Linaro Forge

Release 25.1

Linaro Limited

Oct 31, 2025

CONTENTS

CHAPTER

ONE

LINARO FORGE

1.1 Introduction to Linaro Forge

This section provides an overview of Linaro Forge.

Linaro Forge combines Linaro DDT for parallel high-performance application debugging, Linaro MAP for performance profiling and optimization advice, and Linaro Performance Reports for summarizing and characterizing both scalar and MPI application performance.

Linaro Forge supports many parallel architectures and models, including MPI, GPUs and OpenMP. Linaro Forge is a cross-platform tool, with support for the latest compilers and C++ standards, and Intel, 64-bit Arm, AMD, OpenPOWER, NVIDIA GPU, AMD GPU and Intel Xe-HPC GPU hardware.

Linaro 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 Linaro DDT and Linaro MAP during code development.

Linaro 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.

1.1.1 Linaro DDT

Linaro DDT is a powerful graphical debugger suitable for many different development environments.

Linaro DDT includes:

- 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.

Linaro 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.

Linaro DDT supports:

- C, C++, and all derivatives of Fortran, including Fortran 90.
- Limited support for Python. See *Reference table*

  For more information, see *Python debugging*.

- Parallel languages/models including MPI, UPC, and Fortran 2008 Co-arrays.
- GPU languages such as OpenMP Accelerators, CUDA, CUDA Fortran, HIP and DPC++/SYCL.

1.1.1.1 Related information

- *Get started with DDT*.

1.1.2 Linaro MAP

Linaro MAP is a parallel profiler that shows you the longest running lines of code, and explains why. Linaro MAP does not require any complicated configuration, and you do not need to have experience with profiling tools to use it.

Linaro 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.

1.1.2.1 Related information

- *Get started with MAP*.

1.1.3 Linaro Performance Reports

Linaro Performance Reports is a low-overhead tool that produces one-page text and HTML reports summarizing and characterizing both scalar and MPI application performance.

Linaro Performance Reports provides the most effective way to characterize and understand the performance of HPC application runs.

One single page HTML report answers a range of vital questions for any HPC site:

- Is this application optimized for the system it is running on?
- Does it benefit from running at this scale?
- Are there I/O or networking bottlenecks affecting performance?
- Which hardware, software or configuration changes can be made to improve performance further?

Linaro Performance Reports is based on the Linaro MAP low-overhead adaptive sampling technology that keeps data volumes collected and application overhead low:

- Runs transparently on optimized production-ready codes by adding a single command to your scripts.
- Just 5% application slowdown even with thousands of MPI processes.

1.1.3.1 Related Information

- *Get started with Performance Reports*
- *Interpret performance reports*

1.1.4 Online resources

Resources to support you using Linaro Forge.

- Get Linaro Forge at Linaro Forge Downloads.
- Known issues
- If you require more support contact Forge Support.

1.2 Installing Linaro Forge

This section describes how to install Linaro Forge on Linux, Windows, and Mac operating systems.

To learn how to install and manage licenses for Linaro Forge tools, and environment variables for managing product behavior, see *Licensing*.

1.2.1 Linux graphical install

Install remotely using the Linaro Forge Linux graphical installer.

1.2.1.1 Procedure

1. Download the installation package from Linaro Forge Downloads
2. Untar the installation package and run the installer executable with these commands:

```
tar xf linaro-forge-<version>-linux-<arch>.tar
```

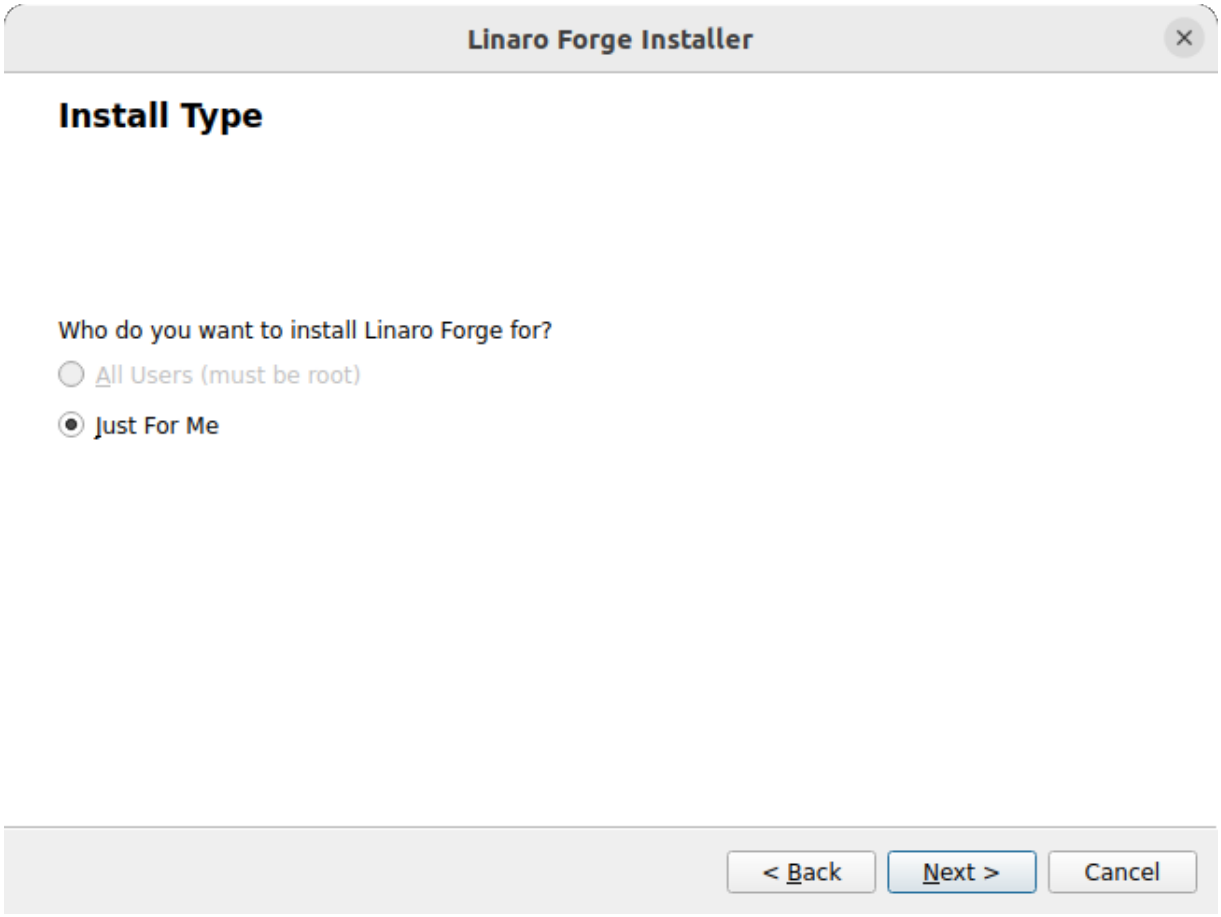
```
cd linaro-forge-<version>-linux-<arch>
```

```
./installer
```

Note: Replace `<version>` with the four digit version number of your installation package, using this format `xx.x.x`. Replace `<arch>` with the required architecture (AArch64, x86_64). For example: `linaro-forge-xx.x.x-linux-aarch64.tar`.

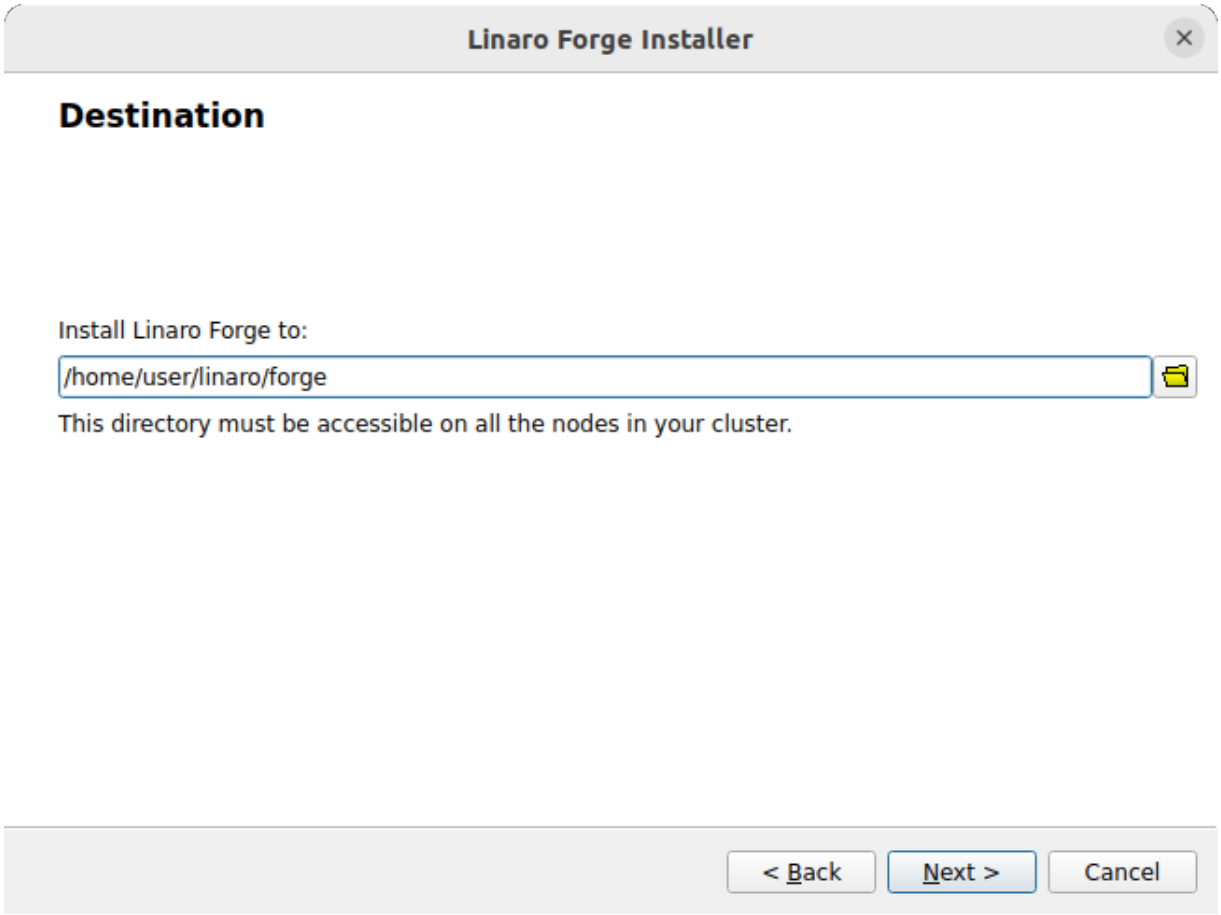
3. On the *Install Type* page, specify who can use Linaro Forge on this system.

Note: Only an administrator (root) can install Linaro Forge for *All Users* in a common directory, such as `/opt` or `/usr/local`. Otherwise, the *Just For Me* option is selected by default, and Linaro Forge is installed in the local directory.



4. On the installer *Destination* page, enter the directory where you want to install Linaro Forge.

Note: For a cluster installation, choose a directory that is shared between the cluster login or frontend node, and the compute nodes. Alternatively, install or copy into the same location on each node.



5. Installation progress displays on the *Install* page.

Note: The default Forge installation path is `/opt/linaro/forge/<version>/bin` when installed by root or `$HOME/linaro/forge/<version>/bin` otherwise. Please add the Linaro Forge installation path to the `PATH` environment variable to make the Linaro Forge tools available to the console:

```
export PATH=<Forge installation path>/bin:$PATH
```

1.2.1.2 Results

The installation adds icons for Linaro Forge to the **Development** menu in your desktop environment. When installation is complete, read the instructions in the `RELEASE-NOTES` file in the install package, for details about how to run Linaro DDT, Linaro MAP, and Linaro Performance Reports.

1.2.1.3 Next steps

Time-limited evaluation licenses are available from the Linaro website: free trial licence.

Linaro Forge supports a large number of different site configurations and MPI distributions. You must ensure that you fully integrate all components into your environment. For example, ensure that you propagate environment variables to remote nodes, and that tool libraries and executables are available on the remote nodes.

1.2.1.4 Related information

- Alternatively, use the *Linux text-mode install* to perform the installation.
- *Licensing*
- *Get started with DDT*
- *Get started with MAP*
- *Get started with Performance Reports*

1.2.2 Linux text-mode install

Install remotely using the `textinstall.sh` text-mode install script.

1.2.2.1 Procedure

1. Download the required package from the Linaro Forge Downloads webpage.
2. Untar the installation package and run the `textinstall.sh` script by using these commands:

```
tar xf linaro-forge-<version>-linux-<arch>.tar
```

```
cd linaro-forge-<version>-linux-<arch>
```

```
./textinstall.sh
```

> **Note:** Replace `<version>` with the three or four digit version number of your installation package (for major releases: `xx.x`, or support releases: `xx.x.x`). Replace `<arch>` with the required architecture (`aarch64`, `x86_64`).

3. When you are prompted, press **Return** to read the license, and enter the path of the installation directory. The directory must be accessible on all the nodes in your cluster.

Note: The default Forge installation path is `/opt/linaro/forge/<version>/bin` when installed by root or `$HOME/linaro/forge/<version>/bin` otherwise. Please add the Linaro Forge installation path to the `PATH` environment variable to make the Linaro Forge tools available to the console:

```
export PATH=<Forge installation path>/bin:$PATH
```

1.2.2.2 Next steps

For details about how to run Linaro DDT, Linaro MAP, and Linaro Performance Reports, see the `RELEASE-NOTES` file in the install package when the installation is complete.

Time-limited evaluation licenses are available from the Linaro website: free trial licence.

Linaro Forge supports a large number of different site configurations and MPI distributions, and therefore you must ensure that you fully integrate all components into your environment. For example, propagate environment variables to remote nodes, and ensure that the tool libraries and executables are available on them.

1.2.2.3 Related information

- Alternatively, use the *Linux graphical install* to perform the installation.
- *Licensing*
- *Get started with DDT*
- *Get started with MAP*
- *Get started with Performance Reports*

1.2.3 Mac remote client installation

The Mac OS X Linaro Forge installation package is a remote client only for connecting to an Linaro Forge installation. The Linaro Forge remote client is supplied as an Apple Disk Image (`.dmg`) file.

1.2.3.1 About this task

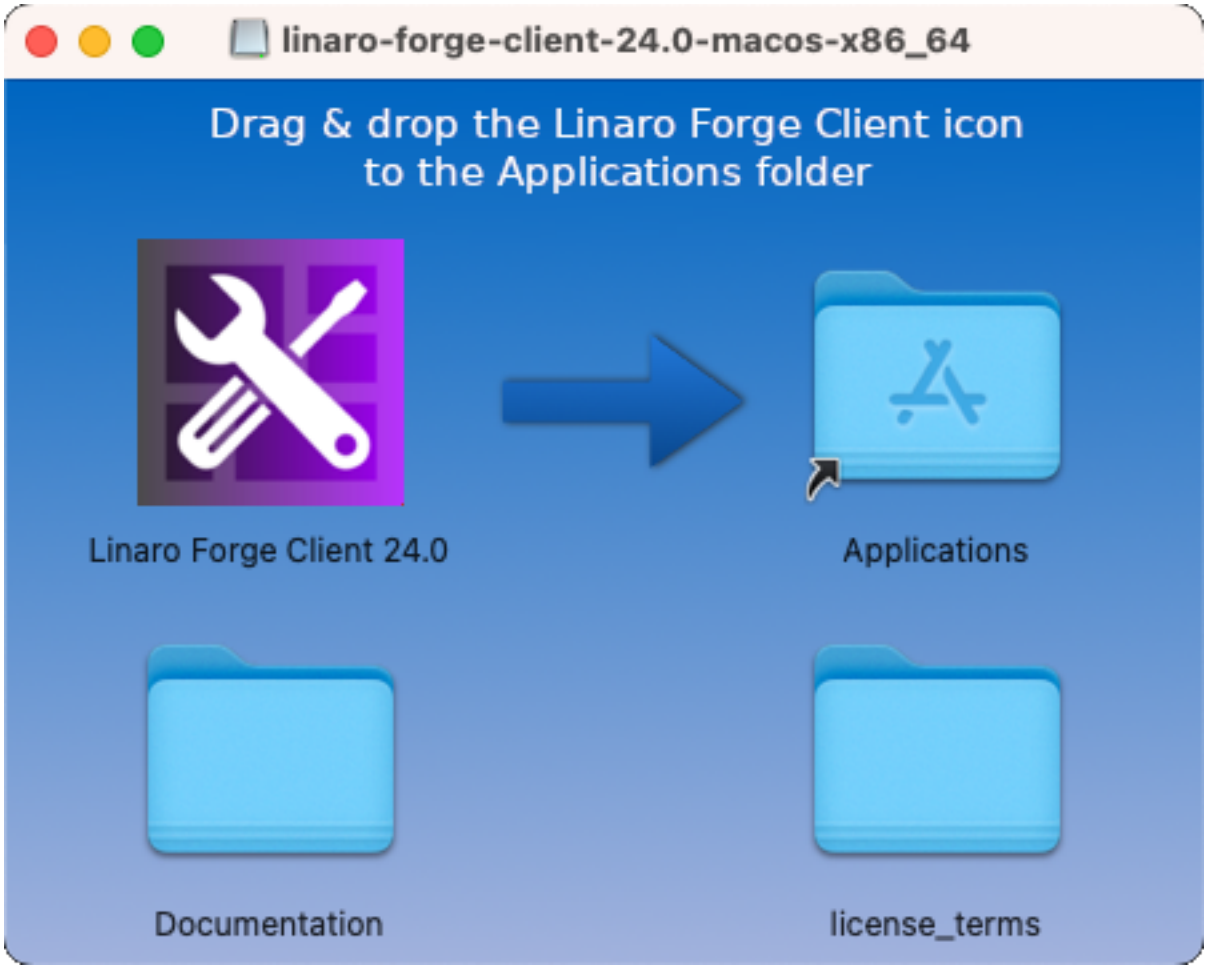
You do not need to install a license file on a machine running the remote client when you connect remotely to Linaro Forge.

1.2.3.2 Procedure

1. Download the required package from Linaro Forge Downloads.

   The `.dmg` file includes the `Documentation` folder and the client application bundle icon. The `Documentation` folder contains a copy of this user guide and the release notes.

2. Drag and drop the client application bundle icon into the `Applications` directory.



1.2.3.3 Related information

- *Licensing*
- *Get started with DDT*
- *Get started with MAP*
- *Get started with Performance Reports*

1.2.4 Windows remote client installation

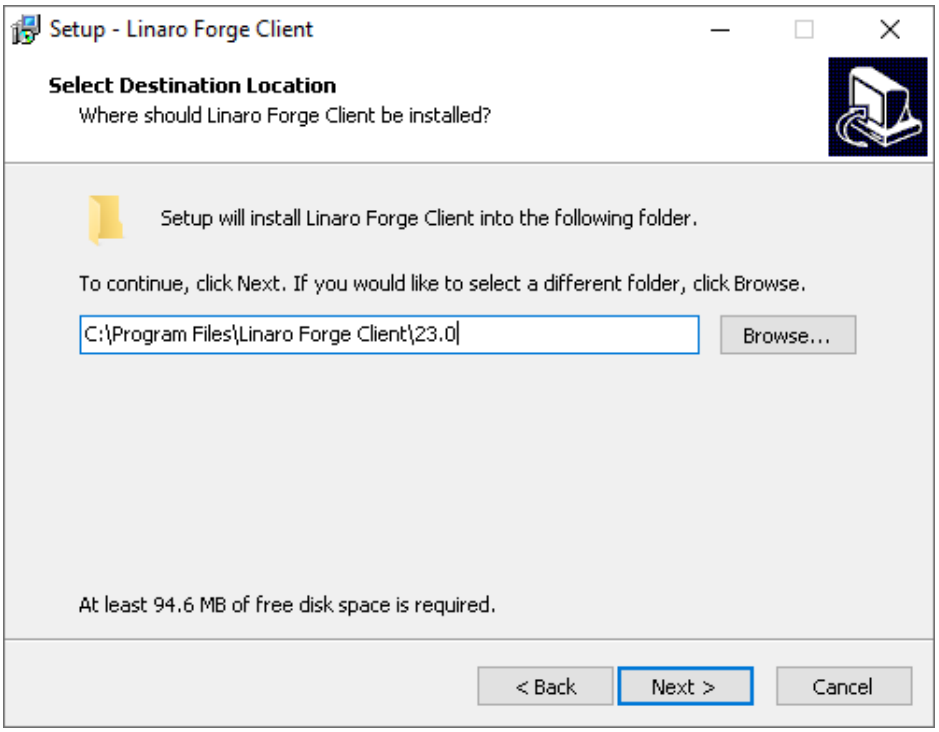
The Windows installation package is a remote client only for connecting to an Linaro Forge installation. The Linaro Forge remote client is supplied as a Windows executable (`.exe`) file.

1.2.4.1 About this task

You do not need to install a license file on a machine running Remote Client for connecting remotely to Linaro Forge.

1.2.4.2 Procedure

1. Download the Remote Client for Windows installation package from the Linaro Forge Downloads webpage.
2. Run the Windows file executable to install the Linaro Forge remote client.
3. On the installer **Destination** page, enter the directory where you want to install Linaro Forge.



Note: If your user account has administrative privileges, then by default Linaro Forge is installed under `C:\Program Files`. If your account does not have administrative privileges, then by default Linaro Forge is installed under `C:\Users\%USERNAME%\AppData\Local`.

1.2.4.3 Related information

- *Licensing*
- *Get started with DDT*
- *Get started with MAP*
- *Get started with Performance Reports*

1.3 Licensing

This section describes how to install and manage licenses for Linaro Forge.

Before you begin:

- You must install a license file on a machine running Linaro Forge tools.
- You do not need to install a license file on a machine running Linaro Forge Remote Client for connecting remotely to Linaro Forge tools on a remote system.

Note: If you do not have a valid license file, the user interface shows an alert in the lower-left corner, and you can not run, debug, or profile new programs.

Time-limited evaluation licenses are available from the Linaro website: free trial licence.

1.3.1 Workstation and evaluation licenses

Linaro Forge supports separate license files for Linaro DDT, Linaro MAP, and Linaro Performance Reports with a single installation of Linaro Forge.

1.3.1.1 About this task

This task shows the steps for installing workstation and evaluation licenses directly on the installation machine.

Workstation and evaluation license files for Linaro Forge do not require Linaro Forge Licence Server.

If there are multiple licenses installed for the same product, Linaro Forge uses the license with the most tokens.

Warning: Do not edit license files because this prevents them from working.

1.3.1.2 Procedure

1. Copy your license files to `{installation-directory}/licences`. For example, `/home/user/linaro/forge/<version>/licences/Licence.ddt`.

   Note: Linaro Forge accepts directories spelled `licence` or `license`. For example, both of these paths are valid license file locations: `/opt/linaro/licenses` and `/opt/linaro/licences`.

2. When Linaro Forge starts, select Linaro DDT or Linaro MAP on the **Welcome** page.

   If you would prefer to store the license files in an alternative location, use the environment variable `FORGE_LICENSE_DIR` to specify it. For example:

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

   Note: You can use `FORGE_LICENSE_DIR`, `FORGE_LICENCE_DIR`, `ALLINEA_LICENSE_DIR`, and `ALLINEA_LICENCE_DIR` interchangeably.

1.3.1.3 Related information

- *Supercomputing and other floating licenses*
- Time-limited evaluation licenses are available from the Linaro website: free trial licence.
- Contact Forge Support if you have any issues with your license.

1.3.2 Supercomputing and other floating licenses

If you are using floating licenses for an HPC cluster, you need to use Linaro Forge Licence Server.

1.3.2.1 Before you begin

- Download the required package from the Linaro Forge Downloads page.

1.3.2.2 About this task

A floating license consists of two files:

- Server license (`Licence.xxxx`)
- Client license (`Licence`)

1.3.2.3 Procedure

1. Copy the client file (`Licence`) to `<installation-directory>`. For example:

```
/home/user/linaro/forge/<version>/licences/Licence
```

2. Edit the `hostname` line to contain the host name or IP address of the machine running the Linaro Forge Licence Server.
3. See the Linaro Forge Licence Server user guide on the Linaro Forge Licence Server web page for instructions on how to install the server license.
4. Ensure that Linaro Forge Licence Server is running on the designated license server machine before you run Linaro Forge.

1.3.2.4 Related information

- More information about Linaro Forge Licence Server on the Linaro Forge Licence Server web page.
- *Workstation and evaluation licenses*.
- Time-limited evaluation licenses are available from the Linaro website: free trial licence.
- Contact Forge Support if you have any licensing issues.

1.3.3 Architecture licensing

This section describes the steps for setting up licenses for different architectures.

1.3.3.1 About this task

Licenses specify the compute node architectures with which they can be used.

The licenses issued in early legacy versions of Linaro Forge enabled the x86_64 architecture by default. If you are using other architectures, you are supplied with suitable licenses to enable your architecture.

If you are using multiple license files to specify multiple architectures, Linaro recommends that you follow these steps.

Contact Forge Support if you have any licensing issues.

1.3.3.2 Procedure

1. Ensure that the default licenses directory is empty.
2. Create a directory for each architecture.
3. When you want to target a specific architecture, set `FORGE_LICENSE_DIR` to the relevant directory. Alternatively, set `FORGE_LICENSE_FILE` to specify the license file.

1.3.3.3 Example

On a site that targets two architectures, x86_64 and AArch64, create a directory for each architecture, and name them `licenses_x86_64` and `licenses_aarch64`. Then, to target the architectures, set the license directories as follows:

To target AArch64:

```
export FORGE_LICENSE_DIR=/path/to/licenses/licenses_aarch64
```

To target x86_64:

```
export FORGE_LICENSE_DIR=/path/to/licenses/licenses_x86_64
```

1.4 Environment variables

This section provides a reference to Linaro Forge environment variables, with guidance on requirements and best practices for using them.

1.4.1 MAP and Performance Reports variables

This table provides a reference to environment variables you can use for Linaro MAP and Linaro Performance Reports, with guidance on requirements and best practices for using them.

FORGE_SAMPLER_INTERVAL

Takes a sample in each 20ms period, which gives a default sampling rate of 50Hz. The 50Hz sampling rate automatically decreases as the run proceeds to ensure a constant number of samples are taken.

For more information, see FORGE_SAMPLER_NUM_SAMPLES.

If your program runs for a very short period of time, you might benefit by decreasing the initial sampling interval. For example, FORGE_SAMPLER_INTERVAL=1 sets an initial sampling rate of 1000Hz, or once per millisecond. Higher sampling rates are not supported.

Linaro recommends that you avoid increasing the sampling frequency from the default if there are many threads or very deep stacks in the target program. Increasing the sampling frequency might not allow enough time to complete one sample before the next sample starts.

Note: Custom values for FORGE_SAMPLER_INTERVAL can be overwritten by values set from the combination of FORGE_SAMPLER_INTERVAL_PER_THREAD and the expected number of threads (from OMP_NUM_THREADS). For more information, see FORGE_SAMPLER_INTERVAL_PER_THREAD.

FORGE_SAMPLER_INTERVAL_PER_THREAD

To keep overhead low, this imposes a minimum sampling interval based on the number of threads. By default, this is 2 milliseconds per thread. For 11 or more threads, the initial sampling interval increases to more than 20ms.

To adjust the minimum per-thread sample time, set FORGE_SAMPLER_INTERVAL_PER_THREAD in milliseconds.

Linaro recommends that you avoid lowering this value from the default if there are many threads. Lowering this value might not allow enough time to complete one sample before the next sample starts.

Note:

- Whether OpenMP is enabled or disabled, the final script or scheduler values set for `OMP_NUM_THREADS` are used to calculate the sampling interval per thread (`FORGE_SAMPLER_INTERVAL_PER_THREAD`). When you configure your job for submission, check whether the final submission script, scheduler, or the Linaro MAP user interface has a default value for `OMP_NUM_THREADS`.
- Custom values for `FORGE_SAMPLER_INTERVAL` are overwritten by values set from the combination of `FORGE_SAMPLER_INTERVAL_PER_THREAD` and the expected number of threads (from `OMP_NUM_THREADS`).

FORGE_MPI_WRAPPER

To use a specific pregenerated MPI wrapper library, set `FORGE_MPI_WRAPPER=<path of shared object>`.

Linaro MAP and Linaro Performance Reports include several precompiled wrappers for common MPI implementations. If your MPI is supported, Linaro Forge will automatically select and use the appropriate wrapper. If no precompiled wrapper is found a MPI wrapper library will be silently compiled and used instead. `FORGE_MPI_WRAPPER` only needs to be used if neither the prepackaged MPI wrapper library or just-in-time compilation of a wrapper work.

To manually compile a wrapper specifically for your system, set `FORGE_WRAPPER_COMPILE=1` and `MPICC`, then run `<MAP-installation-directory>/map/wrapper/build_wrapper`. Running `build_wrapper` generates the wrapper library `~/.allinea/wrapper/libmap_sampler_pmpi_<hostname>.so` with symlinks.

Symlinks are generated for these 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
```

Warning: To prevent unauthorized users modifying sensitive files and directories, and introducing unsafe code into your environment, ensure that you assign only the minimum permissions that are required, and avoid group or world-writable permissions.

FORGE_WRAPPER_COMPILE

To instruct Linaro MAP and Linaro Performance Reports to fall back to create and compile a just-in-time wrapper, set `FORGE_WRAPPER_COMPILE=1`.

To generate a just-in-time wrapper, an appropriate compiler must be available on the machine where Linaro Forge is running, or on the remote host when using remote connect.

Linaro MAP and Linaro Performance Reports attempt to auto detect your MPI compiler. However, Linaro recommends that you set the `MPICC` environment variable to the path to the correct compiler.

FORGE_MPIRUN

The path of mpirun, mpiexec, or equivalent.

If set, `FORGE_MPIRUN` has higher priority than the path set in the user interface and the `mpirun` found in `PATH`.

FORGE_SAMPLER_NUM_SAMPLES

By default, Linaro MAP and Linaro Performance Reports collect 1000 samples per process. To avoid generating too much data on long runs, the sampling rate automatically decreases as the run progresses, to ensure that only 1000 evenly spaced samples are stored.

To adjust the sampling rate, set FORGE_SAMPLER_NUM_SAMPLES=<positive integer>.

Note: Linaro recommends that you leave this value at the default setting. Higher values are not generally beneficial, and add extra memory overheads while running your code. Consider that with 512 processes, the default setting already collects half a million samples over the job, and the effective sampling rate can be very high indeed.

FORGE_KEEP_OUTPUT_LINES

Specifies the number of lines of program output to record in .map files. Setting to 0 removes the line limit restriction.

However, Linaro recommends that you avoid setting FORGE_KEEP_OUTPUT_LINES=0 because it can result in very large .map files if the profiled program produces a lot of output.

To learn more, see *Restrict Output*.

FORGE_KEEP_OUTPUT_LINE_LENGTH

The maximum line length for program output that is recorded in .map files. Lines that contain more characters than the line length limit are truncated. Setting this to 0 removes the line length restriction.

However, Linaro recommends that you avoid setting FORGE_KEEP_OUTPUT_LINE_LENGTH=0 because it can result in very large .map files if the profiled program produces a lot of output per line.

To learn more, see *Restrict Output*.

FORGE_PRESERVE_WRAPPER

To gather data from MPI calls, Linaro MAP generates a wrapper to the chosen MPI implementation. See *Prepare a program for profiling*.

By default, the generated code and shared objects are deleted when Linaro MAP no longer needs them.

To prevent Linaro MAP and Linaro Performance Reports from deleting these files, set FORGE_PRESERVE_WRAPPER=1.

Note: If you use remote launch, this variable must be exported in the remote script. See *Remote connections dialog*.

FORGE_SAMPLER_NO_TIME_MPI_CALLS

Set this to prevent Linaro MAP and Linaro Performance Reports from timing the time spent in MPI calls.

FORGE_SAMPLER_TRY_USE_SMAPS

To allow Linaro MAP and Linaro Performance Reports to use /proc/[pid]/smaps to gather memory usage data, set FORGE_SAMPLER_TRY_USE_SMAPS.

Note: FORGE_SAMPLER_TRY_USE_SMAPS significantly slows down sampling.

MPICC

If a prepackaged MPI wrapper library is not available for your MPI Linaro MAP and Linaro Performance Reports will attempt to compile one as and when it is required. The MPICC variable will be the first MPI compiler used to attempt to compile the just-in-time MPI wrapper library.

If using MPICC fails, Linaro MAP and Linaro Performance Reports 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), set MPICC to the appropriate MPI compiler to use.

1.4.2 Performance Report customization variables

Environment variables to customize your reports.

FORGE_NOTES

Any text in this environment variable is also included in all reports.

1.4.3 Licensing variables

Environment variables to handle licensing.

FORGE_LICENSE_FILE

Location of the licenses directory.

The default is /path/to/forge/licences.

Note: You can set FORGE_LICENSE_FILE with the path to a specific license to ensure that Linaro Forge detects the license in that location.

However, Linaro Forge continues detecting other licenses that are available when FORGE_LICENSE_FILE is set, and there is no guarantee of Linaro Forge using the specified license.

To force Linaro Forge to use a specific license file, use the variable: FORGE_FORCE_LICENSE_FILE

FORGE_FORCE_LICENSE_FILE

Location of the license file. This ensures that Linaro Forge uses a specified license file.

FORGE_LICENSE_DIR

Colon separated list of license directories.

The default is /path/to/forge/licences.

Note: You can set this variable with the path to a specific directory to ensure that Linaro Forge detects the licenses in the directory.

However, Linaro Forge continues detecting licenses that are available in other locations, and there is no guarantee of Linaro Forge using the license at the specified paths.

To force Linaro Forge to use a specific license file, use the variable: FORGE_FORCE_LICENSE_FILE.

FORGE_MAC_INTERFACE

Specify the network interface name to which the license is tied.

1.4.4 Warning suppression variables

Environment variables for warning suppression. Use these when autodetection generates incorrect messages.

FORGE_NO_APPLICATION_PROBE

Do not attempt to auto-detect MPI or CUDA, ROCm or Intel Xe executables.

FORGE_DETECT_APRUN_VERSION

Automatically detect Cray MPT by passing `-version` to the `aprun` wrapper, and parsing the output.

1.4.5 I/O behavior variables

Environment variables for handling default I/O behavior.

FORGE_NEVER_FORWARD_STDIN

Do not forward the `stdin` of the `perf-report` command `stdin` to the program under analysis, even if you are running without the user interface. By default, Linaro Performance Reports only forwards `stdin` when running without the user interface.

FORGE_ENABLE_ALL_REPORTS_GENERATION

Enables the option in Linaro Performance Reports to generate all types of results at the same time, using the `all` extension.

1.4.6 Timeout variables

Environment variables for handling timeouts.

FORGE_NO_TIMEOUT

Do not time out if nodes do not connect after a specified length of time. This might be necessary if the MPI subsystem takes unusually long to start processes.

FORGE_PROCESS_TIMEOUT

Length of time (ms) to wait for a process to connect to the front end.

FORGE_MPI_FINALIZE_TIMEOUT_MS

Length of time (ms) to wait for `MPI_Finalize` to end, and the program to exit. The default is `300000` (5 minutes). `0` waits forever.

1.4.7 Troubleshooting variables

Environment variables for basic troubleshooting.

FORGE_DEBUG_HEURISTICS

To print the weights and heuristics used to autodetect which MPI is loaded, set to `1`.

1.5 Connecting to a remote system

This section describes the Linaro Forge Remote Client that allows you to debug and profile remote jobs, while running the user interface on your local machine.

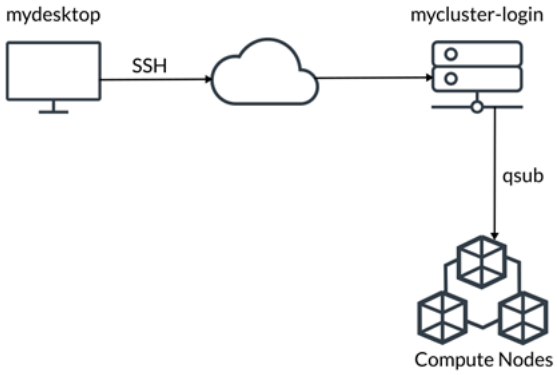
This is faster than remote-X11 (particularly for slow connections) and provides a native user interface.

1.5.1 Connecting remotely

The remote client is available for Windows, Mac OS X, and Linux, and can also be used as a local viewer for collected Linaro MAP profiles (`.map` files).

1.5.1.1 About this task

The Linaro Forge Remote Client connects and authenticates using SSH (typically a login node), and uses existing licensing from your remote resource (compute nodes).



1.5.1.2 Before you begin

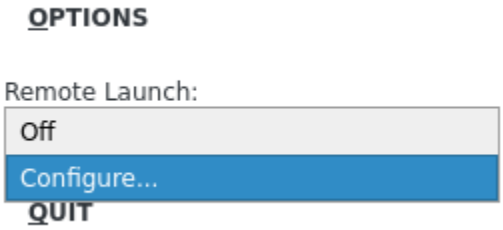
- For working remotely using Linaro DDT or Linaro MAP, ensure that the versions match between the locally installed Linaro Forge Remote client and the Linaro Forge tools installed on remote systems.
- You do not require a license file for the locally installed remote client. Linaro Forge uses the license of the remote system when it connects.
- Ensure that you can establish a working SSH connection to the remote server.

Note: On Windows if you normally use an OpenSSH key see *Convert OpenSSH private key on Windows with PuTTYgen* to obtain a version of your key in PuTTY's key format.

1.5.1.3 Procedure

1. Use SSH to login from your desktop machine **mydesktop** to the login node **mycluster-login**.
2. Start a job using the queue submission command `qsub`.
3. Connect Linaro Forge using `Reverse Connect`, typically to a batch compute node.

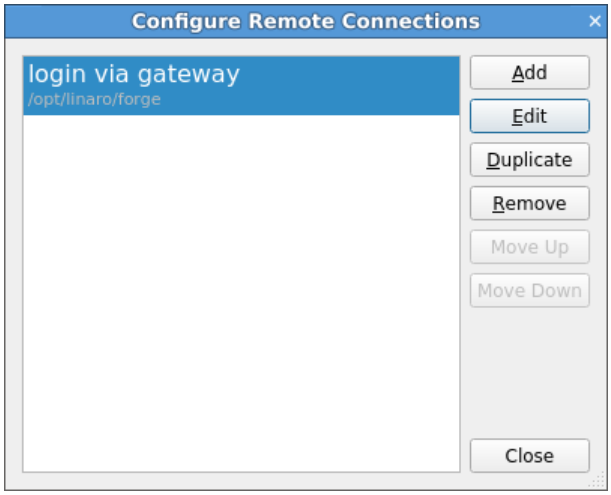
   See *Reverse Connect* for more information on Reverse Connect.
4. To connect to a remote system, click the *Remote Launch* drop-down list and select *Configure*.



The *Remote connections dialog* opens and allows you to edit the required settings.

1.5.2 Remote connections dialog

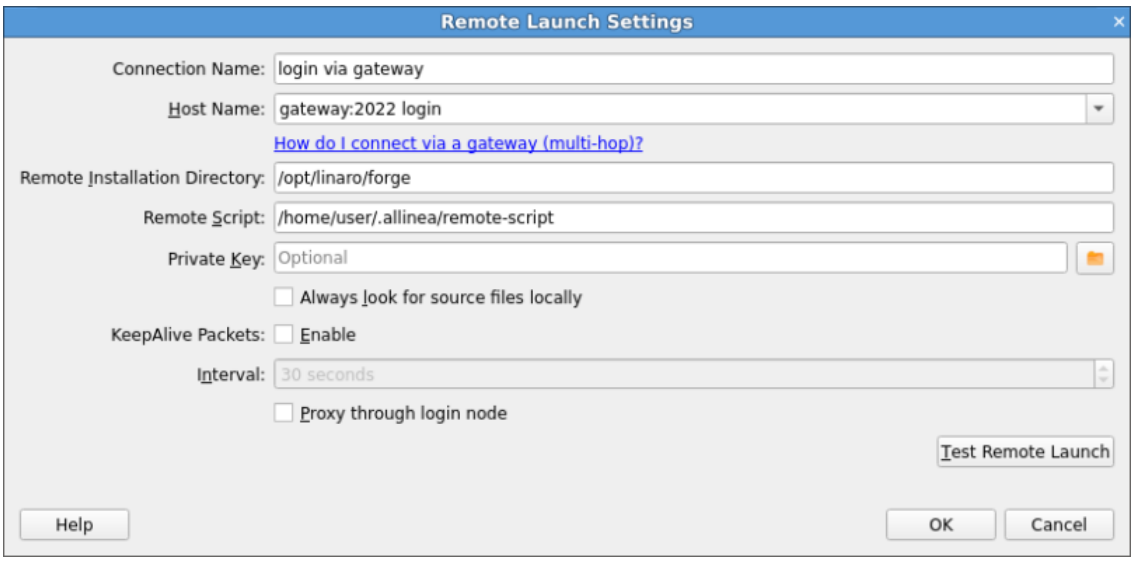
The **Remote Connections Dialog** allows you to add, remove and edit connections to remote systems.



To display the Remote launch settings, add or edit a host (or any duplicated hosts) in the list.

1.5.2.1 Remote launch settings

The **Remote launch settings** displays when you edit or add a remote host to the **Remote connections dialog**.



- *Connection Name*: Optional for this connection. If no name is specified, the *Host Name* is used.
- *Host Name* is the host name of the remote system that you are connecting to.

  The syntax of the host name field is:

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

  - `username` - Optional. The name that you use on the remote system. If this is not specified, your local user name is used instead.
  - `hostname` - The host name of the remote system.
  - `port` - Optional. The port number that the SSH daemon on the remote host listens on. If not specified, the default of 22 is used.

  To log in using one or more intermediate hosts (such as a gateway), enter the host names in order, separated by spaces. For example, `gateway.company.com cluster.lan`.

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

  Note: You can specify more SSH options in the `remote-exec` script covered in *Connecting to compute nodes and remote programs (remote-exec)*.

- *Remote Installation Directory* is the full path to the installation on the remote system.
- *Private Key*. This optional file on the local system is used as private key for the SSH connection. For the remote client on Linux and Mac OS X, OpenSSH private key files are supported. For the remote client on Windows, PuTTY private key files (`.ppk`) are supported. See *Convert OpenSSH private key on Windows with PuTTYgen* if you normally use a OpenSSH key on Windows.
- *Remote Script*. This optional script runs before starting the remote daemon on the remote system. You can use this script to load the required modules for Linaro Forge, and your MPI and compiler.

See the following sections for more details.

Note: The optional script is usually not necessary when using **Reverse Connect**.

- *Always look for source files locally*. Select this option to use the source files on the local system instead of the remote system.
- *KeepAlive Packets*. Select this option to enable KeepAlive packets. These are placeholder packets that are 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*. When using more than one intermediate host, select this to use the local SSH key for connecting to both the login and the batch nodes. This is equivalent to not setting `FORGE_NO_SSH_PROXYCOMMAND`.

  When this option is not set, Linaro Forge connects to the login node using your local SSH key, and then uses the key on the remote SSH configuration folder to connect to the batch node. This is the same as setting `FORGE_NO_SSH_PROXYCOMMAND=1`.

Note: If the above settings are insufficient you can access more advanced settings by configuring a session to your target host in PuTTY before connecting with Forge. See *Configure SSH on Windows with PuTTY*.

1.5.2.2 Remote script

The script can load modules using the `module` command, or set environment variables. Linaro Forge sources this script before running its remote daemon (your script does not need to start the remote daemon itself).

Run the script 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 can install a site-wide script that is sourced for all users at `/path/to/remote-init`.

You can also install a user-wide script that is sourced for all of your connections at `$FORGE_CONFIG_DIR/remote-init`.

Note: `FORGE_CONFIG_DIR` defaults to `$HOME/.allinea` if it is not set.

1.5.2.3 Example Script

Create this script file on the remote system, and ensure that the full path to the file is entered in the Remote Script field box.

```
module load forge
module load mympi
module load mycompiler
```

1.5.3 Reverse Connect

The Linaro Forge Reverse Connect feature allows you to submit your job from a shell terminal as you currently do, with a minor adjustment to your `mpirun` (or equivalent), to allow that job to connect back to the Linaro Forge user interface.

1.5.3.1 About this task

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

Note: Node-locked licenses such as workstation or Linaro Forge Cluster licenses do not include the Reverse Connect feature.

1.5.3.2 Procedure

1. Start Linaro Forge and connect to your remote system (typically a login node) with SSH.
2. To enable Reverse Connect, modify your current (or equivalent) command line inside your interactive queue allocation, or queue submission script.

   In most of the cases it is sufficient to prefix the script with `ddt/map --connect`. Almost all arguments beside `--offline` and `--profile` are supported by Reverse Connect.

1.5.3.3 Example

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

1. To debug the job using Reverse Connect and *Express Launch (DDT)*, run:

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

2. To profile the job using Reverse Connect and *Express Launch (MAP)*, run:

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

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

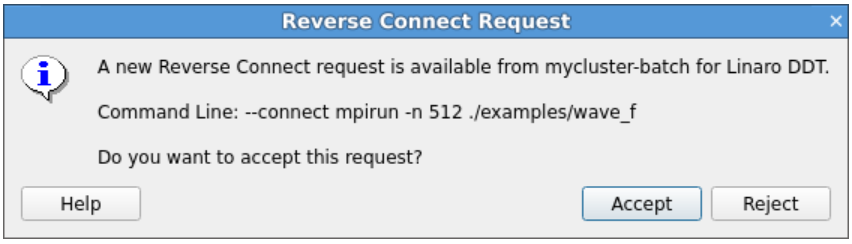
   Debug:

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

   Profile:

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

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



4. To accept the request, click *Accept*.

Note: Linaro Forge connects to the specified host and executes what you specified with the command line. If you do not want to accept the request, click *Reject*.

If a Reverse Connect is initiated, for example with `ddt --connect`, Linaro Forge starts a server listening on a port in the range between 4201 and 4240 on the remote system (typically a login node).

If this port range is not suitable for some reason, such as when ports are already taken by other services, you can override the port range with the environment variable `FORGE_REMOTED_PORTS`.

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

The server selects 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.

1.5.4 Treeserver or general debugging ports

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

1.5.4.1 Using a queue submission or 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.

1.5.4.2 Using reverse connect

- See **Connection details** in *Reverse Connect* 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.

1.5.5 Starting Linaro Forge

To start Linaro Forge, type one of the following commands into a terminal window:

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

To start Linaro Forge on Mac OS X, click the Linaro Forge icon or type this command in the terminal window:

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

To launch additional instances of the Linaro Forge application, right-click the Dock icon of a running instance of Linaro Forge, and select *Launch a new instance of Linaro Forge*.

Alternatively, you can use the following command in a terminal:

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

When Linaro Forge starts, the *Welcome Page* displays, in which you can select the tool you would like to use (Linaro DDT or Linaro MAP). Linaro Performance Reports can be run using the command line. Use the icons on the left-hand side to switch between Linaro MAP and Linaro DDT.

Select the tool you want to use, and click the buttons in the menu to select a debugging or profiling activity.

Note: To pipe input directly to the program, you must operate Linaro Forge in Express Launch mode. For information about how to send input to your program, see *Program input and output* or *Run MAP from the command line*.

Note: In Express Launch mode (see Linaro DDT or Linaro MAP), the Welcome Page does not display. Instead, Linaro Forge directly displays the *Run* dialog. If no valid license is found, the program exits and the appropriate message displays in the console output.

1.6 Command-line support

Linaro Forge provides a list of useful command options and is a convenient quick reference for the command syntax to use with Linaro Forge components. Use the `--help` command prefixed with the target product component name.

For example, `/path/to/forge/bin/forge --help` displays the syntax for the command to debug an MPI program, and lists the options and arguments that you can use to adjust the outcome:

```
Usage: ddt [OPTION...] [PROGRAM [PROGRAM_ARGS]]
    ddt [OPTION...] (mpirun|mpiexec|aprun|...) [MPI_ARGS] PROGRAM
 [PROGRAM_ARGS]
Start DDT and use it to select and start an MPI program for debugging.

--attach=[host1:]pid1,[host2:]pid2... [PROGRAM]
                  attach to PROGRAM being run by list of host:pid pairs
--attach-file=FILE [PROGRAM]
                  attach to PROGRAM being run by list of host:pid pairs
                  specified in FILE
...
```

Note: Linaro Forge and Linaro DDT command-line help lists are identical.

Linaro Forge
```
/path/to/forge/bin/forge --help
```

Linaro DDT
```
/path/to/forge/bin/ddt --help
```

Linaro MAP
```
/path/to/forge/bin/map --help
```

Linaro Performance Reports
```
/path/to/forge/bin/perf-reports --help
```

1.6.1 Related information

Online resources

CHAPTER

TWO

DDT

2.1 Get started with DDT

Learn how to get started using Linaro DDT.

2.1.1 Prepare a program for debugging

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

We recommend that you turn off compiler optimizations as they can produce unexpected results when debugging. 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 enables you to choose the 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.

2.1.2 Express Launch (DDT)

All of the Linaro Forge products can be launched by typing their 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.

Note: Please add the Linaro Forge installation path to the `PATH` environment variable to make the Linaro DDT available to the console:

```
export PATH=<Forge installation path>/bin:$PATH
```

The MPI implementations supported by Express Launch are:

- Bullx MPI
- Cray X-Series (MPI/SHMEM/CAF)
- Intel MPI
- MPICH 3
- MPICH 4
- Open MPI (MPI/SHMEM)
- Open MPI (Cray XT/XE/XK)
- Cray XT/XE/XK (UPC)
- SLURM (MPMD)
- Cray PALS

If your MPI is not supported by Express Launch, an error message will display:

```
$ 'Generic' MPI 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.

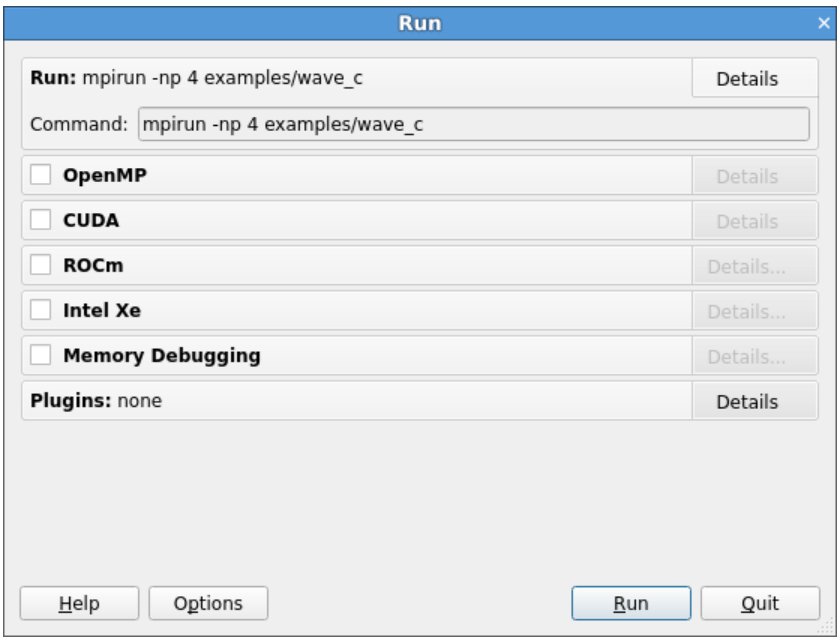
One advantage of Express Launch mode is that it is easy to modify existing queue submission scripts to run your program under one of the Linaro Forge products. This works best for Linaro DDT with Reverse Connect, `ddt --connect`, for interactive debugging or in offline mode (`ddt --offline`).

See *Reverse Connect* for more details.

If you cannot use Reverse Connect and want to use interactive debugging from a queue, you might need to configure Linaro DDT to generate job submission scripts for you. More details on this can be found in *Start a job in a queue* and in *Integration with queuing systems*.

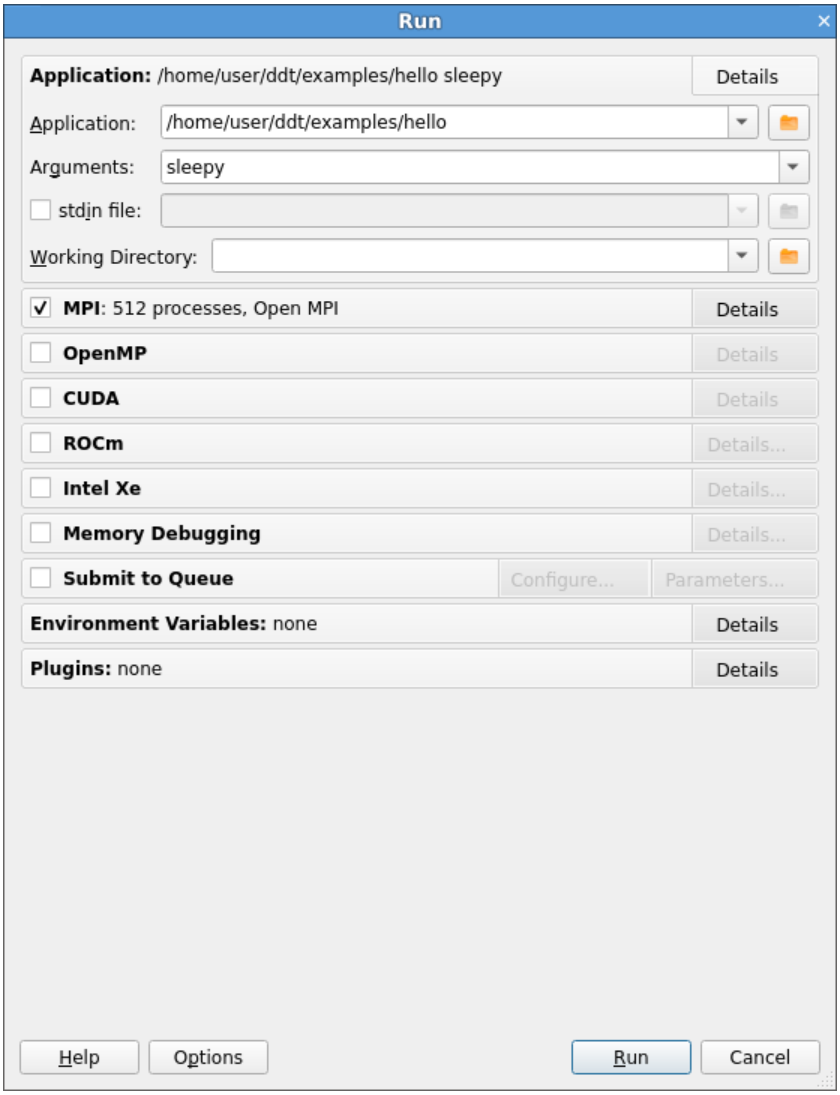
2.1.2.1 Run dialog box

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



2.1.3 Run a program

When you click *Run* on the Welcome page, the *Run* window displays.



The settings are grouped into sections. Click *Details* to expand a section.

2.1.3.1 Application

Application:
The full file path to your application. If you specified one on the command line, this is automatically filled in. You can browse and select your application.

Note: Many MPIs have problems working with directory and program names that contain spaces. We recommend that you do not use 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 Linaro

DDT and your command shell. If you must use these characters but cannot get them to work as expected, contact Forge Support.

stdin file (optional):
This enables you to choose a file to be used as the standard input (`stdin`) for your program. Arguments are automatically added to `mpirun` to ensure your input file is used.

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

2.1.3.2 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 Linaro DDT is in single process mode. See *Debug single-process programs* for more details about using Linaro DDT with a single process.

Number of processes:
The number of processes that you want to debug. Linaro 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 want 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. Click *Change* to change the MPI implementation.

Note: The choice of MPI implementation is critical to correctly starting Linaro DDT. Your system will normally use one particular MPI implementation. If you are unsure which to choose, try generic, consult your system administrator or Forge Support. A list of settings for common implementations is provided in *MPI distribution notes and known issues*.

Note: If the MPI command you want is not in your `PATH`, or you want to use an MPI run command that is not your default one, you can configure this using the *Options* window (See *System* on *Optional configuration*).

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

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

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

2.1.3.3 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.

For more information on debugging OpenMP programs see *Debug OpenMP programs*.

2.1.3.4 CUDA

If your license supports it, you can also debug GPU programs by enabling CUDA support. For more information on debugging CUDA programs see *NVIDIA GPU debugging*.

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

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

Note: **Detect invalid accesses** (`memcheck`) is not supported with CUDA 12.

Note: Debugging applications using more than one GPU technology (CUDA, ROCm or Intel Xe) is not supported.

2.1.3.5 ROCm

If your license supports it, you can also debug GPU programs by enabling ROCm support. For more information on debugging ROCm programs see *ROCm GPU debugging*.

Note: Debugging applications using more than one GPU technology (CUDA, ROCm or Intel Xe) is not supported.

2.1.3.6 Intel Xe

If your license supports it, you can also debug GPU programs by enabling Intel Xe support. For more information on debugging Intel Xe programs see *Intel Xe GPU debugging*.

Note: Debugging applications using more than one GPU technology (CUDA, ROCm or Intel Xe) is not supported.

2.1.3.7 Memory debugging

Click *Details* to open the *Memory Debugging Options* window.

See *Memory debugging options* for details of the available settings.

2.1.3.8 Environment variables

The optional *Environment Variables* section should contain additional environment variables that should be passed to mpirun or its equivalent. These environment variables can also be passed to your program, depending on which MPI implementation your system uses. Most users will not need to use this section.

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

2.1.3.9 Plugins

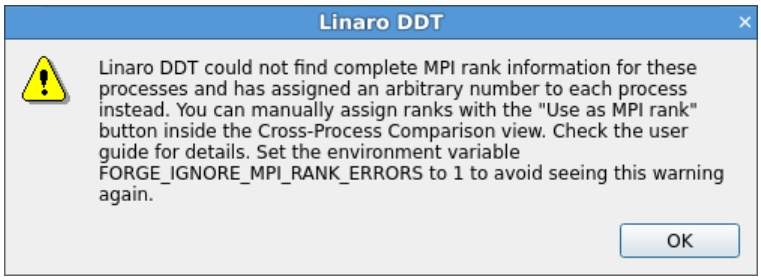
The optional *Plugins* section lets you enable plugins for various third-party libraries, such as the Intel Message Checker or Marmot. See *Use and write plugins* for more information.

2.1.3.10 Run the program

Click *Run* to start your program, or *Submit* if working through a queue (see *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 Linaro 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, 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, Linaro DDT attempts to determine the MPI world rank of each process. If this fails, an error message displays:



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

See *Assign MPI ranks* for details.

2.1.3.11 End the session

To end your current debugging session, select *File* ▸ *End Session*. This closes all processes and stops any running code. If any processes remain you might have to clean them up manually using the `kill` command, or a command provided with your MPI implementation.

2.1.4 `remote-exec` required by some MPIs (DDT)

When using the MPMD variants of *MPICH 3*, *MPICH 4*, or *Intel MPI*, you can use mpirun to start all the processes, then attach to them while they are inside MPI_Init.

This method is often faster than the generic method, but requires the `remote-exec` facility to be correctly configured if processes are being launched on a remote machine. For more information on `remote-exec`, see *Connecting to compute nodes and remote programs (remote-exec)* and *Choose hosts* on *Attach to running programs*.

Note: If Linaro DDT is running in the background (for example, `ddt &`) 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 Linaro DDT a foreground process, or run Linaro DDT again, without using `&`.

If Linaro 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.

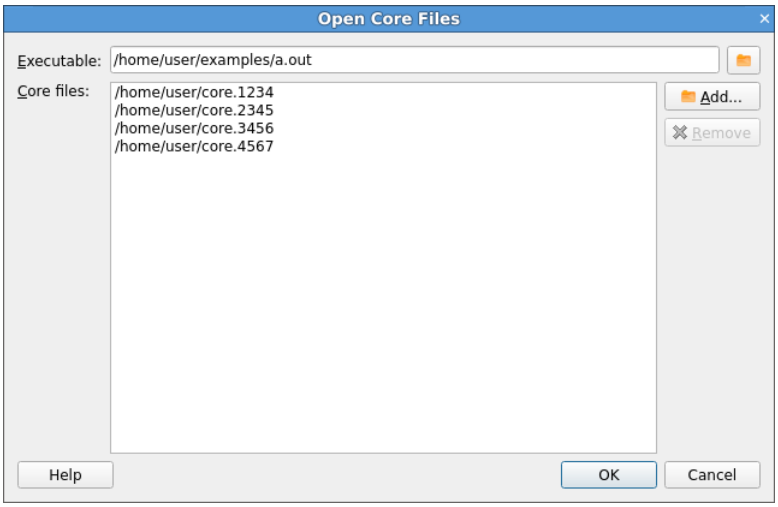
In addition to the listed MPI implementations above, all MPI implementations except for *Cray MPT* Linaro DDT require password-free access to the compute nodes when explicitly starting by attaching.

2.1.5 Open core files

You can open and debug one or more core files generated by your application.

2.1.5.1 Procedure

1. On the Welcome page click *Open Core Files*. The *Open Core Files* window opens.



2. Select an executable and a set of core files, then click *OK* to open the core files and start debugging them.

Note: While Linaro 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.

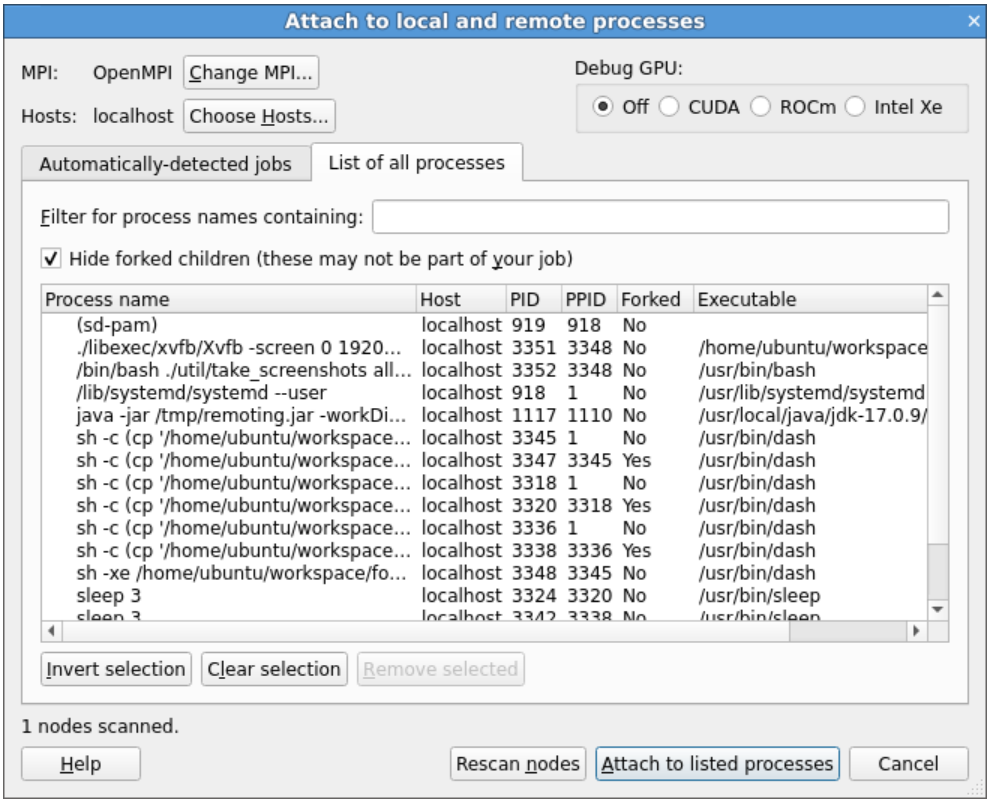
2.1.5.2 Next steps

To end your current debugging session, select *File* ▸ *End Session*.

2.1.6 Attach to running programs

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

Click *Attach to a Running Program* on the Welcome page to open the *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.

2.1.6.1 Automatically-detected jobs

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

The list of detected MPI jobs is shown on the *Automatically-detected jobs* tab of the *Attach* window. Click the header for a particular job to see more information about that job. When 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`

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

2.1.6.2 List of all processes

You can manually select which processes to attach to from the list of processes on the *List of all processes* tab of the *Attach* window.

If you want to attach to a process on a remote host, see *Connecting to compute nodes and 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 *Filter for process names containing* 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 can modify the list by selecting and removing unwanted processes, or alternatively selecting the processes you want to attach to and clicking *Attach to selected processes*. If no processes are selected, DDT attaches to all processes in the list.

On Linux you can 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*. Linaro DDT creates a process group for each distinct executable.

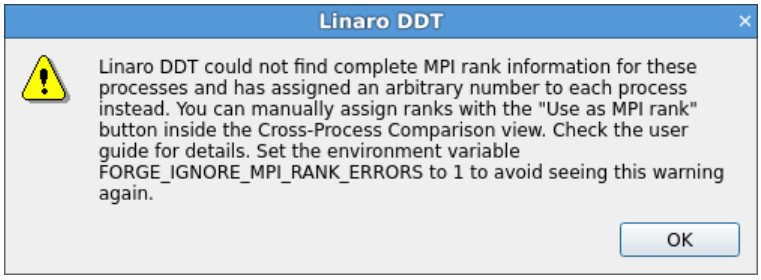
With some supported MPI implementations (for example, Open MPI) Linaro DDT shows MPI processes as children of the (or equivalent) command.

Process name	Host	PID	PPID	Forked	Executable
▾ mpirun	login1	10407	10204	No	/openmpi/mpirun
hello_c	login1	10409	10407	No	/home/user/hello_c
hello_c	login1	10410	10407	No	/home/user/hello_c
hello_c	login1	10411	10407	No	/home/user/hello_c
hello_c	login1	10412	10407	No	/home/user/hello_c
hello_c	login1	10413	10407	No	/home/user/hello_c
hello_c	login1	10414	10407	No	/home/user/hello_c
hello_c	login1	10415	10407	No	/home/user/hello_c
hello_c	login1	10416	10407	No	/home/user/hello_c
hello_c	login1	10417	10407	No	/home/user/hello_c
hello_c	login1	10418	10407	No	/home/user/hello_c

If you click the command, this automatically selects all the MPI child processes.

When you click *Attach to selected/listed processes*, Linaro 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 Linaro DDT. When you end the debug session, Linaro DDT detaches from the processes rather than terminating them. This means you can attach to them again later if you want.

Linaro 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, you might see this message:



If there is no rank, for example, if you have attached to a non-MPI program, you can ignore this message and use Linaro DDT as normal. If there is a rank, you can easily tell Linaro DDT the correct rank for each process via *Use as MPI Rank* in the *Cross-Process Comparison View*. See *Assign MPI ranks* for details.

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

2.1.6.3 Choose hosts

To attach to remote hosts, click *Choose Hosts* in the *Attach* window. The *Hosts* window displays the list of hosts that you can use for attaching.

In the *Hosts* windows you can add and remove hosts, and uncheck hosts that you want to temporarily exclude.

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

The hosts list populates using the *Attach Hosts File*. To configure the hosts, click *File* ▸ *Options* (*Linaro Forge* ▸ *Preferences* on Mac OS X) to open the *Options* window.

Each remote host is scanned for processes, and the result is displayed in the *Attach* window. If you have trouble connecting to remote hosts, see *Connecting to compute nodes and remote programs (remote-exec)*.

2.1.6.4 Use command-line arguments

As an alternative to starting Linaro DDT and using the Welcome page, Linaro 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:

```
user@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:

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

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

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

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

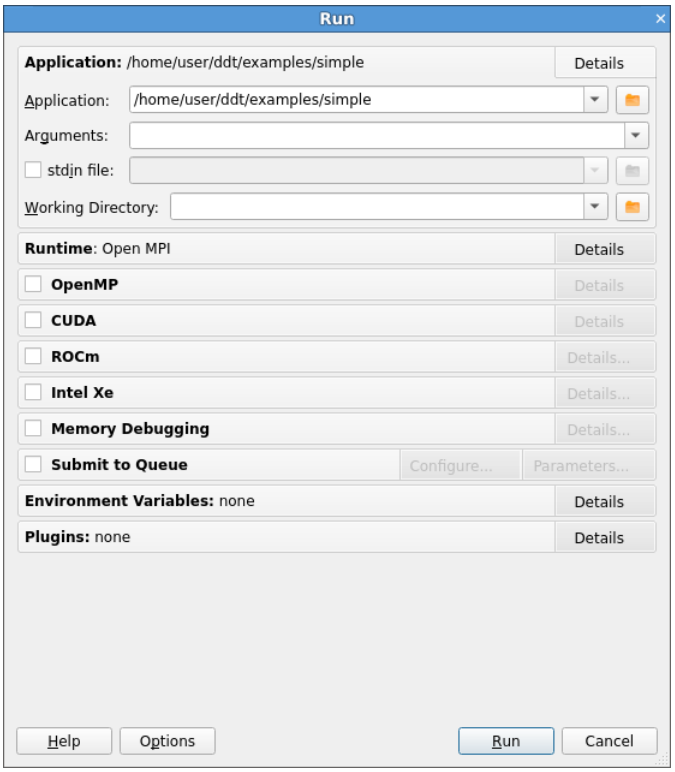
These command-line options work for both single- and multi-process attaching.

2.1.7 Debug single-process programs

You can use the *Run* dialog to start debugging single-process programs.

2.1.7.1 About this task

- If you have a single-process license you will immediately see the *Run* dialog that is appropriate to run a single-process program.
- If you have a multi-process license you can clear the *MPI* checkbox to run a single-process program.



2.1.7.2 Procedure

1. Type the full file path to your application, or browse and select your application.
2. If required, type the arguments to pass to your program.
3. Click *Run* to start your program.

Note: If your program has been compiled with Intel `ifort`, you might not see your code and highlight line when Linaro DDT starts. This is because this compiler creates a pseudo `MAIN` function, above the top level of your code. To fix this you can either open your **Source code viewer**, add a breakpoint in your code, then play to that breakpoint, or you can use the Step Into function to step into your code.

2.1.7.3 Next steps

To end your current debugging session select *File* ▸ *End Session*. This will close all processes and stop any running code.

2.1.8 Debug OpenMP programs

When you run an OpenMP program, set the *Number of OpenMP threads* value to the number of threads you require. Linaro 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 when you debug OpenMP programs:

- 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.
- 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.
- You cannot step into a parallel region. Instead, select *Step threads together* 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.
- You cannot step out of a parallel region. Instead, use the *Run to here* command to leave it. Most OpenMP libraries work best if you keep *Step threads together* selected 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.
- Leave *Step threads together* clear when you are outside a parallel region, as OpenMP worker threads usually do not follow the same program flow as the main thread.
- To control threads individually, use *Focus on Thread*. 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.
- 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.

- Parallel regions may be displayed as a new function in the *Stacks* view. Many OpenMP libraries implement parallel regions as automatically-generated `outline` functions, and Linaro 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.
- 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.
- 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.
- 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 Linaro 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 Linaro DDT with OpenMP and would like to tell us about your experiences, please contact Forge Support, with the subject title *OpenMP feedback*.

2.1.9 Debug MPMD programs

The easiest way to debug multiple program, multiple data (MPMD) programs is by using Express Launch to start your application. You can also debug MPMD programs without Express Launch or in Compatibility mode.

2.1.9.1 About this task

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

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

For more information on Express Launch, and compatible MPI implementations, see *Express Launch (DDT)*.

If you are using Open MPI, MPICH 3, MPICH 4, or Intel MPI, DDT can be used to debug MPMD programs without Express Launch. This procedure shows how to start an MPMD program in DDT.

2.1.9.2 Procedure

1. (MPICH 3 and Intel MPI only) Select the MPMD variant of the MPI Implementation on the *System* page of the *Options* window. For example, for MPICH 3 select *MPICH 3 (MPMD)*.
2. On the Welcome page click *Run*.
3. In *Application*, choose one of the MPMD programs. It does not matter which executable you choose.
4. Enter the total amount of processes for the MPMD job in *Number of Processes*.
5. In the MPI section of the *Run* window, enter an MPMD style command line in *mpirun Arguments*. Make sure that the sum of processes in this command is equal to the number of processes set in *Number of Processes*. For example:

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

or

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

6. Click *Run*.

2.1.9.3 Example: 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 *mpirun Arguments* (or in `appfile`) with:

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

2.1.10 Manual launch of multi-process non-MPI programs

DDT can only launch MPI programs and scalar (single process) programs. You need to use the *Manual Launch (Advanced)* button on the **Welcome Page** to debug multi-process and multi-executable programs.

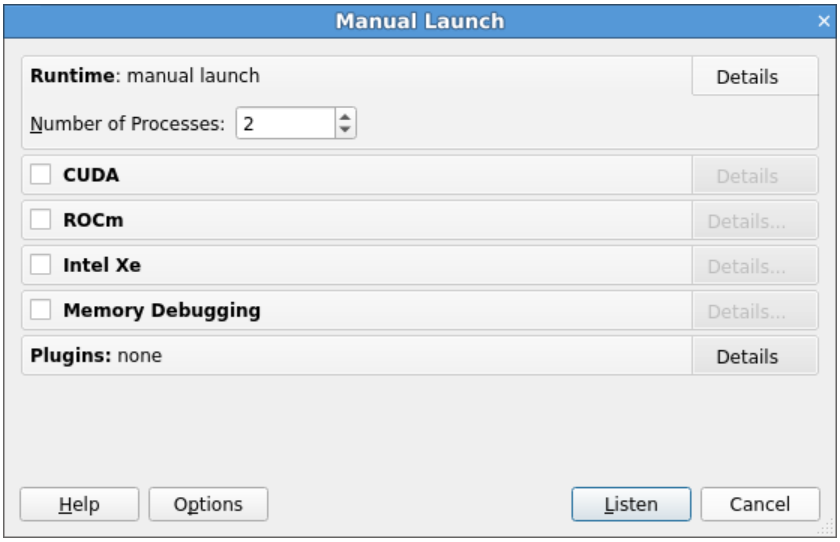
2.1.10.1 About this task

These programs do not 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 Linaro DDT session.

You must run each program you want to debug manually using the `forge-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 session, if your license permits it. Each program you run will show up as a new process in the Linaro DDT window. In this example we show you how to debug both client and server in the same session.

2.1.10.2 Procedure

1. On the Welcome page click *Manual Launch (Advanced)*.
2. Set *Number of Processes* to 2.

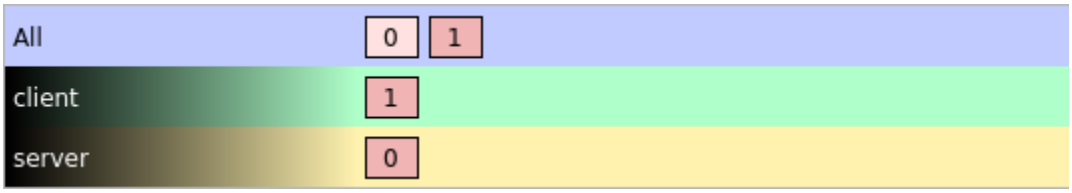


3. Click *Listen*.
4. At the command line run:

```
forge-client server &
forge-client client &
```

2.1.10.3 Results

The server process displays as process 0, and the client as process 1 in Linaro DDT.



2.1.10.4 Next steps

- After you have run the initial programs, you can add extra processes to the DDT session. For example, to add extra clients using forge-client in the same way use `forge-client client2 &`.
- If you select *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.

2.1.11 Start a job in a queue

In most cases you can debug a job 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, a message is displayed prompting you with the option to debug the job when it starts.

See *Express Launch (DDT)* and *Reverse Connect* for more details.

If Linaro DDT has been configured to be integrated with a queue/batch environment, as described in *Integration with queuing systems* then you can use Linaro DDT to submit your job directly from the user interface. In this case, a *Submit* button is displayed on the *Run* window, instead of a *Run* button. When you click *Submit* on the *Run* window the queue status is displayed until your job starts. Linaro DDT will execute the display command every second and show you the standard output. If your queue display is graphical or interactive you cannot use it here.

If your job does not start or you decide not to run it, click *Cancel Job*. If the regular expression you entered for getting the job id is invalid, or if an error is reported, Linaro DDT will not be able to remove your job from the queue. In this case we strongly recommend 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.

When your job is running, it connects to Linaro DDT and you can debug it.

2.1.12 Job scheduling with jsrun

Launching jobs with jsrun in a job scheduling system enables the topology of processes and threads on the node to be split into individual resource sets (the number of GPUs, CPUs, threads, and MPI tasks). You can specify the amount of computational resource allocated to a resource set.

How you decide to allocate resources has an impact on the runtime of Linaro Forge tools. For example, it is possible to allocate all of the CPUs on the node to just one resource set. Alternatively, you could allocate each CPU to its own resource set. In this case there are as many resource sets as there are CPUs on the node.

The more resource sets you have on each node, the longer the runtime is when using Linaro Forge. To minimize runtime, we recommend that you aim to reduce the number of resource sets required.

For example, we recommend:

```
jsrun --rs_per_host=1 --gpu_per_rs=0 --cpu_per_rs=42 --tasks_per_rs=42 ...
```

to launch a job with 42 MPI processes per node in a single resource set, instead of:

```
jsrun --rs_per_host=42 --gpu_per_rs=0 --cpu_per_rs=1 --tasks_per_rs=1 ...
```

which launches 42 MPI processes per node, but uses 42 resource sets.

2.1.13 Use custom MPI scripts

On some systems a custom mpirun replacement is used to start jobs, such as mpiexec. Linaro DDT normally uses whatever the default for your MPI implementation is, so for Open MPI it would look for mpirun and not mpiexec. Here we explain how to configure Linaro 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 Linaro DDT supports them both.

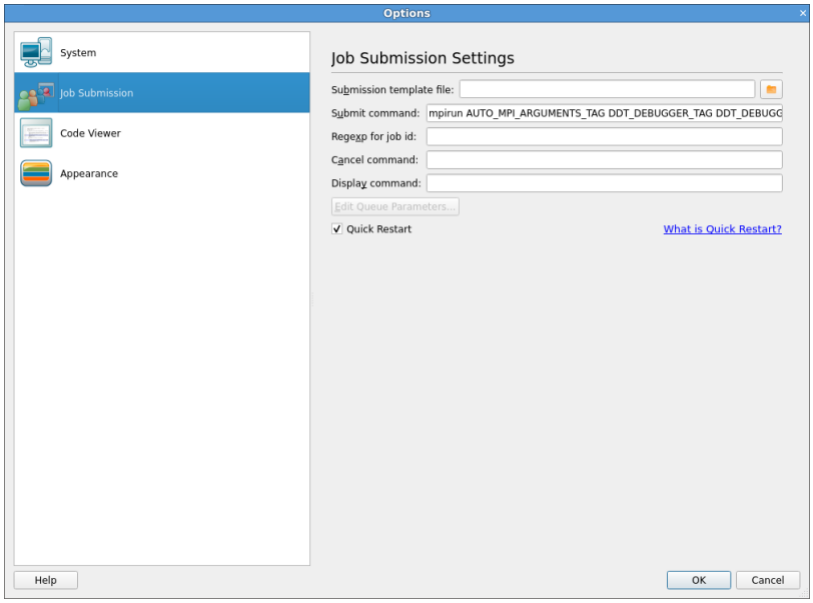
In the first way, you pass all the arguments on the command-line, like this:

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

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

- The number of processes (4 in the above example).
- The name of your program (`/home/mark/program/chains.exe`).
- 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 Linaro DDT, you adapt the queue submission system described in *Job scheduling with jsrun*. For this mpiexec example, the settings are:



As you can see, most of the settings are left blank. There are some differences between the *Submit command* in Linaro DDT and what you would type at the command-line:

- The number of processes is replaced with `NUM_PROCS_TAG`.
- The name of the program is replaced by the full path to `forge-backend`.
- The program arguments are replaced by `PROGRAM_ARGUMENTS_TAG`.

Note: It is not necessary to specify the program name here. Linaro DDT takes care of that during its own startup process. The important thing is to make sure your MPI implementation starts `forge-backend` instead of your program, but with the same options.

In the second way, you start a job using a custom mpirun replacement with a settings file:

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

Where `myfile.nodespec` might contain something similar to the following:

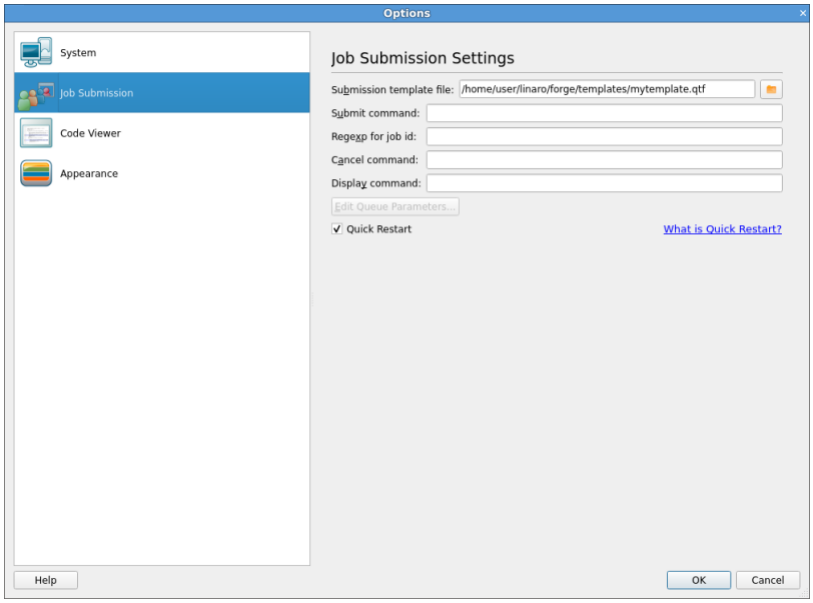
```
comp00 comp01 comp02 comp03 : /home/mark/program/chains.exe /tmp/mydata
```

Linaro 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:

```
comp00 comp01 comp02 comp03 : DDTPATH_TAG/libexec/forge-backend DDT_DEBUGGER_ARGUMENTS_TAG PROGRAM_
↪ARGUMENTS_TAG
```

This follows the same replacement rules described above and in detail in *Integration with queuing systems*.

The settings in the *Options* window for this example might be:



Note *Submit command* and *Submission template file* in particular. Linaro DDT creates a new file and appends it to the submit command before executing it. In this case, `mpiexec -config /tmp/ddt-temp-0112` or similar is executed. Therefore, any argument like `-config` must be last on the line, because Linaro DDT adds a file name to the end of the line. If there are any other arguments, they can come first.

If your system uses a non-standard startup command, Linaro recommends that you read *Job scheduling with jsrun*, which describes many features that might be useful to you.

If you do use a non-standard command, contact Forge Support.

2.1.14 Start DDT from a job script

The usual way of debugging a program with Linaro DDT in a queue/batch environment is with Reverse Connect where it connects back from inside the queue to the user interface (see *Reverse Connect*). You can also debug a program in a queue/batch environment by configuring Linaro DDT to submit the program to the queue for you (see *Start a job in a queue*). This procedure describes another way - how to start Linaro DDT from a job script that is submitted to the queue/batch environment.

2.1.14.1 Procedure

1. Configure Linaro DDT with the correct MPI implementation.
2. Disable queue submission in the Linaro DDT options.
3. Create a job script that starts Linaro 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. The `--once` argument tells DDT to exit when the session ends.
4. Submit the job script to the queue.

2.1.15 Numactl (DDT)

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

2.1.15.1 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 (DDT)*.

```
$ 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. When using compatibility mode, you must specify the full path to `numactl` in the *Application* field of the *Run* window. You can find the full path by running:

```
which numactl
```

Enter the name of the required application in *Arguments*, 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. See *Memory Statistics*.

2.1.15.2 Non-MPI Programs

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

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

When 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.

2.1.16 Python debugging

This task describes how to debug Python scripts.

2.1.16.1 About this task

These Linaro DDT features are supported in Python debugging:

- Debugs Python scripts running under the CPython interpreter. For supported versions, see *Reference table*.
- Decodes the stack to show Python frames, function names, and line numbers.
- Displays both stacks in mixed Python/native programs where the script calls out into a native C library.
- Displays Python local variables when a Python frame is selected.
- Evaluations which can also include Python expressions and statements.
- The **Multi-Dimensional Array Viewer**. See *Multi-Dimensional Array Viewer (MDA)* for more information.

Note: Python sequence types (`list`, `tuple`, `range`) are supported by the **Multi-Dimensional Array Viewer**. Other built-in iterable types are also supported when evaluated as `list`, for example `list(my_set)`.

- Breakpoints, Tracepoints, and stepping in Python code.
- Stop on raised and unhandled exception. See *Default breakpoints* for more information.
- After a module is imported, you can see its Python source files listed in the *Project Files* tree.
- Debugs MPI programs written in Python using mpi4py.
- Offline Python debugging.

This feature is useful when debugging a mixed C, C++, Fortran, and Python program.

Note: Python global variables are only shown in the *Locals* tab if the selected frame is at the module level. To see a global variable, you can add it in the *Evaluate* window. You can see all the global variables, if you add `globals()`.

These Linaro DDT features are not supported in Python debugging:

- Manual launch and attaching to a Python process.

- Python debugging by opening a core file.
- Watchpoints.
- Python programs with multiple Python threads running concurrently as this causes Linaro DDT to hang.
- Sub-processing libraries (such as `multiprocessing`) because it forks separate processes.
- Stepping from Python frames into native frames.
- Memory debugging only covers the Python interpreter.

2.1.16.2 Procedure

1. To debug Python scripts, specify the Python interpreter followed by `%allinea_python_debug%` and then the path to the script that you wish to debug. For example:

```
$ ddt python3 %allinea_python_debug% my-script.py
```

2. To debug Python scripts that use MPI, the same applies, except mpirun is also appended to the beginning:

```
$ ddt mpirun -np 4 python3 %allinea_python_debug% my-mpi-script.py
```

3. When passing arguments, they must appear after `%allinea_python_debug%` and the name of your script. To run the demo in the examples folder, change into the examples folder.
4. Run:

```
$ make -f python.makefile
```

5. Run:

```
$ ../bin/ddt python3 %allinea_python_debug% python-debugging.py
```

   Note: On loading into Linaro DDT you will be inside the C code. This is normal as you are debugging the python binary. Depending on the interpreter that you are debugging you may also see a message about missing debug symbols. Clicking *Play/Continue* once after launching will bring you to the first line of your script.

6. Click *Run*.
7. Click *Play/Continue* to run to the first line of the script.
8. Set a breakpoint on a line inside the `call_out_to_a_library` function.
9. Use the *Add breakpoint* dialog to set a breakpoint on the function name *library_function*.
10. Click *Play/Continue* to run to the Python function and observe that local variables are visible.
11. Click *Play/Continue* again to run to the native function and observe how the stack appears when calling out of the interpreter.

Note: To disable Python debugging, omit `%allinea_python_debug%` from the command line.

2.1.17 Save and load sessions

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

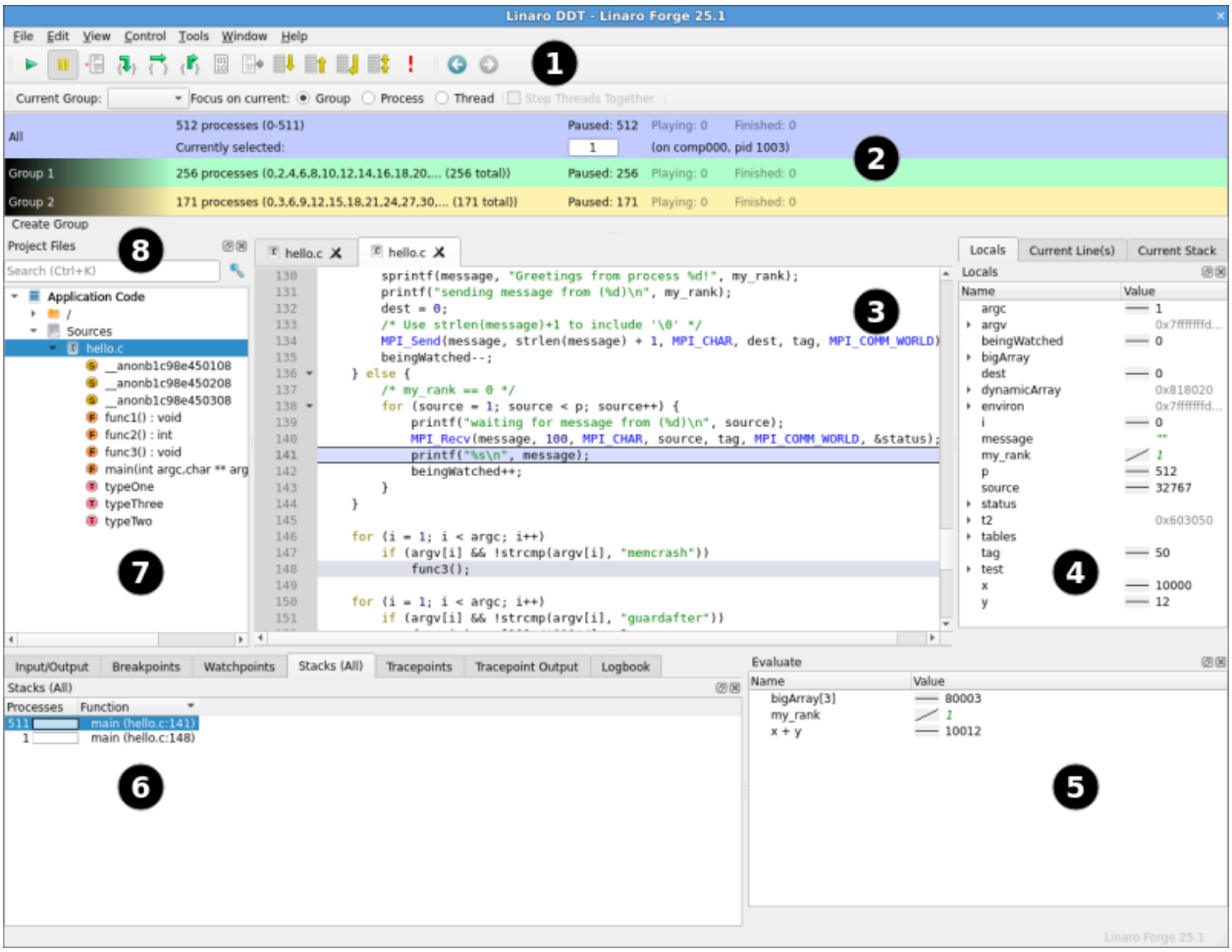
Linaro DDT can also load and save all these options concurrently to minimize the inconvenience in restarting sessions. Saving the session stores such things as *Process Groups*, the contents of the *Evaluate* window, and more. This makes it easy to debug code with the same parameters set every time.

