Arm Performance Reports User Guide

Version 18.2

Contents

1 Introduction

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

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

One single page HTML report elegantly 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?

It is based on MAP's 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.

Chapters 3 to 6 of this manual describe Arm Performance Reports in more detail.

1.1 Online resources

You can find links to tutorials, training material, webinars and white papers in our online knowledge center:

Knowledge Center Arm help and tutorials

Known issues and the latest version of this user guide may be found on the support web pages:

Support Arm Developer website

2 Installation

A release of Arm Performance Reports can be downloaded from the Arm Developer website.

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

2.1 Linux installation

2.1.1 Graphical install

Untar the package and run the `installer` executable, using:

```
tar xf arm-reports-18.2-<distro>-<arch>.tar
cd arm-reports-18.2-<distro>-<arch>
./installer
```

replacing `<distro>` and `<arch>` with the OS distribution and architecture of your tar package, respectively. For example, the tarball package for Redhat 7.4 OS and Armv8-A (AArch64) architecture is: arm-reports-18.2-Redhat-7.4-aarch64.tar

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

The *Install Type* page allows you to choose which user(s) to install Arm Performance Reports for.

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

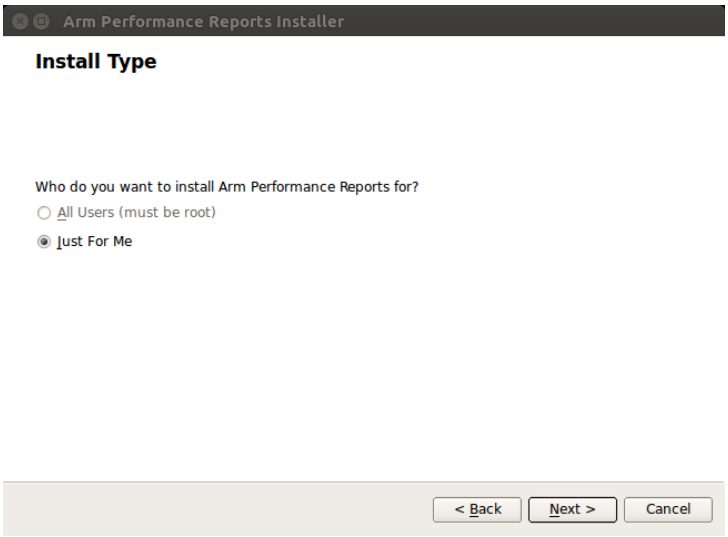


Figure 1: *Arm Performance Reports Installer—Installation type*

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

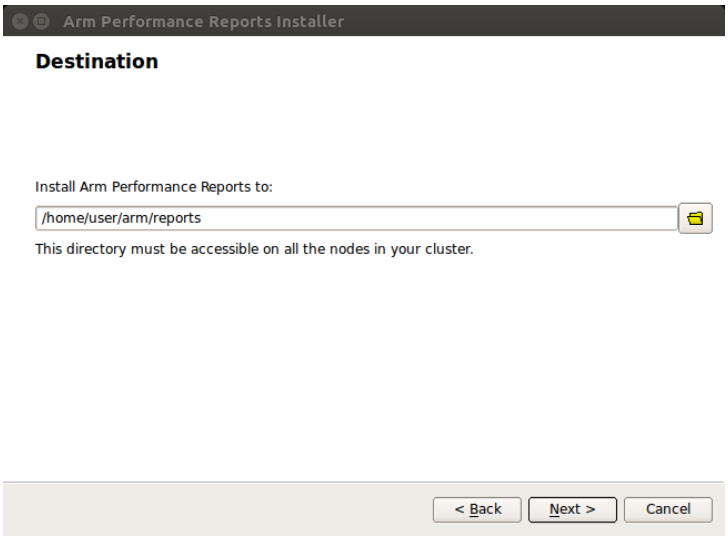


Figure 2: *Arm Performance Reports Installer—Installation directory*

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

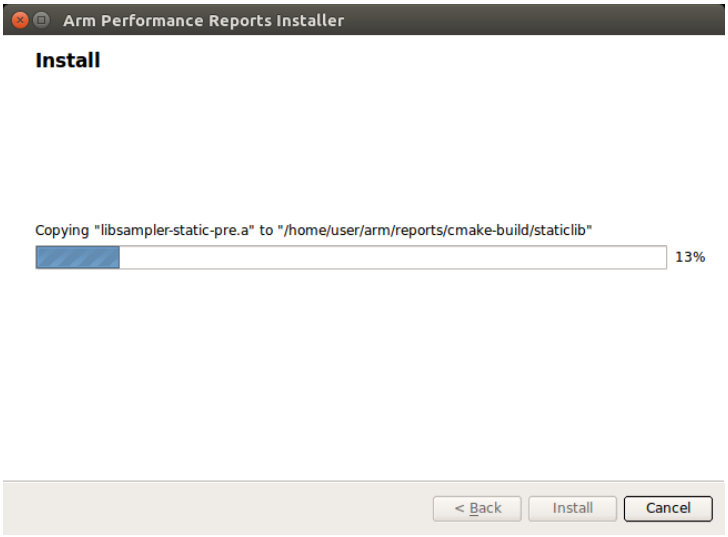


Figure 3: *Install in progress*

Arm Performance Reports does not have a GUI and does not add any desktop icons.

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

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

2.1.2 Text-mode install

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

To install using the text-mode install script, untar the package and run the `textinstall.sh` script, using:

```
tar xf arm-reports-18.2-<distro>-<arch>.tar
cd arm-reports-18.2-<distro>-<arch>
./text-install.sh
```

replacing `<distro>` and `<arch>` with the OS distribution and architecture of your tar package, respectively. For example, the tarball package for Redhat 7.4 OS and Armv8-A (AArch64) architecture is: arm-reports-18.2-Redhat-7.4-aarch64.tar

Next, you are prompted with the license agreement. To read the license, press *Return*. Following the license prompt, you are requested to enter the directory where you want to install Arm Performance Reports. This directory must be accessible on all the nodes in your cluster. Enter a directory for the installation.

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

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

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

2.2 License files

Arm Performance Reports requires a license file for its operation.

