected file has an associated header or source code file, it can be opened by right-clicking in the editor and choosing *Open <filename>.<extension>*. There is a global shortcut on function key F4, available in the *Edit* menu as *Switch Header/Source* option.

To edit a source file in an external editor, right-click the editor for the file and choose *Open in external editor*. To change the editor used, or if the file does not open with the default settings, open the Options window by selecting *File → Options* (*Arm Forge → Preferences* on Mac OS X) and enter the path to the preferred editor in the Editor box, for example `/usr/bin/gedit`.

If a file is edited the following warning is displayed at the top of the editor.

Figure 106: *File Edited Warning*

This is to warn you that the source code shown is not the source that was used to produced the currently executing binary, so the source code and line numbers may not match the executing code.

Rebuilding and restarting

To configure the build command choose *File → Configure Build…*, enter a build command and a directory in which to run the command, and click *Apply*.

To issue the build command choose *File → Build*, or press Ctrl+B (Cmd+B on Mac OS X). When a build is issued the *Build Output* view is shown.

Committing changes

Changes to source files may be committed using one of Git, Mercurial, and Subversion. To commit changes choose *File → Commit…*, enter a commit message to the resulting dialog and click the commit button.

Selected lines view

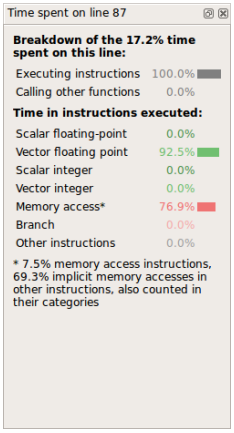

Figure 107: *Selected Lines View*

Note: The selected lines view is currently only available for profiles generated on x86_64 systems.

The *Selected Lines View* view allows you to get detailed information on how one or more lines of code are spending their time.

To access this view, open one of your program's source files in the code viewer and highlight a line.

The Selected Lines View, which is by default shown on the right hand side of the source view, automatically updates to show a detailed breakdown of how the selected lines are spending their time.

You can select multiple lines, and MAP will show information for all of the lines together.

You can also select the first line of a collapsed region to see information for the entire code block. See section 18.1 for more information.

If you use the metrics view to select a region of time, the selected lines view only shows details for the highlighted region. See section 24 for more information.

The panel is divided into two sections.

The first section gives an overview of how much time was spent executing instructions on this line, and how much time was spent in other functions.

If the time spent executing instructions is low, consider using the stacks view, or the functions view to locate functions that are using a lot of CPU time. For more information on the Stacks View see section 20. For more information on the Functions View see section 22.

The second section details the CPU instruction metrics for the selected line.

These largely show the same information as the global program metrics, described in section 24.1, but for the selected lines of source code.

Unlike the global program metrics, the line metrics are divided into separate entries for scalar and vector operations, and report time spent in "implicit memory accesses".

On some architectures, computational instructions (such as integer or vector operations) are allowed to access memory implicitly. When these types of instruction are used, MAP cannot distinguish between time performing the operation and time accessing memory, and therefore reports time for the instruction in both the computational category and the memory category.

The amount of time spent in “explicit” and “implicit” memory accesses is reported as a footnote to the time spent executing instructions.

Some guidelines are listed here:

- In general, aim for a large proportion of time in vector operations.
- If you see a high proportion of time in scalar operations, try checking to see if your compiler is correctly optimising for your processor’s SIMD instructions.
- If you see a large amount of time in memory operations then look for ways to more efficiently access memory in order to improve cache performance.
- If you see a large amount of time in branch operations then look for ways to avoid using conditional logic in your inner loops.

Section 24.1 offers detailed advice on what to look for when optimizing the types of instruction your program is executing.

Limitations

Modern superscalar processors use instruction-level parallelism to decode and execute multiple operations in a single cycle, if internal CPU resources are free, and will retire multiple instructions at once, making it appear as if the program counter “jumps” several instructions per cycle.

Current architectures do not allow profilers such as MAP (or Intel VTune, Linux perftools and others) to efficiently measure which instructions were “invisibly” executed by this instruction-level parallelism. This time is typically allocated to the last instruction executed in the cycle.

Most MAP users will not be affected by this for the following reasons:

1. Hot lines in a HPC code typically contain rather more than a single instruction such as `nop`. This makes it unlikely that an entire source line will be executed invisibly via the CPU’s instruction-level parallelism.
2. Any such lines executed “for free” in parallel with another line by a CPU core will clearly show up as a “gap” in the source code view (but this is unusual).
3. Loops with stalls and mispredicted branches still show up highlighting the line containing the problem in all but the most extreme cases.

To summarize key points:

- Experts users: those wanting to use MAP’s per-line instruction metrics to investigate detailed CPU performance of a loop or kernel (even down to the assembly level) should be aware that instructions executed in parallel by the CPU will show up with time only assigned to the last one in the batch executed.
- Other users: MAP’s statistical instruction-based metrics correlate well with where time is spent in the application and help to find areas for optimization. Feel free to use them as such. If you see lines with very few operations on them (such as a single add or multiply) and no time assigned to them inside your hot loops then these are probably being executed “for free” by the CPU using instruction-level parallelism. The time for each batch of such is assigned to the last instruction completed in the cycle instead.

GPU profiling

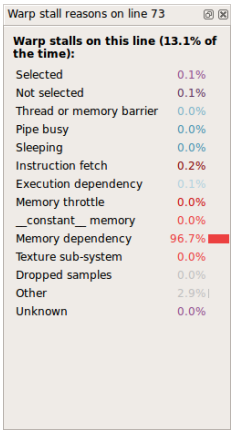

Figure 108: *Selected lines view (GPU kernel)*

When CUDA kernel analysis is enabled (see section 30) and the selected line is executed on the GPU then a breakdown of warp stall reasons on this line will be shown in this view. For a description of each of these warp stall reasons, refer to the tooltip for each of the entries or section 30.1.

Stacks view

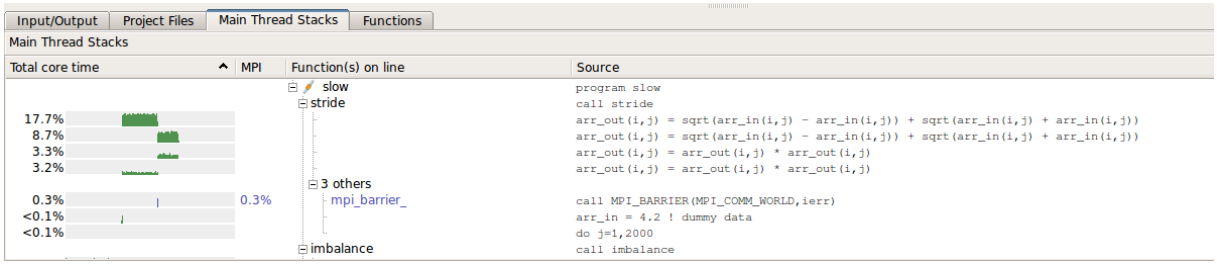

Figure 109: *MAP Stacks View*

The *Stacks* view offers a good top-down view of your program. It is easy to follow down from the main function to see which code paths took the most time. Each line of the *Stacks* view shows the performance of one line of your source code, including all the functions called by that line.

The sparkline graphs are described in detail in section 18.

You can read the above figure as follows:

1. The first line, `slow`, represents the entire program run. Collapsing this node The first line, `program slow`, represents the entire program.
2. Beneath it, you see a call to the `stride` function, almost all of which was in single-threaded compute (dark green). 1.4% of the time was spent in MPI.
3. The next major function called from program slow is the overlap function, seen at the bottom of this figure. A more detailed breakdown is described in section 24. The `stride` function itself spent most of that time on the line `a(i,j)=x*j` at slow.f90 line 107. In fact, 43.2% of the entire run was spent executing this line of code.
4. The 1% MPI time inside `stride` comes from an `MPI_Barrier` on line 124.
5. The next major function called from `program slow` is the `overlap` function, seen at the bottom of this figure. This function ran for 24.8% of the total time, almost all of which was runtime. This line of code was executed at the start of the overlap function, and other calls which are not visible in the figure accounted for the rest.

Clicking on any line of the *Stacks* view jumps the *Source Code* view to show that line of code. This makes it a very easy way to navigate and understand the performance of even complex codes.

The percentage MPI time gives an idea as to how well your program is scaling and shows the location of any communication bottlenecks. As you discussed in section 18, any sloping blue edges represent imbalance between processes or cores.

In the above example you can see that the `MPI_Send` call inside the `overlap` function has a sloping trailing edge. This means that some processes took significantly longer to finish the call than others, perhaps because they were waiting longer for their receiver to become ready.

Stacks view shows which lines of code spend the most time running, computing or waiting. As with most places in the GUI you can hover over a line or chart for a more detailed breakdown.

OpenMP Regions view

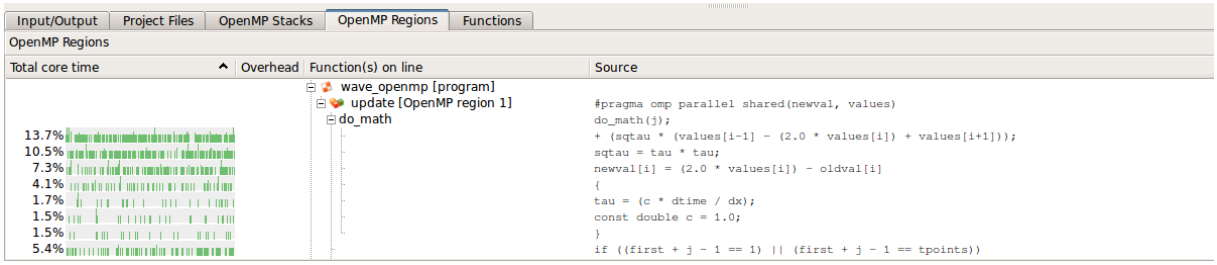

Figure 110: *OpenMP Regions View*

The *OpenMP Regions* view gives insight into the performance of every significant OpenMP region in your program. Each region can be expanded just as in the Stacks view to see the performance of every line beneath it across every core in your job. The sparkline graphs are described in detail in section 18.

Note: If you are using MPI and OpenMP, this view summarizes all cores across all nodes and not just one node.

You can read the above figure as follows:

1. The most time-consuming parallel region is in the `update` function at line 207. Clicking on this shows the region in the *Source Code* view.
2. This region spends most of its time in the `do_math` function. Hovering on the line or clicking on the [–] symbol collapses the view down to show the figures for how much time.
3. Of the lines of code inside `do_math`, the `(sqtau * (values[i-1] ...)` one takes longest with 13.7% of the total core hours across all cores used in the job.
4. Calculating `sqtau = tau * tau` is the next most expensive line, taking 10.5% of the total core hours.
5. Only 0.6% of the time in this region is spent on OpenMP overhead, such as starting/synchronizing threads.

From this you can see that the region is optimized for OpenMP usage, that is, it has very low overhead. If you want to improve performance you can look at the calculations on the lines highlighted in conjunction with the CPU instruction metrics, in order to answer the following questions:

- Is the current algorithm is bound by computation speed or memory accesses? If the latter, you may be able to improve cache locality with a change to the data structure layout.
- Has the compiler generated optimal vectorized instructions for this routine? Small things can prevent the compiler doing this and you can look at the vectorization report for the routine to understand why.
- Is there another way to do this calculation more efficiently now that you know which parts of it are the most expensive to run?

See section 24 for more information on CPU instruction metrics.

Clicking on any line of the *OpenMP Regions* view jumps the *Source Code* view to show that line of code.

The percentage OpenMP synchronization time gives an idea as to how well your program is scaling to multiple cores and highlights the OpenMP regions that are causing the greatest overhead. Examples of

things that cause OpenMP synchronization include:

- Poor load balancing, for example, some threads have more work to do or take longer to do it than others. The amount of synchronization time is the amount of time the fastest-finishing threads wait for the slowest before leaving the region. Modifying the OpenMP chunk size can help with this.
- Too many barriers. All time at an OpenMP barrier is counted as synchronization time. However, `omp atomic` does *not* appear as synchronization time. This is generally implemented as a locking modifier to CPU instructions. Overuse of the `atomic` operator shows up as large amounts of time spent in *memory accesses* and on lines immediately following an `atomic` pragma.
- Overly fine-grained parallelization. By default OpenMP synchronizes threads at the start and end of each parallel region. There is also some overhead involved in setting up each region. In general, the best performance is achieved when outer loops are parallelized rather than inner loops. This can also be alleviated by using the `no_barrier` OpenMP keyword if appropriate.

When parallelizing with OpenMP it is extremely important to achieve good single-core performance first. If a single CPU core is already bottlenecked on memory bandwidth, splitting the computations across additional cores rarely solves the problem.

Functions view

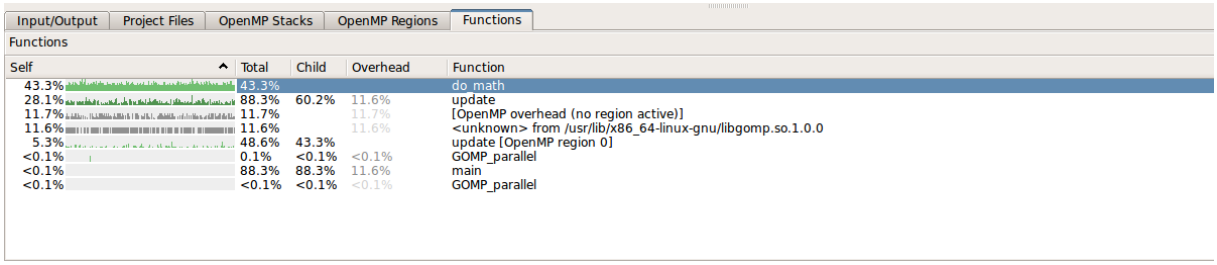

Figure 111: *Functions View*

The *Functions* view shows a flat profile of the functions in your program. The first three columns show different measures of the time spent in a given function:

1. *Self* shows the time spent in code in the given function itself, but not its callees, that is, not in the other functions called by that function.
2. *Total* shows the time spent in code in the given function itself, and all its callees.
3. *Child* shows the time spent in the given functions's callees only.

You can use the *Functions* view to find costly functions that are called from many different places.

Project Files view

Figure 112: *Project files view*

The *Project Files* view offers an effective way to browse around and navigate through a large, unfamiliar code base.

The project files view distinguishes between *Application Code* and *External Code*. You can choose which folders count as application code by right-clicking. *External Code* is typically system libraries that are hidden away at startup.

Metrics View

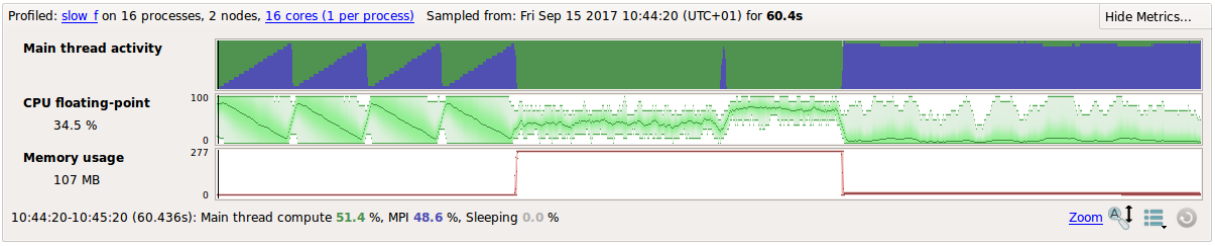

Figure 113: *Metrics view*

Now that you are familiar with the source code, the stacks and the project files view, you will now see how the metrics view works with all of these to help you identify, focus on and understand performance problems.

As with all graphs in MAP, the horizontal axis is wall clock time. By default three metric graphs are shown. The top-most is the *main thread activity* chart, which uses the same colors and scales as the per-line sparkline graphs described in section 18. To understand the *main thread activity* chart, read that section first.

For CUDA programs profiled with CUDA kernel analysis mode enabled a “warp stall reasons” graph is also displayed. This shows the warp stalls for all CUDA kernels detected in the program, using the same colors and scales as the GPU kernel graphs described in section 30.1). To understand the *warp stall reason* chart, read that section first.

All of the other metric graphs show how single numerical measurements vary across processes and time. Initially, two frequently used ones are shown: *CPU floating-point* and *memory usage*. However, there are many other metric graphs available, and they can all be read in the same way. Each vertical slice of a graph shows the distribution of values across processes for that moment in time. The minimum and maximum are clear, and shading is used to display the mean and standard deviation of the distribution.

A thin line means all processes had very similar values. A ‘fat’ shaded region means there is significant imbalance between the processes. Extra details about each moment in time appear below the metric graphs as you move the mouse over them.

The metrics view is at the top of the GUI as it ties all the other views together. Move your mouse across one of the graphs, and a black vertical line appears on every other graph in MAP, showing what was happening at that moment in time.

You can also click and drag to select a region of time within it. All the other views and graphs now redraw themselves to show just what happened during the selected period of time, ignoring everything else. This is a useful way to isolate interesting parts of your application’s execution. To reselect the entire time range just double-click or use the *Select All* button.

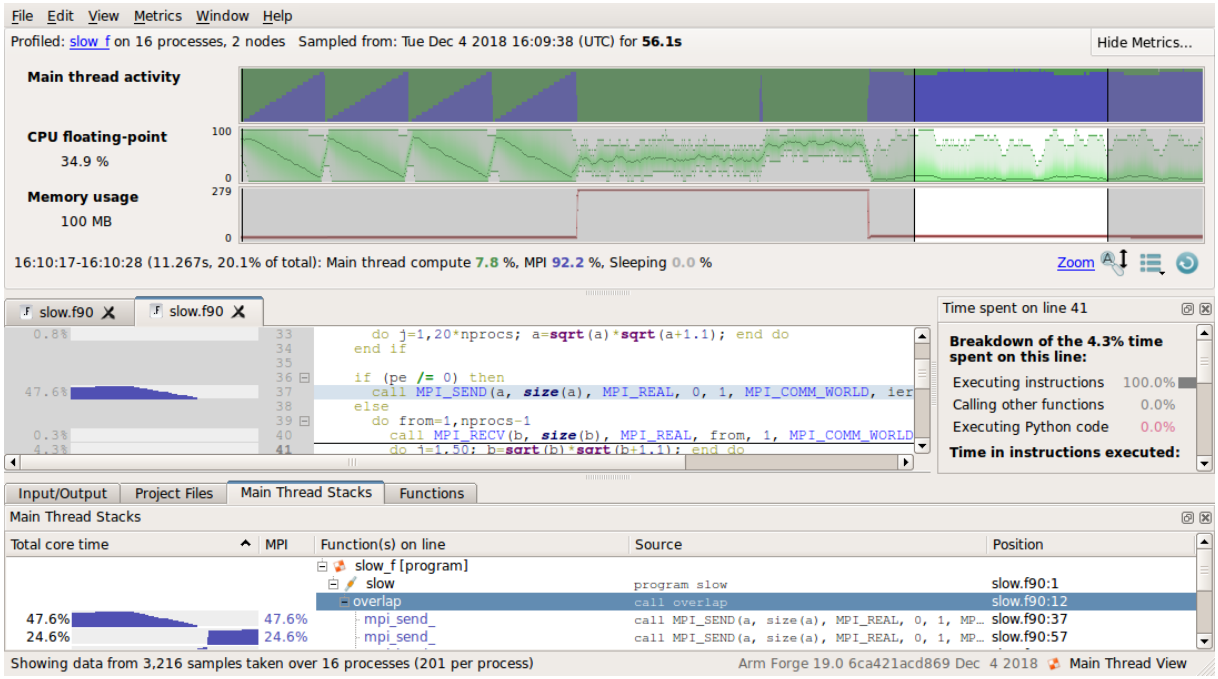

Figure 114: *Map with a region of time selected*

In the above screenshot a short region of time has been selected around an interesting sawtooth in time in `MPI_BARRIER` because PE 1 is causing delays. The first block accepts data in PE order, so is badly delayed, the second block is more flexible, accepting data from any PE, so PE 1 can compute in parallel. The *Code View* shows how compute and comms are serialized in the first block, but overlap in the second.

There are many more metrics other than those displayed by default. Click the *Metrics* button or right-click on the metric graphs and you can choose one of the following presets or any combination of the metrics beneath them. You can return to the default set of metrics at any time by choosing the *Preset: Default* option.

CPU instructions

Note: All of the metrics described in this section are only available on x86_64 systems.

All of these metrics show the percentage of time active cores spent executing different classes of instruction. They are most useful for optimizing single-core and OpenMP performance:

CPU floating-point: The percentage of time each rank spends in floating-point CPU instructions. This includes vectorized instructions and standard x87 floating-point. High values here suggest CPU-bound areas of the code that are probably functioning as expected.

CPU integer: The percentage of time each rank spends in integer CPU instructions. This includes vectorized instructions and standard integer operations. High values here suggest CPU-bound areas of the code that are probably functioning as expected.

CPU memory access: The percentage of time each rank spends in memory access CPU instructions, such as `move`, `load` and `store`. This also includes vectorized memory access functions. High values here may indicate inefficiently-structured code. Extremely high values (98% and above) almost always indicate cache problems. Typical cache problems include cache misses due to incorrect loop orderings but may also include more subtle features such as false sharing or cache line collisions.

CPU floating-point vector: The percentage of time each rank spends in vectorized floating-point instructions. Optimized floating-point-based HPC code should spend most of its time running these operations. This metric provides a good check to see whether your compiler is correctly vectorizing hotspots. See section H.6 for a list of the instructions considered vectorized.

CPU integer vector: The percentage of time each rank spends in vectorized and integer instructions. Optimized integer-based HPC code should spend most of its time running these operations. This metric provides a good check to see whether your compiler is correctly vectorizing hotspots. See section H.6 for a list of the instructions considered vectorized.

CPU branch: The percentage of time each rank spends in test and branch-related instructions such as `test`, `cmp` and `je`. An optimized HPC code should not spend much time in branch-related instructions. Typically the only branch hotspots are during MPI calls, in which the MPI layer is checking whether a message has been fully-received or not.

Per-line CPU instructions

When you select one or more lines of code in the code view, MAP will show a breakdown of the CPU Instructions used on those lines. Section 19 describes this view in more detail.

Perf metrics

This section presents key CPU performance measurements gathered using the Linux perf event subsystem.

Due to differences in processor models, some metrics might be unavailable on your system. MAP displays all available metrics and where metrics are not available error messages are displayed.

Note: Metrics described in this section are only available on Armv8 and IBM Power systems. These metrics are not available on virtual machines. Linux perf events performance events counters must be accessible on all systems on which the target program runs. See section G.6.1 or G.7.2 for details.

Cycles per instruction The average amount of CPU cycles lapsed for each retired instruction. This metric is calculated from the CPU cycles and Instructions Perf metrics.

CPU FLOPS lower bound The rate at which floating-point operations completed.

Note: This is a lower bound because the counted value does not account for the length of vector operations. Note: This metric is available on IBM Power 8 systems only.

CPU Memory Accesses The processor's data cache was reloaded from local, remote, or distant memory due to a demand load.

Note: This metric is available on IBM Power systems only.