To save a session select *File* ▸ *Save Session*, enter a file name for the save file (or select an existing file), then click *OK*. To load a session select *File* ▸ *Load Session*, choose the correct file, then click *OK*.

2.2 DDT user interface

Linaro DDT uses a tabbed-document interface to display multiple documents. This means you can 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 Linaro DDT is a dockable window that you can drag around by a handle, usually on the top or left edge. You can also double-click or drag a component outside of Linaro DDT, to form a new window. You can hide or show most of the components using the *View* menu. You can also select preset or custom metrics displays. The screenshot shows the default layout.



The table shows the main components:

Key	Component
1	*Process controls*
2	*Process groups*
3	*Source Code view*
4	*Variables and stack of current process or thread*
5	*Evaluate window*
6	*Parallel Stack view*, *IO*, *Breakpoints*, *Watchpoints*, *Tracepoints*, *Tracepoint output*, *Logbook*
7	*Project files*
8	*Find a file or function*

Note: On some platforms, the default screen size might be insufficient to display the status bar. If this occurs, expand the Linaro DDT window until it is completely visible.

2.3 Source code (DDT)

Describes source code viewing, editing, and rebuilding features.

Linaro DDT integrates with the Git, Subversion, and Mercurial version control systems, and provides static analysis to automatically detect many classes of common errors.

2.3.1 Source code viewer

The source code editing and rebuilding capabilities are not designed for developing programs from scratch. 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 you are running remotely (using the remote client) or if you are connected directly to your system.

2.3.1.1 View source code

When you start 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 in the *Project Files* tab of the **Project Navigator** window. Click on the file name to view the source file.

When a selected process is stopped, the **Source Code viewer** will automatically move 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. Click the '+' glyph to expand the function again.

2.3.1.2 Edit source code

Source code can be edited in the **Source code viewer**. 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 can be opened, saved, reverted, and closed from the *File* menu.

Note: Information from Linaro 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, you can open it by right-clicking in the editor and choosing *Open <filename>.<extension>*. There is a global shortcut on function key `F4`, or you can use *Edit* ▸ *Switch Header/Source*.

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, select *File* ▸ *Options* to open the *Options* window (Linaro Forge Preferences on Mac OS X) then enter the path to the preferred editor in *Editor*, for example `/usr/bin/gedit`.

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

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.

2.3.1.3 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, then 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. When a rebuild succeeds we recommend that you restart the session with the new build by choosing *File* ▸ *Restart Session*.

2.3.1.4 Committing changes

Changes to source files can be committed using Git, Mercurial, or Subversion. To commit changes, choose *File* ▸ *Commit*, enter a commit message in the *Commit changes* dialog then click *Commit*.

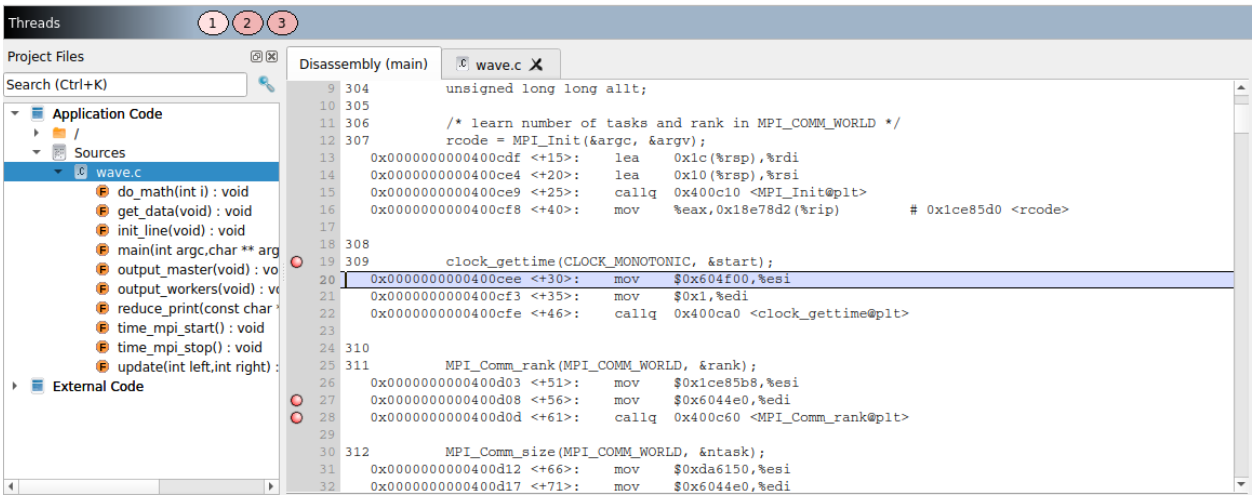
2.3.2 Assembly debugging

You can view disassembly, step over instructions, step into instructions, and set breakpoints on instructions in the *Disassembly* viewer when you are in assembly debugging mode.

2.3.2.1 Enable assembly debugging

To enable assembly debugging mode click the *0101* button in the toolbar.

When you enable assembly debugging mode the *Disassembly* viewer opens where you can view the disassembly of the current symbol that contains the program counter and changes the behavior of the step buttons to operate on the instruction level.



The *Disassembly* viewer auto updates the disassembly when the current symbol that contains the program counter changes.

When assembly debugging mode is disabled, the *Disassembly* viewer closes and reverts the behavior of the step buttons back to stepping source lines.

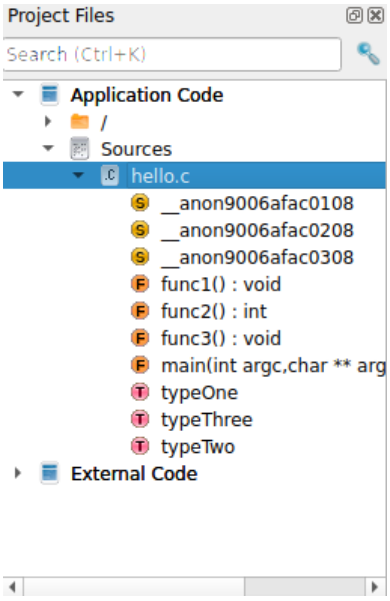
2.3.2.2 Breakpoints

The *Disassembly* viewer enables you to set breakpoints on instructions and also on source lines.

Every breakpoint is listed on the *Breakpoints* tab towards the bottom of the window and can be further edited from the *Breakpoints* tab. See *Set breakpoints*.

2.3.3 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 *Source Code viewer*. You can also expand a source file to see a list of classes, functions, and so on defined in that source file (C / C++ / Fortran only).



Click on a source code element (class, function, and so on) to display it in the *Source Code viewer*.

2.3.3.1 Application Code and External Code

Linaro 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 helps you 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. When you have made your changes click *OK* to update the *Project Files* tree.

2.3.4 Finding lost source files

Sometimes not all source files are found automatically. This can also occur, for example, if the executable or source files have been moved since compilation. To search for source files in additional directories, right-click in the *Project Files* tab, and select *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 *Add File*.

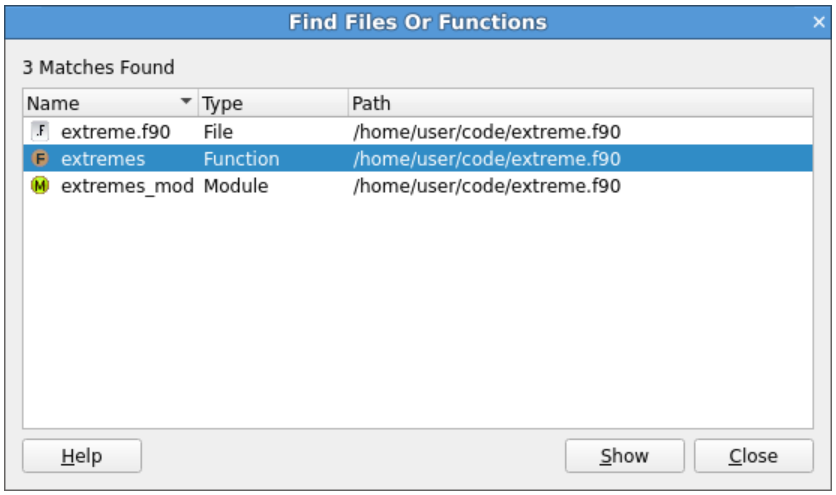
Any directories or files you have added are saved and restored when you use *File ▸ Save Session* or *File ▸ Load Session*. If Linaro DDT does not find the sources for your project, you might find these commands save you a lot of unnecessary clicking.

2.3.5 Find Files or Functions

The **Find Files Or Functions** dialog is displayed above the sources file list in the **Project Files** tree.

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

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



2.3.5.1 Find code or variables

You can use *Edit* ▸ *Find* to find occurrences of an expression in the currently open source file.

The search starts from the current cursor position for the next or previous occurrence of the search term. Click the magnifying glass icon for more search options:

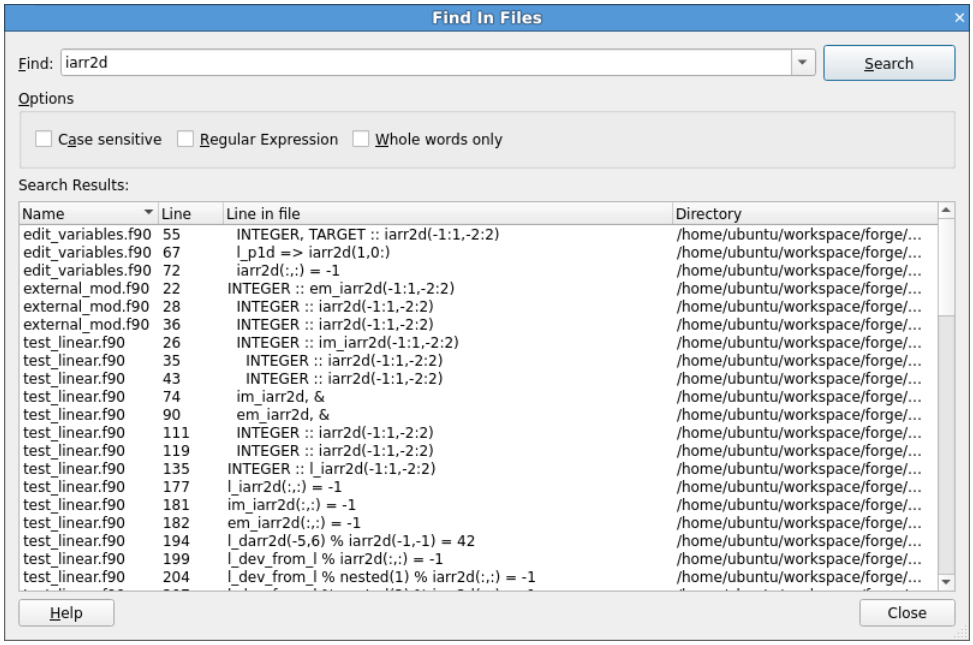
Case sensitive:
When selected, the search is case sensitive, so for example, `Hello` can not match `hello`.

Whole words only:
When selected, the search only matches your search term against whole words in the source file. For example, `Hello` does not match `HelloWorld`.

Use Regular Expressions:
When selected, your search can use Perl-style regular expressions.

2.3.5.2 Find code or variables in files

You can use *Edit* ▸ *Find In Files* to search all source and header files associated with your program. Click the search results list to display the file and line number in the **Source Code viewer**. This can be used for setting a breakpoint at a function.



Case sensitive:
When selected, the search is case sensitive, so for example, `Hello` does not match `hello`.

Whole words only:
When selected, the search only matches your search term against whole words in the source file. For example, `Hello` does not match `HelloWorld`.

Regular Expression:
When selected, the search term is interpreted as a regular expression rather than a fixed string. The syntax of the regular expression is identical to that described in *Job ID regular expression*.

2.3.6 Go To Line

The Go To Line feature enables you to go directly to a particular line of source code.

Click *Edit* ▸ *Go To Line* to display a dialog. Enter the line number in the source code that you want to view, then click *OK*. This will take you to that line in the code (if the line exists). You can also use the `Ctrl+L` shortcut to open this dialog.

2.3.7 Navigate through source code history

After jumping to a particular source code location or opening a new file, you can return to the previous location using *Edit* ▸ *Navigate backwards in source code history* (or click *Navigate backwards in source code history* on the toolbar). You can navigate back to several previous locations in the source code.

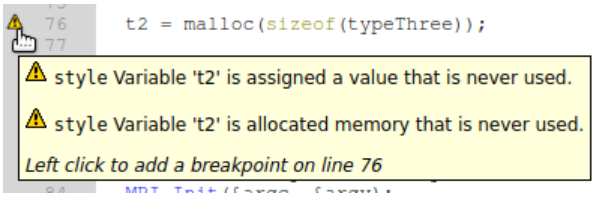
After navigating backwards, you can use *Edit* ▸ *Navigate forwards in source code history* (or click *Navigate forwards in source code history* in the toolbar) to return to the original location.

2.3.8 Static analysis

Static analysis is a powerful companion to debugging. Linaro DDT helps you 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.

Linaro 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:

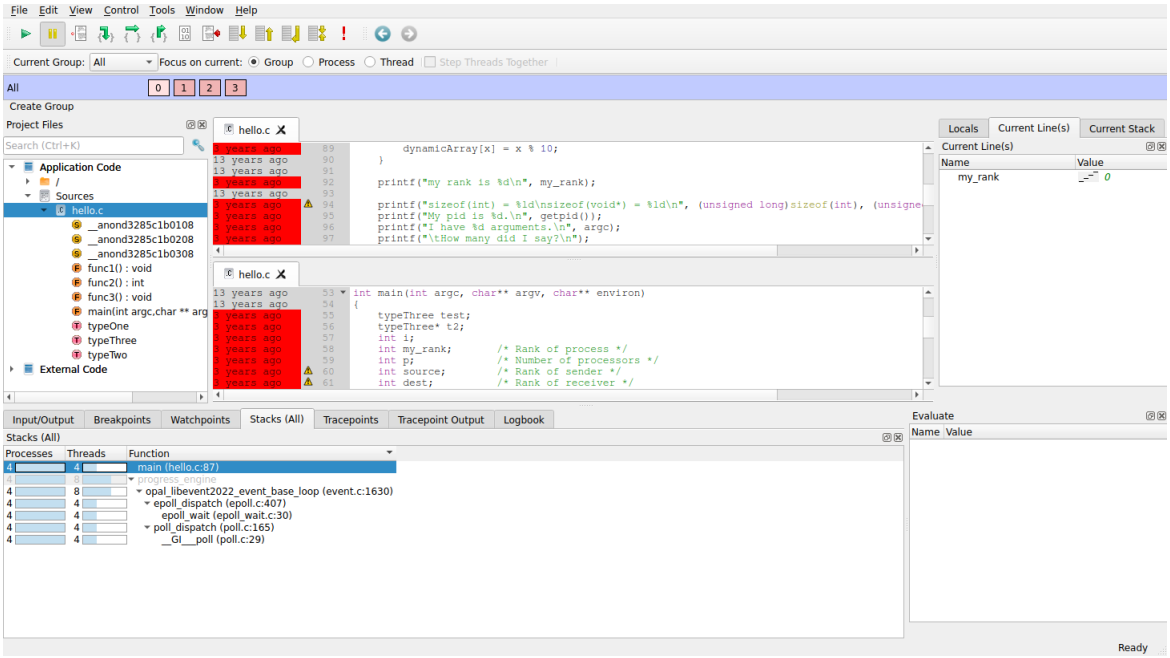
- Buffer overflows. Accessing beyond the bounds of heap or stack arrays.
- Memory leaks. Allocating memory within a function and there being a path through the function which does not deallocate the memory and the pointer is not assigned to any externally visible variable, nor returned.
- Unused variables,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



Static analysis is not guaranteed to detect all, or any, errors. An absence of warnings does not mean there are no bugs.

2.3.9 Version control information

The version control integration in Linaro DDT and Linaro MAP enables you to see line-by-line information from Git, Mercurial, or Subversion next to the source file. Version control information is color-coded to indicate the age of the source line.

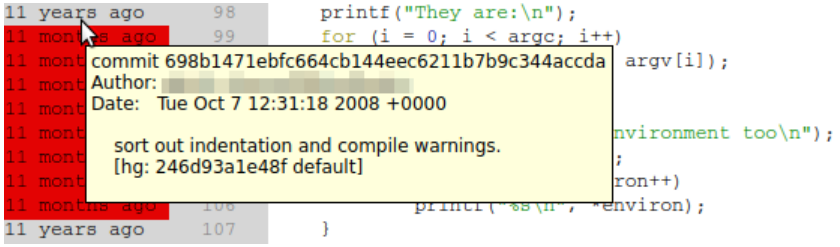


To view version control information, select *View* ▸ *Version Control Information*. The column displayed to the left of the source code viewer shows how long ago the line was added or modified. Each line in the column is highlighted in a color that indicates its age:

- Lines modified in the current revision are highlighted in red.
- Lines that have been modified but not committed are highlighted in purple.
- All other lines are highlighted with a blend of transparent blue and opaque green, where blue indicates old changes and green indicates recent changes.

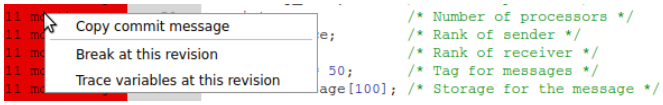
Note: Changes that have not been committed can only be viewed if you are using Git. Version control information for files with uncommitted changes is not displayed if you are using Mercurial or Subversion.

If you hover the cursor over the version control information column, a tooltip displays a preview of the commit message for the commit that last changed the line.



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

To copy the commit message, right-click on the desired row in the column then select *Copy commit message*.



Also see *Version control breakpoints and tracepoints*.

2.4 Control program execution

Whether debugging multi-process code or 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**.

For single-process mode the commands and behaviors are identical, but apply to only a single process. In this case you do not need to work with process groups.

2.4.1 Process control and process groups

MPI programs are designed to run as more than one process and can span many machines. Linaro DDT lets you group these processes so that you can perform actions on more than one process at a time. The status of processes is displayed in the **Process Group viewer**.

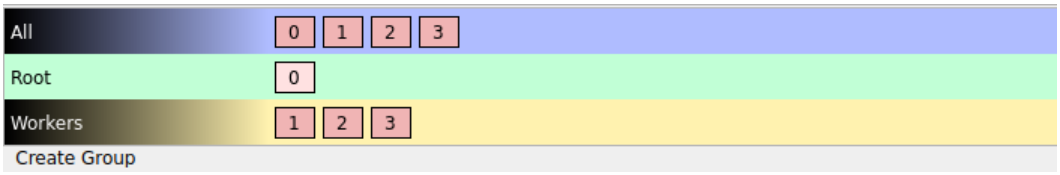
The **Process Group viewer** is displayed (by default) at the top of the screen with multi-colored rows. Each row relates to a group of processes and operations that can be performed on the currently highlighted group (for example, playing, pausing, and stepping) by clicking on the toolbar buttons. You can 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 at any time, with the exception of the *All* group, which cannot be modified.

To add a group click the *Create Group* button, or right-click in the **Process Group viewer** and choose *Create Group* from the context-sensitive menu that is displayed. This context 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 source 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. These methods are called the *Detailed view* and the *Summary view*.

2.4.1.1 Detailed view

The Detailed view is ideal for working with small numbers of processes. If your program has 32 processes or less, DDT defaults to the Detailed view. You can use the context menu to switch to this 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.

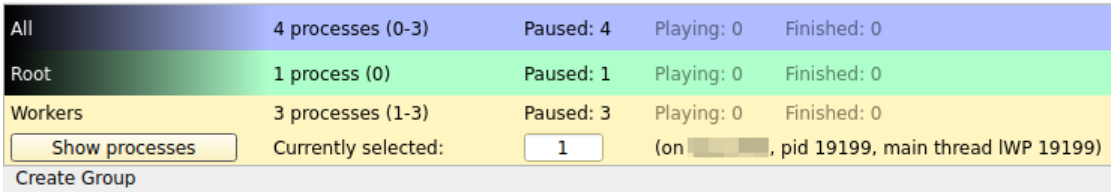
To copy processes from one group to another, click and drag the processes. To delete a process, press the `Delete` key. When modifying groups it is useful to select more than one process by holding down one or more of the following keys:

Key	Description
`Ctrl`	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 + drag` to move a window. You must disable this feature in your window manager if you want to use DDT's area select.

2.4.1.2 Summary view

The Summary view is ideal for working with moderate to large numbers of processes. If your program has 32 processes or more, DDT defaults to this view. You can use the context menu to switch to this view.



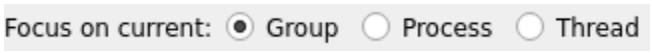
In the Summary view, individual processes are not shown. Instead, for each group, Linaro DDT shows:

- 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 **...**. If you hover the mouse over the list you can see more details.
- The number of processes in each state (playing, paused, or finished). If you hover the mouse over each state you can see 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, then pressing Enter. Only ranks belonging to the current group will be accepted.

Show processes lets you 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.

2.4.2 Focus control

The focus control allows you to focus on individual processes, or threads, or on process groups. When focused on a particular process or thread, actions such as stepping, playing/pausing, or adding breakpoints, will only apply to that process or thread rather than the entire group.

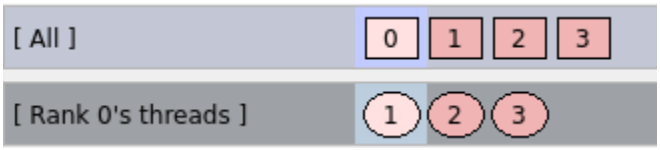


Focusing affects a number of different controls in the main window. The user interface will change depending on whether you are focused on group, process, or thread. The relevant information about your currently focused object will be displayed.

Note: Focus control does not affect all windows. Windows such as the Multi-Dimensional Array Viewer, Memory Debugger, and Cross-Process Comparison are unchanged.

2.4.2.1 Process group viewer

The changes to the process group viewer are the most obvious in the user interface when using focus control. When focus on current group is selected, you see your process groups. If you switch to focus on current process or thread, the view changes to show the processes in the currently selected group, with their corresponding threads.



If there are 32 threads or more, by default the threads will be displayed using a Summary view (as in the Process Group View). You can change the view mode using the context menu.

If focused on process, a tooltip shows the OpenMP thread ID of each thread, if the value exists.

2.4.2.2 Breakpoints tab

The **Breakpoints** tab is filtered to only display breakpoints relevant to your current group, process, or thread. When focused on a process, the **Breakpoints** 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.

2.4.2.3 Source Code viewer

The line highlight color in the Source Code viewer shows you a stack back trace of where each thread is in the call stack. This is 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 you add breakpoints using the Source Code viewer, they are added for the group, process, or thread that is currently focused.

2.4.2.4 Parallel Stack View

The **Parallel Stack View** can also be filtered by focusing on a particular process group, process, or thread.

2.4.2.5 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 process group, the entire group is affected
- When focused on thread, only the current thread is executed
- When focused on process, only the current process is executed. In this case you can also use the **Step threads together** feature.

2.4.2.6 Step threads together

The **Step threads together** feature is only available when focused on process. If this option is enabled, the threads in the current process are synchronized when performing actions such as stepping, pausing, and using **Run to here**.

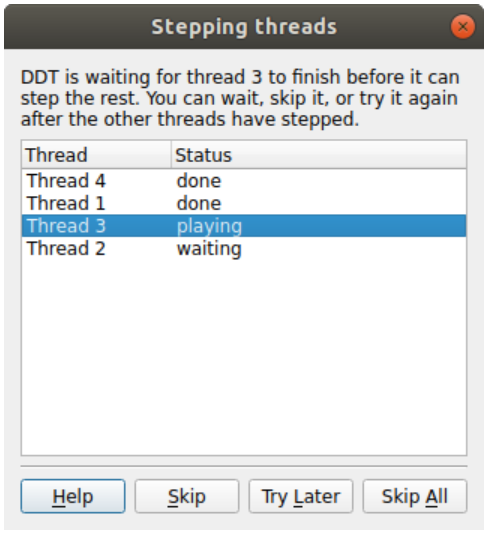
For example, if you have a process with two threads and you choose *Run to here*, your program will be paused when either of the threads reaches the specified line. If **Step threads together** is selected, both of the threads will be played to the specified line before the program is paused.

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, because threads can get "left behind" inside system-specific locking libraries and may not enter the next parallel region on the first attempt.

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:

- One or more threads may branch into a different section of code (and never reach the target). This is especially common in OpenMP codes, where worker threads are created and remain in holding functions during sequential regions.
- As most of the supported debug interfaces cannot play arbitrary groups of threads together, this behavior is simulated 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). Linaro DDT attempts to resolve this automatically but cannot always do so.

If either of these conditions occur, the **Stepping threads** window opens, displaying the threads which have not yet reached their target.



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. Linaro 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:** Linaro 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 closes the window.

2.4.3 Start, stop, and restart a program

If a program is running you can end it, run it again, or run another program using commands on the *File* menu. The *File* menu can be accessed at almost any time.

When the 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.

Restart a program: When a job has run to the end, you are asked if you want to restart the job. If you select *Yes*, any remaining processes are killed, temporary files are cleared, and the the session restarts from scratch with the same program settings.

Stop a program: When ending a job, Linaro DDT attempts to shut down all the processes and clear up any temporary files. If this fails for any reason you may have to manually `kill` your processes using kill, or a method provided by your MPI implementation or job scheduler such as `scancel` for SLURM.

2.4.4 Step through a program

To continue a program playing click *Play/Continue*.

To stop a program playing click *Pause*.

In multi-process mode these commands will start/stop all the processes in the current process group (see *Process control and process groups*).

There are three different types of step available:

- *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.
- *Step Over* moves to the next line of source code in the bottom stack frame.
- *Step Out* executes the rest of the function and then stops 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 step out of the main function, as doing this ends your program.

2.4.5 Stop messages

There are five reasons why your program might be automatically paused:

- The program hit one of the default breakpoints, for example, `exit` or `abort`. See *Default breakpoints* for more information.
- The program hit a user-defined breakpoint, that is a breakpoint shown in the *Breakpoints* view.
- The value of a watched variable changed.
- The program was sent a signal. See *Signal handling* for more information.
- The program encountered a memory debugging error. See *Pointer error detection and validity checking* for more information.

The message displayed informs you why the program was paused. To copy the message text to the clipboard, select it, right-click, then select *Copy*.

If you want to suppress these messages, for example if you are playing from one breakpoint to another, use the *Control Messages* menu to enable or disable stop messages.

2.4.6 Set breakpoints

You can add a breakpoint from two windows.

2.4.6.1 Use the Source Code viewer

Right-click in the **Source Code viewer** where you want to place a breakpoint, then choose to add a breakpoint.

You can also add a breakpoint by clicking in the margin to the left of the line number.

If you have lots of source code files, use the *Find/Find In Files* window to search for a particular function.

In multi-process mode, this sets the breakpoint for every member of the current process group.

Every breakpoint is listed on the *Breakpoints* tab.

If you add a breakpoint at a location where there is no executable code, the line you selected as having a breakpoint is highlighted. However, when the breakpoint is reached, the program stops at the next executable line of code.

2.4.6.2 Use the Add Breakpoint window

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

You can 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. In this case, select *Function* and enter the name of the function in the field next to it.

You can specify which process group/process/thread you want the breakpoint to apply to using the *Applies To* section. You can also make the breakpoint conditional by selecting the *Condition* check box and entering a condition in the field next to it.

2.4.6.3 Pending breakpoints

Note: Pending breakpoints are not supported on all platforms.

If you try to add a breakpoint in a function that is not defined, you will be asked 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.

2.4.7 Conditional breakpoints

The *Breakpoints* tab displays all the breakpoints in your program. You can add a condition to any breakpoint by clicking on the *Condition* cell in the breakpoints 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 to automatically set it as the condition.

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

We recommend that you avoid using functions with side effects as they 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, we recommend that you use brackets explicitly, to ensure correct evaluation.

2.4.8 Suspend breakpoints

To deactivate a breakpoint, either:

- Clear the activated column in the *Breakpoints* tab. To reactivate a breakpoint, select the activated column.
- Right-click the breakpoint icon in the source code editor and choose *Disable*. To reactivate a breakpoint, right-click and choose *Enable*.
- Hold `Shift` and select a breakpoint icon in the source code editor.

Breakpoints that are disabled are grayed out.

2.4.9 Delete breakpoints

There are a number of ways to delete breakpoints:

- Right-click on the breakpoint in the *Breakpoints* tab and select *Delete breakpoint*.
- Right-click in the file/line of the breakpoint while in the correct process group, then select *Delete breakpoint*.
- Click the breakpoint icon in the margin to the left of the line number in the **Source Code viewer**.

2.4.10 Load and save breakpoints

To load or save the breakpoints in a session, right-click in the *Breakpoints* tab and select *Load/save*. Breakpoints are also loaded and saved as part of the load/save session.

2.4.11 Default breakpoints

There are a number of default breakpoints that stop your program when particular conditions are met. You can enable/disable these while your program is running using *Control* ▸ *Default Breakpoints*.

- **Stop at exit/_exit** (Disabled by default)

  When enabled, your program is paused as it is about to end under normal exit conditions. Your program pauses both before and after any exit handlers have been executed.

- **Stop at abort/fatal MPI Error** (Enabled by default)

  When enabled, your program is paused as it about to end after an error has been triggered. This includes MPI and non-MPI errors.

- **Stop on throw (C++ and Python exceptions)** (Disabled by default)

  When enabled, your program is paused whenever an exception is thrown (regardless of whether or not it will be caught). Due to the nature of 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.

  The raised Python exception is displayed as `allinea_exception` in the **Locals** view.

- **Stop on catch (C++ exceptions)** (Disabled by default)

  As for **Stop on throw**, but triggered when your program catches a thrown exception. Again, you may have trouble stepping your program.

- **Stop on unhandled Python exception** (Enabled by default)

  When enabled, your Python program is paused when an exception is raised but never caught by your program.

  You can inspect the stack, including locals, as if you would have stopped the program when the exception was raised. The raised Python exception is displayed as `allinea_exception` in the **Locals** view.

  Your program will terminate if you continue the program.

- **Stop at fork**

  Your program stops when 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 session, and can be debugged along with the original process.

- **Stop at exec**

  When your program calls the exec system call, your program 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.

2.4.12 Synchronize processes

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

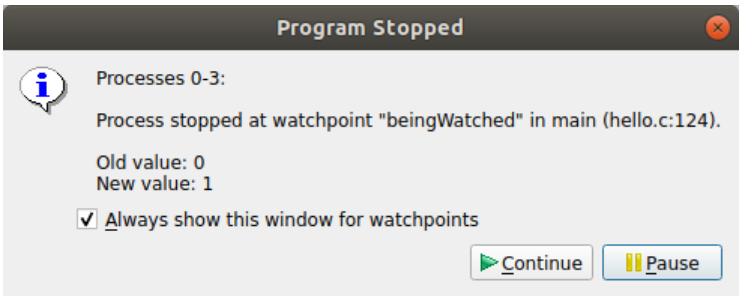
Note: Breakpoints are ignored while the groups are synchronized, but they are not removed.

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

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

2.4.13 Set watchpoints

A watchpoint is a variable or expression that is monitored, so when it is changed or accessed the program is paused.



To add a watchpoint:

- Right-click on the *Watchpoints* tab and select *Add Watchpoint*.
- Right-click over a variable in the **Source Code viewer** then select *Add Watchpoint*.
- Add watchpoints automatically by dragging a variable from the *Local Variables*, *Current Line*, or *Evaluate* views in to the *Watchpoints* tab.

When you add a watchpoint, the *Add Watchpoint* dialog is displayed. You can use this dialog to apply restrictions to the watchpoint:

- *Process Group* restricts the watchpoint to the chosen process group (see *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.

Unlike breakpoints, watchpoints are not displayed in the **Source Code viewer**.

The automatic watchpoints are write-only by default.

A watchpoint is automatically removed when the target variable goes out of scope. If you are watching the value pointed to by a variable, that is, `*p`, you might 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* tab and selecting *Pin to address*. 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.

2.4.14 Tracepoints

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

Tracepoints can be particularly useful in these situations, for example:

- Recording entry values in a function that is called many times, but crashes only occasionally. If you set a tracepoint it is 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 *Use a plugin*.
- Observing progress of a program and variation of values across processes without having to interrupt the program.

2.4.14.1 Set tracepoints

To add a tracepoint:

- Right-click on a line in the **Source Code viewer** and select *Add Tracepoint*.
- Right-click in the *Tracepoints* tab and select *Add Tracepoint*.

When 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 if they are placed in locations likely to generate a lot of passing. For example, if a tracepoint is placed inside a loop with N iterations, then N separate tracepoint passes will be recorded.

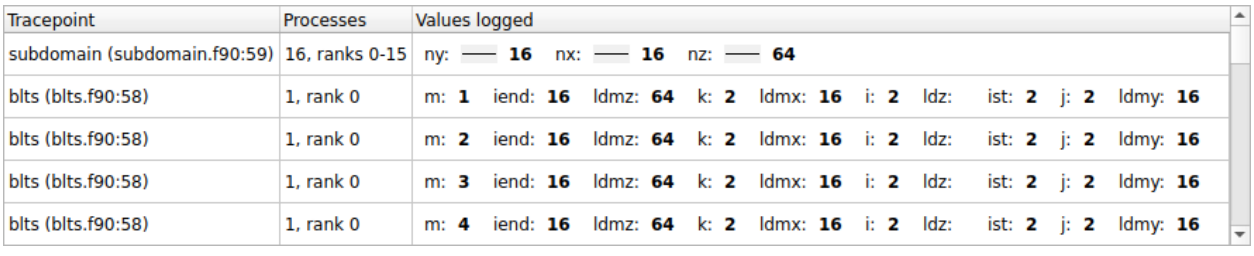
There is an attempt to merge such data in a scalable manner, but when alike tracepoints are passed in order between processes, where process behavior is likely to be divergent and unmergeable, then a considerable load would result.

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 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 you will see a slowdown in your program.

2.4.14.2 Tracepoint output

The output from tracepoints can be viewed on the *Tracepoint Output* tab.



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, 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, this will be shown as one passing of both process 0 and process 1, with no ordering inferred.

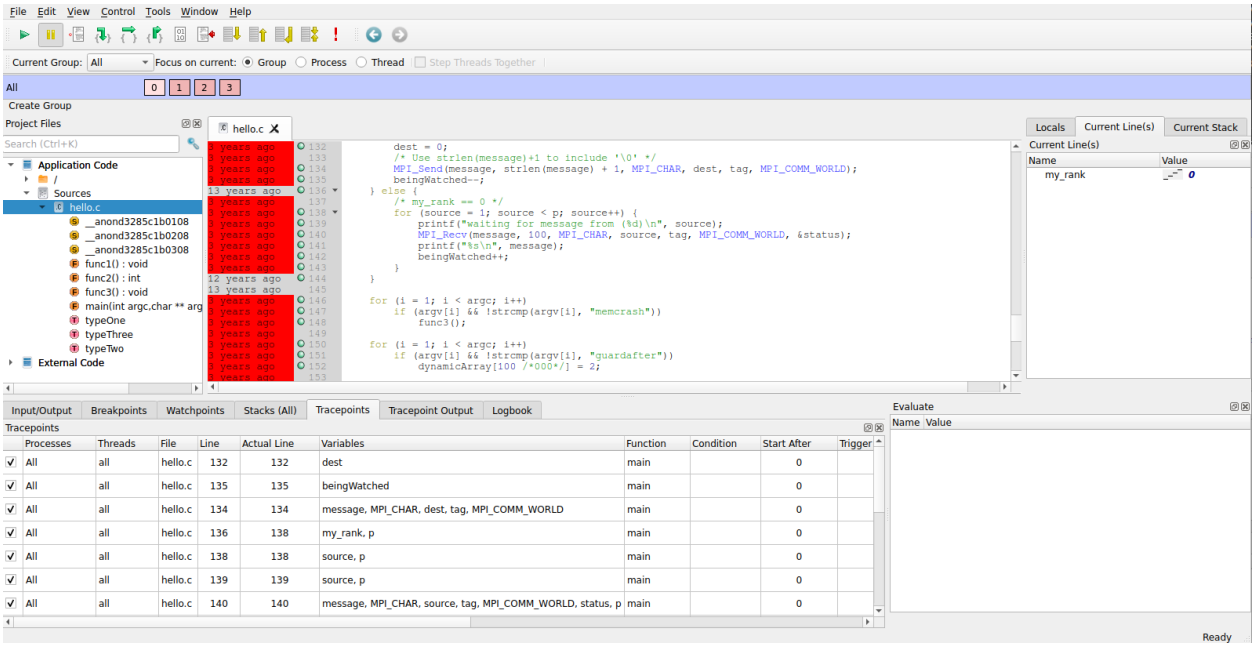
Sequential consistency is preserved during merging, so for any process, the sequence of tracepoints for that process will be in order.

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

For more detailed analysis, export the tracepoints by right-clicking and choosing *Export*. An HTML tracepoint log will be written using the same format as in offline mode.

2.4.15 Version control breakpoints and tracepoints

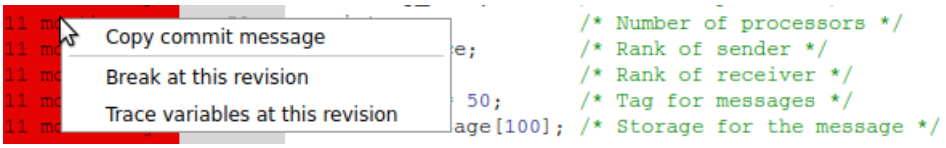
Version control breakpoint and tracepoint insertion allows you to quickly record the state of the parts of the target program that were last modified in a particular revision. The resulting tracepoint output can be viewed in the *Tracepoint Output* tab or the *Logbook* tab, and can be exported or saved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



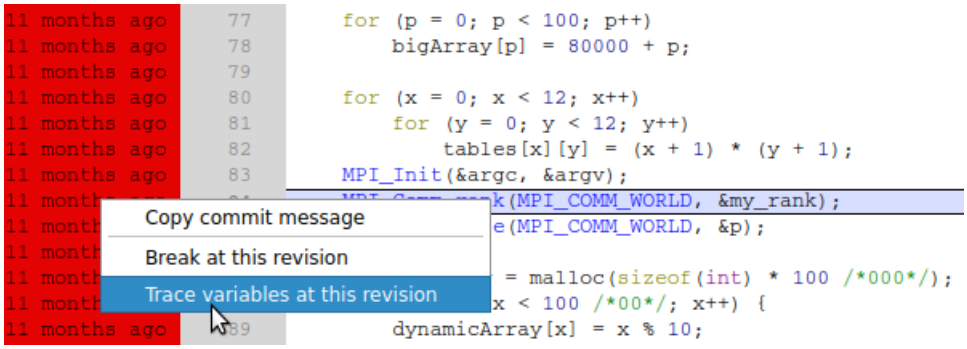
Processes	Threads	File	Line	Actual Line	Variables	Function	Condition	Start After	Trigger
All	all	hello.c	132	132	dest	main		0	
All	all	hello.c	135	135	beingWatched	main		0	
All	all	hello.c	134	134	message, MPI_CHAR, dest, tag, MPI_COMM_WORLD	main		0	
All	all	hello.c	136	138	my_rank, p	main		0	
All	all	hello.c	138	138	source, p	main		0	
All	all	hello.c	139	139	source, p	main		0	
All	all	hello.c	140	140	message, MPI_CHAR, source, tag, MPI_COMM_WORLD, status, p	main		0	

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

In interactive mode, enable *View* ▸ *Version Control Information*. The annotation column is displayed in the **Source Code viewer** for files that are tracked by a supported version control system.



To find all the source files, detect the variables on the lines modified in the revision, and insert tracepoints (pending if necessary), right-click a line last modified by the revision of interest and choose *Trace variables at this revision*.



A progress dialog may be displayed for lengthy tasks.

You can double-click on the tracepoints and tracepoint output in the *Tracepoints*, *Tracepoint Output*, and *Logbook* tabs during a session to jump to the corresponding line of code in the **Source Code viewer**.

In offline mode, supply the additional argument `--trace-changes` to apply 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 `FORGE_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*.

2.4.16 Examine the stack frame

The stack back trace for the current process and thread are displayed on the **Current Stack** tab of the **Variables** window.

When you select a stack frame you jump to that position in the code, if it is available, and the local variables for that frame will be displayed. The toolbar can also be used to step up or down the stack, or jump straight to the bottom-most frame.

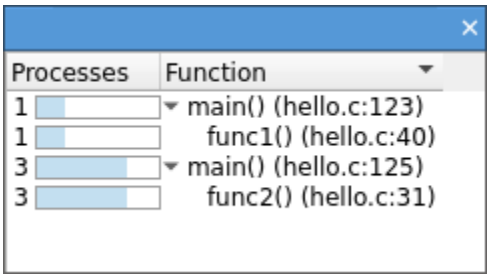
2.4.17 Align stacks

Click **Align stacks** (`Ctrl+Shift+A`) to highlight the same stack-frame depth of the specific thread number across all processes.

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.

2.4.18 View stacks in parallel

You can use the **Parallel Stack View** to see the status of your program in one view.



To find out the location of a group's processes, click on the group. The **Parallel Stack View** displays 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 branches, 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 to view a list of the processes at that location in a popup. 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, labeled by the function's own name.

The **Parallel Stack View** can be used to create groups, to display and select large numbers of processes based on their location in your code (this is invaluable when dealing with moderate to large numbers of processes), and watch what happens as you step processes through your code.

The **Parallel Stack View** takes over much of the work of the *Current 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 (typically using `-g`), extra information is added, displaying the exact source file and line number that your code is on.

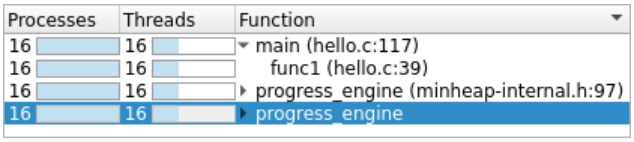
Any functions without debug information are grayed-out and are not shown by default. Functions without debug information are typically library calls or memory allocation subroutines and are not generally of interest. To see the entire list of functions, right-click on one and choose *Show Children*.

You can click on any function to select it as the 'current' function in Linaro DDT. If it was compiled with debug information, its source code is displayed 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:

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

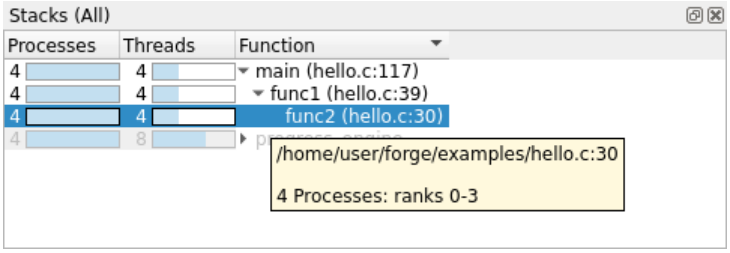
The function that is currently displayed and being used for the variable views is highlighted in dark blue. If you click on another function in the **Parallel Stack View** this selects another frame for the source code and variable views. It also updates the **Stack** display, since these two controls are complementary. If the processes are at several different locations, only the location of the current process is displayed in dark blue. The locations of the other processes are displayed in light blue:



In the example above, the processes of the program are at two different locations. One process is in the `main` function, at line 147 of `hello.c`. The other 15 processes are inside a function called `func1`, at line 39 of `hello.c`. To see the line of source code a function corresponds to, and display any local variable in that stack frame, click on the function.

There are two optional columns in the **Parallel Stack View**. **Processes** shows the number of processes at each location. **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 program, each process has one thread and these two columns will show identical information.

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



Linaro 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.

To create a new process group that contains only the processes sharing that location in code, right-click on a function in the combined call tree and choose *New Group*. By default the name of the function is used for the group, or the name of the function with the file and line number if it is necessary to distinguish the group further.

The contents of the **Parallel Stack View** can be exported in CSV or XML format via the Right-click context menu, or copied to clipboard.

2.4.19 Browse 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.

Lines of the source code are highlighted 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, lines containing that process are highlighted in the group color, while other processes from that group are highlighted in gray.

Lines of code that are on the stack are also highlighted, 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.

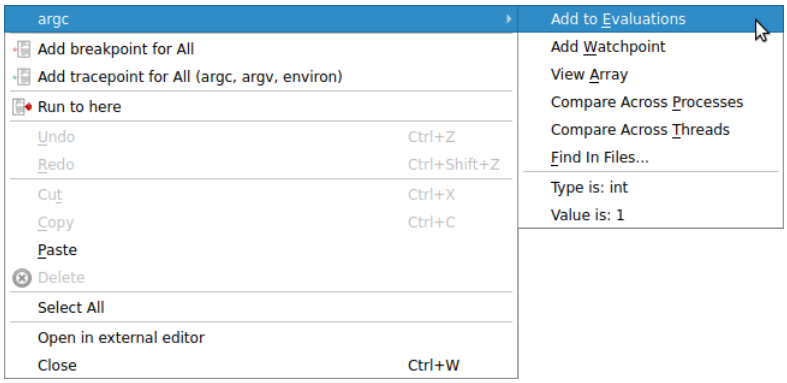
Focus on Process
: A list of the processes from the current group that are on this line, along with the threads from the current process on the selected line.

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

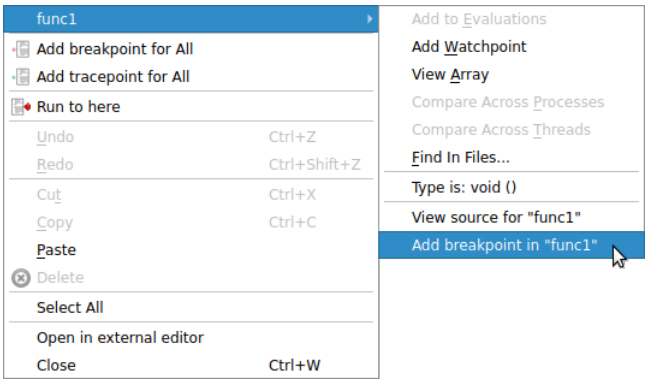
  The tooltip distinguishes between processes and threads that are currently executing that line, and ones that are on the stack by grouping them under the headings *On the stack* and *On this line*.

Variables and Functions
: Right-click on a variable or function name in the **Source Code viewer** 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 View** (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 section.



  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.

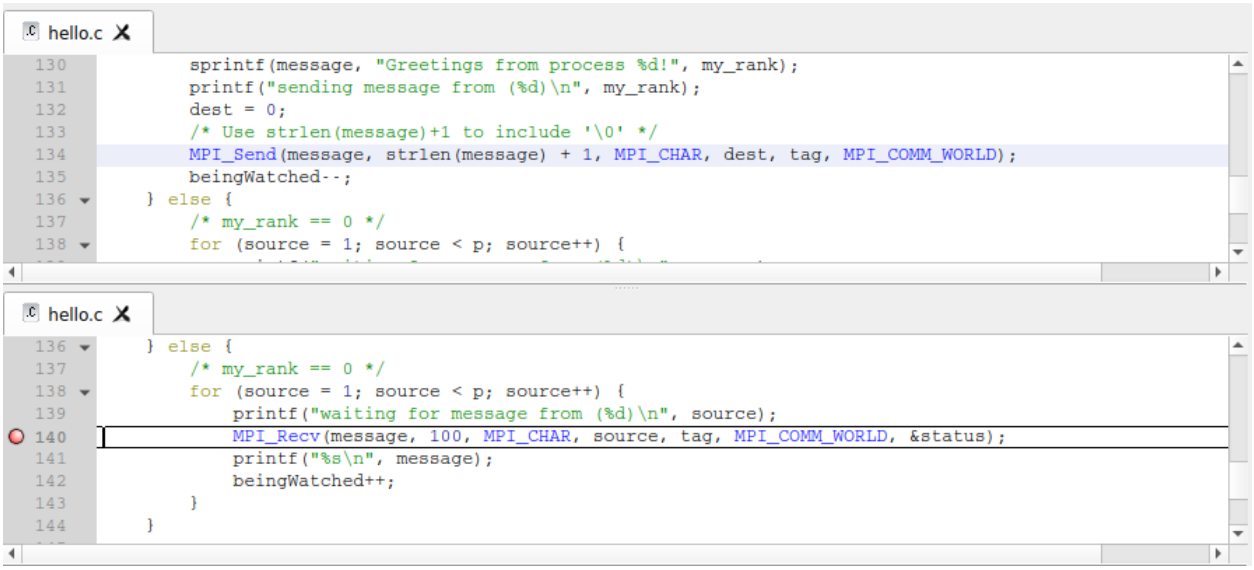


2.4.20 View multiple files simultaneously

Occasionally it may be useful to view two source files at the same time. For example, if you are tracking two different processes.

To view two files simultaneously, right-click in the **Source Code view** to split the view. This displays a second panel beneath the first panel. When viewing multiple files, the currently 'active' panel displays the file. Click on one of the panels to make it active.

To remove the split view and return to viewing one source file, right-click in the **Source Code viewer** and clear the split view option.



2.4.21 Signal handling

By default a process is stopped if it encounters one of the standard signals. 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 *Memory debugging* feature might 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

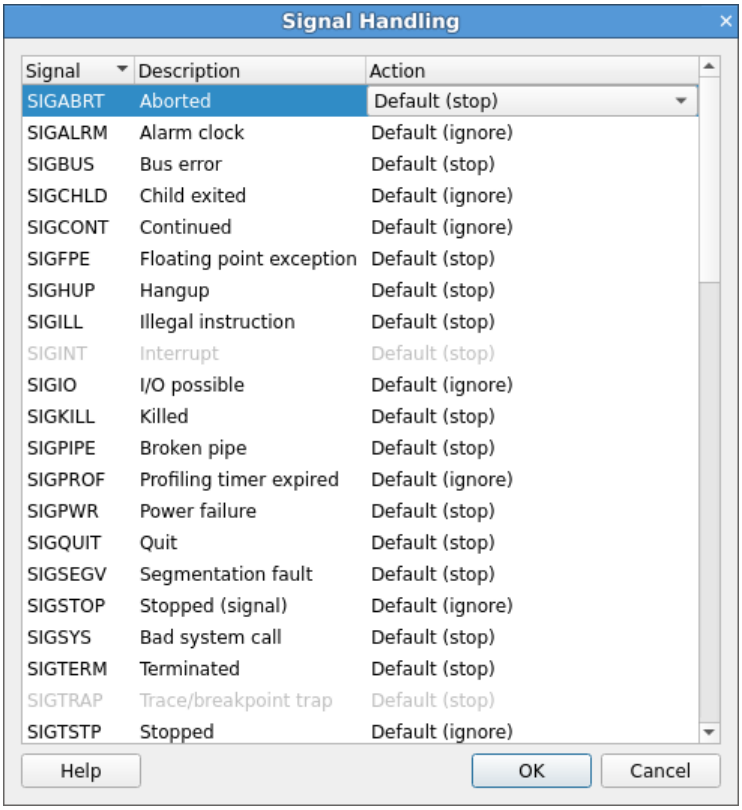
  A broken pipe has been detected while writing.

- `SIGILL` - Illegal Instruction

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

2.4.21.1 Custom signal handling (signal dispositions)

You can change the way that individual signals are handled using the *Signal Handling* dialog. To open the dialog, select *Control* ▸ *Signal Handling*.



To stop the process when it encounters a signal, set the *Action* to *Stop*.

To let the process receive the signal and continue playing without being stopped by the debugger, set the *Action* to *Ignore*.

2.4.21.2 Send signals

The *Send Signal* window lets you send a signal to the debugged processes. To send a signal, select *Control* ▸ *Send Signal*, select the signal you want to send, then click *Send to process*.

2.5 Variables and data

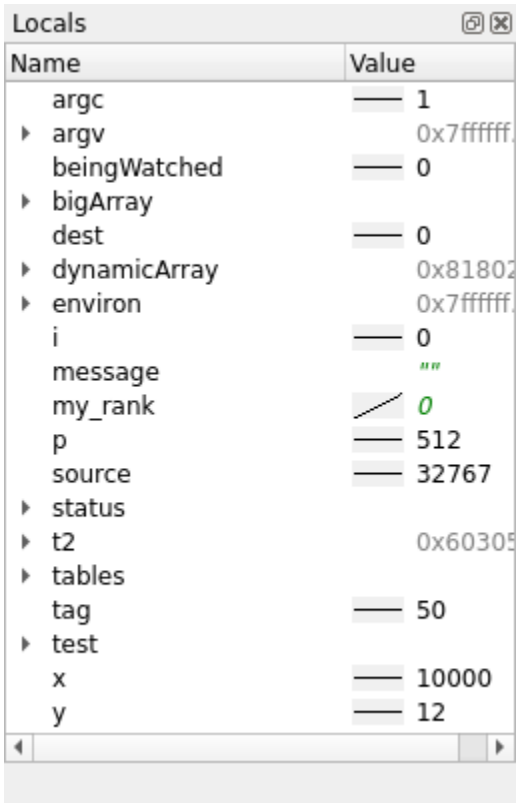
This section describes the **Variables** window.

2.5.1 Variables window

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. The *Current Line(s)* tab displays all the variables referenced on the currently selected lines.

Note: Several compilers and libraries (such as Cray Fortran, and OpenMP) generate extra code, including variables that are visible in Linaro DDT's windows.

The right-click menu in the *Variables* window enables you to edit values, change the display base, compare data across processes and threads, and choose whether the fields in structures (classes or derived types) are displayed alphabetically by element name or not (which is useful for when structures have many different fields).



2.5.2 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 *Focus control*.

When focused on process groups, 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, clicking on a sparkline displays the *Cross-Process Comparison* window for closer analysis. Otherwise, clicking on a sparkline displays the *Cross-Thread Comparison* window.

2.5.3 Current line

You can select a single line by clicking on it in the **Source Code viewer**. You can select 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 tab into the *Evaluate* window. It is then evaluated in whichever stack frame, thread, or process you select.

2.5.4 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. If this is a problem, it is best to conceal this tab underneath the *Current Line(s)* tab so it will not update after every step.

Note: Variables defined within common blocks might not appear in the *Locals* tab with some compilers. This is because they are considered to be global variables when defined in a common memory space.

The *Locals* tab 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.

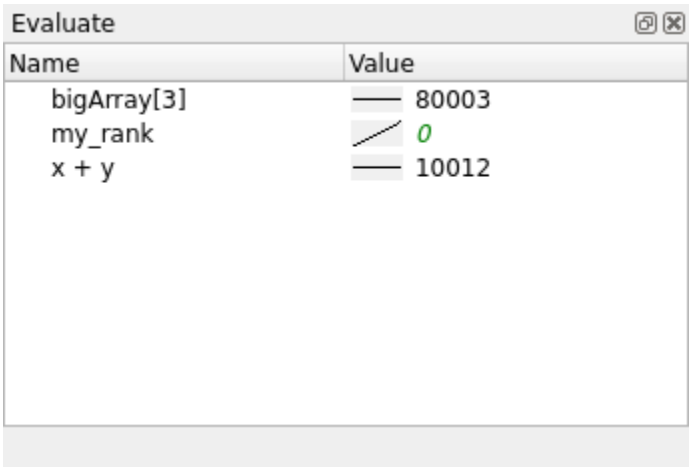
When stepping or switching processes, if the value of a variable is different from the previous position or process, it is highlighted in blue.

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

2.5.5 Arbitrary expressions and global variables

The global variables and arbitrary expressions are not displayed with the local variables. You can click on the line in the **Source Code viewer** that contains a reference to the global variable and use the *Current Line(s)* tab.

Alternatively, the *Evaluate* window can be used to view the value of any arbitrary expression.



Right-click on the *Evaluate* window, click *Add Expression*, then 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: Linaro DDT does not apply the usual rules of precedence to logical Fortran expressions, such as `x .ge. 32 .and. x .le. 45.`. You need to bracket such expressions thoroughly, for example: `(x .ge. 32) .and. (x .le. 45)`.

Note: Although the Fortran syntax allows you to use keywords as variable names, Linaro DDT is not able to evaluate such variables on most platforms. Contact Forge Support if this issue affects you.

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

2.5.5.1 Fortran intrinsics

The following Fortran intrinsics are supported by the default GNU debugger included with Linaro 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.

2.5.5.2 Changing the language of an expression

By default, expressions in the *Evaluate* window, *Locals* tab, and *Current Line(s)* tab are evaluated in the language of the current stack frame. This might not always be appropriate. For example, a pointer to user-defined structure might be passed as value within a Fortran section of code, and you might want to view the fields of the C structure. Alternatively, you might want to view a global value in a C++ class while your process is in a Fortran subroutine.

You can change the language of an expression by right-clicking on the expression, choosing *Change Type/Language*, and selecting the appropriate language for the expression. To restore the default behavior, change this back to *Auto*.

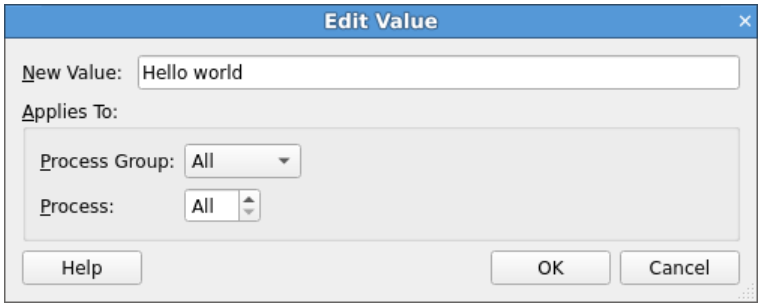
2.5.5.3 Macros and #defined constants

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.

For the GNU compiler, when you add the `-g3` option to the command line options this generates extra definition information which will then be displayed.

2.5.6 Edit variables

You can edit the values of simple types such as scalars, pointers, and c-strings. To edit a value, right-click the value in a variable view and select *Edit Value*. Enter the new value in the *Edit Value* dialog then click *Ok*.



2.5.7 Help with Fortran modules

An executable that contains Fortran modules presents a special set of problems:

- If there are many modules, and they all contain 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 be difficult.
- When the *Locals* or *Current Line(s)* tabs (in 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 these problems, use the *Fortran Modules* tab in the *Project Navigator* window.

When a session starts, 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 in the *Fortran Modules* tab of the *Project Navigator* window.

To expand a module, click on the + symbol to the left of the module name. This displays a list of member procedures, member variables, and the current values of those member variables.

When you click on one of the procedure names the **Source Code viewer** jumps 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 ()`.

When you click on one of the displayed variable names, the type of that variable is 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 ::.

If you right-click in the *Fortran Modules* tab, you can use the context menu to send the variable to the *Evaluate* window, *Multi-Dimensional Array Viewer*, or *Cross-Process Comparison View* window.

There are some limitations to the information displayed in the *Fortran Modules* tab:

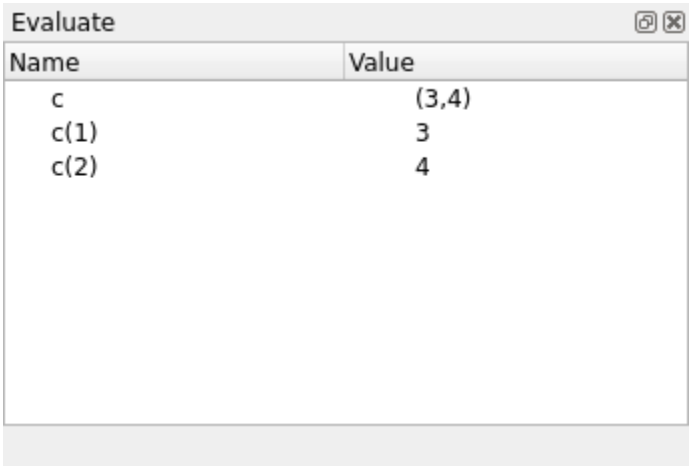
- The *Fortran Modules* tab is not displayed if the underlying debugger does not support the retrieval and manipulation of Fortran module data.
- The *Fortran Modules* tab displays an empty module list if the debug data for the Fortran modules is not present or is not in a recognized format.
- The debug data compiled into an executable does not include any indication of the module `USE` hierarchy. For example, if module A USEs module B, the inherited members of module B are not shown under the data displayed for module A. Consequently, the *Fortran Modules* tab shows the module `USE` hierarchy in a flattened form, one level deep.

2.5.8 View complex numbers in Fortran

When working with complex numbers, you might want 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 *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.



2.5.9 C++ STL support

Linaro DDT uses pretty printers for the GNU C++ STL implementation (version 4.7 and later), 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 want to see the underlying implementation of an STL type, you can disable pretty printing using the environment variable setting `FORGE_DISABLE_PRETTY_PRINT=1`.

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

2.5.10 Custom pretty printers

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

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, copy it to `~/.allinea/gdb`.

> **Warning:** Only install pretty printers from trusted third-party sources.

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

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

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

Start Linaro DDT with the example program:

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

When the program has started, right-click on line 20 then use the *Run to here* command. The internal variable of `myFruit` is displayed on the *Locals* tab.

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

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

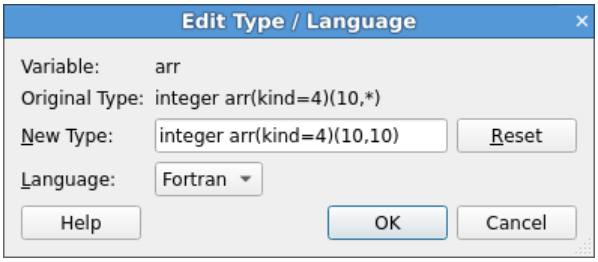
Re-run the program and run to line 20, as before. Click on the *Locals* tab and you can see that now, instead of the internal variable `myFruit`, the type of fruit is displayed instead.

2.5.11 View array data

2.5.11.1 Fortran Array

Fortran users might find that it is impossible to view the upper bounds of an array. This is due to a lack of information from the compiler. In these circumstances, the array is displayed 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 input the size of the array, right-click on the array and select *Edit Type*. In the *Edit Type* window, enter the real type of the array in *New Type*.

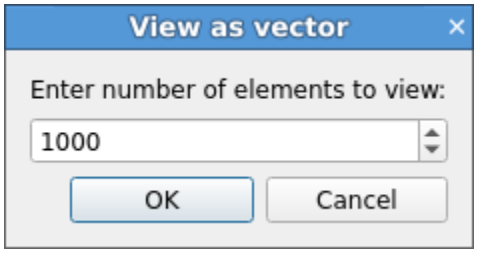


Alternatively, the *Multi-Dimensional Array Viewer* can be used to view the entire array.

2.5.11.2 Flexible Array

C and many C++ language extensions support the use of flexible array members in structs. You may find that these arrays are not displayed fully within the variable view. This is due to a lack of information from the compiler.

To view the array you can use the **View As Vector** feature. Right-click the array and select *View ▸ Vector (C/C++ only)*. Then enter the size of the flexible array.



Alternatively, the *Multi-Dimensional Array Viewer* can also be used.

2.5.12 UPC support

Linaro DDT supports the Cray UPC compiler.

When debugging UPC applications, processes are identified as **UPC Threads**. This is purely a terminology change for consistency with the UPC language terminology. UPC Threads have identical behavior to that of separate processes. For example, groups, process control, and cross-process data comparison apply across UPC Threads.

All other components are identical to debugging any multi-process code.

2.5.13 Change data values

To set the value of an expression, right-click in the *Evaluate* window and select *Edit Value*. This enables 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.

2.5.14 View numbers in different bases

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

2.5.15 Examine pointers

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

In the *Evaluate* window, you can also use the *View As –> Vector*, *Get Address*, and *Dereference Pointer* menu items:

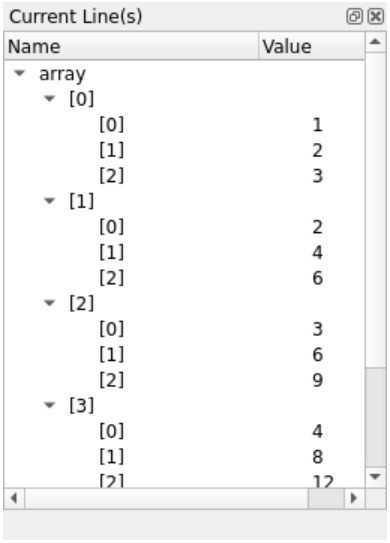
- *Dereference Pointer* wraps the expression in `*()`.
- *Get Address* strips a single layer of `*()` from the expression (if one exists).
- Both *Get Address* and *Dereference Pointer* only support raw pointers and not other pointer implementations, such as, C++11 smart pointers.

See also *Multi-Dimensional Array Viewer (MDA)*.

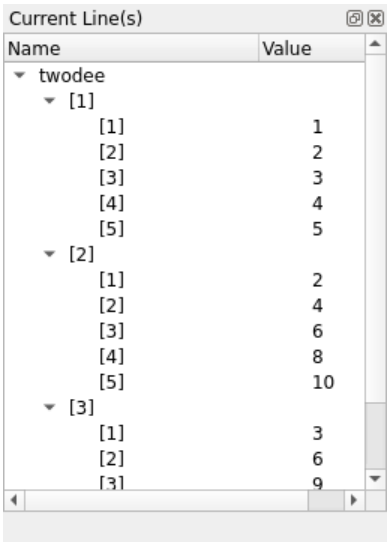
2.5.16 Multi-dimensional arrays in the Variable View

When you view a multi-dimensional array in 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.



In Fortran the opposite will be seen with arrays being displayed from right to left as you read it so `x`, `y`, `z` will have `z` as the first column with `y` under each `z` cell.

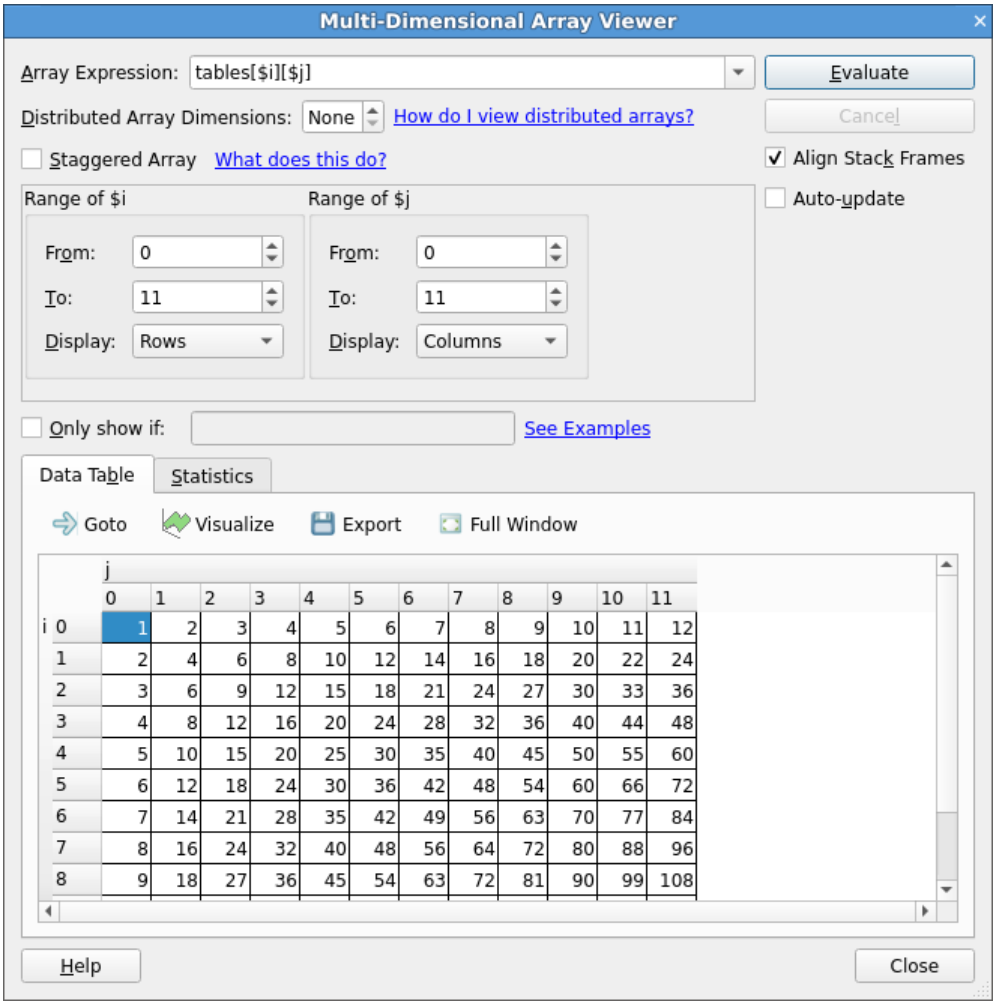


The first thousand elements in an array are shown in the *Locals* or *Current Line(s)* tabs. Larger arrays are truncated. Elements after the first thousand can be viewed by evaluating an expression or by using the **Multi-Dimensional Array Viewer**.

2.5.17 Multi-Dimensional Array Viewer (MDA)

You can use the *Multi-Dimensional Array Viewer* (MDA) to view multi-dimensional arrays.

To open the *Multi-Dimensional Array Viewer*, right-click on a variable in the **Source Code viewer**, *Locals* tab, *Current Line(s)* tab or *Evaluate* window and select *View Array (MDA)*. You can also open the MDA directly by selecting *View* ▸ *Multi-Dimensional Array Viewer*.



i \ j	0	1	2	3	4	5	6	7	8	9	10	11
0	1	2	3	4	5	6	7	8	9	10	11	12
1	2	4	6	8	10	12	14	16	18	20	22	24
2	3	6	9	12	15	18	21	24	27	30	33	36
3	4	8	12	16	20	24	28	32	36	40	44	48
4	5	10	15	20	25	30	35	40	45	50	55	60
5	6	12	18	24	30	36	42	48	54	60	66	72
6	7	14	21	28	35	42	49	56	63	70	77	84
7	8	16	24	32	40	48	56	64	72	80	88	96
8	9	18	27	36	45	54	63	72	81	90	99	108

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

Use *Full Window* to expand the table of values (and hide the settings at the top of the window). This enables you to make full use of your screen space. Click *Full Window* again to view the settings.

2.5.17.1 Array expression

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

The range of each metavariable is defined in the fields below the expression, for example *Range of $i*. The *Array Expression* is evaluated for each combination of `$i`, `$j`, and so on. The results are shown in the *Data Table*. You can 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, you can also specify these ranges as *expressions* in terms of program variables. These expressions are then evaluated in the debugger. To allow the entry of these expressions, select the *Staggered Array* checkbox. This converts all the range entry fields from spin boxes to line edits allowing the entry of free-form text.

The metavariables can 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 cannot be reordered, they must always be the most major dimensions.

2.5.17.2 Filter by value

You can configure the *Data Table* to only show elements that fit certain criteria, for example elements that are zero.

If the *Only show if* checkbox is selected, only elements that match the boolean expression in the adjacent field 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 can be used here, including references to variables in scope and function calls. We recommend that you avoid functions with side effects as these will be evaluated many times.

2.5.17.3 Distributed arrays

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

The *Multi-Dimensional Array Viewer* can display certain types of distributed arrays, namely UPC shared arrays (for supported UPC implementations), and general arrays where the distributed dimensions are the most major, 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. Right-click on the array variable and select *View Array (MDA)*.

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

If the array is distributed over all processes in your job, select the *All* group when you right-click on the local array variable in the **Source Code viewer**, *Locals* tab, *Current Line(s)* tab or *Evaluate* window.

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

Enter the number of *Distributed Array Dimensions*. A new subscript metavariable (such as `$p`, `$q`) will be automatically added for each distributed dimension.

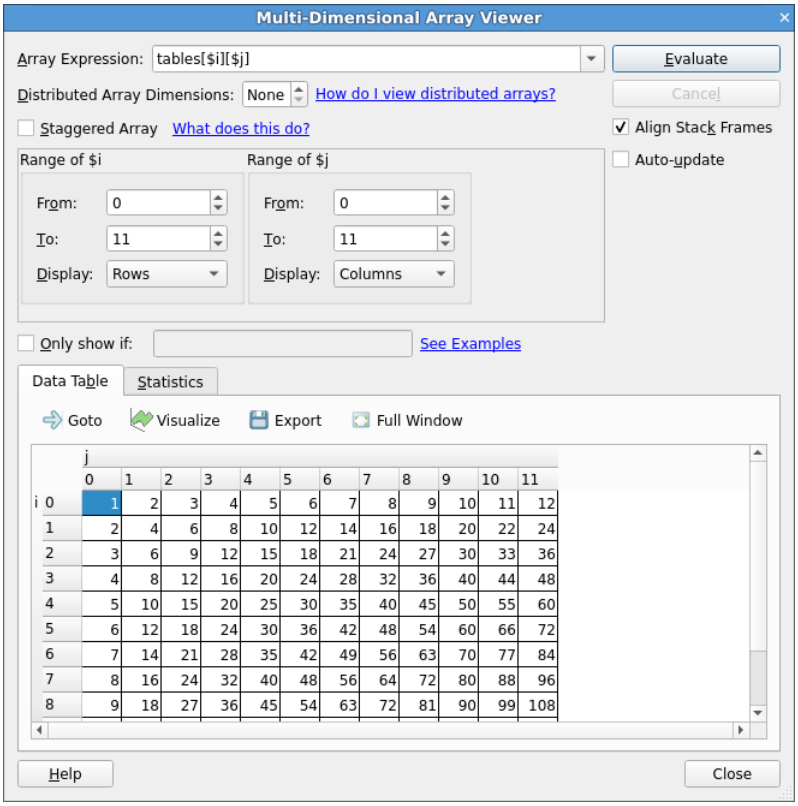
Enter the ranges of the distributed dimensions so that the product is equal to the number of processes in the current process group, then click *Evaluate*.

2.5.17.4 Advanced: how arrays are laid out in the data table

The *Data Table* is two-dimensional, but the *Multi-Dimensional Array Viewer* can 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`.

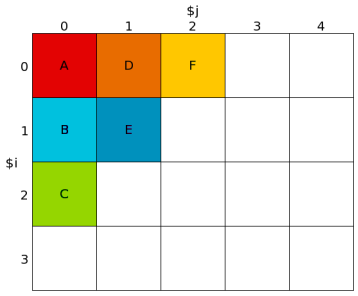
The figure below corresponds to the expression `myArray($i, $j)` with `$i = 0..3` and `$j = 0..4`.



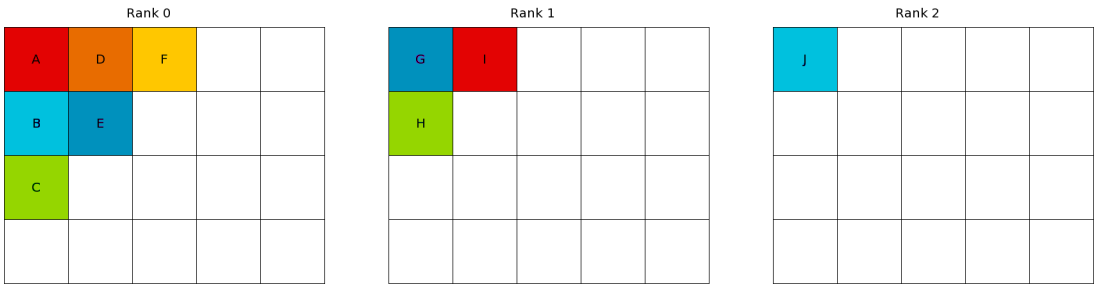
i \ j	0	1	2	3	4	5	6	7	8	9	10	11
0	1	2	3	4	5	6	7	8	9	10	11	12
1	2	4	6	8	10	12	14	16	18	20	22	24
2	3	6	9	12	15	18	21	24	27	30	33	36
3	4	8	12	16	20	24	28	32	36	40	44	48
4	5	10	15	20	25	30	35	40	45	50	55	60
5	6	12	18	24	30	36	42	48	54	60	66	72
6	7	14	21	28	35	42	49	56	63	70	77	84
7	8	16	24	32	40	48	56	64	72	80	88	96
8	9	18	27	36	45	54	63	72	81	90	99	108

For 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.

This example shows the local array `myArray($i, $j)` with `$i = 0..3` and `$j = 0..4` on ranks 0-2.



This figure shows a three-dimensional distributed array comprised of the local array `myArray($i, $j)`, with `$i = 0..3`, and `$j = 0..4` on ranks 0-2, and with `$p` the distributed dimension:

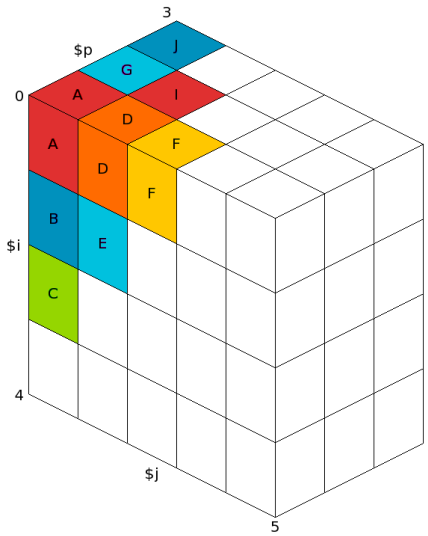


This cube is projected (just like 3D projection) onto the two dimensional *Data Table*. Dimensions marked *Display* as *Rows* are shown in rows, and dimensions marked *Display* as *Columns* are shown in columns, as you would expect.

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*. This figure shows a three-dimensional distributed array comprised of the local array `myArray($i, $j)` with `$i = 0..3` and `$j = 0..4` on ranks 0-2. It is projected onto the Data Table with `$p` (the distributed dimension), `$j` displays as Columns, and `$i` displays as Rows:



2.5.17.5 Auto update

If you select the *Auto update* checkbox, the *Data Table* will automatically update as you switch between processes/threads and step through the code.

2.5.17.6 Compare elements across processes

When viewing an array in the *Data Table*, double-click an element or right-click an element and choose *Compare Element Across Processes* to open the *Cross-Process Comparison View* for the selected element.

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

2.5.17.7 Statistics tab

The *Statistics* tab displays information which might 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`.

2.5.17.8 Export

You can export the contents of the table to a file in the comma-separated values (CSV) or HDF5 format so that it can be plotted or analyzed in your favorite 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 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.

2.5.17.9 Visualization

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

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

After filling the table of the MDA viewer with values, click *Visualize* to open a 3-D view of the surface.

To display surfaces from two or more different processes on the same plot, select another process in the main process group window then click *Evaluate* in the MDA viewer. When the values are ready, click *Visualize* again.

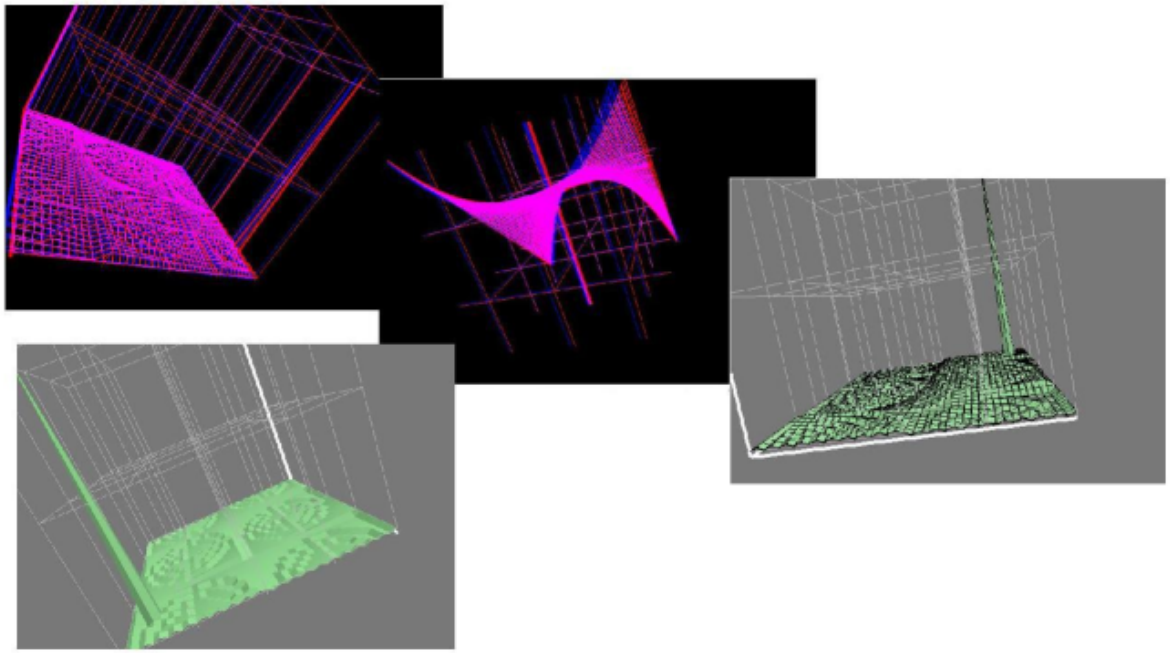
The surfaces displayed on the graph can be hidden and shown using the checkboxes on the right side of the window.

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

Note: Linaro DDT requires OpenGL 2.0+ to run. On Linux, Linaro DDT includes the Mesa 3D library to provide software rasterization, enabling use on machines without OpenGL-capable graphics hardware.



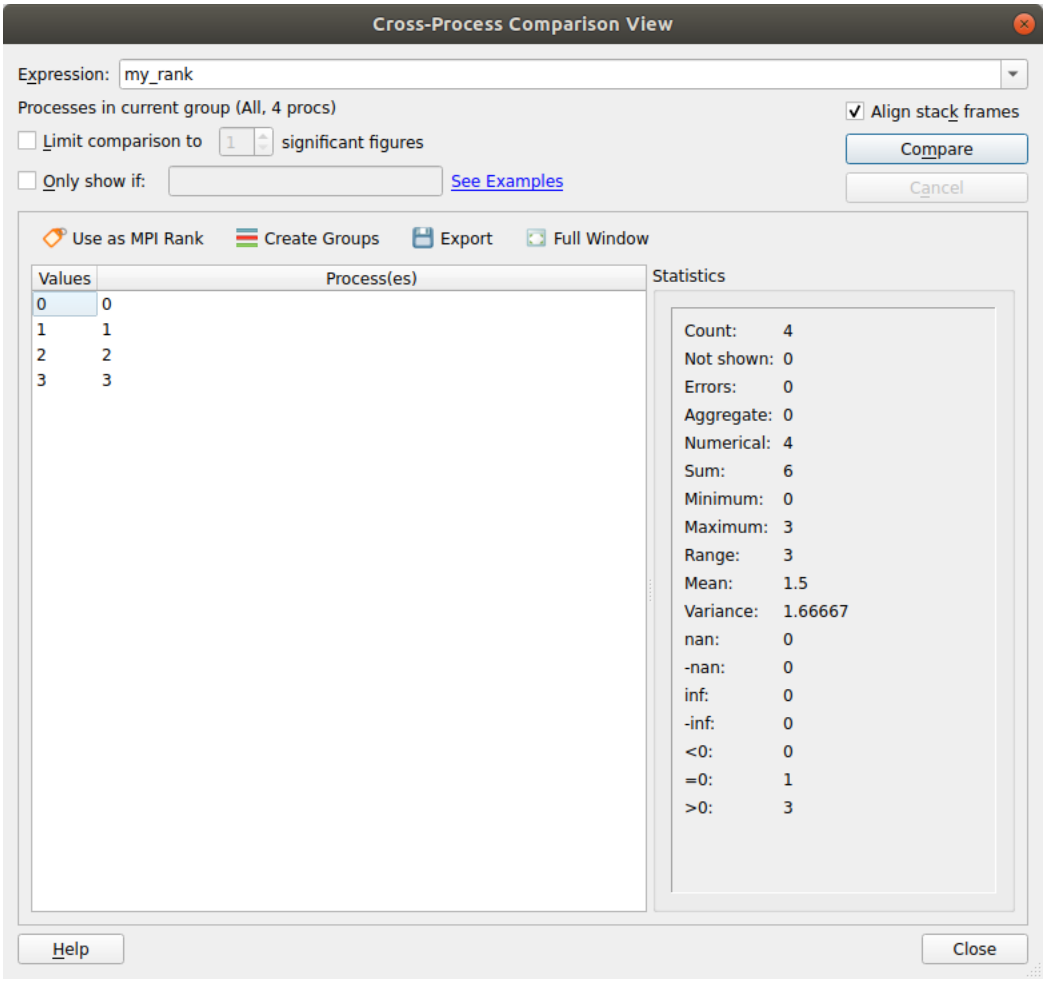
The toolbar and menu have options to configure lighting and other effects, including a function to save an image of the surface as it currently displays.

2.5.18 Cross-process and cross-thread comparison

The **Cross-Process Comparison View** (CPC) and **Cross-Thread Comparison View** (CTC) 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 *Evaluate* window, *Locals* tab, and *Current Line(s)* tab for multi-process sessions.

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



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 using the *Limit comparison to* fields.

If you are comparing across processes, you can turn each of these groupings of processes into a process group by clicking *Create Groups*. 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 current language (that is, C/C++/Fortran).

For threaded applications, when using the CTC, if OpenMP thread IDs can be identified, a third column will 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 are available when using Intel and NVIDIA HPC compilers. However, if these have been removed by compiler optimizations, recompile using"-O0". OpenMP thread IDs can only be obtained from GCC compiled programs if the compiler itself was compiled with TLS enabled. Packaged GCC installs on some supported Linux distributions might not be compiled with TLS enabled.

The display of OpenMP thread IDs is not currently supported when using the Cray compiler.

You can enter a second boolean expression in the **Only show if** field 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 *See Examples* to view examples.

Select *Align stack frames* to ensure that the stack frame of the same depth in every thread (using the CTC) is

examined when comparing the variable value. When using the CPC, *Align stack frames* ensures that the same thread number is also selected when examining variable values. This is very helpful for most programs, but you might want to disable it if different processes or threads run entirely different programs.

The *Use as MPI Rank* button is described in *Assign MPI ranks*.

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

The *Export* button enables 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 that the list of values occupies the full window. This enables you to make full use of your screen space. Click *Full Window* again to display the settings.

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

2.5.19 Assign MPI ranks

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

1. Choose a variable that holds the MPI world rank for each process, or an expression that calculates it.
2. Use *Cross-Process Comparison View* to evaluate the expression across *all* the processes. If the variable is valid, *Use as MPI Rank* will be enabled.
3. Click *Use as MPI Rank*. All of its processes will be relabeled with these new values.

The criteria for a variable or an expression to be valid are:

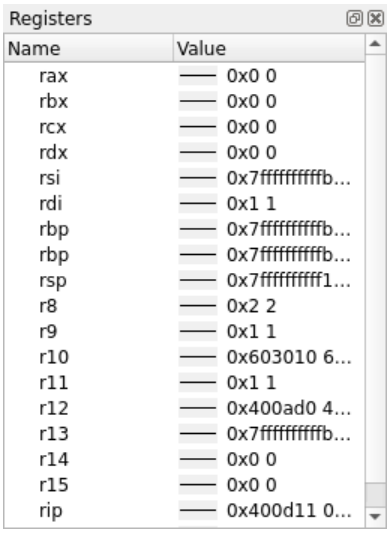
- It must be an integer.
- Every process must have a unique number afterwards.

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.

2.5.20 View registers

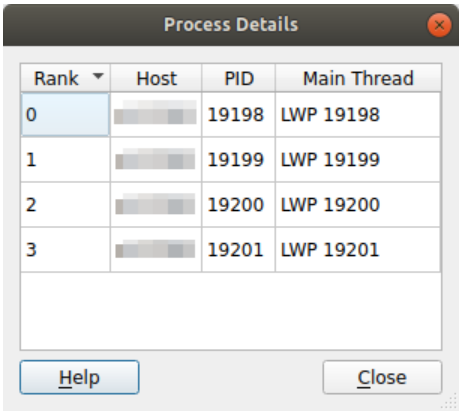
To view the values of machine registers on the currently selected process, select *View* ▸ *Registers*. These values will be updated after each instruction, change in thread, or change in stack frame.

To edit the value of a register either double-click on the register, or right-click and select *Edit Value*.