Time-limited evaluation licenses are available from the Arm Developer website.

2.3 Workstation and evaluation licenses

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

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

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

`ALLINEA_LICENSE_DIR` is an alias for `ALLINEA_LICENCE_DIR`.

2.4 Supercomputing and other floating licenses

Licensing!Floating licenses

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

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

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

A floating license consists of two files: the server license, a file name `Licence.xxxx`, and a client license file `Licence`.

The client file should be copied to {*`installation-directory`*}`/licences`, for example, `/home/user/arm/reports/licences/Licence`.

You need to edit the `hostname` line to contain the host name or IP address of the machine running the Licence Server.

See the Licence Server user guide for instructions on how to install the server license.

2.5 Architecture licensing

Licenses issued after the release of Arm Performance Reports 6.1 specify the compute node architectures that they may be used with. Licenses issued prior to this release will enable the x86_64 architecture by default.

Existing users for other architectures will be supplied with new

2.5.1 Using multiple architecture licenses

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

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

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

2.6 Environment variables

2.6.1 Report customization

Environment variables to customize your reports:

ALLINEA_NOTES

Any text in this environment variable will be included in all reports produced.

ALLINEA_MAP_TO_DCIM

Allows you to specify a `.map` file when using the `--dcim-output` argument.

ALLINEA_DCIM_SCRIPT

Path to the script to use to communicate with DCIM. Default is `${ALLINEA_TOOLS_PATH}/performance-repor ganglia-connector/pr-dcim`.

ALLINEA_GMETRIC

Path to the `gmetric` instance to use. This is specific to the `pr-dcim` script. Default is `which gmetric`.

2.6.2 Warning suppression

Environment variables for warning suppression (for use when autodetection is resulting in erroneous messages):

ALLINEA_NO_APPLICATION_PROBE

Do not attempt to auto-detect MPI or CUDA executables.

ALLINEA_DETECT_APRUN_VERSION

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

2.6.3 I/O behavior

Environment variables for handling default I/O behavior:

ALLINEA_NEVER_FORWARD_STDIN

Never forward the `stdin` of the `perf-report` command `stdin` to the program being analyzed, even if not using the GUI. Normally Arm Performance Reports only forwards `stdin` when running without the GUI.

ALLINEA_ENABLE_ALL_REPORTS_GENERATION

Enables the option in Arm Performance Reports to generate all types of results at once, using the `.all` extension.

2.6.4 Licensing

Environment variables to handle licensing:

ALLINEA_LICENCE_FILE

Location of the license file. Default is `${ALLINEA_TOOLS_PATH}/Licence`

ALLINEA_FORCE_LICENCE_FILE

Location of the license file. This ensures the license file being pointed to is used.