CPU FLOPS vector lower bound The rate at which vector floating-point instructions completed.

Note: This is a lower bound because the counted value does not account for the length of vector operations. Note: This metric is available on IBM Power 8 systems only.

CPU cycles The rate at which CPU cycles lapsed.

Note: This metric is affected by CPU frequency scaling. Note: This metric is available on Armv8 systems only.

Instructions The rate at which instructions retired.

Note: This metric is affected by various issues, most notably hardware interrupt counts. Note: This metric is available on Armv8 systems only.

L2 cache accesses The rate of level 2 data cache accesses. This includes memory-write and memory-read operations that access the level 2 data or unified cache.

Note: This metric is available on Armv8 systems only.

L2 cache misses The rate of level 2 data cache refills.

Note: This metric is available on Armv8 systems only.

CPU branch mispredictions The rate of mispredicted branch instructions. This counts the number of incorrectly predicted retired branches that are conditional, unconditional, branch and link, return or eret.

Note: This metric is available on Armv8 systems and IBM Power 8 systems only.

Stalled backend cycles The percentage of CPU cycles that lapsed, on which operation instructions were not issued even though instructions were available from the fetch unit.

Note: This metric is available on Armv8 systems only.

Stalled frontend cycles The percentage of CPU cycles that lapsed, on which operation instructions were not issued because operation instructions were not available in the fetch unit.

Note: This metric is available on Armv8 systems only.

Stalled cycles The percentage of CPU cycles that lapsed, on which operation instructions were not issued.

Note: This metric is available on Armv8 systems only.

Non-stalled cycles The percentage of CPU cycles that lapsed, on which operation instructions were issued.

Note: This metric is available on Armv8 systems only.

CPU time

These metrics are particularly useful for detecting and diagnosing the impact of other system daemons on your program's run.

CPU time This is the percentage of time that each thread of your program was able to spend on a core. Together with *Involuntary context switches*, this is a key indicator of oversubscription or interference from system daemons. If this graph is consistently less than 100%, check your core count and CPU affinity settings to make sure one or more cores are not being oversubscribed. If there are regular spikes in this graph, show it to your system administrator and ask for their help in diagnosing the issue.

User-mode CPU time The percentage of time spent executing instructions in user-mode. This should be close to 100%. Lower values or spikes indicate times in which the program was waiting for a system call to return.

Kernel-mode CPU time Complements the above graph and shows the percentage of time spent inside system calls to the kernel. This should be very low for most HPC runs. If it is high, show the graph to your system administrator and ask for their help in diagnosing the issue.

Voluntary context switches The number of times per second that a thread voluntarily slept, for example while waiting for an I/O call to complete. This is normally very low for a HPC code.

Involuntary context switches The number of times per second that a thread was interrupted while computing and switched out for another one. This will happen if the cores are *oversubscribed*, or if other system processes and daemons start running and take CPU resources away from your program. If this graph is consistently high, check your core count and CPU affinity settings to make sure one or more cores are not being oversubscribed. If there are regular spikes in this graph, show it to your system administrator and ask for their help in diagnosing the issue.

System load The number of active (running or runnable) threads as a percentage of the number of physical CPU cores present in the compute node. This value may exceed 100% if you are using hyperthreading, if the cores are *oversubscribed*, or if other system processes and daemons start running and take CPU resources away from your program. A value consistently less than 100% may indicate your program is not taking full advantage of the CPU resources available on a compute node.

I/O

These metrics show the performance of the I/O subsystem from the application's point of view. Correlating these with the I/O time in the *Application Activity* chart helps to diagnose I/O bottlenecks.

POSIX I/O read rate: The total I/O read rate of the application. This may be greater than *Disk read transfer* if data is read from the cache instead of the storage layer.

POSIX I/O write rate: The total I/O write rate of the application. This may be greater than *Disk write transfer* if data is written to the cache instead of the storage layer.

Disk read transfer: The rate at which the application reads data from disk, in bytes per second. This includes data read from network filesystems (such as NFS), but may not include all local I/O due to page caching.

Disk write transfer: The rate at which the application writes data to disk, in bytes per second. This includes data written to network filesystems.

POSIX read syscall rate: The rate at which the application invokes the `read` system call. Measured in calls per second, not the amount of data transferred.

POSIX write syscall rate: The rate at which the application invokes the `write` system call. Measured in calls per second, not the amount of data transferred.

Note: Disk transfer and I/O metrics are not available on Cray X-series systems as the necessary Linux kernel support is not enabled.

Note: I/O time in the Application Activity chart done via direct kernel calls will not be counted.

Note: Even if your application does not perform I/O, a non-zero amount of I/O will be recorded at the start of profile because of internal I/O performed by MAP.

Memory

Here the memory usage of your application is shown in both a per-process and per-node view. Performance degrades severely once all the node memory has been allocated and swap is required. Some HPC systems, notably Crays, will terminate a job that tries to use more than the total node memory available.

Memory usage: The memory in use by the processes currently being profiled. Memory that is allocated and never used is generally not shown. Only pages actively swapped into RAM by the OS are displayed. This means that you will often see memory usage ramp up as arrays are initialized. The slopes of these ramps can be interesting in themselves.

Note: this means if you `malloc` *or* `ALLOCATE` *a large amount of memory but do not actually use it the* ***Memory Usage*** *metric will not increase.*

Node memory usage: The percentage of memory in use by all processes running on the node, including operating system processes and user processes not in the list of selected ranks when specifying a subset of processes to profile. If node memory usage is far below 100% then your code may run more efficiently using fewer processes or a larger problem size. If it is close to or reaches 100% then the combination of your code and other system daemons are exhausting the physical memory of at least one node.

MPI

A detailed range of metrics offering insight into the performance of the MPI calls in your application. These are all per-process metrics and any imbalance here, as shown by large blocks with sloped means, has serious implications for scalability.

Use these metrics to understand whether the blue areas of the *Application Activity* chart are problematic or are transferring data in an optimal manner. These are all seen from the application's point of view.

An asynchronous call that receives data in the background and completes within a few milliseconds will have a much higher effective transfer rate than the network bandwidth. Making good use of asynchronous calls is a key tool to improve communication performance.

In multithreaded applications, MAP only reports MPI metrics for MPI calls from main threads. If an application uses `MPI_THREAD_SERIALIZED` or `MPI_THREAD_MULTIPLE`, the *Application Activity* chart will show MPI activity, but some regions of the MPI metrics may be empty if the MPI calls are from non-main threads.

MPI call duration: This metric tracks the time spent in an MPI call so far. PEs waiting at a barrier (MPI blocking sends, reductions, waits and barriers themselves) will ramp up time until finally they escape. Large areas show lots of wasted time and are prime targets for investigation. The PE with no time spent in calls is likely to be the last one to arrive, so should be the focus for any imbalance reduction.

MPI sent/received: This pair of metrics tracks the number of bytes passed to MPI send/receive functions per second. This is not the same as the speed with which data is transmitted over the network, as that information is not available. This means that an MPI call that receives a large amount of data and completes almost instantly will have an unusually high instantaneous rate.

MPI point-to-point and collective operations: This pair of metrics tracks the number of point-to-point and collective calls per second. A long shallow period followed by a sudden spike is typical of a late sender. Most processes are spending a long time in one MPI call (very low #calls per second) while one computes. When that one reaches the matching MPI call it completes much faster, causing a sudden spike in the graph.

MPI point-to-point and collective bytes: This pair of metrics tracks the number of bytes passed to MPI send and receive functions per second. This is not the same as the speed with which data is transmitted over the network, as that information is not available. This means that an MPI call that receives a large amount of data and completes almost instantly will have an unusually high instantaneous rate.

Note: (for SHMEM users) MAP shows calls to `shmem_barrier_all` *in* ***MPI collectives****,* ***MPI calls*** *and* ***MPI call duration****. Metrics for other SHMEM functions are not collected.*

Detecting MPI imbalance

The metrics view shows the distribution of their value across all processes against time, so any 'fat' regions are showing an area of imbalance in this metric. Analyzing imbalance in MAP works like

this:

1. Look at the metrics view for any ‘fat’ regions. These represent imbalance in that metric during that region of time. This tells us (A) that there is an imbalance, and (B) which metrics are affected.
2. Click and drag on the metrics view to select the ‘fat’ region, zooming the rest of the controls in to just this period of imbalance.
3. Now the stacks view and the source code views show which functions and lines of code were being executed during this imbalance. Are the processes executing different lines of code? Are they executing the same one, but with differing efficiencies? This tells us (C) which lines of code and execution paths are part of the imbalance.
4. Hover the mouse over the fattest areas on the metric graph and watch the minimum and maximum process ranks. This tells us (D) which ranks are most affected by the imbalance.

Now you know (A) whether there is an imbalance and (B) which metrics (CPU, memory, FPU, I/O) it affects. You also know (C) which lines of code and (D) which ranks to look at in more detail.

Often this is more than enough information to understand the immediate cause of the imbalance (for example, late sender, workload imbalance) but for a deeper view you can now switch to DDT and rerun the program with a breakpoint in the affected region of code. Examining the two ranks highlighted as the minimum and maximum by MAP with the full power of an interactive debugger helps get to the root cause of the imbalance behavior.

Accelerator

If you have Arm Forge Professional, the NVIDIA CUDA accelerator metrics are enabled on x86_64. Please contact Arm Sales at HPCToolsSales@arm.com for information on how to upgrade.

Note: Accelerator metrics are not available when linking to the static MAP sampler library.

GPU temperature: The temperature of the GPU as measured by the on-board sensor.

GPU utilization: Percent of time that the GPU card was in use, that is, one or more kernels are executing on the GPU card. If multiple cards are present in a compute node this value is the mean across all the cards in a compute node. Adversely affected if CUDA kernel analysis mode is enabled (see section 30.1).

Time in global memory accesses: Percent of time that the global (device) memory was being read or written. If multiple cards are present in a compute node this value is the mean across all the cards in a compute node.

GPU memory usage: The memory allocated from the GPU frame buffer memory as a percentage of the total available GPU frame buffer memory.

Energy

The energy metrics are only available with Arm Forge Professional. All metrics are measured per node. If you are running your job on more than one node, MAP shows the minimum, mean and maximum power consumption of the nodes.

Note: energy metrics are not available when linking to the static MAP sampler library.

GPU power usage: The cumulative power consumption of all GPUs on the node, as measured by the NVIDIA on-board sensor. This metric is available if the Accelerator metrics are present.

CPU power usage: The cumulative power consumption of all CPUs on the node, as measured by the Intel on-board sensor (Intel RAPL).

System power usage: The power consumption of the node as measured by the Intel Energy Checker or the Cray metrics.

Requirements

CPU power measurement requires an Intel CPU with RAPL support, for example Sandy Bridge or newer, and the intel_rapl powercap kernel module to be loaded.

Node power monitoring is implemented via one of two methods: the Arm IPMI energy agent which can read IPMI power sensors, or the Cray HSS energy counters.

For more information on how to install the Arm IPMI energy agent please see I.7 Arm IPMI Energy Agent. The Cray HSS energy counters are known to be available on Cray XK6 and XC30 machines.

Accelerator power measurement requires a NVIDIA GPU that supports power monitoring. This can be checked on the command-line with `nvidia-smi -q -d power`. If the reported power values are reported as "N/A", power monitoring is not supported.

Lustre

Lustre metrics are enabled if your compute nodes have one or more Lustre filesystems mounted. Lustre metrics are obtained from a Lustre client process running on each node. Therefore, the data presented gives the information gathered on a per-node basis. The data presented is also cumulative over all of the processes run on a node, not only the application being profiled. Therefore, there may be some data reported to be read and written even if the application itself does not perform file I/O through Lustre. However, an assumption is made that the majority of data read and written through the Lustre client will be from an I/O intensive application, not from background processes. This assumption has been observed to be reasonable. For generated application profiles with more than a few megabytes of data read or written, almost all of the data reported in Arm MAP is attributed to the application being profiled.

The data that is gathered from the Lustre client process is the read and write rate of data to Lustre, as well as a count of some metadata operations. Lustre does not just store pure data, but associates this data with metadata, which describes where data is stored on the parallel file system and how to access it. This metadata is stored separately from data, and needs to be accessed whenever new files are opened, closed, or files are resized. Metadata operations consume time and add to the latency in accessing the data. Therefore, frequent metadata operations can slow down the performance of I/O to Lustre. Arm MAP reports on the total number of metadata operations, as well as the total number of file opens that are encountered by a Lustre client. With the information provided in Arm MAP you can observe the rate at which data is read and written to Lustre through the Lustre client, as well as be able to identify whether a slow read or write rate can be correlated to a high rate of expensive metadata operations.

Notes:

- *For jobs run on multiple nodes, the reported values are the mean across the nodes.*
- *If you have more than one Lustre filesystem mounted on the compute nodes the values are summed across all Lustre filesystems.*
- *Metadata metrics are only available with Arm Forge Professional.*

Lustre read transfer: The number of bytes read per second from Lustre.

Lustre write transfer: The number of bytes written per second to Lustre.

Lustre file opens: The number of file open operations per second on a Lustre filesystem.

Lustre metadata operations: The number of metadata operations per second on a Lustre filesystem. Metadata operations include file open, close and create as well as operations such as readdir, rename, and unlink.

Note: depending on the circumstances and implementation 'file open' may count as multiple operations, for example, when it creates a new file or truncates an existing one.

Zooming

To examine a small time range in more detail you can horizontally zoom in the metric graphs by selecting the time-range you wish to see then left-clicking inside that selected region.

All the metric graphs will then resize to display that selection in greater detail. This only effects the metric graphs, as the graphs in all the other views, such as the code editor, will already have redrawn to display only the selected region when that selection was made.

A right-click on the metric graph zooms the metric graphs out again.

This horizontal zoom is limited by the number of samples that were taken and stored in the MAP file. The more you zoom in the more 'blocky' the graph becomes.

While you can increase the resolution by instructing MAP to store more samples (see `ALLINEA_SAMPLER_NUM_SAMPLES` and `ALLINEA_SAMPLER_INTERVAL` in 16.11 MAP environment variables) this is not recommended as it may significantly impact performance of both the program being profiled and of MAP when displaying the resulting `.map` file.

You can also zoom in vertically to better see fine-grained variations in a specific metric's values. The auto-zoom button beneath the metric graphs will cause the graphs to automatically zoom in vertically to fit the data shown in the currently selected time range. As you select new time ranges the graphs automatically zoom again so that you see only the relevant data.

If the automatic zoom is insufficient you can take manual control of the vertical zoom applied to each individual metric graph. Holding down the *CTRL* key (or the *CMD* key on Mac OS X), while either dragging on a metric graph or using the mouse-wheel while hovering over one, will zoom that graph vertically in or out, centered on the current position of the mouse.

A vertically-zoomed metric graph can be panned up or down by either holding down the *SHIFT* key while dragging on a metric graph or just using the mouse-wheel while hovering over it. Manually adjusting either the pan or zoom will disable auto-zoom mode for that graph, click the auto-zoom button again to reapply it.

Action	Usage	Description
Select	Drag a range in a metric graph.	Selects a time range to examine. Many components (but not the metric graphs) will rescale to display data for this time range only.
Reset 	Click the Reset icon (under the metric graphs).	Selects the entire time range. All components (including the metric graphs) will rescale to display the entire set of data. All metric graphs will be zoomed out.
Horizontal zoom in	Left click a selection in a metric graph.	Zoom in (horizontally) on the selected time range.
Horizontal zoom out	Right-click a metric graph.	Undo the last horizontal zoom in action.
Vertical zoom in/out	Ctrl + mouse scroll wheel or Ctrl + Drag on a metric graph.	Zoom a single metric graph in or out.
Vertical pan	Mouse scroll wheel or Shift+Drag on a metric graph.	Pan a single metric graph up or down.
Automatic vertical zoom 	Toggle the Automatic Vertical Zoom icon (under the metric graphs).	Automatically change the zoom of each metric graph to best fit the range of values each graph contains in the selected time range. Manually panning or zooming a graph will disable auto vertical zoom for that graph only.

Viewing totals across processes and nodes

The metric graphs show the statistical distribution of the metric across ranks or compute nodes (depending on the metric). So, for example, the *Nodes power usage* metric graph shows the statistical distribution of power usage of the compute nodes.

If you hover the mouse over the name of a metric to the left hand side of the graph MAP will display a tool tip with additional summary information. The tool tip will show you the *Minimum*, *Maximum*, and *Mean* of the metric across time and ranks or nodes.

For metrics which are not percentages the tool tip will also show the peak sum across ranks / nodes. For example, the *Maximum (∑ all nodes)* line in the tool tip for *Nodes power usage* shows the peak power usage summed across all compute nodes. This does not include power used by other components, for example, network switches.

For some metrics which are rates (for example, *Lustre read transfer*) MAP will also show the cumulative total across all ranks / nodes in the tool tip, for example, *Lustre bytes read (∑ all nodes)*.

Custom metrics

Custom metrics can be written to collect and expose additional data (for example, PAPI counters) in the metrics view.

User custom metrics should be installed under the appropriate path in your home directory, for example, `/home/your_user/.allinea/map/metrics`. Custom metrics can also be installed for all

users by placing them in the MAP installation directory, for example, `/arm_installation_directory/map/metrics`. If a metric is installed in both locations, the user installation will take priority.

Detailed information on how to write custom metrics can be found in supplementary documentation bundled with the Arm Forge installation in `allinea-metric-plugin-interface.pdf`.

PAPI metrics

Note: Arm Forge Professional is required to make use of this feature. Please contact Arm Sales at HPCToolsSales@arm.com for details on how to upgrade.

The PAPI metrics are additional metrics available for MAP which use the Performance Application Programming Interface (PAPI). They can be used on any system supported by PAPI.

Note: In this release PAPI metrics will be collected from the main thread only.

Due to the limitations of PAPI, some metrics may be unavailable on your system. MAP displays all available metrics and where metrics are not available error messages are displayed.

As there is a limit on the type and number of events that can be counted together, PAPI metrics have been split up into small groups of compatible events, so that the user can choose which events to view.

To change which group of metrics MAP uses, navigate to the directory indicated on completion of the installation process and modify the `PAPI.config` file.

Installation

To use these metrics, download and install PAPI from http://icl.cs.utk.edu/papi/index.html. Then run the metrics installer `papi_install.sh` from the Arm Forge directory.

PAPI config file

Once installation has completed, edit the `PAPI.config` file to set your configuration as required.

By default a template `PAPI.config` file is provided in your installation directory at `/arm_installation_directory/map/metrics`. Alternatively, the `PAPI.config` file can be located inside your configuration directory as set by the `ALLINEA_CONFIG_DIR` environment variable. By default your configuration directory is `\$HOME/.allinea`.

To use a `PAPI.config` file located elsewhere, set and export the `ALLINEA_PAPI_CONFIG` environment variable to point to your `PAPI.config` file. For example:

```
export ALLINEA_PAPI_CONFIG=/opt/arm/map/metrics/PAPI.config.
```

This needs to be set before running MAP.

If you are using a queuing system, be sure that the `ALLINEA_PAPI_CONFIG` variable is set and exported to all the compute nodes, by adding the `ALLINEA_PAPI_CONFIG` export line to the job script before the MAP command line.

The PAPI config file contains all the metrics sets that can be used and the location of it has been indicated at the end of the installation process. The default metric set is `Overview`. If you want to use another PAPI metrics set, modify the value of the variable called `set` to the desired PAPI metrics set of either `CacheMisses`, `BranchPrediction` or `FloatingPoint`.

PAPI overview metrics

This group of metrics gives a basic overview of the program which has been profiled.

DP FLOPS: The number of double precision floating-point operations performed per second. This uses the PAPI_DP_OPS (double precision floating-point operations) event. What it actually counts differs

across architectures. Additionally, there are many caveats surrounding this PAPI preset on Intel architectures. See http://icl.cs.utk.edu/projects/papi/wiki/PAPITopics:SandyFlops for more details.

Cycles per instruction: The number of CPU cycles per instruction executed. This uses the PAPI_TOT_CYC (total cycles) and PAPI_TOT_INS (total instructions) events.

L2 data cache misses: The number of L2 data cache misses per second. This uses the PAPI_L2_DCM (L2 data cache misses) event. This metric is only available in this preset if the system has enough hardware counters (5 at least) to collect the required events.

PAPI cache misses

This group of metrics focuses on cache misses at various levels of cache.

L1 cache misses: The number of L1 cache misses per second. This uses the PAPI_L1_TCM (L1 total cache misses) event, although if this event is unavailable the L1 data cache misses metric (using the PAPI_L1_DCM event) will be displayed instead.

L2 cache misses: The number of L2 cache misses per second. This uses the PAPI_L2_TCM (L2 total cache misses) event, although if this event is unavailable the L2 data cache misses metric (using the PAPI_L2_DCM event) will be displayed instead.

L3 cache misses: The number of L3 cache misses per second. This uses the PAPI_L3_TCM (L3 total cache misses) event, although if this event is unavailable the L3 data cache misses metric (using the PAPI_L3_DCM event) will be displayed instead.

PAPI branch prediction

This group of metrics focuses on branch prediction instructions.

Branch instructions: The number of branch instructions per second. This uses the PAPI_BR_INS (branch instructions) event.

Mispredicted branch instructions: The number of conditional branch instructions that are mispredicted each second. This uses the PAPI_BR_MSP (mispredicted branch instructions) event.

Completed instructions: The number completed instructions per second. This uses the PAPI_TOT_INS event, and is included to provide context for the above other metrics in this group.

PAPI floating-point

This group of metrics focuses on floating-point instructions.

Floating-point scalar instructions: The number of scalar floating-point instructions per second. This uses the PAPI_FP_INS event.

Floating-point vector instructions: The number of vector floating-point instructions per second. This uses the PAPI_VEC_SP (single-precision vectorized instructions) and PAPI_VEC_DP (double-precision vectorized instructions) events, although if those events are unavailable the Vector Instructions metric will be displayed instead.

Vector instructions: The number of vector instructions (floating-point and integer) per second. This uses the PAPI_VEC_INS event, but is only displayed if the events needed for the Floating-point vector instructions metric are not available.

Completed instructions: The number completed instructions per second. This uses the PAPI_TOT_INS event, and is included to provide context for the above other metrics in this group.

Main-thread, OpenMP and Pthread view modes

The percentage values and activity graphs shown alongside the source code and in the *Stacks*, *OpenMP Regions* and *Functions* views can present information for multithreaded programs in a variety of different ways.

MAP will initially choose the most appropriate view mode for your program. However, in some cases, for example such as when you have written a program to use raw pthreads rather than OpenMP, you may wish to change the mode to get a different perspective on how your program is executing multiple threads and using multiple cores. You can switch between view modes from the *View* menu.

Main thread only mode

In this view mode only the main thread from each process is displayed; the presence of any other thread is ignored. A value of 100% for a function or line means that all the processes' main threads are at that location. This is the best mode to use when exploring single-threaded programs and programs that are unintentionally/indirectly multithreaded (that is, recent implementations of both Open MPI and CUDA will start their own thread).

This is the default mode for all non-OpenMP programs. The *OpenMP Regions* tab is not displayed in this mode.