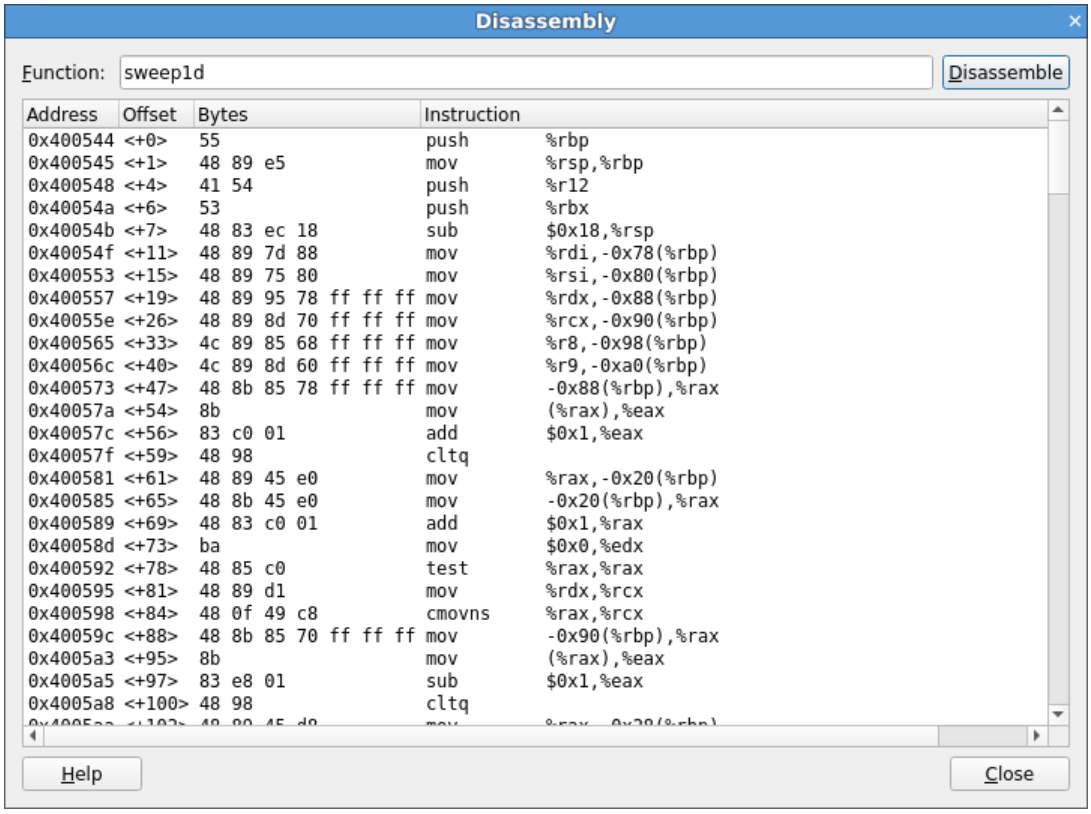
2.5.21 Process details

To view the *Process Details* dialog, select *Tools* ▸ *Process Details*. Details can be sorted by any column, in ascending or descending order.



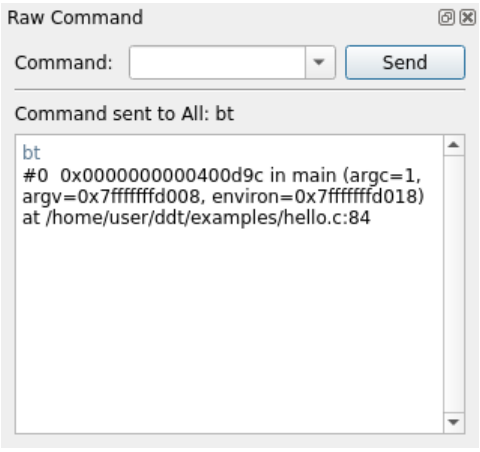
2.5.22 Disassembly

To view the disassembly (assembly instructions) of a function, select *Tools* ▸ *Disassemble*. By default you will see the disassembly of the current function. You can view the disassembly of a different function by typing the name in the *Function* field and clicking *Disassemble*.



2.5.23 Interact directly with the debugger

The *Raw Command* window enables you to send commands directly to the debugger interface. This window bypasses Linaro DDT and its book-keeping. So, if you set a breakpoint in this window, it will not be listed in the *Breakpoints* tab.



Be careful with this window. We recommend that you only use it if the graphical interface does not provide the information or control you require. If you send commands such as `quit` or `kill`, this may cause the interface to stop responding to Linaro DDT.

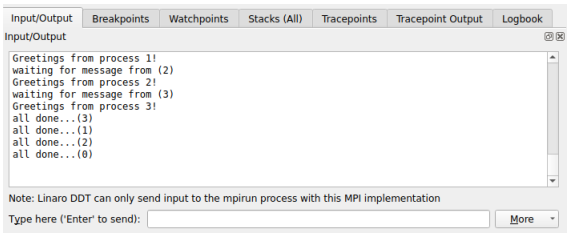
Each command is sent to the current group or process, depending on the current focus. If the current group or process is running, you will be prompted to pause the group or process.

2.6 Program input and output

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

2.6.1 View standard output and error

The output from all processes is collected and can be viewed on the *Input/Output* tab. Standard output and error are both displayed. However, on most MPI implementations, error is not buffered (output is buffered), so might be delayed.



The output can 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.

Note: 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.

2.6.2 Save output

Right-click on the text to either save it to a file or copy a selection to the clipboard.

2.6.3 Send standard input

You can use the *stdin* field in the *Run* window to choose the file to use as the standard input (stdin) for your program. Arguments will be automatically added to mpirun to ensure your input file is used.

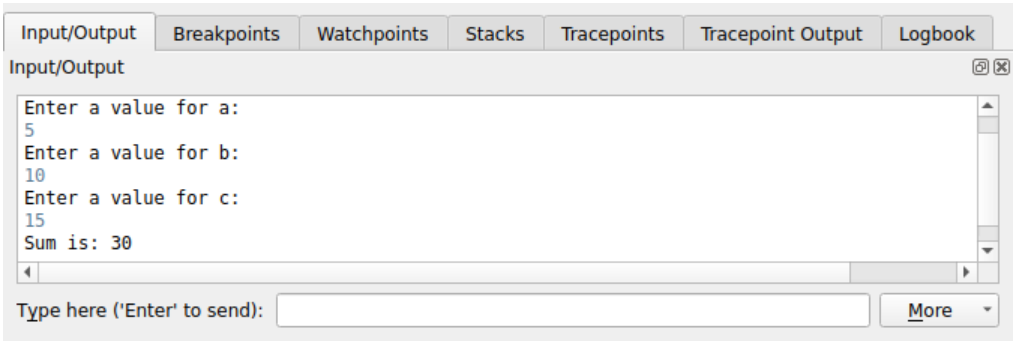
Alternatively, you can enter the arguments directly in the *mpirun Arguments* field. For example, if using MPI directly from the command-line you would normally use an option to the mpirun such as `-stdin filename`, then you can add the same options to the *mpirun Arguments* field when you start your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

It is also possible to enter input during a session. To do this, start your program as normal, then open the *Input/Output* tab. Type the input you want to send.

Click *More* 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.



2.7 Logbook

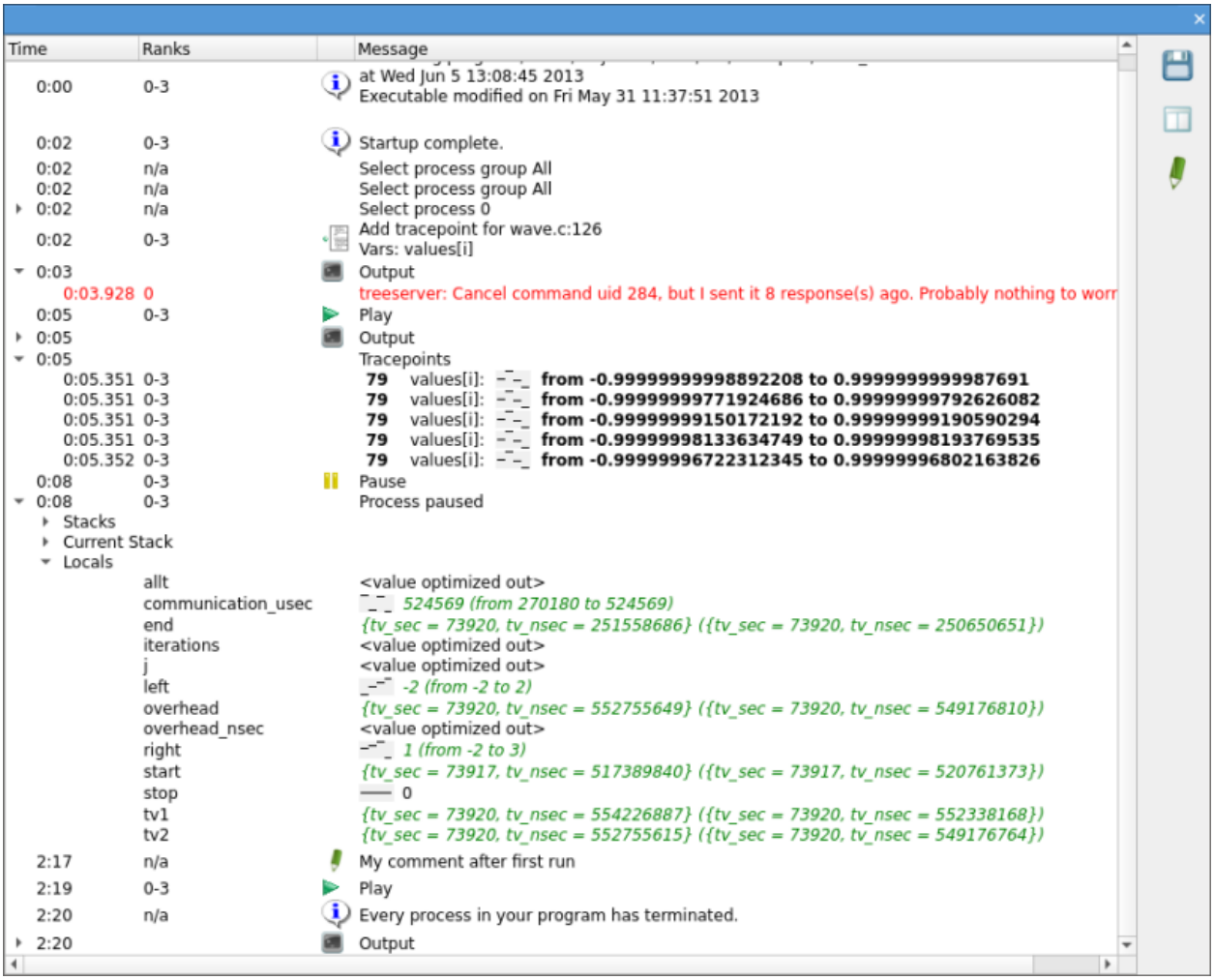
This section describes how to use the logbook.

2.7.1 Usage

The **Logbook** tab is located at the bottom of the main window.

The **Logbook** automatically generates a log of all of your actions. It does not require any additional configuration.

For example, if you add a breakpoint or a tracepoint, or play the program, these actions are logged. For each stop of the program, the reason and location is recorded together with the parallel stacks and local variables for one process.



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

To export the logbook, click the disk icon on the right side of the **Logbook** tab and specify a filename.

You can open previously saved logbooks from the **Tools** menu.

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.

2.7.2 Annotation

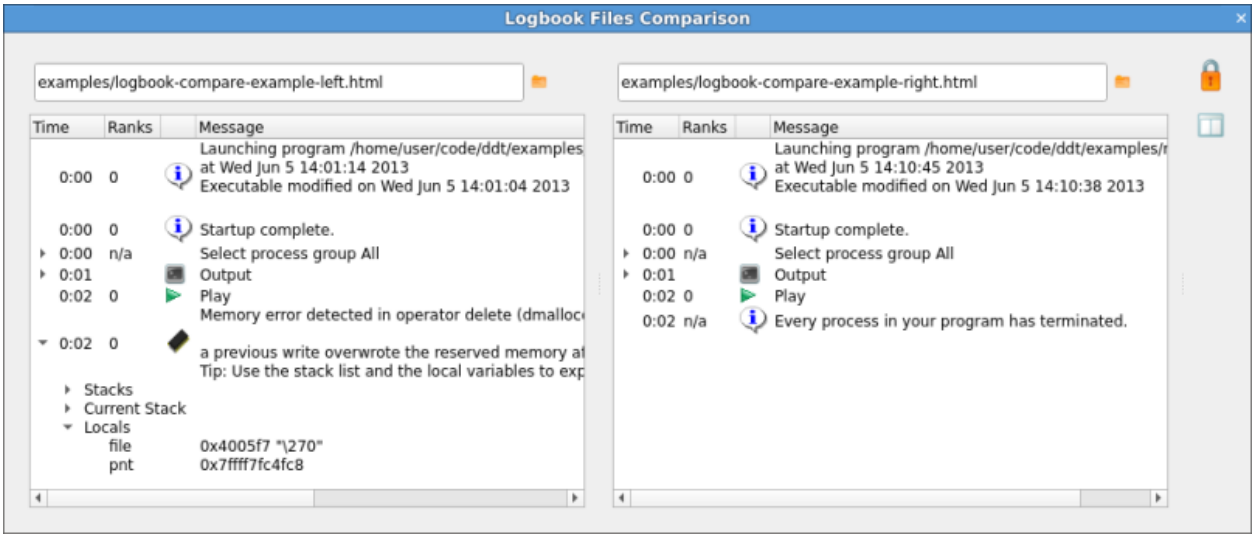
You can add annotations to the logbook using either the pencil icon on the right side of the *Logbook* tab, or by right-clicking the logbook and choosing *Add annotation*.

2.7.3 Logbook comparison

Two logbooks can be compared side-by-side using the **Logbook Files Comparison** window.

To run a comparison, click the 'compare' icon on the right side of the **Logbook** tab. 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 click the 'lock' icon. This fixes the vertical and horizontal scrollbars of the logbooks so that they scroll together. This figure shows the **Logbook Files Comparison** window with tracepoint differences selected.



2.8 Message queues

The **Message Queues** window shows the status of the message buffers of MPI.

You can use it to, for example:

- Show the messages that have been sent by a process but not yet received by the target.
- Detect common errors such as deadlock. This is where all processes are waiting for each other.
- Detect when messages are present that are unexpected, which can correspond to two processes disagreeing about the state of progress through a program.

This feature relies on the MPI implementation supporting it via a debugging support library: the majority of MPIs provide this. Not all implementations support this capability to the same extent, so a variation between the information provided by each implementation is to be expected.

2.8.1 View message queues

To open the *Message Queues* window, select *Tools* ▸ *Message Queues*. This will query the MPI processes for information about the state of the queues.

When the window is open, click *Update* to refresh the current queue information. Note that this will stop all playing processes. A dialog might be displayed while the data is being gathered. You can cancel the request at any time.

The message queue support library from your MPI implementation (if one exists) will be automatically loaded. If it fails to load, an error message will display.

Common reasons for failing 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 and MVAPICH must be configured with the `--enable-debuginfo` and `--enable-shared` 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 *Intel MPI* for details.

  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 `FORGE_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 might require you to specify the installation directory as a configuration option.

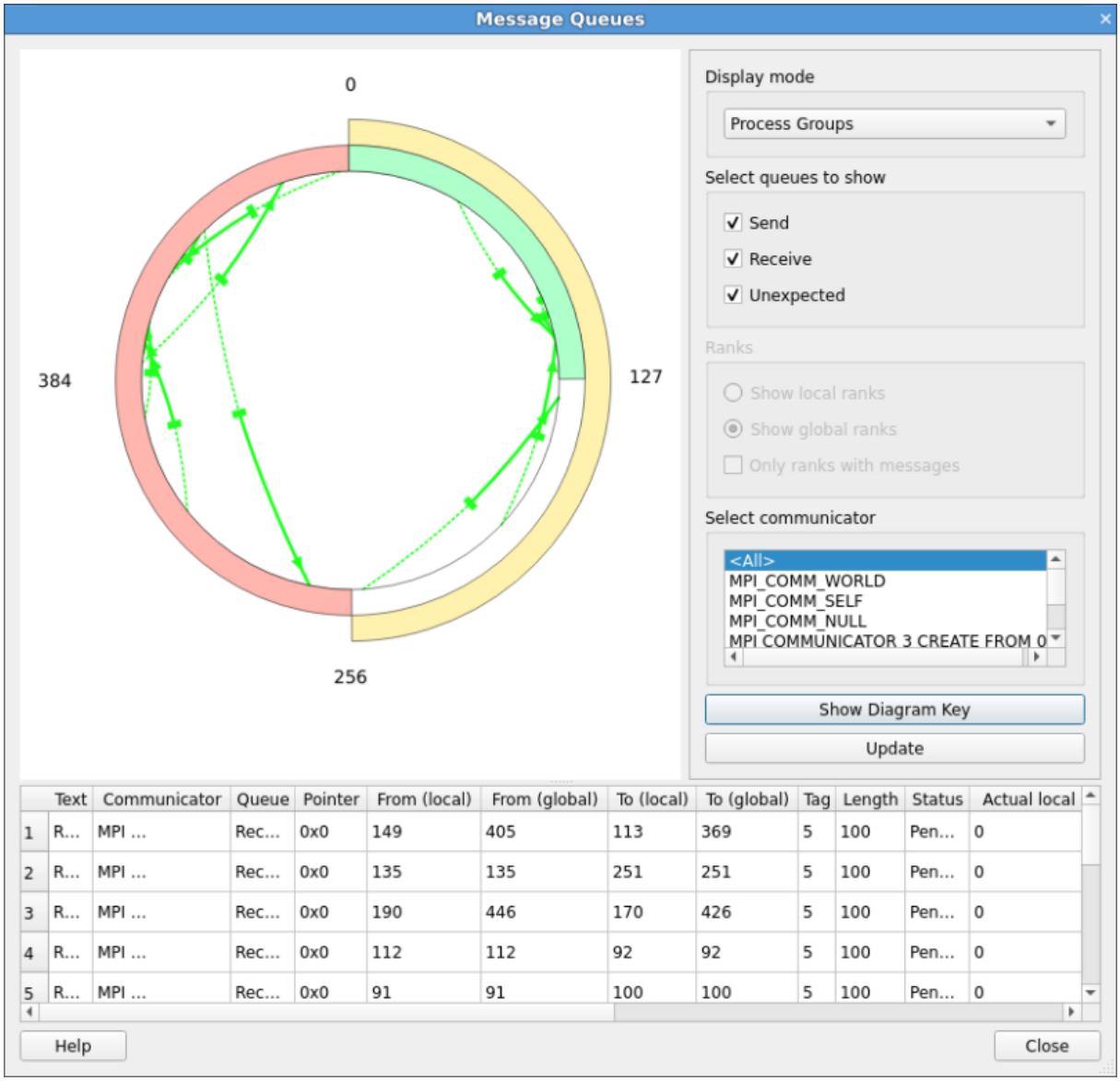
  Alternatively, you can specifically include the path to the support library in the `LD_LIBRARY_PATH`. If this is not convenient you can set the environment variable, `FORGE_QUEUE_DLL`, to the absolute path of the library itself (for example, `/usr/local/mpich-3.3.0/lib/libtvmpich.so`).

- 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, the debugger might not be able to find a support library that is the correct size. This is unsupported.

2.8.2 Interpret message queues

To see the messages in a group, you must choose an option in *Select communicator*. The ranks displayed in the diagram are the ranks within the communicator (not `MPI_COMM_WORLD`), if *Show local ranks* 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 *Display mode*, then select the rank of the process. To display a group of processes, select *Process Groups* in *Display mode* then 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 recipient side (to represent a message that has been sent but not yet 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 center of the diagram.

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

2.8.3 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.

2.9 Memory debugging

The powerful parallel memory debugging 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 that can be resolved using memory debugging:

- Memory exhaustion due to memory leaks can be prevented using the **Current Memory Usage** display, which groups and quantifies memory according to the location at which blocks have been allocated. Combine with **Node Memory Threshold Detection** to detect and diagnose potential out of memory errors early.
- Persistent but random crashes caused by access of memory beyond the bounds of an allocation block can be diagnosed 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.

2.9.1 Enable memory debugging

To enable memory debugging, in the *Run* window select the *Memory Debugging* checkbox.

The default options are usually sufficient, but you might 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.

When memory debugging is enabled, start your application as normal. The settings will be 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. You should refer to *Memory debugging options* for possible solutions.

2.9.2 CUDA memory debugging

There are two options for debugging memory errors in CUDA programs. They can be found in the *CUDA* section of the *Run* window.

See *Prepare to debug CUDA GPU code* before debugging the memory of a CUDA application.

When *Track GPU allocations* is enabled, CUDA memory allocations *made by the host* are tracked. That is, allocations made using functions such as `cudaMalloc`. You can find out how much memory is allocated, and where it was allocated from using the *Current Memory Usage* window.

Note: CUDA memory allocations *made by the GPU/device* such as `cuMemAlloc` are currently not tracked as well allocations made to *Unified Memory* with `cudaMallocManaged`. Furthermore, memory allocations made for CUDA arrays with functions such as `cudaMallocArray` are not tracked.

Allocations are tracked separately for each GPU and the host. If you enable *Track GPU allocations*, host-only memory allocations made using functions such as `malloc` will be tracked as well. You can choose 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 switches on the CUDA-MEMCHECK error detection tool. This tool 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 selected or not.

Note: Detect invalid accesses (`memcheck`) is not supported with CUDA 12.

For further details about CUDA hardware exceptions, see the NVIDIA documentation.

Note: It is not possible to track GPU allocations created by an OpenACC compiler because it does not directly call `cudaMalloc`.

2.9.3 PMDK memory debugging

Memory debugging can be used to track all allocations made by `libpmemobj`, an object store library that is part of the Persistent Memory Development Kit (PMDK).

To use PMDK memory debugging, enable memory debugging in the *Run* dialog. Optionally, a backtrace can be stored for each allocation. If memory debugging is not enabled, only the call site of the allocation is stored. No other configuration options have an effect on PMDK.

When the pool is opened with `pmemobj_open`, all the allocations that exist in the pool are tracked. The call site is where the pool is opened. The root object of the pool is not tracked. Allocation tracking persists after an aborted transaction.

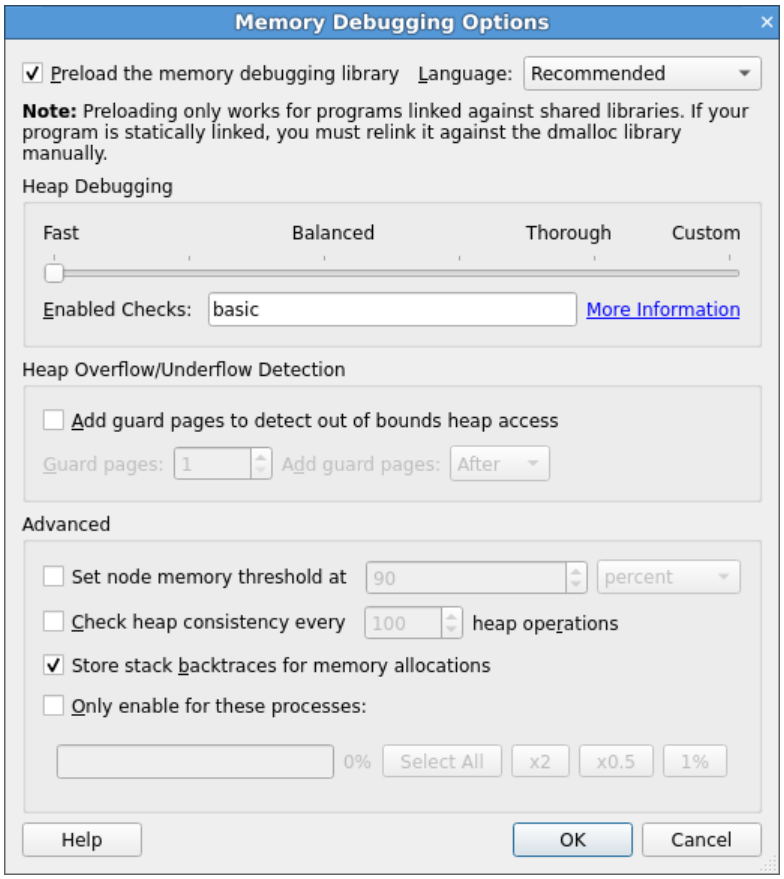
When tracking allocations, to see if a pointer was allocated by PMDK, right-click on a pointer in a variable view and select *View Pointer Details*. In the pointer details, you also see the backtrace, or call site, of the allocation. In the main menu, *Tools* ▸ *Current Memory Usage*, and *Tools* ▸ *Overall Memory Stats*, are enabled. By default, allocations made by libc are shown. To see the *Memory Usage* window, *Allocation Table*, and *Memory Statistics* window that you see with regular memory debugging, select *PMDK* in *Allocations from*. The sizes displayed are the sizes returned by `pmemobj_alloc_usable_size`, not the sizes you request.

2.9.4 Memory debugging options

Manual configuration is often unnecessary, but it can be used to adjust the memory checks and protection, or to alter the information which is gathered.

The current settings are displayed in the *Memory Debugging* section on the *Run* dialog.

To examine or change the settings, click *Details* adjacent to the *Memory Debugging* checkbox. The *Memory Debugging Options* window displays.



The available settings are:

Preload the memory debugging library:
When this is selected, the memory debugging library will be automatically loaded. The memory debugging library can only preload when you start a program. It uses shared libraries.

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

When attaching, you will need to change these settings after you attach. See the *Change settings at run time* section below for more information.

Language:
Choose the option that best matches your program, for example C, Fortran, No threads, threads. If it mixed language, or you are unsure which option to use, it is often sufficient to leave this set to *Recommended* rather than continually changing this value.

Heap Debugging:
This section allows you to trade speed for thoroughness. Some important things to remember are:

- Even the fastest (furthest left) setting will catch trivial memory errors such as deallocating memory twice.
- 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 the *Change settings at run time* section.

Enabled Checks:
Shows which checks are enabled for each setting. See the *Available checks* section for a complete list of available checks.

Heap Overflow/Underflow Detection:
This section can be used to detect out-of-bounds heap access. See *Writing beyond an allocated area* on *Pointer error detection and validity checking* for more details.

Set node memory threshold:
Can be used to detect potential out of memory errors early. See *Node memory threshold detection* for more details.

Check heap consistency:
Almost all users can leave the heap check interval at the default value. 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. We recommend a higher value (1000 or above) if your program allocates and deallocates memory very frequently, for example, inside a computation loop.

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

Only enable for these processes:
When this is selected, you can enable memory debugging only for the specified MPI ranks. Note that when you enable this feature, the memory debugging library is still preloaded into the other processes, but no errors are reported. Furthermore, backtraces for memory allocation are not stored and guard pages are not added for the other processes.

Note: If you choose the wrong library to preload, or the wrong number of bits, your job might not start, or might make memory debugging unreliable. You should check these settings if you experience problems when memory debugging is enabled.

2.9.4.1 Static linking

If your program is statically linked you must explicitly link the memory debugging library with your program to use the memory debugging feature.

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`

Where

- `--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 *Intel compilers* or *NVIDIA HPC SDK compilers* for compiler-specific information.

2.9.4.2 Available checks

The following heap checks are available in **Enable Checks**:

Name	Description
`basic`	Detect invalid pointers passed to memory functions (such as `malloc`, `free`, `ALLOCATE`, and `DEALLOCATE`)
`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 the known value of `dmalloc-alloc` byte (hex `0xda`, decimal `218`).
`free-blank`	Overwrite the bytes of freed memory with the known value of the `dmalloc-free` byte (hex `0xdf`, decimal `223`). If this check is enabled, the library overwrites memory when it is freed, using `dmalloc-free`. This can be used, for example, to check for corrupted allocations. This also checks and reports when a free byte has been written to, which can indicate if a freed pointer is being used.
`check-blank`	Check to see if blanked space has been overwritten. Space is blanked when it either has a pointer allocated to it, or the pointer has been freed. This 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.

2.9.4.3 Change settings at run time

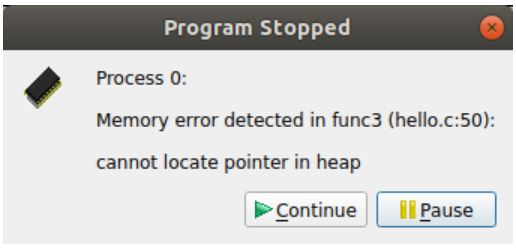
You can change most memory debugging settings while your program is running by selecting *Control* ▸ *Memory Debugging Options*. This means you can enable memory debugging with a minimal set of options when your program starts, set a breakpoint at a place you want to investigate for memory errors, then switch on more settings when the breakpoint is hit.

2.9.5 Pointer error detection and validity checking

When 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 for automatic monitoring for errors, and for user-driven inspection of pointers.

2.9.5.1 Library usage errors

If the memory debugging library reports an error, an error message will display. This briefly reports the type of error detected, and gives the options to continue playing the program, or to pause execution.



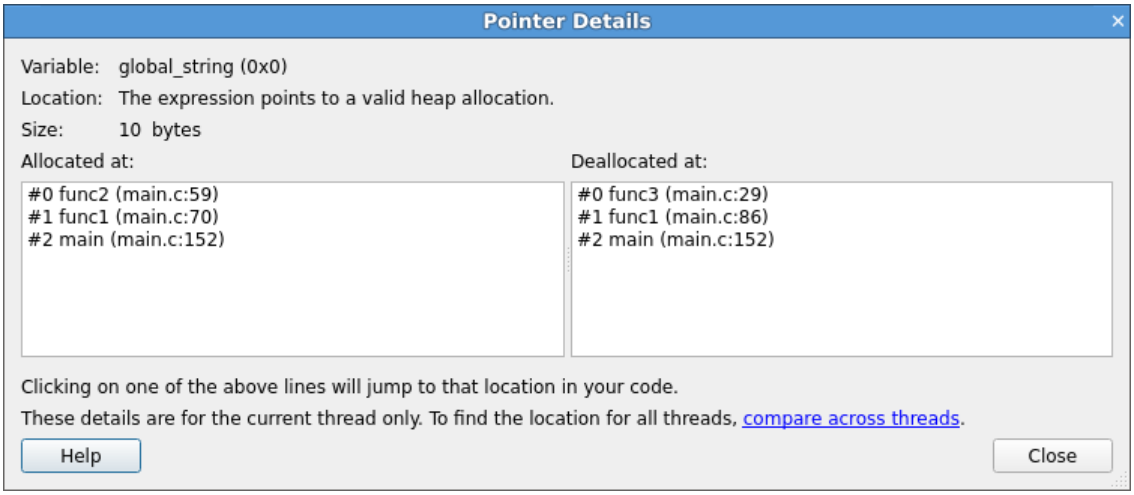
If you choose to pause the program, the line of your code that was being executed when the error was reported will be highlighted.

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, it is possible to examine some of the pointers referenced to see whether they are valid, and which line they were allocated on. This is explained in the following sections.

2.9.5.2 View pointer details

When memory debugging is enabled, right-click on any of the variables or expressions in the *Evaluate* window and choose *View Pointer Details*. This will display the amount of memory allocated to the pointer, and which part of your code originally allocated and deallocated that memory:

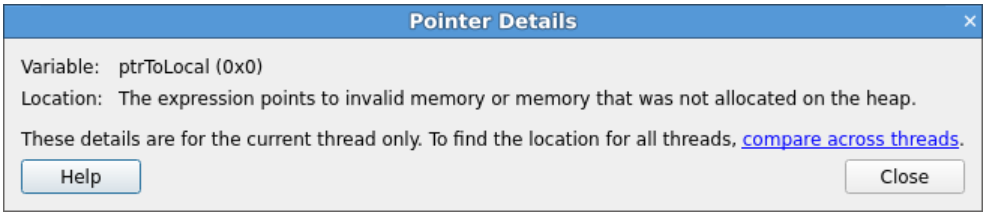


Click on any of the stack frames to display the relevant section of your code, so that you can see where the variable was allocated or deallocated.

Note: Only a single stack frame will be displayed if *Store stack backtraces for memory allocations* in the *Memory Debugging Options* window 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 a message will inform 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.



This is particularly useful for checking function arguments, and key variables when things seem to be going wrong. 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, you can often get an indication of the type and location of the memory being pointed to. The different types are:

- 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: It may only be possible to identify certain memory types with higher levels of memory debugging enabled. See *Memory debugging options* for more information.

For more information on fence post checking, see the *Fencepost checking* section.

2.9.5.3 Cross-process comparison of pointers

Memory debugging has an impact on the *Cross-Process Comparison View* and *Cross-Thread Comparison View*. See *Cross-process and cross-thread comparison*.

If you are evaluating a pointer variable, the *Cross-Process Comparison View* 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.

2.9.5.4 Writing beyond an allocated area

Use the *Heap Overflow/Underflow Detection* section on the *Memory Debugging Options* window 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 that stops your program.

Add the guard pages after the block to detect heap overflows, or before to detect heap underflows. The default value of 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: Small overflows/underflows (for example less than 16 bytes) might 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 the *Fencepost checking* section.

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

2.9.5.5 Fencepost checking

'Fence Post' checking will also be performed when the *Heap Debugging* section on the *Memory Debugging Options* window 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, this will be noticeable. However, your program will not be stopped at the exact location that your program wrote beyond the allocated data, it will only be stopped at the next heap consistency check.

2.9.5.6 Suppress an error

If your program stops at an error, but you wish to ignore it (for example, if it is in a third party library that you cannot fix), you can select *Suppress memory errors from this line in future*. This will open the *Suppress Memory Errors* window, where you can select which function you want to suppress errors from.

2.9.6 Node memory threshold detection

Running out of memory often causes a job to be killed instantly with no further debugging or diagnostic information available.

2.9.6.1 Usage

To detect potential out of memory errors early, enable *Set node memory threshold at* in *Memory debugging options* and optionally adjust the threshold if the default value of 90 percent is not suitable in your case.

There are two ways to set the node memory threshold:

- As percentage of node memory, by choosing *percent*.
- As an absolute value, by choosing *gigabytes* or *megabytes*.

If an absolute value is chosen, it must be less than the node memory capacity, otherwise the node memory threshold detection feature will be disabled. Setting an absolute threshold slightly above the typical memory usage of your program may allow you to detect a memory leak earlier than a percentage threshold.

If the node memory threshold detection is enabled, Linaro DDT will report a `over node memory threshold limit` memory error as soon as a dynamic memory allocation (such as `malloc` or `ALLOCATE`) would take the total memory usage of a compute node over the specified threshold.

When a `over node memory threshold limit` memory error is reported, you have the option to continue playing the program, or to pause the execution.

If you choose to continue playing the program, it is likely that future allocations will continuously trigger the `over node memory threshold limit` memory error. To suppress this memory error in the future, use *Control ▸ Memory Debugging Options* to increase or to disable the node memory threshold detection.

If you choose to pause the program, the line of your code that was being executed when the error was reported will be highlighted, however

- the reported location might not be the root cause of reaching the threshold, use *Tools ▸ Current Memory Usage* (see *Current memory usage* for more details) to diagnose the issue.
- the reported process might not be the root cause of reaching the threshold, if you debug more than one process on the same compute node. Use *Tools ▸ Current Memory Usage* to diagnose the issue or use *Tools ▸ Process Details* to find the affected nodes.
- the root cause of reaching the threshold might be an external process, use standard Linux command line utilities (such as `ps` or `top`) to diagnose the issue.

Note: The memory debugging library of Linaro DDT uses a custom allocator which will behave differently to the default allocator:

- It is likely that the total memory usage with memory debugging enabled is higher than without due to additional meta data.
- Only allocations which request additional memory from the operating system will report the `over node memory threshold limit` memory error.
- The custom allocator might not give back freed memory to the operating system.
- CUDA memory allocations do not check if the node memory threshold has been reached.

2.9.6.2 Offline usage

You can use the node memory threshold detection with offline debugging. There are two ways to enable the node memory threshold detection:

- Specify the `--mem-debug-threshold` command line option followed by the threshold percentage (must be between 1 and 99 percent, with or without %), for example `--mem-debug-threshold=90`.
- Specify the `--mem-debug-threshold` command line option followed by threshold bytes value (either MB or GB), for example `--mem-debug-threshold=32GB`.

If an absolute value is chosen, it must be less than the node memory capacity, otherwise the node memory threshold detection feature will be disabled.

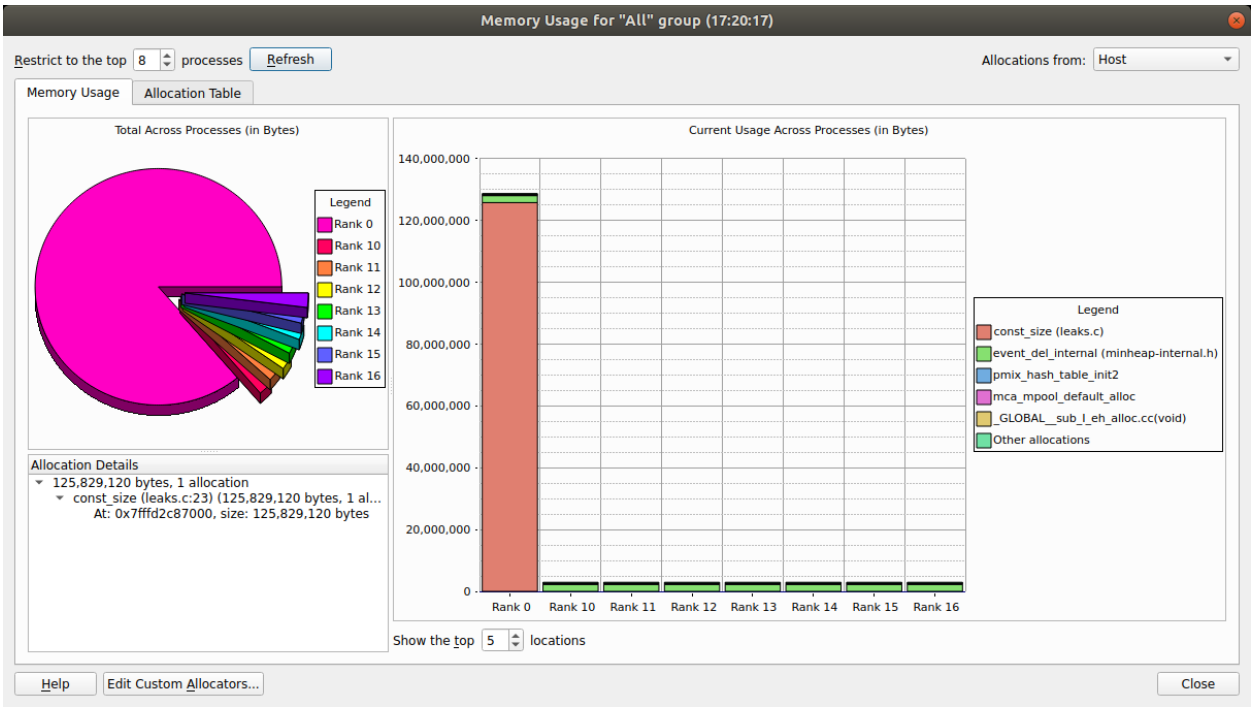
If the node memory threshold limit is reached, the program will be terminated and the offline log will contain a memory leak report, which can be used to diagnose the issue. See *Offline report HTML output* for more details.

2.9.7 Current memory usage

Memory leaks can be a significant problem for software developers. If your program's memory usage grows faster than expected, or continues to grow through its execution, 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.

At any point in your program, select *Tools* ▸ *Current Memory Usage* to 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.



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 one 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 six blocks in total. The first five represent the five 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, they are severe memory leaks.

Typically, if the memory leak does not make it into the top five 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. If you double-click on any one of these the *Pointer Details* window will open, showing you exactly where that pointer was allocated in your code.

Note: Only a single stack frame will be displayed if *Store stack backtraces for memory allocations* on the *Memory Debugging Options* window 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 includes 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. An allocation of 50 bytes will be reported against `func2` in the *Total Size* column. A cumulative allocation of 50 bytes will also be recorded against both functions `func1` and `func2` in the *Total Size (including called functions)* column.

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 that continue to grow. These are problematic leaks.

2.9.7.1 Detect leaks when using custom allocators/memory wrappers

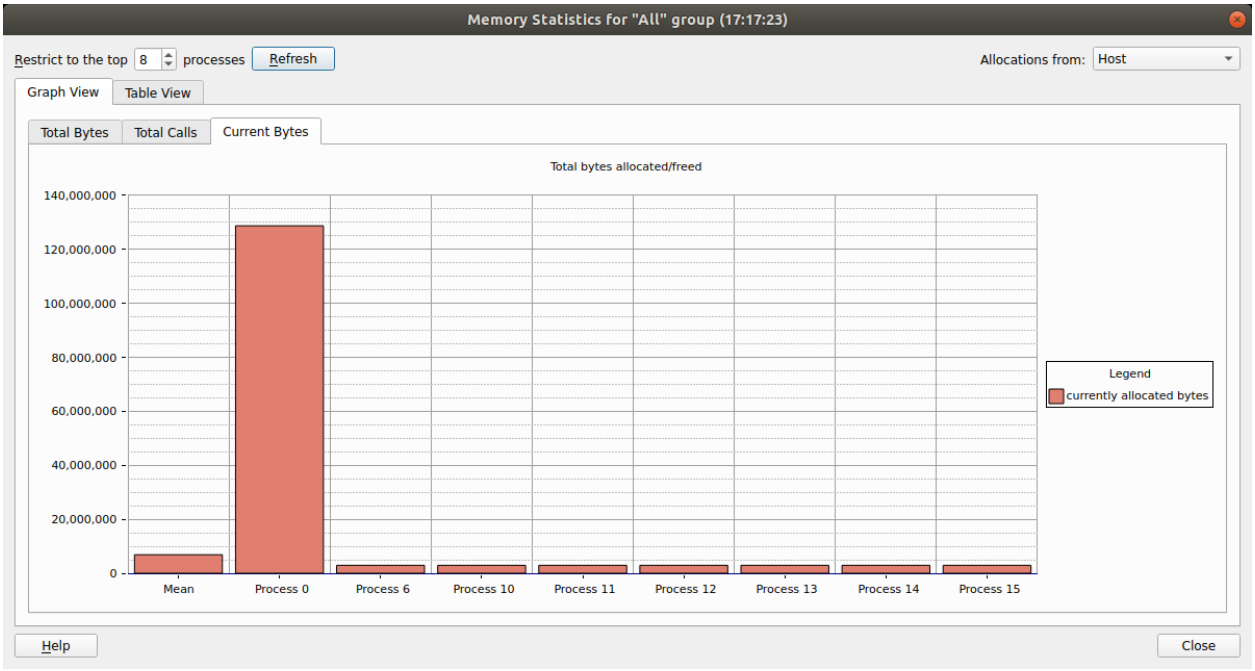
Some compilers wrap memory allocations inside many other functions. In this case you might find, for example, that all Fortran 90 allocations are inside the same routine. This can also happen if you have written your own wrapper for memory allocation functions.

In these circumstances you will see one large block in the *Memory Usage* tab. 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 *Add Custom Allocator*. Memory allocated by a custom allocator is recorded against its caller instead.

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

2.9.8 Memory Statistics

The *Memory Statistics* window (*Tools* ▸ *Overall Memory Statistics*) shows a total of memory usage across the processes in a program. 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.



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 program 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 window will be broken down into each type of memory in use.

2.10 Use and write plugins

Plugins are a quick and easy way to preload a library into your program, and define where to set breakpoints and tracepoints. A plugin is an XML file.

Examples include MPI correctness checking libraries, or if you define a library that is preloaded with your program that performs your own monitoring of the program.

Plugins can also be used to display a message when breakpoints are hit, showing, for example, an error message where the message is provided by the library in a variable.

2.10.1 Supported plugins

Plugin for MPI correctness checking functionality:

- Intel Message Checker, part of the Intel Trace Analyser and Collector (Commercial with free evaluation: https://www.intel.com/content/www/us/en/developer/tools/oneapi/trace-analyzer.html) version 7.1

Plugins for the GNU and LLVM compiler sanitizers:

- Address Sanitizer (also known as ASan)

  This 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 program whilst passing the `-fsanitize=address` compiler option to your compiler.
  2. Enable the Address Sanitizer plugin in Linaro DDT. For information how to enable a plugin, see *Use a plugin*.

  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 program:

     ```
     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 memory debugging.

- Thread Sanitizer (also known as TSan)

  This 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 while you pass the `-fsanitize=thread` compiler option to your compiler.
  2. Enable the Thread Sanitizer plugin in Linaro DDT. For information how to enable a plugin, see *Use a plugin*.

Note: TSan is not compatible with memory debugging.

2.10.2 Install a plugin

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

```
/path/to/forge/plugins/
```

It will then be included in the list of available plugins on the **Run** dialog.

Each plugin is an XML file in this directory. These files are usually provided by third-party vendors to enable their application to integrate with Linaro DDT.

Warning: Only install plugins from trusted third-party sources.

2.10.3 Use a plugin

To activate a plugin, select the checkbox next to it, then run your application.

Plugins can 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 *Write a plugin* for more details.
- Set tracepoints that log the variables during an execution.

If one of the plugins you have selected cannot be loaded, check that the program is correctly installed, and that the paths inside the XML plugin file match the installation path of the program.

Example Plugin: MPI History Library

The `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 to preload the library and set tracepoints to log the correct variables.
- `wrapper-source` - Used to automatically generate the source code for the library which will wrap the original MPI calls.

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

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

This package must be compiled before first use so it is compatible with your MPI version. When compiled it will be listed in the user interface.

To install as a non-root user in your local directory, type:

```
make local
```

To install as root in the `plugins` directory, type:

```
make
```

To enable the plugin, click *Details* to expand the *Plugins* section of the *Run* window. Then select *History v1.0*, and start your job as normal. The library will preload and set 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* tab.

The function counts are available in the form of the 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, in components such as the *Cross-Process Comparison View*, 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 program 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 `FORGE_FORGE_PATH` or `~/.allinea/plugins`). This would make execution considerably faster, but still retain the byte and function counts for the MPI functions.

2.10.4 Write a plugin

XML plugin files must be structured like this 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 optional.

A complete description of each tag can be found in *Plugin reference*.

Note: If you are interested in providing a plugin as part of your application bundle, Linaro can provide you with any assistance you need to get up and running. Contact Forge Support for more information.

2.10.5 Plugin reference

This table describes the tags used in the plugin files.

Tag	Attribute	Description
`plugin`	`name`	The plugin's unique name. This should include the program/library the plugin is for, and its version. This is shown in the **Run** dialog.
`plugin`	`description`	A short snippet of text to describe the purpose of the plugin/program to the user. This is shown in the **Run** dialog.
`preload`	`name`	Preloads a shared library of this name into the user's program. The shared library must be locatable using `LD_LIBRARY_PATH`, or the OS will not be able to load it.
`environment`	`name`	Sets a particular environment variable before running the user's program.
`environment`	`value`	The value that this environment variable should be set to.
`breakpoint`	`location`	Adds a breakpoint at this location in the code. The location can be in a preloaded shared library (see above). Typically this is 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 stop the program 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. Identical messages from different processes will be grouped together before displaying them to the user in a message box.
`extra_control_proc`	`hide`	Starts one more MPI process than the user requested. The optional `hide` attribute can be `first` or `last`, and will 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`	Similar to `breakpoint location`.
`tracepoint`	`variables`	A comma-separated list of variables to log on every passing of the tracepoint location.

2.11 GPU debugging

Linaro DDT can be used to debug programs that use GPU devices. The code running on the GPU is debugged simultaneously with the code on the host CPU.

Linaro DDT supports a number of GPU device families:

- NVIDIA GPUs
- AMD GPUs
- Intel Xe-HPC GPUs

The following sections outline GPU features that are available for NVIDIA, AMD and Intel Xe-HPC GPUs.

2.11.1 Control GPU threads

To control GPU threads use the standard play, pause, and breakpoints controls. They are all applicable to GPU kernels.

However, because GPUs have different execution models to CPUs, there are a few behavioral differences that are described below.

2.11.1.1 GPU breakpoints

GPU breakpoints can be set in the same way as other breakpoints. See *Set breakpoints*.

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

For more information about NVIDIA GPU breakpoint handling, see *NVIDIA GPU Breakpoints*.

For more information about AMD GPU breakpoints handling, see *AMD GPU Breakpoints*.

For more information about Intel Xe GPU breakpoints handling, see *Intel Xe GPU Breakpoints*.

2.11.1.2 Stepping

The GPU execution model is noticeably different to 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.

NVIDIA
: The smallest execution unit on a NVIDIA GPU is a warp, which on current GPUs is 32 threads. All threads in a warp execute in lockstep, which means that you cannot step each thread individually. All active threads in the warp execute step at the same time.

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

  Note: NVIDIA GPU functions are often inlined by the compiler. This can be avoided (dependent on hardware) by specifying the `__noinline__` keyword in your function declaration.

AMD
: The smallest execution unit on an AMD GPU is a wavefront, which on current GPUs is 64 threads. All threads in a wavefront execute in lockstep, which means that you cannot step each thread individually. All active threads in the wavefront execute step at the same time.

Intel Xe
: The smallest execution unit on an Intel Xe GPU is a sub-group, which on current GPUs is typically 8, 16 or 32 threads. All threads in a sub-group execute in lockstep, which means that you cannot step each thread individually. All active threads in the sub-group execute step at the same time.

2.11.1.3 Running and pausing

Click *Play/Continue* to run all GPU threads. It is not possible to run individual blocks, warps, or threads (NVIDIA) or workgroups, wavefronts, or threads (AMD), or workgroups, sub-groups or threads (Intel Xe).

Click *Pause* to pause a running kernel. Note that the pause operation is not as quick for GPUs as for regular CPUs.

2.11.2 Examine GPU threads and data

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

2.11.2.1 Select GPU threads

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

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

Note: For Intel Xe the 3D thread index is based on the global work-item ID; the block index is not used (grayed-out).

Please note that the global work-item ID might be transposed due to an optimization the Intel oneAPI runtime often performs, for example work-item ID `<<<10,20,30>>>` might become `<<30,20,10>>`.

The offset parameter for global work-item IDs, which is deprecated since SYCL 2020, is not applied when displaying the 3D thread index.

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:

- You change 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.

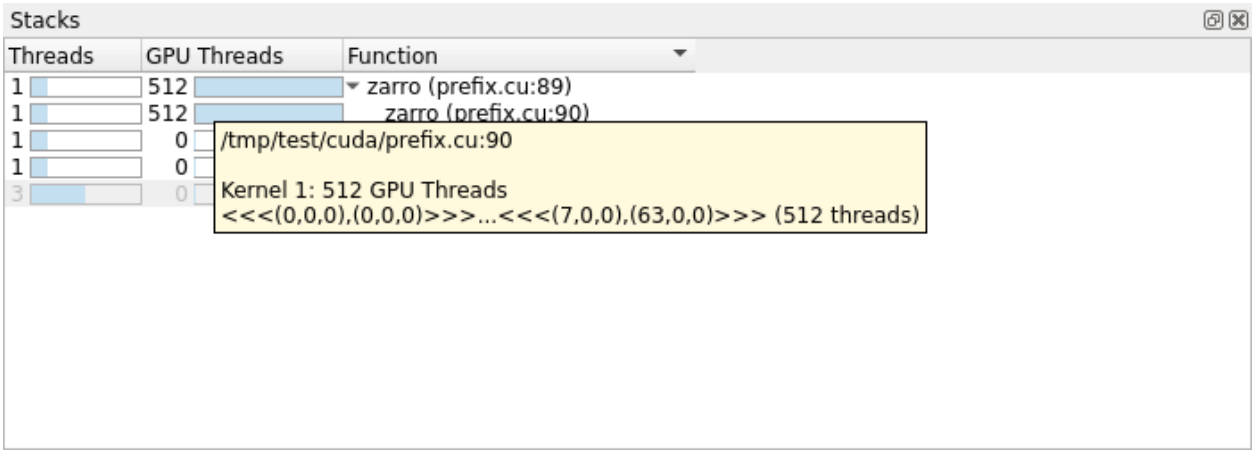
The *Thread Selector* also displays the dimensions of the grid and blocks in your program.

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.

2.11.2.2 View GPU thread locations

The *Parallel Stack View* displays the location and number of GPU threads.



Click an item in the *Parallel Stack View* to select the appropriate GPU thread, update the variable display components accordingly, and move the *Source Code viewer* to the appropriate location.

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

It is not possible to collect the stack trace for all threads in a timely manner. The stack traces are gathered by collecting one for each thread that has stopped in a unique location.

For more information about Intel Xe GPU and the *Parallel Stack View*, see *Intel Xe GPU Thread Details*.

2.11.2.3 Kernel Progress View

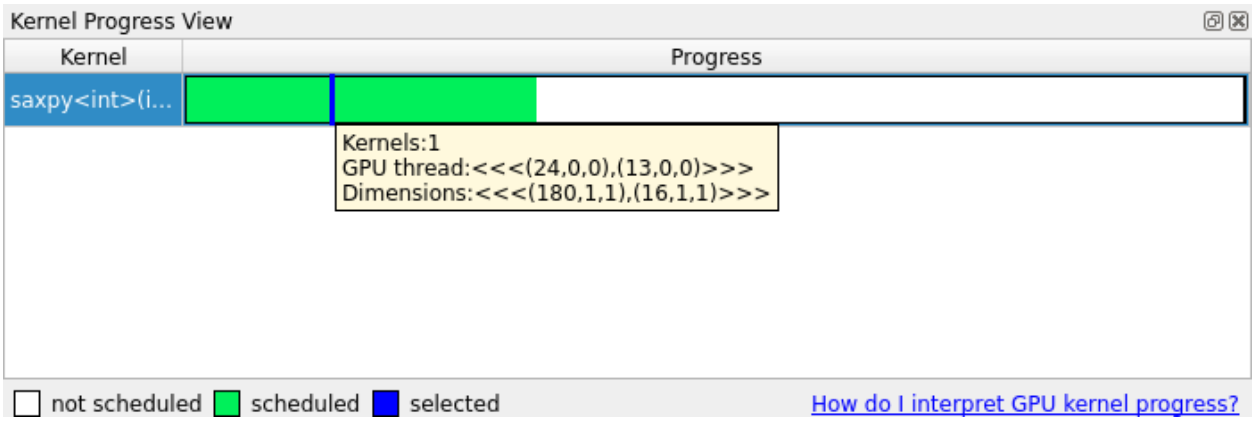
The *Kernel Progress View* displays at the bottom of the user interface by default when a kernel is in progress.

This view provides the necessary detail to help you decide whether array data is fresh or stale during debugging.

For 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.

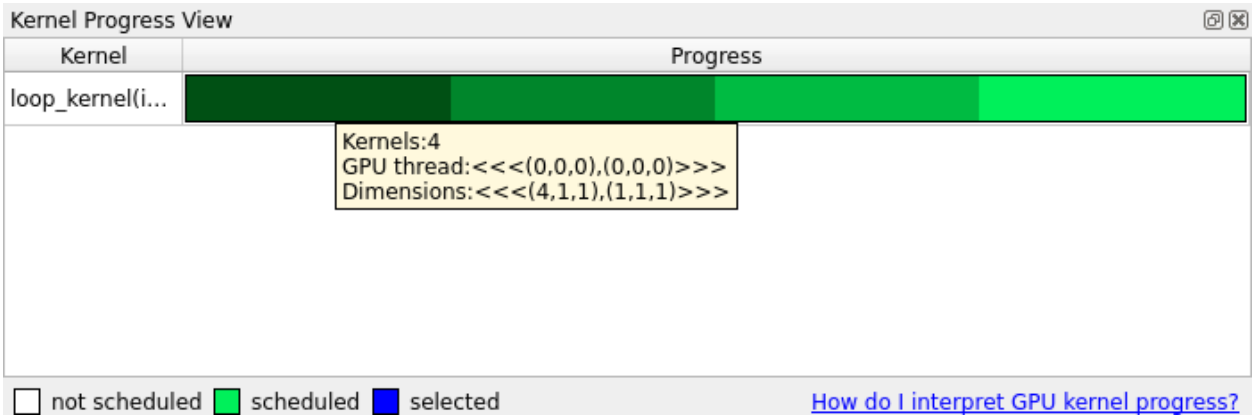
The *Kernel Progress View* identifies the kernels that are in progress. The number of kernels are identified and grouped by different kernel identifiers across processes. The identifier is the kernel name.



A colored progress bar shows which GPU threads are in progress. The progress bar is a projection onto a straight line of the GPU block and thread indexing system, which is potentially 6-dimensional. It illustrates the sizes of the kernels operating in the program.

Click the color-highlighted sections of the progress bar to select a thread that closely matches the click location. Blue represents the GPU thread that you selected.

Green GPU threads are threads which are scheduled to run. Multiple scheduled threads display in different shades of green to differentiate them.



White areas of the progress bar represent items which are inactive. They are inactive either because they have already run, or are not scheduled to run.

Kernels with the same name are stacked, and the shade of green becomes darker. If these kernels are different in size, then the maximum in each of the 6-dimensions is shown.

Kernels with different names display on separate rows.

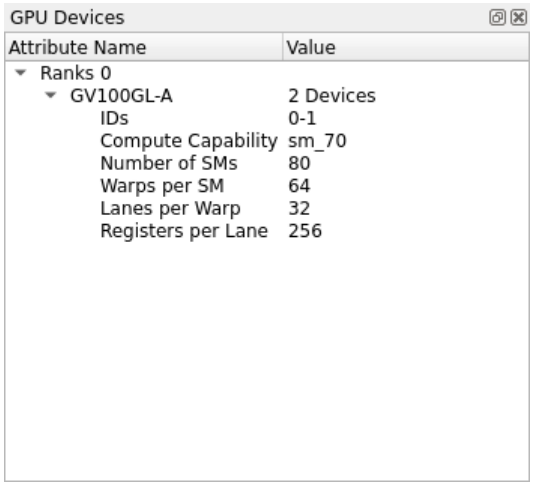
2.11.2.4 Source Code viewer

The *Source Code viewer* helps you 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 *Source Code viewer* will display a summary of the GPU threads on that line. See *Source code viewer*.

2.11.3 GPU devices information

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

The *GPU Devices* tab examines the GPUs that are present and in use across a program, and groups the information together scalably for multi-process systems.

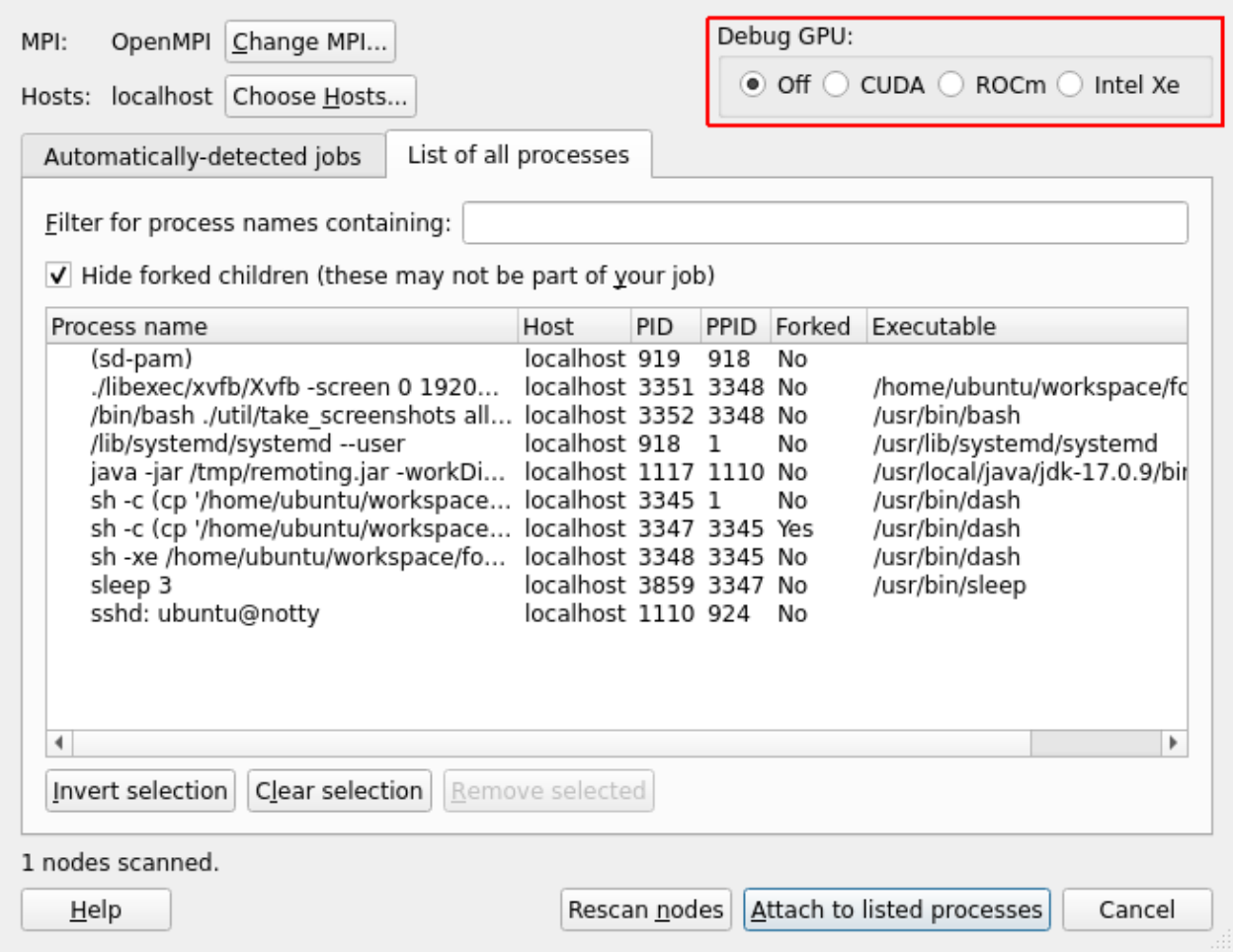


Note: GPU devices are only listed after initialization.

2.11.4 Attach to running GPU programs

You can attach to a running GPU program, and then debug the GPU threads.

For details how to attach to a running job, see *Attach to running programs*, and select the appropriate GPU runtime on the *Attach* window.



For more information about AMD GPU and Attaching to GPU Processes, see *Attach to running AMD GPU programs*.

For more information about Intel Xe GPU and Attaching to GPU Processes, see *Attach to running Intel Xe GPU programs*.

2.11.5 NVIDIA GPU debugging

Linaro DDT supports a number of GPU compilers that target NVIDIA GPU devices:

- NVIDIA's CUDA Compiler
- Cray OpenACC
- NVIDIA HPC SDK OpenACC and CUDA Fortran

From CUDA 13.0 onward CUDA GPU Debugging requires `cuda-gdb` to be available in your environment.

2.11.5.1 CUDA licensing

To debug CUDA programs with Linaro DDT you need a CUDA-enabled license key. This is an additional option. If CUDA is not included with a license, the CUDA options will be grayed-out on the *Run* and *Attach* dialogs.

Note: To serve a floating CUDA license you must use Linaro Forge Licence Server.

2.11.5.2 Prepare to debug CUDA GPU code

You might 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 optimizations that would hinder debugging. To use memory debugging with CUDA, `--cudart shared` must also be passed to `nvcc`.

For other compilers, see *NVIDIA GPU language support*, and your vendor's own documentation.

Note: OpenCL debugging of GPUs is not supported.

2.11.5.3 Launch the program

To launch a CUDA job, select *CUDA* on the *Run* dialog before you click *Run/Submit*. You can also enable memory debugging for CUDA programs in the *CUDA* section. See *CUDA memory debugging* for details.

It is not possible to attach to running CUDA programs if the program 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.

2.11.5.4 NVIDIA GPU Breakpoints

Breakpoints affect all GPU threads, and cause the program to stop when a thread reaches the breakpoint. Where kernels have similar workload across blocks and grids, 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 program again.

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.

2.11.5.5 Open CUDA core files

NVIDIA GPU core files can be opened in exactly the same way as core files generated by CPU code.

See *Open core files* for details.

2.11.5.6 Known issues and limitations

This section provides information about known issues and limitations with NVIDIA CUDA debugging.

2.11.5.6.1 Environment

From CUDA 13.0 onward NVIDIA GPU Debugging support in Linaro DDT requires `cuda-gdb` to be available in your environment. This `cuda-gdb` must be compatible with the NVIDIA GPU Driver installed on the system.

Linaro DDT will fail to start if *CUDA* is selected in the *Run Dialog* and `cuda-gdb` is not detected in the environment. `cuda-gdb` is available in the standard CUDA Toolkit installation.

In the event that `cuda-gdb` fails to start, you can still debug your application with the command-line option `--no-cuda` or deselecting *CUDA* in the *Run Dialog*. However, NVIDIA GPU Debugging will not be possible.

2.11.5.6.2 Limitations with system `cuda-gdb`

This section provides information about known issues and limitations with CUDA debugging with the `cuda-gdb` from the environment.

- The contents of `libstdc++` containers and container iterators are unreadable.
- Cannot reference anonymous variables using the `(anonymous namespace)::myvar` syntax in evaluations.
- Chance of segmentation fault when attaching to a CUDA process.
- Only the first element of array variables on the device are displayed by default. Workaround by using the *View As Vector* context menu item.
- Function evaluation in expressions when using OpenMP is not supported.
- CUDA Kernel Breakpoint may not be hit for Fortran program if a another breakpoint has been set.

Additionally, *Host-side debugging limitations* lists the differences that may be expected in host-side debugging when GPU debugging support is enabled.

2.11.5.6.3 Debug multiple NVIDIA 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:

```
FORGE_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 select *File* ▸ *Options* ▸ *Open MPI (Compatibility)* (*Linaro Forge* ▸ *Preferences* on Mac OS X). (Do not select *Open MPI*).
- The device selected for each process will be the only device visible when enumerating GPUs. This causes manual GPU selection code to stop working (due to changing device IDs, and so on).

2.11.5.6.4 Thread control

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

2.11.5.6.5 Detect invalid accesses (`memcheck`)

Detect invalid accesses (`memcheck`) is not supported with CUDA 12.

2.11.5.6.6 Notes

- NVIDIA CUDA toolkit and driver - Linaro recommends using the most recent version of the toolkit. For more information, see *Reference table*.
- X11 cannot be running on any GPU used for debugging. (Any GPU running X11 is 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.
- It is not 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.
- Device memory allocated via `cudaMalloc()` is not visible outside of the kernel function. Add `@global` to the type specifier to view the data of device memory pointers while on the host. See *View array data* for more information.
- Not all illegal program behavior can be caught in the debugger, for example, divide-by-zero.
- Breakpoints in divergent code might 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, make sure no X server is running on any of the GPUs.

  Note: Any GPU running X will be excluded from enumeration, which can affect the device IDs.

- If memory debugging and CUDA support are enabled, only thread-safe memory debugging libraries are supported.
- You may encounter an issue stopping at a GPU breakpoint with the CUDA 12.9 driver on SuSE 15. Contact Forge Support should you encounter this issue.

2.11.5.7 NVIDIA GPU language support

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

Note: Debugging of OpenCL is not supported on the device.

2.11.5.7.1 Cray OpenACC

Cray OpenACC is fully supported. Code pragmas are highlighted, most variables are visible within the device, and stepping and breakpoints in the GPU code are supported. The compiler flag `-g` is required for enabling device (GPU-based) debugging.

These are currently 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 program.

2.11.5.7.2 NVIDIA HPC OpenACC and CUDA Fortran

Linaro DDT supports debugging both the host and CUDA parts of NVIDIA HPC OpenACC and CUDA Fortran programs when compiled with the NVIDIA HPC compiler.

For information about currently supported software versions, see *Reference table*.

2.11.6 ROCm GPU debugging

Linaro DDT supports a number of GPU compilers that target ROCm devices:

- AMD's HIP Compiler
- AMD's amdclang OpenMP offloading

ROCm GPU Debugging requires `rocgdb` to be available in your environment.

Note: The `rocgdb` in your environment must be compatible with the AMD GPU Driver installed on the system. See *Known issues and limitations* for more information.

2.11.6.1 ROCm licensing

To debug ROCm programs with Linaro DDT you need a ROCm-enabled license key. This is an additional option. If ROCm is not included with a license, the ROCm options will be grayed-out on the *Run* and *Attach* dialogs.

Note: To serve a floating ROCm license you must use Linaro Forge Licence Server.

2.11.6.2 Prepare to debug ROCm code

For the `hipcc` compiler, kernels must be compiled with the `-g` flag. This enables generation of information for debuggers in the kernels, and also disables some optimizations that would hinder debugging.

For other compilers, see *AMD GPU language support*, and your vendor's own documentation.

Note: OpenCL debugging of GPUs is not supported.

2.11.6.3 Launch the program

Note: `rocgdb` must be detectable in your environment and compatible with the AMD GPU Driver installed on the system. See *Known issues and limitations* for more information.

To launch a ROCm job, select *ROCm* in the *Run* dialog before you click *Run/Submit*.

For MPI applications it is essential to place all ROCm initialization after the `MPI_Init` call.

2.11.6.4 Attach to running AMD GPU programs

Attaching to a running AMD GPU program requires `rocgdb` to be available in your environment.

Note: When attaching to an AMD ROCm application, to see dispatches started prior to attaching, start the user application with the environment variable `HSA_ENABLE_DEBUG=1`.

2.11.6.4.1 Remote Attach

When attaching to an AMD GPU process on a remote host, first see *Connecting remotely*.

Attaching to a running AMD GPU program on a remote host requires `rocgdb` be available in your environment on the remote host.

To ensure this, add the relevant `module` command and export any required environment variables for adding `rocgdb` to your `PATH` in the *Remote script*.

2.11.6.5 AMD GPU Breakpoints

Breakpoints affect all GPU threads, and cause the program to stop when a thread reaches the breakpoint. Threads of the same wavefront reach the breakpoint together and the kernel pauses once per wavefront.

The number of breakpoints hit in a GPU kernel can be refined using:

- Conditional Breakpoints, see *Conditional breakpoints*.
- Hit Limits, see *Set breakpoints*.

Furthermore, where kernels have divergent distributions of work across threads, all GPU threads will stop at a given breakpoint. However, the GPU Threads that are not involved in the work at this breakpoint will appear as in-active and cannot be selected.

For example, a divergent if-else statement with a breakpoint set on each branch will result in all GPU threads stopping at the first breakpoint, but with those not satisfying the if-statement criteria being marked as in-active. When the first breakpoint has been hit by all GPU threads, all threads will progress to the next branch of the if-else statement but with the GPU threads who do not satisfy the else-statement criteria being marked as inactive.

To apply breakpoints to individual workgroups, wavefronts, 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.

2.11.6.6 Open AMD GPU core files

AMD GPU core files can be opened in exactly the same way as core files generated by CPU code. See *Open core files* for details.

Note: Opening an AMD GPU core file requires `rocgdb` to be available in your environment.

Note: AMD GPU core files are only supported for ROCm 6.2+.

Instructions on how to create a combined host (CPU) and device (GPU) core file from the AMD ROCm runtime can be found in the `rocgdb` debugger manual.

2.11.6.7 Known issues and limitations

2.11.6.7.1 Environment

AMD GPU Debugging support in Linaro DDT requires `rocgdb` to be detectable in your environment. This `rocgdb` must be compatible with the AMD GPU Driver installed on the system.

Linaro DDT will fail to start if *ROCm* is selected in the *Run Dialog* and `rocgdb` is not detected in the environment. `rocgdb` is available in the standard ROCm Toolkit installation.

Linaro DDT will fail to start if the ROCm environment is incompatible with the installed AMD GPU Driver. More information about the error messages observed in this case can be found on AMD's `rocgdb` documentation, see Debugging with ROCgdb.

In the event where these environment errors cannot be resolved, you can still debug your application with the command-line option `--no-rocm` or deselecting *ROCm* in the *Run Dialog*. However, AMD GPU Debugging will not be possible.

Please contact Forge Support if you encounter an issue.

2.11.6.7.2 Limitations

This section provides information about known issues and limitations with ROCm debugging.

- AMD GPU core files cannot be opened in the same way as core files generated by CPU code.
- By default, Linaro DDT gathers information about all lanes of each AMD GPU wavefront to show the correct number of GPU threads in the *Parallel Stack View*, or to allow you to select the 3D thread index in the *GPU Thread Selector*.

  This might cause performance issues. To only gather information at the wavefront level, use the environment variable `FORGE_ROCM_LANES=0`. With this environment variable set, the *Parallel Stack View* only shows the number of wavefronts for GPU Threads. The 3D thread index selector is one dimensional and only allow you to switch between wavefronts.
- Switching to inactive lanes is currently disabled. If you attempt to switch to an inactive lane, an error message displays.
- To see dispatches started prior to attaching, start the user application with the environment variable `HSA_ENABLE_DEBUG=1`.
- Symbolic debugging is only available with ROCm 5.1 and later and the AMD AFAR compiler.
- GPU Device watchpoints are only supported for global memory.
- The HIP runtime currently performs deferred code object loading by default. This will result in conditional breakpoints not being hit if set before the first kernel is launched. To set conditional breakpoints before the first kernel is launched, start the user application with the environment variable `HIP_ENABLE_DEFERRED_LOADING=0`.
- There may be issues with dereferencing STL pointers with ROCm 5.1. Please contact Forge Support if you encounter this issue.

Additionally, *Host-side debugging limitations* lists the differences that may be expected in host-side debugging when GPU debugging support is enabled.

2.11.6.8 AMD GPU language support

In addition to the native `hipcc` compiler, other compilers support offloading to AMD GPU.

2.11.6.8.1 ROCmCC with offloading OpenMP

Linaro DDT supports debugging both the host and AMD parts of OpenMP programs using offloading when compiled with the AMD ROCm compiler `ROCmCC`.

Linaro recommends using these compiler flags to get an optimal debugging experience when offloading OpenMP regions with `amdclang`, `amdclang++` and `amdflang`:

```
-fopenmp -fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-target=amdgcn-amd-amdhsa -march=<arch e.g gfx906>
```

For information about currently supported software versions, see *Reference table*.

2.11.6.8.2 Cray with offloading OpenMP

Linaro DDT supports debugging both the host and AMD parts of OpenMP programs using offloading when compiled with the Cray Compiler.

Note: Debugging of OpenCL is not supported on the device.

2.11.7 Intel Xe GPU debugging

Linaro DDT supports a number of GPU compilers that target Intel Xe GPU devices:

- Intel oneAPI DPC++/C++ Compiler (SYCL kernels and OpenMP target offload)
- Intel Fortran Compiler (OpenMP target offload)

Intel Xe GPU Debugging requires `gdb-oneapi` and `gdbserver-ze` to be available in your environment, which are typically installed by the Intel oneAPI toolkit.

Note: You must ensure that your environment is configured to enable Intel Xe GPU Debugging, in particular you must use a GPU driver that supports debugging. Additionally, if you are debugging by attaching to a running process, you need to make sure the program was run with the environment variable `ZET_ENABLE_PROGRAM_DEBUGGING=1` set.

Your system administrator might already have configured these required settings for you.

For further guidance on how to configure your environment for debugging, see the Intel documentation GPU Debugging and Get Started with Intel Distribution for GDB.

2.11.7.1 Intel Xe licensing

To debug Intel Xe programs with Linaro DDT you need an Intel Xe-enabled license key. This is an additional option. If Intel Xe is not included with a license, the Intel Xe options will be grayed-out on the *Run* and *Attach* dialogs.

Note: To serve a floating Intel Xe license you must use Linaro Forge Licence Server.

2.11.7.2 Prepare to debug Intel Xe code

2.11.7.2.1 Intel oneAPI DPC++/C++ Compiler

For the `icpx` (or `mpiicpx`) compiler, SYCL kernels or OpenMP regions targeted for an Intel Xe GPU device must be compiled with the `-g -O0` flags. This enables generation of information for debuggers in the kernels, and also disables some optimizations that would hinder debugging.

2.11.7.2.2 Intel Fortran Compiler

For the `ifx` (or `mpiifx`) compiler, OpenMP regions targeted for an Intel Xe GPU device must be compiled with the `-g -O0` flags. This enables generation of information for debuggers in the OpenMP regions, and also disables some optimizations that would hinder debugging.

2.11.7.3 Launch the program

To launch an Intel Xe GPU job, select *Intel Xe* in the *Run* dialog before you click *Run/Submit*.

For MPI applications it is essential to place all Intel Xe initialization after the `MPI_Init` call.

Note: Due to a known issue, after the startup you must not step out of `MPI_Init`, instead run to a line (or set a breakpoint) after the `MPI_Init` call, which can be a line in the kernel that you want to debug.

2.11.7.4 Attach to running Intel Xe GPU programs

Attaching to a running Intel Xe GPU program requires `gdb-oneapi` to be available in your environment.

Note: If you are debugging by attaching to a running process, you need to make sure the program was run with the environment variable `ZET_ENABLE_PROGRAM_DEBUGGING=1` set.

2.11.7.4.1 Remote Attach

When attaching to an Intel Xe GPU process on a remote host, first see *Connecting remotely*.

Attaching to a running Intel Xe GPU program on a remote host requires:

- `gdb-oneapi` be available in your environment on the remote host.
- `ZET_ENABLE_PROGRAM_DEBUGGING=1` to be set on the remote host.

To ensure this, add the relevant `module` command and export any required environment variables for adding `gdb-oneapi` to your `PATH` in the *Remote script*.

For example:

```
export ZET_ENABLE_PROGRAM_DEBUGGING=1
source /path/to/oneAPI/installation/setvars.sh
```

2.11.7.5 Intel Xe GPU Breakpoints

Breakpoints affect all GPU threads, and cause the program to stop when a thread reaches the breakpoint. Threads of the same sub-group reach the breakpoint together and the kernel pauses once per sub-group.

The number of breakpoints hit in a GPU kernel can be refined using:

- Conditional Breakpoints, see *Conditional breakpoints*.
- Hit Limits, see *Set breakpoints*.

2.11.7.5.1 Divergent Execution

Where kernels have divergent execution within a sub-group, GPU threads which are currently in-active (diverge) will not be shown by Linaro DDT.

For example, a divergent if-else statement with a breakpoint set on each branch, will result only in GPU threads stopping at the first breakpoint that satisfy the if-condition. GPU threads which do not satisfy the if-condition will not be shown.

When the first breakpoint has been hit by all GPU threads of a sub-group, the GPU threads will progress to the next branch of the if-else statement, but with the GPU threads who do not satisfy the else-statement condition being hidden.

Attempting to select an in-active GPU thread will result in an error message.

2.11.7.5.2 Conditional Breakpoints

Breakpoints can be applied to individual work-item global IDs, work-item local IDs or work-group IDs.

- To apply breakpoints to individual work-items, based on their global ID use the GDB convenience variable `$_workitem_global_id`.
- To apply breakpoints to individual work-items, based on their local ID and work-group ID use the GDB convenience variables `$_workitem_local_id` and `$_thread_workgroup` respectively.

These GDB convenience variables are 3-dimensional and each dimension can be accessed using the respective array element, for example `$_workitem_global_id[0]`, `$_workitem_local_id[1]` or `$_thread_workgroup[2]`.

Example: To apply a breakpoint only to work-item with the global ID <<<4,6,9>>> use the following condition: `$_workitem_global_id[0] == 4 && $_workitem_global_id[1] == 6 && $_workitem_global_id[2] == 9`. Please note that the global work-item ID might be transposed.

Note: Breakpoints of GPU threads are reported in the order they are scheduled on Execution Units (EU).

Due to internal GPU thread scheduling behavior, it might take a significant amount of time until a conditional breakpoint is hit.

2.11.7.6 Intel Xe GPU Thread Details

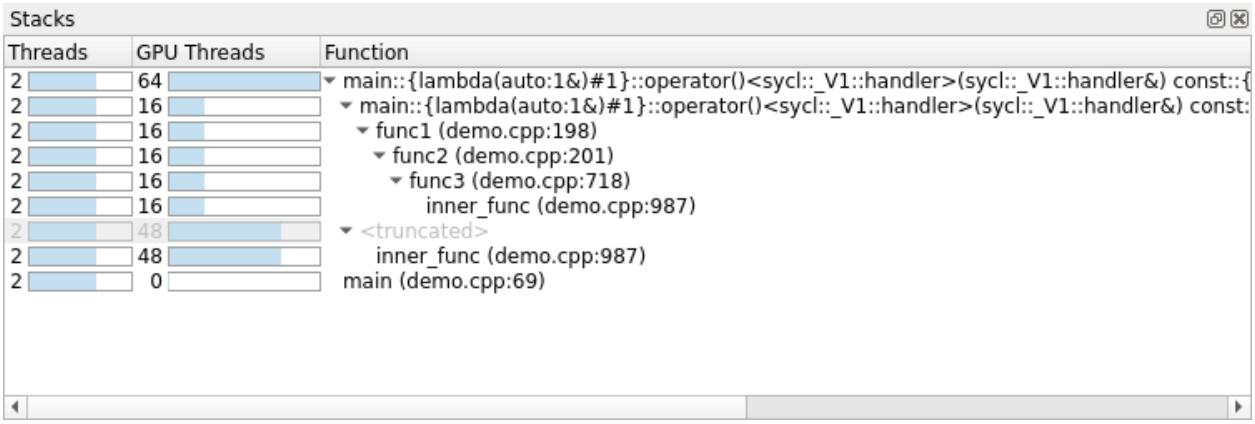
When a process stops, Linaro DDT collects certain data about all threads of a process, including GPU threads. For a GPU thread, this collected data includes the stack and the global work-item ID.

The level of detail collected for each GPU thread can be controlled by the environment variable `FORGE_INTEL_XE_THREAD_DETAILS`.

`FORGE_INTEL_XE_THREAD_DETAILS`

Value	Details
`full`	Stack and global work-item ID information is collected for all GPU Threads.
`shallow` (default)	The full stack is only collected for one GPU Thread of a kernel. For all other GPU threads in the kernel, only the newest (current) frame is collected. Global work-item ID information is collected for all GPU Threads.
`single`	Stacks and global work-item ID information is only collected for one GPU thread of a kernel. All other GPU Threads are not displayed.

In the default case `shallow`, the GPU Threads with only the newest frame collected will be grouped under an entry marked `<truncated>` in the *Parallel Stack View*.



For `shallow` and `single` thread detail levels, the stack information will be collected for the first reported GPU Thread.

Note: The first reported GPU Thread will not always correspond to global work-item ID 0. The GPU Threads are reported in the order they are scheduled on Execution Units (EU).

2.11.7.7 Known issues and limitations

2.11.7.7.1 Environment

Intel Xe GPU Debugging support in Linaro DDT requires `gdb-oneapi` to be available in your environment.

Linaro DDT will fail to start if the *Intel Xe* is selected in the *Run Dialog* and `gdb-oneapi` is not detected in the environment. `gdb-oneapi` is available in the standard Intel oneAPI toolkit.

Please contact Forge Support if you encounter an issue.

2.11.7.7.2 Limitations

There are a number of generic issues that may impact debugging on Intel Xe GPUs you should be aware of. For a complete list of known issues see Intel Distribution for GDB Release Notes, in particular:

- Applications using unified shared memory (USM):

  > "Applications that use unified shared memory (USM) may appear as raising a SIGSEGV when a USM-allocated memory is being accessed. This is a mechanism used by the runtime to trigger memory migration."

  In such cases, continue the program, which should forward the `SIGSEGV` signal to the application.

- Breakpoint location:

  > "If you define a breakpoint at a location before a kernel (inside the host code), the breakpoint is also defined at the start of the kernel. This is similar to defining a breakpoint at a comment line or an empty line: in these cases, the breakpoint is defined for the next source line."

- When using OpenMP offload, the mapping of kernel dimensions and global work-item IDs to parallel for loops depends on the Intel oneAPI runtime. A 1:1 mapping is unlikely, especially if multiple nested parallel for loops are used within one region. Use the `thread_limit` and `num_teams` clauses to control the number of work-items spawned.
- Conditional breakpoints might take a significant amount of time until they are hit, due to internal GPU thread scheduling behavior.
- Hardware watchpoints are not implemented for Intel Xe GPUs. Software watchpoints might work, but are unsupported and impose significant performance slowdown on the application being debugged.
- When running MPI SYCL applications, it has been observed that some kernels may not launch until other kernels have finished, even if it would be queued to a different device. If this occurs, we advise manually setting a unique affinity for each MPI process (e.g by setting `ZE_AFFINITY_MASK` in a wrapper script).
- When pausing a process running a GPU Kernel, Linaro DDT may report a process stopped with signal SIGTRAP. Select *Pause* in the dialog to stop the process.
- When running to a breakpoint, it has been observed that a dialog may not appear. The processes will stop and can be examined.

Linaro DDT specific issues:

- After the startup you must not step out of `MPI_Init`, instead run to a line (or set a breakpoint) after the `MPI_Init` call, which can be a line in the kernel that you want to debug.
- When using Memory Debugging, you must set `NEOReadDebugKeys=1` and `NEO_DisableDeepBind=1` (or `DisableDeepBind=1`) to disable the use of `RTLD_DEEPBIND` in the Intel oneAPI runtime, which otherwise would conflict with the memory debugging library of Linaro DDT.
- Memory debugging will only capture allocations made on the host.

Additionally, *Host-side debugging limitations* lists the differences that may be expected in host-side debugging when GPU debugging support is enabled.

2.11.8 Host-side debugging limitations

Linaro DDT GPU debugging on *NVIDIA* (>= CUDA 13.0), *AMD* and *Intel* GPUs all use a system debugger found in the environment. Whilst required for GPU debugging, these system debuggers may have the following differences in functionality when examining code running on the host compared to Linaro DDT debugging a non-GPU program.

- The Fortran modules pane is not populated.
- Setting pinned watchpoints in Fortran is inoperative.
- Cannot drill down through inheritance hierarchies using the variable panes. Work around by explicitly casting to the base class in the evaluation expression.
- Long strings may not be elided. Particularly long strings may not be shown at all.
- Smart pointers will not be inspectable in the evaluation panes.
- The debugging session may terminate if your application contains bad DWARF information.

If you do not require debugging on the GPU, you can workaround the above issues by launching Forge with the command-line option `--no-cuda`, `--no-rocm` or `--no-xe` as applicable, or deselecting the relevant GPU support in the *Run Dialog*. However, GPU Debugging will not be possible.

2.12 Offline debugging

Offline debugging is when a program is run under the control of the debugger, but without user intervention and without a user interface.

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

2.12.1 Use offline debugging

To use offline debugging, specify the `--offline` argument. Optionally, specify an output filename with the `--output=<filename>` argument. A filename with a `.html` or `.htm` extension will generate an HTML version of the output. In other cases, a plain text report will be generated. If the `--output` argument is not used, an HTML output file will be generated 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, and set expressions to be evaluated on every program pause.

Settings from your current 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 *File ▸ Save Session*.
- `--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 *Run-time job progress reporting*.
- `--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 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 `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 program will run to completion, or to the end of the job.

When errors occur, for example a program crash, the stack back trace of crashing processes is recorded to the offline output file. In offline mode, it is as if you clicked *Continue* if the continue option was available in an equivalent 'online' debugging session.

2.12.1.1 Read 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>
```

2.12.1.2 Write a file from standard output

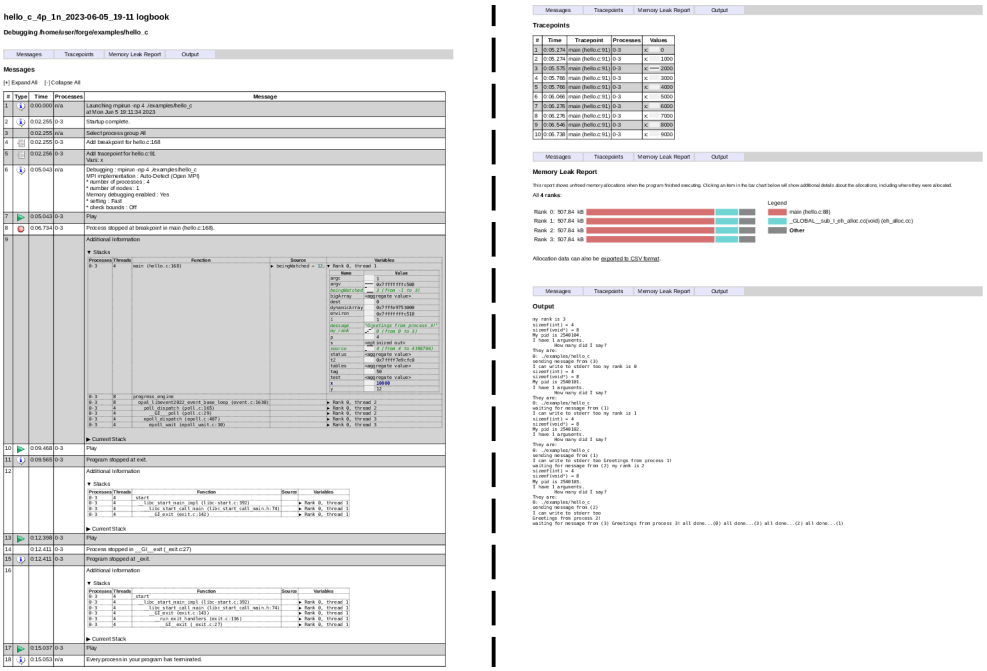
Normal redirection can also be used to write data to a file from the executable's standard output:

```
ddt --offline -o myjob.html ... > <output-file>
```

2.12.2 Offline report HTML output

The output file contains four sections: **Messages**, **Tracepoints**, **Memory Leak Report**, and **Output**. At the end of a job, the four sections of the log output are merged (tracepoint data, error messages, memory leak data, and program output) into one file. If the process is terminated abruptly, for example by the job scheduler, these separate files will remain and the final single HTML report might not be created.

Note: The **Memory Leak Report** section is only created when memory debugging is enabled.



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.

2.12.2.1 Messages

The **Messages** section contains the following:

- Error messages: for example if memory debugging detects an error, the message and the stack trace at the time of the error will be recorded from each offending process.
- 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, 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 show the variables across the paused processes. The text highlighting scheme is the same as for the local variables in the user interface. 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.