ALLINEA_LICENCE_DIR

Location of the licenses directory. Default is `${ALLINEA_TOOLS_PATH}/licences`.

ALLINEA_MAC_INTERFACE

Specify the host name of the network interface the license is tied to.

2.6.5 Timeouts

Environment variables for handling timeouts:

ALLINEA_NO_TIMEOUT

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

ALLINEA_PROCESS_TIMEOUT

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

ALLINEA_MPI_FINALIZE_TIMEOUT_MS

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

2.6.6 Sampler

Environment variables for handling sampler-related setup, runtime behavior, and backend processing:

ALLINEA_SAMPLER_INTERVAL

Arm Performance Reports takes a sample in each 20 milliseconds period, giving it a default sampling rate of 50Hz. This will be automatically decreased as the run proceeds to ensure a constant number of samples are taken. See `ALLINEA_SAMPLER_NUM_SAMPLES`.

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

Increasing the sampling frequency from the default is not recommended if there are lots of threads or very deep stacks in the target program because this may not leave sufficient time to complete one sample before the next sample is started.

i{Note: Custom values for `ALLINEA_SAMPLER_INTERVAL` may be overwritten by values set from the combination of `ALLINEA_SAMPLER_INTERVAL_PER_THREAD` and the expected number of threads (from `OMP_NUM_THREADS`). For more information, see `ALLINEA_SAMPLER_INTERVAL_PER_THREAD.`

ALLINEA_SAMPLER_INTERVAL_PER_THREAD

To keep overhead low, Arm Performance Reports imposes a minimum sampling interval based on the number of threads. By default, this is 2 milliseconds per thread, thus for eleven or more threads Arm Performance Reports will increase the initial sampling interval to more than 20 milliseconds.

To adjust this behavior set `ALLINEA_SAMPLER_INTERVAL_PER_THREAD` to the minimum per thread sample time, in milliseconds.

Lowering this value from the default is not recommended if there are lots of threads as this may not leave sufficient time to complete one sample before the next sample is started.

Notes:

- *Whether OpenMP is enabled or disabled in Arm Performance Reports, the final script or scheduler values set for `OMP_NUM_THREADS` will be used to calculate the sampling interval per thread (`ALLINEA_SAMPLER_INTERVAL_PER_THREAD`). When configuring your job for submission, check whether your final submission script, scheduler or the Arm Performance Reports GUI has a default value for `OMP_NUM_THREADS`.*
- *Custom values for `ALLINEA_SAMPLER_INTERVAL` will be overwritten by values set from the combination of `ALLINEA_SAMPLER_INTERVAL_PER_THREAD` and the expected number of threads from `OMP_NUM_THREADS`.*

ALLINEA_MPI_WRAPPER

To direct Arm Performance Reports to use a specific wrapper library set `ALLINEA_MPI_WRAPPER=<pathofsharedobject>`.

Arm Performance Reports ships with a number of precompiled wrappers, when your MPI is supported Arm Performance Reports will automatically select and use the appropriate wrapper.

To manually compile a wrapper specifically for your system, set `ALLINEA_WRAPPER_COMPILE=1` and `MPICC` and run `<path to Arm Performance Reports installation>/map/wrapper/build_wrapper`.

This generates the wrapper library `~/.allinea/wrapper/libmap-sampler-pmpi-<hostname>.so` with symlinks to the following files:

- `~/.allinea/wrapper/libmap-sampler-pmpi-<hostname>.so.1`
- `~/.allinea/wrapper/libmap-sampler-pmpi-<hostname>.so.1.0`
- `~/.allinea/wrapper/libmap-sampler-pmpi-<hostname>.so.1.0.0.`

ALLINEA_WRAPPER_COMPILE

To direct Arm Performance Reports to fall back to creating and compiling a just-in-time wrapper, set `ALLINEA_WRAPPER_COMPILE=1`.

In order to be able to generate a just-in-time wrapper an appropriate compiler must be available on the machine where Arm Performance Reports is running, or on the remote host when using remote connect.

Arm Performance Reports will attempt to auto detect your MPI compiler, however, setting the `MPICC` environment variable to the path to the correct compiler is recommended.

ALLINEA_MPIRUN

The path of `mpirun`, `mpiexec` or equivalent.

If set, `ALLINEA\_MPIRUN` has higher priority than that set in the GUI and the `mpirun` found in PATH.

ALLINEA_SAMPLER_NUM_SAMPLES

Arm Performance Reports collects 1000 samples per process by default. To avoid generating too much data on long runs, the sampling rate is automatically decreased as the run progresses to ensure only 1000 evenly spaced samples are stored.