Note that the *CPU instruction* metric graphs (showing the proportion of time in various classes of CPU instructions: such as integer, floating-point, and vector) are *not* restricted to the main thread when in the *Main thread only* view mode. These metric graphs always represent the data gathered from all the CPU cores.

OpenMP mode

This view mode is optimized for interpreting programs where OpenMP is the primary source of multithreaded activity. Percentage values and activity graphs for a line or function indicate the proportion of the available resources that are being used on that line. For serial code on a main thread this is the proportion of processes at that location, for OpenMP code the contribution from each process is further broken down by the proportion of CPU cores running threads that are at that location in the code.

For example, a timeslice of an activity graph showing 50% dark green (serial, main-thread computation) and 50% light green (computation in an OpenMP region) means that half the processes were in serial code and half the processes were in an OpenMP region. Of the processes in an OpenMP region 100% of the available cores (as determined by the cores per process value, see 27 Processes and cores view) were being used for OpenMP.

This is the default mode for OpenMP programs. It is only available for programs where MAP detected an OpenMP region.

Pthread mode

This view mode is optimized for interpreting programs that make explicit use of pthreads. Percentage values and activity graphs reflect the proportion of CPU cores that are being used out of the maximum number of expected cores per process, see 27 Processes and cores view.

A value of 100% for a function or line means that 100% of the expected number of CPU cores per process were working at that location. The main thread's contribution gets no special attention so activity on the main thread(s) will appear the same height as activity from any other thread.

The advantage of this is that it makes it obvious when the program is not making full use of all the CPU cores available to it. But it has the downside of it being harder to analyze the performance of the intentionally serial sections of code performed by each process. This is because activity occurring only on one thread per process will be restricted to at most $1/n^{th}$ of a percentage value or height on an activity graph, where n is the number of cores per process.

This mode is not used by default so must be explicitly selected. It is only available for multithreaded programs.

The *OpenMP Regions* tab is not displayed in this mode.

Processes and cores view

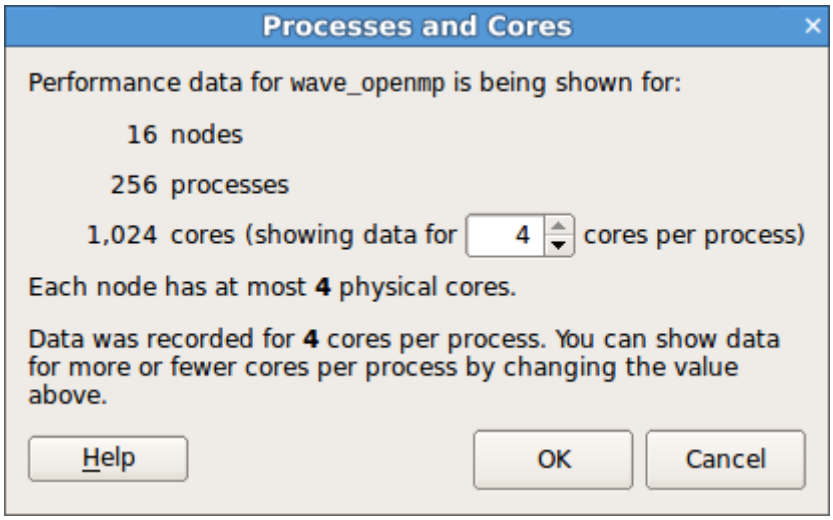

Figure 115: *Process and Cores Window*

Most modern CPUs support hyperthreading and report multiple *logical* cores for each *physical* core. Some programs run faster when scheduling threads or processes to use these hyperthreaded cores, while most HPC codes run more slowly. Rather than show all of the sparklines at half-height simply because the hyperthreaded cores are (wisely) not being used, MAP tries to detect this situation and will rescale its expectations to the number of physical cores used by your program.

If this heuristic goes wrong for any reason you will see large portions of unusual colors in your sparklines and the application activity chart (for example, bright red). When that happens, open this dialog and increase the *cores per process* setting.

You can find this dialog via the `View` → `Processes and Cores` menu or by clicking on the *X cores (Y per process)* hyperlinked text in the application details section above the metric graphs.

Running MAP from the command line

MAP can be run from the command line with the following arguments:

`--no-mpi`

Run MAP with 1 process and without invoking `mpirun`, `mpiexec`, or equivalent.

`--queue`

Force MAP to submit the job to the queueing system.

`--no-queue`

Run MAP without submitting the job to the queueing system.

`--view=VIEW`

Start MAP using `VIEW` as the default view. `VIEW` must be one of `(main|pthread|openmp)`. If the selected view is not available, the `main` view will be displayed.

`--export=OUTPUT.json PROFILEDATA.map`

Export `PROFILEDATA.map` to `OUTPUT.json` in JSON format, without user interaction. For the format specification see 29.1 JSON format.

`--profile`

Generate a MAP profile but without user interaction. This will not display the MAP GUI. Messages are printed to the standard output and error. The job is not run using the queueing system unless used in conjunction with `--queue`. When the job finishes a map file is written and its name is printed.

`--export-functions=FILE`

Export all the profiled functions to `FILE`. Use this in conjunction with `--profile`. The output should be CSV file name. Examples:

```
map --profile --export-functions=foo.csv ...
```

`--start-after=TIME`

Start profiling `TIME` seconds after the start of your program. Use this in conjunction with `--stop-after=TIME` to focus MAP on a particular time interval of the run of your program.

`--stop-after=TIME`

Stop profiling `TIME` seconds after the start of your program. This will terminate your program and proceed to gather the samples taken after the time given has elapsed.

When running without the GUI, normal redirection syntax can be used to read data from a file as a source for the executable's standard input. Examples:

```
cat <input-file> | map --profile ...
map --profile ... < <input-file>
```

Normal redirection can also be used to write data to a file from the executable's standard output:

```
map --profile ... > <output-file>
```

For OpenMP jobs, simply use the `OMP_NUM_THREADS` environment variable (or leave it blank) exactly as you usually would when running your application. There is no need to pass the number of threads to MAP as an argument.


```
OMP_NUM_THREADS=8 map --profile ... > <output-file>
```

`--enable-metrics=METRICS`
`--disable-metrics=METRICS`

Allows you to specify comma-separated lists which explicitly enable or disable metrics for which data is to be collected. If the metrics specified cannot be found, or if a metric is both enabled and disabled, an error message is displayed and MAP exits. Metrics which are always enabled or disabled cannot be explicitly disabled or enabled, respectively. A metrics source library which has all its metrics disabled, either in the XML definition or via `--disable-metrics`, will not be loaded. Metrics which can be explicitly enabled or disabled can be listed using the `--list-metrics` option.

The enabled/disabled metrics settings do not persist when running MAP without the GUI, so they will need to be specified for each profiling session. When running MAP in GUI mode, the effect of these settings will be displayed in the Metrics section of the run dialog, where the user can further refine their settings. These settings will then persist to the next GUI session.

`--cuda-kernel-analysis`

Enables CUDA kernel analysis mode, providing line level profiling information on CUDA kernels running on a GPU at the cost of potentially significant overhead. See section 30.

Profiling MPMD programs

The easiest way to profile MPMD programs is by using Express Launch to start your application.

To use Express Launch, simply prefix your normal MPMD launch line with `map`. For example, to profile an MPMD application without user interaction you can use:

```
map --profile mpirun -n 1 ./master : -n 2 ./worker
```

For more information on Express Launch, and compatible MPI implementations, see section 16.1.

Profiling MPMD programs without Express Launch

The command to create a profile from an MPMD program using MAP is:

```
map <map mode> --np=<#processes> --mpiargs=<MPMD command> <one
   MPMD program>
```

This example shows how to run MAP without user interaction using the flag `--profile`:

```
map --profile --np=16 --mpiargs="-n 8 ./exe1 : -n 8 ./exe2" ./exe1
```

First the number of processes used by the MPMD programs is set, in this case 8+8=16. Then an MPMD style command as an mpi argument is specified, followed by one of the MPMD programs.

Exporting profiler data in JSON format

MAP provides an option to export the profiler data in machine readable JSON format.

To export as JSON, first you need to open a `.map` file in MAP. Then the profile data can be exported by clicking *File* and selecting the *Export Profile Data as JSON* option.

For a command line option, see 28 Running MAP from the command line.

JSON format

The JSON document contains a single JSON object containing two object members, `info` containing general information about the profiled program, and `samples` with the sampled information. An example of profile data exported to a JSON file is given in Section 29.4.

- `info` (Object): If some information is not available, the value is null instead.
  - `command_line` (String): Command line call used to run the profiled application (for example `aprun -N 24 -n 256 -d 1 ./my_exe`).
  - `machine` (String): Hostname of the node on which the executable was launched.
  - `notes` (String): A short description of the run or other notes on configuration and compilation settings. This is specified by setting the environment variable `ALLINEA_NOTES` berfore running MAP.
  - `number_of_nodes` (Number): Number of nodes run on.
  - `number_of_processes` (Number): Number of processes run on.
  - `runtime` (Number): Runtime in milliseconds.
  - `start_time` (String): Date and time of run in ISO 8601 format.
  - `create_version` (String): Version of MAP used to create the map file.
  - `metrics` (Object): Attributes about the overall run, reported once per process, each represented by an object with `max`, `min`, `mean`, `var` and `sums` fields, or `null`, when the metric is not available. The `sums` series contains the sum of the metric across all processes / nodes for each sample. In many cases the values over all nodes will be the same, that is the max, min and mean values are the same, with variance zero. For example, in homogeneous systems `num_cores_per_node` is the same over all nodes.
    * `wchar_total` (Object): The number of bytes written in total by I/O operation system calls (see `wchar` in the Linux Programmer's Manual page 'proc': `man 5 proc`).
    * `rchar_total` (Object): The number of bytes read in total by I/O operation system calls (see `rchar` in the Linux Programmer's Manual page 'proc': `man 5 proc`).
    * `num_cores_per_node` (Object): Number of cores available per node.
    * `memory_per_node` (Object): RAM installed per node.
    * `nvidia_gpus_count` (Object): Number of GPUs per node.
    * `nvidia_total_memory` (Object): GPU frame buffer size per node.
    * `num_omp_threads_per_process` (Object): Number of OpenMP worker threads used per process.

- `samples` (Object)
  - `count` (Number): Number of samples recorded.
  - `window_start_offset` (Array of Numbers): Offset of the beginning of each sampling window, starting from zero. The actual sample might have been taken anywhere in between this offset and the start of the next window, that is the window offsets w_i and w_{i+1} define a semi-open set $(w_i, w_{i+1}]$ in which the sample was taken.
  - `activity` (Object): Contains information about the proportion of different types of activity performed during execution, according to different view modes. The types of view modes possibly shown are OpenMP, PThreads and Main Thread, described in Section 26. Only available view modes are exported, for example, a program without OpenMP sections will not have an OpenMP activity entry.

    Note: The sum of the proportions in an activity might not add up to 1, this can happen when there are fewer threads running than MAP has expected. Occasionally the sum of the proportions shown for a sample in PThreads or OpenMP threads mode might exceed 1. When this happens, the profiled application uses more cores than MAP assumes the maximum number of cores per process can be. This can be due to middleware services launching helper threads which, unexpectedly to MAP, contribute to the activity of the profiled program. In this case, the proportions for that sample should not be compared with the rest of proportions for that activity in the sample set.
  - `metrics` (Object): Contains an object for each metric that was recorded. These objects contain four lists each, with the minimum, maximum, average and variance of that metric in each sample. The format of a `metrics` entry is given in Section 29.3. All metrics recorded in a run are present in the JSON, including custom metrics. The names and descriptions of all core MAP metrics are given in Section 29.3. It is assumed that a user including a custom metrics library is aware of what the custom metric is reporting. See the Arm Metric Plugin Interface documentation.

Activities

Each exported object in an activity is presented as a list of fractional percentages (0.0 – 1.0) of sample time recorded for a particular activity during each sample window. Therefore, there are as many entries in these list as there are samples.

Description of categories

The following is the list of all of the categories. Only available categories are exported, see sections 29.2.2 and 29.2.3.

- `normal_compute`: Proportion of time spent on the CPU which is not categorized as any of the following activities. The computation can be, for example, floating point scalar (vector) addition, multiplication or division.
- `point_to_point_mpi`: Proportion of time spent in point-to-point MPI calls on the main thread and not inside an OpenMP region.
- `collective_mpi`: Proportion of time spent in collective MPI calls on the main thread and not inside an OpenMP region.
- `point_to_point_mpi_openmp`: Proportion of time spent in point-to-point MPI calls made from any thread within an OpenMP region.

- `collective_mpi_openmp`: Proportion of time spent in collective MPI calls made from any thread within an OpenMP region.
- `point_to_point_mpi_non_main_thread`: Proportion of time spent in point-to-point MPI calls on a pthread, but not on the main thread nor within an OpenMP region.
- `collective_mpi_non_main_thread`: Proportion of time spent in collective MPI calls on a pthread, but not on the main thread nor within an OpenMP region.
- `openmp`: Proportion of time spent in an OpenMP region, that is compiler-inserted calls used to implement the contents of a OpenMP loop.
- `accelerator`: Proportion of time spent in calls to accelerators, that is, blocking calls waiting for a CUDA kernel to return.
- `pthreads`: Proportion of compute time on a non-main (worker) pthread.
- `openmp_overhead_in_region`: Proportion of time spent setting up OpenMP structures, waiting for threads to finish and so on.
- `openmp_overhead_no_region`: Proportion of time spent in calls to the OpenMP runtime from an OpenMP region.
- `synchronisation`: Proportion of time spent in thread synchronization calls, such as pthread_-mutex_lock.
- `io_reads`: Proportion of time spent in I/O read operations, such as ‘read’.
- `io_writes`: Proportion of time spent in I/O write operations. Also includes file open and close time as these are typically only significant when writing.
- `io_reads_openmp`: Proportion of time spent in I/O read operations from within an OpenMP region.
- `io_writes_openmp`: Proportion of time spent in I/O write operations from within an OpenMP region.
- `mpi_worker`: Proportion of time spent in the MPI implementation on a worker thread.
- `mpi_monitor`: Proportion of time spent in the MPI monitor thread.
- `openmp_monitor`: Proportion of time spent in the OpenMP monitor thread.
- `sleep`: Proportion of time spent in sleeping threads and processes.

Categories available in `main_thread` activity

- `normal_compute`
- `point_to_point_mpi`
- `collective_mpi`
- `point_to_point_mpi_openmp`
- `collective_mpi_openmp`
- `openmp`
- `accelerator`
- `openmp_overhead_in_region`

- openmp_overhead_no_region
- synchronisation
- io_reads
- io_writes
- io_reads_openmp
- io_writes_openmp
- sleep

Categories available in openmp and pthreads activities

- normal_compute
- point_to_point_mpi
- collective_mpi
- point_to_point_mpi_openmp
- collective_mpi_openmp
- point_to_point_mpi_non_main_thread
- collective_mpi_non_main_thread
- openmp
- accelerator
- pthreads
- openmp_overhead_in_region
- openmp_overhead_no_region
- synchronisation
- io_reads
- io_writes
- io_reads_openmp
- io_writes_openmp
- mpi_worker
- mpi_monitor
- openmp_monitor
- sleep

Metrics

The following list contains the core metrics reported by MAP.

Only available metrics are exported to JSON. For example, if there is no Lustre filesystem then the Lustre metrics will not be included. If any custom metrics are loaded, they will be included in the JSON, but are not documented here.

For more information on the metrics see 24 Metrics View.

- CPU Instructions: see 24.1 CPU instructions
  - `instr_fp`: See CPU floating-point (percentage)
  - `instr_int`: See CPU integer (percentage)
  - `instr_mem`: See CPU memory access (percentage)
  - `instr_vector_fp`: See CPU floating-point vector (percentage)
  - `instr_vector_int`: See CPU floating-point vector (percentage)
  - `instr_branch`: See CPU branch (percentage)
  - `instr_scalar_fp`: The percentage of time each rank spends in standard x87 floating-point operations.
  - `instr_scalar_int`: The percentage of time each rank spends in standard integer operations.
  - `instr_implicit_mem`: Implicit memory accesses. The percentage of time spent executing instructions with implicit memory accesses.
  - `instr_other`: The percentage of time each rank spends in instructions which cannot be categorized as any of the ones given above.
- CPU Time: see 24.3 CPU time
  - `cpu_time_percentage`: See CPU time
  - `user_time_percentage`: See User-mode CPU time
  - `system_time_percentage`: See Kernel-mode CPU time
  - `voluntary_context_switches`: See Voluntary context switches (1/s)
  - `involuntary_context_switches`: See Involuntary context switches (1/s)
  - `loadavg`: See System load (percentage)
- I/O: see 24.4 I/O
  - `rchar_rate`: See POSIX I/O read rate (B/s)
  - `wchar_rate`: See POSIX I/O write rate (B/s)
  - `bytes_read`: See Disk read transfer (B/s)
  - `bytes_written`: See Disk write transfer (B/s)
  - `syscr`: See POSIX read syscall rate (calls/s)
  - `syscw`: See POSIX write syscall rate (calls/s)
- Lustre
  - `lustre_bytes_read`: Lustre read transfer (B/s)
  - `lustre_bytes_written`: Lustre write transfer (B/s)

  - `lustre_rchar_total`: Lustre bytes read
  - `lustre_wchar_total`: Lustre bytes written
- Memory: see 24.5 Memory
  - `rss`: See Memory usage in bytes (Resident Set Size)
  - `node_mem_percent`: See Node memory usage (percentage)
- MPI: see 24.6 MPI
  - `mpi_call_time`: See MPI call duration (ns)
  - `mpi_sent`: See MPI sent (B/s)
  - `mpi_recv`: See MPI received (B/s)
  - `mpi_calls`: Number of MPI calls per second per process
  - `mpi_p2p`: See MPI P2P (calls/s).
  - `mpi_collect`: See MPI collectives (calls/s)
  - `mpi_p2p_bytes`: See MPI point-to-point bytes
  - `mpi_collect_bytes`: See MPI collect bytes
- Accelerator: see 24.8 Accelerator
  - `nvidia_temp`: See GPU temperature (Celsius)
  - `nvidia_gpu_usage`: See GPU utilization (percentage)
  - `nvidia_memory_sys_usage`: See Time in global memory accesses (percentage)
  - `nvidia_memory_used_percent`: See GPU memory usage (percentage)
  - `nvidia_memory_used`: GPU memory usage in bytes
- Energy: see 24.9 Energy
  - `nvidia_power`: See GPU power usage (mW/node)
  - `rapl_power`: See CPU power usage (W/node)
  - `system_power`: See System power usage (W/node)
  - `rapl_energy`: CPU energy, integral of `rapl_power` (J)
  - `system_energy`: CPU energy, integral of `system_power` (J)

Example JSON output

In this section an example is given of the format of the JSON that is generated from a MAP file. This illustrates the description that has been given in the previous sections. This is not a full file, but should be used as an indication of how the information looks after export.

```
{
"info" : {
    "command_line" : "mpirun␣-np␣4␣./exec",
    "machine" : "hal9000",
    "number_of_nodes" : 30,
    "number_of_processes" : 240,
    "runtime" : 8300,
```



```
    "start_time" : "2016-05-13T11:36:31",
    "create_version" : "6.0.4"
    "metrics": {
        wchar_total: {max: 384605588, min: 132, mean: 24075798, var: 546823},
        rchar_total: {max: 6123987, min: 63, mean: 9873, var: 19287},
        num_cores_per_node: {max: 4, min: 4, mean: 4, var: 0},
        memory_per_node: {max: 4096, min: 4096, mean: 4096, var: 0},
        nvidia_gpus_count: {max: 0, min: 0, mean: 0, var: 0},
        nvidia_total_memory: {max: 0, min: 0, mean: 0, var: 0},
        num_omp_threads_per_process: {max: 6, min: 6, mean: 6, var: 0},
    }
},
"samples" : {
    "count" : 4,
    "window_start_offsets" : [ 0, 0.2, 0.4, 0.6 ],
    "activity" : {
        "main_thread" : {
            "normal_compute" : [ 0.762, 0.996, 1, 0.971 ],
            "io_reads" : [ 0.00416, 0.00416, 0, 0.00416 ],
            "io_writes" : [ 0.233, 0, 0, 0 ],
            "openmp" : [ 0, 0, 0, 0.01667 ],
            "openmp_overhead_in_region" : [ 0, 0, 0, 0.1 ],
            "openmp_overhead_no_region" : [ 0, 0, 0, 0.00417 ],
            "sleep" : [ 0, 0, 0, 0 ]
        },
        "openmp" : {
            "normal_compute" : [ 0.762, 0.996, 1, 0.971 ],
            "io_reads" : [ 0.00416, 0.00416, 0, 0.00416 ],
            "io_writes" : [ 0.233, 0, 0, 0 ],
            "openmp" : [ 0, 0, 0, 0.01319 ],
            "openmp_overhead_in_region" : [ 0, 0, 0, 0 ],
            "openmp_overhead_no_region" : [ 0, 0, 0, 0 ],
            "sleep" : [ 0, 0, 0, 0 ]
        },
        "pthreads" : {
            "io_reads" : [ 0.00069, 0.00069, 0, 0.00069 ],
            "io_writes" : [ 0.0389, 0, 0, 0 ],
            "normal_compute" : [ 0.1270, 0.1659, 0.1666, 0.1652 ],
            "openmp" : [ 0, 0, 0, 0.01319 ],
            "openmp_overhead_in_region" : [ 0, 0, 0, 0.02153 ],
            "openmp_overhead_no_region" : [ 0, 0, 0, 0.00069 ],
            "sleep" : [ 0, 0, 0, 0 ]
        }
    },
    "metrics" : {
        "wchar_total" : {
            "mins"  : [ 3957, 3957, 3958, 4959 ],
            "maxs"  : [ 4504, 4959, 5788, 10059 ],
            "means" : [ 3965.375, 4112.112, 4579.149, 6503.496 ],
            "vars"  : [ 2159.809, 49522.783, 169602.769, 2314522.699 ],
            "sums"  : [ 15860, 16448, 18316, 26012 ]
        },
        "bytes_read" : {
            "mins"  : [ 0, 0, 0, 0 ],
            "maxs"  : [ 34647.255020415301, 0, 0, 0 ],
            "means" : [ 645.12988722358205, 0, 0, 0 ],
            "vars"  : [ 9014087.0327749606, 0, 0, 0 ],
            "sums"  : [ 2580, 0, 0, 0]
        },
        "bytes_written" : {
            "mins"  : [ 0, 0, 0, 0 ],
            "maxs"  : [ 123, 0, 0, 0 ],
            "means" : [ 32, 0, 0, 0 ],
```



```
            "vars"  : [ 12, 0, 0, 0 ],
            "sums"  : [ 128, 0, 0, 0]
        }
    }
} }
```


GPU profiling

When profiling applications that use CUDA 8.0 and above, GPU kernels that can be tracked by NVIDIA's CUDA Profiling Tools Interface (CUPTI) will be displayed in a new "GPU Kernels" tab.