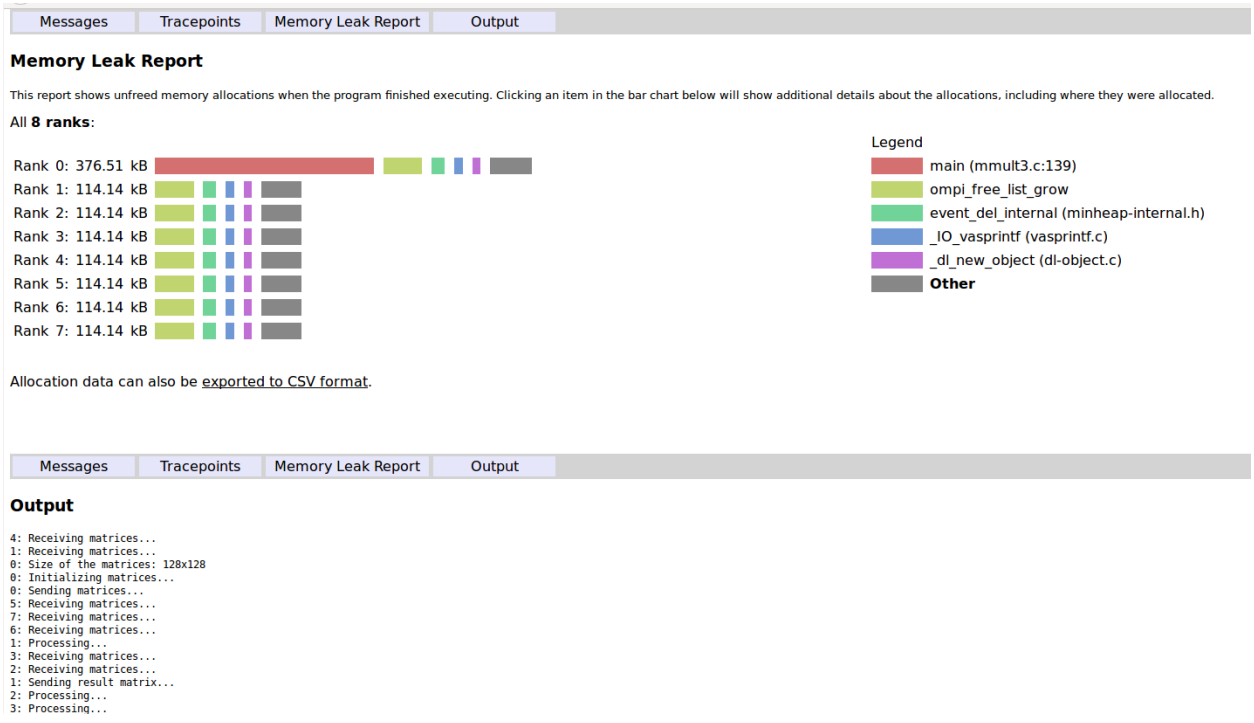
2.12.2.2 Tracepoints

The **Tracepoints** section contains the output from tracepoints, similar to that shown in the **Tracepoints** tab in an online debugging session. This includes sparklines displaying the variable distribution.

2.12.2.3 Memory Leak Report

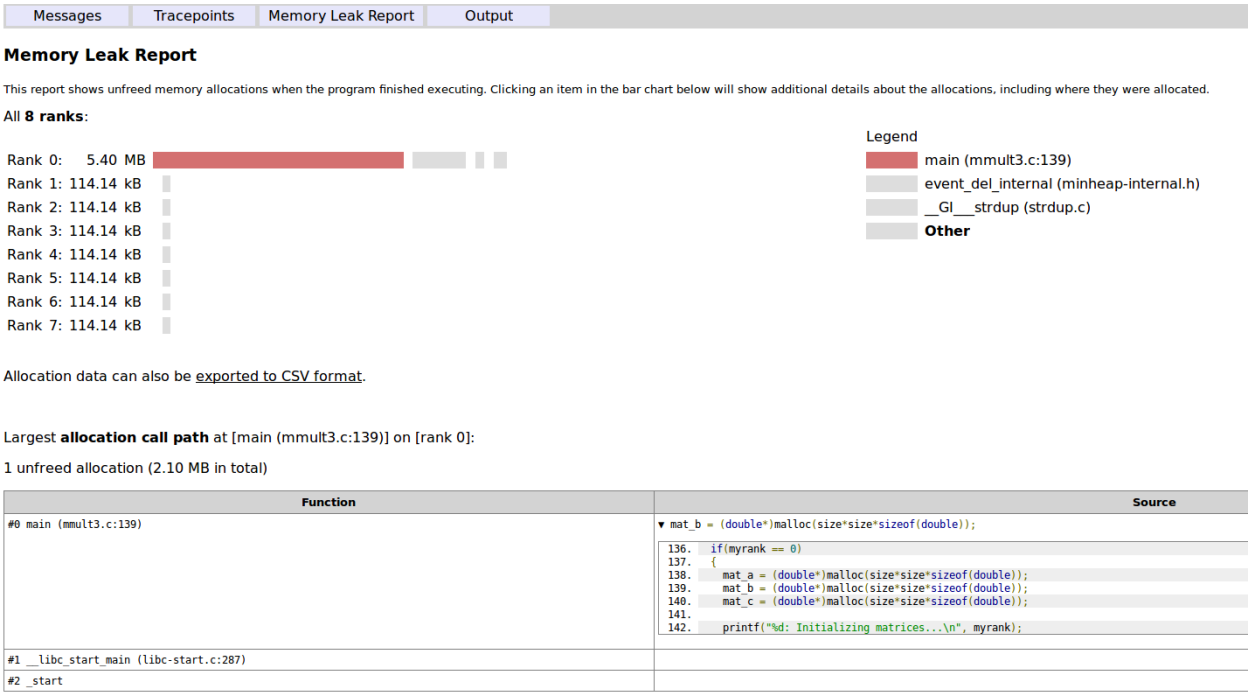
The **Memory Leak Report** section displays a graphical representation of the largest memory allocations that were not freed by the end of the program.

Note: The memory leak report will only contain data if the program completed successfully.



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:



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 `FORGE_LEAK_REPORT_MIN_SEGMENT` environment variable to a percentage. For example, `FORGE_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 can be exported to CSV format by clicking the export link.

Useful command line options:

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`)

2.12.2.4 Output

Output from the program 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, which processes have generated the output will be reported.

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

2.12.3 Offline report plain text output

Unlike the offline HTML report, the plain text report does not separate the tracepoint, breakpoint, memory leak, and program output into separate sections.

Lines in the offline plain text report are identified as messages, standard output, error output, and tracepoints, as described in *Offline report HTML output*.

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
```

2.12.4 Run-time job progress reporting

In offline mode, you can 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 when 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 front-end.

2.12.4.1 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
```

2.12.4.2 Signal-triggered snapshots

Snapshots can also be triggered by sending a SIGUSR1 signal to the front-end process (called `forge.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 forge.bin
> 18032
> 18039

# Use pstree to identify the parent if there are multiple PIDs:
pstree -p

# Trigger the snapshot:
kill -SIGUSR1 18032
```

CHAPTER

THREE

MAP

3.1 Get started with MAP

Learn how to get started using Linaro MAP.

3.1.1 Welcome page

Linaro MAP is a source-level profiler that can show how much time was spent on each line of code. To see the source code in Linaro MAP, compile your program with the debug flag. For most compilers this is `-g`. Do not use a debug build, you should always keep optimization flags turned on when profiling.

You can also use Linaro 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 Linaro MAP 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 Linaro MAP profiling run. It contains the program name and has a '.map' extension.

Note: Please add the Linaro Forge installation path to the `PATH` environment variable to make the Linaro MAP available to the console:

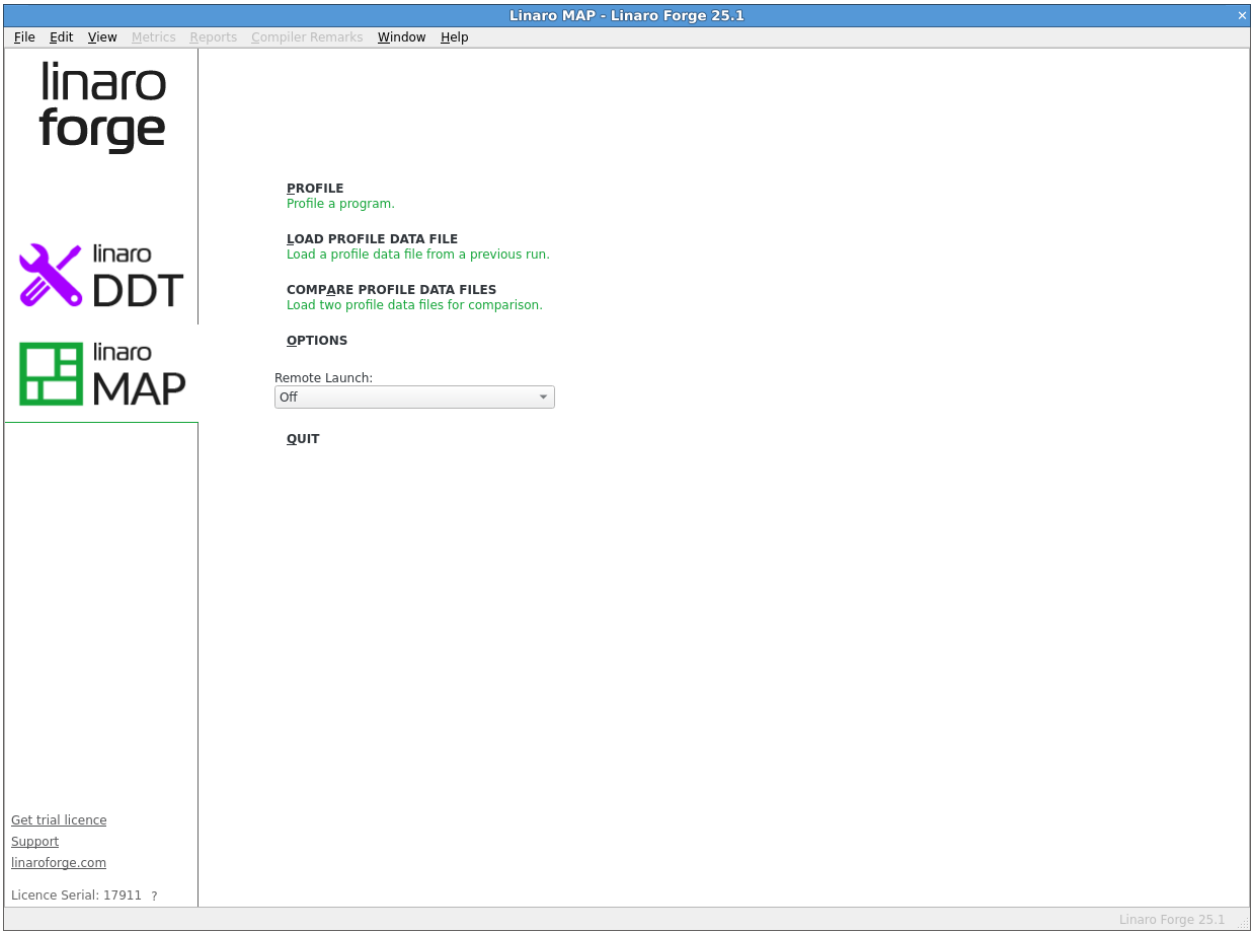
```
export PATH=<Forge installation path>/bin:$PATH
```

Notes:

- When starting Linaro MAP to examine an existing profile file, a valid license is not needed.
- Unless you are using Express Launch mode (see *Express Launch (MAP)*), you should not attempt to pipe input directly to Linaro MAP. For information about how to achieve the effect of sending input to your program, see *Program input and output*.

Linaro recommend that you start Linaro MAP with `--profile`. This runs without the interactive user interface and saves a `.map` file to the current directory, so is ideal for profiling jobs submitted to a queue.

When started in interactive mode, the *Welcome* page displays:



The *Welcome* page enables you to choose the kind of profiling you want to do, for example you can:

- Profile a program.
- Load a profile from a previous run.
- Load two profiles for comparison, see *Compare MAP Profiles*.
- Connect to a remote system and accept a Reverse Connect request.

Note: In Express Launch mode (see *Express Launch (MAP)*) the *Welcome* page is not shown and you will be brought directly to the **Run** dialog instead. If no valid license is found, the program exits, and a message is shown in the console output.

3.1.2 Express Launch (MAP)

All of the Linaro Forge products can be launched by typing their 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.

The MPI implementations supported by Express Launch are:

- Bullx MPI
- Cray X-Series (MPI/SHMEM/CAF)
- Intel MPI
- MPICH 3
- MPICH 4
- Open MPI (MPI/SHMEM)
- Open MPI (Cray XT/XE/XK)
- Cray XT/XE/XK (UPC)
- SLURM (MPMD)

If your MPI is not supported by Express Launch, an error message will display:

```
$ 'Generic' MPI 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.
```

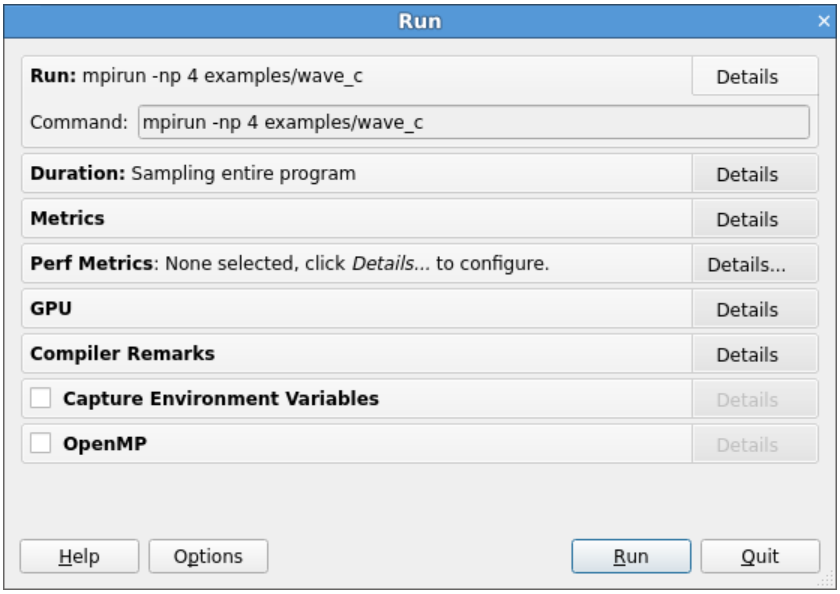
This is referred to as *Compatibility Mode*, in which the `mpiexec` command is not included and the arguments to `mpiexec` are passed via a `--mpiargs="args here"` parameter.

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 Linaro Forge products. This works best for MAP, which gathers data without an interactive user interface (`map --profile`) or Reverse Connect (`map --connect`, see *Reverse Connect* for more details) for interactive profiling.

If you cannot use Reverse Connect and want to use interactive profiling from a queue, you might need to configure Linaro MAP to generate job submission scripts for you. More details on this can be found in *Start a job in a queue (MAP)* and *Integration with queuing systems*.

3.1.2.1 Run dialog box

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



3.1.3 Prepare a program for profiling

In most cases, if your program is already compiled with debugging symbols (-g), you do not need to recompile your program to use it with MAP. However, in some cases it might need to be relinked, as explained in *Linking*.

Typically you should keep optimization flags enabled when profiling (rather than profiling a debug build). This will give more representative results.

The recommended set of compilation flags are:

- Arm Compiler for Linux: `-g1 -O3 -fno-inline-functions -fno-optimize-sibling-calls`
- Cray Fortran: `-G2 -O3 -h ipa0`
- Cray Clang C and C++: `-g1 -O3 -fno-inline-functions -fno-optimize-sibling-calls`
- GNU: `-g1 -O3 -fno-inline -fno-optimize-sibling-calls`
- Intel: `-debug minimal -O3 -fno-inline -no-ip -no-ipo -fno-omit-frame-pointer -fno-optimize-sibling-calls`
- NVIDIA HPC: `-g -O3 -Meh_frame -Mnoautoinline`

These flags preserve most performance optimizations whilst enabling file and line number information and maximizing stack trace readability by disabling features that might prevent MAP from obtaining stack traces (such as function inlining and tail call optimization). Minimal debug information is also used to reduce memory usage during profiling.

3.1.3.1 Debug symbols

If your compiler supports *minimal* debug info, consider using it (for file and line number information only) instead of full debug info. For GCC, Arm® Compiler for Linux, and Intel, this means using `-g1` for compiling instead of `-g`.

Although this can cause inlined functions not to be shown in profiles, it can significantly reduce the memory overhead when profiling.

This is particularly relevant for complex C++ codes, memory-constrained compute nodes, or when profiling many processes per node.

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.

For some compilers, it is necessary to explicitly enable frame pointer information to ensure stack traces, particularly when debug information has been disabled. This is normally done with `-fno-omit-frame-pointer` (or `-Meh_frame` for NVIDIA HPC).

Cray compiler

For the Cray compiler we recommend using the `-G2` option.

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
```

3.1.3.2 Disable function inlining

While compilers can inline functions, their ability to include sufficient information to reconstruct the original call tree varies between vendors and is not possible if compiling your program with minimal debug info (file & line info only) or without debug info.

To maximize readability of call trees, we recommend that you disable function inlining using the appropriate compiler-specific flags (see *Prepare a program for profiling*).

Note: Some compilers might still inline functions even when they are explicitly instructed not to do so.

There is typically a small performance penalty for disabling function inlining or enabling profiling information.

3.1.3.3 Disable tail call optimization

A function can return the result of calling another function, for example:

```
int someFunction()
{
   ...
   return otherFunction();
}
```

In this case, the compiler can 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, called tail recursion optimization, can be disabled by passing the `-fno-optimize-sibling-calls` argument to most compilers.

3.1.3.4 Linking

To collect data from your program, MAP uses two small profiler libraries, `map-sampler` and `map-sampler-pmpi`. These profiler libraries 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 Linaro MAP and Linaro Performance Reports.

This automatic linking when you start your program only works if your program is dynamically-linked. Programs can be dynamically-linked or statically-linked. 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 Linaro 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 *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 Linaro MAP warns you that it could not pre-load the Linaro Forge 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:

- 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 program at runtime.
- Link MAP's `map-sampler` and `map-sampler-pmpi` libraries with your program at link time manually.

  See *Dynamic linking on Cray X-Series systems*, or *Static linking* and *Static linking on Cray X-Series systems*.
- 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.

3.1.3.5 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.

Compile the MPI Wrapper Library

1. Compile the 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.
```

2. Link with the MPI wrapper library

```
mpicc -G2 -o hello hello.c -dynamic -L/home/user/myprogram \
      -lmap-sampler-pmpi -lmap-sampler -Wl,--eh-frame-hdr
```

NVIDIA HPC Compiler

When linking OpenMP programs, you must pass the `-Bdynamic` command line argument to the compiler when linking dynamically

3.1.3.6 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, you must explicitly link your program with the Linaro Forge sampler and MPI wrapper libraries.

Compile the MPI Wrapper Library

1. Compile the 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 Forge sampler and MPI wrapper libraries in
      the link line.  There's a comprehensive description of the link ordering
      requirements in the 'Prepare a Program for Profiling' section of
      userguide-forge.pdf, located in /opt/linaro/forge/x.y.z/doc/.
```

2. Link with the 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 Linaro Forge sampler. Usually this is sufficient, but not in all cases. The rest of this section explains how to manually add the Linaro Forge sampler to your link line.

NVIDIA HPC Compiler

The NVIDIA HPC C runtime static library contains an undefined reference to `__kmpc_fork_call`. This causes compilation to fail when linking allinea-profiler.ld. Add `--undefined __wrap___kmpc_fork_call` to your link line before linking to the Linaro Forge sampler.

Cray

When linking C++ programs you might encounter a conflict between the Cray C++ runtime and the GNU C++ runtime used by the Linaro 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.

When linking your program you might encounter undefined references similar to the ones below:

```
ld.lld: error: undefined symbol: pstart_pes
ld.lld: error: undefined symbol: pshmem_init
ld.lld: error: undefined symbol: p_my_pe
ld.lld: error: undefined symbol: pshmem_barrier_all
ld.lld: error: undefined symbol: pshmem_finalize
```

You can resolve this by ensuring that the `cray-openshmemx` and `cray-pmi` modules are loaded.

-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 Linaro Forge sampler library `-lmap-sampler`.

The Linaro Forge sampler library `-lmap-sampler` must go:

1. *After* the 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/linaro \
    -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/linaro \
    -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
```

3.1.3.7 Static linking on Cray X-Series systems

Compile the MPI Wrapper Library

1. On Cray X-Series systems, you can compile the MPI wrapper library using `make-profiler-libraries --platform=cray --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 (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 Forge sampler and MPI wrapper libraries in
   the link line.  There's a comprehensive description of the link ordering
   requirements in the 'Prepare a Program for Profiling' section of
   userguide-forge.pdf, located in /opt/linaro/forge/x.y.z/doc/.
```

2. Link with the MPI wrapper library using:

```
cc hello.c -o hello -g -Wl,@allinea-profiler.ld

ftn hello.f90 -o hello -g -Wl,@allinea-profiler.ld
```

3.1.3.8 Dynamic and static linking on Cray X-Series systems using the modules environment

If your system has the Linaro Forge module files installed, you can load them and build your program as usual. See *map-link modules installation on Cray X-Series*.

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.

3.1.3.9 map-link modules installation on Cray X-Series

To facilitate dynamic and static linking of user programs with the MPI wrapper library and Linaro Forge 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 Linaro 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, `forge/map-link-static`. The correct prefix depends on the subdirectory (if any) under the module search path the `map-link-*` module files 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-dynamic` and `map-link-static` modules should match the prefix set in the **conflict** line of the module sources.

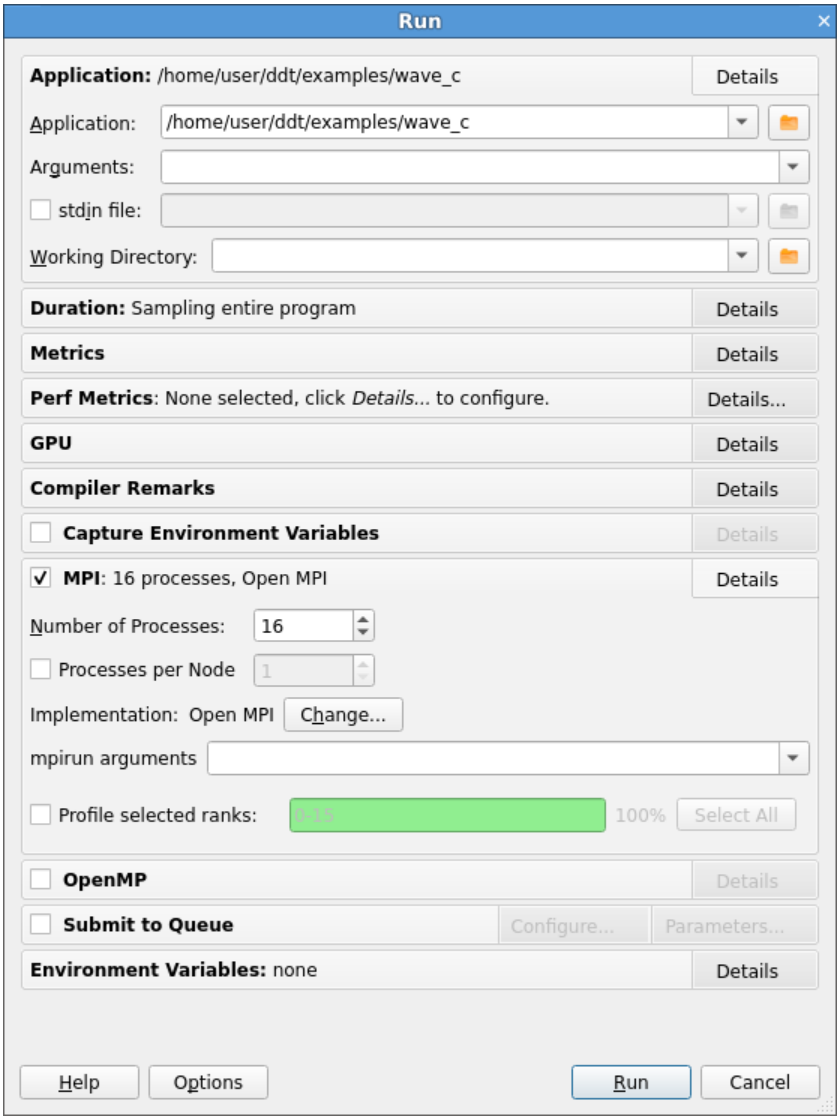
3.1.3.10 Unsupported user applications

Ensure that the program to be profiled does not set or unset the SIGPROF signal handler. This interferes with the MAP profiling function and can cause it to fail.

We recommend that you do not use Linaro 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.

3.1.4 Profile a program

When you click *Profile* on the Welcome page, the *Run* window displays.



The settings are grouped into sections. Click *Details* to expand a section.

3.1.4.1 Application

Application:
The full path name to your application. If you specified one on the command line, this is automatically filled in. You can browse and select your application.

Note: Many MPIs have problems working with directory and program names that contain spaces. We recommend that you do not use 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 MAP and your command shell. If you must use these characters but cannot get them to work as expected, contact Forge Support.

---

stdin file (optional):
This enables you to choose a file to be used as the standard input (`stdin`) for your program. Arguments are automatically added to `mpirun` to ensure your input file is used.

Working Directory (optional):
The working directory to use when running your program. If this is blank then MAP's working directory will be used instead.

3.1.4.2 Duration

Start profiling after (optional):
This enables you to delay profiling by a number of seconds into the run of your program.

Stop profiling after (optional):
This enables you to specify a number of seconds after which the profiler will terminate your program.

3.1.4.3 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 the **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 definitions, the previous user interface session, and those specified with the `--enable-metrics` and `--disable-metrics` command-line options. The command-line options take preference over the previous user interface session settings, and both take preference over the metric XML definitions settings. Of course, metrics that are always enabled or always disabled cannot be toggled.

All *CPU instructions* displays if available for enabling/disabling.

3.1.4.4 Capture Environment Variables

This option enables the feature to collect and view the Environment Variables set at the point of launch in the resulting profile.

See *Program details* for more information.

3.1.4.5 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 *Profile a single-process program* for more details about using a single process.

Number of Processes:
The number of processes that you want to profile. MAP supports hundreds of thousands of processes, but this is limited by your license. This option might not be displayed if it is disabled on the *Job Submission* page in the *Options* window.

Number of nodes:
This is the number of compute nodes that you want to use to run your program. This option is only displayed for certain MPI implementations, or if it is enabled on the *Job Submission* page in the *Options* window.

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* page in the *Options* window.

Implementation:
The MPI implementation to use, for example, Open MPI, MPICH 3. 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. Click *Change* to change the MPI implementation.

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 which to pick, try generic, consult your system administrator, or Forge Support. A list of settings for common implementations is provided in *MPI distribution notes and known issues*.

Note: If the MPI command you want is not in your `PATH`, or you want to use an MPI run command that is not your default one, you can configure this using the **Options** window. See *Optional configuration*.

---

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 field can be left empty.

Note: You should not enter the `-np` argument because MAP will do this for you.

Profile selected ranks (optional):
If you do not want to profile all the ranks, you can use the `--select-ranks` command-line option to

specify a set of ranks to profile. The ranks should be separated by commas, and intervals are accepted. Example: `5,6-10`.

Note: This option acts as a filter for which processes collects and uses profiling data. Your license will still be limited to the number of processes you are launching and is not affected by the selected ranks.

3.1.4.6 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.

3.1.4.7 Environment variable

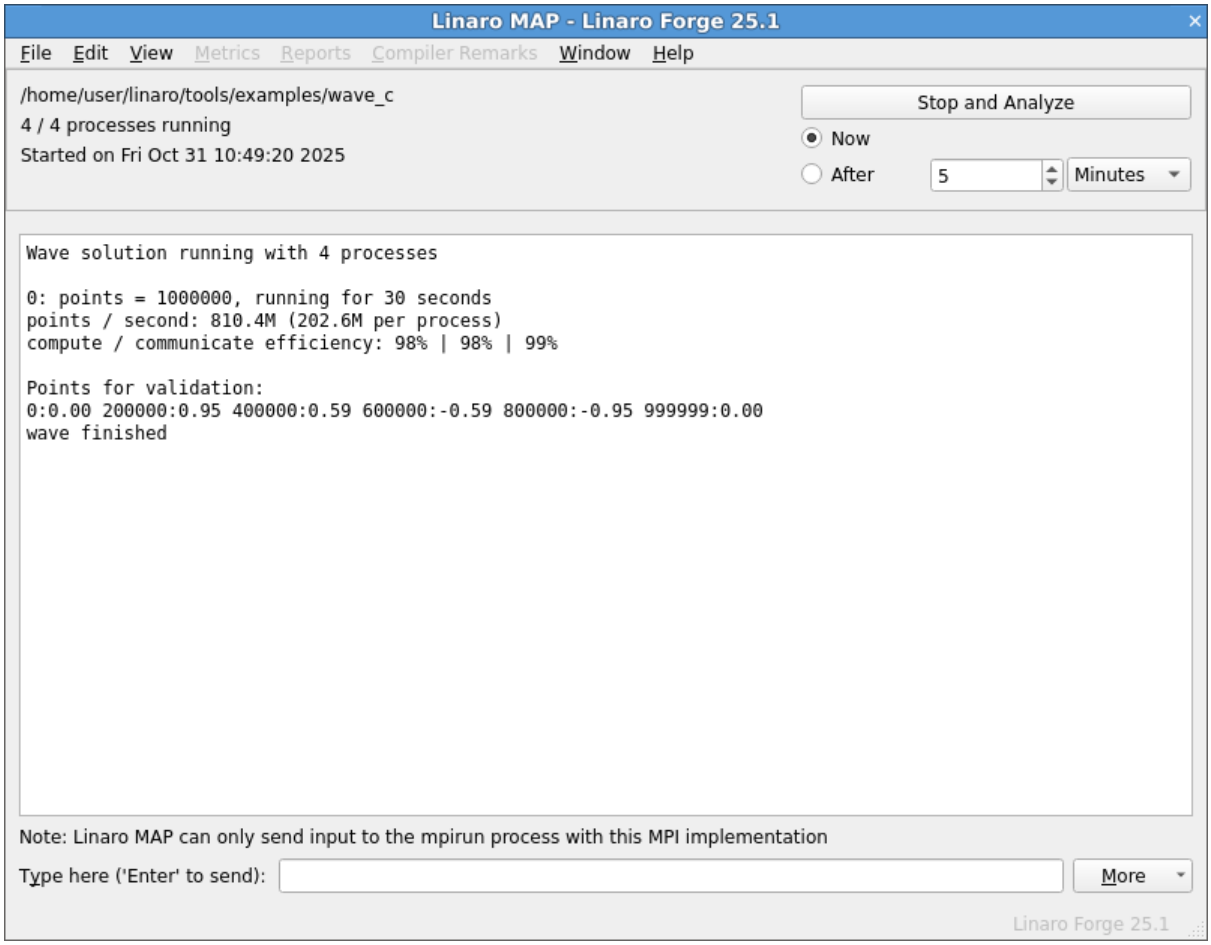
The optional *Environment Variables* section should contain additional environment variables that should be passed to mpirun or its equivalent. These environment variables may also be passed to your program, depending on which MPI implementation your system uses. Most users will not need to use this section.

3.1.4.8 Profiling

Click *Run* to start your program, or *Submit* if working through a queue. See *Integration with queuing systems*. This will compile an MPI wrapper library that can intercept the `MPI_INIT` call and gather statistics about MPI use in your program. If you have problems, see *MPI wrapper libraries*. Then Linaro 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.

Linaro MAP collects data for the entire program run by default. Its 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.



3.1.4.9 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` and `--stop-after` command-line options. These options enable you to specify a range of wall-clock time (the job starts at 0 seconds) during which the job should be profiled. When 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 can choose to start profiling programmatically at a later point by instrumenting your code. To do this you must set the `FORGE_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 can 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 can only delay the start of sampling and stop sampling early.

C

To start sampling programmatically you should `#include "mapsampler_api.h"` and call the `allinea_start_sampling` function. You must point your C compiler at the Linaro MAP include direc-

tory, by passing the arguments `-I /path/to/forge/map/wrapper` and also link with the Linaro Forge sampler library, by passing the arguments `-L /path/to/forge/lib/64 -lmap-sampler`.

To stop sampling programmatically call the `allinea_stop_sampling` function.

Fortran

To start sampling programmatically call the `ALLINEA_START_SAMPLING` subroutine. You must also link with the Linaro Forge sampler library, for example by passing the arguments `-L /path/to/forge/lib/64 -lmap-sampler`.

To stop sampling programmatically call the `ALLINEA_STOP_SAMPLING` subroutine.

Python

To start sampling programmatically you should import the mapsampler_api module methods (`from mapsampler_api import *`) and call the `allinea_start_sampling` function.

To stop sampling programmatically call the `allinea_stop_sampling` function.

3.1.5 `remote-exec` required by some MPIs (MAP)

When using the MPMD variants of *MPICH 3* or *Intel MPI*, you can use mpirun to start all the processes, then attach to them while they are inside `MPI_Init`.

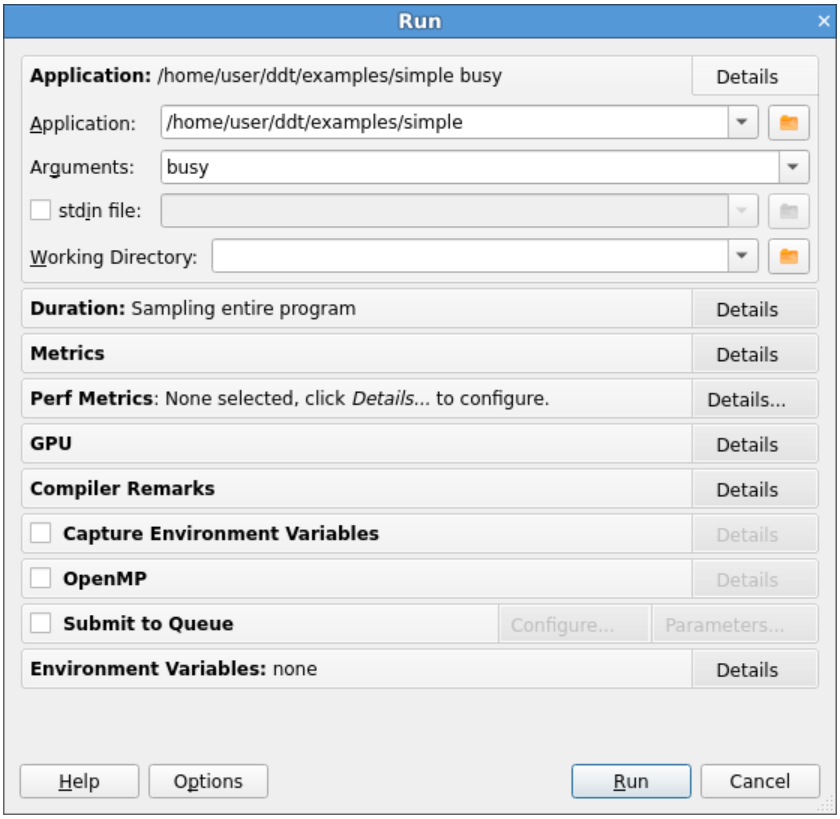
This method is often faster than the generic method, but requires the `remote-exec` facility to be correctly configured if processes are being launched on a remote machine. For more information on `remote-exec`, see *Connecting to compute nodes and remote programs (remote-exec)*.

Note: If MAP is running in the background, for example using `map &`, this process might 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, you will not be able to use the specialized startup options. Instead, you can use *generic*, although startup might be slower for large numbers of processes.

3.1.6 Profile a single-process program

If you have a single-process license you will immediately see the *Run* dialog that is appropriate to run a single-process program.

If you have a multi-process license you can clear the *MPI* checkbox to run a single-process program.



1. Type the full file path to your application, or browse and select your application.
2. If required, type the arguments to pass to your program.
3. If required, select the *OpenMP* checkbox and select the *Number of OpenMP threads* to start your program with.
4. Click *Run* to start your program.

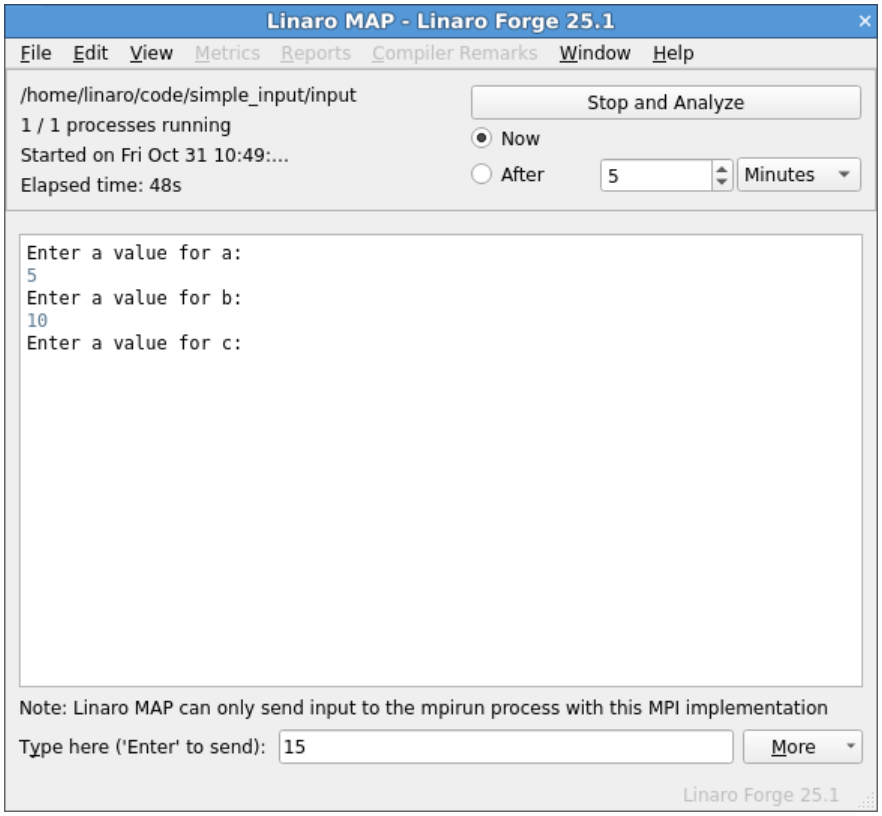
3.1.7 Send standard input (MAP)

You can use the *stdin* field in the *Run* window to choose the file to use as the standard input (stdin) for your program. Arguments will be automatically added to mpirun to ensure your input file is used.

Alternatively, you can enter the arguments directly in the *mpirun Arguments* field. 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 can add the same options to the *mpirun Arguments* field when you start your session in the *Run* window.

It is also possible to enter input during a session. To do this, start your program as normal, then open the *Input/Output* tab. Type the input you want to send.

Click *More* to send input from a file, or send an EOF character.



3.1.8 Start a job in a queue (MAP)

If Linaro MAP is configured to integrate with a queue/batch environment, you can use Linaro MAP to submit your job directly from the user interface.

For details see *Integration with queuing systems*.

In this case, a *Submit* button is displayed on the *Run* window, instead of a *Run* button. When you click *Submit* on the *Run* window the queue status is displayed 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 you cannot use it here.

If your job does not start or you decide not to run it, click *Cancel Job*. If the regular expression you entered for getting the job id is invalid, or if an error is reported, MAP will not be able to remove your job from the queue. In this case we strongly recommend 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 profiling sessions.

After the sampling (program run) phase is complete, the analysis phase starts, collecting and processing the distinct samples. This can be a lengthy process depending on the size of the program. For very large programs, it could take up to 10 or 20 minutes.

You must ensure that your job does not hit its queue limits during the analysis process. Set the job time large enough to cover both the sampling and analysis phases.

Linaro MAP also requires extra memory, both in the sampling and in the analysis phases. If these fail and your program alone approaches one of these limits, you might need to run with fewer processes per node or a smaller data set, to generate a complete set of data.

When your job is running, it connects to Linaro MAP and you can profile it.

3.1.9 Use custom MPI scripts (MAP)

On some systems, a custom mpirun replacement is used to start jobs, such as mpiexec. MAP typically uses whatever the default for your MPI implementation is, so for Open MPI it would look for mpirun and not mpiexec, for SLURM it would use `srun`, and so on. Here we explain how to configure MAP to use a custom mpirun command for job startup.

MAP supports two ways that you can use to start jobs using a custom script.

In the first way, you pass all the arguments on the command-line, like this:

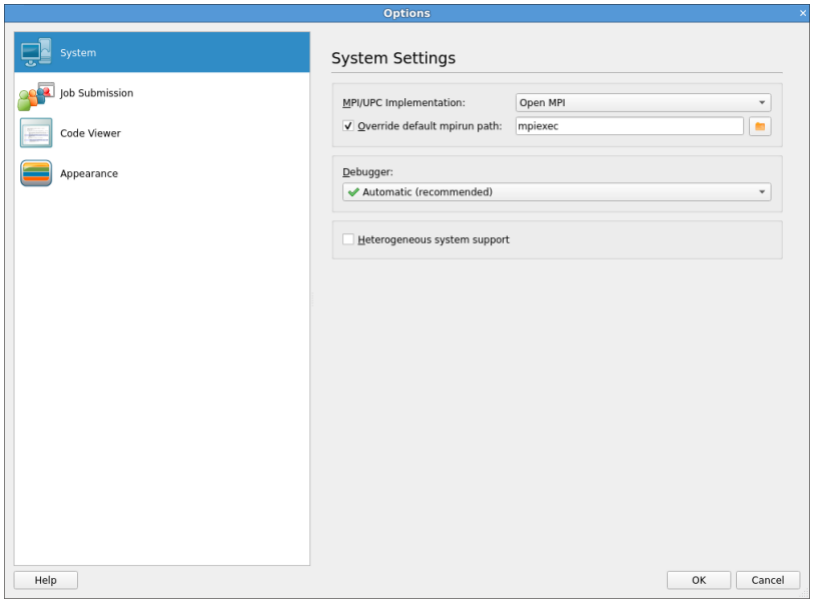
```
mpiexec -n 4 /home/<user>/program/chains.exe /tmp/mydata
```

There are several key variables in this command that MAP can complete for you:

- The number of processes (4 in the above example).
- The name of your program (`/home/<user>/program/chains.exe`).
- 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, you adapt the queue submission system described in *Start a job in a queue (MAP)*. For this mpiexec example, the settings are :



As you can see, most of the settings are left blank.

There are some differences between the *Submit command* in Linaro MAP and what you would type at the command-line:

- The number of processes is replaced with `NUM_PROCS_TAG`.
- The name of the program is replaced by the full path to `forge-backend`, used by both DDT and MAP.
- 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 `forge-backend` instead of your program, but with the same options.

In the second way, you start a job using a custom mpirun replacement with a settings file:

```
mpiexec -config /home/<user>/myapp.nodespec
```

Where `myfile.nodespec` might contain something similar to the following:

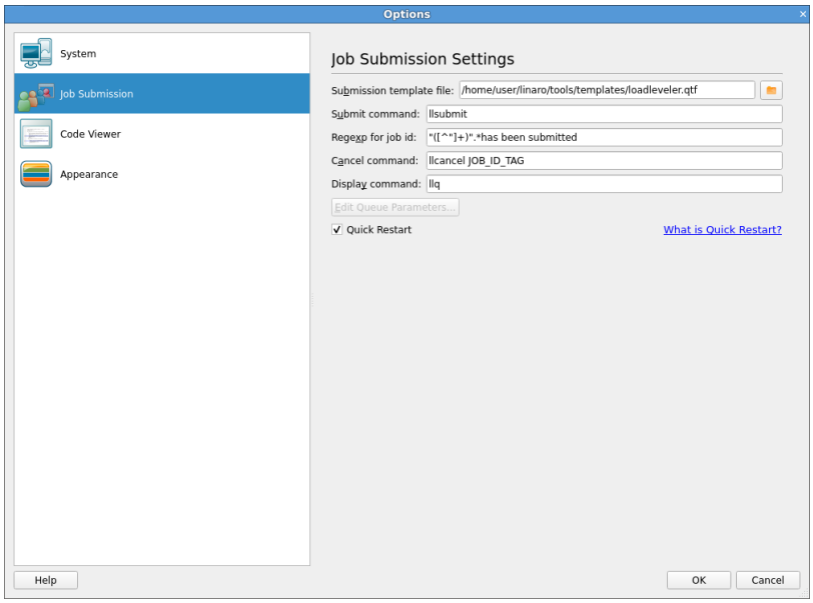
```
comp00 comp01 comp02 comp03 : /home/<user>/program/chains.exe /tmp/mydata
```

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

```
comp00 comp01 comp02 comp03 : DDTPATH_TAG/libexec/forge-backend DDT_DEBUGGER_ARGUMENTS_TAG PROGRAM_
↪ARGUMENTS_TAG
```

This follows the same replacement rules described above and in detail in *Integration with queuing systems*.

The settings in the *Options* window for this example might be:



Note *Submit command* and *Submission template file* in particular. Linaro MAP creates a new file and appends it to the submit command before executing it. In this case what would actually be executed might be `mpiexec -config /tmp/linaro-temp-0112` or similar. Therefore, any argument like `-config` must be last on the line, because Linaro MAP will add a file name to the end of the line. Other arguments, if there are any, can come first.

Linaro recommend that you read the section on queue submission, because 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, contact Forge Support.

3.1.10 Start MAP from a job script

When debugging it is common to submit runs from inside a debugger. When profiling the usual approach is to run the program offline, producing a profile file that can be inspected later.

To do this, replace your usual program invocation such as:

```
mpirun -n 4 PROGRAM [ARGUMENTS]...
```

With a Linaro MAP command such as:

```
map --profile mpirun -n 4 PROGRAM [ARGUMENTS]...
```

```
map --profile --np=4 PROGRAM [ARGUMENTS]...
```

Linaro MAP runs without a user interface, gathering data to a `.map` profile file. Its filename is based on a combination of program name, process count, and timestamp, such as `program_2p_2012-12-19_10-51.map`.

If you are using OpenMP, the value of `OMP_NUM_THREADS` is also included in the name after the process count, such as `program_2p_8t_2014-10-21_12-45.map`.

This default name can be changed with the `--output` argument. To examine this file, either run MAP and select *Load Profile Data File*, or access it directly with the command:

```
map program_2p_2012-12-19_10-51.map
```

Note: When you start MAP to examine an existing profile file, a valid license is not needed.

When running without a user interface, 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 *Reverse Connect*) to connect back to the user interface 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]...
```

3.1.11 Numactl (MAP)

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 (MAP)*.

```
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. When using compatibility mode, you must specify the full path to `numactl` in the *Application* field of the *Run* dialog . You can find the full path by running:

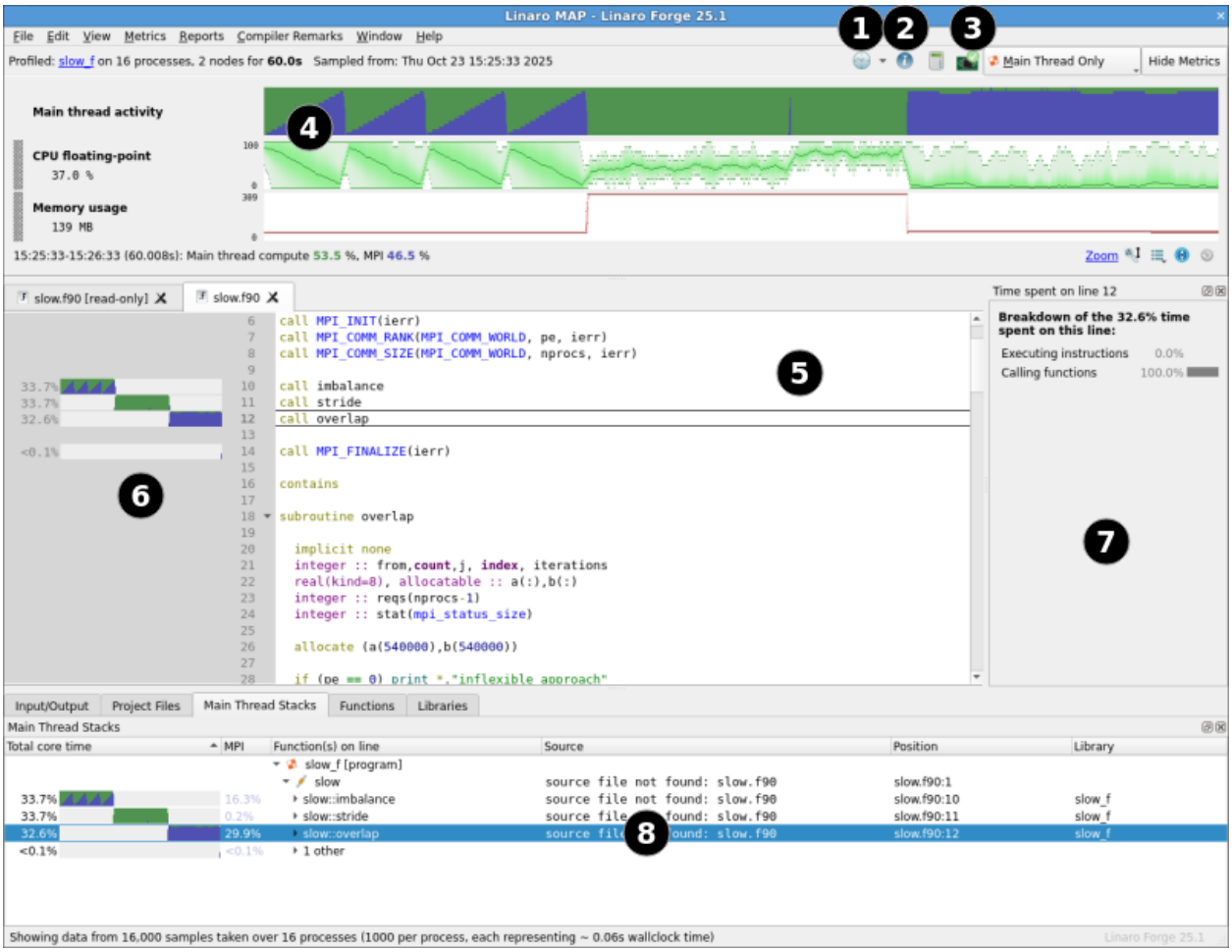
```
which numactl
```

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.

3.2 MAP user interface

Linaro MAP uses a tabbed-document interface to display multiple documents. This means you can 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 Linaro MAP is a dockable window that you can drag around by a handle, usually on the top or left edge. You can also double-click or drag a component outside of Linaro MAP, to form a new window. You can hide or show most of the components using the *View* menu. You can also select preset or custom metrics displays. The screenshot shows the default layout.



This table shows the main components:

Key	Component
1	*Time display toggle button*
2	*Program details*
3	*Thread affinity advisor*
4	*Metrics view*
5	*Source Code view*
6	*Sparkline charts*
7	*Selected Lines view*
8	*Input/Output values*, *Project Files list*, *Stacks view*, *Functions view*, *Library view*

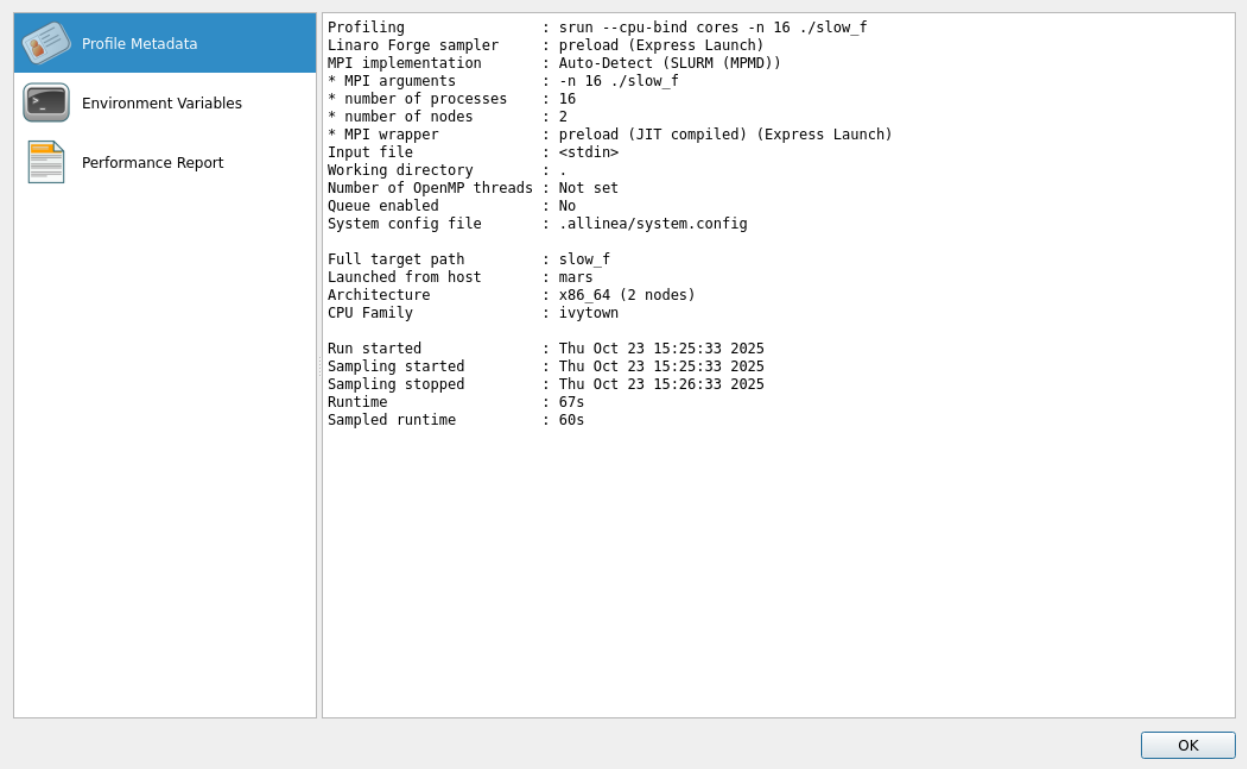
Note: On some platforms, the default screen size might be insufficient to display the status bar. If this occurs, expand the Linaro MAP window until it is completely visible.

3.3 Program details

3.3.1 Profile Metadata

Linaro MAP displays profile metadata in the *Program details* dialog, including:

- Command line
- MPI implementation
- Input file
- Full target path
- Run started timestamp
- Runtime and Sampled Runtime



3.3.2 Environment Variables

View the launch environment used when generating the profile by selecting *Environment Variables*.

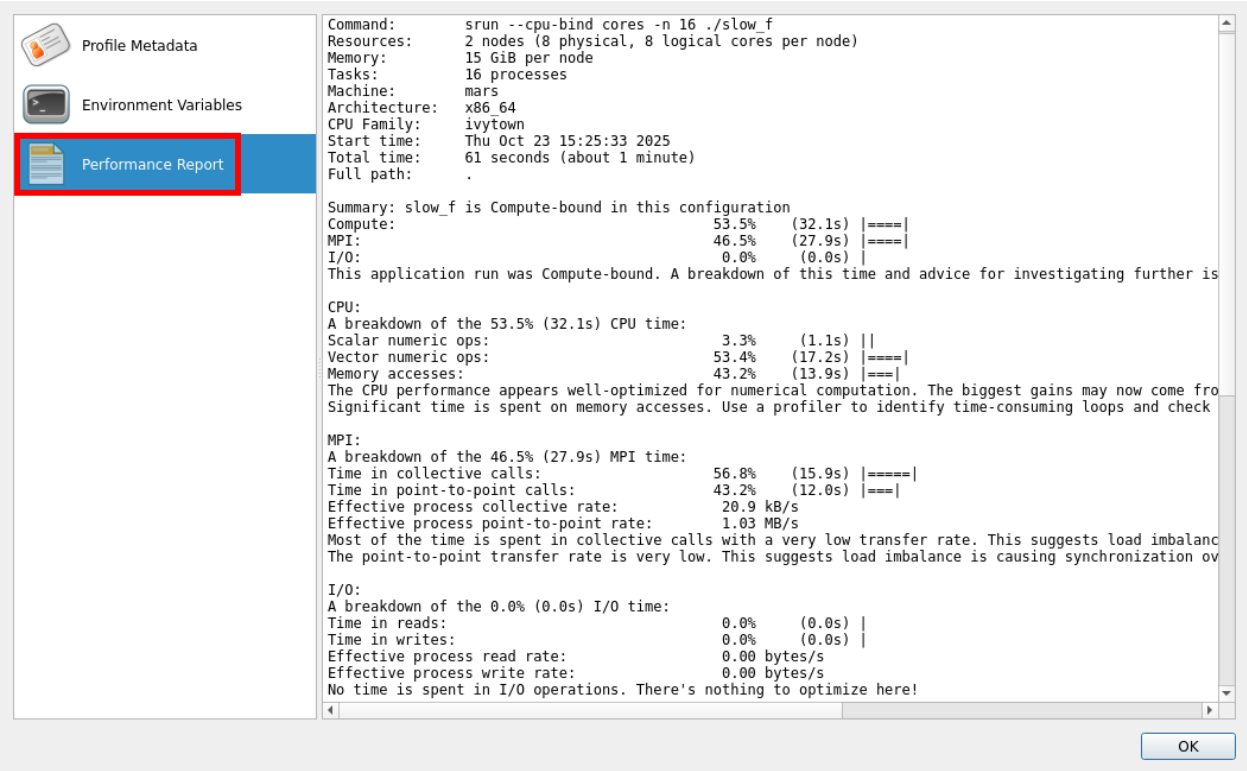
The collection of Environment Variables is disabled by default. To enable *Environment Variables* in the *Program details* dialog, enable it either by using the `Capture Environment Variables` option in the Run Dialog or by using the command line argument `--capture-environment-variables`.

Note: If you are profiling using `--profile`, the `--capture-environment-variables` command line argument is always required to enable this feature.

Note: Enabling this feature will capture all variables for your applications environment and store them in the Linaro MAP file. For security reasons, consider what information is stored in the environment before enabling this option.

3.3.3 Performance Report

Select *Performance Report* to view the profile in Linaro Performance Reports textual format.



3.4 Time display mode

3.4.1 Time display toggle button

Use the *Time display toggle button* to switch between displaying time values in Linaro MAP as a percentage of profiled runtime, or total core-time.

Note: Core-time is estimated by multiplying the average sampling interval by the total number of samples representing any particular measurement across all threads and processes.

The accuracy of this estimate is dependent on the average sampling interval, which is proportional to the total profiling runtime, but with a lower bound dictated by the number of threads per process. In general, a higher profiling runtime leads to a higher average sampling interval. The average sampling interval of a profile is displayed in the status bar of Linaro MAP.

For more information on the average sampling interval in Linaro MAP, see `FORGE_SAMPLER_INTERVAL` and `FORGE_SAMPLER_INTERVAL_PER_THREAD`.

It is important to keep in mind these limitations when comparing small-percentage differences in core-time between different Linaro MAP profiles. Differences in core-time may be negligible when considering the differences in average sampling intervals.

When displaying core-time, use the dropdown menu to switch units in which core-time is displayed. Linaro MAP chooses units to compactly display each value if *Auto* is selected.

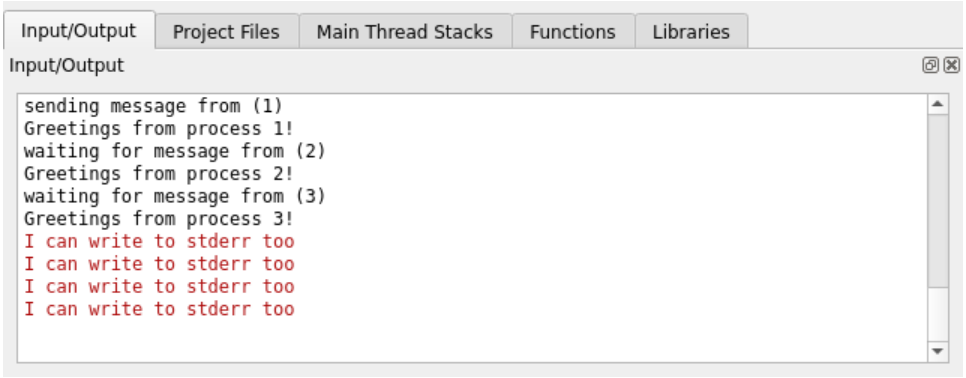
3.5 Program output

Linaro MAP collects and displays output from all processes on 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 Linaro DDT.

3.5.1 View standard output and error

The *Input/Output* tab is at the bottom of the window by default.



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

3.5.2 Restrict 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 `FORGE_KEEP_OUTPUT_LINES` set to the preferred total line limit (`FORGE_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 `FORGE_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.

3.5.3 Save output

Right-click on the text to either save it to a file or copy a selection to the clipboard.

3.6 Source code (MAP)

Linaro MAP provides source 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 source code editing and rebuilding capabilities are not designed for developing programs 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 you are running remotely (using the remote client) or are connected directly to your system.

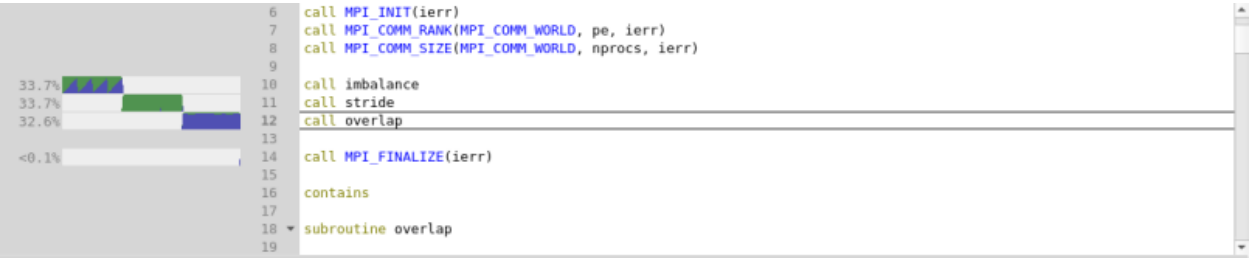
3.6.1 View source code (MAP)

Source and header files found in the executable are reconciled with the files present on the front-end server, and are displayed in a tree view in the *Project Files* tab of the **Project Navigator** window.

Click on the file name to view the source file.

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. Click the '+' glyph to expand the function again.



The center pane shows your source code, annotated with performance information. All the charts you see in Linaro MAP share a common horizontal time axis. Your job starts from the left and ends on the right. Next to each line of source code are the sparkline charts. The sparkline charts show 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 viewer**, Linaro MAP uses *inclusive* time, that is, time spent on this line of code or inside functions called by this line. The `main()` function of a single-threaded C or MPI program is typically at 100% for the entire run.

Only 'interesting' lines generate charts, that is, lines in which at least 0.1% of the selected time range was spent. In the previous figure, three different lines meet this criterion. 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 runs for 30.4% of the wall-clock time. Looking closer, as well as a large block of green, there is a blue sawtooth pattern. Color identifies 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. Time spent inside MPI calls is blue, regardless of whether that is in `MPI_Send` or `MPI_Barrier`. Typically you want to minimize the amount of blue, because the purpose of most codes is parallel *computation*, not communication.
- **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. It is important to minimize the time spent in I/O. On many systems, the complex data storage hierarchy can cause unexpected bottlenecks to occur when scaling a code up. Linaro MAP always shows the time from the program's point of view, so all the underlying complexity is captured and represented as simply as possible.
- **Dark purple** Accelerator. The time the CPU is waiting for the accelerator to return the control to the CPU. Typically you want to minimize this time, to make 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 around 55% of its time in computation (dark green) and around 45% 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 is investigated later.
- Then, the program moves on to a function called `stride`, which spends almost all of its time computing. You will see how to tell whether this time is well spent or not. At the end, you can also see an MPI synchronization. The triangle shape is typical of ranks finishing their work at different times, and spending varying amounts of time waiting at a barrier. Triangles in these charts indicate 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.

Because this example 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. The sawtooth pattern showing MPI usage indicates that:

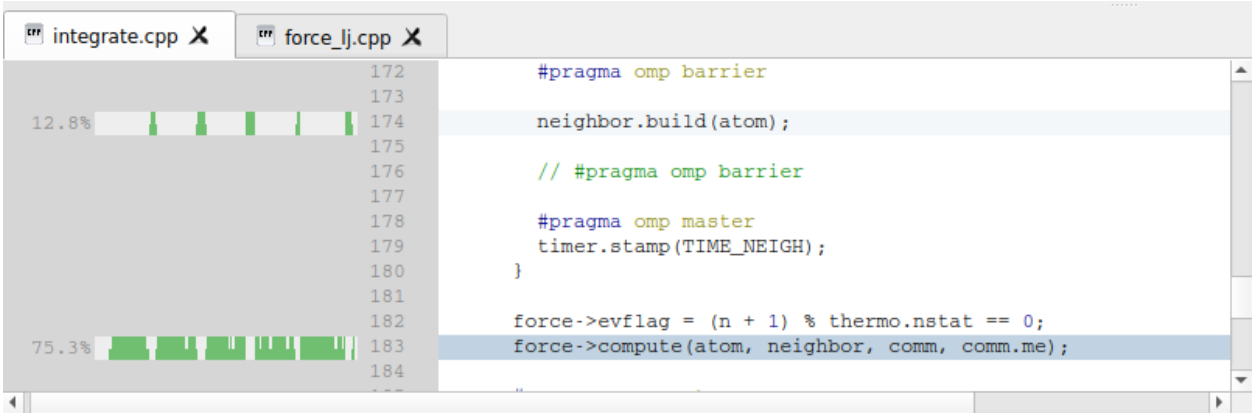
- The `imbalance` function goes through several iterations.
- In each iteration, all processes begin computing. There is more green than blue.
- As execution continues, more processes finish computing and transition to waiting in an MPI call. The transition causes the distinctive triangular sawtooth pattern which illustrates a workload imbalance.
- As each sawtooth pattern ends, all ranks finish communicating and the pattern begins again with the next iteration.

This is a classic sign of MPI imbalance. A sawtooth pattern in MAP graphs show that at first, only a few processes change to a different state of execution. Subsequently, more of these processes change until they all synchronize and move on to another state together. These are areas of interest that are worthwhile investigating.

To explore this example in more detail, open the `examples/slow.map` file and look at the `imbalance` function. Identify and investigate why some processes take longer to finish computing than others.

3.6.2 OpenMP programs

For an OpenMP or multi-threaded program (or a mixed-mode MPI+OpenMP program) you will also see other 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 good 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 12.8% of the time is spent calling `neighbor.build(atom)` and 75.3% 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. Linaro 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:

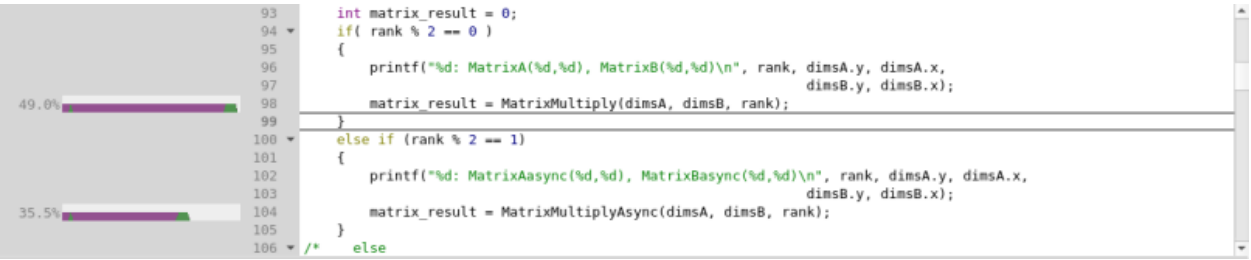
- A large section of blue in a mixed-mode MPI code means that all the processes in the program were inside MPI calls during this period. Try to reduce these, especially if they have a triangular shape suggesting

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. Linaro MAP only supports this if the OpenMP primary thread is the one that makes the MPI calls. In this case, the blue block of MPI time are smaller, demonstrating that one OpenMP thread is in an MPI function while the rest are doing something else, such as useful computation.

3.6.3 GPU programs

In a program using NVIDIA CUDA or AMD ROCm, CPU time spent waiting for GPU kernels to complete is shown in **Purple** in the **Source code viewer**.



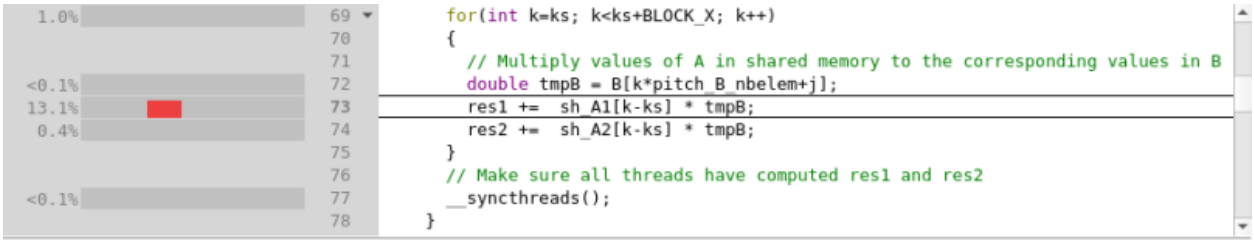
In this screenshot, a CPU was waiting for a GPU kernel to complete on the highlighted lines in the **Source code viewer**.

3.6.3.1 NVIDIA CUDA Kernel Analysis

When CUDA kernel analysis mode is enabled (see *NVIDIA GPU profiling*) Linaro MAP will 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 *CUDA Kernel analysis*. Refer to the tooltip or selected line display (*NVIDIA GPU CUDA profiles*) 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.

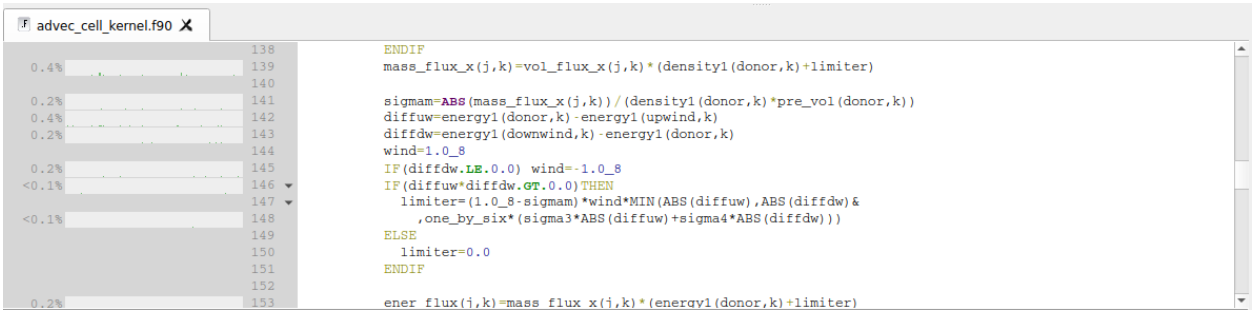
Note that warp stalls are only reported per-kernel, so it is not possible to obtain the times within a kernel invocation at which different categories of warp stalls occurred. As function calls in CUDA kernels are also automatically fully inlined it is not possible to see warp stalls for 'time spent inside function(s) on line' for GPU kernel code.



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 for more information on how warps are sampled.

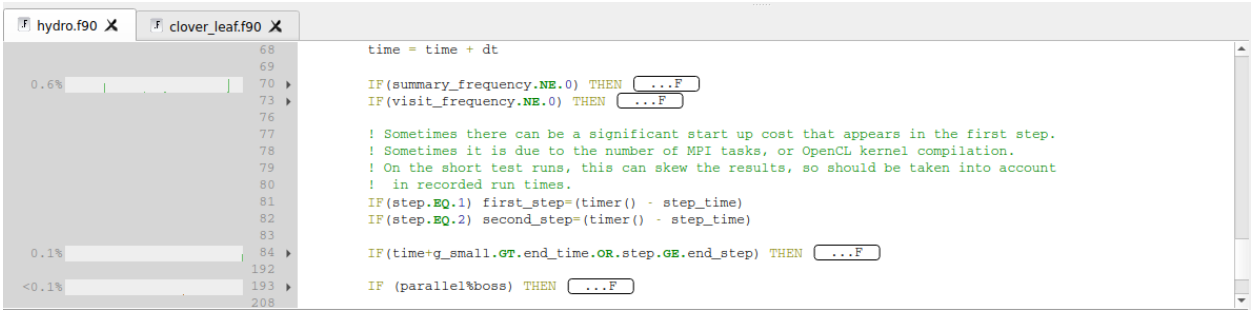
3.6.4 Complex code: code folding

Real-world scientific code does not look much like the examples above. It looks more like the following:



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, Linaro MAP allows *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:



This helps you identify the conditional blocks where most of the processing occurs.

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.

3.6.5 Edit source code (MAP)

Source code can be edited in the **Source code viewer**. 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 can be opened, saved, reverted, and closed from the *File* menu.

Note: Information from Linaro 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, you can open it by right-clicking in the editor and choosing *Open <filename>.<extension>*. There is a global shortcut on function key `F4`, or you can use *Edit* ▸ *Switch Header/Source*.

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, select *File* ▸ *Options* to open the *Options* window (*Linaro Forge* ▸ *Preferences* on Mac OS X) then enter the path to the preferred editor in *Editor*, for example `/usr/bin/gedit`.

If a file is edited, a warning will be displayed at the top of the editor.

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.

3.6.6 Rebuild and restart (MAP)

To configure the build command choose *File* ▸ *Configure Build*, enter a build command and a directory in which to run the command, then 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.

3.6.7 Commit changes (MAP)

Changes to source files can be committed using one of Git, Mercurial, or Subversion. To commit changes, choose *File* ▸ *Commit*, enter a commit message in the *Commit changes* dialog then click *Commit*.

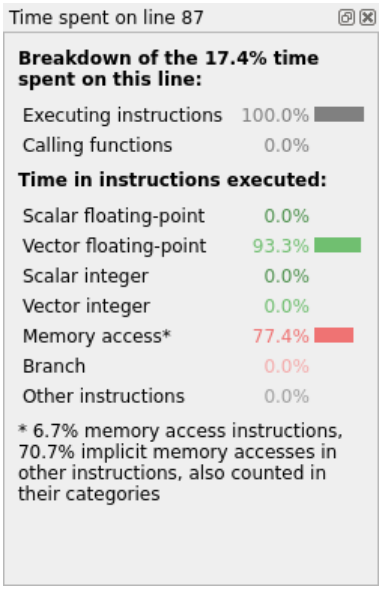
3.7 Selected lines view

The **Selected lines** view enables you to get detailed information about how one or more lines of code are spending their time.

Note: The **Selected lines** view is only available for profiles generated on x86_64 systems.

3.7.1 Use Selected lines view

To access the **Selected lines** view, open one of your program's source files in the **Source code viewer** and highlight a line.



The **Selected lines** view is by default shown on the right side of the **Source code viewer**. It updates automatically to show a detailed breakdown of how the selected lines are spending their time.

You can select multiple lines, and Linaro 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 *View source code (MAP)* 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 *Metrics view* for more information.

The **Selected lines** view 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 *Stacks view*. For more information on the **Functions** view see *Functions view*.

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 *CPU instructions*, 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, Linaro 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 optimizing for your processor's SIMD instructions.
- If you see a large amount of time in memory operations, look for ways to more efficiently access memory to improve cache performance.
- If you see a large amount of time in branch operations, look for ways to avoid using conditional logic in your inner loops.

CPU instructions offers detailed advice on what to look for when optimizing the types of instruction your program is executing.

3.7.2 NVIDIA GPU CUDA profiles

When NVIDIA CUDA kernel analysis is enabled and the selected line is executed on the GPU, a breakdown of warp stall reasons on this line displays in this view.

For details about how to enable NVIDIA GPU profiling see *NVIDIA GPU profiling*.

For a description of each of these warp stall reasons, refer to the tooltip for each of the entries or see *CUDA Kernel analysis*.

3.7.3 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:

- Hot lines in an 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.
- 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).
- Loops with stalls and mispredicted branches still show up highlighting the line containing the problem in all but the most extreme cases.

Key points:

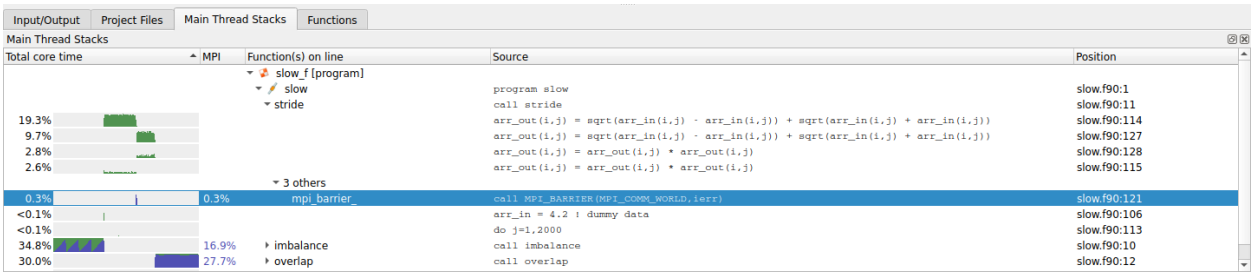
- Expert users: those wanting to use MAP's per-line instruction metrics to investigate detailed CPU performance of a loop or kernel (even down to the assembly level) should be aware that instructions executed in parallel by the CPU will show up with time only assigned to the last one in the batch executed.
- Other users: MAP's statistical instruction-based metrics correlate well with where time is spent in the program 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.

3.8 Stacks view

The **Stacks** view(s) offer a good top-down view of your program. Depending on your program and *thread view mode* you will see some combination of *Main Thread Stacks*, *Thread Stacks*, and *OpenMP Stacks* tabs.

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 *Source code (MAP)*.



The **Stacks** view shows:

- The first line, `slow`, represents the entire program run.

- Nested below `slow` is a call to the `stride` function. Almost all of this call was in single-threaded compute (dark green).
- The `stride` function spent most of that time on the line `arr_out(i,j)=sqrt(...)` at `slow.f90`, line 114. 19.3% of the entire run was spent executing this line of code.
- The 0.3% MPI time inside `stride` comes from an `MPI_Barrier` on line 121.
- The next major function called from program `slow` is the `overlap` function, shown at the bottom of this figure. A more detailed breakdown is described in *Metrics view*. This function ran for 30.0% of the total time, and almost all of this was in MPI calls.

When you click on any line of the **Stacks** view, the **Source Code viewer** jumps to 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 discussed in *Source code (MAP)*, a sawtooth pattern in the view represents imbalance between processes or cores.

In the figure above, the `MPI_Send` call inside the `overlap` function has a sawtooth pattern. 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.

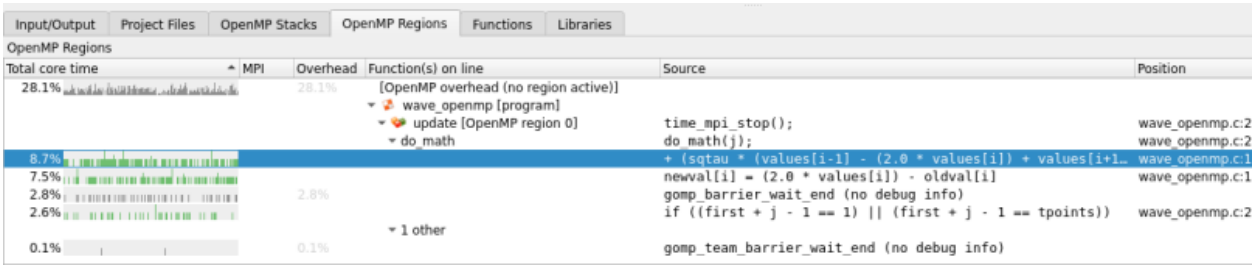
The **Stacks** view shows which lines of code spend the most time running, computing, or waiting. As with most places in the user interface, you can hover over a line or chart for a more detailed breakdown.

3.9 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 like 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 *Source code (MAP)*.



Note: If you are using MPI and OpenMP, this view summarizes all cores across all nodes, not just one node.

The **OpenMP Regions** view shows:

- The most time-consuming parallel region is in the `update` function at line 207. Clicking on this shows the region in the **Source Code viewer**.
- 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.

- 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.
- Calculating `sqtau = tau * tau` is the next most expensive line, taking 10.5% of the total core hours.
- 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 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 *Metrics view* for more information on CPU instruction metrics.

Click on any line of the **OpenMP Regions** view to jump to the **Source Code viewer** to show that line of code.

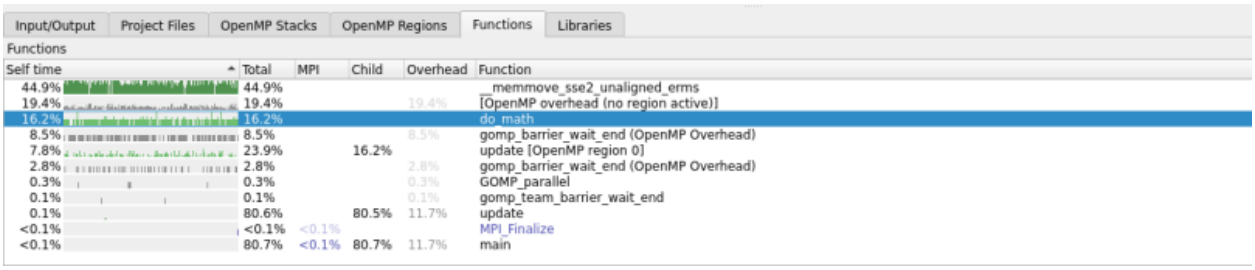
The percentage OpenMP synchronization time gives an idea as to how well your program is scaling to multiple cores and highlights the OpenMP regions that are causing the greatest overhead. Examples of things that cause OpenMP synchronization include:

- 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.

3.10 Functions view

The *Functions* view shows a flat profile of the functions in your program.



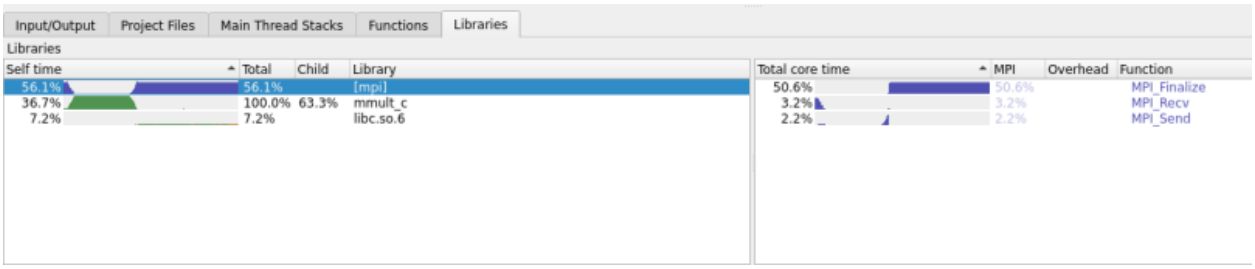
The first three columns show different measures of the time spent in a given function:

- *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.
- *Total* shows the time spent in code in the given function itself, and all its callees.
- *Child* shows the time spent in the given function's callees only.

You can use the *Functions* view to find costly functions that are called from many different places.

3.11 Library view

The *Library* view shows a flat profile of the shared libraries in your program.



The display on the left shows the shared libraries used by your program, plus the program binary. Similar to the *Functions view* the first three columns show different measures of the core time spent in a given library:

- *Self* shows the time spent in code in the given library itself, but not its callees, that is, not in the other libraries called by that library.
- *Total* shows the time spent in code in the given library itself, and all its callees.
- *Child* shows the time spent in the given library's callees only.

The display on the right lists the sampled entry point functions to the library currently selected in the display on the left.

You can use the *Library* view to identify the most significant libraries being used by your program, and evaluate the impact of upgrading, replacing or improving such libraries (i.e. swapping one math library with a functionally equivalent alternative with different performance characteristics).

3.12 Project Files view

The *Project Files* view offers an effective way to browse and navigate through a large, unfamiliar code base.

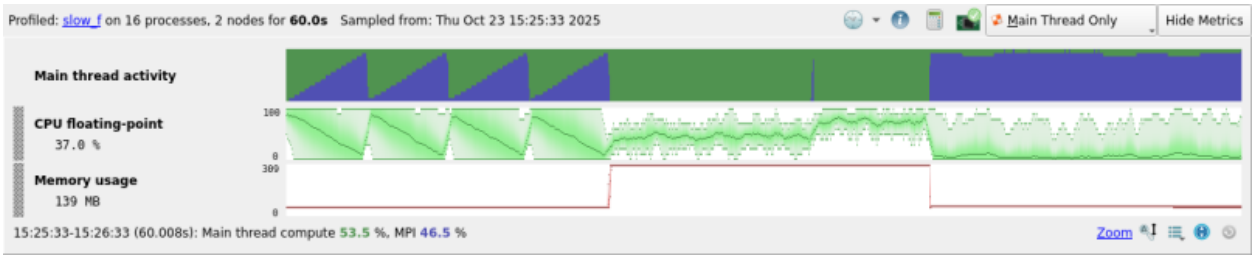
The *Project Files* view distinguishes between *Application Code* and *External Code*. You can choose which folders count as application code by right-clicking. *External Code* is typically system libraries that are hidden at startup.

3.13 Metrics view

This section describes how the **Metrics** view works with the **Source code viewer**, **Stacks** view, and **Project Files** view to help you identify and understand performance problems.

3.13.1 User interface

In the **Metrics** view, the horizontal axis is wall clock time. By default three metric graphs are shown.



The top graph is the **Main thread activity** chart, which uses the same colors and scales as the per-line sparkline graphs described in *Source code (MAP)*. Linaro recommends that you read that section to help understand the **Main thread activity** chart.

For CUDA programs profiled with CUDA kernel analysis mode enabled, a **Warp stall reasons** graph also displays. 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 *CUDA Kernel analysis*. Linaro recommends that you read that section to help understand the **Warp stall reasons** graph.

All of the other metric graphs show how single numerical measurements vary across processes and time. Two frequently used graphs are **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.

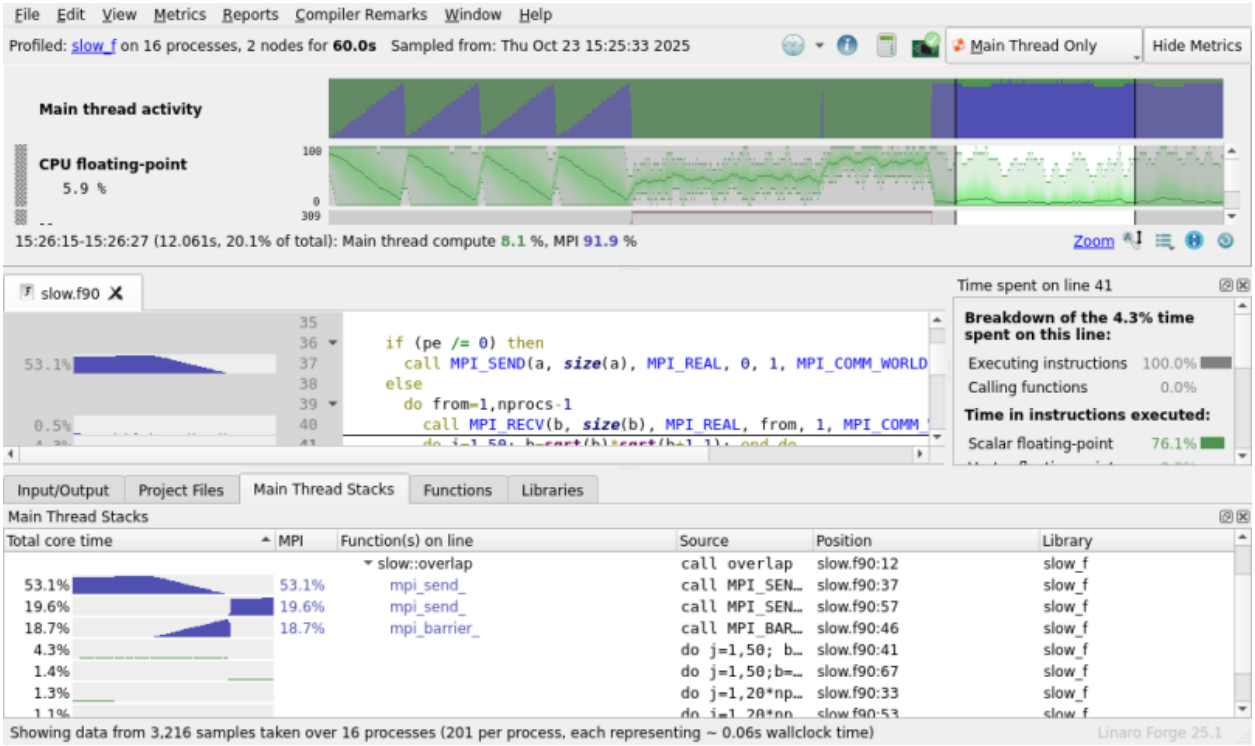
A thin line means all processes had very similar values. A large shaded region means there is significant imbalance between the processes. Extra details about each moment in time appear below the metric graphs as you move the mouse over them.

The **Metrics** view is at the top of the user interface because 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 Linaro 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 the execution of your program. To reselect the entire time range double-click or click the Select All Button  .

The value displayed under the metric graph label is the **Selection Mean**. This value represents the mean value of the metric across the selected time range in the **Metrics** view. As the region of time selected changes, this value should represent the mean for selection, not the entire graph. Observe the different **Selection Mean** values in the first image and when a region of time is selected in the second image.

In this figure, a short region of time has been selected around an interesting sawtooth pattern in time in `MPI_Barrier` because PE 1 is causing delays.



The first block accepts data in PE order, and is severely delayed. The second block is more flexible, accepting data from any PE, so that PE 1 can compute in parallel. The **Source code viewer** shows how compute and communication are serialized in the first block, but overlap in the second.

3.13.1.1 Changing Metrics

There are many more metrics other than those displayed by default. To change the visible metrics:

- Click *Metrics* on the main menu
- Or, right-click on the metric graphs
- Or, click the **Change Metrics** button  in the **Metrics** view

to choose one of the available presets or any combination of the metrics beneath them. For more information on available metrics, see *Metrics*.