You may adjust this by setting `ALLINEA_SAMPLER_NUM_SAMPLES=<positiveinteger>`.

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

ALLINEA_KEEP_OUTPUT_LINES

Specifies the number of lines of program output to record in `.map` files. Setting to `0` will remove the line limit restriction, although this is not recommended as it may result in very large `.map` files if the profiled program produces lots of output.

ALLINEA_KEEP_OUTPUT_LINE_LENGTH

The maximum line length for program output that will be recorded in `.map` files. Lines containing more characters than this limit will be truncated. Setting to `0` will remove the line length restriction. This is not recommended because it may result in very large `.map` files if the profiled program produces lots of output per line.

ALLINEA_PRESERVE_WRAPPER

To gather data from MPI calls Arm Performance Reports generates a wrapper to the chosen MPI implementation.

By default, the generated code and shared objects are deleted when Arm Performance Reports no longer needs them.

To prevent Arm Performance Reports from deleting these files set `ALLINEA_PRESERVE_WRAPPER=1`.

Note: If you are using remote launch then this variable must be exported in the remote script.

ALLINEA_SAMPLER_NO_TIME_MPI_CALLS

To prevent Arm Performance Reports from timing the time spent in MPI calls, set `ALLINEA_SAMPLER_NO_TIME_MPI_CALLS`.

ALLINEA_SAMPLER_TRY_USE_SMAPS

To allow Arm Performance Reports to use `/proc/[pid]/smaps` to gather memory usage data, set this `ALLINEA_SAMPLER_TRY_USE_SMAPS`. This is not recommended since it slows down sampling significantly.

MPICC

To create the MPI wrapper Arm Performance Reports will try to use `MPICC`, then if that fails search for a suitable MPI compiler command in `PATH`. If the MPI compiler used to compile the target binary is not in `PATH` (or if there are multiple MPI compilers in `PATH`) then `MPICC` should be set.

2.6.7 Simple troubleshooting

Environment variables for simple troubleshooting:

ALLINEA_DEBUG_HEURISTICS

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

3 Running with an example program

This section takes you through compiling and running one of the the example programs.

3.1 Overview of the example source code

3.2 Compiling

Arm provides 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 are built using the same makefile, `wave.makefile`. To navigate and run `wave.makefile`, use:

```
cd <INSTALL_DIR>/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 `<INSTALL_DIR>/examples` directory.

Depending on the default compiler on your system you may 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 appriopriate compilation command for the compiler you want to use, and comment those of the GNU compiler.

Note: The compilation commands for other popular compilers are already present within the makefile, separated by compiler.

Note: Although the example makefiles include the `-g` flag, Arm Performance Reports does *not* require this and you should not use them in your own makefiles.

In most cases Arm Performance Reports can run on an unmodified binary with no recompilation or linking required.

3.2.1 Cray X-series

On Cray X-series systems the example program must either be dynamically linked (using `-dynamic`) or explicitly linked with the Arm profiling libraries.

Example how to dynamically link:

```
cc -dynamic -g -O3 wave.c -o wave -lm -lrt

ftn -dynamic -G2 -O3 wave.f90 -o wave -lm -lrt
```


Example how to explicitly link with the Arm profiling libraries: First 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 Arm sampler and MPI wrapper
      libraries in
   the link line.  There's a comprehensive description of the link
       ordering
   requirements in the `Preparing a Program for Profiling' section
       of either
   userguide-forge.pdf or userguide-reports.pdf, located in
   /opt/arm/forge/doc/.
```

Then follow the instructions in the output to link the example program with the Arm profiling 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
```

3.3 Running

As this example uses MPI you need to run on a compute node on your cluster. Your site's help pages and support staff 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, as follows:

```
$ qsub -I
qsub: waiting for job 31337 to start
qsub: job 31337 ready
$ cd arm/reports/examples
$ mpiexec -n 4 ./wave_c
Wave solution running with 4 processes

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
```


If you see output similar to this then the example program is compiled and working correctly.

3.4 Generating a performance report

Make sure the Arm Performance Reports module for your system has been loaded:

```
$ perf-report --version
Arm Performance Reports
Copyright (c) 2002-2018 Arm Limited (or its affiliates). All
   rights reserved.
...
```

If this command cannot be found consult the site documentation to find the name of the correct module.

Once the module is loaded, you can add the `perf-report` command in front of your existing `mpiexec` command-line:

```
perf-report mpiexec -n 4 examples/wave_c
```

If your program is submitted through a batch queuing system, then modify your submission script to load the Arm module and add the 'perf-report' line in front of the `mpiexec` command you want to generate a report for.

The program runs as usual, although startup and shutdown may take a few minutes longer while Arm Performance Reports 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 that both `.txt` and `.html` versions are automatically generated.

4 Running with real programs

This section will take you through compiling and running your own programs.

Arm 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.

4.1 Preparing a program for profiling

In most cases you do not need to recompile your program to use it with Performance Reports, although in some cases it may need to be relinked, as explained in section 4.1.1 Linking.

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
```

Arm®v8 (AArch64) machines

Unwind information is not always compiled in by default on this platform. For accurate results programs that are not compiled with debug information (`-g`) should at least be compiled with the `-fasynchronous-unwind-tables` flag or the `-funwind-tables` flag, preferably the former.

4.1.1 Linking

To collect data from your program, Performance Reports uses two small profiler libraries, `map-sampler` and `map-sampler-pmpi`. These 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.

Note: Although these libraries contain the word 'map' they are used for both Arm Performance Reports and Arm MAP.

This automatic linking when starting your program only works if your program is dynamically-linked. Programs may be dynamically-linked or statically-linked, and for MPI programs this is normally determined by your MPI library. Most MPI libraries are configured with `--enable-dynamic` by default, and `mpicc/mpif90` produce dynamically-linked executables that 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 G.1.3 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 Performance Reports warns you that it could not pre-load the sampler libraries, this often means that your MPI library was not configured with `--enable-dynamic`, or that the `LD_PRELOAD` mechanism is not supported on your platform. You now have three options:

1. Try compiling and linking your code dynamically. On most platforms this allows Performance Reports to use the `LD_PRELOAD` mechanism to automatically insert its libraries into your application at runtime.
2. Link MAP's `map-sampler` and `map-sampler-pmpi` libraries with your program at link time manually.

   See 4.1.2 Dynamic linking on Cray X-Series systems, or 4.1.3 Static linking and 4.1.4 Static linking on Cray X-Series systems.
3. Finally, it may be that your system supports dynamic linking but you have a statically-linked MPI. You can try to recompile the MPI implementation with `--enable-dynamic`, or find a dynamically-linked version on your system and recompile your program using that version. This will produce a dynamically-linked program that Performance Reports can automatically collect data from.

4.1.2 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 Performance Reports sampling libraries.

Compiling the Arm MPI Wrapper Library

First you must compile the Arm MPI wrapper library for your system using the `make-profiler-libraries --platform=cray --lib-type=shared` command.

Note: Performance Reports also uses this library.

```
user@login:~/myprogram$ make-profiler-libraries --platform=cray
   --lib-type=shared

Created the libraries in /home/user/myprogram:
libmap-sampler.so       (and .so.1, .so.1.0, .so.1.0.0)
libmap-sampler-pmpi.so  (and .so.1, .so.1.0, .so.1.0.0)

To instrument a program, add these compiler options:
compilation for use with MAP - not required for Performance
   Reports:
    -g (or '-G2' for native Cray Fortran) (and -O3 etc.)
linking (both MAP and Performance Reports):
    -dynamic -L/home/user/myprogram -lmap-sampler-pmpi -lmap-
       sampler -Wl,--eh-frame-hdr

Note: These libraries must be on the same NFS/Lustre/GPFS
   filesystem as your
program.

Before running your program (interactively or from a queue), set
LD_LIBRARY_PATH:
export LD_LIBRARY_PATH=/home/user/myprogram:$LD_LIBRARY_PATH
map  ...
```



```
or add -Wl,-rpath=/home/user/myprogram when linking your program.
```

Linking with the Arm MPI Wrapper Library

```
mpicc -G2 -o hello hello.c -dynamic -L/home/user/myprogram \
      -lmap-sampler-pmpi -lmap-sampler -Wl,--eh-frame-hdr
```

PGI Compiler

When linking OpenMP programs you must pass the `-Bdynamic` command line argument to the compiler when linking dynamically.

When linking C++ programs you must pass the `-pgc++libs` command line argument to the compiler when linking.

4.1.3 Static linking

If you compile your program statically, that is your MPI uses a static library or you pass the `-static` option to the compiler, then you must explicitly link your program with the Arm sampler and MPI wrapper libraries.