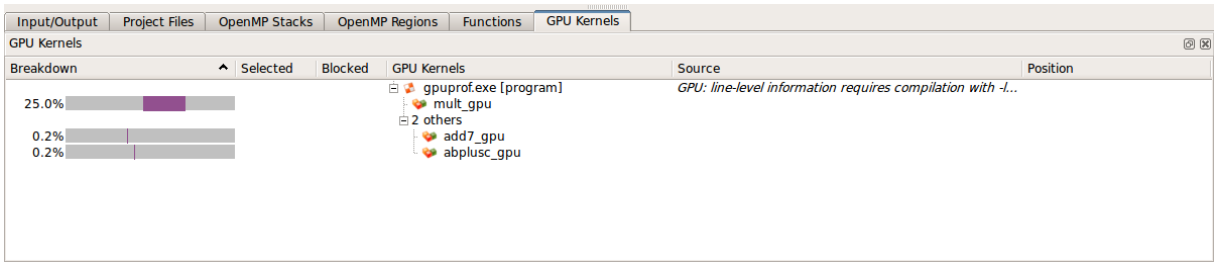

Figure 116: *GPU Kernels View*

This lists the CUDA kernels that were detected in the program alongside graphs indicating when those kernels were active. If multiple kernels were identified in a process within a particular sample they will have equal weighting in this graph.

Note that:

- CUDA kernels generated by OpenACC, CUDA Fortran, or offloaded OpenMP regions are not yet supported by MAP.
- GPU profiling is only supported with CUDA 8.0 and above.
- GPU profiling is not supported if the CUDA driver and toolkit versions do not match (for example, profiling a CUDA 8.0 program with a CUDA 9.0 driver is not supported).
- GPU profiling is not supported when statically linking the MAP sampler library.

Kernel analysis

CUDA kernel analysis mode is an advanced feature that provides insight into the activity within CUDA kernels. This mode can be enabled from the MAP run dialog or from the command line with `--cuda-kernel-analysis`.

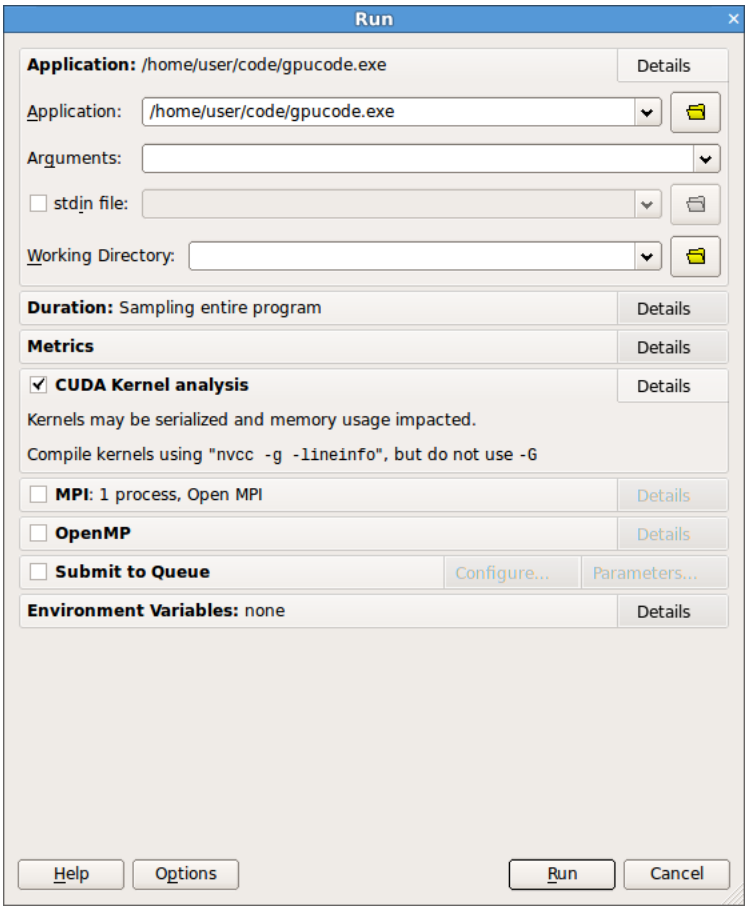

Figure 117: *Run window with CUDA kernel analysis enabled*

When enabled the "GPU Kernels" tab is enhanced to show a line-level breakdown of warp stalls. The possible categories of warp stall reasons are as listed in the `enum CUpti_ActivityPCSamplingStallReason` in the CUPTI API documentation (http://docs.nvidia.com/cuda/cupti/group__CUPTI__ACTIVITY__API.html):

Selected No stall, instruction is selected for issue.

Instruction fetch Warp is blocked because next instruction is not yet available, because of an instruction cache miss, or because of branching effects.

Execution dependency Instruction is waiting on an arithmetic dependency.

Memory dependency Warp is blocked because it is waiting for a memory access to complete.

Texture sub-system Texture sub-system is fully utilized or has too many outstanding requests.

Thread or memory barrier Warp is blocked as it is waiting at `__syncthreads` or at a memory barrier.

__constant__ memory Warp is blocked waiting for `__constant__` memory and immediate memory access to complete.

Pipe busy Compute operation cannot be performed due to required resource not being available.

Memory throttle Warp is blocked because there are too many pending memory operations.

Not selected Warp was ready to issue, but some other warp issued instead.

Other Miscellaneous stall reason.

Dropped samples Samples dropped (not collected) by hardware due to backpressure or overflow.

Unknown The stall reason could not be determined. Used when CUDA kernel analysis has not been enabled (see above) or when an internal error occurred within CUPTI or MAP.

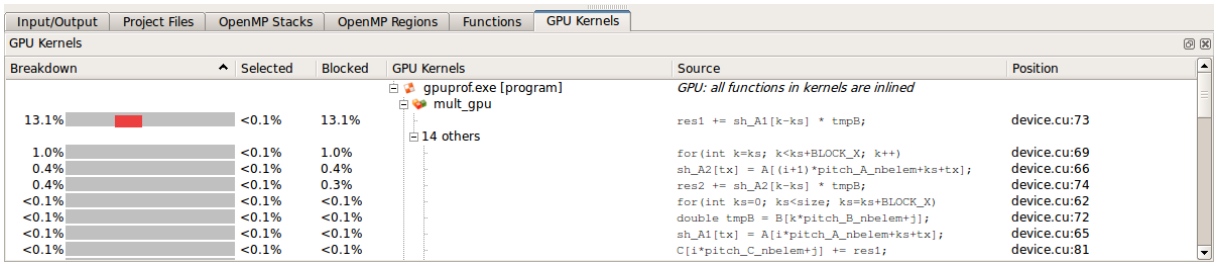

Figure 118: *GPU kernels view (with CUDA kernel analysis)*

Note that warp stalls are only reported per-kernel, so it is not possible to obtain the times within a kernel invocation at which different categories of warp stalls occurred. As function calls in CUDA kernels are automatically fully inlined it is not possible to see a stack trace of code within a kernel on the GPU.

Warp stall information is also present in the code editor (section 18.3), the selected line view (section 19.2), and in a warp stall reason graph in the metrics view (section 24).

Compilation

When compiling CUDA kernels do not generate debug information for device code (the `-G` or `--device-debug` flag) as this can significantly impair runtime performance. Use `-lineinfo` instead, for example:

```
nvcc device.cu -c -o device.o -g -lineinfo -O3
```

Performance impact

Enabling the CUPTI sampling will impact the target program in the following ways:

1. A short amount of time will be spent post-processing at the end of each kernel. This will depend on the length of the kernel and the CUPTI sampling frequency.
2. Kernels will be serialized. Each CUDA kernel invocation will not return until the kernel has finished and CUPTI post-processing has been performed. Without CUDA kernel analysis mode kernel invocation calls return immediately to allow CUDA processing to be performed in the background.
3. Increased memory usage whilst in a CUDA kernel. This may manifest as fluctuations between two memory usage values, depending on whether a sample was taken during a CUDA kernel or not.

Taken together the above may have a significant impact on the target program, potentially resulting in orders of magnitude slowdown. To combat this profile and analyse CUDA code kernels (with `--cuda-kernel-analysis`) and non-CUDA code (no `--cuda-kernel-analysis`) in separate profiling sessions.

The NVIDIA GPU metrics will be adversely affected by this overhead, particularly the “GPU utilization” metric. See section 24.8.

When profiling CUDA code it may be useful to only profile short subsection of the program so time is not wasted waiting for CUDA kernels you do not need to see data for. See section 16.3.8 for instructions on how to do this.

Customizing GPU profiling behavior

The interval at which CUPTI samples GPU warps can be modified by the environment variable `ALLINEA_SAMPLER_GPU_INTERVAL`. Accepted values are `max`, `high`, `mid`, `low`, and `min`, with the default value being `high`. These correspond to the values in the `enum CUpti_ActivityPCSamplingPeriod` in the CUPTI API documentation (http://docs.nvidia.com/cuda/cupti/group__CUPTI__ACTIVITY__API.html).

Reducing the sampling interval means warp samples are taken more frequently. While this may be needed for very short-lived kernels, setting the interval too low can result in a very large number of warp samples being taken which then require significant post-processing time once the kernel completes. Overheads of twice as long as the kernel's normal runtime have been observed. It is recommended that the CUPTI sampling interval is not reduced.

Known issues

- GPU profiling is only supported using CUDA 8.0 and above.
- GPU profiling is not supported if the CUDA driver and toolkit versions do not match (for example, profiling a CUDA 8 program with a CUDA 9 driver is not supported).
- When preparing your program for profiling, it is advised to match the version of the CUDA toolkit to that of the CUDA driver.
- CUPTI allocates a small amount of host memory each time a kernel is launched. If your program launches many kernels in a tight loop this overhead can skew the memory usage figures.
- CUDA kernels generated by OpenACC, CUDA Fortran or offloaded OpenMP regions are not yet supported by MAP.
- The graphs are scaled on the assumption that there is a 1:1 relationship between processes and GPUs, each process having exclusive use of its own CUDA card. The graphs may be of an unexpected height if some processes do not have a GPU, or if multiple processes share the use of a common GPU.
- Enabling CUDA kernel analysis mode can have a significant performance impact as described in section 30.3.
- GPU profiling is not supported when statically linking the MAP sampler library.

Python profiling

Arm Forge 19.0 adds the Python profiling capabilities you need to find and resolve bottlenecks for your Python codes.

For the latest information about Python profiling in MAP, see the Python Profiling web page.

Profile a Python script

This task describes how to profile a Python script. This feature is useful when profiling a mixed C, C++, Fortran, and Python program.

Python profiling replaces main thread stack frames originating from the Python interpreter with Python stack frames of the profiled Python script. To disable this feature, set `ALLINEA_SAMPLER_DISABLE_PYTHON_PROFILING=1`.

Python profiling in MAP has the following limited support:

- Profiling Python scripts running under the CPython interpreter (version 2.7, 3.5 and 3.6 only).
- Profiling Python scripts running under the Intel Distribution for Python.
- Profiling Python scripts running under the Anaconda Python distribution.
- Profiling Python scripts running under virtual enviornments.
- Profiling Python scripts that import modules which perform MPI on the main thread, such as mpi4py.
- Profiling Python scripts that import modules which use OpenMP. Only Python scripts running on the main thread of the Python interpreter are sampled by MAP.

Note: MAP will output warnings if the threading model of the MPI module is `MPI_THREAD_MULTIPLE`, such as in mpi4py. To prevent these warnings, change the default settings in mpi4py with the following: `mpi4py.rc.threaded = False` or `mpi4py.rc.thread_level = "funneled"`.

Note: If you are profiling on a system using ALPS or SLURM and the Python script does not use MPI, environment variables (section H.2) must be set.

Procedure

1. Check that the Python script runs successfully:

   ```
   $ python myscript.py
   ```

2. To profile the Python script with MAP, prepend the run command with `map`:

   ```
   $ map python myscript.py
   ```


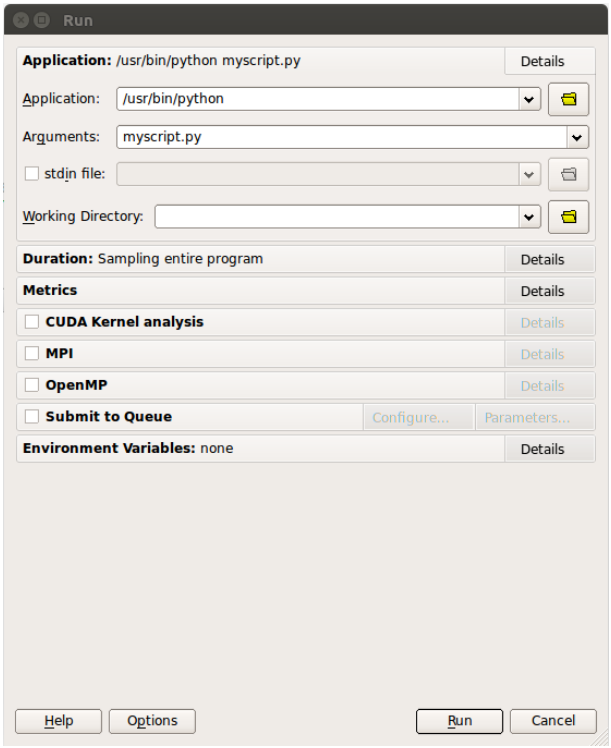

Figure 119: *Profiling a Python script*

3. Click **Run** and wait for MAP to finish profiling the Python script.
4. View the profiling results in MAP.

Results

When MAP finishes profiling the Python script, it saves a `.map` file in the current working directory and opens it for viewing in the GUI (unless you are using the offline feature).

Example: Profiling a simple Python script

This section demonstrates how to profile the Python example script `python-profiling.py` located in the `examples` directory.

1. Change into the `examples` directory and run the makefile to compile the example.

   ```
   $ make -f python-profiling.makefile
   ```

2. Start MAP

   ```
   $ ../bin/map python ./python-profiling.py --index 35
   ```

3. Click Run.
4. Wait for MAP to finish analyzing samples after the Python script has completed.

   Note: The MAP GUI launches showing the Python script and the line in the script where the most time was spent is selected.

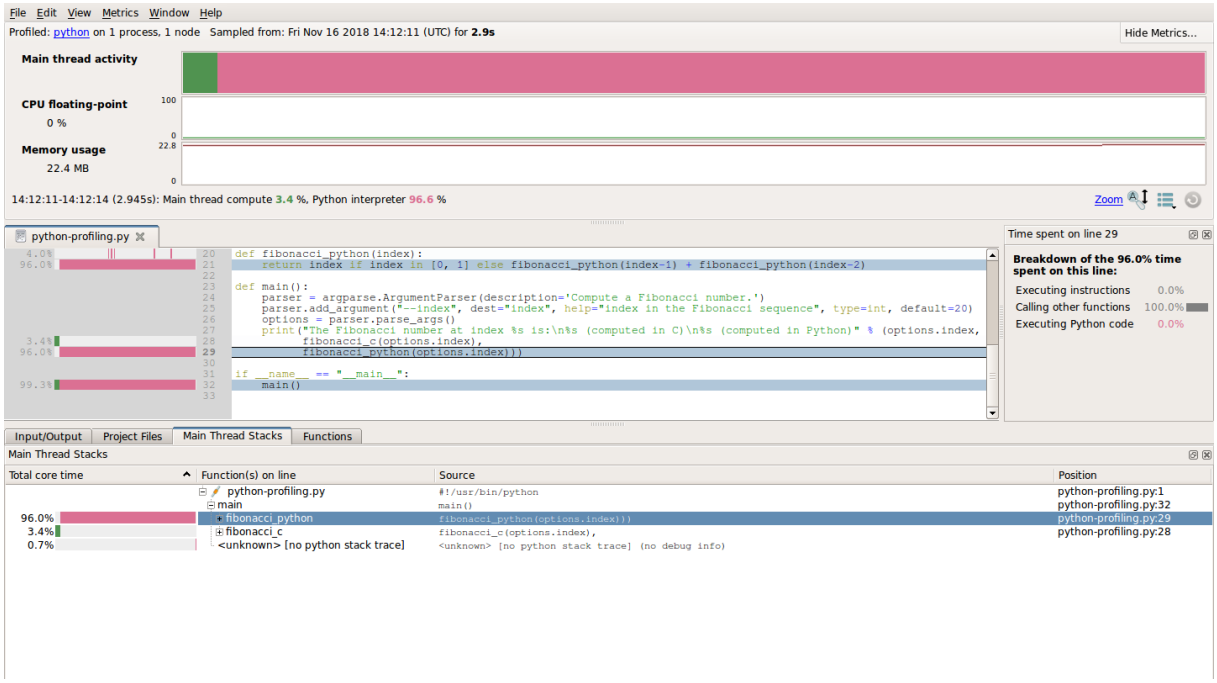

Figure 120: *Viewing Python profiling results*

5. Locate the first `fibonacci_c` stack frame in the *Main Thread Stacks* view. The callout to the C function is appended under `main` Python stack frame.

Next steps

- Examine the *Main Thread Activity* graph (section 24) for an overview of time spent in Python code compared with non-Python code.
- View source code lines (section 18.1) on which time was spent executing Python code and non-Python code.
- Compare time spent on the selected line executing Python code with non-Python code in the *Selected Lines View* (section 19).
- View a breakdown of time spent in different code paths in the *Main Thread Stacks* view (section 20).

  Note: Python stack frames are only displayed for the main thread of the Python interpreter. Normal stack frames are displayed for non-main threads of the Python interpreter.

Related information

- For more information on using MAP, see section 16.
- For information on debugging Python scripts with DDT, see section 5.16.

Known Issues

- MAP requires a significant amount of time to analyze samples when profiling a Python script that imports modules which use OpenBLAS, such as NumPy. This is caused by the lack of unwind information in OpenBLAS. This results in `partial trace` nodes being displayed in MAP.

- mpi4py uses some MPI functions that were introduced in MPI version 3. For example `MPI_Mrecv`. MAP does not collect metrics from these functions, therefore MPI metrics for mpi4py will be inaccurate. To workaround this, use a custom Python MPI wrapper that only uses functions that were available before MPI version 3.
- When using reverse connect (`--connect`) and quick start (`--start`) in conjunction, the full path to the Python application must be provided.

Part IV

Appendix

Configuration

Arm Forge shares a common configuration file between Arm DDT and Arm MAP. This makes it easy for users to switch between tools without reconfiguring their environment each time.

Configuration files

Arm Forge uses two configuration files: the system wide `system.config` and the user specific `user.config`. The system wide configuration file specifies properties such as MPI implementation. The user specific configuration file describes user's preferences such as font size. The files are controlled by environment variables:

Environment Variable	Default
`ALLINEA_USER_CONFIG`	`\${ALLINEA_CONFIG_DIR}/user.config`
`ALLINEA_SYSTEM_CONFIG`	`\${ALLINEA_CONFIG_DIR}/system.config`
`ALLINEA_CONFIG_DIR`	`\${HOME}/.allinea`

Sitewide configuration

If you are the system administrator, or have write-access to the installation directory, you can provide a configuration file which other users are automatically given a copy of the first time that they start Arm Forge. In this case users no longer need to provide configuration for site-specific aspects such as queue templates and job submission.

First configure Arm Forge normally and run a test program to make sure all the settings are correct. When you are satisfied with your configuration execute one of the following commands:

```
forge --clean-config
```

This will remove any user-specific settings from your system configuration file and will create a `system.config` file that can provide the default settings for all users on your system. Instructions on how to do this are printed when `--clean-config` completes. Note that only the `system.config` file is generated. Arm Forge also uses a user-specific `user.config` which is not affected.

If you want to use DDT to attach to running jobs you also need to create a file called `nodes` in the installation directory with a list of compute nodes you want to attach to. See section 5.9 Attaching to running programs for details.

Startup scripts

When Arm Forge is started it searches for a sitewide startup script called `allinearc` in the root of the installation directory. If this file exists it is sourced before starting the tool. When using the remote client this startup script is sourced before any sitewide `remote-init` remote daemon startup script.

Similarly, you can also provide a user-specific startup script in `~/.allinea/allinearc`.

Note: If the `ALLINEA_CONFIG_DIR` environment variable is set then the software will look in `\$ALLINEA_CONFIG_DIR/allinearc` instead. When using the remote client the user-specific startup script will be sourced before the user-specific `~/.allinea/remote-init` remote daemon startup script.

Importing legacy configuration

If you have used a version of Arm DDT prior to 4.0 your existing configuration will be imported automatically. If the `DDTCONFIG` environment variable is set, or you use the `--config` command-line argument, the existing configuration will be imported. However, the legacy configuration file will not be modified, and subsequent configuration changes will be saved as described in the previous sections.

Converting legacy sitewide configuration files

If you have existing sitewide configuration files from a version of Arm DDT prior to 4.0 you will need to convert them to the new 4.0 format. This can easily be done using the following command line:

```
forge --config=oldconfig.ddt --system-config=newconfig.ddt --clean
   -config
```

Note: `newconfig.ddt` must not exist beforehand.

Using shared home directories on multiple systems

If your site uses the same home directory for multiple systems you may want to use a different configuration directory for each system.

You can do this by specifying the `ALLINEA_CONFIG_DIR` environment variable before starting Arm Forge. If you use the `module` system you may choose to set `ALLINEA_CONFIG_DIR` according to which system the module was loaded on.