The display order of metrics can be changed by left clicking the *handle* at the left-most point on each metric graph (to the left of the metrics label). This can be dragged to a new position in the metrics graph.

Adding more metrics to the **Metrics View** may result in a scroll bar appearing for navigating to these additional metrics.

You can return to the default set of metrics at any time by choosing the *Preset: Default* option.

3.13.1.2 Metrics Height

The height of each metric graph can be modified by selecting the **Change Metrics Height** icon  in the **Metrics** view. When this height is altered, it will persist in the user's local settings between sessions. See *Configuration files* for more information about changing these settings.

```
[map metric graphs]
graph height = 50
metrics table autoresize cap = 425
```

These settings can also be reset by selecting *Window* and *Default Layout* in the menu options.

3.13.2 Zooming

The following is a list of actions applicable to the **Metrics** view graphs, including their corresponding button in the view, if the button exists.

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 	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.
Reset	Double-click outside selected range of samples.	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 **Auto Vertical Zoom** icon 	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.

3.13.2.1 Horizontal Zoom

To examine a small time range in more detail you can horizontally zoom in the metric graphs by selecting the time-range you want 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 **Source code viewer**, will already have redrawn to display only the selected region when that selection was made.

A right-click on the metric graph zooms out on the metric graph.

This horizontal zoom is limited by the number of samples that were taken and stored in the Linaro MAP file. The more you zoom in the more 'blocky' the graph becomes.

Although you can increase the resolution by instructing Linaro MAP to store more samples (see `FORGE_SAMPLER_NUM_SAMPLES` and `FORGE_SAMPLER_INTERVAL` in *MAP and Performance Reports variables*), Linaro recommends avoiding this because it might significantly impact performance of both the program being profiled, and of Linaro MAP when displaying the `.map` output file.

3.13.2.2 Vertical Zoom

You can also zoom in vertically to better see fine-grained variations in the values of a specific metric. The **Auto Vertical Zoom** button  beneath the metric graphs causes 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. Hold down `Ctrl` (or `CMD` on Mac OS X), while either dragging on a metric graph, or using the mouse-wheel while hovering over one, to zoom that graph vertically in or out, and centered on the current position of the mouse.

A vertically-zoomed metric graph can be panned up or down by either holding down `Shift` while dragging on a metric graph or just using the mouse-wheel while hovering over it. Manually adjusting either the pan or zoom disables auto-zoom mode for that graph. Click the **Auto Vertical Zoom** button again to re-apply it.

3.13.3 View 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 side of the graph, a tooltip will display additional summary information. The tooltip will show you the *Minimum*, *Maximum*, and *Mean* of the metric across time and ranks or nodes.

For metrics which are not percentages, the tooltip will also show the peak sum across ranks / nodes. For example, the *Maximum (Σ all nodes)* line in the tooltip 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*) the tooltip will also show the cumulative total across all ranks / nodes, for example, *Lustre bytes read (Σ all nodes)*.

3.13.4 Metrics

This section describes various metrics visible in the **Metrics** view.

3.13.4.1 CPU instructions

The following sections describe the CPU instruction metrics available on each platform, x86_64 and Arm®v8-A.

Note: Due to differences in processor models, not all metrics are available on all systems.

Tip: When you select one or more lines of code in the **Source code viewer**, Linaro MAP shows a breakdown of the CPU instructions used on those lines. *Selected lines view* describes this view in more detail.

3.13.4.1.1 CPU instruction metrics available on x86_64 systems

These metrics show the percentage of time that the 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 x86 floating-point. All *CPU floating-point vector* instructions are included. 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. All *CPU integer vector* instructions are included. 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, and may overlap with instructions classified elsewhere. 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 *Linaro MAP does not correctly identify vectorized instructions* for a list of the instructions considered vectorized.

CPU integer vector
The percentage of time each rank spends in vectorized 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 *Linaro MAP does not correctly identify vectorized instructions* 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.

3.13.4.1.2 CPU instruction metrics available on Arm®v8-A systems

Note: 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.

The CPU instruction metrics available on Arm®v8-A systems are:

Cycles per instruction
The number of CPU cycles to execute an instruction. Lower is better. This metric can be less than one when the program takes advantage of instruction-level parallelism. Please consult L2 Data cache misses,

Stalled backend cycles and Stalled frontend cycles for further insight on how the cycles per instruction might be reduced.

L2 Data cache misses
The ratio of L2 data caches misses to kilo instructions retired. Lower is better. Often it is possible to reduce this figure by analyzing loops for poor cache performance and problematic memory access patterns, improving performance significantly.

Branch mispredicts
The rate of speculatively-executed instructions that do not retire due to incorrect prediction.

Stalled backend cycles
The percentage of cycles where no operation was issued because of the backend, due to a lack of required resources. Lower is better. This metric captures inefficiencies in backend units like execution units, data cache misses and translation delays caused by data TLB walks [1]. See the L2D cache miss ratio for further insight. Additionally, pressure on the arithmetic units will appear as stalled backend cycles [1].

Stalled frontend cycles
The percentage of cycles where no operation was issued because of the frontend, due to fetch starvation. Lower is better. This metric captures inefficiencies in the branch prediction unit, fetch latency due to instruction cache misses and translation delays caused by Instruction TLB walks [1].

References
[1] Mundichipparakkal,J., 2023, June. Arm Neoverse V1 Core: Performance Analysis Methodology

3.13.4.2 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 **User-mode CPU time** 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 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 happens 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.

3.13.4.3 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 might 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 might 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.
- I/O time in the *Application Activity* chart done using direct kernel calls is not counted.
- Even if your application does not perform I/O, a non-zero amount of I/O is recorded at the start of profile because of internal I/O performed by Linaro MAP.

3.13.4.4 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 that if you malloc or ALLOCATE a large amount of memory but do not actually use it, the **Memory usage** metric does 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.

3.13.4.5 MPI calls

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 has 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, Linaro 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 shows MPI activity, but some regions of the MPI metrics might 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, and therefore 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.

Note: For more information about the MPI standard definitions for these types of operations, see chapters 3 and 5 in the MPI Standard (version 2.1).

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) Linaro MAP shows calls to shmem_barrier_all in **MPI collectives**, **MPI calls**, and **MPI call duration**. Metrics for other SHMEM functions are not collected.

3.13.4.5.1 Detecting MPI imbalance

The **Metrics** view shows the distribution of their value across all processes against time, so any large regions are showing an area of imbalance in this metric. Analyzing imbalance in Linaro MAP works like this:

1. Look at the **Metrics** view for any large 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 large region, zooming the rest of the controls in to just this period of imbalance.
3. Now the **Stacks** view and the **Source code viewer** 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 largest 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 Linaro 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 Linaro MAP with the full power of an interactive debugger helps get to the root cause of the imbalance behavior.

3.13.4.6 Accelerator

3.13.4.6.1 NVIDIA

The NVIDIA CUDA metrics are enabled if you have Linaro Forge Ultimate or the Accelerator add-on. Contact Forge Support for upgrade information.

Note: Accelerator metrics are not available when linking to the static Linaro Forge sampler library.

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 *CUDA Kernel analysis*.

GPU memory usage
The memory allocated from the GPU frame buffer memory as a percentage of the total available GPU frame buffer memory.

GPU memory transfers
Metrics summarizing CUDA memory transfers are available for CUDA 11+ programs, including heterogeneous workloads where some processes use GPUs and others do not.

Three categories of metric are available:

- **Byte Transfer Rate:** Bytes transferred per second per process.
- **Memory Transfer Rate:** Transfers per second per process.
- **Time Spent in Memory Transfers:** Proportion of time in transfers per process.

Note: If a very large number of memory transfer events occur in the program, the **time spent in memory transfers** metric might only provide an approximation.

Different types of memory transfer can occur in the program you are profiling. For example, the program can transfer data between host memory and GPU device, or between different GPU devices on the host. Six memory transfer types are available within each category:

Host to Device
A host to device memory copy.

Device to Host
A device to host memory copy.

Device to Device
A device to device memory copy on the same device.

Host to Host
A host to host memory copy.

Peer to Peer
A peer to peer memory copy across different devices.

Off-device
Sum of host-to-device, device-to-host, and peer-to-peer types (everything using PCIe or NVLink).

Selecting the category using the *preset* mechanism displays the relevant metrics for all memory transfer types occurring within the program.

3.13.4.6.2 AMD

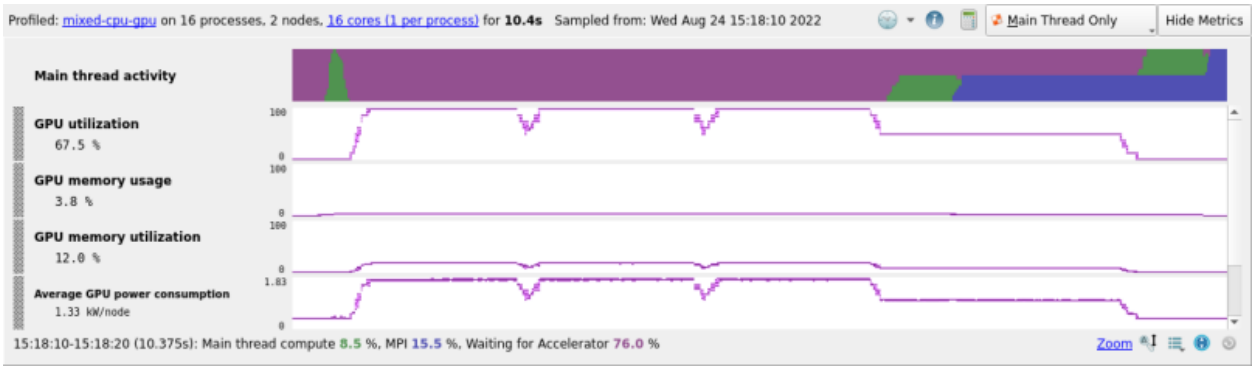
The AMD ROCm metrics are enabled if you have a Linaro Forge license with ROCm support. Contact Forge Support for upgrade information.

Note: Accelerator metrics are not available when linking to the static Linaro Forge sampler library.

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.

GPU memory usage
The memory allocated from the GPU Video RAM (VRAM) as a percentage of the total available GPU memory VRAM.

GPU memory utilization
Percentage of time that the GPU memory was in use. If multiple cards are present in a compute node, this value is the mean across all the cards in a compute node.



3.13.4.7 Energy

The energy metrics are only available with Linaro Forge Ultimate. All metrics are measured per node. If you are running your job on more than one node, Linaro MAP shows the minimum, mean, and maximum power consumption of the nodes.

Note: Energy metrics are not available when linking to the static Linaro Forge sampler library.

3.13.4.7.1 NVIDIA

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.

Note: NVIDIA GPU power measurement requires an 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.

3.13.4.7.2 AMD

GPU power usage
The average GPU power consumption on each node, as measured by the `rocm_smi` utility. This metric is available if the Accelerator metrics are present.

3.13.4.7.3 CPU

CPU power usage
The cumulative power consumption of all CPUs on the node, as measured by the Intel on-board sensor (Intel RAPL).

Note: 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.

3.13.4.7.4 Cray

Cray PM CPU power usage
The cumulative power consumption of all CPUs on the node, as measured by the Cray Power Monitoring metrics (Cray PM).

System power usage
The power consumption of the node as measured by the Cray metrics.

Note: Node power monitoring is implemented through the Cray HSS energy counters. The Cray HSS energy counters are known to be available on Cray XK6 and XC30 machines.

3.13.4.8 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 program being profiled. Therefore, there might be some data reported to be read and written even if the program 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 is from an I/O intensive program, not from background processes. This assumption has been observed to be reasonable. For generated program profiles with more than a few megabytes of data read or written, almost all of the data reported in Linaro MAP is attributed to the program that is 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. Linaro 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 Linaro MAP, you can observe the rate at which data is read and written to Lustre through the Lustre client. You can also identify whether a slow read or write rate can be correlated to a high rate of expensive metadata operations.

Note:

- 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 Linaro Forge Ultimate.

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` might count as multiple operations. For example, this might happen when a `file open` creates a new file or truncates an existing one.

3.13.5 Custom metrics

Custom metrics can be written to collect and expose additional data 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 Linaro MAP installation directory, for example, `/path/to/forge/map/metrics`. If a metric is installed in both locations, the user installation takes priority.

Detailed information about how to write custom metrics can be found in this supplementary Custom Metric Plugin Interface documentation web page.

Warning: Only install custom metrics from trusted third-party sources.

3.14 Thread affinity advisor

Linaro MAP collects CPU affinity binding information of threads that are sampled and presents it in the *Thread Affinity Advisor* dialog. To open this dialog, select the *View* ▸ *Thread Affinity Advisor* menu option or click the *Thread Affinity Advisor* button in the toolbar.

Note: The *Thread Affinity Advisor* dialog is not available for `.map` files that were generated without the Thread Affinity Advisor feature included in the license.

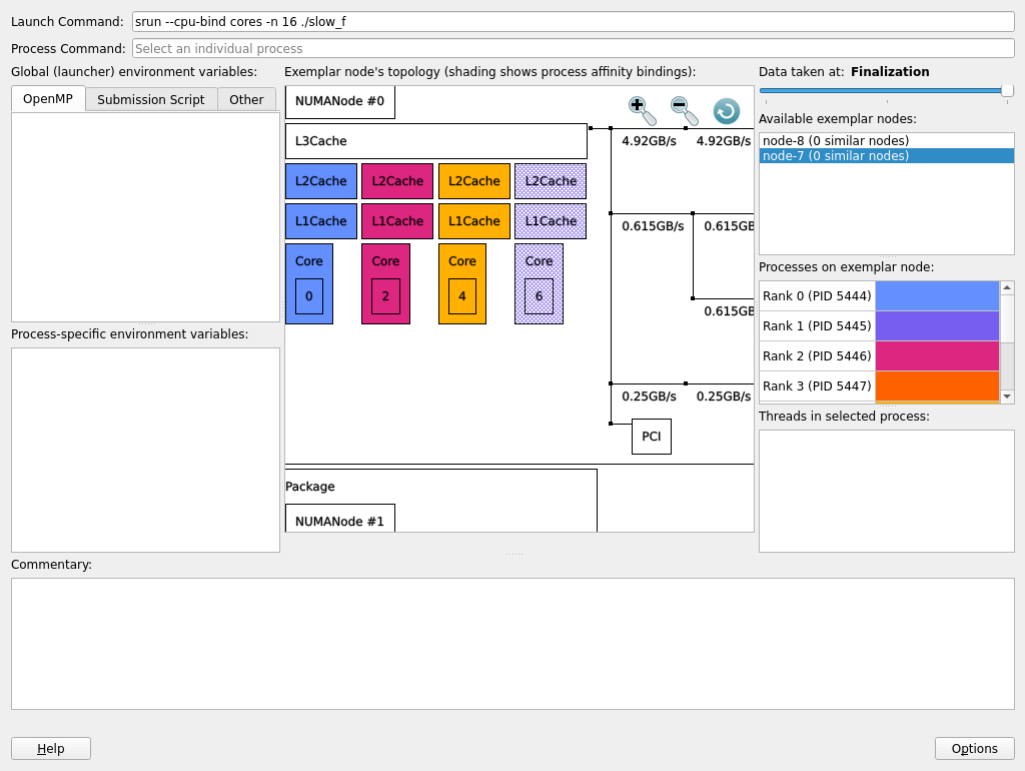


Fig. 3.1: Thread affinity advisor dialog

The *Thread Affinity Advisor* dialog provides affinity binding information in two main parts: settings that affect affinity bindings, and snapshots of the resultant affinity bindings recorded at different points in the application's runtime. Sections can be resized as needed.

3.14.1 Affinity binding snapshots

The affinity bindings snapshots are taken at the following points:

Library load
: Before execution enters the main method, typically before OpenMP thread pools are initialized.

`MPI_Init` or `MPI_Init_thread`
: In the `MPI_Init` or `MPI_Init_thread` call initializing MPI communication. For non-MPI applications this will take place at the start of the application's main method.

MPI_Finalize
: In the `MPI_Finalize` call. For non-MPI programs this will take place when the application is about to shut down. OpenMP threads typically still exist at this point.

Toggle between these using the selection slider.

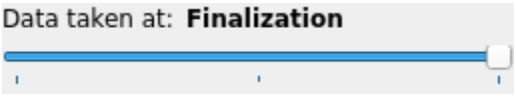


Fig. 3.2: Thread affinity snapshot slider

3.14.2 Affinity binding variables

The MPI launch and process command lines are presented at the top of the *Thread Affinity Advisor* dialog. Select an individual process to display its command line.

Global environment variables taken when the launch line was executed, and process-specific environment variables set by the job scheduler. Global environment variables are separated between OpenMP variables, submissions script variables (e.g. variables set as a consequence of `#SBATCH` lines in a job submission script), and all other launch-time variables of relevance.

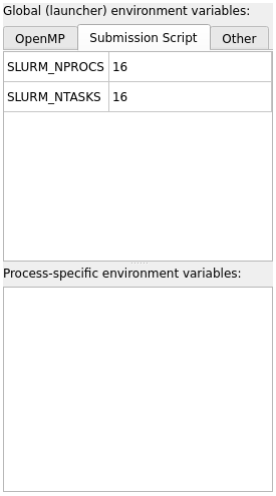


Fig. 3.3: Thread affinity relevant environment variables

3.14.3 Node topology viewer

The center of the *Thread Affinity Advisor* dialog contains the node topology viewer, which is an abstract depiction of the main hardware components of the selected exemplar node.

Note: CPUs reserved by the system or omitted via Linux `cgroups` cannot be displayed. This can be the case when a system is configured with "core-specialization" or a "low-noise" mode.

Note: The hardware displayed in the node topology viewer may not be accurate in a virtualized environment.

The node topology viewer can be panned by clicking and dragging with the mouse, or zoomed in/out with the mouse-wheel. Buttons to zoom in/out or reset the zoom level are available in the top right when the mouse cursor is in the viewer.

CPUs are displayed by their OS index in the view. An option may be set to view the CPUs by their logical index in the view. See *Configuration* for more information.

Match the CPU item's background color and pattern in the processes table to find the MPI process that was bound to that CPU. A solid red background indicates that threads from different processes are bound to that CPU (i.e. there are overlapping affinity masks), while red text is used to indicate that multiple threads from a single process are bound.

Select a CPU item to highlight the processes and threads that are bound to it in their respective tables.

Selecting any item displays additional details underneath the node topology viewer if available.

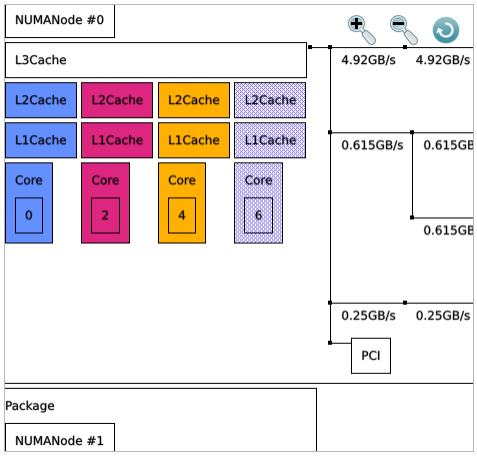


Fig. 3.4: Thread affinity node topology viewer

3.14.4 Exemplar nodes

The *Thread Affinity Advisor* dialog condenses the amount of information displayed when conveying the affinity binding information of the sampled program. Therefore, affinity bindings that are similar across different nodes are incorporated into a single exemplar node, where the lowest ranked MPI process was run. Nodes that are similar to the exemplar have:

- similar hardware
- same number of MPI processes
- same number and type of threads in those processes
- same affinity bindings of those threads

These nodes are listed under *Exemplar nodes*. Hover over exemplar nodes to show tooltips which highlight the hardware differences between them.

3.14.5 Processes and threads

Processes that were run on the selected exemplar node are displayed on the right, along with their corresponding MPI rank. Select a process to display its threads.

Threads are categorized by their type and activity. Low-activity "monitor" or "helper" threads from known MPIs, OpenMP or accelerator runtimes are listed with gray text and unchecked by default. These threads do not contribute to the background colors and patterns of CPU items in the node topology viewer. This behavior can be overridden by checking the box next to the listed thread.

Select a thread to highlight the CPUs in the node topology viewer that it was bound to.

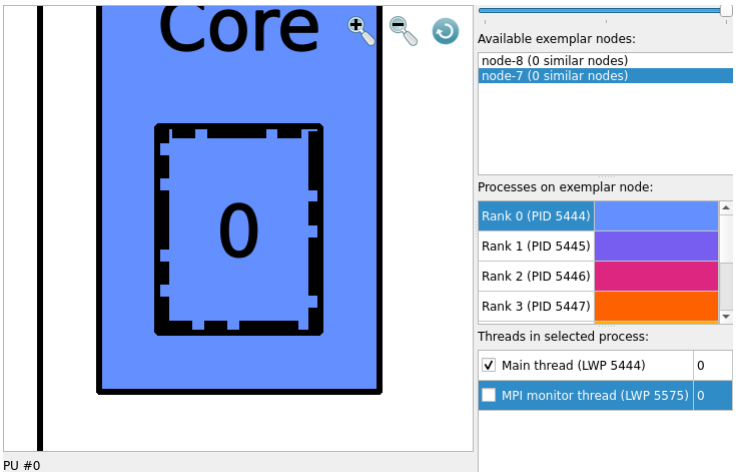


Fig. 3.5: Highlighted CPU in tread affinity node topology viewer

3.14.6 Commentary

Commentary is presented at the bottom of the *Thread Affinity Advisor* dialog, which provides brief summaries of the main affinity binding issues detected in the sampled application. These are prefixed by their severity level.

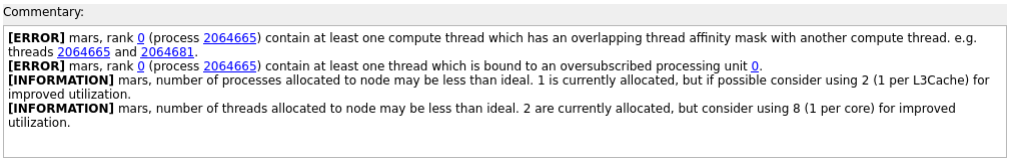


Fig. 3.6: Thread affinity commentary

3.14.7 Configuration

There are display configuration options available for the *Thread Affinity Advisor* dialog, which may persist between sessions.

To examine or change the settings, click *Options* in the *Thread Affinity Advisor* dialog.

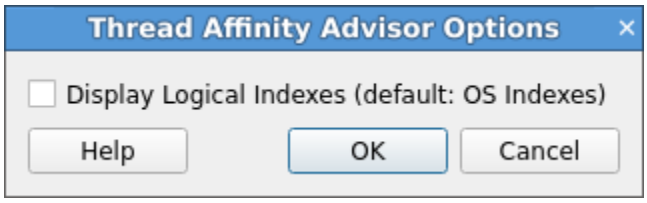


Fig. 3.7: Thread affinity advisor options

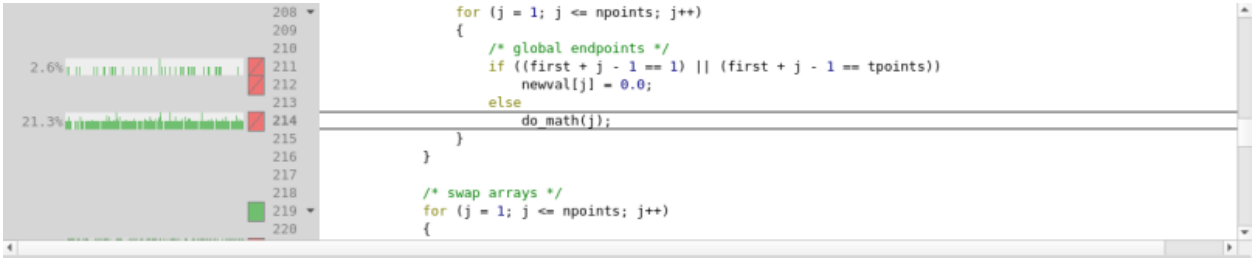
The available settings are:

Display Logical Indexes:
Use logical indexes (rather than OS indexes) when displaying CPUs in the *Thread Affinity Advisor*. The default value of this setting is false.

Note: These indexes are an abstraction, and may not match the physical or OS indexes as used by tools such as `numactl` or `taskset`.

3.15 Compiler remarks

Linaro MAP can annotate your source code with compiler remarks extracted from compiler optimization reports.



A compiler optimization report is created for each compilation unit. Linaro MAP currently supports the following compilers with this feature:

- Arm Compiler for Linux
- HPE Cray Clang C and C++ (Cray Compiling Environment 12 and later)
- GNU C/C++/Fortran Compiler 13 and later
- Intel oneAPI HPC Toolkit

To generate the compiler report, introduce the following compiler-dependent compilation flag to your build scripts:

- `-fsave-optimization-record=yaml` (HPE Cray Clang C and C++, and Arm Compiler for Linux)
- `-fsave-optimization-record` (GNU C/C++/Fortran Compiler 13 and later, and Intel oneAPI HPC Toolkit 2025 and later)
- `-qopt-report[=min|med|max]` (Intel oneAPI HPC Toolkit less than 2025)

The compiler generates an optimization report for every compilation unit in your build system. The report filename extensions are compiler-dependent, for example:

- `example.opt.yaml` (HPE Cray Clang C and C++, Arm Compiler for Linux, and Intel oneAPI HPC Toolkit)
- `program-example.c.opt-record.json.gz` (GNU C/C++/Fortran Compiler 13 and later)

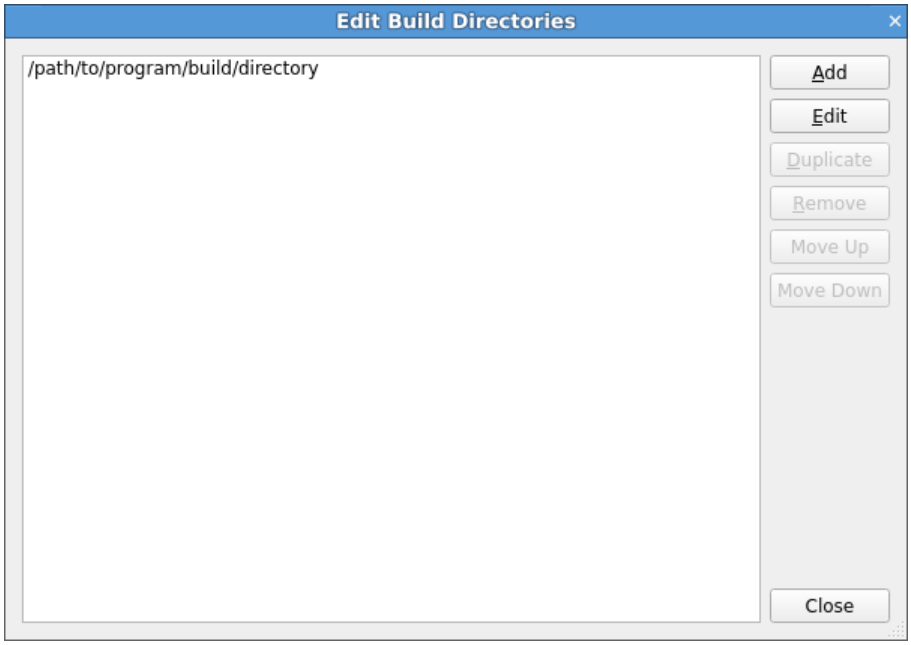
After profiling your program with Linaro MAP, ensure that compiler remarks are enabled by checking the *Compiler remarks* ▸ *Enabled* menu option. Linaro MAP automatically locates the compiler report associated with your source code and displays them as annotations.

Note: As an advanced feature, compiler remarks are disabled by default.

3.15.1 Setting build directories

Linaro MAP does not locate compiler remarks for non-trivial build systems. In this case, specify the build directory of your program using the *Compiler remarks* ▸ *Build directories…* menu option.

Note: Build directories are searched in the order they are specified.



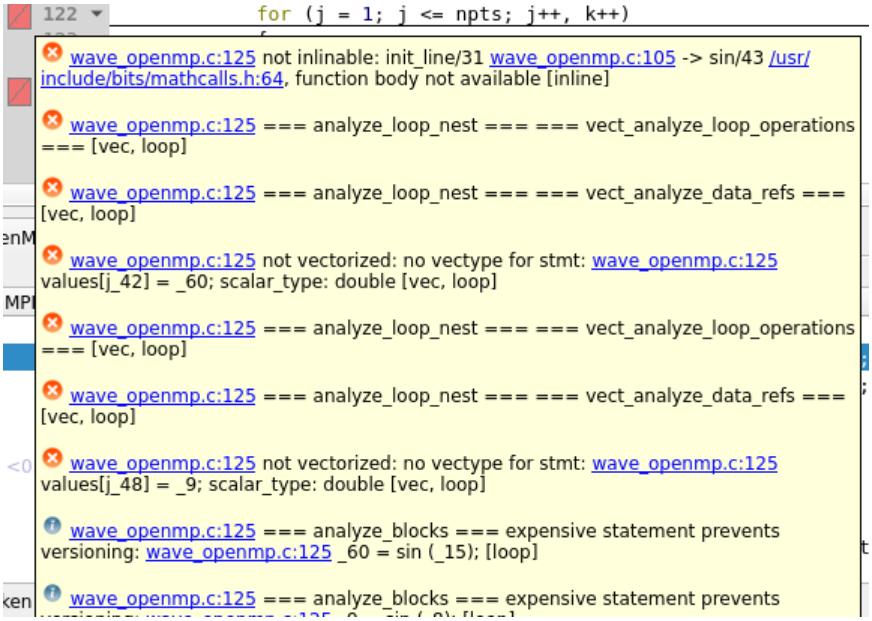
Alternatively, set your program's build directories before profiling by using the `--build-directories` command line argument (see *Command line arguments*) or the *Run* dialog.

3.15.2 Source code annotations

Compiler remarks are displayed as annotations next to your source code. Their color indicates the type of remark present in the following priority order:

- Red: failed or missed optimizations
- Green: successful or passed optimizations
- White: information or analysis notes

If you hover the mouse cursor over an annotation, a remark tooltip will be displayed listing the compiler remarks associated with the source code line. If you click on an annotation, the tooltip will become movable and hovering your mouse cursor over the tooltip will highlight the code segments to which the remarks refer. Click on the hyperlinks to jump to the specified source code locations.



3.15.3 Remarks filtering

Several compiler remarks can be associated with a single source code line. If a remark tooltip becomes too long, it will be truncated with a **...** to indicate there is additional content. To view the hidden content, try filtering out compiler remarks that are not of concern by using the *Compiler remarks* menu options. Compiler remarks can be filtered by type, as well as the compiler optimization pass that generated them.

Note: Enable the compiler remarks feature to view the remark types and optimization passes associated with your source file.

3.15.4 Embedding remarks

Linaro MAP requires access to the compiler report in order to display compiler remarks. This may not be possible if, for example, the `.map` file has been moved to another system, or the build directory has been removed.

Compiler remarks can be embedded into the `.map` file so that they are present wherever the profile is viewed. Use the *File* ▸ *Save Profile Data As* menu option to create a `.map` file with embedded remarks.

Note: Embedding remarks will generate a `.map` file of a larger size.

Only compiler remarks for source code files that have been opened while viewing the `.map` file will be embedded. The current filtering options will be applied (see the *Remarks filtering* section) to which remarks are embedded. Consider adjusting the filters to generate a `.map` file of a manageable size.

Another option is for Linaro MAP to generate a `.map` file with embedded remarks after profiling your program. Use the `--embed-compiler-remarks` command line option, or check the *Run* dialog option to accomplish this. Ensure that build directories have been set (see the *Setting build directories* section) so that Linaro MAP finds the necessary compiler reports.

Note: Linaro MAP will require additional time during the analysis phase when embedding remarks with this method.

Additionally, the types of remarks that are embedded can be fine-tuned with the `--exclude-compiler-remarks` option (see *Command line arguments*), or in the *Run* dialog.

Note: A `.map` file containing embedded remarks will not load additional compiler optimization reports.

3.16 Compare MAP Profiles

Comparing MAP profiles can be a useful approach when optimizing an application. For example, when measuring the impact of different compilation flags or performance variances between compilers.

Comparing MAP profiles can also be useful when carrying out regression analysis. For example, comparing the impact of porting an application from one architecture to another, introducing accelerators to areas of the application, or identifying slow-downs in an application run-time from a previous build.

Linaro MAP can compare two profiles in the user interface by aligning the activity of two profiles at key **Alignment Points** (see *Alignment Points*) in the profile timeline. This can be achieved by:

- Comparing two pre-existing Linaro MAP profiles (`.map` files).
- Comparing a pre-existing Linaro MAP profile (`.map` file) with an application being profiled.

When comparing MAP profiles, one is defined as **current** and the other as **baseline**. The **current** profile will refer to:

- the application being profiled
- or, the `.map` file being loaded.

The **baseline** will refer to the profile the **current** is being compared against.

For example, the **current** profile could refer to a profile generated on an aarch64 architecture. This profile could be compared against a **baseline** profile generated on an x86_64 architecture, if porting an application from x86_64 to aarch64, or vice versa.

3.16.1 Loading two profiles

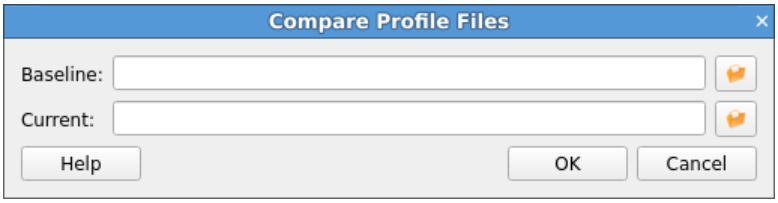
3.16.1.1 Compare pre-existing profiles

To compare two pre-existing Linaro MAP files, the paths to these `.map` files must be provided to Linaro MAP.

To load two profiles via the command-line use the `--baseline` option to provide the path to the **baseline** profile. This will result in both profiles being opened in the GUI.

```
map --baseline=<baseline-profile>  <current-profile>
```

To load two profiles via the Welcome Page (see *Welcome page*), select *Compare Profile Data Files* to open the *Compare Profile Files* Dialog.



The file icon can be used to navigate to the location of the **current** and **baseline** profiles. When paths are provided, selecting *OK* will result in both profiles being opened in the GUI.

Note: The **baseline** and **current** profiles must have distinct filenames.

3.16.1.2 Profiling an application

To compare a pre-existing **baseline** profile with an application being profiled, provide the path to the **baseline** profile via the command-line `--baseline` when launching Linaro MAP. Select *Profile* on the Welcome Page (see *Welcome page*) to launch the *Run* Dialog and provide the application arguments.

```
map --baseline=<baseline-profile>
```

Or, launch Linaro MAP with the **baseline** profile followed by your application launch arguments:

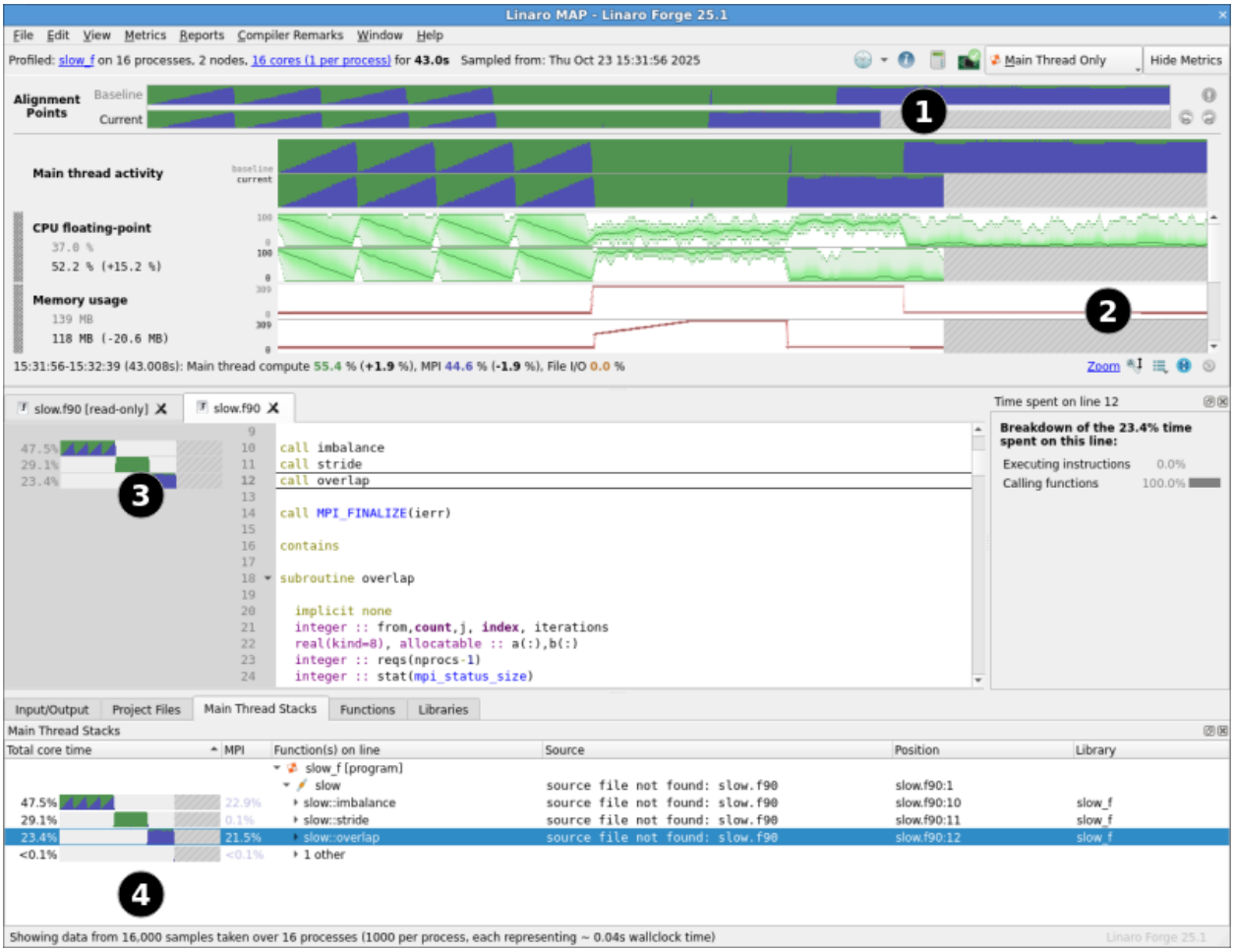
```
map --baseline=<baseline-profile> program-name [arguments]
```

In both instances, upon completion of the application being profiled, this **current** profile will be opened in the GUI with the **baseline** profile provided on the command-line.

Note: Comparing MAP files with `--baseline` cannot be used when generating an offline profile with `--profile`.

3.16.2 User-Interface

When comparing a **current** and **baseline** profile, one profile will often be faster than the other. Linaro MAP will represent this difference in runtimes with shaded gaps at the end of the shorter profile in the **Metrics** View (see *Metrics view*).



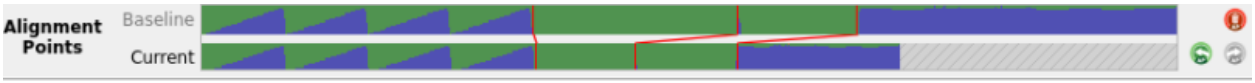
This table shows the Linaro MAP components currently displaying comparison information:

Key	Component
1	*Alignment Points*
2	*Metrics View*
3	*Source Code Viewer*
4	*Stacks View*

Note: Not all features in Linaro MAP support comparing profiles.

3.16.2.1 Alignment Points

The most notable feature to indicate that two `.map` files are being compared is the presence of the **Alignment Points** view. Alignment Points are used for inserting gaps in the Metrics' graphs such that common events, such as the start of various phases of program execution, can be lined up vertically, even when preceding events take longer in one of the profiles.



This view displays the **Main thread activity** graphs for the **current** and **baseline** profiles. The horizontal length of these graphs are scaled, indicated with a gap at the end of the shortest graph, to align the wall-clock times of these two profiles.

These graphs are available for connecting **Alignment Points** between the **current** and **baseline** profiles. These points are connected by a red line between two points on each graph. Alignment points may not overlap in the **Alignment Points** view.

The following is a list of actions applicable to the **Alignment Points** view, including their corresponding button in the view, if the button exists.

Left Click	Add an Alignment Point to both **current** and **baseline** profiles.
Drag	Reposition an Alignment Point on the selected graph, creating a gap on Metrics' graphs.
Left Click and Drag	Add an Alignment Point to both **current** and **baseline** graphs, then reposition.
Right Click	Delete Alignment Point under mouse cursor.
	Delete All Alignment Points.
	Undo Last Alignment Point Action (Add, Move, Delete, Delete All)
	Redo Last Alignment Point Action (Add, Move, Delete, Delete All)

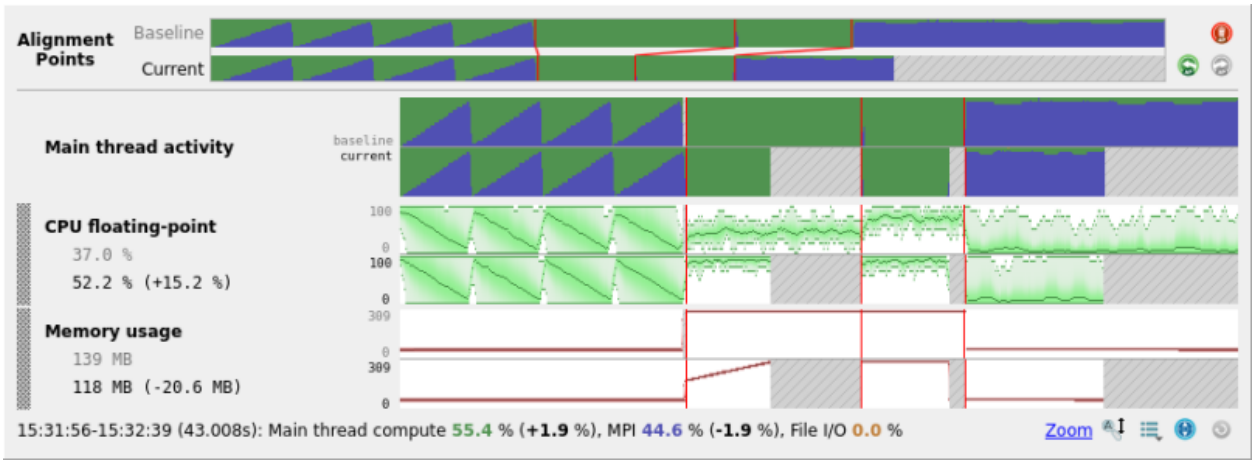
Note: It is not possible to select a time range or horizontal zoom using the **Alignment Points** view. To select a time range or horizontal zoom, use the metrics graphs beneath the **Alignment Points** view, see *Zooming*.

The tooltips for each graph in the **Alignments Points** view will provide the *Program Metadata* for each profile. This will include information about how each profile was generated. The corresponding `.map` file name for each profile can be found in this tooltip.

3.16.2.2 Metrics View

When comparing two profiles, each graph in the **Metrics** view (see *Metrics view*) will display the **baseline** graph above the **current** graph.

The **baseline** and **current** graphs are distinguished by their different font colors on the **Metrics** view legend.



On **Metrics** graphs, the Alignment Points will always align vertically. However, there will be gaps inserted in the **Metrics** graphs to reflect variances between runtimes between **Alignment Points**. These **Alignment Points** are indicated by a red line in each **current** and **baseline** graph, preceded by a shaded gap created by aligning these samples between the profiles.

For example, the Sawtooth pattern at the start of each profile in the image above can be aligned by creating a minimal size gap on the **baseline** metrics graphs. The runtime of these common events are similar.

However, the large area of compute (green) with minimal MPI communication (blue) requires a large gap be added to the **current** profile to align the MPI communication spike in the middle of each profile.

In the **Metrics** view legend, the **Selection Mean** (see *Selection Mean*) for the **baseline** graph is shown above the selection mean for the **current** graph. These selection means are distinguished by their corresponding font colors. The difference between these two values is displayed to the right of the **current** graph selection mean. Hovering over the tooltips for each of these entries in the **Metrics** view will provide more details about *notable* differences between the profiles.

Due to the increased area attributed to each metric in the **Metrics** view when displaying both the **baseline** and **current** graphs, a scroll bar may appear for navigating the metrics view.

The height of each metric graph pair (**current** and **baseline**) can be modified, see *Metrics Height*. When this height is altered, it will persist in the user's local settings between sessions. See *Configuration files* for more information about changing these settings.

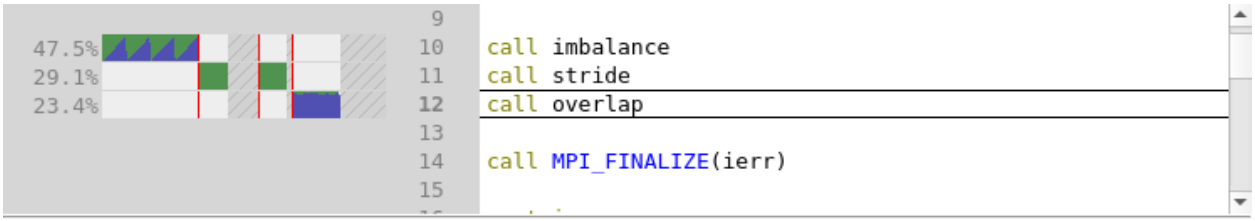
```
[map metric graphs]
mapdiff graph height = 35
metrics table autoresize cap = 425
```

These settings can also be reset by selecting *Window* and *Default Layout* in the menu options.

Note: If a metric is available in one profile but is missing in another (for example, in the case of architecture or accelerator specific metrics), then an error message will be displayed in the corresponding graph detailing why this metric was not captured.

3.16.2.3 Source Code Viewer

Each line in the **Source Code Viewer** (see *View source code (MAP)*) will continue to display the project files for the **current** profile. These source lines will be annotated with performance information for the **current** profile but will also contain representation of gaps created by **Alignment Points** and gaps representing if the **current** profile is slower than the **baseline** profile.

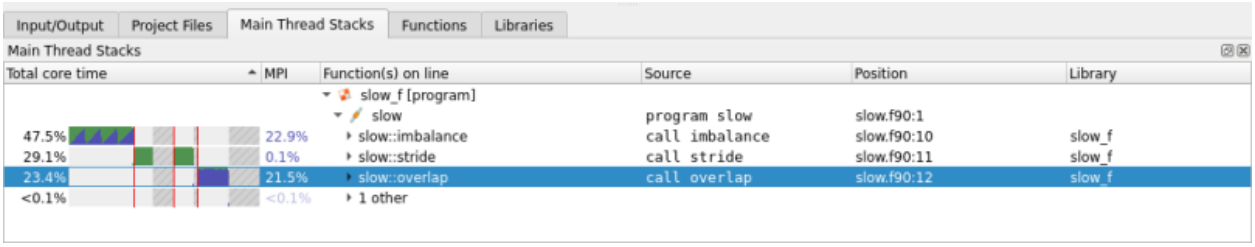


3.16.2.4 Various Stacks Views

The following views also contain gaps displayed in the performance annotation for each line:

- *Stacks View*
- *OpenMP Regions View*
- *Functions View*
- *Library View*
- GPU Kernels Tab, see *NVIDIA GPU Kernels* and *AMD GPU Kernels*.
- *NVIDIA GPU Memory Transfers View*

The line in the view will show performance annotation for the **current** profile, including gaps representing where the **current** profile is slower than the **baseline** profile and where the **Alignment Points** have introduced gaps between the profiles.



In the image above, we can see clearly where the **Alignment Points** have been placed to delineate the three major regions of the application.

3.17 MPI Call Statistics dialog

Linaro MAP collects data and statistics from all MPI calls made on the **main thread of each process**, regardless of whether it has been sampled. The *MPI Call Statistics* dialog provides a breakdown of these non-sampled statistics, breaking it down per MPI function.

To open this dialog use the *View* ▸ *MPI Call Statistics* menu option, or click the *Show the 'MPI Call Statistics' dialog* button in the toolbar.

Note: This dialog is only available if there were MPI calls detected during the run and is disabled when profiling non-MPI programs.

3.17.1 MPI call table and statistics

The table at the top of the dialog shows statistics for each MPI function called. Values in this table are the **mean** value across all processes. The table can be sorted by any one of these columns.

The following values are recorded:

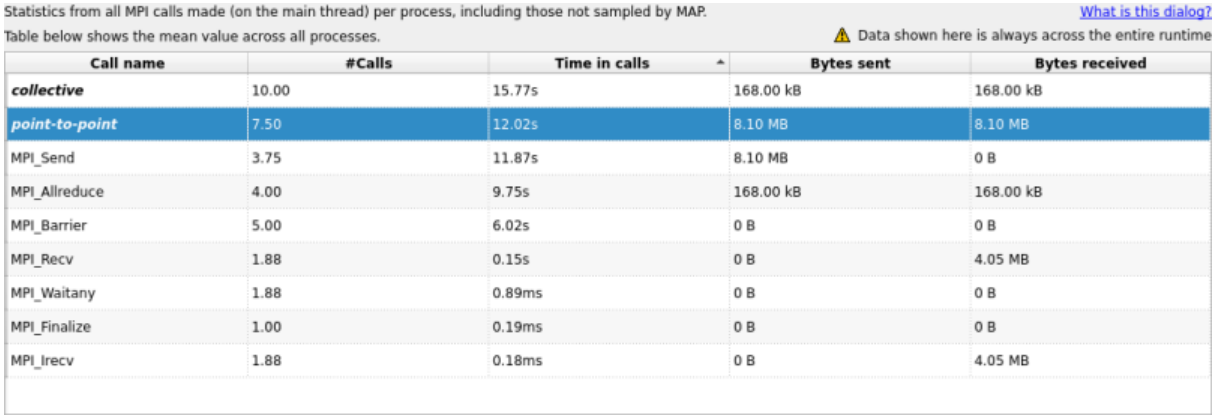
#Calls
The number of times that MPI function has been called.

Time in calls
The total time spent within that function throughout the whole runtime.

Bytes sent
The number of bytes that function has sent.

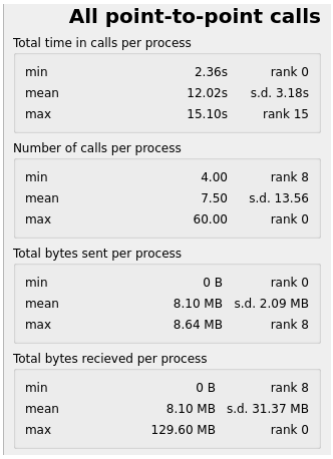
Bytes received
The number of bytes that the function has received.

In addition to the actual MPI functions, there are also entries which include the totals of all calls categorized as either **point-to-point** or **collective**.


Statistics from all MPI calls made (on the main thread) per process, including those not sampled by MAP. What is this dialog?
Table below shows the mean value across all processes. ⚠ Data shown here is always across the entire runtime

Call name	#Calls	Time in calls	Bytes sent	Bytes received
collective	10.00	15.77s	168.00 kB	168.00 kB
point-to-point	7.50	12.02s	8.10 MB	8.10 MB
MPI_Send	3.75	11.87s	8.10 MB	0 B
MPI_Allreduce	4.00	9.75s	168.00 kB	168.00 kB
MPI_Barrier	5.00	6.02s	0 B	0 B
MPI_Recv	1.88	0.15s	0 B	4.05 MB
MPI_Waitany	1.88	0.89ms	0 B	0 B
MPI_Finalize	1.00	0.19ms	0 B	0 B
MPI_Irecv	1.88	0.18ms	0 B	4.05 MB

By selecting a row in the table, a further breakdown of the MPI call will be displayed in the widget below. This breakdown further includes the *minimum*, *maximum* and *standard deviation* across all processes as well as a histogram (see *MPI call histogram* for more details).



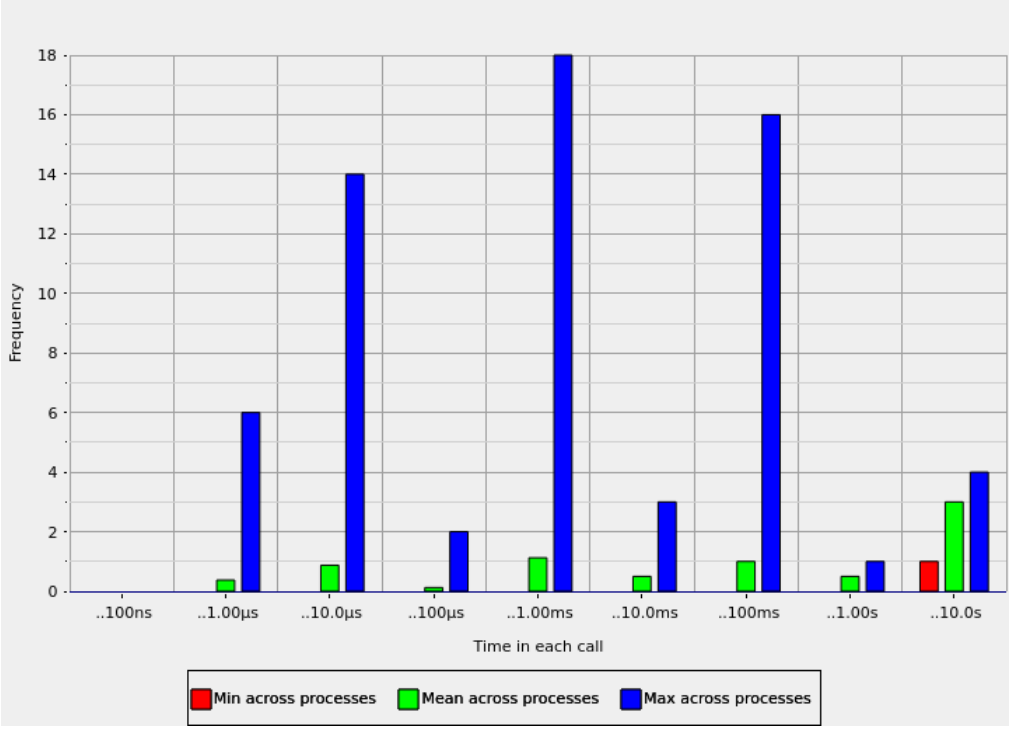
Note: All the data shown in this dialog is always across the entire sample runtime. As such, the statistics do not changed if you select a particular region within the main Linaro MAP window.

3.17.2 MPI call histogram

Alongside the MPI call statistics, selecting an MPI function from the table will also display a relevant histogram. The histogram has two modes which can be toggled between using the radio buttons below the MPI call statistics:

Message size in each call
: Show counts for the size of messages being transferred by the selected MPI function. In this case, each message is counted separately. For example, if a function both sends and receives a message, the sizes will be counted twice. Some functions such as *MPI_Barrier* do not send any data and so no histogram will be displayed.

Time in each call
: Shows counts for the time that each call of the selected MPI function took.



The histogram shows the *minimum*, *mean* and *maximum* across all sampled processes. The x-axis shows the range for each bin, including values more than the last bin range; up to and including the value specified.

Hovering over a bar will bring up a tooltip, showing a more detailed breakdown of the values and ranges of that bin.

3.18 Configurable Perf metrics

The Perf metrics use the Linux kernel `perf_event_open()` system call to provide additional CPU related metrics available for Linaro MAP.

They can be used on any system supported by the Linux perf command (also called `perf_event`). These cannot be tracked on typical virtual machines.

Note: This feature is available to use in Linaro Forge Ultimate edition. Contact Linaro Sales for details about how to upgrade.

Note: The following features are disabled when using configurable Perf metrics:

- CPU instruction metrics on Arm®v8-A (see *CPU instruction metrics available on Armv8-A systems* in *CPU instructions*).

3.18.1 Performance events

Perf metrics count the rate of one or more performance events that occur in a program. There are some software events that the Linux kernel provides, but most are hardware events tracked by the Performance Monitoring Unit (PMU) of the CPU. *Generalized* hardware events are event name aliases that the Linux kernel identifies.

The quantity and combinations (in some cases) of events that can be simultaneously tracked is limited by the hardware. This feature does not support multiplexing performance events.

If the set of events you requested cannot be tracked at the same time, Linaro MAP ends the profiling session immediately with an error message. Try requesting fewer events, or a different combination. See the PMU reference manual for your architecture for more information on incompatible events.

3.18.2 Permissions

On some systems, using Perf hardware counters can be restricted by the value of `/proc/sys/kernel/perf_event_paranoid`.

perf_event_paranoid	Description
3	Disable use of Perf events
2	Allow only user-space measurements
1	Allow kernel and user-space measurements
0	Allow access to CPU-specific data, but not raw trace-point samples.
-1	No restrictions

The value of `/proc/sys/kernel/perf_event_paranoid` must be 2 or lower to collect Perf metrics. To set this until the next reboot, run the following commands:

```
sudo sysctl -w kernel.perf_event_paranoid=2
```

To permanently set the paranoid level, add the following line to `/etc/sysctl.conf`:

```
kernel.perf_event_paranoid=2
```

3.18.3 Probe target hosts

You must probe an example of a typical host machine before using these metrics. As well as other properties, this also collects the CPU ID used to identify the set of potential hardware events for the host, and tests which generalized events are supported.

Ensure that `/proc/sys/kernel/perf_event_paranoid` is set to `2` or lower (*Permissions*) before performing the probe.

Note: It is not necessary to probe every potential host, a single compute node in a homogeneous cluster is sufficient.

If your home directory is writable, you can generate a probe file and install it in your config directory by running the following on the intended host:

```
/path/to/forge/bin/forge-probe --install=user
```

If the Forge installation directory is writable, you can generate and install the probe file for the current host with:

```
/path/to/forge/bin/forge-probe --install=global
```

To generate the probe file, but install it manually, execute:

```
/path/to/forge/bin/forge-probe
```

The probe is named `<hostname>_probe.json` and is generated in your current working directory. You must manually copy it to the location specified in the forge-probe output. This is typically only necessary when the compute node that you are probing does not have write access to your home file system.

Check that the expected probe files are correctly installed with `--target-host`:

```
/path/to/forge/bin/map --target-host=list
```

This shows something like:

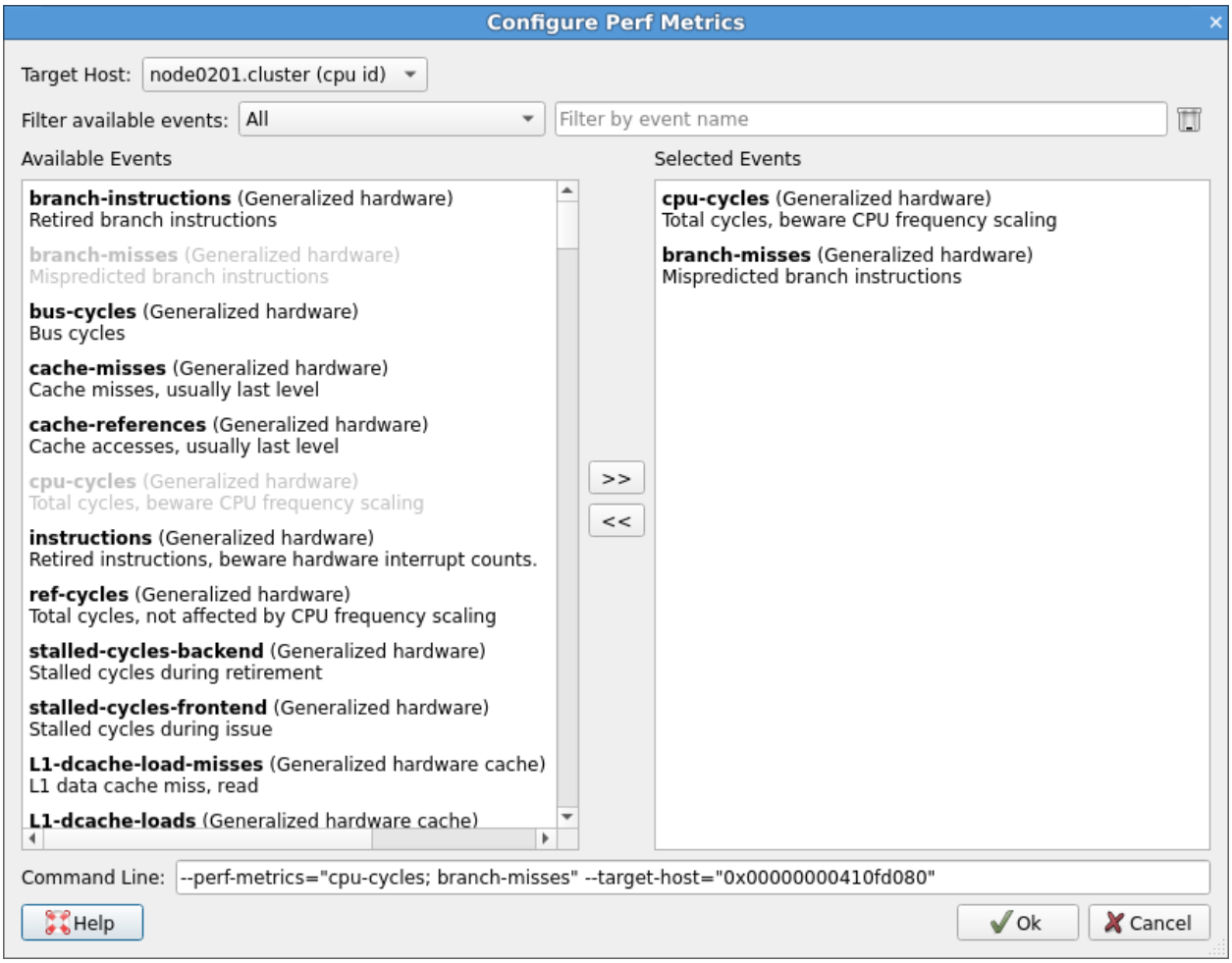
```
0x00000000420f5160    (thunderx2)   e.g. node07.myarmhost.com
GenuineIntel-6-4E     (skylake)     e.g. node01.myintelhost.com
```

If you have exactly one probe file installed, this is automatically assumed to be the target host. If there are multiple installed probe files, you must specify the intended target whenever you use the configurable Perf metrics feature. When using the command line, use the `--target-host` argument. You can specify the intended target CPU ID (such as, `0x00000000420f5160`), family name (such as, thunderx2), or a unique substring of the hostname (`myarmhost`).

3.18.4 Specify Perf metrics using the Run window

You can build your `--perf-metrics` argument from the *Run* window.

1. Click *Configure Perf metrics* in the *Run* window to open the *Configure Perf Metrics* window.



2. Select the target host from the drop-down list of installed hosts (see *Probe target hosts*).
3. Double-click an event, or use the arrow buttons to add or remove events from this list.

Note: On the left of the window is the list of Perf events available on the currently selected host, and on the right is the list of events you have selected for tracking.

---

4. Filter the list of available events by typing a substring of characters in the *Filter* field.

Note: The bottom of the window displays a preview of the section of the command line with the `--perf-metrics` command, based on the currently selected list of events.

---

5. From the *File* menu, open the *Perf metric selection* dialog to help you construct a suitable `--perf-metrics` command line without starting a job, that you can copy into a queue submission script.

3.18.5 Specify Perf metrics using the command line

You can list available events for a given probed host using:

```
/path/to/forge/bin/map --target-host={hostname} --perf-metrics={options}
```

Where {*options*} is a semicolon separated list of options of perf event names. For example:

```
/path/to/forge/bin/map --profile --target-host={hostname} \
   --perf-metrics="cpu-cycles; bus-cycles; instructions" mpirun ...
```

`--perf-metrics` also can take `avail` to see the available events in a compact block, or `list`, to see those events listed one per line.

3.18.6 Specify Perf metrics using a file

The `--perf-metrics` argument can also take the name of a plain text file:

```
/path/to/forge/bin/map --profile --target-host={hostname} \
      --perf-metrics=./myevents.txt mpirun ...
```

`myevents.txt` lists the events to track on separate lines, such as:

```
cpu-cycles
bus-cycles
instructions
```

`--perf-metrics=template` outputs a more complex template that lists all possible events with accompanying descriptions. Redirect this output to a file and uncomment the events to track, for example:

```
/path/to/forge/bin/map --target-host={hostname} --perf-metrics=template > myevents.txt

vim myevents.txt

/path/to/forge/bin/map --profile --perf-metrics=myevents.txt mpirun ...
```

3.18.7 View events

To view Perf event counts:

- For Linaro MAP, see the **Metrics** view (*Metrics view*) under the **Linux Perf CPU events** preset.

  You can view the total number of events (over the entire program, or just within a selected time range) in the tooltip of the legend.

- In Linaro Performance Reports, see the **CPU Metrics** section of the report.

All Perf metrics are reported as events per second with a suitable SI prefix (such as, K, M, G) that is automatically determined.

The default values that are reported are the mean of means:

- The mean value is taken across all processes for each sample (averaging across processes).
- The mean value is taken of those per-sample results (averaging across time).

3.18.8 Advanced configuration

You can override the default settings used by Linaro MAP when making `perf_event_open` calls. Specify one or more flags in a preamble section in square brackets at the start of the perf metrics definition string (either on the command line or at the top of a template file).

```
/path/to/forge/bin/map --profile --target-host=myhost \
   --perf-metrics="[optional,noinherit]; instructions; cpu-cycles"
```

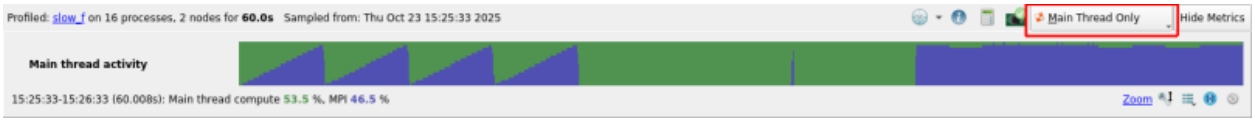
Possible options are:

- **[optional]:** Do not abort the program if the requested metrics cannot be collected. Set this if you want to continue profiling even if the no Perf metric results is returned.
- **[noinherit]:** Disable multithreading support (new threads will not inherit the event counter configuration). If you specified events, they are only collected on the main thread (in the case of MPI programs, the thread that called `MPI_thread_init`).
- **[nopinned]:** Disable pinning events on the PMU. If you have specified this, event counting might be multiplexed. We recommend that you do not do this as it interacts poorly with the Linaro Forge sampling strategy.
- **[noexclude=kernel]:** Do not exclude kernel events that happen in kernel space. This might require a more permissive `perf_event_paranoid` level.
- **[noexclude=hv]:** Do not exclude events that happen in the hypervisor. This is mainly for PMUs that have built-in support for handling this. Most machines require extra support for handling hypervisor measurements.
- **[noexclude=idle]:** Do not exclude counting software events when the CPU is running the idle task. This is only relevant for software events.

3.19 Main-thread, OpenMP, and Thread 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.

Linaro 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 might want 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 using the **View** menu or the **Thread Mode Selector**.



3.19.1 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 (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** view does not display 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.

3.19.2 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 time-slice 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 *Processes and cores window*) were being used for OpenMP.

This is the default mode for OpenMP programs. It is only available for programs where Linaro MAP detected an OpenMP region.

3.19.3 Thread 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 *Processes and cores window*.

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. The disadvantage is that it is 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.

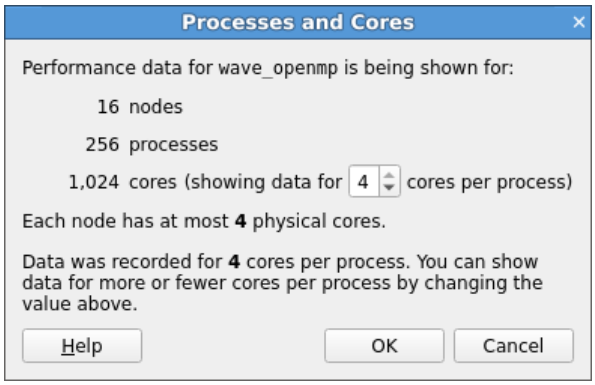
This mode is not used by default so must be explicitly selected. It is only available for multithreaded programs.

The **OpenMP Regions** view is not displayed in this mode.

3.20 Processes and cores window

The **Processes and Cores** window enables you to change the number of cores per process.

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, Linaro MAP tries to detect this situation and will rescale its expectations to the number of physical cores used by your program.



If this 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* value.

To open this dialog use the *View* ▸ *Processes and Cores* menu option, or click the *X cores (Y per process)* hyperlinked text in the application details section above the metric graphs.

3.21 Run MAP from the command line

Linaro MAP can be run from the command line.

3.21.1 Command line arguments

Here are a number of command line arguments to use with Linaro MAP.

`--capture-environment-variables`

Capture the environment variables of the profiled application in the Linaro MAP *Program details* dialog.

Note: Enabling `--capture-environment-variables` will capture all variables for your applications environment and store them in the Linaro MAP file. For security reasons, consider what information is stored in the environment before enabling this option.

`--no-mpi`

Run Linaro MAP with one process and without invoking `mpirun`, `mpiexec`, or equivalent.

`--queue`

Force Linaro MAP to submit the job to the queuing system.

--no-queue

Run Linaro MAP without submitting the job to the queuing system.

--view=<view>

Start Linaro MAP using the specified initial view mode (see *Main-thread, OpenMP, and Thread view modes*). This must be one of `(main|openmp|pthread)`. If the selected view is not available, the `main` view will be displayed.

--export=<output.json>

Export an input `.map` file to `output.json` in JSON format, without user interaction. e.g.:

```
map --export=output.json input.map
```

For the format specification see *JSON format*.

--profile

Generate an Linaro MAP profile but without user interaction. This will not display the Linaro MAP user interface. Messages are printed to the standard output and error. The job is not run using the queuing 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. For example, `--profile --export-functions=foo.csv ...`

Note: Exported functions will be from Main Thread Only view mode.

--select-ranks=<ranks>

Only collect profiling data from the specified set of ranks. `<ranks>` is a comma separated list of ranks and/or rank ranges. Example `5,6-10`.

--start-after=<time>

Start profiling `<time>` seconds after the start of your program. Use this in conjunction with `--stop-after` to focus Linaro 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. Use this in conjunction with `--start-after` to focus Linaro MAP on a particular time interval of the run of your program.

--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 Linaro 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 Linaro MAP without the user interface, so they will need to be specified for each profiling session. When running Linaro MAP in user interface mode, the effect of these settings will be displayed in the **Metrics** section of the **Run** dialog, where you can further refine the settings. These settings will then persist to the next user interface session.

--list-metrics

Lists the metric IDs which can be explicitly enabled or disabled using `--enable-metrics` and `--disable-metrics`.

--cuda-kernel-analysis

Enables CUDA kernel analysis mode, providing line level profiling information on CUDA kernels running on a NVIDIA GPU at the cost of potentially significant overhead. See *CUDA Kernel analysis*.

--cuda-transfer-analysis

Enables CUDA transfer analysis mode, providing insights into the memory transfers managed via CUDA. See *Memory transfers analysis*.

--report=<types>

Generate a report in addition to the standard `.map` file when using `--profile`. `<types>` is a either a comma separated list of one or more from `(txt|csv|html|summary)`, or `none`.

For `txt`, `csv` and `html` a Linaro Performance Reports file of the corresponding format and file extension will be created using the same base file name as the `.map` file that is generated.

`summary` prints a minimal summary report to the standard output. This is the default behavior if no `--report` option is provided.

`none` (mutually exclusive with all other `<type>` values) blocks the creation of any type of report other than the `.map` file, slightly reducing Linaro MAP's post-processing time.

To generate a report from an existing `.map` file, See *Summarize an existing MAP file*.

--spe=<type>

For Arm®v8.2 or greater targets implementing the option Arm Statistical Profiling Extension (SPE) this option enables gathering the specified type of SPE events. `<type>` may be one of `(cache|mispredict|tlb|sve-predicate)`. See *Arm Statistical Profiling Extension (SPE)*.

--perf-metrics=<file>

Defines one or more Linux perf event counters to sample when profiling. These will be displayed as metric graphs by Linaro MAP and report lines within Linaro Performance Reports. Often used along with `--target-host`. See *Specify Perf metrics using a file* and *Specify Perf metrics using the command line*.

--target-host=<hostname>

Specifies the intended CPU from which perf-metrics are to be acquired as described by an accompanying `--perf-metrics` argument. The known CPU types can be listed with `--target-host=list`. CPUs may be added to this list by using the ``forge-probe` utility, see *Probe target hosts*.

--build-directories=PATH1[:PATH2]

Colon delimited list of build directories to search for compiler optimization reports.

--embed-compiler-remarks

Embed compiler remarks into the generated `.map` file.

--exclude-compiler-remarks=TYPE1[,TYPE2,...]

Comma delimited list of remark types or optimization passes to exclude from embedding into the `.map` file. Types include Failed, Success, or Information. Optimization passes are compiler-specific.

--baseline=<file>

Provide another .map file to compare results against.

When running without the user interface, 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 program. There is no need to pass the number of threads to Linaro MAP as an argument.

```
OMP_NUM_THREADS=8 map --profile ... > <output-file>
```

3.21.2 Profile MPMD programs

The easiest way to profile MPMD programs is by using Express Launch to start your program.

To use Express Launch, prefix your normal MPMD launch line with map. For example, to profile an MPMD program without user interaction you can use:

```
map --profile mpirun -n 1 ./main : -n 2 ./worker
```

For more information about Express Launch, and compatible MPI implementations, see *Express Launch (MAP)*.

3.21.3 Profile MPMD programs without Express Launch

The command to create a profile from an MPMD program using Linaro MAP is:

```
map <map mode> --np=<#processes> --mpiargs=<MPMD command> <one MPMD program>
```

This example shows how to run Linaro MAP without user interaction using the flag --profile:

```
map --profile --np=16 --mpiargs="-n 8 ./exe1 : -n 8 ./exe2" ./exe1
```

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.

3.22 Export profiler data in JSON format

You can export the profiler data in machine readable JSON format.

To export as JSON, open a .map file in Linaro MAP. Then the profile data can be exported by selecting *File* ▸ *Export Profile Data as JSON*.

For a command-line option, see *Run MAP from the command line*.

3.22.1 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.

See an example of profile data exported to a JSON file in *Example JSON output*.

3.22.1.1 `info` object

`command_line`

Type: String

Command line call used to launch the profiled program. For example,

```
aprun -N 24 -n 256 -d 1 ./my_exe
```

`machine`

Type: String

Hostname of the node on which the executable was launched.

`notes`

Type: String

A short, optional, user-provided description of the run or other notes on configuration and compilation settings. This is specified by setting the environment variable `FORGE_NOTES` before running Linaro MAP.

`number_of_nodes`

Type: Number

Number of nodes run on.

`number_of_processes`

Type: Number

Number of processes run on.

`runtime`

Type: Number

Runtime in milliseconds.

`start_time`

Type: String

Date and time of run in ISO 8601 format.

`create_version`

Type: String

Version of MAP used to create the `.map` file.

`metrics`

Type: 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 or nodes for each sample.

In many cases, the values over all nodes is the same. This means that 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

Type: 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

Type: 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

Type: Object

Number of cores available per node.

memory_per_node

Type: Object

RAM installed per node.

nvidia_gpus_count

Type: Object

Number of NVIDIA GPUs per node.

nvidia_total_memory

Type: Object

NVIDIA GPU frame buffer size per node.

rocm.gpus_count

Type: Object

Number of AMD GPUs per node.

rocm.gpu_total_memory

Type: Object

AMD GPU Video Random Access Memory (VRAM) size per node

num_omp_threads_per_process

Type: Object

Number of OpenMP worker threads used per process.

3.22.1.2 `samples` object

`count`

Type: Number

Number of samples recorded.

`window_start_offset`

Type: 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`

Type: 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 Main Thread, OpenMP, and pthreads as described in *Main-thread, OpenMP, and Thread view modes*. 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 Linaro 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 program uses more cores than Linaro MAP assumes the maximum number of cores per process can be. This can be due to middleware services launching helper threads which, unexpectedly to Linaro 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`

Type: 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 *Metrics*. All metrics recorded in a run are present in the JSON, including custom metrics. The names and descriptions of all core Linaro MAP metrics are given in *Metrics*. It is assumed that a user including a custom metrics library is aware of what the custom metric is reporting. See the Custom Metric Plugin Interface documentation.

3.22.2 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.

3.22.2.1 Description of categories

The following is the list of all of the categories. Only available categories are exported, see *Main Thread Activity Categories* and *OpenMP and Pthreads Activity Categories* below.

- `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.

3.22.2.2 Main Thread Activity Categories

The following categories are available in the `main_thread` `activity` view.

- `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`

3.22.2.3 OpenMP and Pthreads Activity Categories

The following categories are available in the `openmp` and `pthreads` `activity` views. The `pthreads` data corresponds to the *Thread mode* in the MAP UI.

- `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`

3.22.3 Metrics

The following list contains the core metrics reported by Linaro 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 *Metrics*

- CPU Instructions: see *CPU instructions*
  - `instr_fp`
  - `instr_int`
  - `instr_mem`
  - `instr_vector_fp`
  - `instr_vector_int`
  - `instr_branch`
  - `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 *CPU time*
  - `cpu_time_percentage`: See *CPU time* in *CPU time*
  - `user_time_percentage`: See *User-mode CPU time* in *CPU time*
  - `system_time_percentage`: See *Kernel-mode CPU time* in *CPU time*
  - `voluntary_context_switches`: See *Voluntary context switches (1/s)* in *CPU time*
  - `involuntary_context_switches`: See *Involuntary context switches (1/s)* in *CPU time*
  - `loadavg`: See *System load* in *CPU time*
- I/O: see *I/O*
  - `rchar_rate`: See *POSIX I/O read rate (B/s)* in *I/O*
  - `wchar_rate`: See *POSIX I/O write rate (B/s)* in *I/O*

  - `bytes_read`: See *Disk read transfer* in *I/O*
  - `bytes_written`: See *Disk write transfer* in *I/O*
  - `syscr`: See *POSIX read syscall rate* in *I/O*
  - `syscw`: See *POSIX write syscall rate* in *I/O*
- 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 *Memory*
  - `rss`: See *Memory usage in bytes (Resident Set Size)* in *Memory*
  - `node_mem_percent`: See *Node memory usage* in *Memory*
- MPI: see *MPI calls*
  - `mpi_call_time`: See *MPI call duration (ns)* in *MPI calls*
  - `mpi_sent`: See *MPI sent* in *MPI calls*
  - `mpi_recv`: See *MPI received* in *MPI calls*
  - `mpi_calls`: Number of MPI calls per second per process
  - `mpi_p2p`: See *MPI point-to-point* in *MPI calls*
  - `mpi_collect`: See *MPI collective operations* in *MPI calls*
  - `mpi_p2p_bytes`: See *MPI point-to-point bytes* in *MPI calls*
  - `mpi_collect_bytes`: See *MPI collective bytes* in *MPI calls*
- Accelerator (NVIDIA): see *Accelerator*
  - `nvidia_gpu_usage`: See *GPU utilization* in *Accelerator*
  - `nvidia_memory_used_percent`: See *GPU memory usage* in *Accelerator*
  - `nvidia_memory_used`: GPU memory usage in bytes
- Accelerator (AMD): see *Accelerator*
  - `rocm.gpu_utilization`: See *GPU utilization* in *Accelerator*
  - `rocm.gpu_memory_used_percent`: See *GPU memory usage* in *Accelerator*
  - `rocm.memory_utilization`: See *GPU memory utilization* in *Accelerator*
  - `rocm.gpu_memory_used`: GPU memory usage in bytes
- Energy: see *Energy*
  - `nvidia_power`: See *NVIDIA GPU power usage (mW/node)* in *Energy*
  - `rocm.gpu_power`: See *AMD GPU power usage (mW/node)* in *Energy*
  - `rapl_power`: See *CPU power usage (W/node)* in *Energy*
  - `system_power`: See *System power usage (W/node)* in *Energy*
  - `rapl_energy`: CPU energy, integral of `rapl_power (J)`

– `system_energy`: CPU energy, integral of `system_power (J)`

3.22.4 Example JSON output

In this section an example is given of the format of the JSON that is generated from a Linaro 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 ],
```

(continues on next page)

(continued from previous page)

```
            "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]
         }
      }
} }
```

3.23 NVIDIA GPU profiling

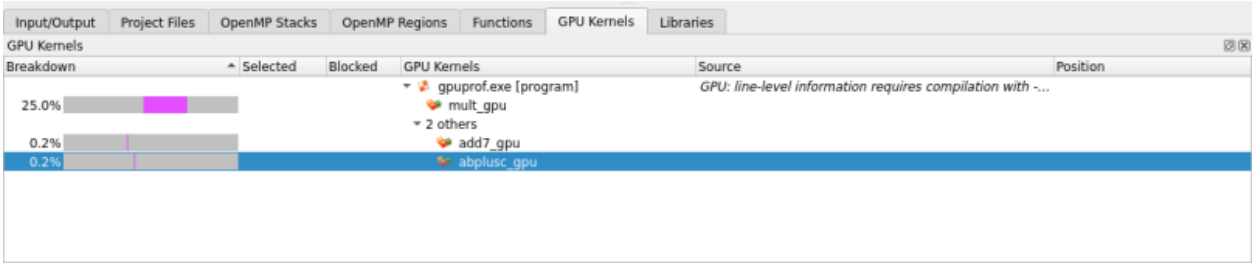
You can use the GPU profiling capabilities when working with NVIDIA CUDA programs.

See also *Accelerator* metrics.

3.23.1 GPU Kernels tab

When profiling programs that use NVIDIA GPUs, GPU kernels that can be tracked by the NVIDIA CUDA Profiling Tools Interface (CUPTI) display in the *GPU Kernels* tab.

Note: For information about currently supported software versions, see *Reference table*.

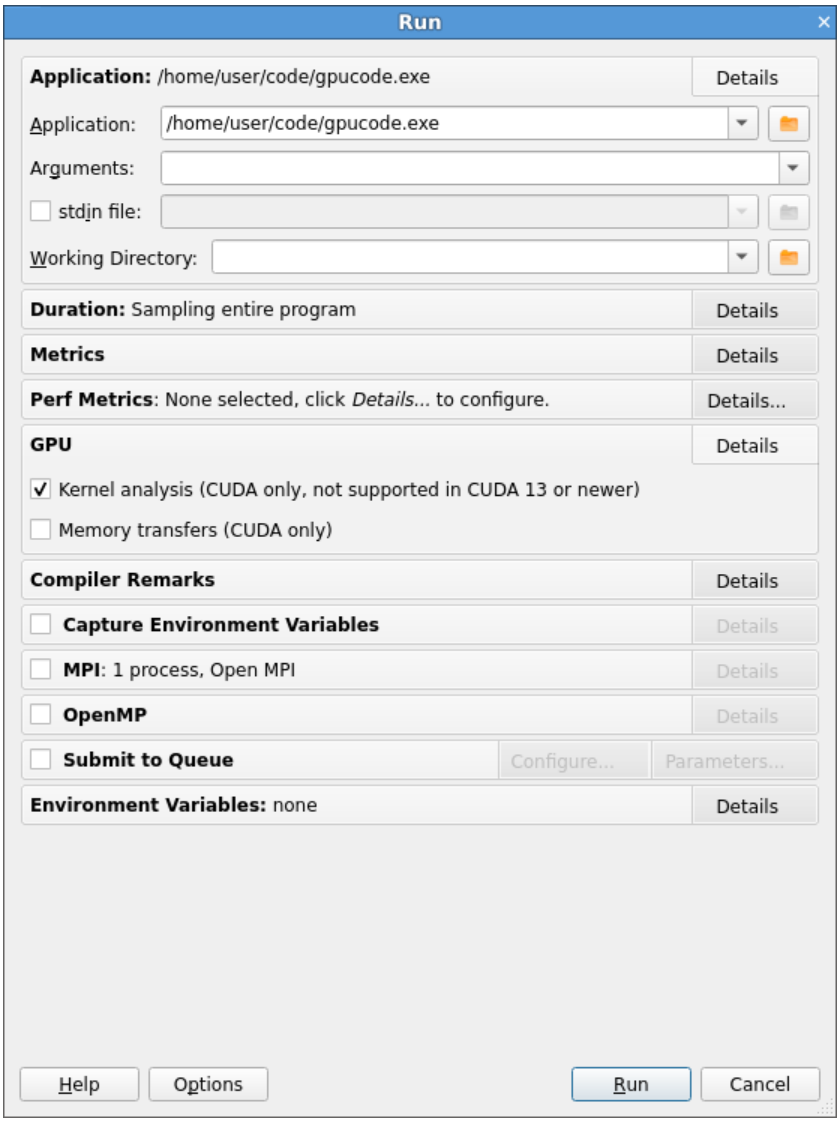


This lists the CUDA kernels that were detected in the program alongside graphs indicating when those kernels were active. If multiple kernels are identified in a process within a particular sample, they are given equal weighting in this graph. Selecting a single GPU Kernel results in the *Source Code* viewer jumping to this kernel if debug information is available.

3.23.2 CUDA 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 *Run* dialog or from the command line with `--cuda-kernel-analysis`.

Note: CUDA kernel analysis is not supported with CUDA 13.



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.

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.

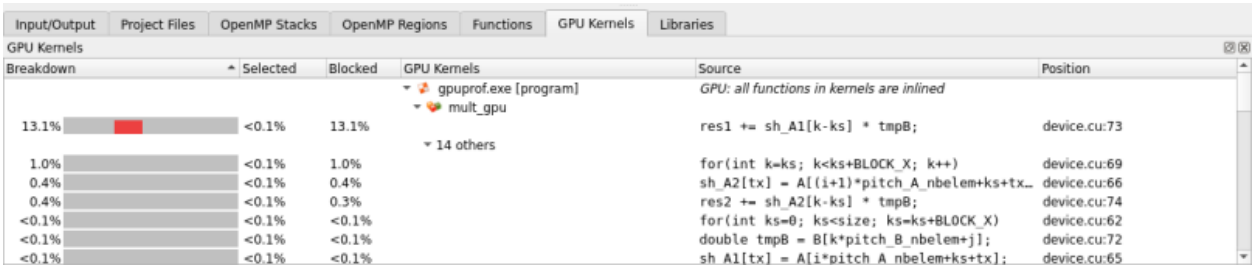
Memory throttle
Warp is blocked because there are too many pending memory operations.

Not selected
Warp was ready to issue, but some other warp issued instead.

Other
Miscellaneous stall reason.

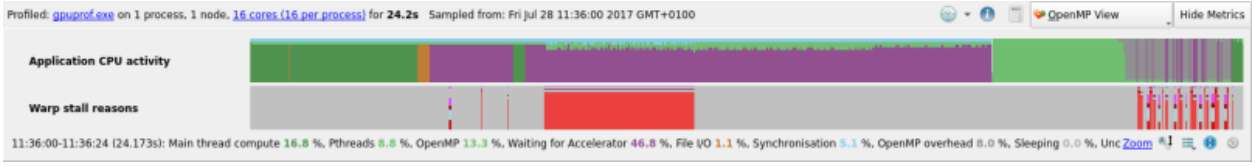
Dropped samples
Samples dropped (not collected) by hardware due to backpressure or overflow.

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 Linaro MAP.



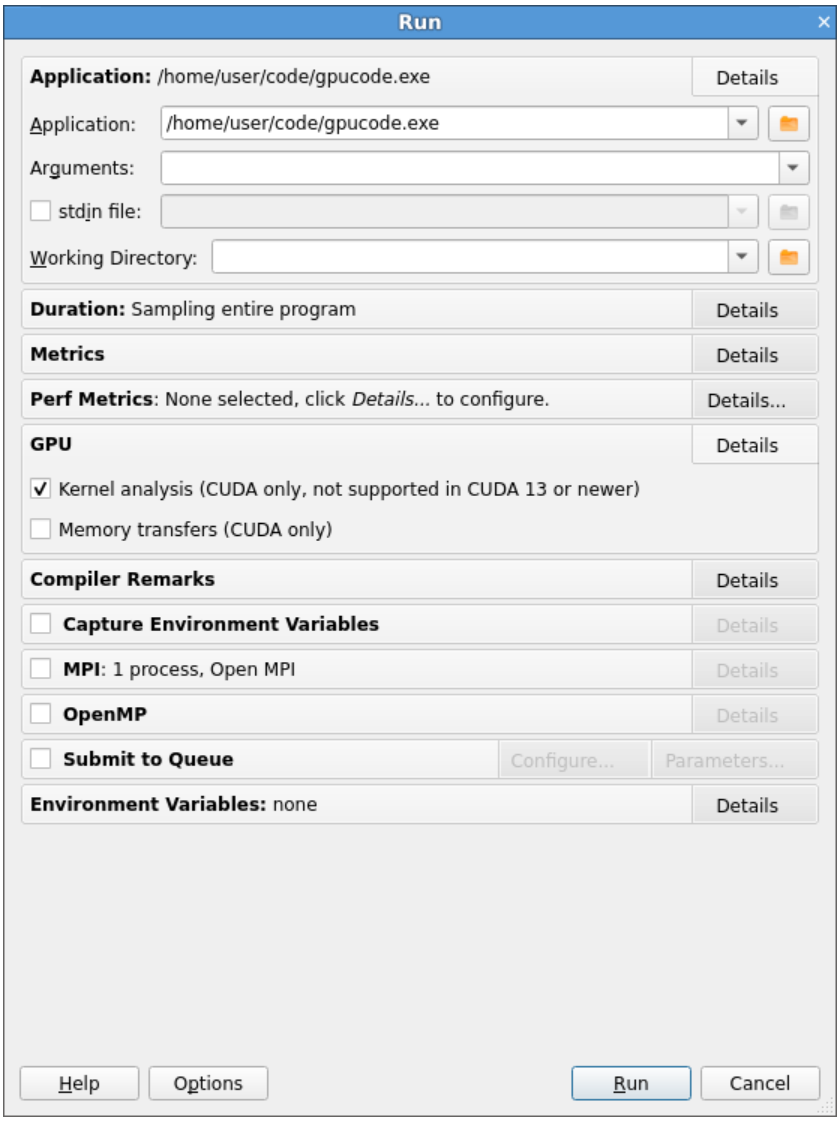
Note: Warp stalls are only reported per-kernel, so it is not possible to obtain the times within a kernel invocation at which different categories of warp stalls occurred. As function calls in CUDA kernels are automatically fully inlined it is not possible to see a stack trace of code within a kernel on the GPU.

Warp stall information is also present in the *Source code editor* (*GPU programs*), the *Selected lines* view (*NVIDIA GPU CUDA profiles*), and in a **Warp stall reasons** graph in the *Metrics* view (*Metrics view*).

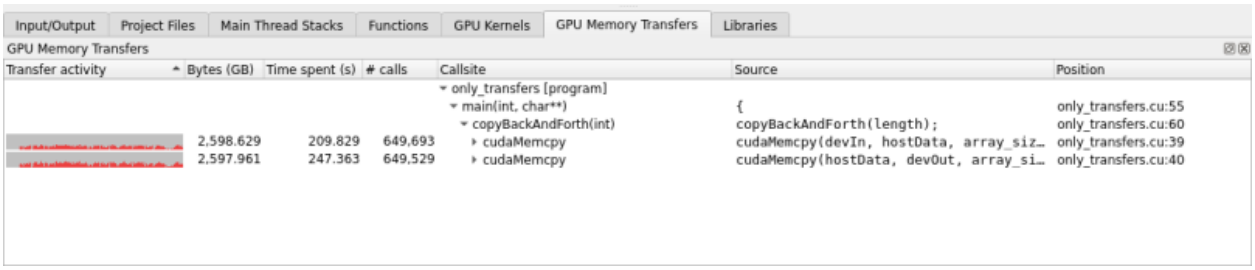


3.23.3 Memory transfers analysis

CUDA memory transfer analysis mode is an advanced feature that provides insight into the memory transfers managed via CUDA, `cudaMemcpy` and similar calls. When profiling programs that use CUDA 11.0 and later, this mode can be enabled from the *Run* dialog or from the command line with `--cuda-transfer-analysis`.



When enabled, the *GPU Memory Transfers* tab shows the locations in the code where memory transfers involving CUDA devices were initiated.



The *GPU Memory Transfers* tab contains a tree view of the stack traces from which GPU memory transfers were initiated. The columns are as follows:

Transfer activity:
: A visual representation of when GPU memory transfers were in progress. This is an approximation of the wall-clock time in which at least one GPU transfer was active.

Bytes:
: The number of bytes transferred in all GPU memory transfers started in the selected range of samples.

Time spent:
: The sum of the time spent in each GPU memory transfer started in the selected range of samples. If multiple memory transfers were in progress simultaneously then this number will be larger than the actual amount of wall-clock time in which transfers were in progress.

 # calls:
: The total number of GPU memory transfer calls that were started in the selected range of samples.

Callsite:
: The stack frames where GPU memory transfers were initiated. Expand to navigate the full stack down to the `cudaMemcpy*` call.

Source:
: The source code line for this frame, if available. The source files must be available and the program must have been compiled with debug information enabled.

Position:
: The source file and line number.

Note: Additional information can be found in the tooltip for each line, including the min/max/average bytes transferred per memory transfer call, and the min/max/average time spent in each call.

3.23.4 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
```

3.23.5 Performance impact

3.23.5.1 CUDA kernel analysis

Note: CUDA kernel analysis is not supported with CUDA 13.