Compiling the Arm MPI Wrapper Library

First you must compile the Arm MPI wrapper library for your system using the `make-profiler-libraries --lib-type=static` command.

Note: Performance Reports also uses this library.

```
user@login:~/myprogram$ make-profiler-libraries --lib-type=static

Created the libraries in /home/user/myprogram:
   libmap-sampler.a
   libmap-sampler-pmpi.a

To instrument a program, add these compiler options:
   compilation for use with MAP - not required for Performance
      Reports:
      -g (and -O3 etc.)
   linking (both MAP and Performance Reports):
      -Wl,@/home/user/myprogram/allinea-profiler.ld ...
         EXISTING_MPI_LIBRARIES
   If your link line specifies EXISTING_MPI_LIBRARIES (e.g. -lmpi)
      , then
   these must appear *after* the Arm sampler and MPI wrapper
      libraries in
   the link line.  There's a comprehensive description of the link
       ordering
   requirements in the 'Preparing a Program for Profiling' section
       of either
   userguide-forge.pdf or userguide-reports.pdf, located in
   /opt/arm/forge/doc/.
```

Linking with the Arm MPI Wrapper Library

The `-Wl,@/home/user/myprogram/allinea-profiler.ld` syntax tells the compiler to look in `/home/user/myprogram/allinea-profiler.ld` for instructions on how to link with the

Arm sampler. Usually this is sufficient, but not in all cases. The rest of this section explains how to manually add the Arm sampler to your link line.

PGI Compiler

When linking C++ programs you must pass the `-pgc++libs` command line argument to the compiler when linking.

The PGI compiler must be 14.9 or later. Using earlier versions of the PGI compiler will fail with an error such as "`Error: symbol 'MPI_F_MPI_IN_PLACE' can not be both weak and common`" due to a bug in the PGI compiler's weak object support.

If you do not have access to PGI compiler 14.9 or later try compiling and the linking Arm MPI wrapper as a shared library as described in 4.1.2 Dynamic linking on Cray X-Series systems Ommit the option `--platform=cray` if you are not on a Cray.

Cray

When linking C++ programs you may encounter a conflict between the Cray C++ runtime and the GNU C++ runtime used by the 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:

1. *After* your program's object (`.o`) files.
2. *After* your program's own static libraries, for example `-lmylibrary`.
3. *After* the path to the profiler libraries (`-L/path/to/profiler-libraries`).
4. *Before* the MPI's Fortran wrapper library, if any. For example `-lmpichf`.
5. *Before* the MPI's implementation library usually `-lmpi`.
6. *Before* the Arm sampler library `-lmap-sampler`.

The sampler library `-lmap-sampler` must go:

1. *After* the Arm MPI wrapper library.
2. *After* your program's object (`.o`) files.
3. *After* your program's own static libraries, for example `-lmylibrary`.

4. *After* `-Wl,--undefined,allinea_init_sampler_now`.
5. *After* the path to the profiler libraries `-L/path/to/profiler-libraries`.
6. *Before* `-lstdc++`, `-lgcc_eh`, `-lrt`, `-lpthread`, `-ldl`, `-lm` and `-lc`.

For example:

```
mpicc hello.c -o hello -g -L/users/ddt/allinea \
   -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/allinea \
   -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
```

4.1.4 Static linking on Cray X-Series systems

Compiling the MPI Wrapper Library

On Cray X-Series systems use `make-profiler-libraries --platform=cray --lib-type=static` instead:

```
Created the libraries in /home/user/myprogram:
   libmap-sampler.a
   libmap-sampler-pmpi.a

To instrument a program, add these compiler options:
   compilation for use with MAP - not required for Performance
      Reports:
      -g (or -G2 for native Cray Fortran) (and -O3 etc.)
   linking (both MAP and Performance Reports):
      -Wl,@/home/user/myprogram/allinea-profiler.ld ...
         EXISTING_MPI_LIBRARIES
   If your link line specifies EXISTING_MPI_LIBRARIES (e.g. -lmpi)
      , then
   these must appear *after* the Arm sampler and MPI wrapper
      libraries in
   the link line.  There's a comprehensive description of the link
       ordering
   requirements in the 'Preparing a Program for Profiling' section
       of either
```



```
userguide-forge.pdf or userguide-reports.pdf, located in
/opt/arm/forge/doc/.
```

Linking with the MPI Wrapper Library

```
cc hello.c -o hello -g -Wl,@allinea-profiler.ld

ftn hello.f90 -o hello -g -Wl,@allinea-profiler.ld
```

4.1.5 Dynamic and static linking on Cray X-Series systems using the modules environment

If your system has the Arm module files installed, you can load them and build your application as usual. See section 4.1.6.