For example, say you have two systems: harvester with login nodes `harvester-login1` and `harvester-login2` and `sandworm` with login nodes `sandworm-login1` and `sandworm-login2`. You may add something similar to the following code to your module file:

```
case $(hostname) in
   harvester-login*)
     ALLINEA_CONFIG_DIR=$HOME/.allinea/harvester
     ;;
   sandworm-login*)
     ALLINEA_CONFIG_DIR=$HOME/.allinea/sandworm
     ;;
esac
```


Using a shared installation on multiple systems

If you have multiple systems sharing a common Arm Forge installation, you may wish to have a different default configuration for each system. You can use the `ALLINEA_DEFAULT_SYSTEM_CONFIG` environment variable to specify a different file for each system. For example, you may add something similar to the following to your `module` file:

```
case $(hostname) in
   harvester-login*)
     ALLINEA_DEFAULT_SYSTEM_CONFIG=/sw/arm/forge/harvester.config
     ;;
   sandworm-login*)
     ALLINEA_DEFAULT_SYSTEM_CONFIG=/sw/arm/forge/sandworm.config
     ;;
esac
```

Integration with queuing systems

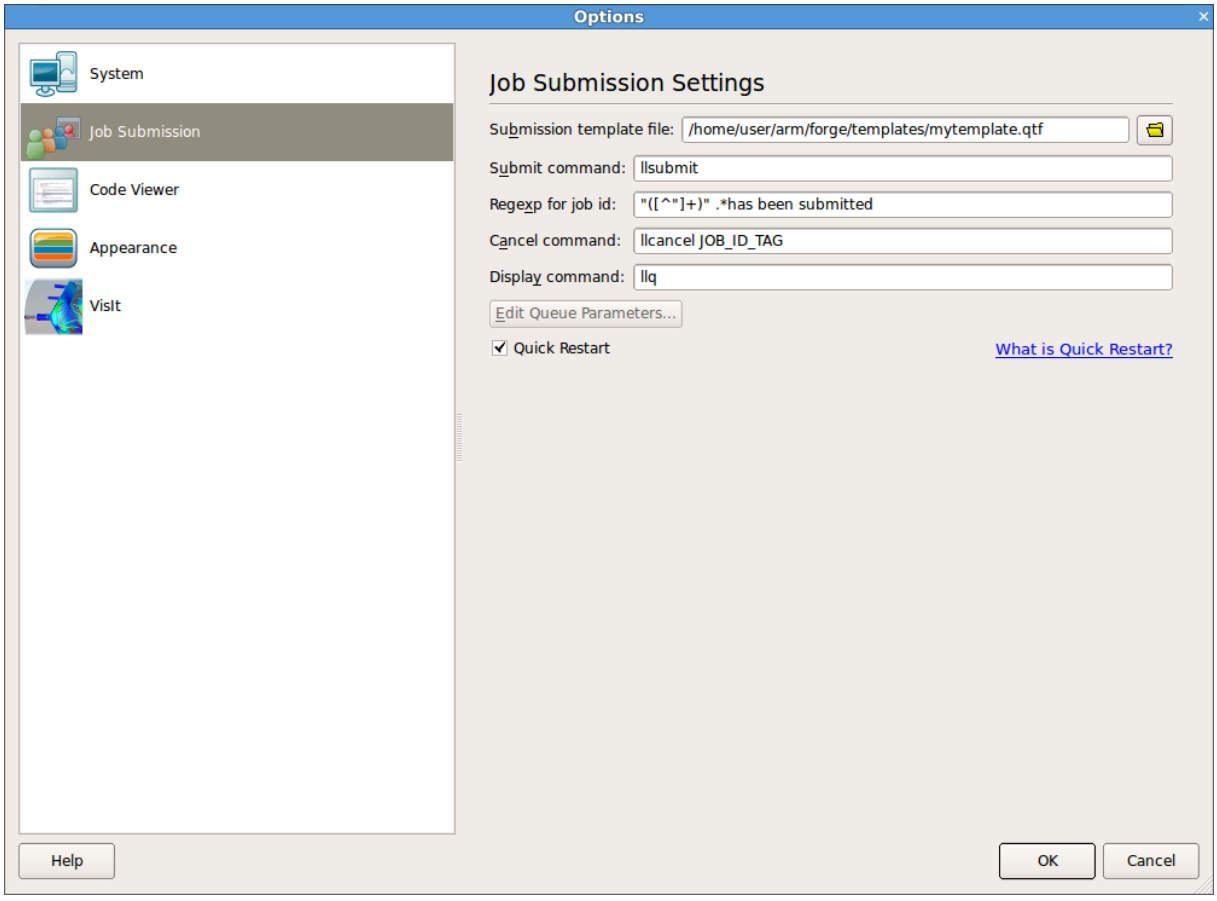

Figure 121: *Queuing Systems*

Arm Forge can be configured to interact with most job submission systems. This is useful if you wish to debug interactively but need to submit a job to the queue in order to do so.

MAP is usually run as a wrapper around `mpirun` or `mpiexec`, via the `map --profile` argument. Arm recommends using this to generate `.map` files instead of configuring MAP to submit jobs to the

queue, but both usage patterns are fully-supported.

In the *Options* window (*Preferences* on Mac OS X), you should choose *Submit job through queue*. This displays extra options and switches the GUI into queue submission mode.

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

1. Making a template script.
2. Setting the commands used to submit, cancel, and list queue jobs.

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

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

Template tutorial

Ordinarily, your queue script will probably end in a line that starts `mpirun` with your target executable. In most cases you can simply replace that line with `AUTO_LAUNCH_TAG`. For example, if your script currently has the line:

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

Then create a copy of it and replace that line with:

```
AUTO_LAUNCH_TAG
```

Select this file as the *Submission template file* on the *Job Submission Settings* page of the *Options*. Notice that you are no longer explicitly specifying the number of processes, and so on. You instead specify the number of processes, program name and arguments in the *Run* window.

Fill in *Submit command* with the command you usually use to submit your job, for example `qsub` or `sbatch`, *Cancel command* with the command you usually use to cancel a job, for example `qdel` or `scancel` and *Display command* with the command you usually use to display the current queue status, for example `qstat` or `squeue`.

You can usually use `(
d+)` as the *Regexp for job id*. This just scans for a number in the output from your *Submit command*.

Once you have a simple template working you can go on to make more things configurable from the GUI. For example, to be able to specify the number of nodes from the GUI you would replace an explicit number of nodes with the NUM_NODES_TAG. In this case replace:

```
#SBATCH --nodes=100
```

With:

```
#SBATCH --nodes=NUM_NODE_TAG
```

See appendix I.1 Queue template tags for a full list of tags.

The template script

The template script is based on the file you would normally use to submit your job. This is 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 Arm Forge prepares your job, it replaces each of these keywords with its value and then submits the new file to your queue.

To refer to tags in comments without Arm Forge detecting them as a required field the comment line must begin with ##.

Configuring queue commands

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

When you start a session Arm Forge will generate a submission file and append its file name to the submit command you give.

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

To cancel a job, Arm Forge 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. See appendix I.6 Job ID regular expression for details.

Configuring how job size is chosen

Arm Forge offers a number of flexible ways to specify the size of a job. You may choose whether *Number of Processes* and *Number of Nodes* options appear in the *Run* window or whether these should be implicitly calculated. Similarly you may choose to display *Processes per node* in the *Run* window or set it to a Fixed value.

Note: if you choose to display Processes per node in the Run window and `PROCS_PER_NODE_TAG` is specified in the queue template file then the tag will always be replaced by the Processes per node value from the Run dialog, even if the option is unchecked there.

Quick restart

DDT allows you reuse an existing queued job to quickly restart a run without resubmitting it to the queue, provided that your MPI implementation supports doing this. Simply check the *Quick Restart* check box on the Job Submission Options page.

In order to use quick restart, your queue template file must use `AUTO_LAUNCH_TAG` to execute your job.

For more information on `AUTO_LAUNCH_TAG`, see I.4.1 Using AUTO_LAUNCH_TAG.

Connecting to remote programs (remote-exec)

When Arm Forge needs to access another machine for remote launch or as part of starting some MPIs, it will attempt to use the secure shell, `ssh`, by default.

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

Arm Forge will use look for the script at `~/.allinea/remote-exec`, and it will be executed as follows:

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

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

SSH based remote-exec

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

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

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

Testing

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

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

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

If you are having trouble setting up `remote-exec`, please contact Arm support at Arm support for assistance.

Windows

The previously described functionality is also provided by the Windows remote client. However, there are two differences:

1. The script is named `remote-exec.cmd` rather than `remote-exec`.
2. The default implementation uses the `plink.exe` executable supplied with Arm Forge.

Optional configuration

Arm Forge providess an *Options* window (*Preferences* on Mac OS X), which allows you to quickly edit the settings outlined below.

System

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

Note: If you are not using Arm Forge to work with MPI programs select `none`.

Override default mpirun path: Allows you to override the path to the `mpirun` (or equivalent) command.

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

On Linux systems Arm Forge ships with four versions of the GNU GDB debugger: GDB 7.6.2, GDB 7.10.1, GDB 7.12.1 and GDB 8.1. GDB 7.12.1 is the recommended debugger for both MAP and DDT. These recommended defaults are selected automatically when *Automatic (recommended)* is selected from the **System Settings** page on the **Options** window.

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

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

Heterogeneous system support: DDT has support for running heterogeneous MPMD MPI applications where some nodes use one architecture and other nodes use another architecture. This requires a little preparation of your Arm Forge installation. You must have a separate installation of DDT for each architecture. The architecture of the machine running the Arm Forge GUI is called the host architecture. You must create symbolic links from the host architecture installation of Arm Forge to the other installations for the other architectures. For example with a 64-bit x86_64 host architecture (running the GUI) and some compute nodes running the 32-bit i686 architecture:

```
ln -s /path/to/arm-forge-i686/bin/ddt-debugger \
/path/to/arm-forge-x86_64/bin/ddt-debugger.i686
```

Enable CUDA software pre-emption: Allows debugging of CUDA kernels on a workstation with a single GPU.

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

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

Attach hosts file: When attaching, DDT will fetch a list of processes for each of the hosts listed in this file. See section 5.9 Attaching to running programs for more details.

Job submission

This section allows you to configure Arm Forge to use a custom `mpirun` command, or submit your jobs to a queuing system. For more information on this, see section A.2 Integration with queuing systems.

Code viewer settings

This allows you to configure the appearance of the Arm Forge code viewer, which is used to display your source code while debugging.

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

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

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

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

Colour Scheme: Color palette to use for the code viewer's background, text and syntax highlighting. Defined in Kate syntax definition format in the `resource/styles` directory of the Arm Forge install.

Visualize Whitespace: Enables or disables this display of symbols to represent whitespace. Useful for distinguishing between space and tab characters.

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

Appearance

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

Look and Feel: This determines the general graphical style of Arm Forge. This includes the appearance of buttons, context menus.

Override System Font Settings: This setting can be used to change the font and size of all components in Arm Forge (except the code viewer).

Getting support

While this user guide attempts to cover as many parts of the installation, features and uses of our tool as possible, there will be scenarios or configurations that are not covered, or are only briefly mentioned, or you may on occasion experience a problem using the product. If the solution to your problem is not in this guide then please contact Arm support at Arm support.

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

- Version number of Arm Forge. For example, `forge --version` and your operating system and the distribution, such as Red Hat Enterprise Linux 6.4. This information is all available by using the `--version` option on the command line of any Arm tool:

```
bash$ forge --version

Arm DDT
Part of Arm Forge.
Copyright (c) 2002-2018 Arm Limited (or its affiliates). All
   rights reserved.

Version: 18.0.2
Build ID: 556f23c4895e
Build Platform: Ubuntu 16.04 x86_64
Build Date: Jan 25 2018 21:15:53

Frontend OS: Ubuntu 16.04.2 LTS
Nodes' OS: unknown
    Last connected ddt-debugger: unknown
```

- The compiler used and its version number.
- The MPI library and CUDA toolkit version if appropriate.
- A description of the issue : what you expected to happen and what actually happened.
- An exact copy of any warning or error messages that you may have encountered.

Supported platforms

The tables in this section describe the architectures, operating systems, MPI distributions, compilers and accelerators that are supported by DDTand MAP.

DDT

The supported platforms and configurations listed here are correct at the time of publication. For a fully up-to-date version, see Arm DDT supported platforms on the Arm Developer website.

Architecture	Operating systems	MPI	Compilers	Accelerators
Intel and AMD (x86_64)	Red Hat Enterprise Linux/CentOS 6 and 7 SuSE Linux Enterprise 11, 12 and 15 Ubuntu 14.04, 16.04 and 18.04 Open SuSE 12, 13, 42.3 and 15.0	Open MPI 1.8, 1.10, 2.0, 2.1, 3.0 and 3.1 MPICH 3.1 to 3.2 MVAPICH2 2.0 to 2.3 Intel MPI 5.1.x to 2018.3 Cray MPT 6.3.1 to 7.7.1 SGI MPT 2.10 to 2.15 HPE MPI 1.1 IBM Platform MPI 9.1.x Bullx MPI 1.2.7.x to 1.2.9.x	GNU C/C++/Fortran Compiler 4.3.x to 8.1.x LLVM Clang 3.3 to 6.0 Intel Parallel Studio XE 2013.x to 2018.3 PGI Compiler 14.0 to 18.7 Cray Compiling Environment 8.3.x to 8.7.x Nvidia CUDA Compiler 7.0 to 9.2	Nvidia CUDA Toolkit 7.0 to 9.2
Armv8 (AArch64)	Red Hat Enterprise Linux/CentOS 7.4+ SuSE Linux Enterprise 12.2+ Ubuntu 16.04+	Open MPI 1.8, 1.10, 2.0, 2.1, 3.0 and 3.1 MPICH 3.1 to 3.2 MVAPICH2 2.0 to 2.3 Cray MPT 7.7.1 HPE MPI 1.1 Bullx MPI 1.2.7.x to 1.2.9.x	GNU C/C++/Fortran Compiler 4.3.x to 8.1.x LLVM Clang 3.3 to 6.0 Arm C/C++/Fortran Compiler 18.0 to 18.3 Cray Compiling Environment 8.7.x	–
Intel Xeon Phi (knl)	Red Hat Enterprise Linux/CentOS 7 SuSE Linux Enterprise 12	Open MPI 1.8, 1.10, 2.0, 2.1, 3.0 and 3.1 Intel MPI 5.1.x to 2018.3 MPICH 3.1 to 3.2 MVAPICH2 2.0 to 2.3 Cray MPT 5.0.x to 7.7.x HPE MPI 1.1 Bullx MPI 1.2.7.x to 1.2.9.x	GNU C/C++/Fortran Compiler 4.3.x to 8.1.x LLVM Clang 3.3 to 6.0 Intel Parallel Studio XE 2013.x to 2018.3 PGI Compiler 14.0 to 18.5 Cray Compiling Environment 8.3.x to 8.7.x	–

Architecture	Operating systems	MPI	Compilers	Accelerators
IBM Power (ppc64 and ppc64le)	Red Hat Enterprise Linux/CentOS 7	Open MPI 1.8, 1.10, 2.0, 2.1, 3.0 and 3.1 IBM Spectrum MPI 10.2	GNU C/C++/Fortran Compiler 4.3.x to 8.1.x IBM XL C/C++ Compiler 13.1.x IBM XL Fortran Compiler 15.1.x IBM XL Compiler 16.1.x PGI Compiler 18.1, 18.5 and 18.7	Nvidia CUDA Toolkit 9.2

Notes:

- Pretty printing of C++ types is supported for GNU and Intel compilers.
- Message queue debugging is supported for Bullx MPI, IBM PE, Intel MPI 4.1.x, MPICH, MVAPICH, and Open MPI.
- Version control integration is supported for Git 1.7+, Mercurial 2.1+ and Subversion 1.6+.

Batch schedulers supported:

- SLURM 2.6.3+ and 14.03+
- PBS
- TORQUE
- Moab
- Oracle Grid Engine
- Loadleveler
- Cobalt.

Batch scheduling systems are supported through *Queue Templates*.

See section A.2 Integration with queuing systems for more information.

See section E.16 SLURM for more details about SLURM support.

MAP

This table is accurate at the point of publishing this document. For a fully up-to-date version, see Arm MAP supported platforms on the Arm Developer website

Architecture	Operating systems	MPI	Compilers	Accelerators
Intel and AMD (x86_64)	Red Hat Enterprise Linux/CentOS 6 and 7 SuSE Linux Enterprise 11, 12 and 15 Ubuntu 14.04, 16.04 and 18.04 Open SuSE 12, 13, 42.3 and 15.0	Open MPI 1.8, 1.10, 2.0, 2.1, 3.0 and 3.1 MPICH 3.1 to 3.2 MVAPICH2 2.0 to 2.3 Intel MPI 5.1.x to 2018.3 Cray MPT 6.3.1 to 7.7.1 SGI MPT 2.10 to 2.15 HPE MPI 1.1 IBM Platform MPI 9.1.x Bullx MPI 1.2.7.x to 1.2.9.x	GNU Compiler C/C++ and Fortran 4.3.x to 8.1.x LLVM Clang 3.3 to 6.0 Intel Parallel Studio XE 2013.x to 2018.3 PGI Compiler 14.0 to 18.7 Cray Compiling Environment 8.3.x to 8.7.x Nvidia CUDA Compiler 7.0 to 9.2	Nvidia CUDA Toolkit 7.0 to 9.2
Armv8 (AArch64)	Red Hat Enterprise Linux/CentOS 7.4+ SuSE Linux Enterprise 12.2+ Ubuntu 16.04+	Open MPI 1.8, 1.10, 2.0, 2.1, 3.0 and 3.1 MPICH 3.1 to 3.2 MVAPICH2 2.0 to 2.3 Cray MPT 7.7.1 HPE MPI 1.1 Bullx MPI 1.2.7.x to 1.2.9.x	GNU Compiler C/C++ and Fortran 4.3.x to 8.1.x LLVM Clang 3.3 to 6.0 Arm C/C++ Compiler 18.0 to 18.3 Arm Fortran Compiler 18.0 to 18.3 Cray Compiling Environment 8.7.x	–
Intel Xeon Phi (knl)	Red Hat Enterprise Linux/CentOS 7 SuSE Linux Enterprise 12	Open MPI 1.8, 1.10, 2.0, 2.1, 3.0 and 3.1 Intel MPI 5.1.x to 2018.3 MPICH 3.1 to 3.2 MVAPICH2 2.0 to 2.3 Cray MPT 5.0.x to 7.7.x HPE MPI 1.1 Bullx MPI 1.2.7.x to 1.2.9.x	GNU Compiler C/C++ and Fortran 4.3.x to 8.1.x LLVM Clang 3.3 to 6.0 Intel Parallel Studio XE 2013.x to 2018.3 PGI Compiler 14.0 to 18.5 Cray Compiling Environment 8.3.x to 8.7.x	–
IBM Power (ppc64 and ppc64le)	Red Hat Enterprise Linux/CentOS 7.2-7.5	Open MPI 1.8 to 3.1 IBM Spectrum MPI 10.2	GNU Compiler C/C++ and Fortran 4.3.x to 8.1.x IBM XL C/C++ Compiler 13.1.x IBM XL Fortran Compiler 15.1.x IBM XL Compiler 16.1.x PGI Compiler 18.1, 18.5 and 18.7	Nvidia CUDA Toolkit 9.2

The following MPIs are also covered by our precompiled wrappers: Open MPI 1.6.x-1.10.x, and 3.0.x,

MPICH 2.x.x and 3.x.x, Intel MPI 4.x.x and 5.x.x, Cray MPT, Bullx MPI 1.2.7 and 1.2.8, MVAPICH 2.x.x.

Version control integration is supported for Git 1.7+, Mercurial 2.1+ and Subversion 1.6+.

The Arm profiling libraries must be explicitly linked with statically linked programs which mostly applies to the Cray X-Series.

Batch schedulers supported:

- SLURM 2.6.3+ and 14.03+
- PBS
- TORQUE
- Moab
- Oracle Grid Engine
- Loadleveler
- Cobalt.

Batch scheduling systems are supported through *Queue Templates*.

See section A.2 Integration with queuing systems for more information.

See section E.16 SLURM for more details about SLURM support.

Known issues

The most significant known issues for the latest release are summarized here:

MAP

The following known issues affect MAP.

- I/O metrics are not available on some systems, including Cray systems.
- CPU instruction metrics are only available on x86_64 systems.
- Thread activity is not sampled whilst a process is inside an MPI call with a duration spanning multiple samples. This can appear as 'uncategorized' (white) time in the Application activity bar when in the Pthread View. The uncategorized time will coincide with long running MPI calls.
- MAP does not support code that spawns new processes, such as `fork`, `exec` and `MPI_Comm_spawn`. In these cases MAP will only profile the original process.

XALT Wrapper

The XALT wrapper is known to cause several issues when used in conjunction with Arm Forge, such as:

- MPI programs cannot be debugged due to a hang during start up.
- Error messages are reported relating to the permissions on `qstat`.

For each case, the workaround is to disable the XALT wrapper. To disable the XALT wrapper, unload the XALT module.

MPICH 3

MPICH 3.0.3 and 3.0.4 do not work with the Arm Forge due to an MPICH defect. MPICH 3.1 is fully supported.

Open MPI

Message queue debugging does not work in Open MPI 1.8.1 to 1.8.5. This issue is fixed in Open MPI 1.8.6.

The following versions of Open MPI do not work with Arm Forge because of bugs in the Open MPI debug interface:

- Open MPI 2.1.0 to 2.1.2.
- Open MPI 3.0.0 when compiled with the Arm Compiler for HPC on Arm® v8 (AArch64) systems.
- Open MPI 3.0.x when compiled with some versions of the GNU compiler on Arm® v8 (AArch64) systems.
- Open MPI 3.x when compiled with some versions of IBM XLC/XLF or PGI compilers on IBM Power (PPC64le little-endian, POWER8, or POWER9) systems.
- Open MPI 3.1.0 and 3.1.1.

- Open MPI 3.x with any version of PMIx < 2.

To resolve any of the above issues, instead select *Open MPI (Compatibility)* for the MPI Implementation.

Open MPI 3.x on IBM Power with the GNU compiler

To use Open MPI 3.x with the GNU compiler on IBM Power systems, you might need to configure the Open MPI build with `CFLAGS=-fasynchronous-unwind-tables`. This fixes a startup bug where Arm Forge is unable to step out of `MPI_Init` into your main function. The startup bug is a result of a lack of debug information and optimization in the Open MPI library. If you already configure with `-g` then you do not need to add this extra flag. An example configure command is:

```
./configure --prefix=/software/openmpi-3.1.2
   CFLAGS=-fasynchronous-unwind-tables
```

If you do not have the option to recompile your MPI, then an alternative workaround is to select *Open MPI (Compatibility)* for the MPI Implementation.

CUDA

The following known issues affect CUDA:

- To debug or profile a CUDA program, compile the program with a version of the CUDA toolkit that matches the version of the installed CUDA driver. For example, if the CUDA 7.5 driver is installed, then you must use the CUDA 7.5 toolkit to compile your program.

  Note: Compiling with mismatched CUDA toolkit and CUDA driver versions will cause errors when debugging or profiling.

  Note: To force DDT to use a particular version of the CUDA debugger, set the `ALLINEA_FORCE_CUDA_VERSION` environment variable to a version number. For example, `ALLINEA_FORCE_CUDA_VERSION=7.5` for CUDA 7.5. This may cause issues due to CUDA version incompatibilities.

- GPU profiling is only supported when using a CUDA 8.0 toolkit with a CUDA 8.0 driver.
- Cray CCE 8.1.2 OpenACC and previous releases will fail to generate debug information for local variables in accelerated regions. Please install CCE 8.1.3.
- When debugging a CUDA application, adding watchpoints on either host or kernel code is not supported.
- When debugging a CUDA application, using the *Step threads together* box and *Run to here* to step into OpenMP regions is not supported. Breakpoints can be used to stop at the desired line.
- Stepping multiple warps simultaneously (e.g. those in the same block or kernel) is not supported in CUDA 9.x. Individual warps can be stepped sequentially to achieve the same effect.
- When CUDA is set to *Detect invalid accesses (memcheck)*, placing breakpoints in CUDA kernels is not supported.
- A driver issue in CUDA 9.1 prevents DDT from debugging CUDA GPU applications on Cray machines using Cray MPT (aprun). As a workaround launch the CUDA application outside of DDT and attach to it.

SLURM

On Cray X-series systems only *native* SLURM is supported, *hybrid* mode is not supported.

PGI compilers

Version 14.9 or later of the PGI compilers is required to compile the Arm MAP MPI wrappers as a static library.

64-bit Arm/Power platforms

For best operation, DDT and MAP require debug symbols for the runtime libraries to be installed in addition to debug symbols for the program itself.

F1 user guide

Sometimes on pressing “F1” the user guide may not display correctly. Some stale files appear to be able to corrupt the document browser. If “F1” leads to invisible documents, please remove these cached files by typing:

```
rm -r ~/.local/share/data/Arm
```

See also

See also additional known issues here:

Category	Known Issues
MPI Distribution	E MPI distribution notes and known issues
Compiler	F Compiler notes and known issues
Platform	G Platform notes and known issues
General	H General troubleshooting and known issues

MPI distribution notes and known issues

This appendix has brief notes on many of the MPI distributions supported by Arm DDT and Arm MAP.