Enabling the CUPTI sampling will impact the target program in the following ways:

- 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.
- 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.
- 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 analyze 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 *Accelerator*.

3.23.5.2 CUDA memory transfer analysis

Enabling the CUDA memory transfer analysis feature will impact the target program in the following ways:

- Time overhead will be incurred at every CUDA memory transfer call. The impact of this will depend on the frequency of such calls. This overhead, if significant, will be shown by Linaro MAP as *Profiler callsite tracing overhead*.
- Minor memory overhead dependent on the number of unique stack traces that lead to CUDA memory transfer calls. This is unlikely to be noticeable unless the number of unique callsites is very large.

This overhead will primarily impact the host (CPU). GPU kernel performance should be unaffected unless the host overhead delays one or more memory transfers that a GPU kernel needs in order to progress.

3.23.5.3 Overhead mitigation

When profiling CUDA code it may be useful to only profile a short subsection of the program so time is not wasted waiting for CUDA kernels you do not intend to examine. See *Profiling only part of a program* in *Profile a program* for instructions.

3.23.6 Customize NVIDIA GPU profiling behavior

The interval at which CUPTI samples GPU warps can be modified by the environment variable `FORGE_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.

Using CUDA 11.0+ on GPUs with compute capability 7.0+, the interval at which CUPTI samples GPU warps can also be modified by providing an integer value $5 \leq x \leq 31$ to the environment variable `FORGE_SAMPLER_GPU_INTERVAL`. This sets the interval in cycles to exactly 2^x.

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 when the kernel completes. Overheads of twice as long as the kernel's normal runtime have been observed. We recommend that the CUPTI sampling interval is not reduced.

3.23.7 Known issues for NVIDIA GPU profiling

There are a few known issues for NVIDIA GPU profiling.

- GPU profiling is only supported with CUDA 8.0 and later. For information about currently supported software versions, see *Reference table*.
- GPU memory transfer analysis is only supported with CUDA 11.0 and later.
- GPU kernel analysis is not supported for CUDA 13.
- 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 CUDA Fortran are not yet supported by Linaro 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 or CUDA memory transfer analysis mode can have a significant performance impact as described in *Performance impact*.
- GPU profiling is not supported when statically linking the Linaro Forge sampler library.
- Stopping GPU profiling mid-process can prevent the *GPU Kernels* tab displaying, and might not report the kernel samples. This occurs when using `--stop-after` or the *Stop and Analyze* button. For better results, run the process for a longer time period with longer running kernels. When kernel samples are reported, they can be truncated.
- You may experience a hang during profiling when CUDA Kernel Analysis mode is enabled for CUDA Toolkit >= 12.0.1 and < 12.2.2. If you encounter this issue, please contact Forge Support.
- The CUDA Kernel Analysis feature is not supported for Blackwell GPUs and beyond. This is because the PC Sampling Activity API used by this feature has been deprecated by NVIDIA and has dropped support for Blackwell.

3.24 AMD GPU profiling

You can use the GPU profiling capabilities when working with AMD ROCm programs. AMD GPU Profiling is initialized by default with a Linaro Forge license with ROCm support. Contact Forge Support for upgrade information.

See also *Accelerator* metrics.

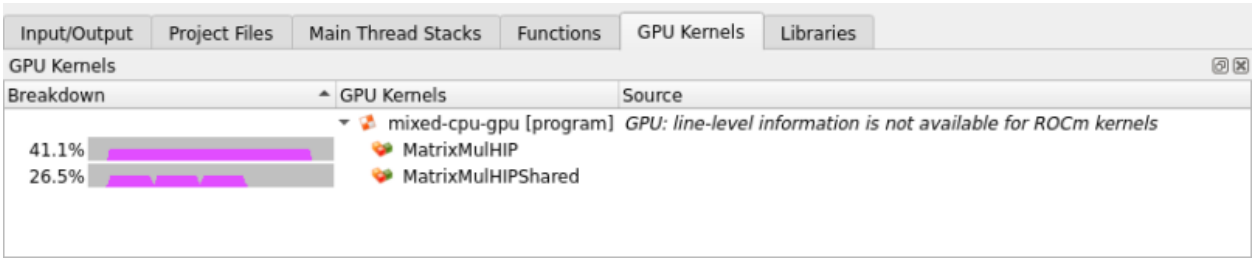
3.24.1 GPU Kernels tab

When profiling programs that use the AMD ROCm Toolkit, GPU kernels that can be tracked by AMD profiling interfaces will display in the *GPU Kernels* tab.

This is supported by:

- the AMD `rocprofiler-sdk` interface for ROCm Toolkit >= 6.2, and
- the AMD `roctracer` interface for ROCm Toolkit < 6.2.

Note: For information about currently supported software versions, see *Reference table*.



The *GPU Kernels* tab lists the HIP kernels that were detected in the program alongside graphs indicating when those kernels were active. If multiple kernels are identified in a process within a particular sample, they are given equal weighting in this graph. Selecting a single GPU Kernel results in the *Source Code* viewer jumping to this kernel if debug information is available.

3.24.2 Compilation

To see the source code in Linaro MAP, compile your program with the debug flag. When profiling with Linaro MAP, always keep optimization flags turned on.

```
hipcc device.cpp -o run_program -g -O3
```

3.24.3 Known issues and limitations

There are a few known issues for AMD GPU profiling.

- The environment variable `HSA_ENABLE_INTERRUPT=0` is enabled by default when running HIP applications with Linaro MAP. This is to address an intermittent hang which has been observed. The environment variable `FORGE_NO_HSA_INTERRUPT_ENABLE_0=1` restores the default behavior of HIP.
- Line-level information and program counter (PC) sampling are not available for ROCm kernels.
- GPU memory transfer analysis is not available for ROCm kernels.
- OpenMP target offload regions are not detected as GPU Kernels when using the AMD `roctracer` interface (for ROCm < 6.2) in Linaro 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 AMD 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.
- GPU profiling is not supported when statically linking the Linaro Forge sampler library.
- Linaro MAP may not initialize if ROCm 6.1.x, where 0 <= x < 5, is detected in the environment. This issue is resolved by reverting to AMD's `roctracer` interface in Linaro MAP by setting the environment variable `FORGE_DISABLE_ROCM_V3_PROFILING=1`.
- Linaro MAP may not initialize if ROCm 6.1.5 is detected in the environment and there are other ROCm > 6.1 installations on the system. This issue is resolved by reverting to AMD's `roctracer` interface in Linaro MAP by setting the environment variable `FORGE_DISABLE_ROCM_V3_PROFILING=1`.

3.25 Python profiling

The Python profiling capabilities can be used to find and resolve bottlenecks for your Python codes.

For supported Python versions, see *Reference table*.

3.25.1 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.

3.25.1.1 About this task

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 `FORGE_SAMPLER_DISABLE_PYTHON_PROFILING=1`.

Linaro MAP supports Python profiling with the following features:

- Profiles Python scripts running under the CPython interpreter.
- Profiles Python scripts running under virtual environments.
- Profiles Python scripts that import modules which perform MPI on the main thread, such as mpi4py.
- Profiles Python scripts that import modules which use OpenMP.
- Profiles Python scripts that use the `threading` module.

Note: Linaro 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, you can set environment variables (*Starting a program*) or you can import the mpi4py module.

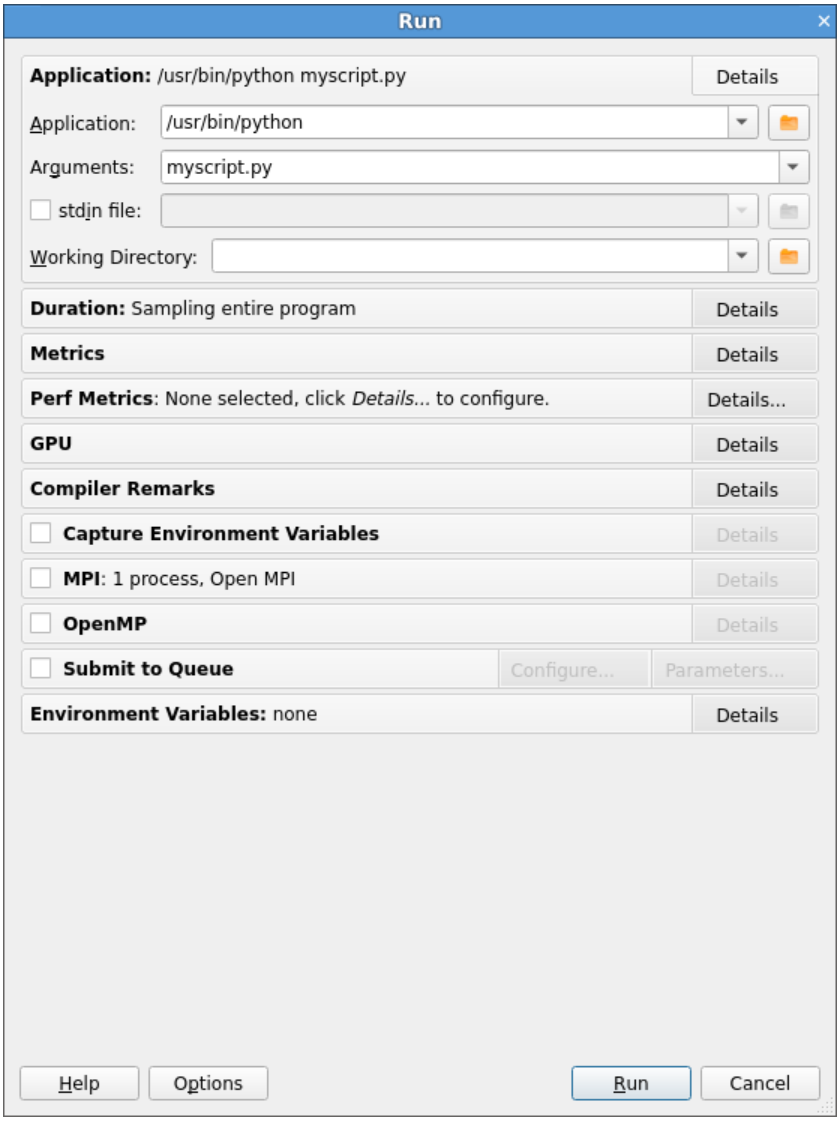
3.25.1.2 Procedure

1. Check that the Python script runs successfully:

```
$ python myscript.py
```

2. To profile the Python script with Linaro MAP, prepend the run command with `map`:

```
$ map python myscript.py
```



3. Click **Run** and wait for Linaro MAP to finish profiling the Python script.
4. View the profiling results in Linaro MAP.

3.25.1.3 Results

When Linaro MAP finishes profiling the Python script, it saves a `.map` file in the current working directory and opens it for viewing in the user interface (unless you are using the offline feature).

3.25.1.4 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.

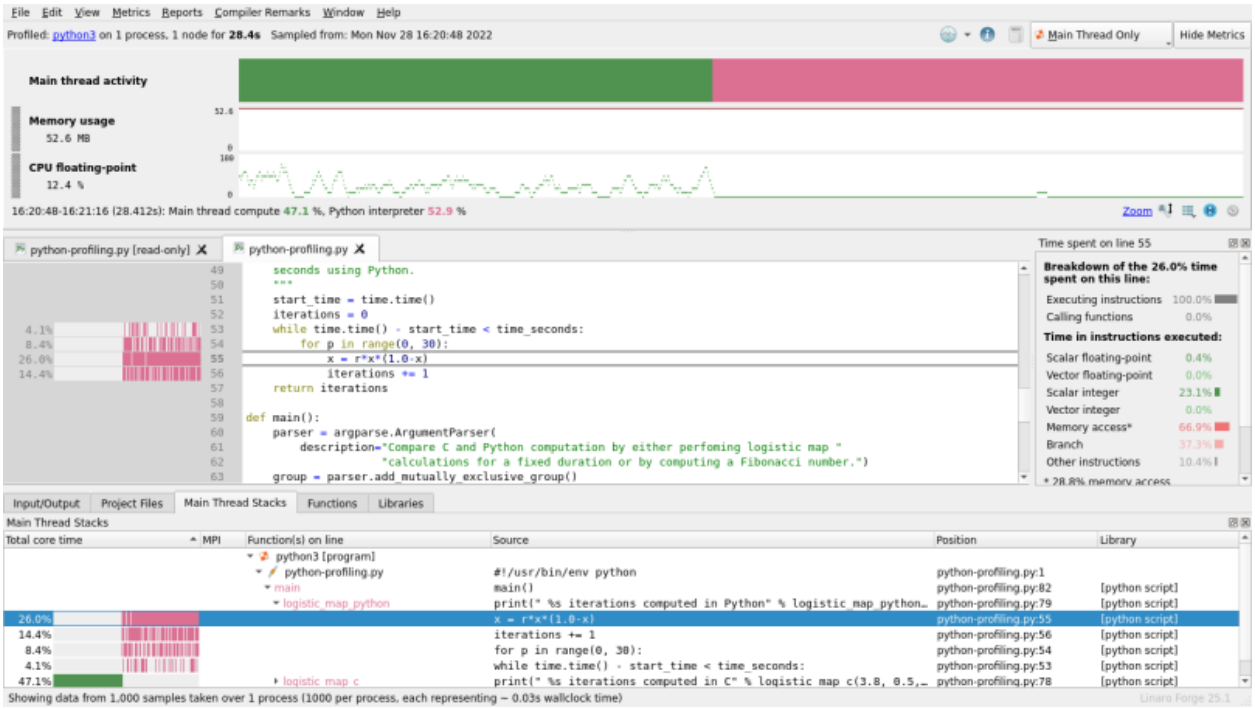
```
$ make -f python-profiling.makefile
```

2. Start Linaro MAP.

```
$ ../bin/map python ./python-profiling.py --seconds 15
```

3. Click **Run**.
4. Wait for Linaro MAP to finish analyzing samples after the Python script has completed.

> **Note:** The Linaro MAP user interface launches showing the Python script and the line in the script where the most time was spent is selected.



5. Locate the first `logistic_map_c` stack frame in the **Main Thread Stacks** view. The callout to the C function is appended under `main` Python stack frame.

3.25.1.5 Next steps

- Examine the **Main Thread Activity** graph (*Metrics view*) for an overview of time spent in Python code compared with non-Python code.
- View source code lines (*Source code (MAP)*) 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 (*Selected lines view*).
- View a breakdown of time spent in different code paths in the **Main Thread Stacks** view (*Stacks view*).

3.25.1.6 Related information

- For more information on using Linaro MAP, see *Get started with MAP*.
- For information on debugging Python scripts with Linaro DDT, see *Python debugging*.
- For more information about supported Python versions, see *Reference table*.

3.25.2 Known issues for Python profiling

There are a few known issues for Python profiling.

- Linaro 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 Linaro MAP.
- mpi4py uses some MPI functions that were introduced in MPI version 3. For example `MPI_Mrecv`. Linaro 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.

3.26 Performance analysis with Caliper instrumentation

Caliper is a program instrumentation and performance measurement framework. It is a performance analysis toolbox in a library, that enables you to insert performance analysis capabilities directly into programs, and activate them at runtime.

Caliper is intended for use with HPC programs, but works for any C/C++/Fortran program on Unix/Linux.

When Linaro MAP profiles a program instrumented with Caliper source code annotations, the stack of Caliper attributes with keys of interest is taken alongside regular samples. The keys of interest use the same attribute names as the Caliper high-level API (function, loop, statement, annotation). In Linaro MAP, you can see where time was spent in any set of these key-attribute pairs.

3.26.1 Get Caliper

Download Caliper (version 2.0.1 or later) from GitHub.

Build and install Caliper, then use it to instrument programs of your choice as described in the Caliper documentation:

- Read a summary
- Full documentation
- Pre-instrumented examples (LULESH2 and Quicksilver)

3.26.2 Annotate your program

Typically, we integrate Caliper into a program by marking source-code sections of interest with descriptive annotations. Linaro MAP can connect to Caliper and access the information provided by these annotations.

3.26.2.1 Annotate in C/C++

Linaro MAP supports Caliper's high-level C/C++ API for annotating functions, loops (although loop iterations are not recorded) and code regions. Refer to the Annotation API in the Caliper documentation, for details and examples.

Note: Using the low-level API for C/C++ applications is not recommended. Low-level API calls must be nested, otherwise they are not supported.

3.26.2.2 Annotate in Fortran

For Fortran programs, the low-level API must be used to emulate the high-level API. These label types are supported: function, loop, statement, annotation. They match the attribute names used by the high-level API.

Note: Caliper regions must be nested. Close all inner regions before closing an outer region.

```
use Caliper
call cali_begin_string_byname('function', 'myFunction')
...
call_end_byname('function')
```

3.26.3 Analyze your program

Use Caliper with Linaro MAP to get a quick idea of how much time is spent in the various phases of your program, to help you decide where to focus your optimization efforts.

3.26.3.1 Procedure

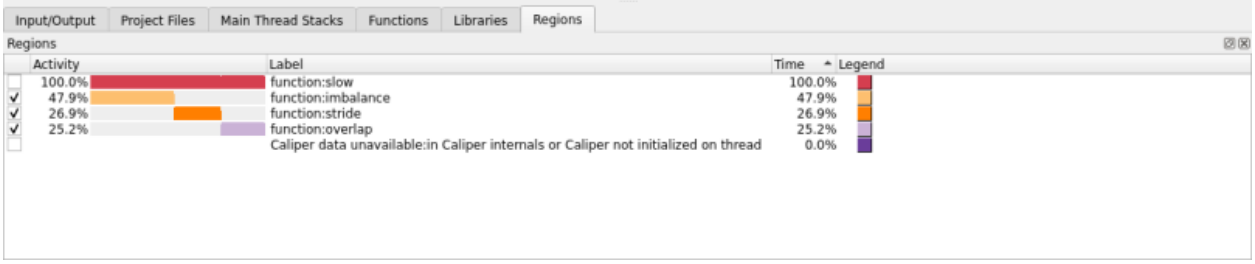
1. Instrument your program with Caliper annotations (or use one of the Caliper-provided examples).
2. Dynamically link your program against Caliper, and profile it with Linaro MAP.

   Note: Linaro Forge automatically detects the Caliper regions of your code, and does not require you to set any specific flags or options.

   When profiling, Caliper might prompt you to enable Caliper services. This is not required. You can enable Caliper services if you want to, but this increases overhead without providing any additional data to Linaro MAP. Enabling it might produce Caliper output files you can manually examine.

   In the *Metrics* view, the *Selected regions* chart (initially empty) displays below the *Main thread activity* chart.

3. Switch to the *Regions* tab at the bottom of the screen to view the regions of your code that you annotated with Caliper.



4. Optionally, select the *Legend* column in the *Regions* tab to change the color that is used to represent a region.

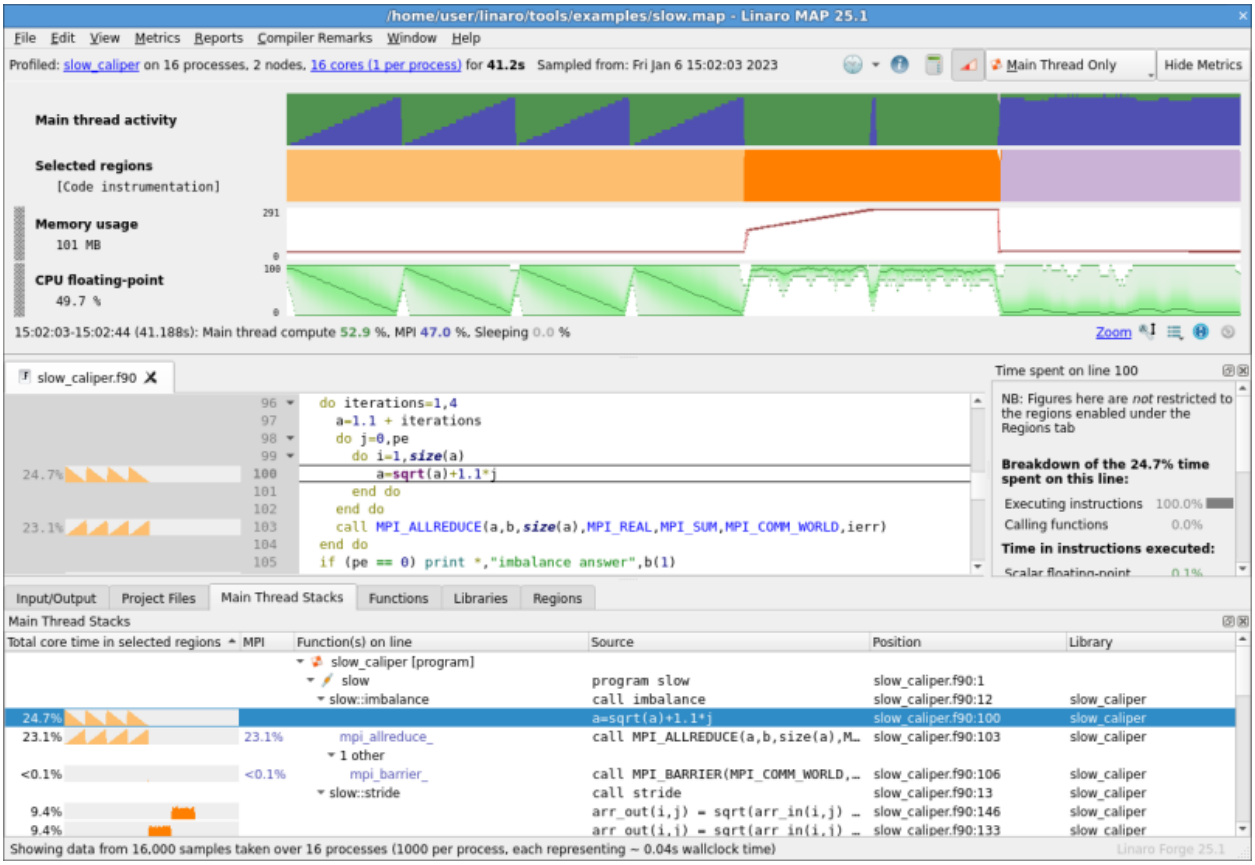
   Note: Forge assigns a color for each area of the code where Caliper instrumentation occurs.

5. Select one or more regions that you want to display in the *Selected regions* chart in the *Metrics* view.

   Note: Press `Ctrl` (or `Shift`), and click to select multiple regions.

   Periods of time where the program was inside that region show in the *Selected regions* chart.

6. Enable *View* ▸ *Regions-focused view*.



3.26.3.2 Results

In each of the charts, the horizontal axis indicates wall clock time.

The *Selected regions* chart, and the *Main thread activity* chart above it, use the same colors and scales as the per-line sparkline charts.

The *Application activity* chart can sometimes display in place of, or in addition to, the *Main thread activity* chart. This depends on the type of code that you profile, and the display mode you use (main thread only, threads, or OpenMP).

For more information about how to read these charts, see *View source code (MAP)*.

The button in the *Selected regions* chart toggles on and off the display of the selected region sparklines in the *Main Thread Stacks* view. These sparklines provide more detail about the percentage of the core time in the selected regions.

Activating the button can also modify the timeglyphs for sparklines in other views (*Thread Stacks*, *OpenMP Stacks*, *OpenMP Regions*, *Functions*)

Application activity timeglyphs in the PSVs, *Functions* view, and **Source code** view switch to showing the time in the currently selected set of regions.

3.26.4 Guidelines

Recommendations for using Caliper.

- The expected usage is that you will only have a few regions enabled at any one time.

  Note: The **Application activity** graph displays the deepest enabled region in any stacks to display.

- Only the default Caliper channel is sampled by Linaro MAP.
- Neither Linaro MAP or Caliper propagate Caliper attributes set on the main thread to OpenMP worker threads when entering an OpenMP parallel region.

3.26.4.1 Next steps

Right-click on a region in the *Regions* tab to access further options. From here you can:

- Enable or disable all regions at once.
- Automatically reassign colors to regions based on the percentage of time in each in the current selected time range.
- Copy a text representation of the tab, or export it to a file.

3.26.4.2 Related information

- Refer to the following paper for more background on Caliper: David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. Caliper: Performance Introspection for HPC Software Stacks. In Supercomputing 2016 (SC16), Salt Lake City, Utah, November 13-18, 2016. LLNL-CONF-699263.
- Refer to the Annotation API in the Caliper documentation for full details and examples of annotations.
- Read a summary about Caliper
- Caliper full documentation
- Pre-instrumented examples (LULESH2 and Quicksilver)

3.27 Arm Statistical Profiling Extension (SPE)

The Arm Statistical Profiling Extension (SPE) is an optional feature in Arm®v8.2 hardware that allows CPU instructions to be sampled and associated with the source code location where that instruction occurred.

Linaro MAP can use Arm SPE to list the source code lines that frequently trigger certain hardware events such as:

- Branch mispredicts
- Level 1 data cache (L1D) refills
- Last level cache (LLC) misses
- Translation lookaside buffer (TLB) walks
- SVE partial predicate and empty predicate events

Note: Arm SPE counts attributed to a source code line are not absolute counts and should be used only in comparisons with counts for other lines from the same profile.

3.27.1 Arm SPE Prerequisites

The Arm Statistical Profiling Extension (SPE) is an optional feature in Arm®v8.2 and is present in CPUs, such as the Neoverse™ N1 and V1. In addition, using this Linaro MAP feature has the following prerequisites:

- Use Linux kernel 5.12 or later to support detecting SVE predicate events. Non-SVE events only require 4.16 or later. To confirm that basic support is available, check for the path `/sys/bus/event_source/devices/arm_spe_XX`.
- For kernel versions 4.20 to 5.1-RC5, apply the patch at https://lkml.org/lkml/2019/4/16/432.
- Disable kernel page table isolation for the target. To ensure that kernel page table isolation is disabled, boot the machine with the command-line argument `kpti=off`.

3.27.2 Enable Arm SPE from the command line

Enable Arm SPE profiling by using the `--spe` argument:

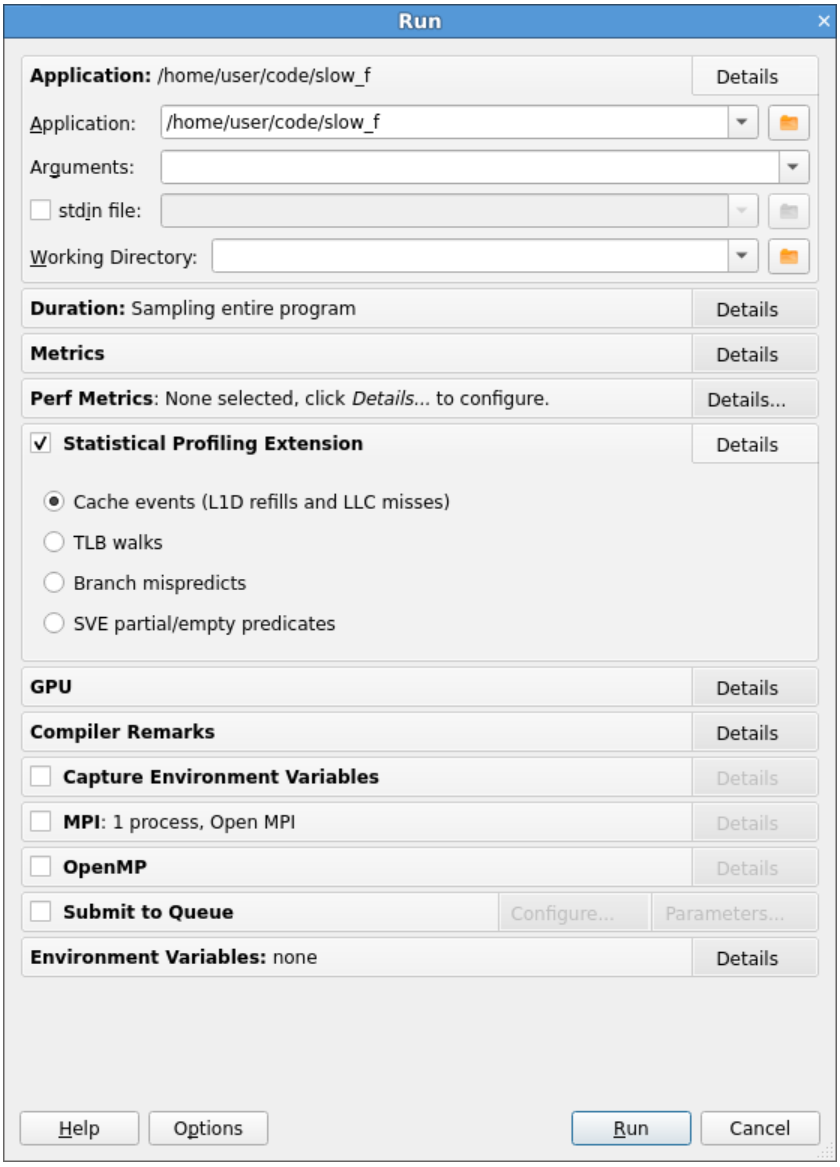
```
/path/to/forge/bin/map --profile --spe=<event-filter> ...
```

Replace `<event-filter>` with one of these values:

- `cache`: collect L1D refills and LLC miss events
- `mispredict`: collect branch mispredict events
- `tlb`: collect TLB walk events
- `sve-predicate`: collect SVE partial predicate and empty predicate events

3.27.3 Enable Arm SPE from the Run dialog

Enable Arm SPE profiling and select the appropriate event filter in the *Statistical Profiling Extension* section of the *Run* dialog. This section is only shown for aarch64 targets.

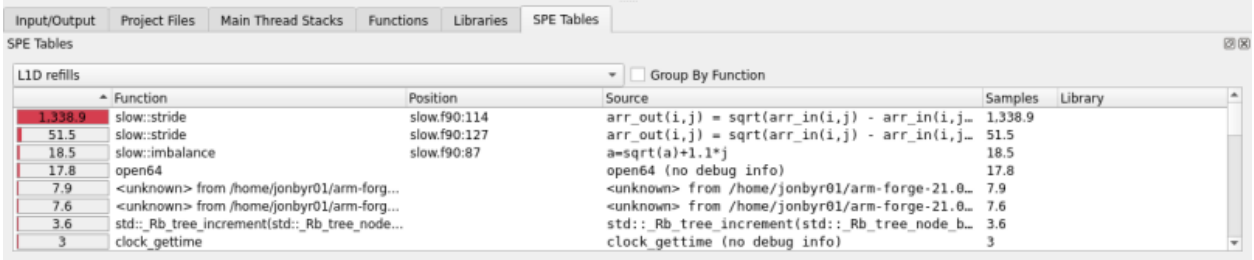


A warning displays if Arm SPE is not supported by the current host. This can be overridden and ignored if you know that Arm SPE is available on the compute nodes.

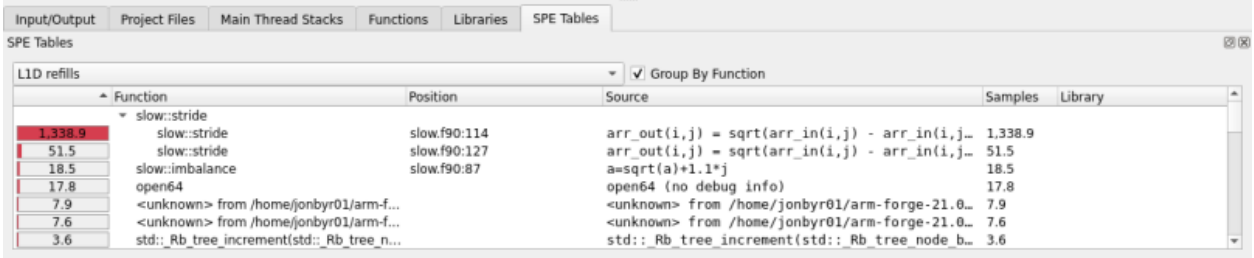
3.27.4 Analyze your program

Use Linaro MAP with the Arm Statistical Profiling Extension (SPE) to find locations in your code where particular hardware events are most commonly triggered.

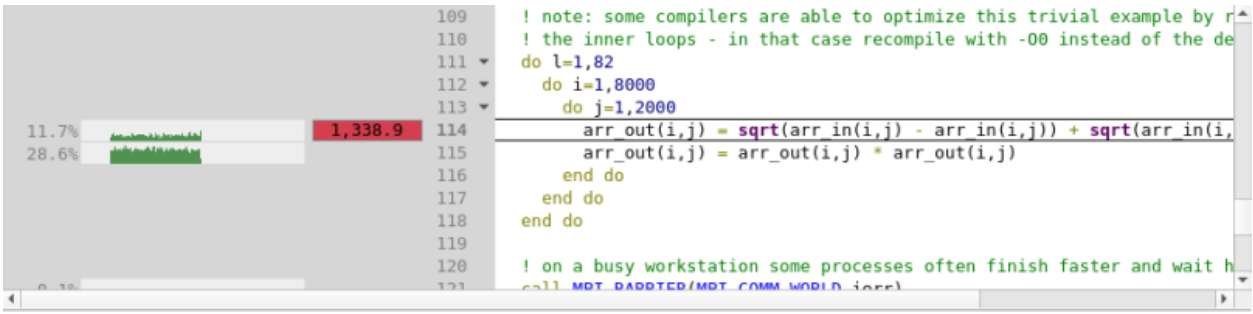
1. Profile your program using Linaro MAP with Arm SPE profiling enabled. Use the filter you are interested in (cache events, tlb walk events, or branch mispredict events).
2. Switch to the *SPE Tables* tab at the bottom of the screen to view the average number of hardware events sampled per process by the SPE feature, by source code line.



3. Optionally, toggle between a plain list of lines containing events, and group the rows by function.



4. Click a row to jump to that location in the source code editor. The average number of SPE samples per process on each line is also displayed here.



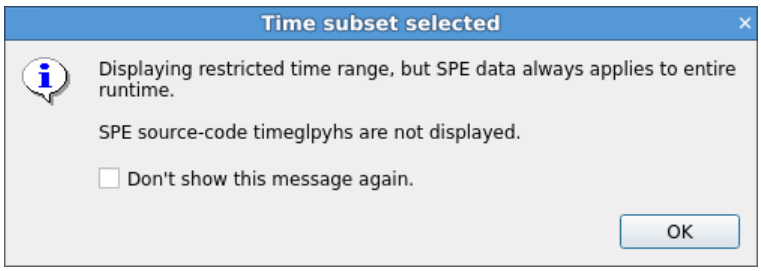
5. If the cache event filter is used, two tables are generated. One table is for all level 1 data cache refill events, and the other is for those for that caused a cache miss in the last level of cache.

   Switch between these tables using the menu at the top of the *SPE Tables* tab. This also updates the count of SPE samples that the code editor displays.

Alternatively, identify bottlenecks using the regular Linaro MAP stacks tabs, and check the surrounding lines in the code editor for SPE samples to draw conclusions about the cause of the bottleneck.

You can profile your application several times using the different SPE filters to collect information on the various types of event.

Note: For performance reasons, Linaro MAP aggregates SPE samples over the lifetime of the program being profiled, and merges samples taken from all threads. Because it is not possible to correctly display which SPE samples apply to a selected time range, the SPE samples display in the Source code editor is hidden when a time range is selected. When this occurs, a dialog box displays that you can dismiss.



The *SPE Tables* tab also displays a warning in this case. When you view the **Main Thread Only** view mode, another warning displays because some of the SPE samples that are listed might have come from one of the non-main threads not normally included in this view.

3.27.5 Guidelines

Recommendations for correctly using and interpreting the results of Arm SPE profiling:

- The bars showing the number of SPE samples for source code lines are for visually comparing the relative number of SPE samples between different lines of code within this profile. A full bar indicates that more events were sampled on a specific source code line than on the others.
- Linaro MAP handles Arm SPE data in a time-agnostic manner: the numbers reported in the *SPE Tables* tab are from the entire program run, and from across all threads. Selecting a time range or using a **Main Thread Only** view mode does not change what is reported by the *SPE Tables* tab or the Arm SPE source code annotations.
- To keep the impact of enabling Arm SPE profiling to reasonable levels, Linaro MAP only utilizes a subset of the samples taken by Arm SPE. Therefore, the number of SPE samples (hits) that are taken, depend on a number of factors that can vary between profiled applications, host machine configuration, and versions of Linaro MAP.
- You can use Arm SPE profiling in conjunction with the *Configurable Perf metrics* feature. Linaro recommends that you enable the CPU instruction metrics which relate to the SPE filter you are using. For example, using the `branch-misses` CPU instruction metric with the `mispredict` Arm SPE filter. The CPU instruction metrics are both time-based and accurate counts, mitigating some of the limitations mentioned here.

3.27.6 Memlock limit exceeded

When using Arm SPE profiling on a system with many logical cores, you can encounter an error regarding an inability to mlock any further memory. This can occur during Linaro MAP profiling initialization or when an MPI or user program attempts to mlock memory.

In order to receive Arm SPE data, Linaro MAP must `mmap` a number of buffers per logical CPUs on the host system. This can sum to a significant amount of memory when the number of logical CPUs is high. In combination with the mlock needs of the target application the system mlock resource limit may be exceeded, preventing more memory from being locked.

3.27.6.1 Solution

Raise the system soft cap to some higher value using `ulimit -l`.

3.27.6.2 Solution

Modify the size of the largest buffers Linaro MAP maps per core using the environment variable `FORGE_SAMPLER_SPE_AUX_BUFFER_SIZE`. This is the number of pages to `mmap` for each logical core on the system. It must be a positive power of 2.

Setting this environment variable too low reduces the amount of Arm SPE data that can be processed by MAP, reducing the number of SPE samples samples obtained.

3.27.7 Arm SPE disabled on Amazon Web Services

Arm SPE is not available on AWS hosts. Arm SPE has been disabled for security reasons on potentially shared hosts.

3.27.7.1 Solution

Currently, you require a `.metal` Arm-64 instance for Arm SPE profiling.

3.27.7.2 Solution

If enabled, Kernel Page Table Isolation (KPTI) blocks the use of Arm SPE (see *Arm SPE Prerequisites*). The stock Amazon Linux AMI boots with KPTI disabled but the standard Red Hat, Suse, and Ubuntu AMIs do not. Either use the Amazon Linux AMI or ensure that the AMI you are using has KPTI disabled.

3.27.8 Known issues for Arm SPE

These known issues affect Arm SPE functionality:

- Neoverse™ N1 hardware has known a sampling bias in its Arm SPE implementation (see errata 1694299, fixed in r4p1). Arm SPE might see unexpectedly high sample counts for branch target instructions and unexpectedly low sample counts for some instructions closely following a branch target.

CHAPTER

FOUR

PERFORMANCE REPORTS

4.1 Get started with Performance Reports

Learn how to get started using Linaro Performance Reports.

4.1.1 Compilers for example programs

Linaro provides a number of example programs to help you get started using Linaro Forge. One of the example programs is a simple 1-D wave equation solver that is useful as a profiling example program. Both C and Fortran variants are provided:

- examples/wave.c
- examples/wave.f90

Both of these variants can be built using the same makefile, wave.makefile. To navigate and run wave.makefile, use:

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

There is also a mixed-mode MPI+OpenMP variant in examples/wave_openmp.c, which is built with the openmp.makefile makefile.

Note: The makefiles for all supplied examples are located in the {installation-directory}/examples directory.

Depending on the default compiler on your system you might see some errors when running the makefile, for example:

```
pgf90-Error-Unknown switch: -fno-inline
```

By default, this example makefile is set up for the GNU compilers. To setup the makefile for a different compiler, open the examples/wave.makefile file, uncomment the appropriate compilation command for the compiler you want to use, and comment those of the GNU compiler.

Notes:

- The compilation commands for other popular compilers are already present in the makefile, separated by compiler.
- Although the example makefiles include the -g flag, Linaro Forge does *not* require this. Do not use them in your own makefiles.

In most cases Linaro Forge can run on an unmodified binary with no recompilation or linking required.

4.1.2 Compile on Cray X-series systems

On Cray X-series systems, the example program must either be dynamically linked (using `-dynamic`) or explicitly linked with the Linaro Forge sampler and MPI wrapper libraries.

4.1.2.1 About this task

To dynamically link the example program with the Linaro Forge sampler and MPI wrapper libraries use:

```
cc -dynamic -g -O3 wave.c -o wave -lm -lrt

ftn -dynamic -G2 -O3 wave.f90 -o wave -lm -lrt
```

This procedure shows you how to explicitly link the example program with the Linaro Forge sampler and MPI wrapper libraries.

4.1.2.2 Procedure

1. Create the libraries using the command `make-profiler-libraries --platform=cray --lib-type=static`:

   ```
   Created the libraries in /home/user/examples:
      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/examplesm/allinea-profiler.ld ... EXISTING_MPI_LIBRARIES
         If your link line specifies EXISTING_MPI_LIBRARIES (e.g. -lmpi), then
         these must appear **after** the Forge sampler and MPI wrapper libraries in
         the link line.  There is a comprehensive description of the link ordering
         requirements in the 'Prepare a Program for Profiling' section of
         userguide-forge.pdf, located in /opt/linaro/forge/x.y.z/doc/.
   ```

2. **Follow the instructions in the output to link the example program**
   with the Linaro Forge sampler and MPI wrapper libraries:

   ```
   cc -g -O3 wave.c -o wave -g -Wl,@allinea-profiler.ld -lm -lrt

   ftn -G2 -O3 wave.f90 -o wave -G2 -Wl,@allinea-profiler.ld -lm -lrt
   ```

4.1.3 Run an example program

Describes how to run an example program with MPI.

4.1.3.1 Before you begin

Make sure that you have compiled the example program. See *Compilers for example programs*.

4.1.3.2 About this task

This example uses MPI, so you must run it on a compute node on your cluster. The help pages and support staff on your site can tell you exactly how to do this on your machine. The simplest way when running small programs is often to request an interactive session.

4.1.3.3 Procedure

Type the command:

```
$ qsub -I
qsub: waiting for job 31337 to start
qsub: job 31337 ready
$ cd /opt/linaro/forge/x.y.z/examples
$ mpiexec -n 4 ./wave_c
Wave solution running with 4 processes
```

4.1.3.4 Results

If the output is similar to this, the example program is compiled and working correctly.

```
0: points = 1000000, running for 30 seconds
points / second: 63.9M (16.0M per process)
compute / communicate efficiency: 94% | 97% | 100%

Points for validation:
0:0.00  200000:0.95  400000:0.59  600000:-0.59  800000:-0.95  999999:0.00
wave finished
```

4.1.3.5 Next steps

Generate a performance report for an example program

4.1.4 Simultaneously generate both a `.map` file and Performance Report

Describes how to load Linaro MAP and then profile and application, generating both a `.map` file and one or more type of Performance Report at once.

4.1.4.1 Before you begin

Make sure that the Linaro MAP component of Linaro Forge installed on your system is loaded:

```
$ map --version
Linaro MAP
Part of Linaro Forge.
Copyright (c) 2023-2025 Linaro Limited. All rights reserved.
...
```

Note: Please add the Linaro Forge installation path to the `PATH` environment variable to make the Linaro MAP available to the console:

```
export PATH=<Forge installation path>/bin:$PATH
```

4.1.4.2 Procedure

1. Type the `map --profile --report=<types>` command in front of your existing `mpiexec` command-line, where `<types>` is a comma separated list of one or more of `txt`, `csv`, `html`, or `summary` (see `--report`) e.g.:

   ```
   map --profile --report=txt,html mpiexec -n 4 examples/wave_c
   ```

2. If your program is submitted through a batch queuing system, modify your submission script to load the Linaro Forge module and add the `map --profile --report=<types>` command in front of the `mpiexec` command for which you want to generate a report.

4.1.4.3 Results

The program runs as usual, although startup and shutdown might take a few minutes longer while Linaro Forge generates and links the appropriate wrapper libraries before running, collecting the data at the end of the run. The runtime of your code (between `MPI_Init` and `MPI_Finalize` is not expected to be affected by more than a few percent at most.

After the run finishes, a `.map` file and whatever Performance Reports were requested are saved to the current working directory, using a name based on the application executable:

```
$ map --profile --report=txt,html,summary mpirun -n 4 ./wave_c

   [ ... normal MAP and wave_c output ... ]

MAP analysing program...
MAP gathering samples...
MAP generated wave_c_4p_1n_2022-11-16_18-53.map
MAP generated wave_c_4p_1n_2022-11-16_18-53.html
```

(continues on next page)

(continued from previous page)

```
MAP generated wave_c_4p_1n_2022-11-16_18-53.txt

Linaro MAP profiling summary
============================
Profiling time:      31 seconds (between MPI init and finalize)
Peak process memory: 52002816 B (~49.6 MiB)

Compute:              92.3%    (28.6s) |========|
MPI:                   7.7%     (2.4s) ||
I/O:                   0.0%     (0.0s) |
  (based on time on the main thread)
```

4.1.5 Generate a performance report for an example program

Describes how to load Performance Reports and then generate a report.

4.1.5.1 Before you begin

Make sure that the Linaro Performance Reports component of Linaro Forge installed on your system is loaded:

```
$ perf-report --version
Linaro Performance Reports
Copyright (c) 2023-2025 Linaro Limited. All rights reserved.
...
```

Note: Please add the Linaro Forge installation path to the `PATH` environment variable to make the Linaro Performance Reports available to the console:

```
export PATH=<Forge installation path>/bin:$PATH
```

4.1.5.2 Procedure

1. Type the `perf-report` command in front of your existing `mpiexec` command-line:

   ```
   perf-report mpiexec -n 4 examples/wave_c
   ```

2. If your program is submitted through a batch queuing system, modify your submission script to load the Linaro Forge module and add the `perf-report` line in front of the `mpiexec` command for which you want to generate a report.

4.1.5.3 Results

The program runs as usual, although startup and shutdown might take a few minutes longer while Linaro Forge generates and links the appropriate wrapper libraries before running, and collects the data at the end of the run. The runtime of your code (between `MPI_Init` and `MPI_Finalize` is not expected to be affected by more than a few percent at most.

After the run finishes, a performance report is saved to the current working directory, using a name based on the application executable:

```
$ ls -lrt wave_c*
-rwx------ 1 mark mark 403037 Nov 14 03:21 wave_c
-rw------- 1 mark mark   1911 Nov 14 03:28 wave_c_4p_2013-11-14_03-27.txt
-rw------- 1 mark mark 174308 Nov 14 03:28 wave_c_4p_2013-11-14_03-27.html
```

Note: Both `.txt` and `.html` versions are automatically generated.

4.1.6 Generate a performance report from an existing .map file

Describes how to generate a Performance Report from an existing .map file

4.1.6.1 Before you begin

Start with a `.map` file previously generated by Linaro MAP.

4.1.6.2 Procedure

From the command line, provide the `.map` file to the `perf-report` command in place of an `mpiexec` command-line:

```
perf-report examples/slow.map
```

4.1.6.3 Results

Performance reports generates report files as if `perf-report` had been used in place of `map` when profiling the application:

```
$ ls -lrt slow_f*
-rw------- 1 mark mark   3321 Nov 14 03:28 slow_f_16p_2n_2022-09-16_11-33.txt
-rw------- 1 mark mark 463962 Nov 14 03:28 slow_f_16p_2n_2022-09-16_11-33.html
```

Note: Both `.txt` and `.html` versions are automatically generated.

4.1.7 View or export a performance report whilst viewing a .map file

Describes how to generate or view a Performance Report whilst looking at an existing `.map` file from within Linaro MAP.

4.1.7.1 Before you begin

View a profile in Linaro MAP. It does not matter whether profiling was triggered through the Linaro MAP graphical user interface or if you open a pregenerated `.map` file.

4.1.7.2 Procedure

1. Open the *Reports* menu on the menu bar
2. Click *View Performance Report in browesr* to view a HTML Performance Report for the current `.map` file in your OS's default web browser.
3. Alternatively click *Export Performance Report…* and specify the filename to write to in the *Save* dialog that is shown.

4.1.7.3 Results

Temporary report files to be displayed in the browser will be generated in your system's default temporary directory.

When explicitly saving a Performance Reports file the file type will be automatically determined by the specified file extension.

4.2 Run real programs

This section shows you how to compile and run your own programs.

Linaro Performance Reports is designed to run on unmodified production executables, so in general no preparation step is necessary. However, there is one important exception: statically linked applications require additional libraries at the linking step.

4.2.1 Link with a program

To collect data from your program, Performance Reports uses two small profiler libraries, `map-sampler` and `map-sampler-pmpi`. These profiler libraries must be linked with your program. On most systems Performance Reports 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.

This automatic linking when you start your program only works if your program is dynamically-linked. Programs may be dynamically-linked or statically-linked. 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 Linaro Performance Reports 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 *No shared home directory*.

The temporary library will be created in the allinea/wrapper subdirectory to this `shared directory`.

Note: Although the profiler libraries contain the word 'map' they are used for both Linaro Performance Reports and Linaro MAP.

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 Linaro Performance Reports warns you that it could not pre-load the Linaro Forge 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:

- Try compiling and linking your code dynamically. On most platforms this allows Linaro Performance Reports to use the LD_PRELOAD mechanism to automatically insert its libraries into your application at runtime.
- Link MAP's `map-sampler` and `map-sampler-pmpi` libraries with your program at link time manually.

  See *Dynamic linking on Cray X-Series systems (Performance Reports)*, or *Static linking (Performance Reports)* and *Static linking on Cray X-Series systems (Performance Reports)*.
- 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.

4.2.2 Dynamic linking on Cray X-Series systems (Performance Reports)

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 Linaro Performance Reports sampling libraries.

4.2.2.1 Procedure

1. Compile the MPI wrapper library for your system using the `make-profiler-libraries --platform=cray --lib-type=shared` command.

```
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.
```

(continues on next page)

(continued from previous page)

```
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.
```

2. Link with the MPI wrapper library.

```
mpicc -G2 -o hello hello.c -dynamic -L/home/user/myprogram \
   -lmap-sampler-pmpi -lmap-sampler -Wl,--eh-frame-hdr
```

NVIDIA HPC Compiler

When linking OpenMP programs, you must pass the `-Bdynamic` command line argument to the compiler when linking dynamically.

4.2.3 Static linking (Performance Reports)

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 Linaro Forge sampler and MPI wrapper libraries.

4.2.3.1 Procedure

1. Compile the MPI wrapper library for your system using the `make-profiler-libraries --lib-type=static` command.

```
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 Forge sampler and MPI wrapper libraries in
the link line.  There's a comprehensive description of the link ordering
requirements in the 'Prepare a Program for Profiling' section of
userguide-forge.pdf, located in /opt/linaro/forge/x.y.z/doc/.
```

2. Link with the 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 Linaro Forge sampler. Usually this is sufficient, but not in all cases. The rest of this section explains how to manually add the Linaro Forge sampler to your link line.

NVIDIA HPC Compiler

The NVIDIA HPC C runtime static library contains an undefined reference to `__kmpc_fork_call`, which will cause compilation to fail when linking allinea-profiler.ld. To resolve this, add `--undefined __wrap___kmpc_fork_call` to your link line before linking to the Linaro Forge sampler.

Cray

When linking C++ programs you may encounter a conflict between the Cray C++ runtime and the GNU C++ runtime used by the Linaro Performance Reports 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:

- *After* your program's object (`.o`) files.
- *After* your program's own static libraries, for example `-lmylibrary`.
- *After* the path to the profiler libraries (-_L/path/to/profiler-_libraries).
- *Before* the MPI's Fortran wrapper library, if any. For example `-lmpichf`.
- *Before* the MPI's implementation library usually `-lmpi`.
- *Before* the Linaro Forge sampler library `-lmap-sampler`.

The Linaro Forge sampler library `-lmap-sampler` must go:

- *After* the MPI wrapper library.
- *After* your program's object (`.o`) files.
- *After* your program's own static libraries, for example `-lmylibrary`.
- *After* `-Wl,--undefined,allinea_init_sampler_now`.
- *After* the path to the profiler libraries -_L/path/to/profiler-_libraries.
- *Before* `-lstdc++`, `-lgcc_eh`, `-lrt`, `-lpthread`, `-ldl`, `-lm` and `-lc`.

For example:

```
mpicc hello.c -o hello -g -L/users/ddt/linaro \
   -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/linaro \
   -lmap-sampler-pmpi \
```

(continues on next page)

(continued from previous page)

```
-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
```

4.2.4 Static linking on Cray X-Series systems (Performance Reports)

If you compile your program statically on a Cray X-Series system, you must explicitly link your program with the Linaro Forge sampler and MPI wrapper libraries.

4.2.5 Procedure

1. On Cray X-Series systems, you can compile the MPI wrapper library using `make-profiler-libraries --platform=cray --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 (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 Forge sampler and MPI wrapper libraries in
the link line.  There is a comprehensive description of the link ordering
requirements in the 'Prepare a program for profiling' section of the
userguide-forge.pdf, located in /opt/linaro/forge/x.y.z/doc/.
```

2. Link with the MPI wrapper library using:

```
cc hello.c -o hello -g -Wl,@allinea-profiler.ld

ftn hello.f90 -o hello -g -Wl,@allinea-profiler.ld
```

4.2.6 Dynamic and static linking on Cray X-Series systems using the modules environment (Performance Reports)

If your system has the Linaro Forge module files installed, you can load them and build your application as usual.

4.2.6.1 About this task

For more information about installing map-link modules, see *map-link modules installation on Cray X-Series systems (Performance Reports)*.

4.2.6.2 Procedure

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.

4.2.7 map-link modules installation on Cray X-Series systems (Performance Reports)

To facilitate dynamic and static linking of your programs with the MPI wrapper and Linaro Forge sampler libraries, Cray X-Series System Administrators can integrate the map-link-dynamic and map-link-static modules into their module system.

4.2.7.1 About this task

Templates for these modules are supplied as part of the Linaro Forge package.

4.2.7.2 Procedure

1. Copy files `share/modules/cray/map-link-*` into a dedicated directory on the system.
2. For each of the two module files copied, find the line starting with **conflict** and correct the prefix to refer to the location the module files were installed, for example, `forge/map_link_static`. The correct prefix depends on the subdirectory (if any) under the module search path the `map-link-*` module files were installed.
3. For each of the two module files copied, 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.

4.2.7.3 Results

After installation you can verify if the prefix has been set correctly using `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.

4.2.8 Unsupported user applications (Performance Reports)

Ensure that the program to be profiled does not set or unset the SIGPROF signal handler. This interferes with the Linaro Performance Reports profiling function and can cause it to fail.

We recommend that you do not use Linaro Performance Reports to profile programs that contain instructions to perform MPI profiling using MPI wrappers and the MPI standard profiling interface, PMPI. This is because Performance Report's own MPI wrappers may conflict with those contained in the program, producing incorrect metrics.

4.2.9 Express Launch mode (Performance Reports)

Linaro Forge can be launched by typing its command name in front of an existing `mpiexec` command:

```
$ perf-report mpiexec -n 256 examples/wave_c 30
```

4.2.9.1 Compatible MPIs

The MPI implementations supported by Express Launch are:

- Bullx MPI
- Cray X-Series (MPI/SHMEM/CAF)
- Intel MPI
- MPICH 3
- MPICH 4
- Open MPI (MPI/SHMEM)
- Open MPI (Cray XT/XE/XK)
- Cray XT/XE/XK (UPC)
- SLURM (MPMD)

If your MPI is not supported by Express Launch, an error message will display:

```
$ 'Generic' MPI programs cannot be started using Express Launch syntax (launching with an mpirun␣
 ↪command).

Try this instead:
    perf-report --processes=256 ./wave_c 20

Type perf-report --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 Linaro Forge products.

Normal redirection syntax may be used to redirect standard input and standard output.

4.2.10 Compatibility Launch mode

Compatibility Launch mode must be used if Linaro Forge does not support Express Launch mode for your MPI, or, for some MPIs, if it is not able to access the compute nodes directly (for example, using `ssh`).

To use Compatibility Launch mode, replace the `mpiexec` command with the `perf-report` command.

For example:

```
mpiexec --np=256 ./wave_c 20
```

becomes

```
perf-report --np=256 ./wave_c 20
```

Only a small number of `mpiexec` arguments are supported by perf-report (for example, `-n` and `-np`). Other arguments must be passed using the `--mpiargs="args here"` parameter.

For example:

```
mpiexec --np=256 --nooversubscribe ./wave_c 20
```

becomes

```
perf-report --mpiargs="--nooversubscribe" --np=256 ./wave_c 20
```

Normal redirection syntax may be used to redirect standard input and standard output.

4.2.11 Generate a performance report for a real program

Describes how to load Performance Reports and then generate a report.

4.2.11.1 Before you begin

Make sure that the Performance Reports component of Linaro Forge installed on your system is loaded:

```
$ perf-report --version
Linaro Performance Reports
Copyright (c) 2023-2025 Linaro Limited. All rights reserved.
...
```

If this command cannot be found, consult the site documentation to find the name of the correct module.

4.2.11.2 Procedure

1. Type the `perf-report` command in front of your existing `mpiexec` command-line:

   ```
   perf-report mpiexec -n 4 examples/wave_c
   ```

2. If your program is submitted through a batch queuing system, modify your submission script to load the Linaro Forge module and add the 'perf-report' line in front of the `mpiexec` command for which you want to generate a report.

4.2.11.3 Results

The program runs as usual, although startup and shutdown may take a few minutes longer while Linaro Forge generates and links the appropriate wrapper libraries before running and collects the data at the end of the run. The runtime of your code (between `MPI_Init` and `MPI_Finalize` should not be affected by more than a few percent at most.

After the run finishes, a performance report is saved to the current working directory, using a name based on the application executable:

```
$ ls -lrt wave_c*
-rwx------ 1 mark mark 403037 Nov 14 03:21 wave_c
-rw------- 1 mark mark   1911 Nov 14 03:28 wave_c_4p_2013-11-14_03-27.txt
-rw------- 1 mark mark 174308 Nov 14 03:28 wave_c_4p_2013-11-14_03-27.html
```

Note: Both `.txt` and `.html` versions are automatically generated.

You can include a short description of the run or other notes on configuration and compilation settings by setting the environment variable `FORGE_NOTES` before running `perf-report`:

```
$ FORGE_NOTES="Run with inp421.dat and mc=1" perf-report mpiexec -n 512 ./parEval.bin --use-mc=1 inp421.
↪dat
```

The string in the `FORGE_NOTES` environment variable is included in all report files produced.

4.2.12 Specify output locations

By default, performance reports are placed in the current working directory using an auto-generated name based on the application executable name, for example:

```
wave_f_16p_2013-11-18_23-30.html
wave_f_2p_8t_2013-11-18_23-30.html
```

This is formed by the name, the size of the job, the date, and the time. If you are using OpenMP, the value of `OMP_NUM_THREADS` is also included in the name after the size of the job. The name will be made unique if necessary by adding a _1/_2 suffix.

You can specify a different location for output files using the `--output` argument.

For example:

- `--output=my-report.txt` will create a plain text report in the file `my-report.txt` in the current directory.
- `--output=/home/mark/public/my-report.html` will create an HTML report in the file `/home/mark/public/my-report.html`.
- `--output=my-report` will create a plain text report in the file `my-report.txt` and an HTML report in the file `my-report.html`, both in the current directory.
- `--output=/tmp` will create reports with names based on the application executable name in /tmp/, for example, /tmp/wave_f_16p_2013-11-18_2330.txt and /tmp/wave_f_16p_2013-11-18_2330.html.

4.2.13 Enable and disable metrics

You can specify comma-separated lists which explicitly enable or disable metrics for which data is to be collected.

```
--enable-metrics=METRICS
--disable-metrics=METRICS
```

If the metrics specified cannot be found, an error message is displayed and Performance Reports exits. Metrics which are always enabled or disabled cannot be explicitly disabled or enabled. 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.

4.3 Summarize an existing MAP file

Linaro Performance Reports can be used to summarize an application profile generated by Linaro MAP.

4.3.1 Summarize an existing MAP file

To produce a performance report from an existing Linaro MAP output file called `profile.map`, simply run:

```
$ perf-report profile.map
```

Command-line options which would alter the execution of a program being profiled, such as specifying the number of MPI ranks, have no effect. Options affecting how Linaro Performance Reports produces its report, such as `--output`, work as expected.

For the best results, ensure that Linaro Performance Reports and Linaro MAP versions match, for example, Linaro Performance Reports 20.2.1 with Linaro MAP 20.2.1. Linaro Performance Reports can use `.map` files from versions of Linaro MAP as old as 5.0.

4.4 Interpret performance reports

This section explains how to interpret the reports produced by Linaro Performance Reports.

Reports are generated in both HTML and textual formats for each run of your application, by default. The information presented in both of these formats is the same.

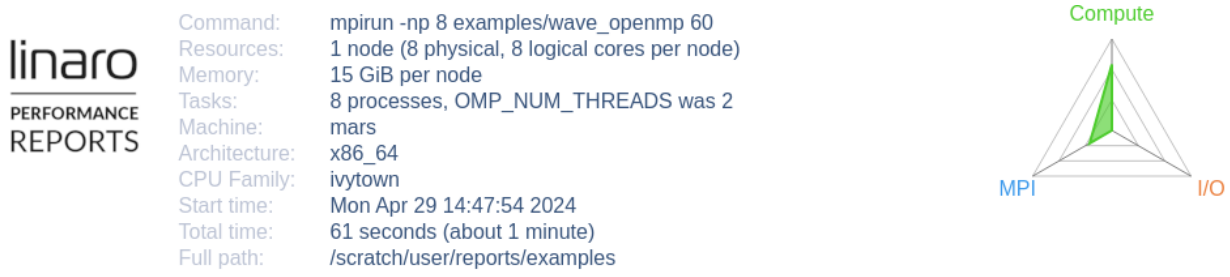
If you want to combine Linaro Forge with other tools, consider using the CSV output format. See *CSV performance reports* for more details.

4.4.1 HTML performance reports

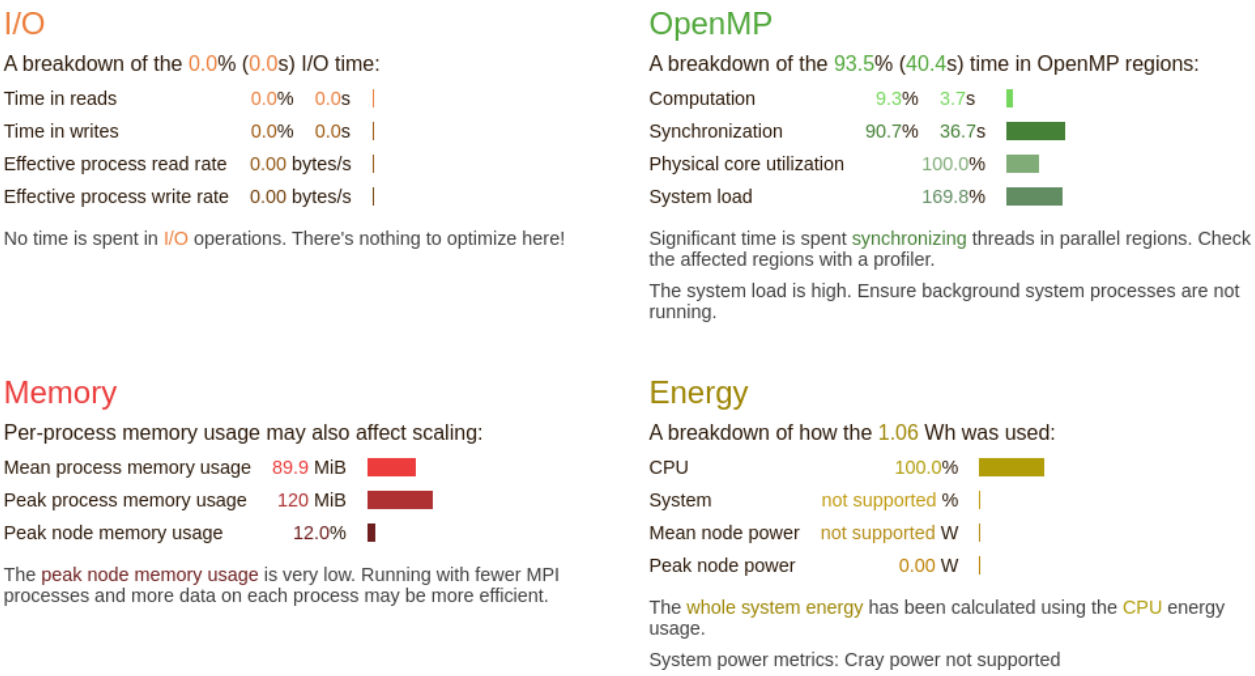
Viewing HTML files is best done on your local machine. Many sites have places you can put HTML files to be viewed from within the intranet. These directories are a good place to automatically send your performance reports. Alternatively, you can use `scp` or `sshfs` to make the reports available on your computer:

```
$ scp login1:/opt/linaro/forge/x.y.z/examples/wave_c_4p*.html .
$ firefox wave_c_4p*.html
```

The following report was generated by running the `wave_openmp.c` example program with 8 MPI processes and 2 OpenMP threads per process on a typical HPC cluster:



When you run a report on this example program, the results might be different to the report shown here depending on the performance and network architecture of the machine on which you run it, but the basic structure of these reports is always the same. This common structure makes comparisons between reports simple, direct, and intuitive.



The following sections describe each section of the report.

4.4.2 Report summary

The Summary shows how the wall clock time of the application was spent, based on the activity of the main thread. It is organized by Compute, MPI, I/O, and (in the case of Python scripts) Python Interpreter.

In the example file, you can see that Linaro Forge has identified that the program is compute-bound, which means that most of its time is spent inside application code rather than communicating or using the filesystem.

The pieces of advice that the program offers, such as `this code may benefit from running at larger scales`, are good starting points for future investigations. They are designed to be informative to scientific users with no previous MPI tuning experience.

The radar chart in the top-right corner of the report reflects the values of these key measurements: compute, MPI, I/O, and (if applicable) Python interpreter. It is helpful to recognize and compare these triangular or rectangular shapes when switching between multiple reports.

4.4.2.1 Compute

Time spent computing. This is the percentage of wall clock time spent in application and in library code, excluding time spent in MPI calls and I/O calls.

4.4.2.2 MPI

Time spent communicating. This is the percentage of wall clock time spent in MPI calls such as `MPI_Send`, `MPI_Reduce` and `MPI_Barrier`.

4.4.2.3 I/O (Input/Output)

Time spent reading from and writing to the filesystem. This is the percentage of wall clock time spent in system library calls such as `read`, `write` and `close`.

Note: All time spent in MPI-IO calls is included here, even though some communication between processes might also be performed by the MPI library. `MPI_File_close` is treated as time spent writing, which is often, but not always, correct.

4.4.2.4 Python Interpreter

Time spent in the Python Interpreter. This is the percentage of wall clock time spent in Python code that does not call out to some precompiled library.

4.4.3 CPU breakdown

This section organizes the time spent in application and library code further by analyzing the kinds of instructions that this time was spent on.

Note: All of the metrics described in this section are only available on x86_64 systems.

Note: All percentages here are relative to the compute time, not to the entire application run. Time spent in MPI and I/O calls is not represented inside this section.

4.4.3.1 Single-core code

The percentage of wall clock in which the application executed using only one core per process, rather than multithreaded or OpenMP code. If you have a multithreaded or OpenMP application, a high value here indicates that your application is bound by Amdahl's law and that scaling to larger numbers of threads will not meaningfully improve performance.

4.4.3.2 OpenMP regions

The percentage of wall clock time spent in OpenMP regions. The higher this is, the better. This metric is only shown if the program spent a measurable amount of time inside at least one OpenMP region.

4.4.3.3 Scalar numeric ops

The percentage of time spent executing arithmetic operations such as `add`, `mul`, `div`. This does not include time spent using the more efficient vectorized versions of these operations.

4.4.3.4 Vector numeric ops

The percentage of time spent executing vectorized arithmetic operations such as Intel's SSE2 / AVX extensions.

Generally it is good if a scientific code spends most of its time in these operations because that is the only way to achieve anything close to the peak performance of modern processors.

If this value is low, you can check the vectorization report of the compiler to understand why the most time consuming loops are not using these operations. Compilers need a lot of help to efficiently vectorize non-trivial loops and the investment in time is often rewarded with 2x-4x performance improvements.

4.4.3.5 Memory accesses

The percentage of time spent in memory access operations, such as `mov`, `load`, `store`. A portion of the time spent in instructions that use indirect addressing is also included here. A high figure here shows the application is memory-bound and is not able to take full advantage of the CPU resources. Often it is possible to reduce this figure by analyzing loops for poor cache performance and problematic memory access patterns, improving performance significantly.

A high percentage of time spent in memory accesses in an OpenMP program is often a scalability problem. If each core spends most of its time waiting for memory, or the L3 cache, then adding further cores rarely improves matters. Equally, false sharing, in which cores block attempts to access the same cache lines, and the over-use of the `atomic` pragma, show up as increased time spent in memory accesses.

4.4.3.6 Waiting for accelerators

The percentage of time that the CPU is waiting for the accelerator.

4.4.4 CPU metrics breakdown

This section presents key CPU performance measurements gathered using the Linux perf event subsystem.

Note: Metrics described in this section are only available on Arm®v8. 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 *Armv8 (AArch64) known issues* in *Platform notes and known issues*.

4.4.4.1 Cycles per instruction

The number of CPU cycles to execute an instruction. Lower is better. This metric can be less than one when the program takes advantage of instruction-level parallelism. Please consult L2 Data cache misses, Stalled backend cycles and Stalled frontend cycles for further insight on how the cycles per instruction might be reduced.

4.4.4.2 L2 Data cache misses

The ratio of L2 data caches misses to kilo instructions retired. Lower is better. Often it is possible to reduce this figure by analyzing loops for poor cache performance and problematic memory access patterns, improving performance significantly.

4.4.4.3 Stalled backend cycles

The percentage of cycles where no operation was issued because of the backend, due to a lack of required resources. Lower is better. This metric captures inefficiencies in backend units like execution units, data cache misses and translation delays caused by data TLB walks [1]. See the L2D cache miss ratio for further insight. Additionally, pressure on the arithmetic units will appear as stalled backend cycles [1].

4.4.4.4 Stalled frontend cycles

The percentage of cycles where no operation was issued because of the frontend, due to fetch starvation. Lower is better. This metric captures inefficiencies in the branch prediction unit, fetch latency due to instruction cache misses and translation delays caused by Instruction TLB walks [1].

4.4.4.5 References

[1] Mundichipparakkal,J., 2023, June. Arm Neoverse V1 Core: Performance Analysis Methodology

4.4.5 MPI breakdown

This section organizes the time spent in MPI calls reported in the summary. It is only interesting if the program spends a significant amount of its time in MPI calls.

All the rates quoted here are inbound and outbound rates. This means that the rate of communication is being measured from the process to the MPI API, not of the underlying hardware directly.

This application-perspective is found throughout Linaro Forge, and in this case allows the results to capture effects such as faster intra-node performance, zero-copy transfers, and other effects.

Note: For programs that make MPI calls from multiple threads (MPI is in `MPI_THREAD_SERIALIZED` or `MPI_THREAD_MULTIPLE` mode), Linaro Forge only displays metrics for MPI calls made on the main thread.

4.4.5.1 Time in collective calls

The percentage of time spent in collective MPI operations such as `MPI_Scatter`, `MPI_Reduce`, and `MPI_Barrier`.

4.4.5.2 Time in point-to-point calls

The percentage of time spent in point-to-point MPI operations such as `MPI_Send` and `MPI_Recv`.

4.4.5.3 Effective process collective rate

The average transfer per-process rate during collective operations, from the perspective of the application code and not the transfer layer. For example, an `MPI_Alltoall` that takes 1 second to send 10 Mb to 50 processes and receive 10 Mb from 50 processes has an effective transfer rate of 10x50x2 = 1000 Mb/s.

Collective rates can often be higher than the peak point-to-point rate if the network topology matches the application's communication patterns well.

4.4.5.4 Effective process point-to-point rate

The average per-process transfer rate during point-to-point operations, from the perspective of the application code and not the transfer layer. Asynchronous calls that allow the application to overlap communication and computation such as `MPI_ISend` can achieve much higher effective transfer rates than synchronous calls.

Overlapping communication and computation is often a good strategy to improve application performance and scalability.

4.4.6 I/O breakdown

This section organizes the amount of time spent in library and system calls relating to I/O, such as read, write and close. I/O that is generated by MPI network traffic is not included. In most cases, this should be a direct measure of the amount of time spent reading and writing to the filesystem, whether local or networked.

Some systems, such as the Cray X-series, do not have I/O accounting enabled for all filesystems. On these systems only Lustre I/O is reported in this section.

Even if your application does not perform I/O, a non-zero amount of I/O is reported because of internal I/O performed by Linaro Performance Reports.

4.4.6.1 Time in reads

The percentage of time spent on average in read operations from the perspective of the application, not the filesystem. Time spent in the `stat` system call is also included here.

4.4.6.2 Time in writes

The percentage of time spent on average in write and sync operations from the perspective of the application, not the filesystem.

Opening and closing files is also included here, because measurements have shown that the latest networked filesystems can spend significant amounts of time opening files with create or write permissions.

4.4.6.3 Effective process read rate

The average transfer rate during read operations from the perspective of the application. A cached read has a much higher read rate than one that has to hit a physical disk. This is particularly important to optimize for because current clusters often have complex storage hierarchies with multiple levels of caching.

4.4.6.4 Effective process write rate

The average transfer rate during write and sync operations from the application's perspective. A buffered write will have a much higher write rate than one that has to hit a physical disk. However, unless there is significant time between writing and closing the file, the penalty will be paid during the synchronous close operation instead. All these complexities are captured in this measurement.

4.4.6.5 Lustre metrics

Lustre metrics are enabled if your compute nodes have one or more Lustre filesystems mounted. Lustre metrics are obtained from a Lustre client process that runs on each node. Therefore, the data gives the information gathered on a per-node basis. The data is also cumulative over all of the processes run on a node, not only the application being profiled. Consequently, there might 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 that is 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 that is read or written, almost all of the data reported in Linaro Performance Reports 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, and 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 filesystem 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.

Linaro Performance Reports reports on the total number of metadata operations, and also the total number of file opens that are encountered by a Lustre client. With the information provided in Linaro Performance Reports you can observe the rate at which data is read and written to Lustre through the Lustre client, and also 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 if you have the Advanced Metrics Pack add-on for Linaro Performance Reports.

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' might count as multiple operations, for example, when it creates a new file or truncates an existing one.

4.4.7 OpenMP breakdown

This section breaks down the time spent in OpenMP regions into computation and synchronization and includes additional metrics that help to diagnose OpenMP performance problems. It is only shown if a measurable amount of time was spent inside OpenMP regions.

4.4.7.1 Computation

The percentage of time threads in OpenMP regions that is spent computing rather than waiting or sleeping. Keeping this high is one important way to ensure that OpenMP codes scale well. If this is high, look at the CPU breakdown to see whether that time is being used optimally on floating-point operations for example, or whether the cores are mostly waiting for memory accesses.

4.4.7.2 Synchronization

The percentage of time threads in OpenMP regions spent waiting or sleeping. By default, each OpenMP region ends with an implicit barrier. If the workload is imbalanced and some threads finish sooner and wait, this value will increase. Also, there is some overhead associated with entering and leaving OpenMP regions and a high synchronization time might show that the threading is too fine-grained. In general, OpenMP performance is better when outer loops are parallelized, rather than inner loops.

4.4.7.3 Physical core utilization

Modern CPUs often have multiple *logical* cores for each *physical* core. This is often referred to as hyper-threading. These logical cores can share logic and arithmetic units. Some programs perform better when using additional logical cores, but most HPC codes do not.

If the value here is greater than 100, `OMP_NUM_THREADS` is set to a larger number of threads than physical cores that are available and performance can be impacted, usually appearing as a larger percentage of time in OpenMP synchronization or memory accesses.

4.4.7.4 System load

The number of active (running or runnable) threads as a percentage of the number of physical CPU cores that are present in the compute node. This value can exceed 100% if you are using hyper-threading, the cores are *oversubscribed*, or other system processes and daemons start running and take CPU resources away from your program. A value consistently less than 100% might indicate your program is not taking full advantage of the CPU resources available on a compute node.

4.4.8 Threads breakdown

This section organizes the time spent by worker threads (non-main threads) into computation and synchronization, and includes additional metrics that help to diagnose multicore performance problems. This section is replaced by the OpenMP Breakdown if a measurable amount of application time was spent in OpenMP regions.

4.4.8.1 Computation (Threads)

The percentage of time that worker threads spend computing rather than waiting in locks and synchronization primitives. If this is high, look at the CPU breakdown to see whether that time is used optimally on floating-point operations for example, or whether the cores are mostly waiting for memory accesses.

4.4.8.2 Synchronization (Threads)

The percentage of time worker threads spend waiting in locks and synchronization primitives. This only includes time in which those threads were active on a core and does not include time spent sleeping while other useful work is being done. A large value here indicates a performance and scalability problem that can be detected with a multicore profiler such as Linaro MAP.

4.4.8.3 Physical core utilization (Threads)

Modern CPUs often have multiple *logical* cores for each *physical* core. This is often referred to as hyper-threading. These logical cores can share logic and arithmetic units. Some programs perform better when using additional logical cores, but most HPC codes do not.

The value here shows the percentage utilization of physical cores. A value over 100% indicates that more threads are executing than there are physical cores, indicating that hyper-threading is in use.

Only threads actively and simultaneously consuming CPU time are included in this metric. A program can have many helper threads that do little except sleep, and are not shown.

4.4.8.4 System load (Threads)

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 can exceed 100% if you are using hyper-threading, 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% might indicate your program is not taking full advantage of the CPU resources available on a compute node.

4.4.9 Memory breakdown

Unlike the other sections, the memory section does not refer to one particular portion of the job. Instead, it summarizes memory usage across all processes and nodes over the entire duration. All of these metrics refer to RSS, meaning physical RAM usage, and not virtual memory usage. Most HPC jobs attempt to stay within the physical RAM of their node for performance reasons.

4.4.9.1 Mean process memory usage

The average amount of memory used per-process across the entire length of the job.

4.4.9.2 Peak process memory usage

The peak memory usage that is seen by one process at any moment during the job. If this varies a lot from the mean process memory usage, it might be a sign of either imbalanced workloads between processes or a memory leak within a process.

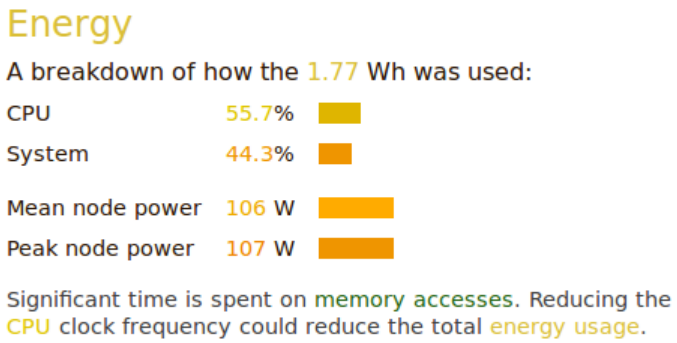
Note: This is not a true high-watermark, but rather the peak memory seen during statistical sampling. For most scientific codes, this is not a meaningful difference because rapid allocation and deallocation of large amounts of memory is generally avoided for performance reasons.

4.4.9.3 Peak node memory usage

The peak percentage of memory that is seen being used on any single node during the entire run. If this is close to 100%, swapping might be occurring, or the job might be likely to hit hard system-imposed limits. If this is low, it might be more efficient in CPU hours to run with a smaller number of nodes and a larger workload per node.

4.4.10 Energy breakdown

This section shows the energy used by the job, organized by component, such as CPU and accelerators.



4.4.10.1 CPU

The percentage of the total energy used by the CPUs.

CPU power measurement requires either an Intel CPU with RAPL or a Cray machine with Cray HSS energy counters version 2 support.

4.4.10.2 Accelerator

The percentage of energy used by the accelerators. This metric is only shown when GPU is present.

4.4.10.3 System (Energy)

The percentage of energy used by other components not shown above. If CPU and accelerator metrics are not available, the system energy will be 100%.

4.4.10.4 Mean node power

The average of the mean power consumption of all the nodes in Watts.

4.4.10.5 Peak node power

The node with the highest peak of power consumption in Watts.

4.4.10.6 Requirements

CPU power measurement requires either an Intel CPU with RAPL support or Cray machines with Cray HSS energy counters support (version 2).

For RAPL support, Sandy Bridge or newer is required, and the `intel_rapl` powercap kernel module must be loaded.

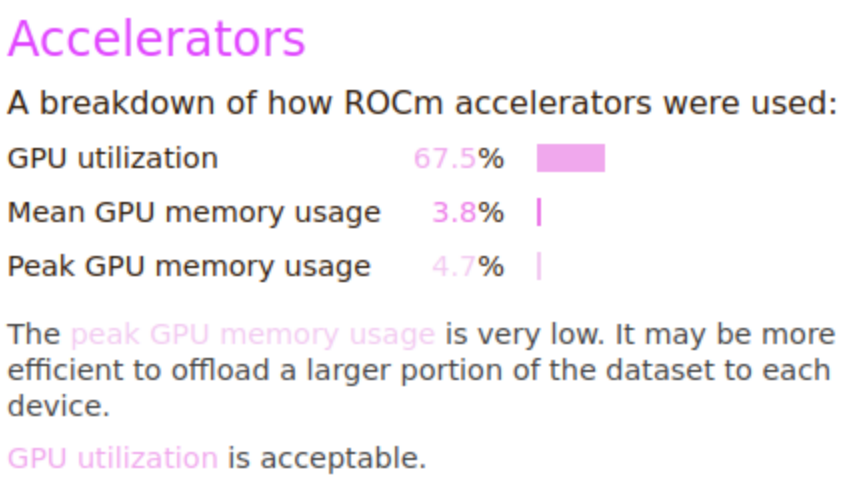
Node power monitoring is implemented using the Cray HSS energy counters.

The Cray HSS energy counters are known to be available, for example, on Cray XK6 and XC30 machines. Version 2 support or higher is needed for CPU power measurements and can be found on machines such as the Cray EX235n. The counter version can be seen at `/sys/cray/pm_counters/version`.

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.

4.4.11 Accelerator breakdown

This section shows the utilization of GPU accelerators by the job. These metrics are collected for NVIDIA GPUs or AMD GPUs.



4.4.11.1 GPU utilization

The percentage of time during which one or more kernels were executing on the GPU, averaged across available GPUs.

4.4.11.2 Mean GPU memory usage

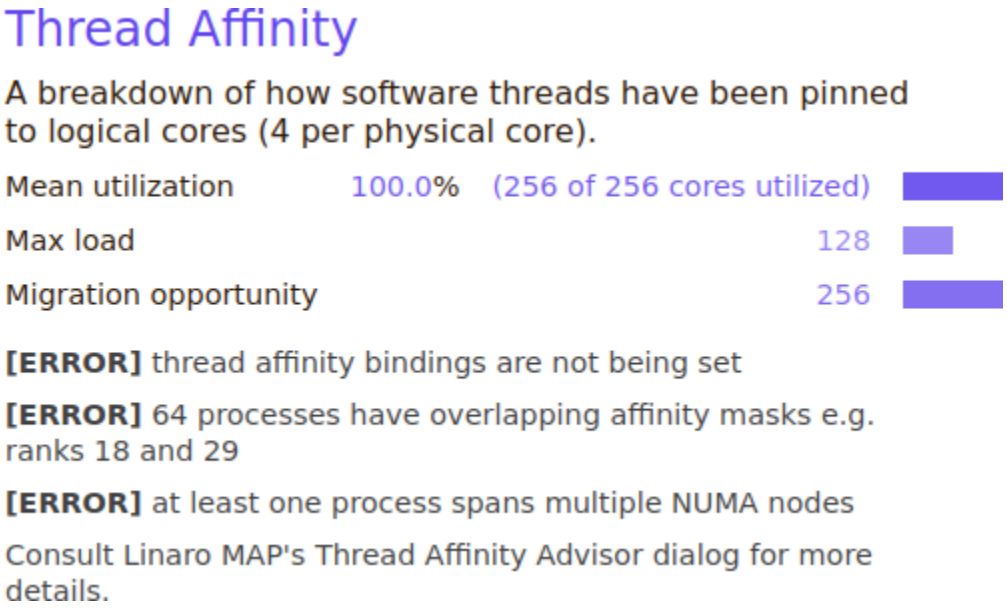
The average amount of memory in use on the GPU cards.

4.4.11.3 Peak GPU memory usage

The maximum amount of memory in use on the GPU cards.

4.4.12 Thread affinity breakdown

This section shows a breakdown of how software threads have been pinned to to logical cores.



Note: The thread affinity breakdown is a licensed feature, and this section will not be generated if it is not available with the currently loaded license.

4.4.12.1 Mean utilization

The percentage and number of logical cores pinned to this job. 100% means all cores have at least one thread that may run on them.

4.4.12.2 Max load

The mean number of threads pinned to each logical core. Values greater than 1 means multiple threads may compete for the same logical core.

4.4.12.3 Migration opportunity

The mean number of logical cores a thread may use. On SMT CPUs this should be less than or equal to the number of logical cores per physical cores. On non-SMT CPUs values greater than 1 should be avoided as it permits thread migration.

4.4.13 Textual performance reports

Textual performance reports contain the same information as *HTML performance reports*, but in a format better suited to automatic data extraction and reading from a terminal:

```
Command:     mpiexec -n 16 examples/wave_c 60
Resources:   1 node (12 physical, 24 logical cores per node, 2 GPUs per node available)
Memory:  15 GB per node, 11 GB per GPU
Tasks:   16 processes
Machine:     node042
Started on:  Tue Feb 25 12:14:06 2014
Total time:  60 seconds (1 minute)
Full path:   /home/user/forge/x.y.z/examples
Notes:

Summary: wave_c is compute-bound in this configuration
Compute:                                     82.4%    (49.4s) |=======|
MPI:                                         17.6%    (10.6s) |=|
I/O:                                          0.0%            |
This application run was compute-bound. A breakdown of this time and advice for investigating further␣
 ↪is found in the compute section below.
Because minimal time is spent in MPI calls, this code might also benefit from running at larger scales.
...
```

You can use a combination of `grep` and `sed` for extracting and comparing values between multiple runs, or for automatically placing this data into a centralized database.

4.4.14 CSV performance reports

A CSV (comma-separated values) output file can be generated using the `--output` argument and specifying a filename with the `.csv` extension:

```
perf-report --output=myFile.csv ...
```

The CSV file will contain lines in a NAME, VALUE format for each of the reported fields. This is convenient for passing to an automated analysis tool, such as a plotting program. It can also be imported into a spreadsheet for analyzing values among executions.