1. `module load reports` 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.

4.1.6 map-link modules installation on Cray X-Series

To facilitate dynamic and static linking of user programs with the Arm MPI Wrapper and Sampler libraries Cray X-Series System Administrators can integrate the map-link-dynamic and map-link-static modules into their module system. Templates for these modules are supplied as part of the Arm Performance Reports package.

Copy files `share/modules/cray/map-link-*` into a dedicated directory on the system.

For each of the two module files copied:

1. Find the line starting with **conflict** and correct the prefix to refer to the location the module files were installed, for example, `arm/map-link-static`. The correct prefix depends on the subdirectory (if any) under the module search path the `map-link-*` modules were installed.
2. Find the line starting with **set MAP_LIBRARIES_DIRECTORY "NONE"** and replace "NONE" with a user writable directory accessible from the login and compute nodes.

After installed you can verify whether or not the prefix has been set correctly with 'module avail', the prefix shown by this command for the map-link-* modules should match the prefix set in the 'conflict' line of the module sources.

4.2 Express Launch mode

Arm Performance Reports can be launched by typing its command name in front of an existing `mpiexec` command:

```
$ perf-report mpiexec -n 256 examples/wave_c 30
```

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


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

Try this instead:
    perf-report --np=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 Arm Performance Reports products.

Normal redirection syntax may be used to redirect standard input and standard output.

4.2.1 Compatible MPIs

The following lists the MPI implementations supported by Express Launch:

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

4.3 Compatibility Launch mode

Compatibility Mode must be used if Arm Performance Reports 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 Mode replace the `mpiexec` command with the `perf-report` command. For example:

```
mpiexec --np=256 ./wave_c 20
```

This would become:

```
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.

4.4 Generating a performance report

Make sure the Arm Performance Reports module for your system has been loaded:

```
$ perf-report --version
Arm Performance Reports
Copyright (c) 2002-2018 Arm Limited (or its affiliates). All
   rights reserved.
...
```

If this command cannot be found consult the site documentation to find the name of the correct module.

Once the module is loaded, you can simply add the `perf-report` command in front of your existing `mpiexec` command-line:

```
perf-report mpiexec -n 4 examples/wave_c
```

If your program is submitted through a batch queuing system, then modify your submission script to load the Arm module and add the 'perf-report' line in front of the `mpiexec` command you want to generate a report for.

The program runs as usual, although startup and shutdown may take a few minutes longer while Arm Performance Reports 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 that 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 `ALLINEA_NOTES` before running `perf-report`:

```
$ ALLINEA_NOTES="Run with inp421.dat and mc=1" perf-report mpiexec
    -n 512 ./parEval.bin --use-mc=1 inp421.dat
```

The string in the `ALLINEA_NOTES` environment variable is included in all report files produced.

4.5 Specifying 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 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:

- `--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\_23\-30.txt` and `/tmp/wave\_f\_16p\_2013-11-18\_23\-30.html`.

4.6 Support for DCIM systems

Performance Reports includes support for Data Center Infrastructure Management (DCIM) systems.

You can output all the metrics generated by the Performance Reports to a script using the `--dcim-output` argument. By default, the `pr-dcim` script is called and the collected metrics are sent to Ganglia (a System Monitoring tool).

The `pr-dcim` script looks for a `gmetric` implementation as part of the Ganglia software, and call it as many times as there are metrics.

4.6.1 Customizing your DCIM script

The default `pr-dcim` script is located in *`installation-directory`*`/performance-reports/ganglia-connector/pr-dcim`.

However, you can use your own custom script by specifying the `ALLINEA_DCIM_SCRIPT` environment variable.

This option is recommended if you are using a System Monitoring tool other than Ganglia.

Such a script is expecting arguments as follows, each argument can be specified once per metric:

- `-V{METRIC}={VALUE}` (mandatory) specifies that the metric `METRIC` has the value `VALUE`.
- `-U{METRIC}={UNITS}` (optional) specifies that the metric `METRIC` is expressed in `UNITS`.
- `-T{METRIC}={TITLE}` (optional) specifies that the metric `METRIC` has title `TITLE`.
- `-t{METRIC}={TYPE}` (optional) specifies that the metric `METRIC` has `TYPE` data type.

4.6.2 Customising the `gmetric` location

You can specify the path to your `gmetric` implementation by using the `ALLINEA_GMETRIC` environment variable.

Your `gmetric` version must accept the following command line arguments:

- `-n` {`NAME`} (mandatory) specifies the name of the metric (starts with `com.allinea`).
- `-t` {`TYPE`} (mandatory) specifies the type of the metric (for example, `double` or `int32`).
- `-v` {`VALUE`} (mandatory) specifies the value of the metric.
- `-g` {`GROUP`} (optional) specifies which groups the metric belongs to (for example `allinea`).
- `-u` {`UNIT`} (optional) specifies the unit of the metric. For example, `%`, `Watts`, `Seconds`, and so on.
- `-T` {`TITLE`} (optional) specifies the title of the metric.

4.7 Enable and disable metrics