Advice on settings and problems particular to a distribution are given here. Note that MAP supports fewer MPI distributions than DDT. See C Supported platforms for more details.

Berkeley UPC

Only the MPI transport is supported. Programs must be compiled with the `-tv` flag, for example:

```
upcc hello.c -o hello -g -tv
```

Bull MPI

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

Select *Bull MPI or Bull MPI 1 for* Bull MPI 1, *or Bull MPI 2* for Bull MPI 2 from the MPI implementations list. In the *mpirun arguments* box of the *Run* window you may also wish to specify the partition that you wish to use by adding the following:

```
-p partition_name
```

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

Cray MPT

This section only applies when using `aprun`. For `srun` ('Native' SLURM mode) see section E.16 SLURM.

DDT and MAP have been tested with Cray XT 5/6, XE6, XK6/7, and XC30 systems. DDT is able to launch and support debugging jobs in excess of 700,000 cores.

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

To attach to a running job on a Cray system the MOM nodes, that is those nodes where `aprun` is launched, must be reachable via `ssh` from the node where DDT is running, for example on a login node. DDT must connect to these nodes in order to launch debugging daemons on the compute nodes. Users can either specify the `aprun` host manually in the attach dialog when scanning for jobs, or configure a hosts list containing all MOM nodes.

Preloading of the memory debugging libraries is not supported with `aprun`.

If the program is dynamically linked, MAP supports preloading of the sampling libraries with `aprun` (requires `aprun`/ALPS 4.1 or later), else MAP requires Arm's sampling libraries to be linked with the application before running on this platform. Preloading is not supported in MPMD mode. See 16.2.2 Linking for a step-by-step guide.

Using DDT with Cray ATP (the Abnormal Termination Process)

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

To debug after a crash when an application is run with ATP but without a debugger, initialize the `ATP_HOLD_TIME` environment variable before launching the job. For a large Petascale system, a value of 5 is sufficient, giving 5 minutes for the attach to complete.

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

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

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

To debug the application at this point, launch DDT.

DDT can attach using the Attaching dialogs described in Section 5.9 Attaching to running programs, or given the PID of the `aprun` process, the debugging set can be specified from the command line.

For example, to attach to the entire job:

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

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

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

HP MPI

Select *HP MPI* as the MPI implementation.

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

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

If your job config file normally contains:

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

Then your template file should contain:


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

Also the *Submit Command* box should be filled with the following:

```
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*.

IBM PE

Ensure that poe is in your path, and select *IBM PE* as the MPI implementation.

A sample Loadleveler script, which starts debugging jobs on POE systems, is included in the *{installation-directory}*/templates directory.

To attach to already running POE jobs, SSH access to the compute nodes is required. Without SSH, DDT has no way to connect to the ranks running on the nodes.

Known issue: IBM PE 2.1 and newer currently do not provide the debugging interface required for MPI message queue debugging.

Intel MPI

Select *Intel MPI* from the MPI implementation list. DDT and MAP have been tested with Intel MPI 4.1.x, 5.0.x and later.

Make sure to pay attention to the changes in the mpivars.sh script with Intel MPI 5.0. You can pass it an argument to say whether you want to use the debug or release version of the MPI libraries. The default, if you omit the argument, is the release version, but message queue debugging will not work if you use this version. The debug version must be explicitly used.

DDT also supports the Intel Message Checker tool that is included in the Intel Trace Analyser and Collector software. A plugin for the Intel Trace Analyser and Collector version 7.1 is provided in DDT's plugins directory. Once you have installed the Intel Trace Analyser and Collector, you should make sure that the following directories are in your LD_LIBRARY_PATH:

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

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

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

Note that the Intel Message Checker will abort the job after 1 error by default. You can modify this by adding -genv VT_CHECK_MAX_ERRORS 0 to the *mpiun arguments* box in the *Run* window. See Intel's documentation for more details on this and other environment variable modifiers.

Attach dialog: DDT cannot automatically discover existing running MPI jobs that use Intel MPI if the processes are started using the `mpiexec` command (which uses the MPD process starting daemon). To attach to an existing job you will need to list all potential compute nodes individually in the dialog.

Please note the `mpiexec` method of starting MPI processes is deprecated by Intel and you are encouraged to use `mpirun` or `mpiexec.hydra` (which use the newer scalable Hydra process starting daemon). All processes that are started by either `mpirun` and `mpiexec.hydra` are discovered automatically by Arm DDT.

If you use Spectrum LSF as workload manager in combination with Intel MPI and you get for example one of the following errors:

- <*target program*> exited before it finished starting up. One or more processes were killed or died without warning
- <*target program*> encountered an error before it initialised the MPI environment. Thread 0 terminated with signal SIGKILL

or the job is killed otherwise during launching/attaching then you may need to set/export `I_MPI_LSF_USE_COLLECTIVE_LAUNCH=1` before executing the job. See Using IntelMPI under LSF quick guide and Resolve the problem of the Intel MPI job …hang in the cluster for more details.

MPC

DDT supports MPC version 2.5.0 and upwards. MPC is not supported by MAP.

In order to debug an MPC program, a script needs adding to the MPC installation. This script is obtained from Download MPC script and should be saved into the `bin/mpcrun_opt` subdirectory of your MPC framework installation.

MPC in the Run window

When the MPC framework is selected as the MPI implementation, there is an additional field in the MPI configuration within the Run window:

Number of MPC Tasks: The number of tasks that you wish to debug. MPC uses threads to split these tasks over the number of processes specified.

Also, the **mpirun arguments** field is replaced with the field:

mpcrun arguments: (optional): The arguments that are passed to `mpcrun`. This should be used for arguments to `mpcrun` not covered by the number of MPC tasks and number of processes fields.

An example usage is to override default threading model specified in the MPC configuration by entering `--multithreading=pthreads` for POSIX threads or `--multithreading=ethreads` for user-level threads.

The documentation for these arguments can be found at http://mpc.paratools.com/UsersGuide/Running. This field is only displayed if the selected MPI implementation is the MPC framework.

Note: The OpenMP options are not available in the Run window, as MPC uses the number of tasks to determine the number of OpenMP threads rather than `OMP_NUM_THREADS`.

MPC on the command line

There are two additional command-line arguments to DDT when using MPC that can be used as an alternative to configuration in the GUI.

`--mpc-task-nb` The total number of MPC tasks to be created.

`--mpc-process-nb` The total number of processes to be started by `mpcrun`.

MPICH 1 p4

Choose *MPICH 1 Standard* as the MPI implementation.

MPICH 1 p4 mpd

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

It should be possible to avoid these problems if `.bashrc` or `.tcshrc/.cshrc` are correct.

However, if unable to resolve these problems, you can pass `HOME` and `LD_LIBRARY_PATH`, plus any other environment variables that you need.

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

Alternatively from the command-line you may simply write:

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

Your shell will then fill in these values for you.

Choose *MPICH 1 Standard* as the MPI implementation.

MPICH 2

If you see the error `undefined reference to MPI_Status_c2f` while building the MAP libraries then you need to rebuild MPICH 2 with Fortran support. See 16.2.2 Linking for more information on linking.

MPICH 3

MPICH 3.0.3 and 3.0.4 do not work with Arm Forge due to an MPICH. MPICH 3.1 addresses this and is supported.

There are two MPICH 3 modes: Standard and Compatibility. If the standard mode does not work on your system select *MPICH 3 (Compatibility)* as the *MPI Implementation* on the *System Settings* page of the *Options* window.

MVAPICH 2

Known issue: If memory debugging is enabled in DDT, this will interfere with the on-demand connection system used by MVAPICH2 above a threshold process count and applications will fail to start. This threshold default value is 64. To work around this issue, set the environment variable `MV2_ON_DEMAND_THRESHOLD` to the maximum job size you expect on your system and then DDT will work with memory debugging enabled for all jobs. This setting should not be a system wide default as it may increase startup times for jobs and memory consumption.

MVAPICH 2 now offers `mpirun_rsh` instead of `mpirun` as a scalable launcher binary. To use this with DDT, from *File → Options* (*Arm Forge → Preferences* on Mac OS X) go to the *System* page, check *Override default mpirun path* and enter `mpirun_rsh`. You should also add `-hostfile <hosts>`, where `<hosts>` is the name of your hosts file, within the *mpirun_rsh arguments* field in the Run window.

To enable message Queue Support MVAPICH 2 must be compiled with the flags `--enable-debug --enable-sharedlib`. These are not set by default.

Open MPI

Arm Forge has been tested with Open MPI 1.6.x, 1.8.x, 1.10.x and 2.0.x. Select *Open MPI* from the list of MPI implementations.

Open MPI 2.1.3 works with Arm Forge. Previous versions of Open MPI 2.1.x do not work due to a bug in the Open MPI debug interface.

There are three different Open MPI choices in the list of MPI implementations to choose from in Arm Forge when debugging or profiling for Open MPI.

- *Open MPI* – the job is launched with a custom 'launch agent' that, in turn, launches the Arm daemons.
- *Open MPI (Compatibility)* – mpirun launches the Arm daemons directly. This startup method does not take advantage of Arm's scalable tree.
- *Open MPI for Cray XT/XE/XK/XC* – for Open MPI running on Cray XT/XE/XK/XC systems. This method is fully able to use Arm's scalable tree infrastructure.

  To launch with `aprun` (instead of `mpirun`) simply type the following on the command line:

  ```
  ddt --mpi="OpenMPI␣(Cray␣XT/XE/XK)" --mpiexec aprun [arguments]
  # or
  map --mpi="OpenMPI␣(Cray␣XT/XE/XK)" --mpiexec aprun [arguments]
  ```

The following section lists some known issues:

- Early versions of Open MPI 1.8 do not properly support message queue debugging. This is fixed in Open MPI 1.8.6.
- If you are using the 1.6.x series of Open MPI configured with the `--enable-orterun-prefix-by-default` flag then DDT requires patch release 1.6.3 or later due to a defect in earlier versions of the 1.6.x series.
- The version of Open MPI packaged with Ubuntu has the Open MPI debug libraries stripped. This prevents the *Message Queues* feature of DDT from working.

- With Open MPI 1.3.4 and Intel Compiler v11, the default build will optimize away a vital call during the startup protocol which means the default Open MPI start up will not work. If this is your combination, either update your Open MPI, or select *Open MPI (Compatibility)* instead as the DDT MPI Implementation.
- On Infiniband systems, Open MPI and CUDA can conflict in a manner that results in failure to start processes, or a failure for processes to be debuggable. To enable CUDA interoperability with Infiniband, set the CUDA environment variable `CUDA_NIC_INTEROP` to 1.

Platform MPI

Platform MPI 9.x is supported, but only with the `mpirun` command. Currently `mpiexec` is not supported.

SGI MPT / SGI Altix

For SGI use one of the following configurations:

- If using SGI MPT 2.10+, select SGI MPT (2.10+, batch) as the MPI implementation.
- If using SGI MPT 2.08+, select SGI MPT (2.08+, batch) as the MPI implementation.
- If using an older version of SGI MPT (2.07 or before) select SGI MPT as the MPI implementation.

If you are using SGI MPT with PBS or SLURM and would normally use `mpiexec_mpt` to launch your program you will need to use the `pbs-sgi-mpt.qtf` queue template file and select `SGI MPT` (`Batch`) as the MPI implementation.

If you are using SGI MPT with SLURM and would normally use `mpiexec_mpt` to launch your program you will need to use `srun --mpi=pmi2` directly.

`mpiexec_mpt` from versions of SGI MPT prior to 2.10 may prevent MAP from starting when preloading the Arm profiler and MPI wrapper libraries. Arm recommends you explicitly link your programs against these libraries to work around this problem.

Preloading the Arm profiler and MPI wrapper libraries is not supported in express launch mode. Arm recommends you explicitly link your programs against these libraries to work around this problem.

Some SGI systems cannot compile programs on the batch nodes (one reason might be because the `gcc` package is not installed). If this applies to your system you must explicitly compile the Arm MPI wrapper library using the `make-profiler-libraries` command and then explicitly link your programs against the Arm profiler and MPI wrapper libraries.

The `mpio.h` header file shipped with SGI MPT 2.09 and SGI MPT 2.10 contains a mismatch between the declaration of `MPI_File_set_view` and some other similar functions and their PMPI equivalents, for example `PMPI_File_set_view`. This prevents MAP from generating the MPI wrapper library. Please contact SGI for a fix.

SGI MPT 2.09 requires the `MPI_SUPPORT_DDT` environment variable to be set to `1` to avoid startup issues when debugging with DDT, or profiling with MAP.

SLURM

To start MPI programs using the `srun` command instead of your MPI's usual `mpirun` command (or equivalent) select *SLURM (MPMD)* as the *MPI Implementation* on the *System Settings* page of the *Op-*

tions.

While this option will work with most MPIs, it will not work with all. On Cray, 'Hybrid' SLURM mode (that is, SLURM + ALPS) is not supported. Instead, you must start your program with Cray's `aprun`. See Section E.3 Cray MPT.

SLURM may be used as a job scheduler with DDT and MAP through the use of a queue template file. See `templates/slurm.qtf` in the Arm Forge installation for an example and section A.2 Integration with queuing systems for more information on how to customize the template.

Spectrum MPI

Spectrum MPI 10.2 is supported for IBM Power (PPC64le little-endian) with the `mpirun` and `mpiexec` commands. Spectrum MPI 10.2 is additionally supported with the `jsrun` (PMIx mode) command.

When using jsrun 1.1.0 [Apr 16, 2018] or earlier, Arm Forge cannot correctly detect the MPI rank for each of your processes. A workaround for DDT can be found in section 8.17 Assigning MPI ranks.

Compiler notes and known issues

When compiling for a DDT debugging session, always compile with a minimal amount of optimization, or no optimization. Some compilers reorder instruction execution and omit debug information when compiled with optimization turned on.

AMD OpenCL compiler

Not supported by MAP.

The AMD OpenCL compiler can produce debuggable OpenCL binaries. However, the target must be the CPU rather than the GPU device. The build flags `-g -O0` must be used when building the OpenCL kernel, typically by setting the environment variable:

```
AMD_OCL_BUILD_OPTIONS_APPEND="-g␣-O0"
```

The example codes in the AMD OpenCL toolkit are able to run on the CPU by adding a parameter `--device cpu` and will result, with the above environment variable set, in debuggable OpenCL.

Arm Fortran compiler

Debugging of Fortran code may be incomplete or inaccurate. For more information, check the known issues section in the ARM HPC Compiler release notes.

Berkeley UPC compiler

Not supported by MAP.

The Berkeley UPC compiler is fully supported by Arm DDT, but only when using the MPI conduit (other conduits are not supported).

Warning: If you do not compile the program fixing the number of threads (using the `-fupc-threads-<numberOfThreads>` flag), a known issue arises at the end of the program execution.

Note: Source files must end with the extension `.upc` in order for UPC support to be enabled.

Cray compiler environment

DDT supports Cray Fast Track Debugging, however only certain versions of GDB support it:

- In DDT 19.0, it is supported in GDB 8.1 and 7.12.1.
- In DDT 18.2.1, it is supported in GDB 7.12.1 and 7.2.
- In DDT 5.0, it is only supported when using GDB 7.2, and not when using GDB 7.6.2.

To enable the supported versions of GDB, access the Systems Settings options by selecting File > Options > System (or Options > System, from the Welcome page), then choose from the Debugger options. To enable Fast Track Debugging, compile your program with `-Gfast` instead of `-g`.

See the *Using Cray Fast-track Debugging* section of the *Cray Programming Environment User's Guide* for more information.

Call-frame information can also be incorrectly recorded, which can sometimes lead to DDT stepping into a function instead of stepping over it. This may also result in time being allocated to incorrect functions in MAP.

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

Known Issue: If compiling static binaries then linking in the DDT memory debugging library is not straightforward for F90 applications. You will need to do the following:

1. Manually rerun the compiler command with the `-v` (verbose) option to get the linker command line. It is assumed that the object files are already created.
2. Run `ld` manually to produce the final statically linked executable. For this, the following path modifications will be needed in the previous `ld` command: Add `-L{ddt-path}/lib/64 -ldmalloc` immediately prior to where `-lc` is located. For multi-threaded programs you have to add `-ldmallocth -lpthread` before the `-lc` option.

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

Arm DDT fully supports the Cray UPC compiler. Not supported by MAP.

Compile scalar programs on Cray

To launch scalar code with aprun, using Arm Forge on Cray, your program needs to be linked with Cray PMI. With some configurations of the Cray compiler drivers, Cray PMI is discarded during linking. For static executables, consider using the `-Wl,-u,PMI_Init` compilation flags to preserve Cray PMI.

If using Arm MAP, see 16.2.2 Linking. If using aprun to launch your program, see H.2.2 Starting scalar programs with aprun. If using SLURM, see H.2.3 Starting scalar programs with srun

GNU

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

For GNU C++, large projects can often result in vast debug information size, which can lead to large memory usage by DDT's back end debuggers. For example, each instance of an STL class used in different object files will result in the compiler generating the same information in each object file.

The `-foptimize-sibling-calls` optimization (used in `-O2`, `-O3` and `-Os`) interfere with the detection of some OpenMP regions. If your code is affected with this issue add `-fno-optimize-sibling-calls` to disable it and allow MAP to detect all the OpenMP regions in your code.

Using the `-dwarf-2` flag together with the `-strict-dwarf` flag may cause problems in stack unwinding, resulting in a "cannot find the frame base" error. DWARF 2 does not provide all the information neceesary for unwinding the call stack, so many compilers add DWARF 3 extensions with the missing information. Using the `-strict-dwarf` flag prevents compilers from doing so, resulting in the aforementioned message. Removing `-strict-dwarf` should fix this problem.

GNU UPC

DDT also supports the GCC-UPC compiler (`upc_threads_model_process` only; the `pthread-tls` threads model is not supported). MAP does not support this.

To compile and install GCC UPC 4.8 without TLS it is necessary to modify the configuration file `path/to/upc/source/code/directory/libgupc/configure`, replacing all the entries *upc_cv_gcc_tls_supported=“yes”* to *upc_cv_gcc_tls_supported=“no”*.

To run a UPC program in DDT you have to select the MPI implementation “GCC libupc SMP (no TLS)”

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 when they are in directories other than that containing the executable. This flag has been known to fail for mpxlf95, and so there may be circumstances when you must right click in the project navigator and add additional paths to scan for source files.

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

Using IBM XL compilers with optimization level `-O2` or higher can lead to some partial traces. This occurs because MAP does not have enough information to fully unwind the call stack.

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

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

For the best OpenMP debug experience, compile your code with `-qsmp=omp:noopt` instead of `-qsmp=omp`. For more information about the issues you may encounter when debugging OpenMP, see 5.5 Debugging OpenMP programs.

DDT has been tested against the C compiler xlc version 13.1 and Fortran/Fortran 90 compiler xlf version 15.1 on Linux.

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

MAP only supports xlc and xlf on Linux.

Intel compilers

DDT and MAP have been tested with versions 13, 14, 16 and 17.

If you experience problems with missing or incomplete stack traces (for example `[partial trace]` entries in MAP or no stack traces for allocations in DDT’s *View Pointer Details* window) try recompiling your program with the `-fno-omit-frame-pointer` argument. The Intel compiler may omit frame pointers by default which can mean Arm Forge cannot properly discover your stack frames and you will be unable to see which lines of code your program has stopped at.

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

Some optimizations performed using Interprocedural Optimization (IPO), which is implicitly enabled by the `-O3` flag, can interfere with MAP's ability to display call stacks, making it more difficult to understand what the program is doing. To prevent this, it is recommended that IPO be disabled by adding `-no-ip -no-ipo` to the compiler flags. The `-no-ip` flag disables IPO within files while `-no-ipo` disables IPO between files.

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

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

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

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

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

Known Issue: If compiling static binaries, for example on a Cray XT/XE machine, then linking in the DDT memory debugging library is not straightforward for F90 applications. You need to manually rerun the last `ld` command (as seen with `ifort -v`) to include `-L{ddt-path}/lib/64-ldmalloc` in two locations:

1. Immediately prior to where `-lc` is located.
2. Include the `-zmuldefs` option at the start of the `ld` line.

STL sets, maps and multi-maps cannot be fully explored as only the total number of items is displayed. Other data types are unaffected.

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

Pathscale EKO compilers

Not supported by MAP.

There are some known issues as shown in the following list:

- The default Fortran compiler options may not generate enough information for DDT to show where memory was allocated from. *View Pointer Details* will not show which line of source code memory was allocated from. To enable this, compile and link with the following flags:


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

- For C programs, simply compiling with `-g` is sufficient.
- When using the Fortran compiler, you may have to place breakpoints in `myfile.i` instead of `myfile.f90` or `myfile.F90`. Arm is currently investigating this. Please contact Arm support at Arm support if this applies to your code.
- Procedure names in modules often have extra information appended to them. This does not otherwise affect the operation of DDT with the Pathscale compiler.
- The Pathscale 3.1 OpenMP library has an issue which makes it incompatible with programs that call the fork system call on some machines.
- Some versions of the Pathscale compiler (for example, 3.1) do not emit complete DWARF debugging information for `typedef`'ed structures. These may show up in DDT with a `void` type instead of the expected type.
- Multi-dimensional allocatable arrays can also be given incorrect dimension upper or lower bounds. This has only been reproduced for large arrays, small arrays seem to be unaffected. This has been observed with version 3.2 of the compiler, newer and older versions may also exhibit the same issue.

Portland Group compilers

DDT has been tested with Portland Tools 9 onwards.

MAP has been tested with version 14 of the PGI compilers. Older versions are **not** supported as they do not allow line level profiling. Always compile with `-Meh_frame` to provide sufficient information for profiling.

If you experience problems with missing or incomplete stack traces (that is `[partial trace]` entries in MAP or no stack traces for allocations in DDT's *View Pointer Details* window) try recompiling your program with the `-Mframe` argument. The PGI compiler may omit frame pointers by default which can mean Arm Forge cannot properly discover your stack frames and you will be unable to see which lines of code your program has stopped at.

Some known issues are listed here:

- Included files in Fortran 90 generate incorrect debug information with respect to file and line information. The information gives line numbers which refer to line numbers from the *included* file but give the *including* file as the file.
- The PGI compiler may emit incorrect line number information for templated C++ functions or omit it entirely. This may cause DDT to show your program on a different line to the one expected, and also mean that breakpoints may not function as expected.
- The PGI compiler does not emit the correct debugging tags for proper support of inheritance in C++, which prevents viewing of base class members.
- When using memory debugging with statically linked PGI executables (`-Bstatic`) because of the in-built ordering of library linkage for F77/F90, you will need to add a `localrc` file to your PGI installation which defines the correct linkage when using DDT and (static) memory debugging. To your `{pgi-path}/bin/localrc` append the following:

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



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

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

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

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

- When you pass an array splice as an argument to a subroutine that has an assumed shape array argument, the offset of the array splice is currently ignored by DDT. Please contact Arm support at Arm support if this affects you.
- DDT may show extra symbols for pointers to arrays and some other types. For example if your program uses the variable `ialloc2d` then the symbol `ialloc2d$sd` may also be displayed. The extra symbols are added by the compiler and may be ignored.
- The Portland compiler also wraps F90 allocations in a compiler-handled allocation area, rather than directly using the systems memory allocation libraries directly for each allocate statement. This means that bounds protection (Guard Pages) cannot function correctly with this compiler.
- DDT passes on all variables that the compiler has told gdb to be in scope for a routine. For the PGI compiler this can include internal variables and variables from Fortran modules even when the only clause has been used to restrict access. DDT is unable to restrict the list to variables actually used in application code.
- Versions of the PGI compiler prior to 14.9 are unable to compile a static version of the Arm MPI wrapper library, attempting to do so will result in messages such as "`Error: symbol 'MPI_-F_MPI_IN_PLACE' can not be both weak and common`". This is due to a bug in the PGI compiler's weak object support.

  For information concerning the Portland Accelerator model and debugging this with DDT, please see the 14 CUDA GPU debugging of this userguide.

Platform notes and known issues

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

CRAY

There are a number of issues you should be aware of:

- For 'Native' SLURM mode systems GDB 7.6.2 must be selected as the debugger or the job might not start properly. See section A.5.1 System for more information on selecting the debugger.
- MAP users on Cray need to read 16.2.1 Debugging symbols and 16.2.5 Static linking on Cray X-Series systems. Arm supplies module files in `FORGE_INSTALLATION_PATH/share/modules/cray`.

  See 16.2.6 Dynamic and static linking on Cray X-Series systems using the modules environment.
- Note that the default mode for compilers on this platform is to link statically. Section F.9 Portland Group compilers describes how to ensure that DDT's memory debugging capabilities will work with the PGI compilers in this mode.
- Message queue debugging is not provided by the XT/XE/XK environment.
- Cray GPU debugging requires a working `TMPDIR` to be available, if `/tmp` is not available. It is important that this directory is not a shared filesystem such as NFS or Lustre. To set `TMPDIR` for the compute nodes only use the `DDT_BACKEND_TMPDIR` environment variable instead. DDT will automatically propagate this environment variable to the compute nodes.
- Running single process scalar codes, that is *non*-MPI/SHMEM/UPC applications, on the compute nodes requires an extra step, as these are required to be executed by `aprun` but `aprun` will not execute these via the ordinary debug-supporting protocols.

  The preferred and simple workaround is to use the `.qtf` templates, for example `cray-slurm.qtf` or `cray-pbs.qtf`, which handle this automatically by (for non-MPI codes) ensuring that an alternative protocol is followed. To use these `qtf` files, select *File* → *Options* (*Arm Forge* → *Preferences* on Mac OS X) , go to the *Job Submission* page and enable submission via the queue, and ensure that the *Also submit scalar jobs via the queue* setting is enabled. The change is to explicitly use aprun for non-MPI processes and this can be seen in the provided queue template files:

  ```
  if [ "MPI_TAG" == "none" ]; then
     aprun -n 1 env AUTO_LAUNCH_TAG
  else
     AUTO_LAUNCH_TAG
  fi
  ```

- Running a dynamically-linked single process non-MPI program that will run on a compute node, that is non-MPI CUDA or OpenACC code, will require an additional flag to the compiler: `-target=native`. This prevents the compiler linking in the MPI job launch routines that will otherwise interfere with debuggers on this platform. Alternatively, convert the program to an MPI one by adding `MPI_Init` and `MPI_Finalize` statements and run it as a one-process MPI job.

GNU/Linux systems

General

There are a number of items you should be aware of:

- When using a 64-bit Linux please note that it is essential to use the 64-bit version of Arm Forge on this platform. This applies regardless of whether the debugged program is 32-bit or 64-bit.
- POSIX thread cancellation does not work when running under a debugger. This is because the 'signal info' associated with a signal is lost when the signal is intercepted and sent again by the debugger, causing the cancellation request to be ignored by the receiving thread. More generally the 'signal info' associated with a signal is not available when running under a debugger.
- Some 64-bit GNU/Linux systems which have a bug in the GNU C library, specifically `libthread_db.so.1`. This can crash the debugger when debugging multi-threaded programs. Check with your Linux distribution for a fix. As a workaround you can try compiling your program as a statically linked executable using the `-static` compiler flag.
- For the Arm architecture breakpoints can be unreliable and will randomly be passed without stopping for some multicore processors (including the NVIDIA Tegra 2) unless a kernel option (fix) is built-in. The required kernel option is:

  ```
  CONFIG_ARM_ERRATA_720789=y
  ```

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

SUSE Linux

There are a number of known issues you should be aware of:

- There is a known issue with SUSE 11 which *may* cause you to experience a crash similar to the following:

  ```
  Other: *** glibc detected *** /home/user/wave_c: free(): invalid
     pointer: 0x00007ffff7e02a80 ***
  Other: ======= Backtrace: =========
  Other: /lib64/libc.so.6[0x7fffeef81118]
  Other: /lib64/libc.so.6(cfree+0x76)[0x7fffeef82c76]
  Other: /lib64/libnss_nis.so.2(_nss_nis_getpwuid_r+0xe9)[0
     x7fffecd4f089]
  Other: /lib64/libnss_compat.so.2[0x7fffed125ab8]
  ```

  The implementation of `libnss_nis.so.2` attempts to resolve symbol names using its direct dependencies before using the global namespace. This causes the libc implementation of, for example, `free` to be linked instead of the intended `libdmalloc` implementation.

  If you encounter this crash, then the only solution is to disable memory debugging and contact SUSE about the availability of a fix.
- There is a known issue with SUSE 11 using the 2.6 kernel where some small fraction of samples may have invalid or incorrect stack traces. This has been observed on the 2.6.27.19-5 kernel and typically affected $<1\%$ of samples. This is caused by some bad unwind information in the kernel's `vdso`, the Virtual Dynamic Shared Object. The solution is to upgrade to a newer version of the kernel (>3).

Attaching

To attach to a running job:

1. Open the *Attach* window by clicking on the *Attach* button on the Welcome page.
2. DDT needs to know which login / batch node `runjob` is running on. Click the *Choose Hosts...* button to add the necessary login / batch node if not already present. You must be able to SSH into the login / batch node without a password.
3. Select the *Automatically-detected jobs* tab. Do not use the *List of processes* tab.
4. Optionally specify a subset of ranks to attach to in the *Attach to processes* box.
5. Click the *Attach to...* button.

The following caveats apply:

- Reattaching to a job is not supported. You may only attach to a job once.
- No other tool must be attached, or have been attached, to the job.
- It is possible to attach to a subset of ranks. However, because reattaching is not supported, it is not possible to subsequently change the subset.
- It may take a little time for a job to show up in the Attach window after you submit it. If a newly started job does not show up wait a while then click *Rescan nodes*.

Intel Xeon

Intel Xeon processors starting with Sandy Bridge include Running Average Power Limit (RAPL) counters. MAP can use the RAPL counters to provide energy and power consumption information for your programs.

Enabling RAPL energy and power counters when profiling

To enable the RAPL counters to be read by MAP you must load the `intel_rapl` kernel module.

The `intel_rapl` module is included in Linux kernel releases 3.13 and later. For testing purposes Arm have backported the `powercap` and `intel_rapl` modules for older kernel releases. You may download the backported modules from:

Download backported modules

Note: These backported modules are unsupported and should be used for testing purposes only. No support is provided by Arm, your system vendor or the Linux kernel team for the backported modules.

Intel Xeon Phi (Knight's Landing)

The Intel Xeon Phi Knight's Landing platform is only supported in self-hosted mode, like an x86_64 platform.

You may experience higher than normal overhead when using MAP on this platform.

See section H.9.12 for more information.

NVIDIA CUDA

CUDA known issues

There are a number of issues you should be aware of:

- DDT's memory leak reports do not track GPU memory leaks.
- Debugging paired CPU/GPU core files is possible but is not yet fully supported.
- CUDA metrics in MAP are not available for statically-linked programs.
- CUDA metrics in MAP are measured at the node level, not the card level.

Arm

Arm®v8 (AArch64) known issues

There are a number of issues you should be aware of:

- For best operation, DDT requires debug symbols for the runtime libraries to be installed in addition to debug symbols for the program itself. In particular, DDT may show the incorrect values for local variables in program code if the program is currently stopped inside a runtime library. At a minimum Arm recommends the glibc and OpenMP (if applicable) debug symbols are installed.
- For best operation, MAP requires debug symbols for the runtime libraries to be installed in addition to debug symbols for the program itself. In particular, MAP may report time in partial traces or unknown locations without debug symbols. At a minimum Arm recommends the glibc and OpenMP (if applicable) debug symbols are installed.
- MAP does not support CPU instruction metrics on this platform. Linux perf event metrics are available instead. To ensure access to performance counters is not restricted, use `sysctl -w kernel.perf_event_paranoid=0`.
- MAP may fail to finalize a profiling session if the cores are oversubscribed on AArch64 platforms. For example, this issue is likely to occur when attempting to profile a 64 process MPI program on a machine with only 8 cores. This issue will appear as a hang after finishing a profile or after pressing the 'Stop and analyze' button.

POWER8 and POWER9 (POWER 64-bit)

Supported features

Split DWARF (Fission) and compressed DWARF are supported by DDT and MAP. Benefits include smaller debug information size, and potentially less memory consumption in DDT due to the ability to load debug symbols on demand. For example if you use the following flags with GCC (which requires using the Binutils Gold linker):

```
gcc -gdwarf-4 -gsplit-dwarf -fdebug-types-section -Wl,-fuse-ld=gold
   ,--gdb-index,--compress-debug-sections=zlib myprogram.c
```

IBM XLC 13.1.7 requires `-qdebug=NDWFSTR -gsplit-dwarf -Wl,--gdb-index,--compress-debug-sections=zlib` flags. Configure the compiler to use the gold linker.

Known issues

There are a number of issues you should be aware of:

- For best operation, DDT and MAP require debug symbols for the runtime libraries to be installed in addition to debug symbols for the program itself. Without debug symbols, DDT may show the incorrect values for local variables in program code if the program is currently stopped inside a runtime library. Similarly, MAP may report time in partial traces or unknown locations without debug symbols. At a minimum Arm recommends the glibc and OpenMP (if applicable) debug symbols are installed. Please refer to your operating system's documentation for instructions on how to install debug symbols.
- MAP does not support CPU instruction metrics on this platform. Linux perf event metrics are available instead. To ensure access to performance counters is not restricted, use `sysctl -w kernel.perf_event_paranoid=0`.

MAC OS X

The following menu items are not supported:

- *Edit* → *Special Characters...*
- *Edit* → *Start Dictation*
- *View* → *Enter Full Screen*
- *View* → *Show Tab Bar*

General troubleshooting and known issues

If you have problems with Arm Forge products, the topics in this section may help you. Additionally, check the support pages on the Arm Developer website, and make sure you have the latest version of the product.

General troubleshooting

Problems starting the GUI

If the GUI is unable to start, this can be due to one of the following reasons:

1. Cannot connect to an X server. If you are running on a remote machine, make sure that your `DISPLAY` variable is set appropriately and that you can run simple X applications such as `xterm` from the same command-line.
2. The license file is invalid. In this case the software will issue an error message. You should verify that you have a license file for the correct product in the license directory and check that the date inside it is still valid. If the program still refuses to start, please contact Arm support at Arm support.
3. You are using Licence Server, but the Arm Forge products cannot connect to it. See the Licence Server user guide for more information on troubleshooting these problems.

Problems reading this document

If when pressing F1 a blank screen appears instead of this document, there may be corrupt files that are preventing the documentation system (Qt Assistant) from starting. You can resolve this by removing the stale files, which are found in `$HOME/.local/share/data/Allinea`.

Starting a program

Starting scalar programs

Before attempting to start a program, check F Compiler notes and known issues and ensure it is compiled correctly.

There are a number of possible sources for problems. The most common is, for users with a multi-process license, that the *Run Without MPI Support* check box has not been checked. If the software reports a problem with MPI and you know your program is not using MPI, then this is usually the cause. If you have checked this box and the software still mentions MPI then please contact Arm support at Arm support.

Other potential problems are:

- A previous Arm 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.
- Arm Forge products' backend daemon, `ddt-debugger`, is missing from the `bin` directory. In this case you should check your installation, and contact Arm support at Arm support.

Starting scalar programs with aprun

For compilation, see F.4.1 Compile scalar programs on Cray. The following environment variables should be exported:

```
export ALLINEA_MPI_INIT=main
export ALLINEA_HOLD_MPI_INIT=1
```

Instead of setting a breakpoint in the default `MPI_Init` location, these environment variables set a breakpoint in main, and hold the program there.

If using compatibility launch with a scalar program, the run dialog automatically detects Cray MPI even though it is a non-MPI program. Keep MPI checked, set one process, and click run.

If the above environment variables do not work, try an alternative solution by exporting:

```
export ALLINEA_STOP_AT_MAIN=1
```

`ALLINEA_STOP_AT_MAIN` holds the program wherever it was when it attached. This can be before main. For Arm DDT, first you should set a breakpoint in the main of your program. Next, run to this breakpoint.

Starting scalar programs with srun

Export the following environment variables:

```
export ALLINEA_MPI_INIT=main
export ALLINEA_HOLD_MPI_INIT=1
```

Instead of setting a breakpoint in the default `MPI_Init` location, these environment variables set a breakpoint in main, and hold the program there.

If using compatibility launch with a scalar program, the run dialog automatically detects SLURM. Keep MPI checked, set one process and click run.

If the above environment variables do not work, try an alternative solution by exporting:

```
export ALLINEA_STOP_AT_MAIN=1
```

`ALLINEA_STOP_AT_MAIN` holds the program wherever it was when it attached. This can be before main. For Arm DDT, first you should set a breakpoint in the main of your program. Next, run to this breakpoint.

Starting multi-process programs

If you encounter problems while starting an MPI program, 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 working correctly by running a job, without any of Arm Forge products applied, such as the example in the examples directory.

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


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

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

Sometimes problems are caused by environment variables not propagating to the remote nodes while starting a job. To a large extent, the solution to these problems depends on the MPI implementation that is being used.

In the simplest case, for rsh based systems such as a default MPICH 1 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 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 Arm support for advice at Arm support.

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 1 on Redhat with SMP support built in.

To check for time-out problems, set the `ALLINEA_NO_TIMEOUT` environment variable to 1 before launching the GUI and see if further progress is made. This is not a solution, but aids the diagnosis. If all processes now start, please contact Arm for support at Arm support.

No shared home directory

If your home directory is not accessible to all the nodes in your cluster then your jobs may fail to start.

To resolve the problem open the file `~/.allinea/system.config` in a text editor. Change the `shared directory` option in the `[startup]` section so it points to a directory that is available and shared by all the nodes. If no such directory exists, change the `use session cookies` option to `no` instead.

DDT or MAP cannot find your hosts or the executable

This can happen when attempting to attach to a process running on other machines. Ensure that the host name(s) that DDT complains about are reachable using `ping`.

If DDT fails to find the executable, ensure that it is available in the same directory on every machine.

See section A.4 Connecting to remote programs (remote-exec) for more information on configuring access to remote machines.

The progress bar does not move and Arm Forge times out

It is possible that the program `ddt-debugger` has not 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 Arm

Forge 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 Arm Forge 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 MPI distribution notes and known issues.

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

If your home directory is not accessible by all the nodes in your cluster then your jobs may fail to start in this fashion.

See section H.2.5 No shared home directory.

Attaching

The system does not allow connecting debuggers to processes (Fedora, Ubuntu)

The Ubuntu ptrace scope control feature does not allow a process to attach to other processes it did not launch directly.

See http://wiki.ubuntu.com/Security/Features#ptrace for details.

To disable this feature until the next reboot run the following command:

```
echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope
```

To disable it permanently, add this line to `/etc/sysctl.d/10-ptrace.conf` (or `/etc/sysctl.conf`):

```
kernel.yama.ptrace_scope = 0
```

This will take effect after the next reboot.

On Fedora, ptrace may be blocked by SELinux *in addition to* Yama. See section H.3.2.

The system does not allow connecting debuggers to processes (Fedora, Red Hat)

The `deny_ptrace` boolean in SELinux (used by Fedora and Red Hat) does not allow a process to attach to other processes it did not launch directly.

See http://fedoraproject.org/wiki/Features/SELinuxDenyPtrace for details.

To disable this feature until the next reboot run the following command:

```
setsebool deny_ptrace 0
```

To disable it permanently run this command:

```
setsebool -P deny_ptrace 0
```

As of Fedora 22, ptrace may be blocked by Yama *in addition to* the SELinux boolean. See section H.3.1.

Running processes do not show up in the attach window

Running processes that do not show up in the attach window 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 are running on your local machine) or with the output of `hostname` on the desired machine.

Secondly try running `/path/to/arm/forge/libexec/remote-exec` manually.

For example, `/path/to/arm/forge/libexec/remote-exec<hostname>ls`. Then 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.

See also A.4 Connecting to remote programs (remote-exec).

If you still experience problems with your script then contact Arm support for assistance at Arm support.

Source Viewer

No variables or line number information

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

Source code does not appear when you start Arm Forge

If you cannot see any text at all, perhaps the default selected font is not installed on your system. Go to *File* → *Options* (*Arm Forge* → *Preferences* on Mac OS X) and choose a fixed width font such as *Courier* and you should now be able to see the code.

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

If the problem persists, contact Arm support at Arm support.

Code folding does not work for OpenACC/OpenMP pragmas

This is a known issue. If an OpenACC or OpenMP pragma is associated with a multi-line loop, then the loop block may be folded instead.

Input/Output

Output to stderr is not displayed

Arm Forge automatically captures anything written to `stdout` / `stderr` and display it.

Some shells, such as `csh`, do not support this feature in which case you may see your `stderr` mixed with `stdout`, or you may not see it at all.

In any case Arm strongly recommends writing program output to files instead, since the MPI specification does not cover `stdout` / `stderr` behavior.

Unwind errors

When using MAP you may see errors reported in the output of the form:

```
Arm Sampler: 3 libunwind: Unspecified (general) error (4/172 samples)
Arm Sampler: 3 Maximum backtrace size in sampler exceeded, stack too
   deep. (1/172 samples)
```

These indicate that MAP was only able to obtain a partial stack trace for the sample. If the proportion of samples that generate such errors is low, then they can safely be ignored.

If a large proportion of samples exhibit these errors, then consult the advice on partial traces in F.7 Intel compilers or F.9 Portland Group compilers if you are using these compilers.

If this does not help, then please contact Arm support at Arm support.

Controlling a program

Program jumps forwards and backwards when stepping through it

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

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

DDT may stop responding when using the Step Threads Together option

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

A workaround is to set a breakpoint at the last statement executed by the thread and turn off *Step Threads Together* when the thread stops at the breakpoint.

If this problem affects you please contact Arm support at Arm support.

Evaluating variables

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

Some compilers produce faulty debug information, forcing DDT to enter a function during the *prologue* or the variable may not yet be in scope.

In this region, which appears to be the first line of the function, some variables have not been initialized yet. To view all the variables with their correct values, it may be necessary to play or step to the next line of the function.

Incorrect values printed for Fortran array

Pointers to non-contiguous array blocks, allocatable arrays using strides, are not supported.

If this issue affects you, please contact Arm support at Arm supportfor a workaround or fix.

There are also many compiler limitations that can cause this. See Appendix F for details.

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

The *Locals*, *Current Line* and *Evaluate* views may not show the contents of these multi-dimensional arrays inside an array of derived types.

However, you can view the contents of the array by clicking on its name and dragging it into the evaluate window as an item on its own, or by using the MDA.

C++ STL types are not pretty printed

The pretty printers provided with DDT are compatible with GNU compilers version 4.7 and above, and Intel C++ version 12 and above.

Memory debugging

The View Pointer Details window says a pointer is valid but does not show you which line of code it was allocated on

The Pathscale compilers have known issues that can cause this.

Please see the compiler notes in section C of this appendix for more details.

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

If this happens with another compiler, please contact Arm support at Arm support with the vendor and version number of your compiler.

`mprotect fails` error when using memory debugging with guard pages

This can happen if your program makes more than 32768 allocations; a limit in the kernel prevents DDT from allocating more protected regions than this. Your options are:

- Running `echo 123456 >/proc/sys/vm/max_map_count` (requires root) will increase the limit to 61728 (123456 / 2, as some allocations use multiple maps).
- Disable guard pages completely. This will hinder DDT's ability to detect heap over/underflows.
- Disable guard pages temporarily. You can disable them at program start, add a breakpoint before the allocations you wish to add guard pages for, and then reenable the feature.

See 12.3 Configuration for information on how to disable guard pages.

Allocations made before or during `MPI_Init` show up in *Current Memory Usage* but have no associated stack back trace

Memory allocations that are made before or during `MPI_Init` appear in Current Memory Usage along with any allocations made afterwards.

However, the call stack at the time of the allocation is not recorded for these allocations and will not show up in the Current Memory Usage window.

Deadlock when calling `printf` or `malloc` from a signal handler

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

POSIX does not require `malloc` to be async-signal-safe but some programs may expect this behavior.

For example, a program that calls `printf` from a signal handler may deadlock when memory debugging is enabled in DDT since the C library implementation of `printf` may call `malloc`.

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

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

Program runs more slowly with Memory Debugging enabled

The Memory Debugging library performs more checks than the normal runtime's memory allocation routines.

However, those checks also makes the library slower.

If your program is running too slow when Memory Debugging is enabled there are a number of options you can change to speed it up.

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

You can increase the heap check interval from the default of 100 to a higher value. The heap check interval controls how many allocations may occur between full checks of the heap, which may take some time.

A higher setting (1000 or above) is recommended if your program allocates and deallocates memory very frequently, for example from inside a computation loop.

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

MAP specific issues

My compiler is inlining functions

While compilers may inline functions, their ability to include sufficient information to reconstruct the original call tree varies between vendors. Arm recommends using the following flags:

- Intel: `-g -O3 -fno-inline-functions`
- Intel 17+: `-g -fno-inline -no-ip -no-ipo -fno-omit-frame-pointer -O3`
- PGI: `-g -O3 -Meh_frame`
- GNU: `-g -O3 -fno-inline`
- Cray: `-G2 -O3 -h ipa0`
- IBM: `-g -O3 -qnoinline`

Be aware that some compilers may still inline functions even when explicitly asked not to.

There is typically some small performance penalty for disabling function inlining or enabling profiling information.

MAP will work fine, but you will often see time inside an inlined function being attributed to its parent in the Stacks view. The Source Code view should be largely unaffected.

Tail call optimization

A function may return the result of calling another function, for example:

```
int someFunction()
{
    ...
    return otherFunction();
}
```

In this case the compiler may change the call to `otherFunction` into a jump. This means that, when inside `otherFunction`, the calling function, `someFunction`, no longer appears on the stack.

This optimization is called tail recursion optimization. It may be disabled for the GNU C compiler by passing the `-fno-optimize-sibling-calls` argument to `gcc`.

MPI wrapper libraries

Unlike DDT, MAP wraps MPI calls in a custom shared library. A precompiled wrapper is copied that is compatible with your system, or one is built for your system each time you run MAP.

See section C.2 MAP for the list of supported MPIs.

You can also try setting `ALLINEA_WRAPPER_COMPILE=1` and `MPICC` directly:

```
$ MPICC=my-mpicc-command bin/map -n 16 ./wave_c
```

If you have problems please contact Arm support at Arm support.

Thread support limitations

MAP provides limited support for programs when threading support is set to `MPI_THREAD_SERIALIZED` or `MPI_THREAD_MULTIPLE` in the call to `MPI_Init_thread`.

MPI activity on non-main threads will contribute towards the MPI-time of the program, but not the MPI metric graphs.

Additionally, MPI activity on a non-main thread may result in additional profiling overhead due to the mechanism employed by MAP for detecting MPI activity.

It is recommended that the pthread view mode is used for interpreting MPI activity instead of the OpenMP view mode, since OpenMP view mode will scale MPI-time depending on the resources requested. Hence, non-main thread MPI activity may provide nonintuitive results when detected outside of OpenMP regions.

Warnings are displayed when the user initiates and completes profiling a program which sets `MPI_THREAD_SERIALIZED` or `MPI_THREAD_MULTIPLE` as the required thread support.

MAP does fully support calling `MPI_Init_thread` with either `MPI_THREAD_SINGLE` or `MPI_THREAD_FUNNELED` specified as the required thread support.

It should be noted that the requirements that the MPI specification make on programs using `MPI_THREAD_FUNNELED` are the same as made by MAP: *all MPI calls must be made on the thread that called `MPI_Init_thread`.*

In many cases, multi-threaded MPI programs can be refactored such that they comply with this restriction.

No thread activity while blocking on an MPI call

Unfortunately MAP is currently unable to record thread activity on a process where a long-duration MPI call is in progress.

If you have an MPI call that takes a significant amount of time to complete, as indicated by a sawtooth on the *MPI call duration* metric graph, MAP will display no thread activity for the process executing that call for most of that MPI call's duration.

See also section 24.6.

I am not getting enough samples

By default starting sampling interval is every 20ms, but if you get warnings about too few samples on a fast run, or want more detail in the results, you can change the sampling rate.

To increase the interval to every 10ms set environment variable `ALLINEA_SAMPLER_INTERVAL=10`.

Note: Sampling frequency is automatically decreased over time to ensure a manageable amount of data is collected whatever the length of the run.

Increasing the sampling frequency is not recommended if there are lots of threads and/or very deep stacks in the target program as this may not leave sufficient time to complete one sample before the next sample is started.

Note: Whether OpenMP is enabled or disabled in MAP, the final script or scheduler values set for `OMP_NUM_THREADS` will be used to calculate the sampling interval per thread (`ALLINEA_SAMPLER_INTERVAL_PER_THREAD`). When configuring your job for submission, check whether your final submission script, scheduler or the MAP GUI has a default value for `OMP_NUM_THREADS`.

Note: Custom values for `ALLINEA_SAMPLER_INTERVAL` will be overwritten by values set from the combination of `ALLINEA_SAMPLER_INTERVAL_PER_THREAD` and the *expected* number of threads (from `OMP_NUM_THREADS`).

I just see main (external code) and nothing else

This can happen if you compile without `-g`. It can also happen if you move the executable out of the directory it was compiled in.

Check your compile line includes `-g` and try right-clicking on the Project Files panel in MAP and choosing *Add Source Directory…*.

Contact Arm support at Arm support if you have any further issues.

MAP is reporting time spent in a function definition

Any overheads involved in setting up a function call (pushing arguments to the stack and so on) are usually assigned to the function definition. Some compilers may assign them to the opening brace '{' and closing brace '}' instead.

If this function has been inlined, the situation becomes further complicated and any setup time, such as for allocating space for arrays, is often assigned to the definition line of the enclosing function.

MAP is not correctly identifying vectorized instructions

The instructions identified as vectorized (packed) are listed here:

- **Packed floating-point instructions:** `addpd addps addsubpd addsubps andnpd andnps andpd andps divpd divps dppd dpps haddpd haddps hsubpd hsubps maxpd maxps minpd minps mulpd mulps rcpps rsqrtps sqrtpd sqrtps subpd subps`
- **Packed integer instructions:** `mpsadbw pabsb pabsd pabsw paddb paddd paddq paddsb paddsw paddusb paddusw paddw palignr pavgb pavgw phaddd phaddsw phaddw phminposuw phsubd phsubsw phsubw pmaddubsw pmaddwd pmaxsb pmaxsd pmaxsw pmaxub pmaxud pmaxuw pminsb pminsd pminsw pminub pminud pminuw pmuldq pmulhrsw pmulhuw pmulhw pmulld pmullw pmuludq pshufb pshufw psignb psignd psignw pslld psllq psllw psrad psraw psrld psrlq psrlw psubb psubd psubq psubsb psubsw psubusb psubusw psubw`

Arm also identifies the AVX-2 variants of these instructions, with a "`v`" prefix.

Contact Arm support at Arm support if you believe your code contains vectorized instructions that have not been listed and are not being identified in the *CPU floating-point/integer vector* metrics.

Linking with the static MAP sampler library fails with an undefined reference to `__real_dlopen`

When linking with the static MAP sampler library you may get undefined reference errors similar to the following:

```
../lib/64/libmap-sampler.a(dl.o): In function `__wrap_dlopen':
/build/overnight/ddt-2015-01-28-12322/code/ddt/map/sampler/build64-
   static/../src/dl.c:21:␣undefined␣reference␣to␣`__real_dlopen'
../lib/64/libmap-sampler.a(dl.o): In function `__wrap_dlclose':
/build/overnight/ddt-2015-01-28-12322/code/ddt/map/sampler/build64-
   static/../src/dl.c:28:␣undefined␣reference␣to␣`__real_dlclose'
collect2: ld returned 1 exit status
```


To avoid these errors follow the instructions in section 16.2.4 Static linking.

Note the use of the `-Wl,@/home/user/myprogram/allinea-profiler.ld` syntax.

Linking with the static MAP sampler library fails with FDE overlap errors

When linking with the static MAP sampler library you may get FDE overlap errors similar to:

```
ld: .eh_frame_hdr table[791] FDE at 0000000000822830 overlaps table
   [792] FDE at 0000000000825788
```

This can occur when the version of binutils on a system has been upgraded to 2.25 or later and is most common seen on Cray machines using CCE 8.5.0 or higher.

To fix this issue rerun `make-profiler-libraries --lib-type=static` and use the freshly generated static libraries and `allinea-profiler.ld` to link these with your program.

See section 16.2.4 Static linking for more details.

If you are *not* using a Cray or SUSE build of Arm Forge and you require a binutils 2.25 compatible static library please contact Arm support at Arm support.

The error message occurs because the version of `libmap-sampler.a` you attempted to link was not compatible with the version of `ld` in binutils versions $\geq$ 2.25.

For Cray machines there is a separate library `libmap-sampler-binutils-2.25.a` provided for use with this updated linker.

The `make-profiler-libraries` script will automatically select the appropriate library to use based on the version of `ld` found in your `PATH`.

If you erroneously attempt to link `libmap-sampler-binutils-2.25.a` with your program using a version of `ld` prior to 2.25 you will get errors such as:

```
/usr/bin/ld.x: libmap-sampler.a(dl.o): invalid relocation type 42
```

If this happens check that the correct version of `ld` is in your `PATH` and rerun `make-profiler-libraries --lib-type=static`.

MAP adds unexpected overhead to my program

MAP's sampler library will add a little overhead to the execution of your program. Usually this is less than 5% of the wall clock execution time.

Under some circumstances, however, the overhead may exceed this, especially for short runs. This is particularly likely if your program has high OpenMP overhead, for example, if it is greater than 40%.

In this case the measurements reported by MAP will be affected by this overhead and therefore less reliable. Increasing the run time of your program for example, by changing the size of the input, decreases the overall overhead, although the initial few minutes still incurs the higher overhead.

At high per-process thread counts, for example on the Intel Xeon Phi (Knight's Landing), MAP's sampler library may incur a more significant overhead.

By default, when MAP detects a large number of threads it will automatically reduce the sampling interval in order to limit the performance impact.

Sampling behavior can be modified by setting the `ALLINEA_SAMPLER_INTERVAL` and `ALLINEA_SAMPLER_INTERVAL_PER_THREAD` environment variables. For more information on the use of these environment variables, see 16.11.

MAP takes an extremely long time to gather and analyze my OpenBLAS-linked application

OpenBLAS versions 0.2.8 and earlier incorrectly stripped symbols from the `.symtab` section of the library, causing binary analysis tools such as MAP and objdump to see invalid function lengths and addresses.

This causes MAP to take an extremely long time disassembling and analyzing apparently overlapping functions containing millions of instructions.

A fix for this was accepted into the OpenBLAS codebase on October 8th 2013 and versions 0.2.9 and above should not be affected.

To work around this problem without updating OpenBLAS, simply run `strip libopenblas*.so`, this removes the incomplete `.symtab` section without affecting the operation or linkage of the library.

MAP over-reports MPI, Input/Output, accelerator or synchronization time

MAP employs a heuristic to determine which function calls should be considered as MPI operations.

If your code defines any function that starts with `MPI_` (case insensitive) those functions will be treated as part of the MPI library resulting in the time spent in MPI calls to be over-reported by the activity graphs and the internals of those functions to be omitted from the *Parallel Stack View*.

Starting your functions names with the prefix `MPI_` should be avoided and is in fact explicitly forbidden by the MPI specification. This is described on page 19 sections 2.6.2 and 2.6.3 of the MPI 3 specification document http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf#page=49:

> All MPI names have an MPI_ prefix, and all characters are capitals. Programs must not declare names, for example, for variables, subroutines, functions, parameters, derived types, abstract interfaces, or modules, beginning with the prefix MPI_.

Similarly MAP categorizes I/O functions and accelerator functions by name.

Other prefixes to avoid starting your function names with include `PMPI_`, `_PMI_`, `OMPI_`, `omp_`, `GOMP_`, `shmem_`, `cuda_`, `__cuda`, `cu[A-Z][a-z]` and `allinea_`.

All of these prefixes are case-insensitive.

Also avoid naming a function `start_pes` or any name also used by a standard I/O or synchronization function, `write`, `open`, `pthread_join`, `sem_wait` and so on.

MAP collects very deep stack traces with boost::coroutine

A known bug in Boost (https://svn.boost.org/trac/boost/ticket/12400) prevents MAP from unwinding the call stack correctly.

This can be worked around by applying the patch attached to the bug report to your boost installation, or by specifying a manual stack allocator that correctly initializes the stack frame.

First add the following custom stack allocator:


```
#include <boost/coroutine/coroutine.hpp>
#include <boost/coroutine/stack_context.hpp>

struct custom_stack_allocator {
    void allocate(
            boost::coroutines::stack_context & ctx,
            std::size_t size ) {

        void * limit = std::malloc( size);
        if ( ! limit)
            throw std::bad_alloc();

        //Fix. RBP in the 1st frame of the stack will contain 0
        const int fill=0;

        std::size_t stack_hdr_size=0x100;
        if (size<stack_hdr_size)
            stack_hdr_size=size;
        memset(static_cast< char * >(limit)+size-stack_hdr_size,
            fill,
            stack_hdr_size);

        ctx.size = size;
        ctx.sp = static_cast< char * >( limit) + ctx.size;
    }

    void deallocate( boost::coroutines::stack_context & ctx) {
        void * limit = static_cast< char * >( ctx.sp) - ctx.size;
        std::free( limit);
    }
};
```

Then modify your program to use the custom allocator whenever a coroutine is created:

```
boost::coroutines::coroutine<int()>  my_coroutine(<func>,
        boost::coroutines::attributes(),custom_stack_allocator());
```

For more information, see the `boost::coroutine` documentation on stack allocators for your version of Boost.

Obtaining support

If this guide has not helped you, then the next step is to contact Arm support with a detailed report.

If possible, obtain a log file for the problem.To generate a log file, either check the *Help* → *Logging* → *Automatic* menu option or start Forge with the `--debug` and `--log` arguments:

```
$ ddt --debug --log=<logfilename>
$ map --debug --log=<logfilename>
```

Where `<logfilename>` is the name of the log file to generate.

Next, reproduce the problem using as few processors and commands as possible. Once finished, close the program as usual.

On some systems this file may be quite large. If so, please compress it using a program such as `gzip` or `bzip2` before sending it to support.

If your problem can only be replicated on large process counts, then please do not use the *Help* → *Logging* → *Debug* menu item or `--debug` argument as this will generate very large log files. Instead use the *Help* → *Logging* → *Standard* menu option or just the `--log` argument.

If you are connecting to a remote system, then the log file is generated on the remote host and copied back to the client when the connection is closed. The copy will not happen if the target application crashes or the network connection is lost.

In these cases, the remote copy of the log file can be found in the `tmp` subdirectory of the Arm configuration directory for the remote user account. The directory is `~/.allinea`, unless overridden by the `ALLINEA_CONFIG_DIR` environment variable.

Sometimes it may be helpful to illustrate your problem with a screenshot of Arm Forge's main window. To take a screenshot, choose the *Take Screenshot…* option under the *Window* menu. You will be prompted for a file name to save the screenshot to.

Queue template script syntax

Queue template tags

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 Arm Forge submits your job is given for each.

Note: It is often sufficient to simply use `AUTO_LAUNCH_TAG`*. See section A.3.1 The template script for an example.*

Tag	Description	After Submission Example
`AUTO_LAUNCH_TAG`	This tag expands to the entire replacement for your '`mpirun`' command line.	`ddt-mpirun -np 4 myexample.bin`
`DDTPATH_TAG`	The path to the Arm Forge installation	`/opt/arm/forge`
`WORKING_DIRECTORY_TAG`	The working directory Arm Forge was launched in	`/users/ned`
`NUM_PROCS_TAG`	Total number of processes	`16`
`NUM_PROCS_PLUS_ONE_TAG`	Total number of processes + 1	`17`
`NUM_NODES_TAG`	Number of compute nodes	`8`
`NUM_NODES_PLUS_ONE_TAG`	Number of compute nodes + 1	`9`
`PROCS_PER_NODE_TAG`	Processes per node	`2`
`PROCS_PER_NODE_PLUS_ONE_TAG`	Processes per node + 1	`3`
`NUM_THREADS_TAG`	Number of OpenMP threads per node (empty if OpenMP if "off")	`4`
`OMP_NUM_THREADS_TAG`	Number of OpenMP threads per node (empty if OpenMP is "off")	`4`
`MPIRUN_TAG`	mpirun binary (can vary with MPI implementation)	`/usr/bin/mpirun`
`AUTO_MPI_ARGUMENTS_TAG`	Required command line flags for mpirun (can vary with MPI implementation)	`-np 4`
`EXTRA_MPI_ARGUMENTS_TAG`	Additional mpirun arguments specified in the Run window	`-partition DEBUG`
`PROGRAM_TAG`	Target path and filename	`/users/ned/a.out`
`PROGRAM_ARGUMENTS_TAG`	Arguments to target program	`-myarg myval`
`INPUT_FILE_TAG`	The stdin file specified in the Run window	`/users/ned/input.dat`

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

Defining new tags

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

Tag definitions have the following format:

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

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

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

```
#PBS -option EXAMPLE_TAG
```

The following attributes are supported:

Attribute	Purpose	Example
`type`	`text`: General text input. `select`: Select from two or more options. `check`: Boolean. `file`: File name. `number`: Real number. `integer`: Integer number.	`type=text`
`label`	The label for the user interface widget.	`label="Account"`
`default`	Default value for this tag	`default="interactive"`
`text` type		
`mask`	Input mask: `0`: ASCII digit permitted but not required. `9`: ASCII digit required. 0–9. `N`: ASCII alphanumeric character required. A–Z, a–z, 0–9. `n`: ASCII alphanumeric character permitted but not required.	`mask="09:09:09"`
`options` type		
`options`	Options to use, separated by the | character	`options="not_-shared|shared"`
`check` type		
`checked`	Value of a `check` tag if checked.	`checked="enabled"`
`unchecked`	Value of a `check` tag if unchecked.	`unchecked="enabled"`
`integer` and `number` types		
`min`	Minimum value.	`min="0"`
`max`	Maximum value.	`max="100"`
`step`	Amount to step by when the up or down arrows are clicked.	`step="1"`
`decimals`	Number of decimal places.	`decimals="2"`
`suffix`	Display only suffix (will not be included in tag value).	`suffix="s"`
`prefix`	Display only prefix (will not be included in tag value).	`prefix="$"`
`file` type		

`mode`	`open-file`: an existing file. `save-file`: a new or existing file. `existing-directory`: an existing directory. `open-files`: one or more existing files, separated by spaces.	`mode="open-file"`
`caption`	Window caption for file chooser.	`caption="Select File"`
`dir`	Initial directory for file chooser.	`dir="/work/output"`
`filter`	Restrict the files displayed in the file chooser to a certain file pattern.	`filter="Text files (*.txt)"`

Examples

```
# JOB_TYPE_TAG: {type=select,options=parallel| \
   serial,label="Job␣Type",default=parallel}

# WALL_CLOCK_LIMIT_TAG: {type=text,label="Wall Clock Limit", \
   default="00:30:00",mask="09:09:09"}

# NODE_USAGE_TAG: {type=select,options=not_shared| \
   shared,label="Node␣Usage",default=not_shared}

# ACCOUNT_TAG: {type=text,label="Account",global}
```

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

To specify values for these tags click the *Edit Template Variables* button on the *Job Submission Options* page (see Figure 121 *Queuing Systems* shown previously) or the *Run* window. You will see a window similar to the one below:

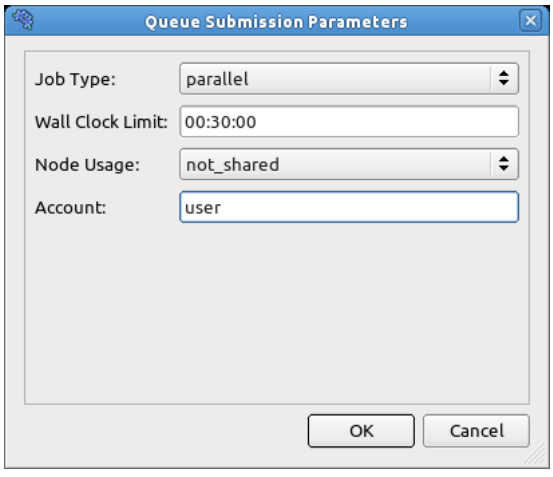

Figure 122: *Queue Parameters Window*

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

Specifying default options

A queue template file may specify defaults for the options on the Job Submission page so that when a user selects the template file these options are automatically filled in.

Name	Job Submission Setting	Example
`submit`	Submit command *Note: the command may include tags*.	`qsub -n NUM_NODES_TAG -t WALL_CLOCK_LIMIT_TAG --mode script -A PROJECT_TAG`
`display`	Display command *The output from this command is shown while waiting for a job to start*.	qstat
`job regexp`	Job regexp	(\d+)
`cancel`	Cancel command	qdel JOB_ID_TAG
`submit scalar`	Also submit scalar jobs through the queue	yes
`show num_procs`	Number of processes: Specify in Run window	yes
`show num_nodes`	Number of nodes: Specify in Run Window	yes
`show procs_per_node`	Processes per node: Specify in Run window	yes
`procs_per_node`	Processes per node: Fixed	16

Example

```
# submit: qsub -n NUM_NODES_TAG -t WALL_CLOCK_LIMIT_TAG \
   --mode script -A PROJECT_TAG
# display: qstat
# job regexp: (\d+)
# cancel: qdel JOB_ID_TAG
```

Launching

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

In a template file, this needs to be modified to run a command that will also launch the Arm Forge backend agents.

Some methods to do this are mentioned in this section.

Using AUTO_LAUNCH_TAG

This is the easiest method, and caters for the majority of cases. Simply replace your `mpirun` command line with `AUTO_LAUNCH_TAG`. Arm Forge will replace this with a command appropriate for your configuration (one command on a single line).

For example an `mpirun` line that looks like this:


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

Becomes:

```
AUTO_LAUNCH_TAG
```

`AUTO_LAUNCH_TAG` is roughly equivalent to:

```
DDT_MPIRUN_TAG DDT_DEBUGGER_ARGUMENTS_TAG \
MPI_ARGUMENTS_TAG PROGRAM_TAG ARGS_TAG
```

A typical expansion is:

```
/opt/arm/forge/bin/ddt-mpirun --ddthost login1,192.168.0.191 \
--ddtport 4242 --ddtsession 1 \
--ddtsessionfile /home/user/.allinea/session/login1-1 \
--ddtshareddirectory /home/user --np 64 \
--npernode 4 myprogram arg1 arg2 arg3
```

Using ddt-mpirun

If you need more control than is available using `AUTO_LAUNCH_TAG`, Arm Forge also provides a drop-in `mpirun` replacement that can be used to launch your job.

Note: this is only suitable for use in a queue template file when Arm Forge is submitting to the queue itself.

You should replace `mpirun` with `DDTPATH_TAG/bin/ddt-mpirun`.

For example, if your script currently has the line:

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

Then (for illustration only) the equivalent that Arm Forge needs to use would be:

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

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

```
DDTPATH_TAG/bin/ddt-mpirun -np NUM_PROCS_TAG \
EXTRA_MPI_ARGUMENTS_TAG DDTPATH_TAG/bin/ddt-debugger \
DDT_DEBUGGER_ARGUMENTS_TAG PROGRAM_TAG PROGRAM_ARGUMENTS_TAG
```

See I.1 Queue template tags for more information on template tags.

MPICH 1 based MPI

If `AUTO_LAUNCH_TAG` or `ddt-mpirun` are not suitable, you can also use the following method for MPICH 1 based MPIs.

If your `mpirun` command line looks like:

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

You need to export the `TOTALVIEW` environment variable, and add the `-tv` parameter to `mpirun`.

For example:


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

Scalar programs

If AUTO_LAUNCH_TAG is not suitable, you can also use the following method to launch scalar jobs with your template script:

```
DDTPATH_TAG/bin/ddt-client DDT_DEBUGGER_ARGUMENTS_TAG \
PROGRAM_TAG PROGRAM_ARGUMENTS_TAG
```

Using PROCS_PER_NODE_TAG

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.

The software 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 the queue template system.

Job ID regular expression

The *Regexp for job id* regular expression is matched on the output from your submit command. The first bracketed expression in the regular expression is used as the job ID. 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	White space
\S	Non-white space
\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 possible regular expression for retrieving 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_removeJOB_ID_TAG as the cancel command.

Arm IPMI Energy Agent

The Arm IPMI Energy Agent allows Arm MAP and Arm Performance Reports to measure the total energy consumed by the compute nodes in a job.

Note: Measuring energy with IPMI Energy agent requires Arm Forge Professional.

The IPMI Energy Agent is available to download from our website: IPMI Energy Agent.

Requirements

- The compute nodes must support IPMI.
- The compute nodes must have an IPMI exposed power sensor.
- The compute nodes must have an OpenIPMI compatible kernel module installed, such as `ipmi_devintf`.
- The compute nodes must have the corresponding device node in `/dev`, for example `/dev/ipmi0`.
- The compute nodes must run a supported operating system.
- The IPMI Energy Agent must be run as root.

To list the names of possible IPMI power sensors on a compute node use the following command:

```
ipmitool sdr | grep 'Watts'
```


Index