4.5 Perf metrics

The Perf metrics use the Linux kernel `perf_event_open()` system call to provide additional CPU related metrics available for Linaro Performance Reports.

They can be used on any system supported by the Linux `perf` command (also called `perf_event`). These cannot be tracked on typical virtual machines.

Linaro Performance Reports generally requires no extra configuration for Perf metrics before use. For more information about Perf metrics, and specific settings for generating Perf metrics in the reports, see *Configurable Perf metrics*.

CHAPTER

FIVE

SUPPORTED PLATFORMS

5.1 Reference table

This table describes the architectures, operating systems, MPI distributions, compilers, and accelerators that are supported by Linaro Forge, including Linaro DDT, Linaro MAP, and Linaro Performance Reports.

CPU Architecture	Arm AArch64	Intel and AMD (x86_64)
OS	Red Hat Enterprise Linux / CentOS 8, 9, and 10 SuSE Linux Enterprise Server 15 Ubuntu 22.04 and later Amazon Linux 2023	Red Hat Enterprise Linux / CentOS 8, 9, and 10 SuSE Linux Enterprise Server 15 Ubuntu 22.04 and later Amazon Linux 2023
MPI	Cray MPT HPE MPT [a] MPICH MVAPICH [a] Open MPI 4.1, 5	Cray MPT HPE MPT [a] HPE Cray PALS Intel MPI MPICH MVAPICH [a] Open MPI 4.1, 5
Compilers	Arm® Compiler for Linux Arm® Toolchain for Linux Cray Compiling Environment GNU C/C++/Fortran Compiler LLVM Clang/Flang 20+ NVIDIA HPC Compiler	Cray Compiling Environment GNU C/C++/Fortran Compiler Intel Parallel Studio Intel oneAPI HPC Toolkit LLVM Clang/Flang 20+ NVIDIA HPC Compiler
Accelerators	NVIDIA CUDA Toolkit 12, 13	NVIDIA CUDA Toolkit 12, 13 AMD ROCm Toolkit 5.2+, 6, 7 Intel oneAPI 2024.0 - 2025.2.1 for Intel Xe-HPC GPUs (Linaro DDT only)
Python	CPython 3.10 - 3.14	CPython 3.10 - 3.14

[a] These MPIs do not support Express Launch. See *Express Launch (DDT)* for more details.

Note: See *HPE Cray PALS* for more details about HPE Cray PALS support.

Note: See *SLURM* for more details about slurm support.

Note: PGI is renamed to NVIDIA HPC after version 20.7.

Note: Free-threaded Python interpreters are not supported.

5.1.1 Forge

- DDT:
  - Pretty printing of C++ types is not supported for the NVIDIA HPC or Cray compilers.
  - Message queue debugging is supported for Intel MPI, MPICH, MVAPICH, and Open MPI.
- MAP and Performance Reports:
  - MPI wrapper libraries are pre-compiled for these MPIs: Open MPI, Intel MPI 5.x.x, 2017.x, 2018.x and 2019.x, 2021.x, Cray MPT, MPICH and MVAPICH. For other MPIs, these will be created dynamically at run time.
  - The appropriate Linaro Forge sampler and MPI wrapper library must be explicitly linked with statically-linked programs. This mostly applies to the Cray X-Series.
  - MPI wrapper libraries support wrapping calls from MPI C/C++ `mpi.h` interface as well as the Fortran `mpi` and `mpi_f08` modules.

Note: There are some known limitations with the MPI wrapper libraries and the `mpi_f08` module. As such, `mpi_f08` is only supported on Linaro MAP and Linaro Performance Reports with Cray MPT 8, Open MPI >= 5.x.x, MPICH >= 4.2.3 and MVAPICH 4.0. See *MAP and Performance Reports* for more details on the limitations.

5.1.2 Forge Remote Client

The Linaro Forge Remote Client is available for the following platforms:

- MacOS 13 (Ventura) and later.
- Windows 11.
- Any of the Linux platforms listed in **Supported platforms**.

CHAPTER

SIX

GET SUPPORT

6.1 Supporting information

This user guide attempts to cover as many parts of the installation, features, and uses of Linaro Forge as possible. However, there are scenarios or configurations that are not covered, or are only briefly mentioned, or you might on occasion experience a problem using the product. If the solution to your problem is not in this guide, contact Forge Support.

Provide as much detail as you can about the scenario, such as:

- Version number of Linaro Forge. For example, `forge --version` and your operating system, and the distribution, such as Red Hat Enterprise Linux 8.10.

  This information is all available by using the `--version` option on the command line of any Linaro Forge tool:

```
bash$ forge --version

Linaro DDT
Part of Linaro Forge.
Copyright (c) 2023-2024 Linaro Limited. All rights reserved.

Version: 23.1.1
Build ID: 1b2e4195d62e3a46d3a2201bd8aedc4371a5127a
Review ID: I2f12bf428ed0df7b624e9ba0cbb25bf6ccfab291
Patchset ID: 1
Build Platform: centos linux 7.9 x86_64
Build Date: Jan 22 2024 17:46:04

Frontend OS: Ubuntu 22.04.3 LTS
Nodes' OS: unknown
Last connected forge-backend: unknown
```

- The compiler in use and its version number.
- The MPI library and CUDA, ROCm or oneAPI 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 have encountered.
- If possible, obtain and supply a log file for the problem, see *Create a log file*.
- Sometimes it might be helpful to illustrate your problem with a screenshot of the Linaro Forge main window. To take a screenshot, choose the *Take Screenshot* option under the *Window* menu. You are prompted for a file name when you save the screenshot.

6.2 Create a log file

A log file can provide invaluable information to help diagnose and resolve a problem you are reporting to Forge Support. To generate a log file, either select the *Help* ▸ *Logging* ▸ *Automatic* menu option or start Linaro Forge with the `--debug` and `--log` arguments:

```
$ ddt --debug --log=<logfilename>
$ map --debug --log=<logfilename>
```

`<logfilename>` is the name of the log file to generate.

Reproduce the problem using as few processors and commands as possible, and when complete, close the program as usual.

On some systems, this file might be quite large. If so, compress it using a program such as `gzip` or `bzip2` before sending it to Forge Support.

If your problem can only be replicated on large process counts, do not use the *Help* ▸ *Logging* ▸ *Debug* menu item or the `--debug` argument because these generate very large log files. Instead, use the *Help* ▸ *Logging* ▸ *Standard* menu option or the `--log` argument.

Note: If you are connecting to a remote system, the log file is generated on the remote host and copied back to the client when the connection is closed. The copy does not happen if the target application crashes or the network connection is lost.

In these cases, the remote copy of the log file is in the `tmp` subdirectory of the Linaro configuration directory for the remote user account. The directory is `~/.allinea`, unless overridden by the `FORGE_CONFIG_DIR` environment variable.

CHAPTER

SEVEN

GENERAL TROUBLESHOOTING

This appendix offers help with common issues you might encounter while using Linaro Forge.

Also, check that you have the latest version of the product on the Linaro Forge Downloads page.

7.1 GUI cannot connect to an X Server

Linaro Forge does not open and you cannot connect to it when running on a remote server. "Map cannot connect to X server" displays.

A known issue with the free version of `XMing` prevents it working well on Linaro Forge and on other Qt5 projects on Windows.

Linaro recommends using a more up to date X server such as `VcXsrv`.

If you continue to experience problems, contact Forge Support.

7.1.1 Related information

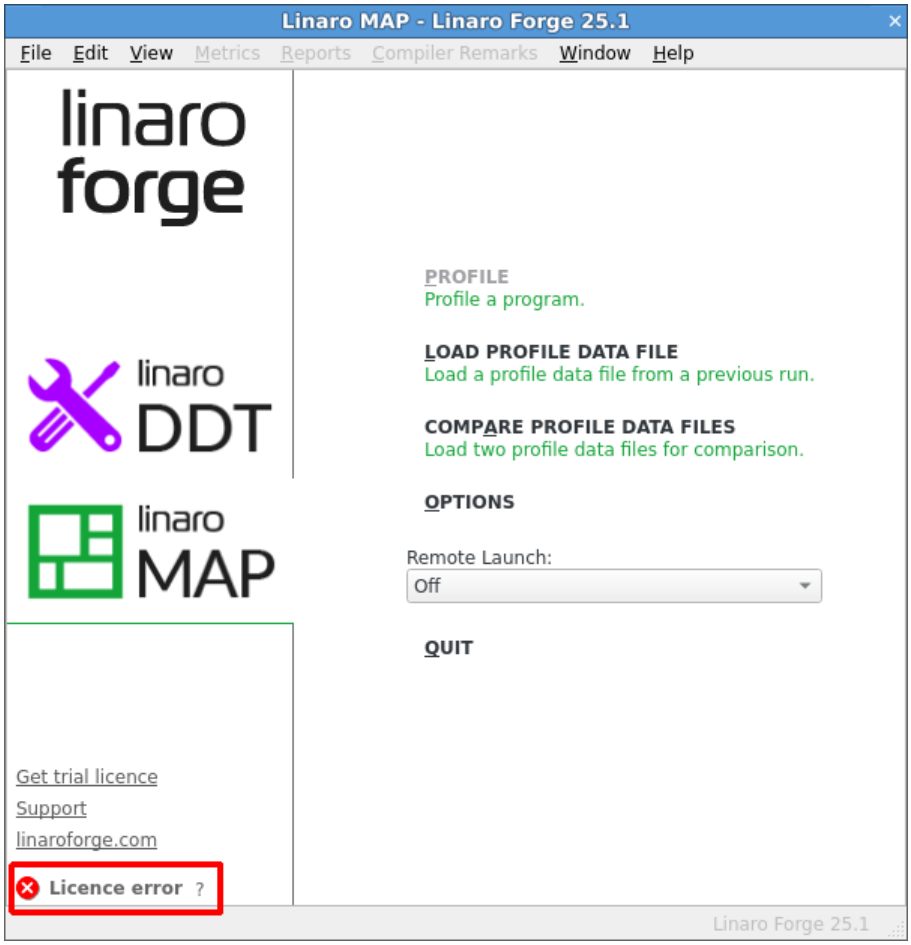
- *Connecting to a remote system*

7.2 Licenses

If you are using Linaro Forge Licence Server, but Linaro Forge products cannot connect to it, see the Use Linaro Forge Licence Server for more troubleshooting information.

7.2.1 License error

The Linaro Forge user interface opens, but shows the message "Licence error ?" at the bottom of the sidebar.

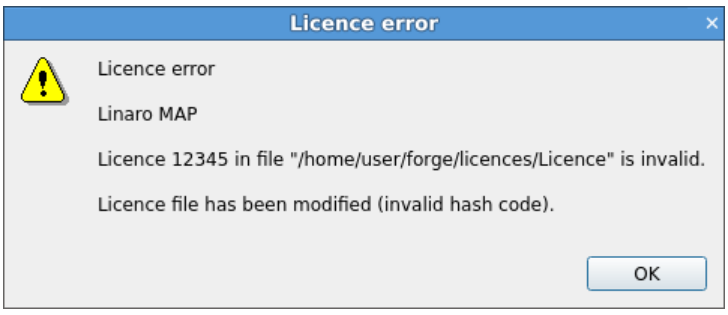


7.2.1.1 Invalid license file

Your license file has been edited, or you are not able to connect to the license server.

7.2.1.2 Solution

- Click ? to see more information about the error.



- Verify that you have a license file for the correct product in the license directory.
- Check the expiry date inside the license to ensure that the license is still valid.

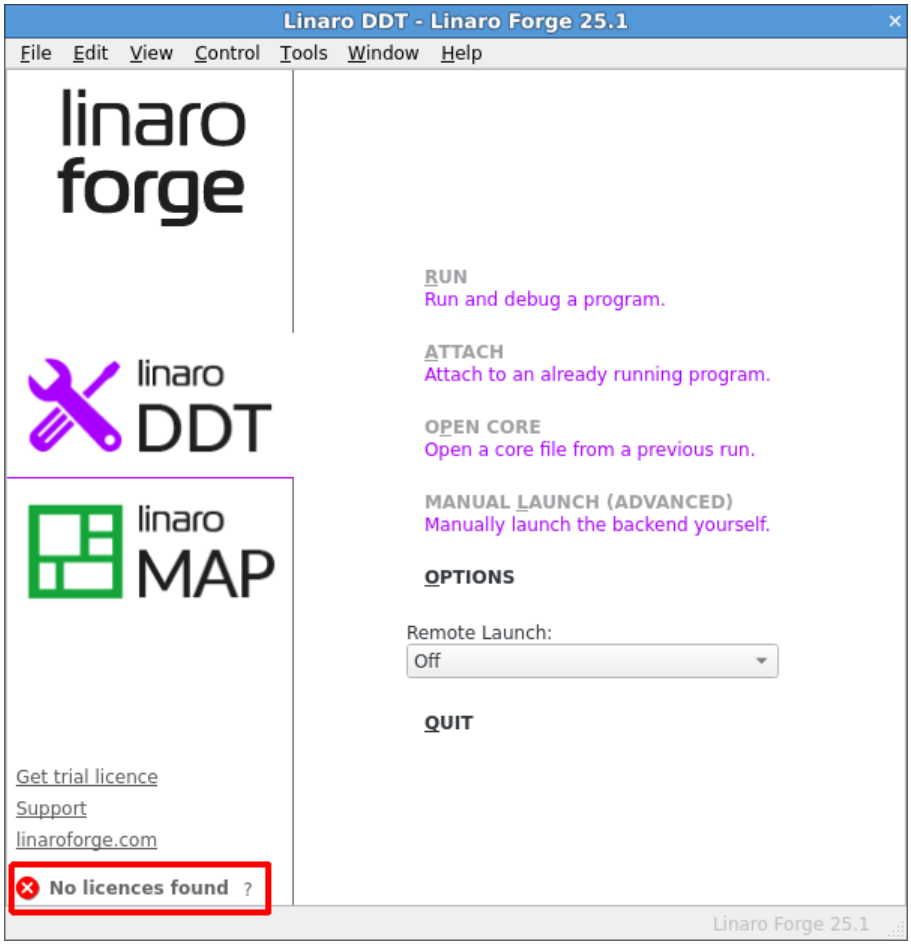
If you continue to experience problems, contact Forge Support.

7.2.1.3 Related information

- *Installing Linaro Forge*
- Use Linaro Forge Licence Server

7.2.2 No licenses found

The user interface opens, but it is unresponsive and shows the message "No licences found" at the bottom of the sidebar.



7.2.2.1 Invalid license file (No licenses)

Linaro Forge requires a license file so that it can run, debug, and profile your programs.

7.2.2.2 Solution

- Contact Linaro Sales to buy a license, or get a free trial license from the Linaro website: free trial licence.
- If you continue to experience problems, contact Forge Support.

7.2.2.3 Related information

- *Installing Linaro Forge*
- Use Linaro Forge Licence Server

7.3 F1 cannot display this document

A blank screen displays instead of this document when you press `F1`.

7.3.1 Corrupt files prevent Qt Assistant starting

There might be corrupt files that are preventing the documentation system (Qt Assistant) from starting.

7.3.2 Solution

Remove the stale files in `$HOME/.local/share/data/Allinea`.

7.4 MPI not detected

When you run an Linaro Forge product (Linaro DDT, Linaro MAP, or Linaro Performance Reports), you are notified about a failure. The nature of the failure is dependent on which product you are running and whether you are running it offline using command-line instructions, or using the Linaro Forge GUI.

7.4.1 MPI settings not configured

When you first run an Linaro Forge product, the `system.config` file is created in your home directory under the `.allinea` folder. This file contains settings for enabling the product to auto detect the correct MPI implementation, and for specifying the default implementation. The failure to run MPI can arise if either of these settings are not configured.

7.4.1.1 Solution

To permanently enable auto detect, and to specify a default MPI type, edit the `system.config` file MPI section in your home directory and ensure that your product can run applications using MPI.

```
[mpi]
auto detect = yes
type = openmpi
```

7.4.1.2 Solution

To permanently set the default MPI type, launch the Linaro Forge GUI from the command-line interface, and select an implementation from the list in **MPI implementation** in the **Run** dialog.

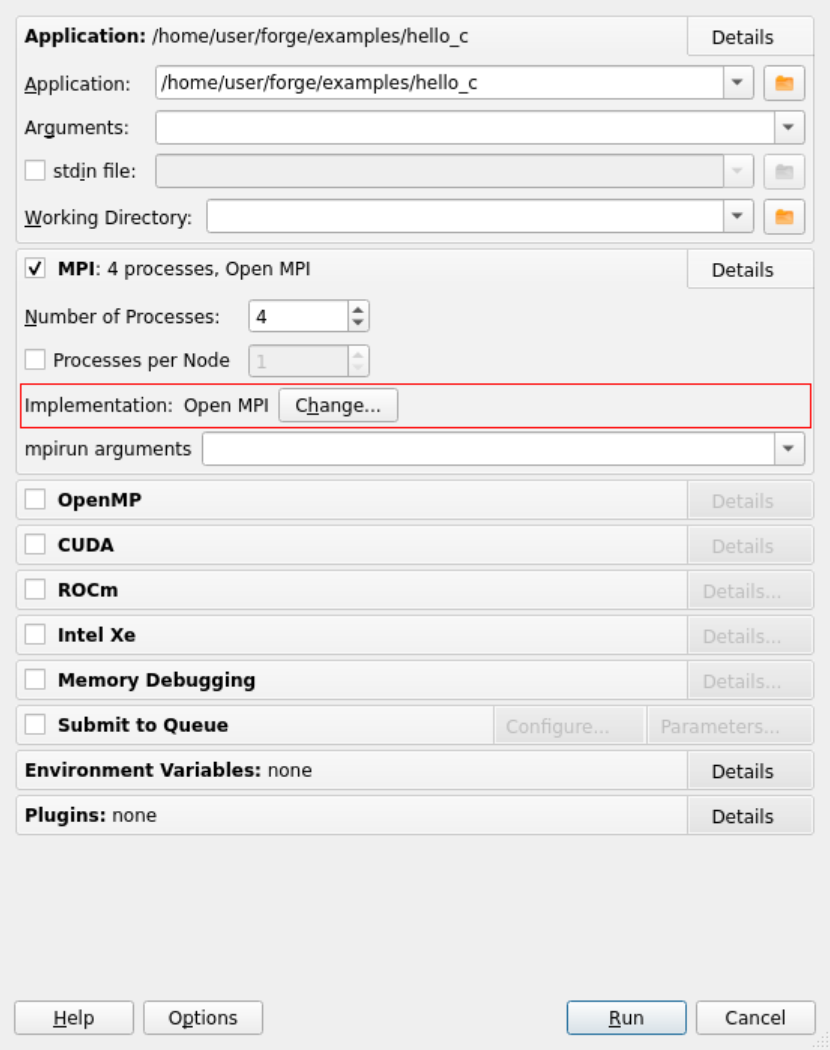
1. Get a list of supported MPIs using `--list-mpis`:

   ```
   $ map --list-mpis
   ```

2. Specify an MPI type for running your program using `--mpi`.

   ```
   $ map --mpi=openmpi-compat -n 1 ./wave_c
   ```

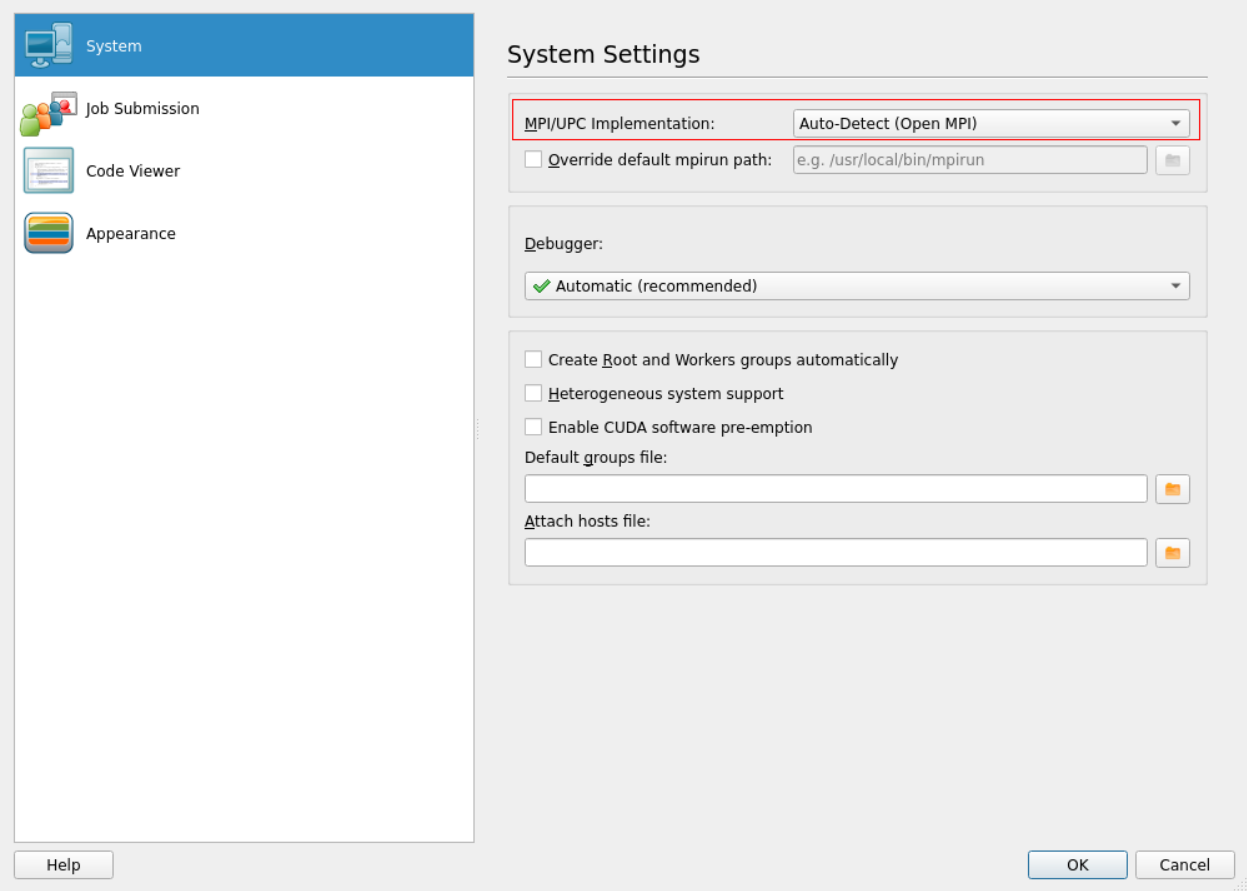
   The GUI launches to display the MPI type you specified on the command line.



7.4.1.3 Solution

To change the MPI type, use the Linaro Forge GUI and set the MPI implementation in the **Run** dialog:

1. Select an MPI implementation in the **Run** dialog.
2. Click *MPI* ▸ *Details* ▸ *Implementation: Change*.
3. Select an implementation from the menu in *Options* ▸ *System Settings* ▸ *MPI/UPC implementation*.



Note: This permanently modifies the MPI section in the `system.config` file with the MPI type you select, and persists for future sessions. However, if auto detect is not set to `yes` in `system.config`, you can still encounter a problem using MPI in subsequent sessions.

7.4.1.4 Solution

To set the MPI type for the current session only, use the command-line interface.

1. Get a list of supported MPIs using `--list-mpis`:

```
$ map --list-mpis
```

2. Specify an MPI type for running your program using `--mpi`.

```
$ map --profile --mpi=openmpi -n 8 ./hello_c
```

Note: This change persists only for the current session and does not modify the `system.config file`.

7.4.1.5 Related information

- *Starting Linaro Forge*.

7.5 Starting a program

This section provides troubleshooting information about starting programs.

7.5.1 Starting scalar programs

There are several potential sources for issues.

7.5.1.1 MPI problem

The most common issue arises when the software reports a problem with MPI and you know your program is not using MPI.

7.5.1.2 Solution

1. Before you attempt to start a program, check *Compiler notes and known issues* and ensure that it is compiled correctly.
2. Select the **Run Without MPI Support** checkbox.

If you have selected this option and the software still refers to MPI, contact Forge Support.

7.5.1.3 Other issues starting a program

Other potential problems are:

- A previous Linaro Forge 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 are connected. You might also see a `QServerSocket` message in the terminal.
- The target program does not exist or is not executable.
- The backend daemon of the Linaro Forge products, `forge-backend`, is missing from the `bin` directory. In this case, check your installation and contact Forge Support.

7.5.2 Starting scalar programs with aprun

Export the following environment variables:

```
export FORGE_MPI_INIT=main
export FORGE_HOLD_MPI_INIT=1
```

(For compilation, see *Compile scalar programs on Cray* in *Cray compiler environment*.)

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 selected, set one process, then click *Run*.

If the above environment variables do not work, try an alternative solution by exporting:

```
export FORGE_STOP_AT_MAIN=1
```

`FORGE_STOP_AT_MAIN` holds the program wherever it was when it attached. This can be before main. For Linaro DDT, set a breakpoint in the main of your program, then select *Play/Continue* to run to this breakpoint.

7.5.3 Starting scalar programs with srun

Export the following environment variables:

```
export FORGE_MPI_INIT=main
export FORGE_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 you are using compatibility launch mode with a scalar program, the *Run* dialog automatically detects SLURM. Keep *MPI* selected, set one process, then click *Run*.

If the above environment variables do not work, try an alternative solution by exporting:

```
export FORGE_STOP_AT_MAIN=1
```

`FORGE_STOP_AT_MAIN` holds the program wherever it was when it attached. This can be before main. For Linaro DDT, set a breakpoint in the main of your program, then select *Play/Continue* to run to this breakpoint.

7.5.4 Problems when you start an MPI program

You encounter problems when you start an MPI program.

7.5.4.1 Solution

- Check whether you can run a single-process (non-MPI) program such as a trivial `Hello, World!` program, resolve any issues that arise, and repeat the attempt to run a multi-process job. Use any issues that you encounter as the starting point for diagnosing the problem.
- Verify that MPI is working correctly by running a job without any of Linaro Forge products applied, such as the example in the **examples** directory of the installation.

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

- Verify that `mpirun` is in the `PATH`, or the environment variable

  `FORGE_MPIRUN` is set to the full pathname of `mpirun`.

7.5.5 Starting multi-process programs

If the progress bar does not report that at least process 0 has connected, the remote `forge-backend` 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.

7.5.5.1 Solution

- If only one, or very few, processes connect, it might be because you have not chosen the correct MPI implementation. Examine the list and look carefully at the options. If you cannot find another suitable MPI, contact Forge Support.
- If a large number of processes are reported by the status bar to have connected, it is possible that some have failed to start because of resource exhaustion, timing out, or, unusually, an unexplained crash.

  To check for time-out problems, set the `FORGE_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 can start, contact Forge Support.

7.5.6 No shared home directory

Your home directory is not accessible to all the nodes in your cluster, and your jobs might fail to start.

7.5.6.1 Solution

1. Open the `~/.allinea/system.config` file in a text editor.
2. Change the shared directory option in the `[startup]` section so that it points to a directory which is available and shared by all the nodes. If no such directory exists, change the `use session cookies` to `no` instead.

7.5.7 Linaro DDT or Linaro 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 names that reports issues with can be reached, using `ping`.

7.5.7.1 Solution

If Linaro DDT fails to find the executable, ensure that it is available in the same directory on every machine.

See *Connecting to compute nodes and remote programs (remote-exec)* for more information on configuring access to remote machines.

7.5.8 Linaro Forge libraries conflict with program libraries

Some of the libraries found in the Linaro Forge `lib` directory structure may conflict with your program libraries, causing your program to fail to run correctly.

7.5.8.1 Solution

Ensure that you do not have the Linaro Forge `lib` directory or any subdirectory defined anywhere in the `LD_LIBRARY_PATH` environment variable.

7.5.9 The progress bar does not move and Linaro Forge times out

This can occur because of one of these issues:

- The program `forge-backend` 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 Linaro Forge times out.

  Ensure that `forge-backend` has all the libraries it needs and that it can run successfully on the nodes using `mpirun`.

- One or more processes (`forge-backend`, `mprun`, `rsh`) could not be terminated.

  This can happen if Linaro Forge is killed during its startup or due to MPI implementation issues.

  You must kill the processes manually, using `ps x` to get the process ids, then `kill` or `kill -9` to terminate them.

This issue can also arise for `mpich-p4mpd`, and the solution is explained in *MPI distribution notes and known issues*.

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

Your home directory must be accessible to all the nodes in your cluster. If not, jobs might fail to start by this method.

7.5.9.1 Related information

See *No shared home directory*.

7.5.10 Resource temporarily unavailable

The Linux system limit on the number of max user processes can prevent Linaro Forge from starting up successfully.

This can generate error messages stating `Resource temporarily unavailable`, or you find that Linaro Forge only starts successfully for a lower number of application processes.

7.5.10.1 Solution

Do one or more of the following to mitigate the issue:

- Increase the system limit on the number of max user processes, for example using `ulimit -u`, or request your system administrator to do so.

  Ensure this is done on the nodes where your application runs.
- Reduce the number of application processes per node.
- Set the environment variable `FORGE_DEBUGGER_CPUS` to a value less than the number of processors per node.

7.6 Attaching

This section provides troubleshooting information about issues when attaching Linaro DDT to running processes.

7.6.1 The system does not allow connecting debuggers to processes (Fedora, Ubuntu)

The Ubuntu `ptrace` scope control feature only allows a process to attach to other processes that it has launched directly.

See https://wiki.ubuntu.com/Security/Features#ptrace for more details.

7.6.1.1 Solution

- To disable this feature until the next reboot, run the following command:

```
echo 0 | sudo tee /proc/sys/kernel/yama/ptrace_scope
```

- To disable the feature permanently, add this line to `/etc/sysctl.d/10-ptrace.conf` or `/etc/sysctl.conf`:

```
kernel.yama.ptrace_scope = 0
```

  This takes effect after the next reboot.

Note: On Fedora, `ptrace` might be blocked by SELinux and Yama. See *The system does not allow connecting debuggers to processes (Fedora, Red Hat)* for more information.

7.6.2 The system does not allow connecting debuggers to processes (Fedora, Red Hat)

The `deny_ptrace` boolean in SELinux, used by Fedora and Red Hat, only allows a process to attach to other processes that it has launched directly.

See https://fedoraproject.org/wiki/Features/SELinuxDenyPtrace for more details.

7.6.2.1 Solution

- To disable this feature until the next reboot, run the following command:

```
setsebool deny_ptrace 0
```

- To disable this feature permanently, run this command:

```
setsebool -P deny_ptrace 0
```

Note: From Fedora version 22 and later, `ptrace` might be blocked by Yama and also the SELinux boolean. See *The system does not allow connecting debuggers to processes (Fedora, Ubuntu)* for more information.

7.6.3 Running processes not shown in the attach window

Running processes that do not show up in the **Attach** window is usually a problem with either the `remote-exec` script or the node list file.

7.6.3.1 Workaround

Ensure 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.

Try running `/path/to/forge/<version>/libexec/remote-exec` manually and check the output.

For example,

```
/path/to/forge/<version>/libexec/remote-exec <hostname> ls
```

If this manual run fails, there is a problem with your `remote-exec` script. If rsh is still 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.

For more information, see *Connecting to compute nodes and remote programs (remote-exec)*.

If you still experience problems with your script, contact Forge Support for assistance.

7.7 Source code view

This section provides troubleshooting information about issues that you might encounter when using the Source code viewer.

7.7.1 No variables or line number information

You must compile your programs with debug information included. You can usually do this, depending on your compiler, by adding the `-g` option to your compile command.

7.7.2 Source code does not appear when you start Forge

If you cannot see any text, the default selected font might not be installed on your system. If not, you can resolve the issue, go to *File* ▸ *Options* (*Linaro Forge* ▸ *Preferences* on Mac OS X) and select a fixed width font such as `Courier`.

If you see a screen of text telling you that Linaro 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 machine 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 Forge Support.

7.7.3 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, the loop block might be folded instead.

7.8 Input/Output

This section provides troubleshooting information for issues you might encounter with a program's input and output.

7.8.1 Output to stderr does not display

Linaro Forge automatically captures anything written to `stdout/stderr` and displays it.

Some shells, such as `csh`, do not support this feature and you might see your `stderr` mixed with `stdout`, or it might not display at all.

7.8.1.1 Solution

Linaro strongly recommends writing program output to files instead, because the MPI specification does not cover `stdout/stderr` behavior.

7.8.2 Unwind errors

When using Linaro MAP, you might see these errors reported in the output in the form:

```
Forge sampler: 3 libunwind: Unspecified (general) error (4/172 samples)
Forge sampler: 3 Maximum backtrace size in sampler exceeded, stack too deep. (1/172 samples)
```

These indicate that Linaro MAP was only able to obtain a partial stack trace for the sample.

7.8.2.1 Solution

If the proportion of samples that generate such errors is low, they can safely be ignored.

If a large proportion of samples exhibit these errors, consult the advice on partial traces in *Intel compilers* or *NVIDIA HPC SDK compilers* if you are using these compilers.

If this does not resolve your issue, contact Forge Support.

7.9 Controlling a program

This section provides troubleshooting information about issues controlling programs.

7.9.1 Program jumps forwards and backwards when stepping through

The program behaves this way when a program is compiled with optimizations. The compiler shuffles your program instructions into a more efficient order and changes the running sequence in the code.

7.9.1.1 Workaround

Linaro recommends compiling with `-O0` when debugging, which disables the reordering behavior, and other optimizations.

Note: If you use the Intel OpenMP compiler, the compiler generates code that appears to jump in and out of the parallel blocks regardless of your `-O0` setting. Therefore, Linaro recommends that you do not step inside parallel blocks.

7.9.2 Linaro DDT might stop responding when using the Step Threads Together option

Linaro DDT might 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 try to **Play to here** to a line that is never reached. In this case, your program would run to the end and then exit.

7.9.2.1 Workaround

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, contact Forge Support.

7.9.3 Stepping into Coroutines

Linaro DDT might not work as expected when trying to **Step Into** C++20 coroutines. When attempting to **Step Into** a coroutine handle, the debugger might actually be in compiler specific code before coroutine initialization. This can cause further stepping through to jump forward and back.

7.9.3.1 Workaround

Linaro recommends using the **Run to here** or **Breakpoint** features if you wish to debug coroutine code.

7.10 Evaluating variables

This section provides troubleshooting information about issues with variables.

7.10.1 Some variables cannot be viewed when the program is at the start of a function

Some compilers produce faulty debug information, forcing Linaro DDT to enter a function during the *prologue*, or the variable might 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.

7.10.1.1 Solution

To view all the variables with their correct values, it might be necessary to play or step to the next line of the function.

7.10.2 Incorrect values printed for Fortran array

Pointers to non-contiguous array blocks (allocatable arrays using strides) are not supported.

7.10.2.1 Workaround

If this issue affects you, contact Forge Support for a workaround or fix.

There are also many compiler limitations that can cause this issue. See Appendix *Compiler notes and known issues* for details.

7.10.3 Evaluating an array of derived types, containing multiple-dimension arrays

The **Locals**, **Current Line** and **Evaluate** views might not show the contents of these multi-dimensional arrays inside an array of derived types.

7.10.3.1 Solution

You can view the contents of the array by clicking the array name and dragging it into the evaluate window as an item on its own, or by using the Multi-Dimensional Array Viewer (MDA). For more information about the MDA viewer, see *Multi-Dimensional Array Viewer (MDA)*.

7.10.4 C++ STL types are not pretty printed

The pretty printers provided with are compatible with GNU compilers version 4.7 and above, and Intel C++ version 12 and above.

7.11 Memory debugging

This section provides troubleshooting information about memory debugging.

7.11.1 The View Pointer Details window says a pointer is valid but does not show you which line of code it was allocated on

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. See the *Compiler notes and known issues* for more details.

The Intel compiler might need the `-fp` argument to allow you to see stack traces on some machines. If this happens with another compiler, contact Forge Support with the vendor and version number of your compiler.

7.11.2 mprotect fails error when using memory debugging with guard pages

This can happen if your program makes more than 32768 allocations; a limit in the kernel prevents Linaro DDT from allocating more protected regions than this.

7.11.2.1 Solution

Try these options to resolve the issue:

- Run `echo 123456 >/proc/sys/vm/max_map_count` (requires root) which increases the limit to 61728 (123456/2, because some allocations use multiple maps).
- Disable guard pages completely. This hinders the ability of Linaro DDT 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 re-enable the feature.

See **Configuration** in *Memory debugging* for information on how to disable guard pages.

7.11.3 Allocations made before or during `MPI_Init` show up in Current Memory Usage but have no associated stack back trace

Memory allocations that are made before or during `MPI_Init` appear in the *Current Memory Usage* window along with any allocations made subsequently.

However, the call stack at the time of the allocation is not recorded for these allocations and does not show up in the *Current Memory Usage* window.

7.11.4 Deadlock when calling printf or malloc from a signal handler

The memory allocation library calls (such as, `malloc`) that are 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 might expect this behavior.

For example, a program that calls `printf` from a signal handler can deadlock when memory debugging is enabled in Linaro DDT, because the C library implementation of `printf` might call `malloc`.

7.11.4.1 Solution

See a table of the functions that can be safely called from an asynchronous signal handler, in the OpenGroup reference to Signal Concepts

7.11.5 Program runs more slowly with Memory Debugging enabled

The Memory Debugging library performs more checks than the memory allocation routines of normal runtime. However, these checks make the library slower.

7.11.5.1 Solution

If your program runs too slowly when Memory Debugging is enabled, there are several options you can change to speed it up.

- 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**.
- Increase the heap check interval from the default of 100 to a higher value. The heap check interval controls how many allocations might occur between full checks of the heap, which can 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.

7.12 MAP specific issues

This section includes details about known issues with Linaro MAP.

7.12.1 MPI wrapper libraries

Unlike Linaro DDT, Linaro 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 Linaro MAP.

See *Supported platforms* for the list of supported MPIs.

You can also try setting `FORGE_WRAPPER_COMPILE=1` and `MPICC` directly:

```
$ MPICC=my-mpicc-command bin/map -n 16 ./wave_c
```

If you have problems, contact Forge Support for more information.

7.12.1.1 MPICH ABI Compatibility and MPI wrapper libraries

The MPICH ABI compatibility initiative is an understanding between various MPICH derived MPI implementations to maintain runtime compatibility between each other. Hence, an application can be compiled with one MPICH and executed with another.

For example, HPE Cray MPI provides a module file `cray-mpich-abi` for running applications built with compatible MPIs using HPE libraries installed on the system.

Linaro MAP will select a precompiled wrapper that is compatible with the loaded environment in this case.

However, if an appropriate compatible precompiled wrapper cannot be detected, Linaro MAP will build one for the system each time you run under Linaro MAP. The following options are available in this scenario.

- Manually compile a MPI wrapper library specifically for the environment in which the application was originally built. Specify this library via the environment variable `FORGE_MPI_WRAPPER` when running Linaro MAP in the MPICH ABI compatible environment.
- Set `FORGE_WRAPPER_COMPILE` and set `MPICC` to the MPI C Compiler that was used to compile the application, not the MPICH ABI compatible compiler.

- Set FORGE_MPI_WRAPPER to a precompiled wrapper shipped with Linaro MAP appropriate for your application. These wrappers can be found in the {<MAP-installation-directory>}/map/wrapper/precompiled directory.

If you have problems, contact Forge Support for more information.

7.12.1.1.1 Related Information

- MPICH ABI Compatibility Initiative
- HPE: Using MPICH ABI Compatibility

7.12.2 Thread support limitations

Linaro MAP and Linaro Performance Reports provide 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 contributes towards the MPI-time of the program, but not the MPI metric graphs.

Additionally, MPI activity on a non-main thread can result in additional profiling overhead due to the mechanism employed by Linaro MAP and Linaro Performance Reports for detecting MPI activity.

7.12.2.1 Solution

Linaro recommends using the threads view mode for interpreting MPI activity instead of the OpenMP view mode, because OpenMP view mode scales MPI-time depending on the resources requested. As a result, non-main thread MPI activity might provide non-intuitive results when detected outside of OpenMP regions.

Warnings display when the user initiates and completes profiling a program that sets MPI_THREAD_SERIALIZED or MPI_THREAD_MULTIPLE as the required thread support.

Linaro MAP and Linaro Performance Reports both fully support making calls to MPI_Init_thread using either MPI_THREAD_SINGLE or MPI_THREAD_FUNNELED to specify the required thread support.

Note: The MPI specification requirements for programs using MPI_THREAD_FUNNELED are the same as the Linaro MAP and Linaro Performance Reports requirements: all MPI calls must be made on the thread that is called MPI_Init_thread.

In many cases, multi-threaded MPI programs can be refactored so that they comply with this restriction.

7.12.3 No thread activity while blocking on an MPI call

Linaro MAP and Linaro Performance Reports are 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, Linaro MAP displays no thread activity for the process executing that call for most of the duration of that MPI call.

7.12.3.1 Solution

See **MPI call duration** in *MPI calls*.

7.12.4 I am not getting enough samples

By default, the starting sample interval is every 20ms. You can change the sampling rate if you get warnings about too few samples on a fast run, or want more detail in the results.

7.12.4.1 Solution

To increase the interval to every 10ms, set environment variable `FORGE_SAMPLER_INTERVAL=10`.

Note: Sampling frequency automatically decreases over time to ensure a manageable amount of data is collected, and does not depend on the length of the run.

Linaro recommends that you avoid increasing the sampling frequency if there are lots of threads or very deep stacks in the target program. This might not leave sufficient time to complete one sample before the next sample is started.

Note: Whether OpenMP is enabled or disabled in Linaro MAP or Linaro Performance Reports, the final script or scheduler values set for `OMP_NUM_THREADS` is used to calculate the sampling interval per thread (`FORGE_SAMPLER_INTERVAL_PER_THREAD`). When configuring your job for submission, check whether your final submission script, scheduler or the Linaro MAP GUI has a default value for `OMP_NUM_THREADS`.

Note: Custom values for `FORGE_SAMPLER_INTERVAL` are overwritten by values set from the combination of `FORGE_SAMPLER_INTERVAL_PER_THREAD` and the expected number of threads (from `OMP_NUM_THREADS`).

7.12.5 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.

7.12.5.1 Solution

To check that your compile line includes `-g`, right-click the *Project Files* panel in Linaro MAP, and select *Add Source Directory*.

If you have any further issues, contact Forge Support for more information.

7.12.6 Linaro MAP reports time spent in a function definition

Any overheads involved in setting up a function call (such as pushing arguments to the stack) are usually assigned to the function definition. Some compilers might assign them to the opening brace ({) and closing brace (}) instead.

If this function is inlined, the situation becomes more complicated and any setup time, such as time for allocating space for arrays, is often assigned to the definition line of the enclosing function.

7.12.7 Linaro MAP does not correctly identify 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
```

Linaro also identifies the AVX-2 variants of these instructions, with a `V` prefix.

If you think that your code contains vectorized instructions that have not been listed and are not being identified in the CPU floating-point/integer vector metrics, contact Forge Support.

7.12.8 Linking with the static Linaro Forge sampler library fails with FDE overlap errors

When linking with the static Linaro Forge sampler library, you might 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 commonly seen on Cray machines using CCE 8.5.0 or later.

7.12.8.1 Solution

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 **Static Linking** in *Prepare a program for profiling* for more details.

If you do not use a Cray or SUSE build of Linaro Forge and you require a binutils 2.25 compatible static library, contact Forge Support.

The error message occurs because the version of `libmap-sampler.a` that you attempted to link was not compatible with the version of `ld` in binutils versions 2.25 or greater.

For Cray machines, there is a separate library called `libmap-sampler-binutils-2.25.a` which is provided for use with this updated linker.

The `make-profiler-libraries` script automatically selects 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, the following errors can occur:

```
/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`.

7.12.9 Linking with the static Linaro Forge sampler library fails with an undefined references to "__real_dlopen"

When linking with the static Linaro Forge sampler library, you might 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
```

7.12.9.1 Solution

To avoid these errors, follow the instructions in *Prepare a program for profiling*.

Note: Look at the use of the `-Wl,@/home/user/myprogram/allinea-profiler.ld` syntax.

7.12.10 Linaro MAP adds unexpected overhead to my program

The Linaro Forge sampler library adds a small overhead to the execution of your program. Usually, this is less than 5% of the wall clock execution time.

However, under some circumstances the overhead can 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 Linaro MAP are 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 incur the higher overhead.

At high per-process thread counts, the Linaro Forge sampler library can incur more significant overhead.

By default, when Linaro MAP detects a large number of threads, it automatically reduces the sampling interval to limit the performance impact.

Sampling behavior can be modified by setting the `FORGE_SAMPLER_INTERVAL` and `FORGE_SAMPLER_INTERVAL_PER_THREAD` environment variables.

For more information on the use of these environment variables, see *MAP and Performance Reports variables*.

The Linaro Forge sampler library can add excessive overhead when processing significant writes to `stdout/stderr` by your application. Avoid this overhead by preventing your application from writing to `stdout/stderr`, for example by setting your application to write to a file instead.

7.12.11 Linaro 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 Linaro MAP and `objdump` to see invalid function lengths and addresses.

This causes Linaro MAP to take an extremely long time disassembling and analyzing apparently overlapping functions containing millions of instructions.

7.12.11.1 Solution

A fix for this was accepted into the OpenBLAS codebase on October 8th 2013. Version 0.2.9 and later are not affected.

To work around this problem without updating OpenBLAS, run `strip libopenblas*.so` to remove the incomplete `.symtab` section without affecting the operation or linkage of the library.

7.12.12 Linaro MAP over-reports MPI, Input/Output, accelerator or synchronization time

Linaro MAP employs a heuristic to determine which function calls to consider as MPI operations.

Any function defined in your code defines that starts with `MPI_` (case insensitive), is treated as part of the MPI library. This causes the time spent in MPI calls to be over-reported by the activity graphs, and the internals of those functions are omitted from the **Parallel Stack View**.

7.12.12.1 Solution

Do not append the prefix `MPI_` to your function names. This is 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:

"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 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.

Do not name a function `start_pes` or any name also used by a standard I/O or synchronization function, such as `write`, `open`, `pthread_join`, and `sem_wait`.

7.12.13 Linaro MAP collects very deep stack traces with boost::coroutine

A known bug in Boost (https://svn.boost.org/trac/boost/ticket/12400) prevents 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.

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);
    }
};
```

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.

7.13 Excessive memory use

If you are running out of memory when you are using an Linaro Forge tool, consider the suggestions in this section.

7.13.1 Reduce processes per node

If your code allows it, run with fewer MPI processes per node. You can reduce the number of MPI processes in total or spread the processes out over more nodes.

Most tool-based memory usage is incurred per-process not per-thread, so you might want to increase the number of threads to compensate.

This impacts where your application spends its time so this might not be applicable when using Linaro MAP or Linaro Performance Reports.

7.13.2 Reduce debug information

Reduce the memory load by compiling some or all of your application with reduced debug information.

Depending on your use case, it might be appropriate to use minimal (file and line information only), or no debug information for some or all of your code.

7.13.2.1 Solution

To enable minimal debug, use the compiler option `-g1` (GCC) or `-debug minimal` (Intel). To disable debug information, do not specify any `-g` option to the compiler or use `-g0`.

- Linaro DDT - Full debug information is required for code that you are debugging. Consider using split DWARF (`-gsplit-dwarf`) if your compiler supports it. Split DWARF allows full debug information to be lazily loaded with finer granularity, potentially saving memory.

  Alternatively, parts of the code that you are sure do not need to be debugged can be compiled with minimal debug information.
- Linaro MAP - Only minimal debug info (file and line numbers only) is required for most functionality. Stack frames that have been inlined can only be displayed if full debug information for that area of code is available).
- Linaro Performance Reports - It is not required to compile with debug information enabled, but avoid stripping binaries of debug information after they have been compiled because this might remove required information.

7.13.3 Profiling settings

7.13.3.1 Issue

Your profiling session exhausts the available free memory on a node while Linaro MAP or Linaro Performance Reports is launching, profiling or analyzing the results when using Open MPI 4+ (excluding Open MPI (Compatibility)), MPICH 4, or Cray MPICH under SLURM (MPMD).

7.13.3.2 Solution

`FORGE_MULTIPLEXER_RATIO` can be used to trade increased time overhead during launch and analysis for reduced memory overhead. This environment variable accepts a range of values depending on how much memory overhead needs reducing. The value takes the form `N:M`. This is the ratio of Linaro MAP worker processes (`N`) to user processes (`M`). Examples are as follows.

- **`FORGE_MULTIPLEXER_RATIO=1:2` results in one Linaro MAP worker process per two ranks.**
  This will provide the most memory reduction for the least increase in time overhead.
- **`FORGE_MULTIPLEXER_RATIO=1:3` results in one Linaro MAP worker process per three ranks.**
  This will further reduce memory overhead for a further increase in time overhead.

Additionally, `FORGE_MULTIPLEXER_RATIO=0` results in one Linaro MAP worker process for all ranks. This provides the most memory reduction with the largest increase in time overhead.

7.13.3.3 Issue

Your program appears to run correctly during profiling, but runs out of memory while Linaro MAP or Linaro Performance Reports collects the results.

7.13.3.4 Solution

Define `FORGE_REDUCE_MEMORY_USAGE=1` in your environment and then rerun Linaro MAP. This environment variable causes the results for each process on a node to be processed sequentially instead of processed in parallel.

This reduces the amount of free memory needed on each node, but takes longer to complete.

CHAPTER

EIGHT

KNOWN ISSUES AND NOTES

8.1 Known issues

Summarizes the most significant known issues for the latest release.

8.1.1 MAP and Performance Reports

The following known issues affect Linaro MAP and Linaro Performance Reports.

- 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 while 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 *Thread view mode*. The uncategorized time coincides with long running MPI calls.
- Linaro MAP and Linaro Performance Reports do not support code that spawns new processes, such as `fork`, `exec` and `MPI_Comm_spawn`. In these cases, Linaro MAP and Linaro Performance Reports only profile the original process.
- Linaro Performance Reports might fail to finalize a profiling session if the cores are oversubscribed on AArch64 architectures. For example, this occurs when attempting to profile a 64 process MPI program on a machine with only 8 cores. This appears as a hang after finishing a profile.
- When Linaro Performance Reports needs to access another machine as part of starting MPICH 3 or Intel MPI it attempts to use the `ssh` secure shell by default. However, this might not always be appropriate if `ssh` is disabled or running on a different port to the default port 22. If startup fails, see *Connecting to compute nodes and remote programs (remote-exec)*.
- Linaro MAP and Linaro Performance Reports might fail to finish a profiling session if an MPI application exits using a means other than `MPI_Finalize`. In this case, try setting `FORGE_JOB_EXIT_TIMEOUT_MS` to ten minutes, to increase the amount of time MAP has to finish profiling.
- When profiling a program where it or it's libraries wrap `dlsym/dlopen` calls (such as `MVAPICH2-GDR`), there is a chance that the Linaro MAP sampler library will get stuck in a loop. In this case, try statically linking the Linaro MAP sampler library instead. See *Static linking* for more details.
- Due to some missing f08 `MPI_Status` conversion functions, Linaro MAP and Linaro Performance Reports only support `mpi_f08` programs run with Cray MPT 8, Open MPI >= 5.0.0, MPICH >= 4.2.3 and MVAPICH 4.0.
- Only precompiled `mpi_f08` wrappers are supported for MVAPICH 4.0, see *MVAPICH*.

- Statically linking against the MPI wrapper library is known to have issues wrapping some `mpi_f08` MPI calls with MPICH-based MPIs (i.e. MPICH and Cray MPT). As such, you may not see some MPI calls and the MPI call metrics may not be accurate. We recommend using the precompiled wrappers or linking with the dynamic libraries instead, if possible. See *Linking* for more details.
- Linaro MAP and Linaro Performance Reports may have undefined behavior when profiling an application that has been linked against the I/O characterization tool darshan. Unload the `darshan` module and recompile the application to workaround this issue.

8.1.2 XALT Wrapper

XALT MPI job launcher wrappers are known to cause several issues when used in conjunction with Linaro 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`.

A workaround is to remove any XALT MPI launcher wrappers from your PATH. If these are provided by a module, this can be achieved by unloading the module.

8.1.3 SLURM support

On Cray X-series systems, only native SLURM is supported. Hybrid mode is not supported.

If you are using Slurm 21.08.0x, where x <= 4, you might see one of these error messages:

```
Invalid generic resource (gres) specification
```

or

```
Invalid Trackable RESource (TRES) specification
```

If you see either of these errors, you can use `FORGE_USE_SSH_STARTUP=1` to get startup working. `FORGE_USE_SSH_STARTUP=1` disables the Forge scalable launch mechanism, which could lead to performance issues if starting a many process job. If you encounter performance issues, try `FORGE_DEBUG_SRUN_ARGS="%jobid% --mem-per-cpu=0 -I -W0 --gpus=0 --overlap"` with the added caveat that this might lead to more issues if you are launching on a GPU compute node.

8.1.4 See also

See also these additional known issues:

Category	Known Issues
MPI Distribution	*MPI distribution notes and known issues*
Compiler	*Compiler notes and known issues*
Platform	*Platform notes and known issues*
General	*General troubleshooting*

8.2 MPI distribution notes and known issues

This appendix has brief notes on many of the MPI distributions supported by Linaro Forge.

Advice on settings and problems particular to a distribution are given here.

Note that Linaro MAP supports fewer MPI distributions than Linaro DDT. See *Reference table* for more details.

8.2.1 Cray MPT

This section only applies when using `aprun`.

For `srun` (Native SLURM mode), see *SLURM*.

Linaro DDT and Linaro MAP have been tested with Cray XT 5/6, XE6, XK6/7, and XC30 systems. Linaro DDT can launch and support debugging jobs in more than 700,000 cores. Linaro Performance Reports has been tested with Cray XK7 and XC30 systems.

Several template files are included in the distribution for launching applications from within the queue, using the Linaro Forge job submission interface. These might 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 where `aprun` is launched, must be accessible using `ssh` from the node where Linaro DDT is running. You can either specify the `aprun` host manually in the **Attach** dialog when scanning for jobs, or configure a hosts list containing all nodes.

Preloading Linaro DDT memory debugging libraries is not supported with `aprun`. If the program is dynamically linked, Linaro MAP and Linaro Performance Reports support preloading the profile libraries with `aprun` (with aprun/ALPS 4.1 or later). Preloading is not supported in MPMD mode and requires that sampling libraries are linked with the application before running on this platform.

See the **Linking** section in *Prepare a program for profiling* for more information.

By default, scripts wrapping Cray MPT are not detected. However, you can force the detection before starting Linaro DDT, Linaro MAP, or Linaro Performance Reports, by setting the environment variable to `yes`.

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

Linaro DDT is compatible with the Cray ATP system, which is the default on some XE systems. This runtime addition to applications automatically gathers crashing process stacks, and can be used to let Linaro 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 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'
```

(continues on next page)

(continued from previous page)

```
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 Linaro DDT.

Linaro DDT can attach using the **Attaching** dialogs described in *Attach 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
```

8.2.2 HPE Cray PALS

Linaro Forge supports the HPE Cray parallel application launch service (PALS) on HPE Cray Shasta and HPCM systems. The Linaro Forge startup method for PALS interacts with the HPE Cray common tools interface (CTI) API.

Note: HPE Cray PALS version >= 1.2.5 and CTI version >= 2.15.10 is required.

Note: Ensure that the `cray-cti` and `cray-pals` modules are loaded.

8.2.2.1 Launching a PALS job

Linaro Forge supports the *Express Launch (DDT)* method for PALS jobs, by prepending the product name to your launch command.

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

Linaro Forge supports the Compatibility Launch mode for PALS jobs.

```
$ map --np=256 examples/wave_c 20
```

8.2.2.2 Attaching to a PALS job

To attach Linaro DDT to a running PALS job, retrieve the associated PBS job ID by running `qstat` and specify it when you launch Linaro DDT from the command line.

```
$ ddt --attach-mpi=1234.login
```

Linaro Forge also supports attaching to a running job by specifying the `mpiexec` job launcher PID, and you can additionally list a specific subset of ranks:

```
$ ddt --attach-mpi=1234 --subset=23,100-112,782,1199
```

Linaro DDT automatically detects and lists running jobs in the **Attach** dialog, described in *Attach to running programs*. Selecting processes to attach to from the list of processes on the **List of all processes** tab of the **Attach** window is not supported.

8.2.2.3 Known issues

- CTI expects MPI arguments compatible with `mpiexec`. Set the `CTI_LAUNCHER_NAME=aprun` environment variable if using `aprun` arguments caused startup issues.
- There is no support for debugging or profiling single-process, non-MPI programs. Convert the program to an MPI program by adding `MPI_Init` and `MPI_Finalize` statements, and run it as a single-process MPI job.

8.2.3 Intel MPI

Select **Intel MPI** from the MPI implementation list.

Linaro DDT, Linaro MAP, and Linaro Performance Reports have been tested with Intel MPI 4.1.x, 5.0.x, and later.

Known issue: Linaro Forge is unable to interoperate with the Intel MPI 2018 and 2019 message queue debugging.

Make sure that you are familiar with 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. If you omit the argument, the default is the release version. However, message queue debugging does not work with this version. To use the debug version, it must be explicitly specified in the argument.

Linaro 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 the Linaro DDT plugins directory. Once you have installed the Intel Trace Analyser and Collector, you must 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
```

Note: The Intel Message Checker only works if you are using the Intel MPI.

Make sure that the Intel `mpiexec` is in your path, and that your application is compiled against the Intel MPI. Launch Linaro DDT, select the plugin checkbox, and debug your application.

If one of the above steps is missed out, Linaro DDT might report an error and say that the plugin could not be loaded.

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

Note: The Intel Message Checker aborts the job after 1 error by default. You can modify this by adding `-genv VT_CHECK_MAX_ERRORS 0` to the **mpirun arguments** field in the **Run** window. See the Intel documentation for more details on this and other environment variable modifiers.

8.2.3.1 Attach dialog

Linaro 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 must list all potential compute nodes individually in the dialog.

Note: The `mpiexec` method of starting MPI processes is deprecated by Intel. Linaro recommends using `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 Linaro DDT.

If you use Spectrum LSF as workload manager in combination with Intel MPI, you might need to set/export `I_MPI_LSF_USE_COLLECTIVE_LAUNCH=1` before executing the job, under these two conditions:

- You get one of these 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
```

- The job is killed during launching/attaching.

8.2.3.2 Known Issues

These versions of Intel MPI do not work with Linaro Forge:

- Intel MPI 2021.12 cannot profile MPI Fortran applications.

8.2.3.3 Related information

- Using IntelMPI under LSF quick guide
- Resolve the problem of the Intel MPI job …hang in the cluster for more details.

8.2.4 MPICH 3

MPICH 3.0.3 and 3.0.4 do not work with Linaro Forge. MPICH 3.1 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.

The message queue data provided by the MPICH debugging support library is limited and results in unreliable information in the **Message Queue** debugging feature.

8.2.5 MPICH 4

MPICH 4.0.x is not supported when launching only one process (see Github).

The message queue data provided by the MPICH 4.0.1 and 4.0.2 debugging support library results in unreliable information in the **Message Queue** debugging feature.

8.2.6 MVAPICH

8.2.6.1 Profiling Wrappers

Only precompiled wrappers (see MPI wrapper libraries) are supported with `mpi_f08` for MVAPICH 4.0. Contact Forge Support should you need to compile wrappers for this MPI version.

8.2.6.2 Message Queue Debugging

To enable message Queue Support, compile MVAPICH with the flags `--enable-debuginfo --enable-shared`. The debug info flag is not enabled by default.

8.2.6.3 Memory Debugging

Memory debugging in Linaro DDT interferes with the on-demand connection system used by MVAPICH2 and MVAPICH 3.0 above a threshold process count, and applications fail to start.

The default threshold value is 64.

As a workaround, set the environment variable `MV2_ON_DEMAND_THRESHOLD` or `MVP_ON_DEMAND_THRESHOLD` (MVAPICH 3.0) to the maximum job size you expect on your system and Linaro DDT can work with memory debugging enabled for all jobs.

Do not set this as a system-wide setting because it might increase startup times for jobs and memory consumption.

8.2.6.4 Scalable Launch

MVAPICH2 and MVAPICH 3.0 offer `mpirun_rsh` instead of `mpirun` as a scalable launcher binary.

1. To use this with Linaro DDT, go to the *System* page (*File* ▸ *Options*, or *Linaro Forge* ▸ *Preferences* on Mac OS X).
2. Select *Override default mpirun path* and enter `mpirun_rsh`.
3. Add `-hostfile <hosts>` in the *mpirun_rsh arguments* field in the *Run* window. `<hosts>` is the name of your hosts file.

8.2.7 Open MPI

There are three different Open MPI choices in the list of MPI implementations to choose from in Linaro Forge when debugging or profiling for Open MPI.

Open MPI
The job is launched with a custom launch agent that, in turn, launches the Linaro Forge daemons.

Open MPI (Compatibility)
`mpirun` launches the Linaro Forge daemons directly. This startup method does not take advantage of the Linaro Forge 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 capable of using the Linaro Forge scalable tree infrastructure.

To launch with `aprun` (instead of `mpirun`), enter one of these commands:

```
ddt --mpi="OpenMPI (Cray XT/XE/XK)" --mpiexec aprun [arguments]
```

```
map --mpi="OpenMPI (Cray XT/XE/XK)" --mpiexec aprun [arguments]
```

8.2.7.1 Known issues

- Message queue debugging does not work with the UCX or Yalla PML, due to UCX and Yalla not storing the required information.
- The version of Open MPI packaged with Ubuntu has the Open MPI debug libraries stripped. This prevents the **Message Queues** feature of Linaro DDT from working.
- 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 debugged. To enable CUDA interoperability with Infiniband, set the CUDA environment variable to `1`.

These versions of Open MPI do not work with Linaro Forge because of bugs in the Open MPI debug interface:

- Open MPI 4.1.0 when compiled with `-O2` or `-O3` optimization flags using NVIDIA HPC compilers.

To resolve any of these issues, select *Open MPI (Compatibility)* for the MPI Implementation.

8.2.8 SLURM

To start MPI programs, use the `srun` command and options instead of your typical MPI `mpirun` command (or equivalent):

```
$ ddt srun -A <account> -t <time> -p <partition_names> -N 1 -n 4 --ntasks-per-node=4 examples/hello
```

Select **SLURM (MPMD)** as the **MPI Implementation** on the **System Settings** page of the **Options** window.

This option works with most, but not all, MPIs. On Cray, Hybrid SLURM mode (that is, SLURM + ALPS) is not supported. Instead, you must start your program with the Cray `aprun` command. See *Cray MPT*.

You can use SLURM as a job scheduler with Linaro DDT and Linaro MAP by using a queue template file. See an example of this in `templates/slurm.qtf` supplied with the installation. See *Integration with queuing systems* for more information on how to customize the template.

8.3 Compiler notes and known issues

When compiling for a Linaro DDT debugging session, always compile with a minimal amount of optimization, or no optimization. Some compilers reorder instruction execution and omit debug information when compiling with optimization enabled.

For a list of supported compiler versions, see *Reference table*.

8.3.1 AMD OpenCL compiler

Not supported by Linaro MAP and Linaro Performance Reports.

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"
```

Run the example codes in the AMD OpenCL toolkit on the CPU by adding the parameter `--device cpu`. With the above environment variable set, this results in debuggable OpenCL.

8.3.2 Arm Compiler for Linux

Debugging of Fortran code might be incomplete or inaccurate. For more information, check the known issues section in the Arm Compiler for Linux release notes.

8.3.3 Arm Toolchain for Linux

See *LLVM*.

8.3.4 Cray compiler environment

Call-frame information can sometimes be incorrectly recorded, which may lead to Linaro DDT stepping into a function instead of stepping over it. This might also result in time being allocated to incorrect functions in MAP.

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

Known Issue: If you are compiling static binaries, then linking in the Linaro DDT memory debugging library is not straightforward for F90 applications. You must do the following:

1. Manually rerun the compiler command with the `-v` (verbose) option to get the linker command line. Ensure that the object files are already created.
2. Run `ld` manually to produce the final statically linked executable. For this, the following path modifications are required 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 accelerator support.

Linaro DDT supports UPC using the Cray UPC compiler, but Linaro MAP and Linaro Performance Reports do not.

8.3.4.1 Compile scalar programs on Cray

To launch scalar code with aprun, using Linaro Forge on Cray, you must link your program 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.

- To use Linaro MAP or Linaro Performance Reports, see **Linking** in *Prepare a program for profiling*.
- To use `aprun` to launch your program, see *Starting scalar programs with aprun*.
- To use SLURM, see *Starting scalar programs with srun*.

8.3.5 GNU

Do not use the compiler flag `-fomit-frame-pointer` because this can prevent Linaro Forge identifying the lines of code that your program has stopped.

When using GCC 7.3.0, time profiled in OpenMP regions can be imprecisely attributed to caller functions.

For GNU C++, large projects can result in large amounts of debug information, which can lead to high memory.

The `-foptimize-sibling-calls` optimization (used in `-O2`, `-O3` and `-Os`) interferes with the detection of some OpenMP regions. If your code is affected by this issue, add `-fno-optimize-sibling-calls` to disable it, and allow Linaro MAP and Linaro Performance Reports to detect all the OpenMP regions in your code.

Using the `-dwarf-2` flag together with the `-strict-dwarf` flag can cause problems with stack unwinding, and result in a `cannot find the frame base` error. DWARF 2 does not provide all the information necessary 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, and the error message is reported. Removing `-strict-dwarf` fixes this problem.

8.3.6 Intel compilers

Linaro Forge supports Intel compilers.

For details see *Reference table*.

- You might experience issues with missing or incomplete stack traces. For example, `[partial trace]` entries display in Linaro MAP, or no stack traces for allocations display in the Linaro DDT **View Pointer Details** window.

  Try recompiling your program with the `-fno-omit-frame-pointer` argument.

- Some optimizations that are performed when you specify `-ax` options in IFC/ICC can result in programs which cannot be debugged or profiled due to lack of frame pointer information.

- Some optimizations that are performed using Interprocedural Optimization (IPO), (implicitly enabled by the `-O3` flag) can interfere with the ability of Linaro MAP to display call stacks. This makes it more difficult to understand what the program is doing.

  To prevent this disable IPO by adding `-no-ip` or `-no-ipo` to the compiler flags. The `-no-ip` flag disables IPO within files, and `-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 always optimizes parallel regions, regardless of `-O0` parameters. This can cause issues with reordered instructions or variables that can not be read.

- 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 rename its extension to `.f` or `.f90` and/or remove the `-fpp` flag from the compile line instead.

  Alternatively, to help Linaro DDT discover the source file, right-click on the *Project Files* window, select *Add/view source directory*, and add the correct directory.

- Some versions of the compiler emit incorrect debug information for OpenMP programs which might 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.

- If you are compiling static binaries, for example on a Cray XT/XE machine, then linking in the Linaro 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:

- If you are compiling static binaries, linking on a Cray XT/XE machine in the Linaro DDT memory debugging library is not straightforward for F90 applications. You must manually rerun the last `ld` command (as seen with `ifort -v`) to include `-L{ddt-path}`/lib/64 -ldmalloc in two locations:

  – Include immediately prior to where `-lc` is located.

  – Include the `-zmuldefs` option at the start of the `ld` line.

- STL sets, maps and multi-maps cannot be fully explored, because only the total number of items is displayed. Other data types are unaffected.

- To disable pretty printing, set the environment variable `FORGE_DISABLE_PRETTY_PRINTING` to 1 before starting Linaro DDT. This enables you to manually inspect the variable in the case of, for example, the incomplete `std::set` implementations.

- The Intel LLVM C++ compiler does not emit debug information for libstdc++ by default. Force it to be included by recompiling your program with the `-fno-limit-debug-info` option.

8.3.7 LLVM

In some cases GDB is unable to recognize symbols from DWARF produced by Flang, where the symbol DW_AT_name differs from the DW_AT_linkage_name. This can be worked around by inspecting the DWARF using `readelf -w`, and using the value of the DW_AT_linkage_name as an expression in Linaro DDT (a cast to the correct type will also be required).

As of Flang 20.1, OpenMP support is classed as experimental.

8.3.8 NVIDIA HPC SDK compilers

Note: PGI 20.7 has been re-branded to the NVIDIA HPC SDK. A result of this change is that pgcc, pgc++, and pgfortran have been aliased to nvc, nvc++ and nvfortran respectively. This has not affected the functionality of the compilers with Linaro Forge. The old compiler directives can still be used directly without making use of the `nv` prefixed directives.

If you experience problems with missing or incomplete stack traces (that is `[partial trace]` entries in Linaro MAP or no stack traces for allocations in the **View Pointer Details** window of Linaro DDT), compile your program with the `-Mframe` and `-Meh_frame` arguments.

Some known issues are listed here:

- Included files in Fortran 90 generate incorrect file and line information in debug information. The information provides refers to **including** file but displays the line numbers from the **included** file.
- The PGI compiler might emit incorrect line number information, particularly for optimized code. This can cause Linaro Forge to show your program on a different line to the one expected.
- When using memory debugging with statically-linked PGI executables (`-Bstatic`), you must add a `localrc` file to your PGI installation because of the in-built ordering of library linkage for F77/F90. The `localrc` file defines the correct linkage when using Linaro DDT and (static) memory debugging. Append the following to `<{>pgi-path>/bin/localrc`:

```
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` now lists `-Bstaticddt` as a compilation flag. You must 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.

- With the PGI compiler, 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 might be ignored by Linaro DDT. If this affects you, contact Forge Support.
- Linaro DDT might 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` might also be displayed. The extra symbols are added by the compiler and can be ignored.
- The PGI 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.
- With the PGI compiler, you might experience issues with **Pin to address** for **Watchpoints**. If this affects you, contact Forge Support.
- Linaro MAP displays a warning when running OpenMP programs compiled with PGI 20.3 or earlier. "Another OpenMP runtime library has been detected in this application…" This warning relates to earlier versions of PGI implementing a subset of OpenMP. Upgrade to PGI 20.4 or later.

  Alternatively, silence the warning by setting the `NV_OMP_DISABLE_WARNINGS=true` environment variable.

8.4 Platform notes and known issues

This appendix provides details about specific issues affecting platforms. If a supported machine is not listed in this section, then there are no known issues.

8.4.1 Cray

This section describes several issues when using Linaro Forge tools on Cray and offers workarounds steps you can take to avoid them.

- If you are using Linaro MAP on Cray, Linaro recommends you read the following topics in *Prepare a program for profiling*.
  - **Debugging symbols**
  - **Static linking on Cray X-Series systems**
  - **Dynamic and static linking on Cray X-Series systems using the modules environment**

  Linaro supplies module files in `FORGE_INSTALLATION_PATH/share/modules/cray`.
- Compilers may link statically on this platform by default. See *Compiler notes and known issues* for compiler-specific information on static linking.
- See *MPI distribution notes and known issues* for information on Cray MPT.
- Cray GPU debugging requires a working `TMPDIR` to be available, if `/tmp` is not available. You must ensure 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. Linaro DDT automatically propagates this environment variable to the compute nodes.
- An extra step is necessary if you run single-process scalar codes (that is, non-MPI/SHMEM/UPC applications) on the compute nodes. These are required to be executed by `aprun`. However, `aprun` does not execute these applications using the typical debug-supporting protocols.

- Running a dynamically-linked, single-process, non-MPI program that runs on a compute node (that is, non-MPI CUDA or OpenACC code) requires that you add the `-target=native` flag to the compiler. This flag prevents the compiler linking in the MPI job launch routines, which can interfere with debuggers on this platform. Alternatively, you can convert the program to an MPI program by adding `MPI_Init` and `MPI_Finalize` statements, and run it as a one-process MPI job.

8.4.2 GNU/Linux systems

This section describes the known issues when using Linaro Forge tools on GNU/Linux.

8.4.2.1 General

The following prerequisite packages are required to run Linaro Forge:

Red Hat Enterprise Linux	SuSE Linux Enterprise Server	Ubuntu	Amazon Linux
expat	libexpat1	libexpat1	expat
fontconfig	libfontconfig1	libfontconfig1	fontconfig
freetype	libfreetype6	libfreetype6	freetype
glibc	glibc	libc6	glibc
gmp	libgmp10	libgmp10	gmp
libICE	libICE6	libice6	libICE
libSM	libSM6	libsm6	libSM
libX11	libX11-6	libx11-6	libX11
libX11-xcb	libX11-xcb1	libx11-xcb1	libX11-xcb
libXext	libXext6	libxext6	libXext
libgcc	libgcc_s1	libgcc-s1	libgcc
libstdc++	libstdc++6	libstdc++6	libstdc++
xz-libs	liblzma5	liblzma5	xz-libs
zlib	libz1	zlib1g	zlib

If Linaro Forge fails to launch, for example with the following error, it can mean that one or more required packages are not installed on your system:

```
Unable to load the Qt Plugins.
```

The above packages are available for installation on all of the Linaro Forge supported platforms. For more information, see *Reference table*.

8.4.2.2 Amazon EC2

In order to access the CPU metrics on this system, a whole machine instance is required (e.g. `.metal` or `hpc` instances).

8.4.2.3 Startup

GDB optimizes its operations by constructing an internal symbol table upon loading a file. This can cause significant delay when starting a session in Linaro Forge with a large application. It is recommended that the symbol data be embedded into the file to improve startup times as follows:

` gdb-add-index ./wave_c `

See the GDB documentation for more details.

8.4.2.4 Known issues

- For best performance, Linaro DDT, Linaro MAP, and Linaro Performance Reports require that debug symbols for the runtime libraries are installed, in addition to debug symbols for the program itself.

  Without debug symbols, Linaro DDT might show the incorrect values for local variables in program code if the program is currently stopped inside a runtime library.

  Without debug symbols, Linaro MAP might report time in partial traces or unknown locations. Linaro recommends as a minimum requirement, that the debug symbols are installed for `glibc` and OpenMP (if applicable).

  See the documentation for your operating system for instructions on how to install debug symbols.

- In order to secure inter-process communication, Linaro Forge makes use of the OpenSSL library (version 3).

  By default, Linaro Forge will try to use the OpenSSL library installed on the system, in preference to a version included with Linaro Forge. If the system OpenSSL cannot be used for any reason, Linaro Forge can time out when trying to connect to other processes at startup. This can occur, for example, when OpenSSL configuration settings offer ciphers that cannot be used (perhaps due to system security settings, such as enabling FIPS mode).

  This issue can be addressed by adjusting the system OpenSSL configuration settings, or by setting the environment variable `QT_EXT_OPENSSL_SKIP_SYSTEM_OPENSSL_LOAD=1` to use the copy of OpenSSL included with Linaro Forge.

8.4.3 Intel Xeon

Intel Xeon processors, starting with Sandy Bridge, include Running Average Power Limit (RAPL) counters. Linaro MAP can use the RAPL counters to provide energy and power consumption information for your programs.

8.4.3.1 Enabling RAPL energy and power counters when profiling

To enable the RAPL counters to be read by Linaro 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, Linaro has backported the `powercap` and `intel_rapl` modules for older kernel releases.

8.4.4 NVIDIA CUDA

There are a number of issues you should be aware of:

- Linaro DDT 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 Linaro MAP and Linaro Performance Reports are not available for statically-linked programs.
- CUDA metrics in Linaro MAP are measured at the node level, not the card level.
- NVIDIA Linux driver 418.43 or later might restrict GPU profiling to users with administrative privileges (`CAP_SYS_ADMIN` capability set). See the following NVIDIA page for details and instructions for disabling this restriction: NVIDIA Development Tools Solutions - ERR_NVGPUCTRPERM: Permission issue with Performance Counters.

- Cray CCE 8.1.2 OpenACC and previous releases fail to generate debug information for local variables in accelerated regions. Install CCE 8.1.3 to address this issue.
- When debugging a CUDA application, adding watchpoints on kernel code is not supported.
- When debugging a CUDA application, adding watchpoints on host code is only supported in CUDA 11.0 or later.
- When debugging a CUDA application, using the **Step threads together** box and **Run to here** to step into OpenMP regions is not supported. Use breakpoints to stop at the required line.
- When CUDA is set to **Detect invalid accesses (memcheck)**, placing breakpoints in CUDA kernels is only supported in CUDA 11.
- **Detect invalid accesses (memcheck)** is not supported with CUDA 12.
- **GPU kernel analysis** is not supported with CUDA 13.
- You may experience a hang during profiling when CUDA Kernel Analysis mode is enabled for CUDA Toolkit >= 12.0.1. If you encounter this issue, please contact Forge Support.

8.4.5 AMD ROCm

There are a number of issues you should be aware of:

- For Linaro DDT see *Known issues and limitations*.
- For Linaro MAP, see *Known issues and limitations*.
- ROCm metrics in Linaro MAP and Linaro Performance Reports are not available for statically-linked programs.
- ROCm metrics in Linaro MAP are measured at the node level, not the card level.

8.4.6 Intel Xe

There are a number of issues you should be aware of:

- For Linaro DDT see *Known issues and limitations*.
- Linaro MAP and Linaro Performance Reports only support profiling the CPU code, profiling of GPU kernels or OpenMP target offload regions is not supported.

8.4.7 Arm

This section describes the known issues when using Linaro Forge tools on Arm.

8.4.7.1 Arm®v8 (AArch64) known issues

- Linaro MAP and Linaro Performance Reports might 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 in Linaro MAP.
- Some Linux kernels have a bug that prevents Linaro MAP unwinding out of the vdso. Common vdso methods are `clock_gettime`, `clock_getres` and `gettimeofday`. This is known to happen in v5.4 and is known to be fixed since v5.8.

8.4.8 Mac OS X

The following menu items are not supported:

- **Edit Special Characters**
- **Edit Start Dictation**
- **View Enter Full Screen**
- **View Show Tab Bar**

CHAPTER

NINE

CONFIGURATION

Linaro Forge shares a common configuration file between Linaro DDT, Linaro MAP, and Linaro Performance Reports. This makes it easy for you to switch between tools without reconfiguring your environment each time.

Linaro Performance Reports generally requires no extra configuration before use. If you only intend to use Linaro Performance Reports and you have verified that it works on your system, you can safely ignore most of the information in this section.

9.1 Configuration files

Linaro Forge uses two configuration files: the system configuration file `system.config` file and the user-specific `user.config`. The system configuration file specifies properties such as MPI implementation. The user-specific configuration file describes user preferences, such as font size. The files are controlled by environment variables:

FORGE_USER_CONFIG

User-specific settings file. Stores GUI settings such as fonts, color schemes, which tabs and columns are shown/hidden. You may wish this file to be shared among all clusters you have access to.

Defaults to `${FORGE_CONFIG_DIR}/user.config`.

FORGE_SYSTEM_CONFIG

System-specific settings file. Stores startup settings, breakpoints, enabled/disabled metrics etc. You will wish this file to be specific to each cluster you have access to.

Defaults to `${FORGE_CONFIG_DIR}/system.config`.

FORGE_CONFIG_DIR

Directory where Linaro Forge configuration, cache, and temporary files are stored. Defaults to `${HOME}/.allinea`.

Warning: These configuration files and directories contain user specific configuration and authentication information.

To prevent unauthorized users modifying sensitive files and directories, and introducing unsafe code into your environment, ensure that you assign only the minimum permissions that are required, and avoid group or world-writable permissions.

9.1.1 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 Linaro Forge. In this case, users no longer need to provide configuration for site-specific aspects such as queue templates and job submission.

Warning: To prevent unauthorized users modifying sensitive files and directories, and introducing unsafe code into your environment, ensure that you assign only the minimum permissions that are required, and avoid group or world-writable permissions.

Configure Linaro Forge normally and run a test program to make sure all the settings are correct. When you are satisfied with your configuration, execute this command:

```
forge --clean-config
```

The `--clean-config` option removes any user-specific settings from your system configuration file and creates a `system.config` file that can provide the default settings for all users on your system. Instructions about how to do this are printed when `--clean-config` completes.

Note: Only the `system.config` file is generated. Linaro Forge also uses a user-specific `user.config`, which is not affected.

If you want to use to attach to running jobs, you must also create a file called `nodes` in the installation directory which lists the compute nodes to which you want to attach. See *Attach to running programs* for details.

9.1.2 Startup scripts

At startup, Linaro Forge searches for a sitewide startup script called `allinearc` in the root of the installation directory. If this file exists, the software sources it and then starts the tool. When using the remote client, the software sources this startup script, and then starts any sitewide remote-init remote daemon startup script.

Similarly, you can also provide a user-specific startup script in `~/.allinea/allinearc`.

Warning: To prevent unauthorized users modifying sensitive files and directories, and introducing unsafe code into your environment, ensure that you assign only the minimum permissions that are required, and avoid group or world-writable permissions.

Note: If the environment variable is set, the software looks in `/allinearc` instead. When using the remote client, the software sources the user-specific startup script followed by the user-specific `~/.allinea/remote-init` remote daemon startup script.

9.1.3 Importing legacy configuration

If you have used a version of Linaro DDT prior to version 4.0, your existing configuration is imported automatically. If the `DDTCONFIG` environment variable is set, or you use the `--config` command-line argument, the existing configuration is imported. However, the legacy configuration file is not modified, and subsequent configuration changes are saved as described in the previous sections.

9.1.4 Converting legacy sitewide configuration files

If you have existing sitewide configuration files from a version of Linaro DDT prior to 4.0 you must convert them to the new 4.0 format. You can do this using the following command line:

Note: Ensure that `newconfig.ddt` does not exist before you submit the command.

```
forge --config=oldconfig.ddt --system-config=newconfig.ddt --clean-config
```

9.1.5 Using shared home directories on multiple systems

If your site uses the same home directory for multiple systems you might want to use a different configuration directory for each system.

You can do this by specifying the `FORGE_CONFIG_DIR` environment variable before starting Linaro Forge. If you use the `module` system, you can set `FORGE_CONFIG_DIR` according to the system on which the module was loaded.

For example, if you have two systems: **harvester** with login nodes `harvester-login1` and `harvester-login2`, and **sandworm** with login nodes `sandworm-login1` and `sandworm-login2`, you can add something like the following code to your module file:

```
case $(hostname) in
  harvester-login*)
    FORGE_CONFIG_DIR=$HOME/.allinea/harvester
    ;;
  sandworm-login*)
    FORGE_CONFIG_DIR=$HOME/.allinea/sandworm
    ;;
esac
```

9.1.6 Using a shared installation on multiple systems

If you have multiple systems sharing a common installation, you can have a different default configuration for each system. You can use the `FORGE_DEFAULT_SYSTEM_CONFIG` environment variable to specify a different file for each system. For example, you can add something like the following code to your `module` file:

```
case $(hostname) in
  harvester-login*)
    FORGE_DEFAULT_SYSTEM_CONFIG=/sw/forge/harvester.config
        ;;
  sandworm-login*)
    FORGE_DEFAULT_SYSTEM_CONFIG=/sw/forge/sandworm.config
```

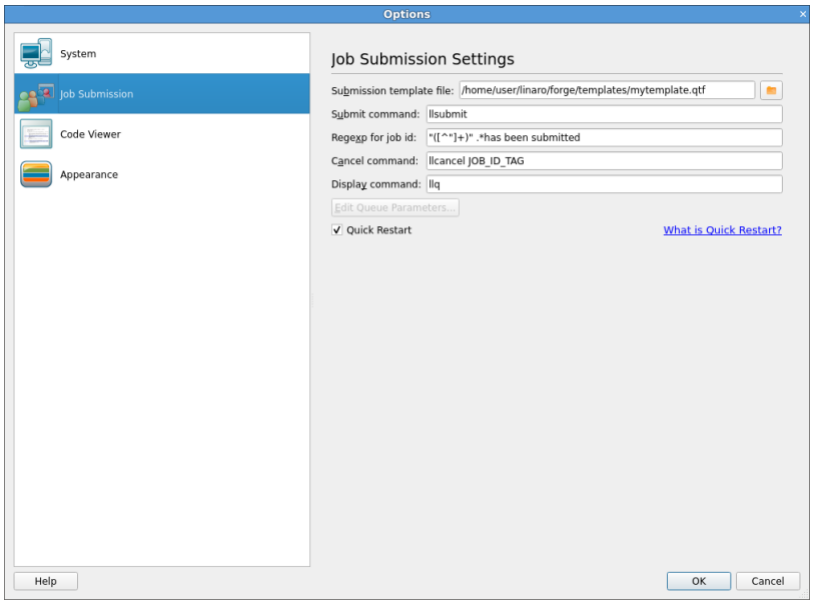
(continues on next page)

(continued from previous page)

```
  ;;
esac
```

9.2 Integration with queuing systems

You can configure Linaro Forge 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.



Linaro MAP is usually run as a wrapper around `mpirun` or `mpiexec`, using the `--profile` argument. Linaro recommends using a modified version of your existing queue submission when your MPI launch command is prefixed by `map --profile` rather than configuring Linaro 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 can provide a configuration file containing the correct settings, and remove the need for individual users to configure their own settings and scripts.

In this mode, Linaro 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. `/path/to/forge//templates/sample.qtf`, demonstrates the process of creating a template file in some detail.

9.3 Template tutorial

Typically, your queue script probably ends in a line that starts `mpirun` with your target executable. In most cases, you can replace that line with `AUTO_LAUNCH_TAG`. For example, if your script currently has this line:

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

Create a copy of it and replace that line with:

```
AUTO_LAUNCH_TAG
```

Select this file as the *Job Submission Settings* ▸ *Submission template file* in the *Options* window.

Note: You no longer need to explicitly specify the number of processes, and so on. Instead, specify the number of processes, program name, and arguments in the *Run* window.

Enter the *Submit command* with the command you usually use to submit your job, such as `qsub` or `sbatch`. Use the *Cancel command* with the command you usually use to cancel a job, for example `qdel` or `scancel`. Use the *Display command* with the command you usually use to display the current queue status, such as `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*.

When 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 could 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 *Queue template tags* for a full list of tags.

9.3.1 The template script

The template script is based on the file you would typically use to submit your job. This is usually 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 Linaro 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 detecting them as a required field, the comment line must begin with `##`.

9.3.2 Configure queue commands

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

When you start a session, Linaro Forge generates a submission file and appends 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`, you must enter `job_submit -u myusername -f` as the submit command.

To cancel a job, Linaro Forge uses a regular expression that 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 *Job ID regular expression* for details.

9.3.3 Configure how job size is chosen

Linaro Forge offers a number of flexible ways to specify the size of a job. You can choose whether *Number of Processes* and *Number of Nodes* options appear in the *Run* window, or whether these should be implicitly calculated. Similarly, you can 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, the tag is always replaced by the *Processes per node* value from the *Run* dialog, even if the option is unselected there.

9.3.4 Quick restart

Linaro DDT allows you reuse an existing queued job to quickly restart a run without resubmitting it to the queue, with the requirement that your MPI implementation supports this. Select the *Job Submission Options* ▸ *Quick Restart* checkbox on the on the *Options* window. See *Optional configuration*.

When using the quick restart, your queue template file must use `AUTO_LAUNCH_TAG` to execute your job.

For more information on `AUTO_LAUNCH_TAG`, see **Using AUTO_LAUNCH_TAG** in *Launching*.

9.4 Connecting to compute nodes and remote programs (remote-exec)

Linaro Forge attempts to use the `ssh` secure shell by default when it needs to access another machine for remote launch, or as part of starting some MPIs.

However, this might not always be appropriate, because `ssh` can be disabled or run on a different port to the default port 22. In this case, you can create a file called `remote-exec` in your `~/.allinea` directory, which Linaro Forge uses instead.

> **Warning:** To prevent unauthorized users modifying sensitive files and directories, and introducing unsafe code into your environment, ensure that you assign only the minimum permissions that are required, and avoid group or world-writable permissions.

Linaro Forge looks for the script at `~/.allinea/remote-exec`, and it is executed as follows:

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

The script must start APPNAME on HOSTNAME with the arguments ARG1 ARG2 without further input (no password prompts). Standard output from APPNAME appears on the standard output of remote-exec.

For example:

```
SSH based remote-exec
```

This shows a remote-exec script using ssh running on a non-standard port.

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

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

9.4.1 Testing

When you have set up your remote-exec script, Linaro recommends that you test it from the command line. For example:

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

This returns the output of uname -n on , without prompting for a password.

If you are having trouble setting up remote-exec, contact Forge Support for assistance.

9.4.2 Windows

The functionality is also provided by the Windows remote client. However, there are two differences:

- The script is named remote-exec.cmd rather than remote-exec.
- The default implementation uses the plink.exe executable supplied with Linaro Forge.

9.5 Optional configuration

This section includes details of the Linaro Forge *Options* window (*Preferences* on Mac OS X), which allows you to edit the system, job submission, code viewer settings, and appearance.

9.5.1 Access the Linaro Forge Options or Preferences window

- To open the *Options* window, select *File* ▸ *Options*.
- To open the *Preferences* window (Mac OS X), select *File* ▸ *Preferences*.

9.5.2 System

This section provides details of the optional system settings that you can apply to Linaro Forge.

Warning: To prevent unauthorized users modifying sensitive files and directories, and introducing unsafe code into your environment, ensure that you assign only the minimum permissions that are required, and avoid group or world-writable permissions.

Name	Description
MPI Implementation	Allows you to identify which MPI implementation you are using.
Override default mpirun path	Allows you to override the path to the mpirun (or equivalent) command.
Select Debugger	Specifies to Linaro DDT the underlying debugger to use. Unless a specific debugger is required, leave this as *Automatic*. On Linux systems, Linaro Forge ships with the following versions of the GNU GDB debugger: GDB 12.1, GDB 13.1 and GDB 16.1 Linaro recommends the GDB 16.1 debugger for Linaro DDT. This recommended default is selected automatically when you select *Automatic (recommended)* in *Options ▸ System Settings*.
Create Root and Workers groups automatically	If this option is selected, Linaro DDT automatically creates a **Root** group for rank 0, and a **Workers** group for ranks `1-n` when you start a new MPI session.

Heterogeneous system support	Linaro 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 installation.
	• You must have a separate installation of Linaro DDT for each architecture. The architecture of the machine running the Linaro Forge GUI is called the host architecture.
	• You must create symbolic links from the host architecture installation of Linaro 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 /{installation-directory(i686)}/libexec/forge-backend \`
	`/{installation-directory(x86_64)}/bin/forge-backend.i686`

Enable CUDA software preemption	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 Linaro DDT when starting an MPI job. If you do not specify a file, Linaro DDT creates the default **Root** and **Workers** groups if the previous option is selected. **Note:** You can create a groups file while your program is running by right-clicking the *Process groups* panel and selecting *Save groups*.
Attach hosts file	When attaching, Linaro DDT fetches a list of processes for each of the hosts listed in this file. See *Attach to running programs* for more details.

9.5.3 Job submission

This section of the *Options* window allows you to configure to use a custom command, or submit your jobs to a queuing system. For more information on this, see *Integration with queuing systems*.

9.5.4 Code viewer settings

This section allows you to configure the appearance of the code viewer, which is used to display your source code while debugging.

Name	Description
Tab size	Sets the width of a tab character in the source code display. A width of 8 means that a tab character has the same width as 8 space characters.
Font name	The name of the font used to display your source code. Linaro recommends 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 Linaro Forge executes if you right-click in the code viewer and choose **Open file in external editor**. This command launches a graphical editor. If no editor is specified, Linaro Forge attempts to launch the default editor that is configured in your desktop environment.
Color Scheme	Color palette to use for the code viewer background, text, and syntax highlighting. Defined in Kate syntax definition format in the `resource/styles` directory of the 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 installation. These tools support F77, C, and C++, and analyze the source code of viewed source files to discover common errors, but they can cause heavy CPU usage on the system running the Linaro Forge user interface. You can disable this by clearing this option.

9.5.5 Appearance

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

Name	Description
Look and Feel	This determines the general graphical style of Linaro 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 (except the code viewer).

9.6 Convert OpenSSH private key on Windows with PuTTYgen

If you use an SSH-2 private key generated by OpenSSH or ssh.com standards, you must convert it to the PuTTY standard (`*.ppk`) using the PuTTYgen key generator before using it with Linaro Forge.

9.6.1 Before you begin

Download and run the Linaro Forge Remote client installation package from the Linaro Forge Downloads page.

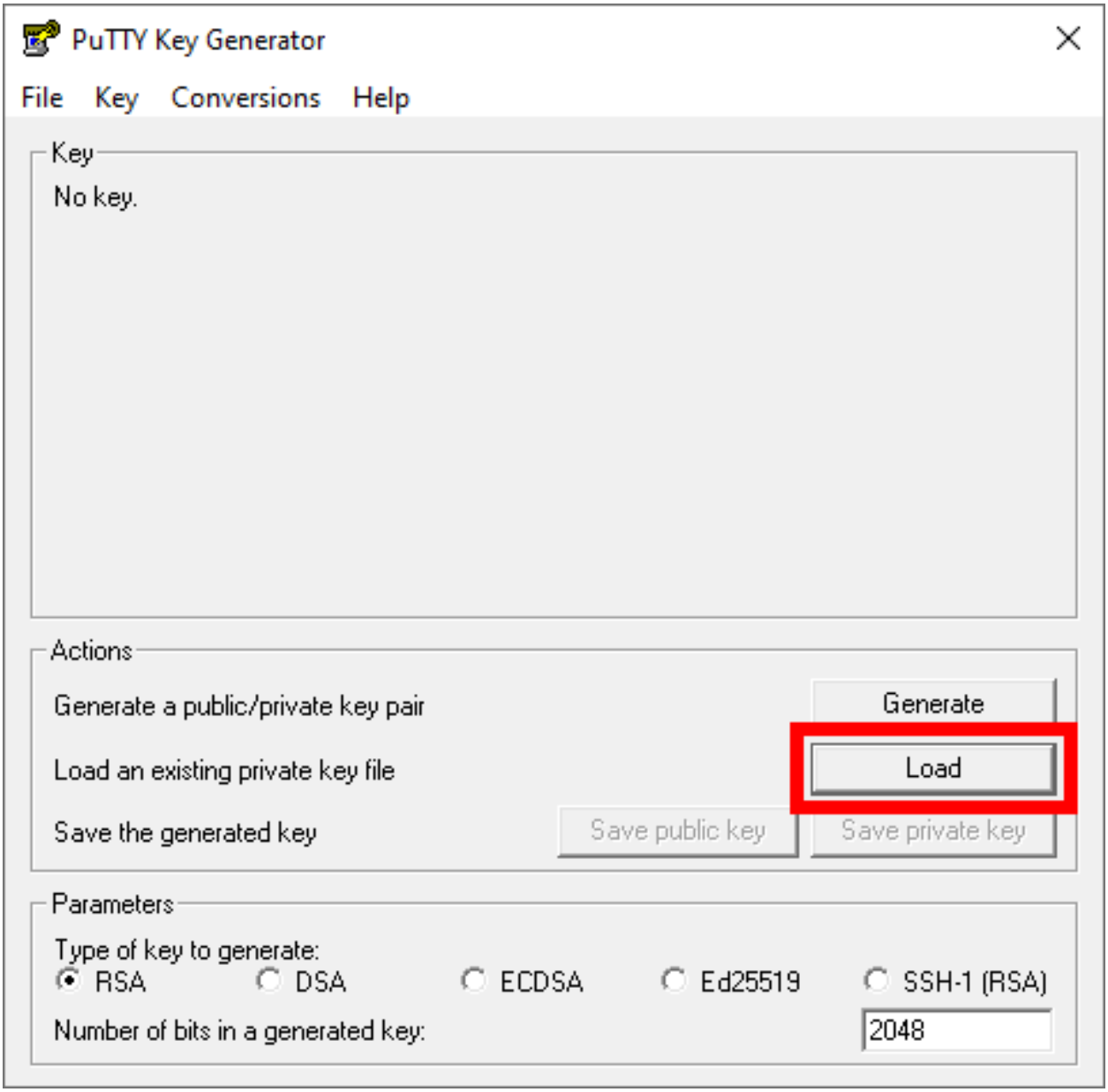
Note: The Linaro Forge Remote Client installation package includes the PuTTY software SSH client and PuTTYgen key generator.

9.6.2 About this task

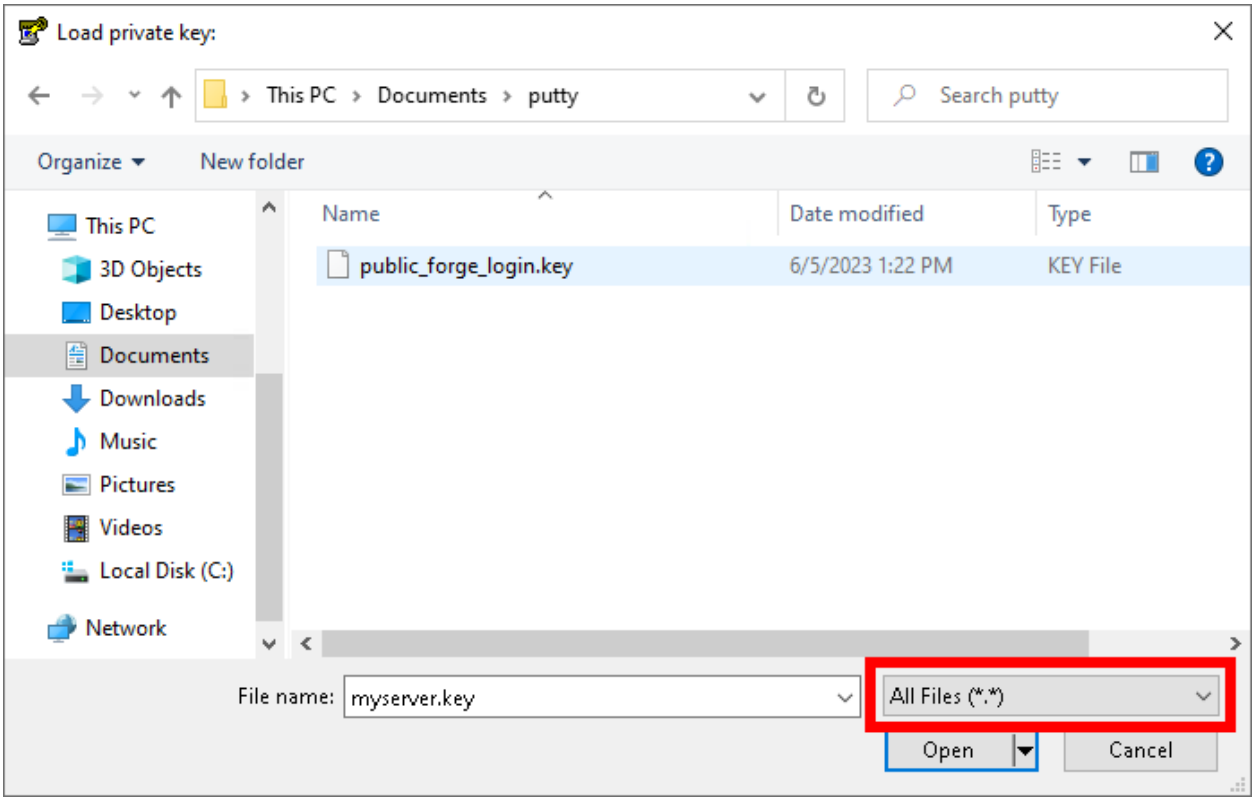
When *Connecting to a remote system* from a Windows hosts, Linaro Forge uses PuTTY for its SSH connection. If you normally use an OpenSSH key (e.g. within a Cygwin session) you must create a PuTTY version of your private key for use by Linaro Forge.

9.6.3 Procedure

1. Start PuTTYgen and click *Load* to browse to your existing key.

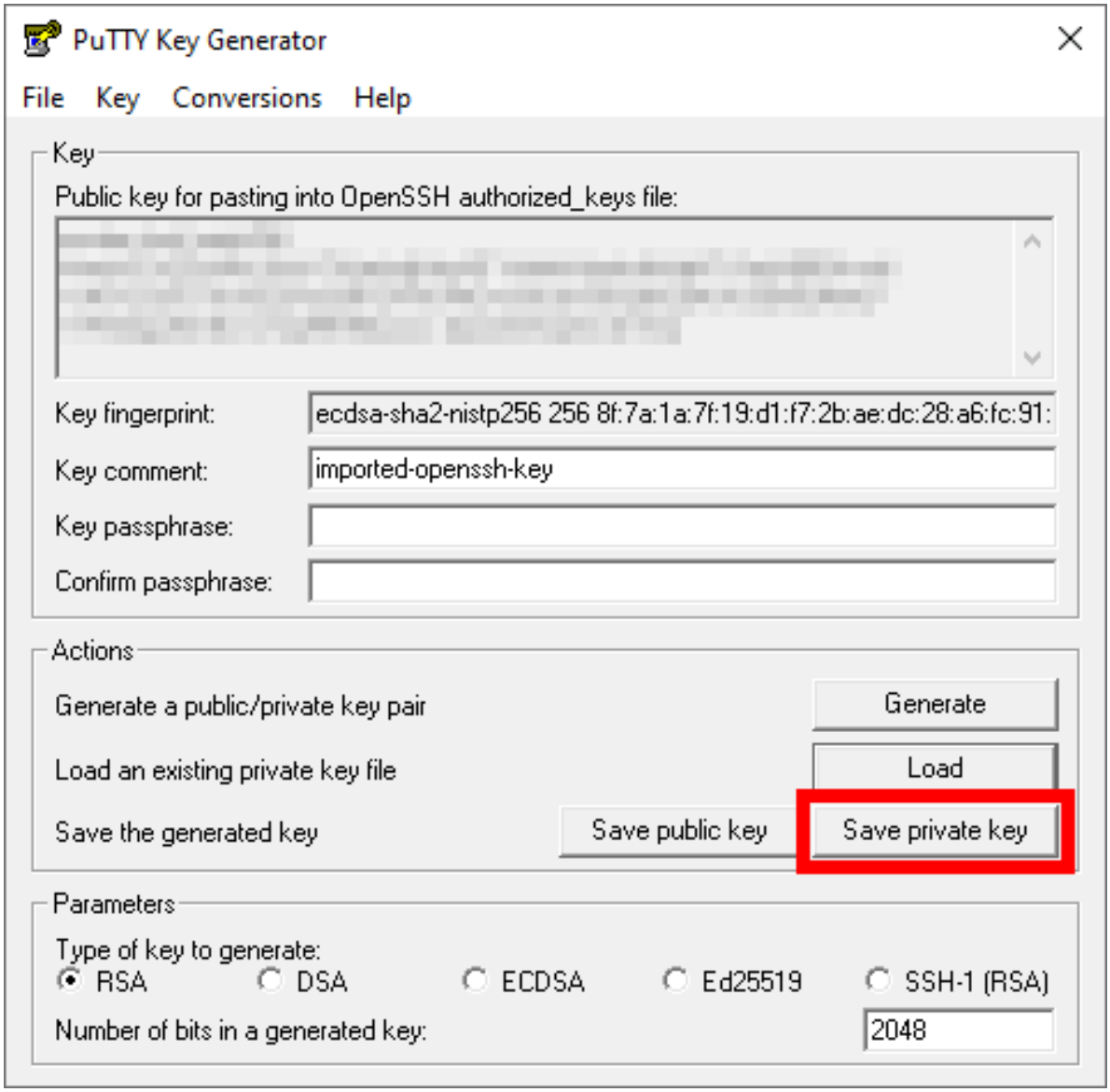


2. Locate and select your existing private key, and click *Open* to load it.



3. Confirm the PuTTYgen prompt.

4. Click *Save private key*.



9.6.4 Related information

- *Configure SSH on Windows with PuTTY*
- *Connecting to a remote system*

9.7 Configure SSH on Windows with PuTTY

When *Connecting to a remote system* from a Windows hosts, the Linaro Forge Remote Client uses PuTTY to SSH to remote systems. Most connections can be configured from the *Remote launch settings* dialog but if you need to configure advanced settings you can do so by preparing and saving a PuTTY session outside of Linaro Forge.

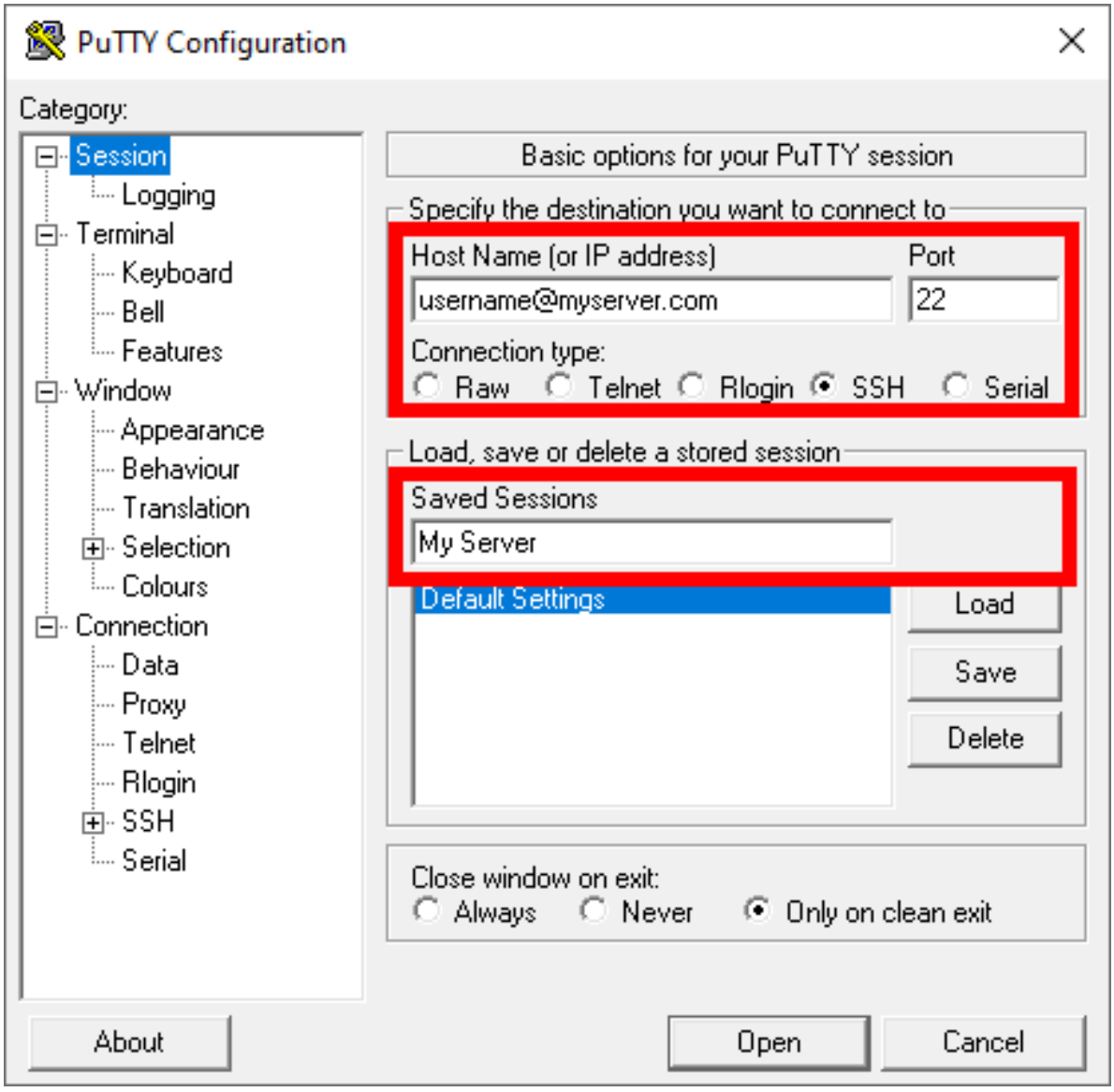
9.7.1 Before you begin

Download and run the Linaro Forge Remote client installation package from the Linaro Forge Downloads page.

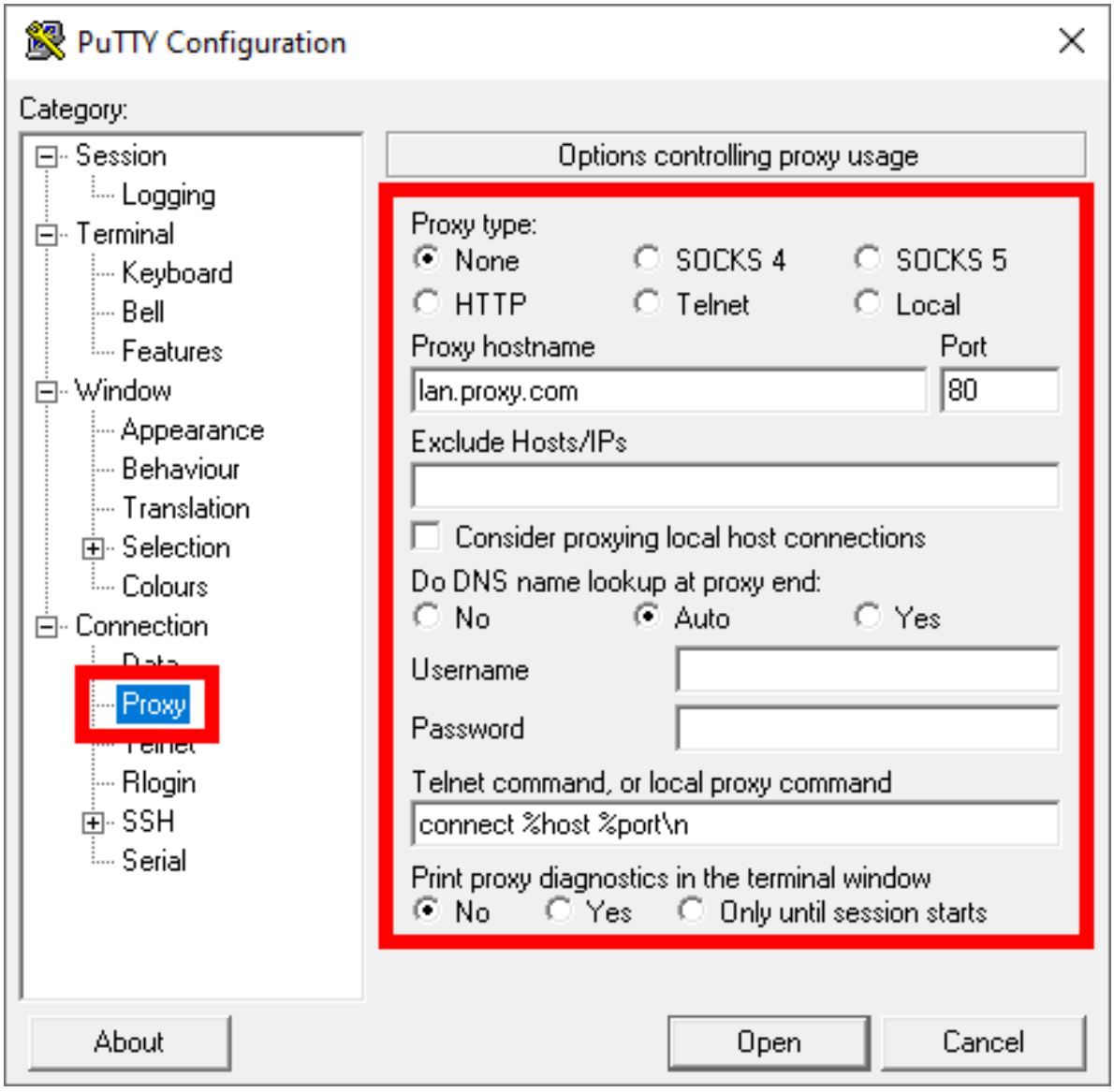
Note: The Linaro Forge Remote Client installation package includes the PuTTY software SSH client.

9.7.2 Procedure

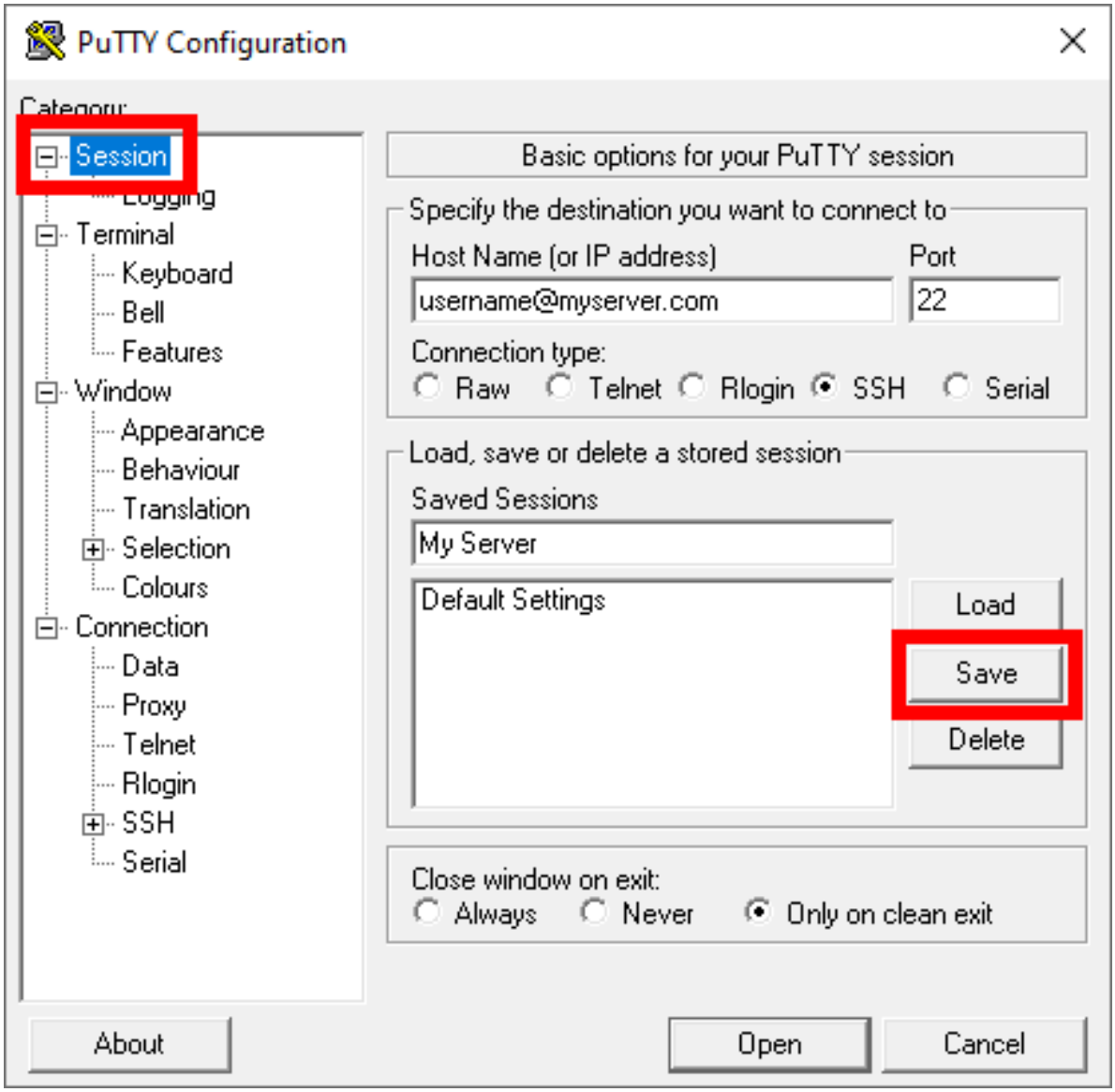
1. Start a new PuTTY session by opening the PuTTY executable file supplied in the archive folder of the Linaro Forge Remote Client installation package.



2. Enter your connection details, such as Proxy and Identity files.

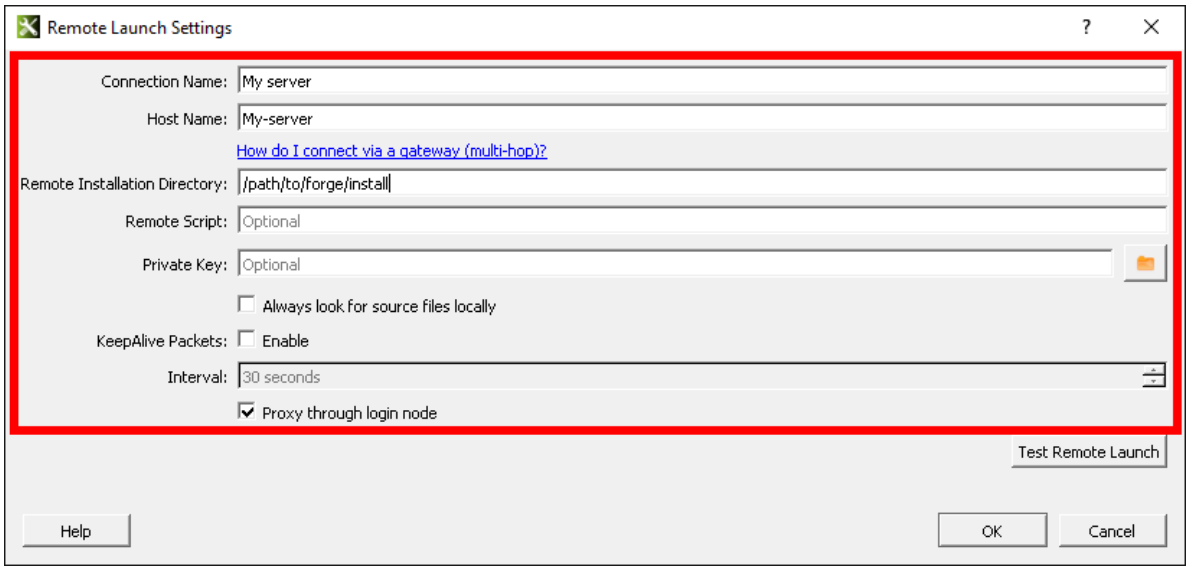


3. Configure and test your advanced settings.
4. Select *Session* and click *Save* to store your connection settings.



Note: Save your session to make it available on your Windows system through the registry. Linaro Forge Remote Client automatically finds this configuration.

5. Start Linaro Forge Remote Client and click *Remote Launch* ▸ *Configure*.
6. Add a new configuration and configure it using the PuTTY session you just saved as the host name.



7. Click *Test Remote Launch* to verify everything is working, and click *OK* when the setup is complete.

9.7.3 Related information

- *Convert OpenSSH private key on Windows with PuTTYgen*
- *Connecting to a remote system*

CHAPTER

TEN

QUEUE TEMPLATE SCRIPT SYNTAX

10.1 Queue template tags

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

Note: It is often sufficient to use AUTO_LAUNCH_TAG. See an example in **The template script** in *Template tutorial*.

Table 10.1: Queue template tags

Tag	Description	After Submission Example
AUTO_LAUNCH_TAG	This tag expands to the entire replacement for your command line.	forge-mpirun -np 4 myexample.bin
DDTPATH_TAG	The path to the installation	/opt/
WORKING_DIRECTORY_TAG	The working directory 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	\S
NUM_NODES_PLUS_ONE_TAG	Number of compute nodes + 1	\S
PROCS_PER_NODE_TAG	Processes per node	\S
PROCS_PER_NODE_PLUS_ONE_TAG	Processes per node + 1	\S
NUM_THREADS_TAG	Number of OpenMP threads per node (empty if OpenMP if off)	\S
OMP_NUM_THREADS_TAG	Number of OpenMP threads per node (empty if OpenMP is off)	\S
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.

10.2 Defining new tags

In addition to the pre-defined tags listed in *Queue template tags*, you can also define new tags in your template script, and you can specify their values in the GUI.

Tag definitions have the following format:

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

`key1` and `key2` are tag attribute names. `value1` and `value2` are the corresponding values.

The tag is replaced with the value specified in the GUI, as shown in this example.

```
#PBS -option EXAMPLE_TAG
```

These are the supported attributes:

Table 10.2: General queue template tag attributes

Attribute	Purpose	Examples
`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"`

Table 10.3: Queue template tag attributes when `type=text`

Attribute	Purpose	Examples
`mask`	Input masks `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"`

Table 10.4: Queue template tag attributes when `type=select`

Attribute	Purpose	Examples
`options`	Options to use, separated by the pipe (`|`) character.	`options="not_shared|shared"`
`checked`	Value of a `check` tag if checked.	`checked="enabled"`
`unchecked`	Value of a `check` tag if unchecked.	`unchecked="enabled"`
`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 (not included in tag value).	`suffix="s"`
`prefix`	Display only prefix (not included in tag value).	`prefix="$"`

Table 10.5: Queue template tag attributes when `type=file`

Attribute	Purpose	Examples
`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 selector to a certain file pattern.	`filter="Text files (*.txt)"`

10.2.1 Examples

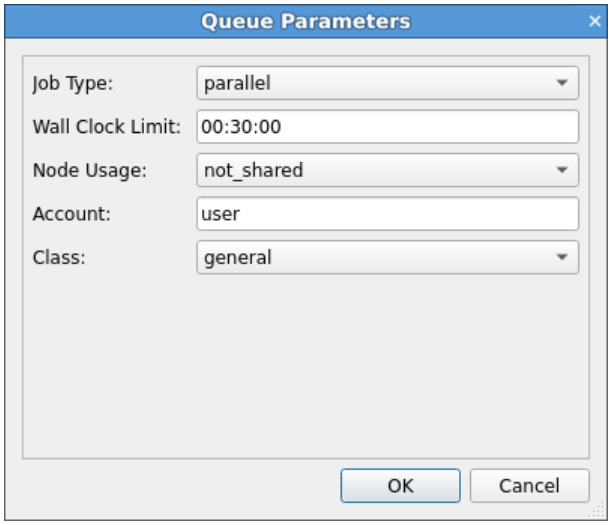
```
# JOB_TYPE_TAG: {type=select,options=parallel| \
  serial,label="Job Type",default=parallel}

# 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.

10.2.2 Specifying tag values

To specify values for these tags, click the *Edit Template Variables* button on the *Job Submission Options* page (see *Integration with queuing systems*) or the *Run* window.

A window displays which is similar to this:



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

10.3 Specifying default options

A queue template file can specify defaults for the options on the **Job Submission** page so that when a user selects the template file, these options are automatically completed.

Table 10.6: Default options

Name	Job Submission Setting	Example
`submit`	Submit command. The command might 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

10.3.1 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
```

10.4 Launching

Usually, your queue script ends in a line that starts with your target executable.

In a template file, this must be modified to run a command that also launches the Linaro Forge backend agents.

This section refers to some of the methods for doing this.

10.4.1 Using AUTO_LAUNCH_TAG

This is the easiest method to launch Linaro Forge backend agents, and caters for the majority of cases. Replace your command line with `AUTO_LAUNCH_TAG`. Linaro Forge replaces 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/linaro/forge/x.y.z/bin/forge-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
```

10.4.2 Using forge-mpirun

If you need more control than is available using `AUTO_LAUNCH_TAG`, Linaro 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 Linaro Forge is submitting to the queue itself.

Replace `mpirun` with `DDTPATH_TAG/bin/forge-mpirun`.

For example, if your script currently has the line:

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

Then (for illustration only) the equivalent that Linaro Forge must use would be:

```
DDTPATH_TAG/bin/forge-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 that they can be specified in the user interface rather than editing the queue script each time:

```
DDTPATH_TAG/bin/forge-mpirun -np NUM_PROCS_TAG \\
EXTRA_MPI_ARGUMENTS_TAG DDTPATH_TAG/libexec/forge-backend \\
DDT_DEBUGGER_ARGUMENTS_TAG PROGRAM_TAG PROGRAM_ARGUMENTS_TAG
```

See *Queue template tags* for more information on template tags.

10.4.3 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/forge-client DDT_DEBUGGER_ARGUMENTS_TAG \\
PROGRAM_TAG PROGRAM_ARGUMENTS_TAG
```

10.5 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`).

See `sample.qtf` for an explanation of the queue template system in the Linaro Forge installation directory, at `<installationdirectory>/templates`.

10.6 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 available in addition to the conventional quantifiers, range and exclusion operators.

Table 10.7: Regular expression characters

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`, you must enter `job_remove JOB_ID_TAG` as the cancel command.

CHAPTER

ELEVEN

WORKED EXAMPLES

This appendix contains tutorials on debugging and optimizing sample codes with Linaro Forge.

11.1 Linaro Forge tutorial with `mmult`

This tutorial describes how to get started with Linaro Forge. It walks you through setting up the Linaro Forge tools and shows you how to use them to debug, analyze and optimize some examples code.

11.1.1 Software requirements

To follow this tutorial, first ensure that:

- You will be working on a Linux platform. Use of the Windows or Mac OS X remote client to connect to a Linux host is not part of this tutorial (see *Connecting to a remote system* for more information).
- You have Linaro Forge installed. See Linaro Forge Downloads and *Installing Linaro Forge*.
- Your Linaro Forge installation is licensed. Copy your license file into the `licences` directory of your Linaro Forge installation.

  Note: If trialing the product, you can obtain a temporary free trial licence.

- A Message Passing Interface (MPI) environment is configured. If you are unsure if you have an MPI environment configured, ask your System Administrator.
- For the final "Next Steps" of *Optimize the code with Linaro MAP*, you will need a Basic Linear Algebra Subprogram (BLAS) library installed.

  Note: On Arm-based platforms, you can use the Arm® Performance Libraries that ship with the Arm® Compiler for Linux.

- You have loaded any environment modules needed for the tools you are using.
- This tutorial uses the `examples/mmult*` files in the Linaro Forge distribution. Since this tutorial involves editing some of these files, you may wish to work off copies in some working directory rather than editing the originals.

```
mkdir mmult-tutorial
cd mmult-tutorial
cp /path/to/forge/examples/mmult* .
cp /path/to/forge/examples/common.makefile .
```

- (Python example only) You have installed Python and its associated libraries:
  - Python ≥ 3.6 is installed.
  - MPI4Py ≥ 3.0.2, NumPy ≥ 1.16.4, and SciPy ≥ 1.3.0 modules are installed:

```
pip3 install numpy scipy mpi4py
```

11.1.2 The `mmult` Algorithm

The application in this tutorial performs the following calculation:

$$C = A \times B + C$$

`A`, `B`, and `C` are double-precision square matrices. The parallel algorithm employed has one *main* process (MPI rank 0) that splits this calculation up and distributes the work across several worker processes. The application performs the following tasks:

- The master process initializes matrices `A`, `B`, and `C`.
- The master process sends the entire matrix `B`, along with slices of `A` and `C`, to the worker processes.
- The master and worker processes perform the matrix multiplication function on the domain that has been given to the processes, and each process computes a slice of `C`.
- The master process retrieves all slices of `C` and puts the results into matrix `C`.
- The master process writes the results of `C` in a file.

Fig. 11.1 shows how all processes contribute in parallel to the creation of the result in matrix `C`.

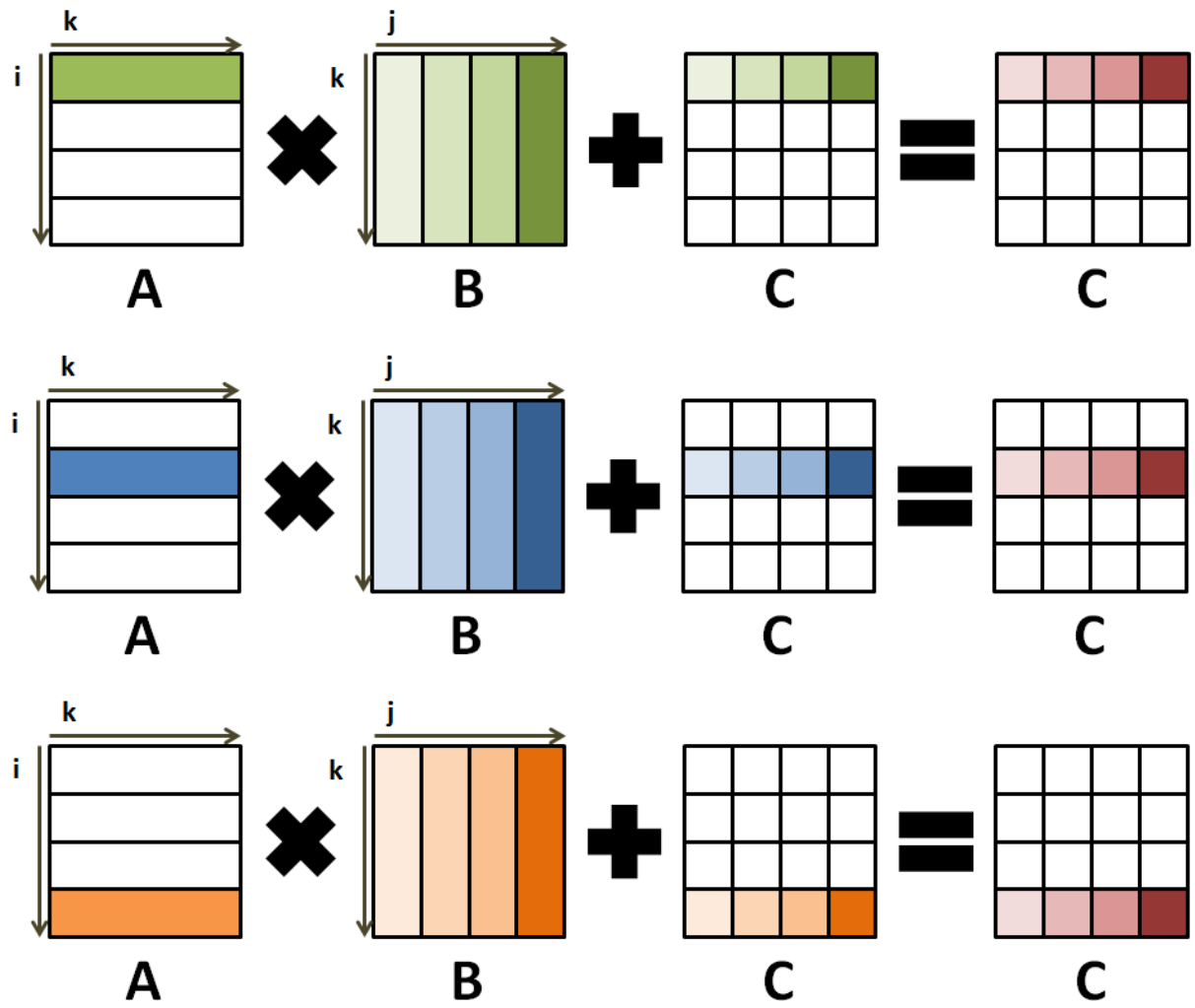


Fig. 11.1: Description of the parallel algorithm

C/C++, Fortran, and Python versions of this application can be found in the `examples/` directory of your Linaro Forge installation, see the files named `mmult*`.

11.1.3 Compile and Run `mmult`

11.1.3.1 Prerequisites

You must install all the necessary tools and copied the `examples/mmult.*` source files as described in *Software requirements*.

11.1.3.2 Procedure

1. Compile the code. To compile the C and Fortran versions of the application, run make on the `mmult.makefile` file.

```
make -f mmult.makefile
```

   For the Python version of the application, run make on the `mmult-py.makefile` to build the C/Fortran compute kernels.

```
make -f mmult_py.makefile
```

   The compiler command and compilation options can be set at the top of `mmult.makefile` and `mmult_py.makefile`.

2. Run the application. To run the application using eight processes, use:

   - For the C version:

```
mpirun -n 8 ./mmult_c
```

   - For the F90 version:

```
mpirun -n 8 ./mmult_f
```

   - For the Python version:

```
mpirun -n 8 python3 ./mmult.py
```

   Additional arguments can be added to change the size of the matrices, for example 512x512 in C:

```
mpirun -n 8 ./mmult_c 512
```

Note: By default, the Python version runs the C kernel. To run the F90 kernel, use:

```
mpirun -n 8 python3 ./mmult.py -k F90
```

The application crashes with a 'Segmentation fault'. This is expected behavior. Each example source file provided in this trials package has an intentional bug.

11.1.3.3 Next Steps

To fix the bug, you must debug the code. *Fix the bug with Linaro DDT* shows you how to debug the code using Linaro DDT, and then how to fix the bug.

11.1.4 Fix the bug with Linaro DDT

This topic describes how to identify and fix the bug in each of the `examples/mmult*` source files with Linaro DDT.

11.1.4.1 Before you begin

- You must install Linaro Forge as described in *Software requirements*.
- You must complete the instructions in *Compile and Run mmult*.

11.1.4.2 Procedure

1. Recompile the application with the `-g` debugging flag.

   To make debugging simpler, Linaro recommends that you compile without using compiler optimizations. To disable optimizations, add the `-O0` option to the `CFLAGS` variable in `mmult.makefile`.

   ```
   CFLAGS = -O0 -g
   ```

   Remove the initial executables with:

   ```
   make -f mmult.makefile clean
   ```

   And recompile with:

   ```
   make -f mmult.makefile
   ```

   Note: In Fortran, the compiler might display a warning. To display more information at runtime, compile using the `-fcheck=bounds` flag with GCC, or `-Mbounds` using the Arm® Compiler for Linux.

2. Debug the application with **`ddt mpirun`** using *express launch*. For example:

   ```
   ddt mpirun -n 8 ./mmult_c
   ddt mpirun -n 8 ./mmult_f
   ddt mpirun -n 8 python3 ./mmult.py
   ```

   If your MPI environment does not support express launch (use `ddt --list-mpis` to list the known MPIs and which are supported by express launch), run the **`ddt`** command:

   ```
   ddt -n 8 ./mmult_c
   ddt -n 8 ./mmult_f
   ddt -n 8 python3 ./mmult.py
   ```

   Note: `-n 8` tells the debugger to debug using 8 processes. However, the application crashes at any scale.

   If you run the application on a remote system, see *Connecting to a remote system* for instructions.

   Once Linaro DDT starts it will display the *Run* dialog (Fig. 11.2).

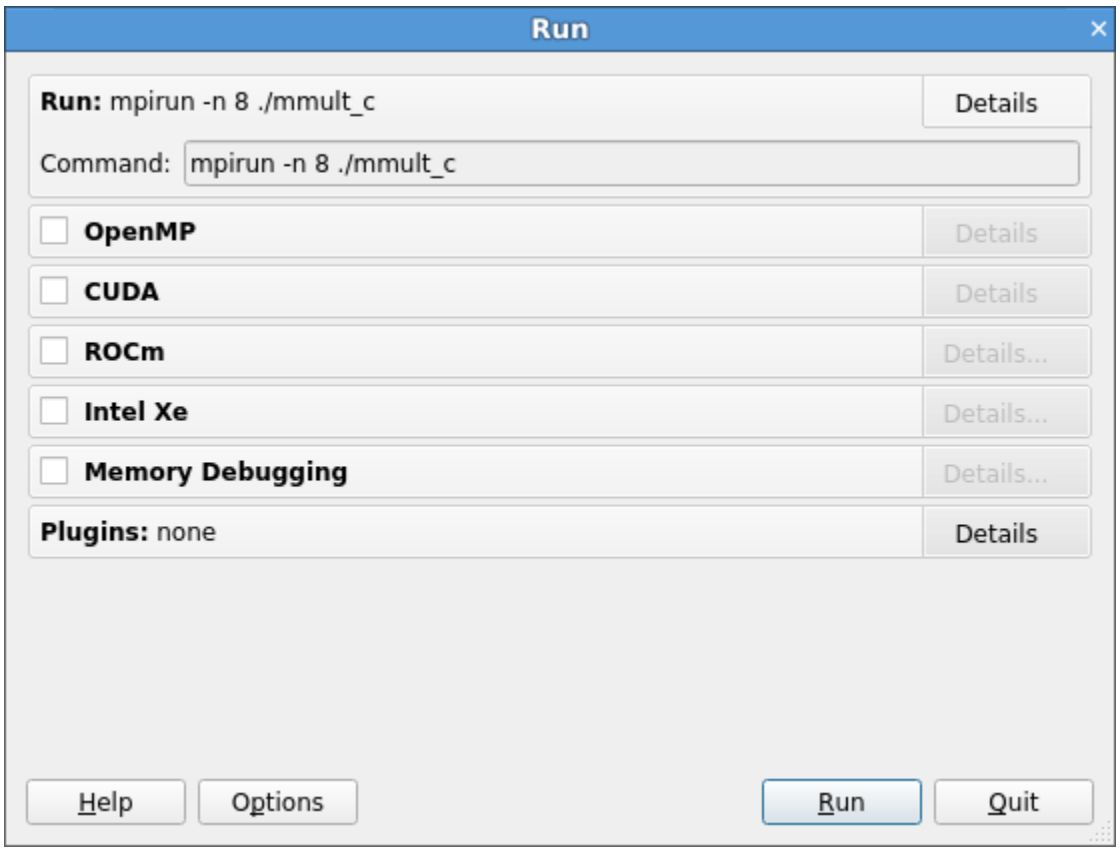


Fig. 11.2: Linaro DDT run dialog

3. To continue, click *Run*.

   This displays the main debugger window. See *DDT user interface*.

Note: At this stage, the C and F90 versions display the source code of the application.

If Linaro DDT does not display the source code, recompile the source file and ensure that the -g debugging flag is included on the compile line. For complete instructions on how to compile the examples, see *Compile and Run mmult*.

If you still experience a problem, contact Forge Support. For the Python version, no source code displays when the debugger attaches. This is expected when running in the Python interpreter. The C or F90 source code of the kernel displays when the application crashes, if you compile the kernels with -g.

---

1. To visualize where the application crashes, click *Play*  (in the top-left corner):

   The debugger stops where the application crashes (Fig. 11.3).

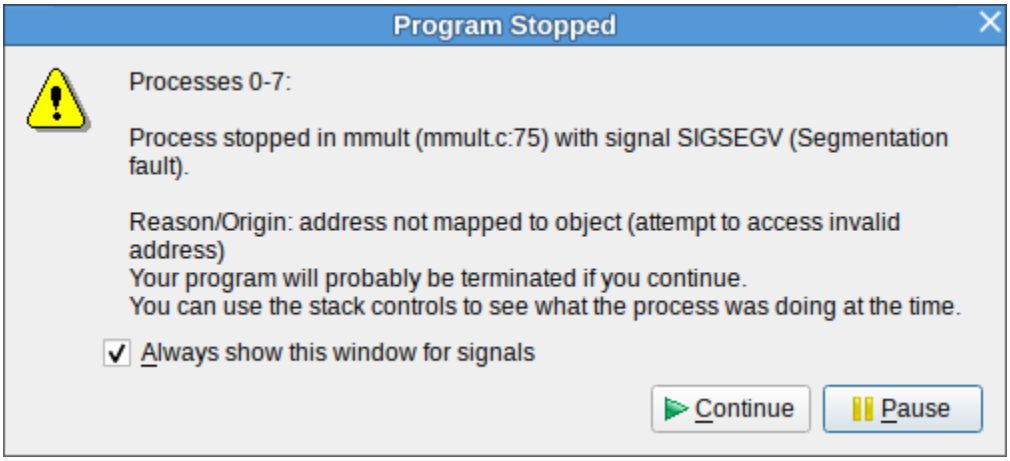


Fig. 11.3: Linaro DDT segmentation fault message

2. Click *Pause*.

   The source code viewer highlights the line of code where the crash occurs. In C:

   ```
   res += A[i*sz+k]*B[k*sz*j];
   ```

   In F90:

   ```
   res=A(k,i)*B(j,k+res)
   ```

   This problem is caused by an error in the expression to compute the index of array B, and this results in an out of bound memory access. The bug can be fixed with:

   In C:

   ```
   res += A[i*sz+k]*B[k*sz+j];
   ```

   In F90:

   ```
   res=A(k,i)*B(j,k)+res
   ```

   Note: To automatically detect an out of bound memory access with Linaro DDT, select the *Memory debugging* box in the *Run* window. Navigate to memory debugging *Details* and enable *Add guard pages to detect out of bound heap accesses*.

   For more information, see *Memory debugging*.

   ---

3. Save the source file: select *File* ▸ *Save Source File*.
4. Recompile the source file: select *File* ▸ *Build*.

   By default, *Build* runs **make** in the current directory.

   Note: To change the Build options, select *File* ▸ *Configure Build* and set the configuration settings.

   ---

5. To run the executable with the fix, select *File* ▸ *Restart session*.

   You are prompted to restart the application. Click *Yes*.

6. Play the application in the debugger again.

   The application runs without any issues until every process in the program has terminated, and outputs the following when running 8 processes:

```
0: Size of the matrices: 64x64
3: Receiving matrices...
6: Receiving matrices...
2: Receiving matrices...
4: Receiving matrices...
7: Receiving matrices...
1: Receiving matrices...
5: Receiving matrices...
0: Initializing matrices...
0: Sending matrices...
1: Processing...
2: Processing...
3: Processing...
4: Processing...
5: Processing...
6: Processing...
7: Processing...
0: Processing...
1: Sending result matrix...
3: Sending result matrix...
5: Sending result matrix...
7: Sending result matrix...
2: Sending result matrix...
4: Sending result matrix...
6: Sending result matrix...
0: Receiving result matrix...
0: Writing results...
0: Done.
```

   When fixed, the application writes the results in the working directory to a file called `res_C.mat`, `res_F90.mat`, or `res_Py.mat` (depending on the version you used). Ensure that your working directory is writable.

Note: To run Linaro DDT in non-interactive mode, use `ddt --offline ...`:

The debugger runs in the background of the application and outputs a debugging report. Open this report in your browser of choice:

```
firefox mmult_8p_1n_YYYY-MM-DD_HH-MM.html
```

`YYYY-MM-DD_HH-MM` corresponds to a timestamp of the report creation date. For more information, see *Offline debugging*.

11.1.5 Analyze the behavior with Linaro Performance Reports

Describes how to analyze the behavior of the **mmult** example code, and how to check if there are any performance issues using Linaro Performance Reports.

11.1.5.1 Prerequisites

- You must install all the necessary tools as described in *Software requirements*.
- You must complete the instructions in *Compile and Run mmult* and *Fix the bug with Linaro DDT*.

11.1.5.2 Procedure

1. Run the application with eight processes on a large test case, for example 3072x3072 matrices:

   ```
   perf-report mpirun -n 8 ./mmult_c 3072
   ```

   or

   ```
   perf-report mpirun -n 8 ./mmult_f 3072
   ```

   or

   ```
   perf-report mpirun -n 8 python3 ./mmult.py -s 3072
   ```

   If your MPI environment does not support express launch, run the following command instead:

   ```
   perf-report -n 8 ./mmult_c 3072
   ```

   or

   ```
   perf-report -n 8 python3 ./mmult.py -s 3072
   ```

   When the execution terminates, Linaro Performance Reports creates two files:

   - `mmult_8p_1n_YYYY-MM-DD_HH-MM.txt`
   - `mmult_8p_1n_YYYY-MM-DD_HH-MM.html`

   `YYYY-MM-DD_HH-MM` corresponds to a timestamp of the report creation date. The two files contain the same data, in two different formats.

2. To visualize the results, open the HTML file in your web browser (either locally or remotely if you have X forwarding enabled). For example, to use Firefox:

   ```
   firefox mmult_8p_1n_YYYY-MM-DD_HH-MM.html
   ```

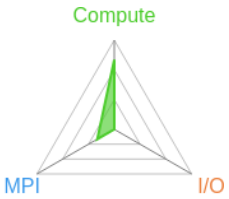
   Alternatively open the `.txt` file in any fixed-width code editor:

   ```
   vim mmult_8p_1n_YYYY-MM-DD_HH-MM.txt
   ```

   The report (Fig. 11.4) shows different sections:

linaro PERFORMANCE REPORTS

Command:	mpirun -n 8 ./mmult_c 3072
Resources:	1 node (8 physical, 16 logical cores per node)
Memory:	30 GiB per node
Tasks:	8 processes
Machine:	ip-10-252-24-58.eu-west-2.compute.internal
Architecture:	x86_64
CPU Family:	cascadelake-x
Start time:	Mon Apr 29 13:20:49 2024
Total time:	98 seconds (about 2 minutes)
Full path:	.



Summary: mmult_c is Compute-bound in this configuration

Compute	78.0%	76.3s	Time spent running application code. High values are usually good. This is **high**; check the CPU performance section for advice
MPI	21.9%	21.5s	Time spent in MPI calls. High values are usually bad. This is **low**; this code may benefit from a higher process count
I/O	<0.1%	0.0s	Time spent in filesystem I/O. High values are usually bad. This is **very low**; however single-process I/O may cause MPI wait times

This application run was Compute-bound (based on main thread activity). A breakdown of this time and advice for investigating further is in the CPU section below.

As little time is spent in MPI calls, this code may also benefit from running at larger scales.

CPU

A breakdown of the 78.0% (76.3s) CPU time:

Scalar numeric ops	11.4%	8.7s
Vector numeric ops	0.0%	0.0s
Memory accesses	88.6%	67.6s

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

No time is spent in vectorized instructions. Check the compiler's vectorization advice to see why key loops could not be vectorized.

MPI

A breakdown of the 21.9% (21.5s) MPI time:

Time in collective calls	33.1%	7.1s
Time in point-to-point calls	66.9%	14.3s
Effective process collective rate	0.00 bytes/s	
Effective process point-to-point rate	24.7 MB/s	

I/O

A breakdown of the <0.1% (0.0s) I/O time:

Time in reads	0.0%	0.0s
Time in writes	100.0%	0.0s
Effective process read rate	0.00 bytes/s	
Effective process write rate	1.15 GB/s	

Most of the time is spent in write operations with a high effective transfer rate. It may be possible to achieve faster effective transfer rates using asynchronous file operations.

Threads

A breakdown of how multiple threads were used:

Computation	100.0%	9776.3s
Synchronization	0.0%	0.0s
Physical core utilization		66.7%
System load		69.8%

Physical core utilization is low. Try increasing the number of threads or processes to improve performance.

Memory

Per-process memory usage may also affect scaling:

Mean process memory usage	136 MiB	
Peak process memory usage	253 MiB	
Peak node memory usage	50.0%	

There is significant variation between peak and mean memory usage. This may be a sign of workload imbalance or a memory leak.

The peak node memory usage is modest. Running with fewer MPI processes and more data on each process may be more efficient.

Energy

A breakdown of how energy was used:

CPU	not supported %
System	not supported %
Mean node power	not supported W
Peak node power	0.00 W

Energy metrics are not available on this system.

CPU metrics: Error reading /sys/class/powercap/intel-rapl:0/energy_uj: Permission denied

Fig. 11.4: Linaro Performance Reports HTML document

Application Details (top)
Describes the system settings (including the number of physical and logical cores), the job configuration (including the number of processes and number of nodes) and the execution time.

Summary (middle)
The *Summary* section shows the amount of time spent in computations (CPU), communications (MPI), and IO.

Breakdown sections (bottom)
Shows a breakdown of:

- The *CPU Breakdown* (`x86_64` only), *CPU Metrics* (`aarch64` only), *MPI*, and *IO* time.
- How multiple *threads* were used.
- How much *memory* was used.

The details of the report will be different and relevant to your system configuration, but the report should indicate that the application is CPU bound.

The *CPU breakdown* section (`x86_64` only) gives more information about the type of instruction run (Fig. 11.5):

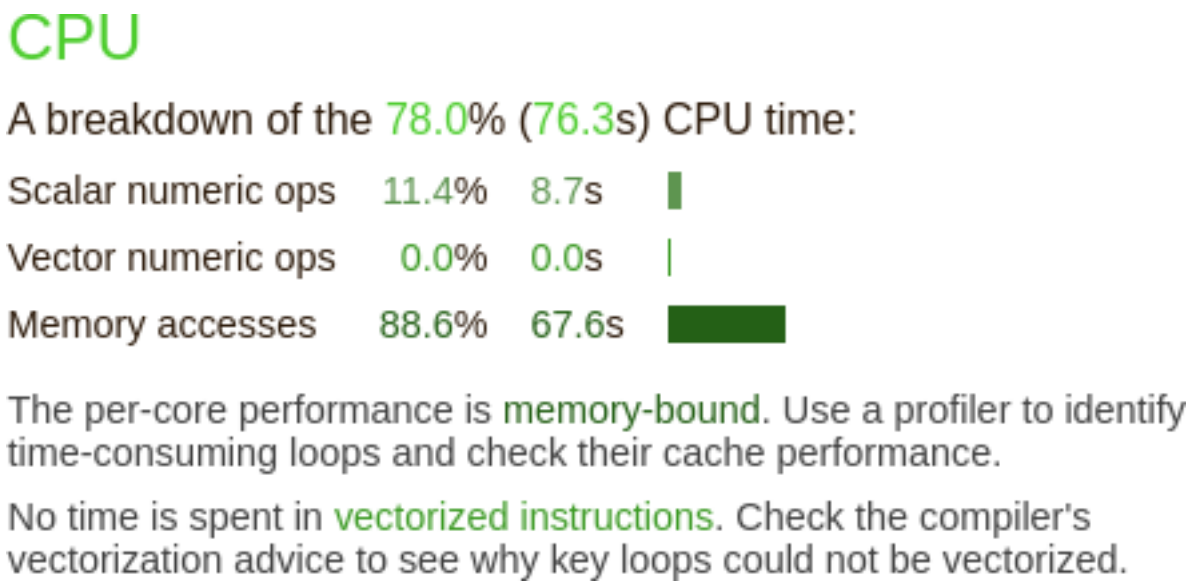


Fig. 11.5: Linaro Performance Reports CPU metrics without compiler optimizations

The compiler does not perform vectorization. As the report suggests, you can change the behavior by changing your compiler options.

Note: On non-x86 architectures, the CPU metrics are *different*. Instead, the tool reports the following metrics:

- Cycles per instructions
- Amount of L2 (or L3) cache accesses
- Amount of processor back-end/front-end stalls

Keep these numbers low for better performance.

---

3. To enable the `-Ofast` compiler optimization (including vectorization), edit `mmult.makefile`:

```
CFLAGS = -Ofast -g
```

4. Remove the previous executable, recompile, and run Linaro Performance Reports again:

```
make -f mmult.makefile clean
make -f mmult.makefile
perf-report mpirun -n 8 ./mmult_c 3072
```

The new report shows a performance improvement because the code has been vectorized by the compiler (Fig. 11.6).

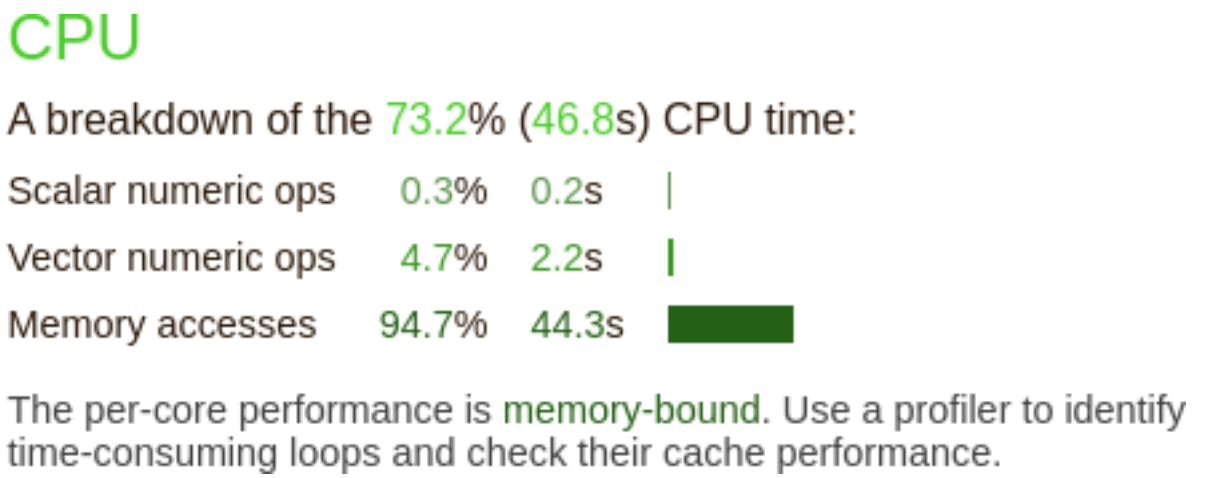


Fig. 11.6: Linaro Performance Reports CPU metrics with compiler optimizations

Tip: You should always profile binaries compiled using the same optimization flags you use in production i.e. `-O2` or `-Ofast`. For best results, use `-g` flag (or `-g1` when you need to minimize the amount of debug information, such as when profiling the `-g` compiled binary triggers out-of-memory errors).

See *Prepare a program for profiling* for more on recommended compilation flags for use with Linaro MAP and Linaro Performance Reports.

11.1.6 Optimize the code with Linaro MAP

Describes how to profile and optimize the **mmult** example code using Linaro Performance Reports and Linaro MAP. Linaro Performance Reports can identify high memory accesses, and Linaro MAP can identify the time-consuming loops in the example code.

11.1.6.1 Prerequisites

- You must install all the necessary tools as described in *Software requirements*.
- You must complete the instructions in *Compile and Run mmult*, *Fix the bug with Linaro DDT*, and *Analyze the behavior with Linaro Performance Reports*
- Ensure the code has been compiled with the `-g` debugging flag.

11.1.6.2 Procedure

1. To profile the code with multiple processes and the 3072x3072 test case, use `map --profile mpirun`. For example:

   ```
   map --profile mpirun -n 8 ./mmult_c 3072
   ```

   or

   ```
   map --profile mpirun -n 8 ./mmult_f 3072
   ```

   or

   ```
   map --profile mpirun -n 8 python3 ./mmult.py -s 3072
   ```

   If express launch is not supported for your MPI environment, run `map --profile`:

   ```
   map --profile -n 8 ./mmult_c 3072
   ```

   or

   ```
   map --profile -n 8 python3 ./mmult.py -s 3072
   ```

   The `--profile` option runs the profiler in non-interactive mode. When the execution terminates, a profile file (`.map`) is created by Linaro MAP:

   ```
   mmult_8p_1n_YYYY-MM-DD_HH-MM.map
   ```

   `YYYY-MM-DD_HH-MM` corresponds to a timestamp of the report creation date.

2. To view the results, run the interactive mode:

   ```
   map mmult_8p_1n_YYYY-MM-DD_HH-MM.map
   ```

   Linaro MAP starts and displays the main profiler window. See *MAP user interface*.

   Depending on your system configuration, the details might vary in your results. The profiler indicates that most of the time is spent in one line of the `mmult` function (or when using the Python version, the corresponding calls in the C of F90 version):

   In C:

```
res += A[i*sz+k]*B[k*sz+j];
```

In F90:

```
res=A(k,i)*B(j,k)+res
```

Select this line of code. The *CPU breakdown window* appears on the right and shows the following results (Fig. 11.7):

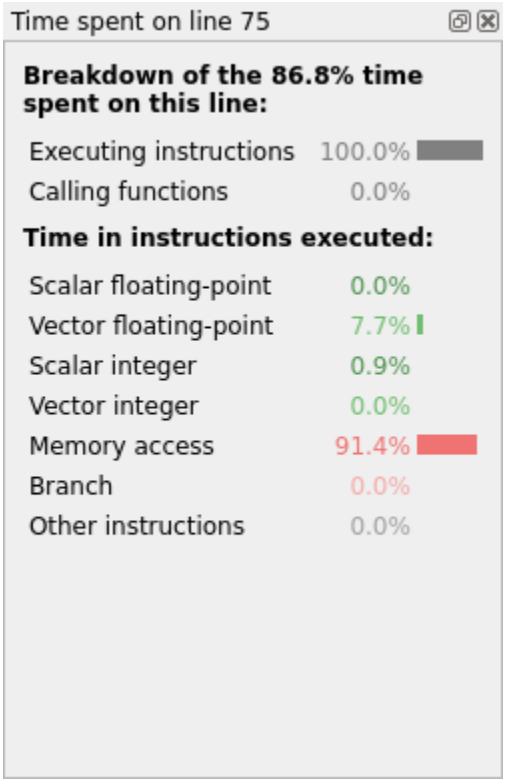


Fig. 11.7: Linaro MAP line breakdown without optimized memory accesses

The results indicate inefficient memory accesses. The loop nest performs strided accesses to array B. In addition to this, a dependency on intermediate results prevents the compiler vectorizing properly.

Note: On non-x86 architectures, the *CPU breakdown* is not available. To visualize the high amount of cycles per instructions, L2 (or L3) cache misses, and stalled back-end cycles when the `mmult` function is being executed, instead use the *CPU instructions metric graphs* by selecting *Metrics* ▸ *Preset: CPU instructions* from the menu.

---

3. In C, replace the following code:

```
for(int i=0; i<sz/nslices; i++)
{
    for(int j=0; j<sz; j++)
    {
        double res = 0.0;
```

(continues on next page)

(continued from previous page)

```
        for(int k=0; k<sz; k++)
        {
            res += A[i*sz+k]*B[k*sz+j];
        }
        C[i*sz+j] += res;
    }
}
```

with:

```
for(int i=0; i<sz/nslices; i++)
{
    for(int k=0; k<sz; k++)
    {
        for(int j=0; j<sz; j++)
        {
            C[i*sz+j] += A[i*sz+k]*B[k*sz+j];
        }
    }
}
```

and in Fortran replace:

```
do i=0,sz/nslices-1
do j=0,sz-1
    res=0.0
    do k=0,sz-1
    res=A(k,i)*B(j,k)+res
    end do
    C(j,i)=res+C(j,i)
end do
end do
```

with:

```
do i=0,sz/nslices-1
do k=0,sz-1
do j=0,sz-1
    C(j,i)=A(k,i)*B(j,k)+C(j,i)
end do
end do
end do
```

4. Remove the previous executable, recompile, and run Linaro MAP again:

```
make -f mmult.makefile clean
make -f mmult.makefile
map --profile -n 8 ./mmult 3072
```

The profiling results show significant performance improvement because of the optimization (Fig. 11.8).

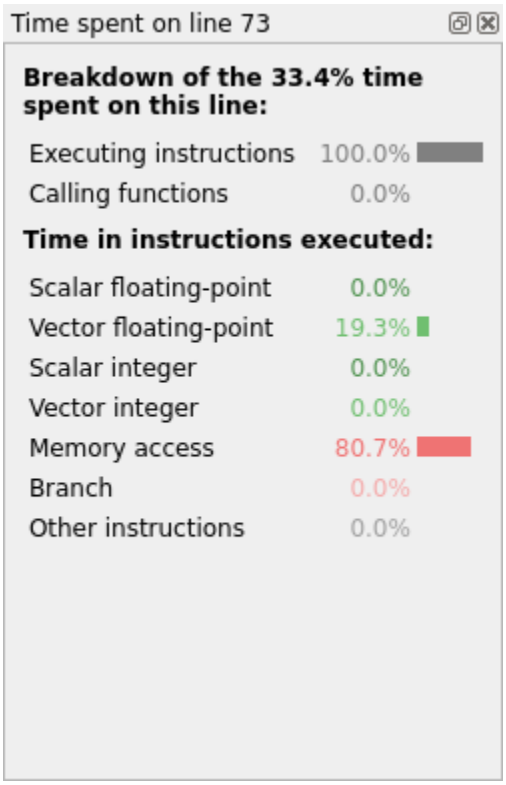


Fig. 11.8: Linaro MAP line breakdown with optimized memory accesses

11.1.6.3 Next Steps

To go further and use an optimized version of the matrix multiplication:

- In the C version, call CBLAS instead of `mmult`:

```
#include <cblas.h>
...
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, sz/nproc, sz, sz, 1.0, mat_a,
sz, mat_b, sz, 1.0, mat_c, sz);
```

- In the F90 version, call BLAS instead of `mmult`:

```
call DGEMM('N','N', sz, sz/nproc, sz, 1.0D0, &
           mat_b, sz, &
           mat_a, sz, 1.0D0, &
           mat_c, sz)
```

Make sure you edit `mmult.makefile` to include the BLAS header and link to your BLAS library, for instance with OpenBLAS:

```
CFLAGS = -Ofast -g -I/opt/openblas/include
LFLAGS = -L/opt/openblas/lib -lopenblas
```

In the Python version, the call to SciPy's DGEMM can be run with the following command:

```
mpirun -n 8 python3 ./mmult.py -k Py -s 3072
```

11.2 Linaro MAP tutorial with `vectorization`

This tutorial describes how to use *Compiler remarks* to optimize the performance of the example code.

11.2.1 Software requirements

To follow this tutorial, first ensure that:

- You will be working on a Linux platform. Use of the Windows or Mac OS X remote client to connect to a Linux host is not part of this tutorial (see *Connecting to a remote system* for more information).
- You have Linaro Forge installed. See Linaro Forge Downloads and *Installing Linaro Forge*.
- Your Linaro Forge installation is licensed. Copy your license file into the `licences` directory of your Linaro Forge installation.

  Note: If trialing the product, you can obtain a temporary free trial licence.

- Either access to the Cray Compiling Environment 12 and later, or Arm Compiler for Linux. This worked example demonstrates using *Compiler remarks* to optimize the runtime of a program. Other compilers may make different decisions when optimizing code, making this tutorial difficult to follow. Results may also vary depending on the version of your compiler and the system you are profiling on.
- You have loaded any environment modules needed for the tools you are using.
- This tutorial uses the `examples/vectorization*` files in the Linaro Forge distribution. Since this tutorial involves editing some of these files, you may wish to work off copies in some working directory rather than editing the originals.

```
mkdir vectorization-tutorial
cd vectorization-tutorial
cp /path/to/forge/examples/vectorization* .
cp /path/to/forge/examples/common.makefile .
```

11.2.2 Compile and Profile `vectorization`

11.2.2.1 Prerequisites

You must install all the necessary tools and copied the `examples/vectorization.*` source files as described in *Software requirements*.

11.2.2.2 Procedure

1. Compile the code. To compile the application, run make on the `vectorization.makefile` file.

```
make -f vectorization.makefile
```

> **Note:** The makefile auto-detects the compiler set with the `CC` environment variable to append the compilation flag necessary for generating a compiler report.
>
> The compilation flags for the compilers supported by the compiler remarks feature are listed in *Compiler remarks*.

2. Profile the application with Linaro MAP:

```
map ./vectorization
```

3. Ensure compiler remarks are enabled by using the *Compiler remarks* ▸ *Enabled* menu option. Focus on vectorization attempts by unchecking the menu options for other optimization passes, leaving *Compiler remarks* ▸ *loop-vectorize* checked.

Notice the three failed vectorization attempts annotating each computational loop in the program.

11.2.2.3 Next Steps

Optimize the code with vectorization shows you how to optimize the performance of the program by introducing vectorization.

11.2.3 Optimize the code with vectorization

Describes how to iteratively optimize the **`vectorization`** example code with vectorization, and profile it with Linaro MAP.

11.2.3.1 Prerequisites

- You must install all the necessary tools as described in *Software requirements*.

11.2.3.2 Procedure

1. Begin with the first failed vectorization attempt. According to the compiler remark annotation in Fig. 11.9, the compiler is unable to identify the array bounds so vectorization was not possible.

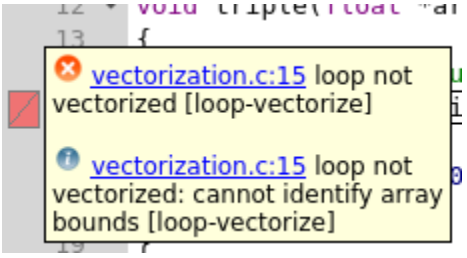


Fig. 11.9: Compiler remark for a failed vectorization attempt due to unknown array bounds

Assist the compiler by clarifying the bounds:

```
for (int i = 0; i < num/2; i++)
{
    arr2[2*i] = 3.0f * arr1[i];
    arr2[2*i+1] = 3.0f * arr1[i];
}
```

2. Compile the application by running make, and profile with Linaro MAP:

```
make -f vectorization.makefile
map ./vectorization
```

Notice in Fig. 11.10 that the vectorization attempt is now successful, and that the runtime of the program has been reduced.

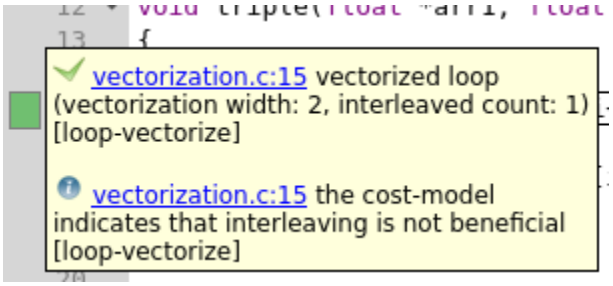


Fig. 11.10: Compiler remark for a successful vectorization attempt after providing array bounds

3. Next focus on the failed vectorization attempt at the nested loop. The compiler has deemed vectorization to be non-beneficial due to the non-sequential memory accesses in this loop (see Fig. 11.11).

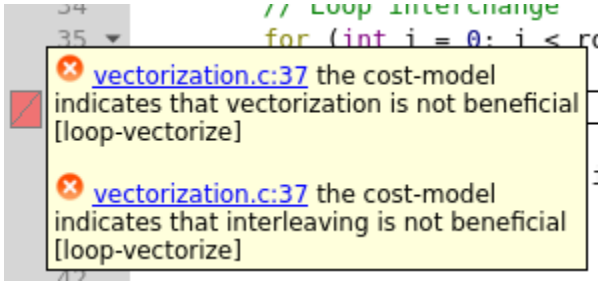


Fig. 11.11: Compiler remark for a failed vectorization attempt due to non-sequential memory accesses

Optimize the performance of this loop by interchanging the loop conditions.

```
for (int j = 0; j < cols; j++)
{
    for (int i = 0; i < rows; i++)
    {
        z[j * cols + i] = x[j * cols + i] - y[j * cols + i];
    }
}
```

Compile and profile the application again. Notice that the vectorization attempt is now successful in Fig. 11.12, and that the runtime of the program has been reduced.

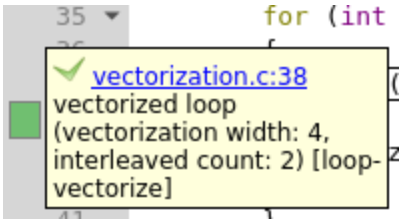


Fig. 11.12: Compiler remark for a successful vectorization attempt after interchanging loop conditions

4. Now focus on the final failed vectorization attempt (Fig. 11.13). This loop contains a loop-carried dependency, i.e. the result of one iteration is dependent on another.

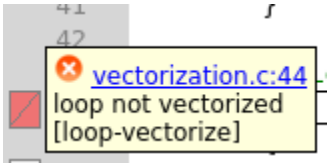


Fig. 11.13: Compiler remark for a failed vectorization attempt due to a loop-carried dependency

It cannot be fully vectorized, but it can be distributed into vectorizable and unvectorizable parts as follows:

```
for (int i = 1; i < size; i++)
{
    x[i] = x[i - 1] * y[i - 1] - z[i];
    y[i] = 2.0f * y[i - 1];
}

for (int i = 1; i < size; i++)
{
    z[i] = x[i] + y[i];
}
```

Compile and profile the application again. Although the compiler has vectorized the second distributed loop in Fig. 11.14, depending on your system the runtime of the program may have increased due to the introduction of another loop.

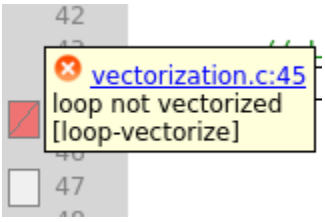


Fig. 11.14: Compiler remarks showing both a failed and a successful vectorization attempt after distributing the loop

This shows that acting upon all failed compiler optimization remarks may not always be beneficial. Revert the last change to optimize the runtime of the program if that is the case.

11.3 Linaro MAP thread affinity tutorial

This tutorial describes how to use the *Thread affinity advisor* to optimize the performance of the `wave_openmp` example code when run under SLURM.

11.3.1 Software requirements

To follow this tutorial, first ensure that:

- You will be working on a Linux platform. Use of the Windows or Mac OS X remote client to connect to a Linux host is not part of this tutorial (see *Connecting to a remote system* for more information).
- You have Linaro Forge installed. See Linaro Forge Downloads and *Installing Linaro Forge*.
- Your Linaro Forge installation is licensed. Copy your license file into the `licences` directory of your Linaro Forge installation.
- Your Linaro Forge license includes the *Thread affinity advisor* feature. Contact Linaro Sales for details about how to upgrade.
- You have access to a multi-node system with the SLURM job scheduler, the GNU C/C++/Fortran Compiler, and a supported MPI implementation. This worked example demonstrates using the *Thread affinity advisor* to optimize the performance of an example program. The suggestions provided may not be applicable to other environments or analogs must be found. Results will also vary depending on your environment and the hardware that you are profiling on. The intention is to demonstrate the different features of the *Thread affinity advisor*, not necessarily what settings should be applied to optimize an application's performance in any given scenario.

11.3.2 Compile and Profile `wave_openmp`

11.3.2.1 Prerequisites

You must install all the necessary tools as described in *Software requirements*.

11.3.2.2 Procedure

1. Compile the code. To compile the application, run make on the `openmp.makefile` file in the **examples** directory of the Linaro Forge installation.

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

2. Profile the application with Linaro MAP. In this worked example, the application is run as a SLURM job using 8 nodes and 2 processes per node:

   ```
   salloc --nodes=8
   map srun --ntasks-per-node=2 ./wave_openmp
   ```

   Take note of the performance of the application from the job output:

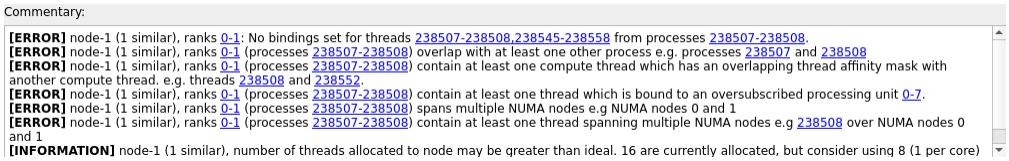
   ```
   points/second: 162.3M (10.1M per process)
   ```

1. Notice that the *Thread affinity advisor* tool button indicates the presence of thread affinity issues: 
2. Open the *Thread affinity advisor* dialog. One of the *Exemplar nodes* is automatically selected.

   In this scenario, the SLURM job is run across 8 compute nodes. Each node is fitted with 2 CPU packages, associated with a single NUMA node each. Each CPU package comprises a single L3 cache shared between 4 physical cores. SMT is not in use.

   The *Node topology viewer* matches the expected hardware topology. The red background indicates that process bindings are overlapping:

   Inspect the *Commentary* to see the issues in greater detail:



11.3.2.3 Next Steps

Optimize the application job with thread affinities shows you how to optimize the performance of the program by amending the thread affinities of the job.

11.3.3 Optimize the application job with thread affinities

Describes how to iteratively optimize the performance of a SLURM job running the **wave_openmp** example code using the *Thread affinity advisor*.

11.3.3.1 Prerequisites

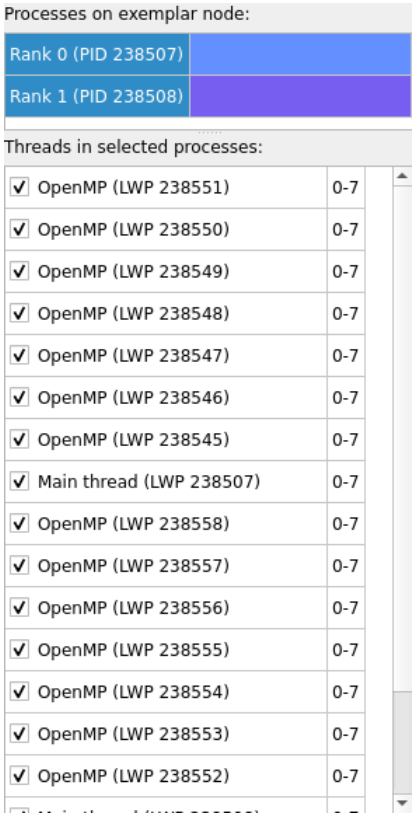
- You must install all the necessary tools as described in *Software requirements*.

11.3.3.2 Procedure

1. Examine the first two commentary items:

```
[ERROR] node-1 (1 similar), ranks 0-1: No bindings set for threads 238507-238508,238545-238558␣
↪from processes 238507-238508.
[ERROR] node-1 (1 similar), ranks 0-1 (processes 238507-238508) overlap with at least one other␣
↪process e.g. processes 238507 and 238508
```

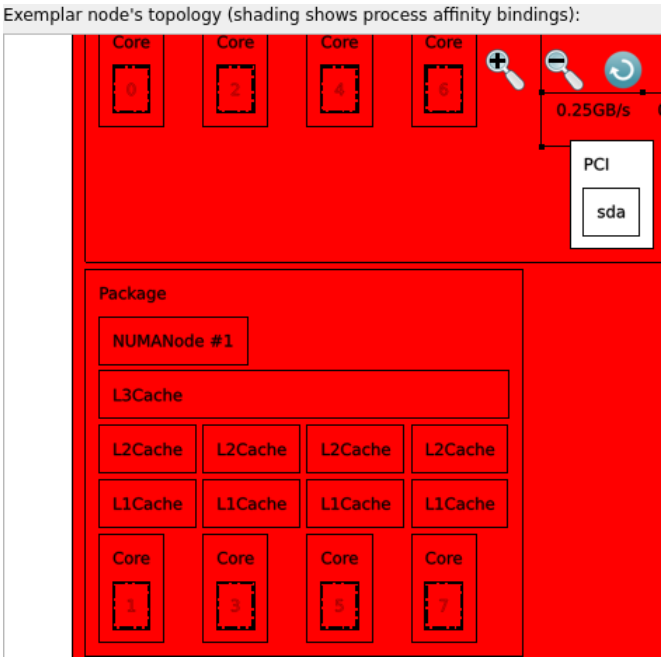
   Click the `0-1` hyperlink to select ranks 0 and 1 under *Processes and threads*. Notice that compute threads from both processes are listed and that they are all bound to logical CPUs 0-7 on the 8 core node (i.e. there are no particular bindings set for these threads):



   This is problematic, because threads spanning NUMA nodes severely impacts performance:

```
[ERROR] node-1 (1 similar), ranks 0-1 (processes 238507-238508) spans multiple NUMA nodes e.g␣
↪NUMA nodes 0 and 1
[ERROR] node-1 (1 similar), ranks 0-1 (processes 238507-238508) contain at least one thread␣
↪spanning multiple NUMA nodes e.g 238508 over NUMA nodes 0 and 1
```

2. Click on a single thread to see this in the *Node topology viewer*:



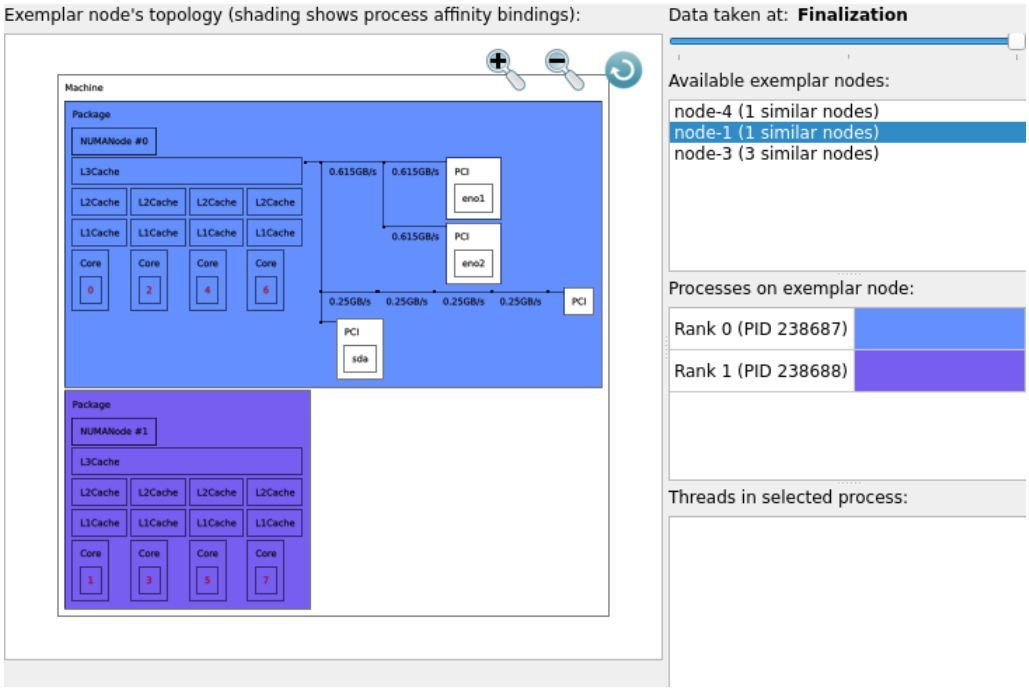
3. Click on a single logical CPU to highlight all threads that are bound to it.
4. Resolve these issues by running the SLURM job and binding each rank to a single socket (in this case, NUMA node):

```
map srun --ntasks-per-node=2 --cpu-bind=sockets ./wave_openmp
```

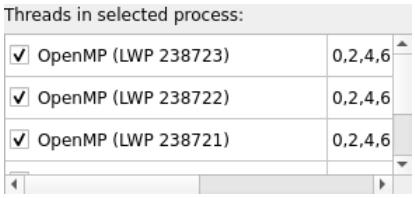
   Notice that the performance of the application has been greatly improved:

```
points/second: 2804.2M (175.3M per process)
```

5. Open the *Thread affinity advisor* dialog to see that each rank is bound to a single NUMA node:



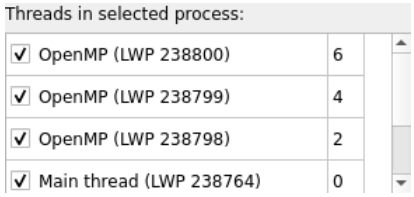
6. Click on rank 0 under *Processes and threads* to verify that each of its compute threads is bound to NUMA node 0:



7. Resolve the remaining commentary item by binding a single compute thread to each logical core. With the GNU C/C++/Fortran Compiler, this is accomplished by setting `GOMP_CPU_AFFINITY=0-7` in the environment:

```
GOMP_CPU_AFFINITY=0-7 map srun --ntasks-per-node=2 --cpu-bind=sockets ./wave_openmp
```

1. Notice that the *Thread affinity advisor* tool button no longer indicates thread affinity issues: 
2. Open the *Thread affinity advisor* dialog to see that the commentary is empty.
3. Select rank 0 to show that each compute thread is uniquely bound to a single logical CPU:



Notice that the performance of the application has not improved:

```
points/second: 2803.1M (175.2M per process)
```

In this scenario, the kernel was able to schedule threads within each NUMA node so that the application could be performant. Acting on every thread affinity issue may be unnecessary.

CHAPTER

TWELVE

SECURITY

12.1 Using Linaro Forge securely in shared HPC environments

Ensure that you maintain proper access controls in your environment. For example, limit who can access your files, such as maintaining the correct permissions for your home directory and project-specific directories, or on specific generated results files for Linaro Forge, such as `.map` files, offline or debug logs.

To prevent unauthorized users modifying sensitive files and directories, and introducing unsafe code into your environment, ensure that you assign only the minimum permissions that are required, and avoid group or world-writable permissions.

12.2 Security Vulnerability Reporting

If you think you have found a security vulnerability in Linaro Forge, then please send an email to the Linaro Product Security Incident Team (PSIRT) at psirt@linaro.org.

For more details see our Security Incident Handling Process.

CHAPTER

THIRTEEN

PROPRIETARY NOTICE

Confidentiality status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Linaro and the party that Linaro delivered this document to.

Product status

The information in this document is Final, that is for a developed product.

Web address

https://www.linaro.org/
https://www.linaroforge.com/

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X