```
--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, 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.

5 Summarizing an existing MAP file

Arm Performance Reports can be used to summarize an application profile generated by Arm MAP. To produce a performance report from an existing 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 Performance Reports produces its report, such as `--output`, work as expected.

For best results the Performance Reports and MAP versions should match, for example, Performance Reports 18.2 with MAP 18.2. Performance Reports can use MAP files from versions of MAP as old as 5.0.

6 Interpreting performance reports

This section takes you through interpreting the reports produced by Arm Performance Reports.

Reports are generated in both HTML and textual formats for each run of your application by default. The same information is presented in both.

If you wish to combine Arm Performance Reports with other tools, consider using the CSV output format.

See 6.13 for more details.

6.1 HTML performance reports

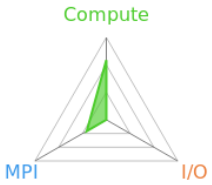
Viewing HTML files is best done on your local machine. Many sites have places you can put HTML files to be viewed from within the intranet. These directories are a good place to automatically send your performance reports to. Alternatively, you can use `scp` or even the excellent `sshfs` to make the reports available to your laptop or desktop:

```
$ scp login1:arm/reports/examples/wave_c_4p*.html .
$ firefox wave_c_4p*.html
```

The following report was generated by a 8 MPI processes and 2 OpenMP threads per process run of the `wave_openmp.c` example program on a typical HPC cluster:

arm PERFORMANCE REPORTS

Command:	mpirun -np 8 examples/wave_openmp 60
Resources:	1 node (8 physical, 8 logical cores per node)
Memory:	15 GiB per node
Tasks:	8 processes, OMP_NUM_THREADS was 2
Machine:	mars
Start time:	Tue Nov 7 2017 15:35:50 (UTC)
Total time:	61 seconds (about 1 minutes)
Full path:	/scratch/user/reports/examples



Summary: wave_openmp is Compute-bound in this configuration

Compute	72.6%	Time spent running application code. High values are usually good. This is **high**; check the CPU performance section for advice
MPI	27.4%	Time spent in MPI calls. High values are usually bad. This is **low**; this code may benefit from a higher process count
I/O	0.0%	Time spent in filesystem I/O. High values are usually bad. This is **negligible**; there's no need to investigate I/O performance

This application run was Compute-bound. 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 72.6% CPU time:

Single-core code	8.2%
OpenMP regions	91.8%
Scalar numeric ops	5.1%
Vector numeric ops	0.0%
Memory accesses	56.9%

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 27.4% MPI time:

Time in collective calls	1.2%
Time in point-to-point calls	98.8%
Effective process collective rate	19.5 kB/s
Effective process point-to-point rate	305 kB/s

Most of the time is spent in point-to-point calls with a very low transfer rate. This suggests load imbalance is causing synchronization overhead; use an MPI profiler to investigate.

I/O

A breakdown of the 0.0% I/O time:

Time in reads	0.0%
Time in writes	0.0%
Effective process read rate	0.00 bytes/s
Effective process write rate	0.00 bytes/s

No time is spent in I/O operations. There's nothing to optimize here!

OpenMP

A breakdown of the 91.8% time in OpenMP regions:

Computation	9.9%
Synchronization	90.1%
Physical core utilization	100.0%
System load	167.0%

Significant time is spent synchronizing threads in parallel regions. Check the affected regions with a profiler.

The system load is high. Ensure background system processes are not running.

Memory

Per-process memory usage may also affect scaling:

Mean process memory usage	38.6 MiB
Peak process memory usage	53.7 MiB
Peak node memory usage	17.0%

The peak node memory usage is very low. 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 are not supported (no intel_rapl module)

Figure 4: *A performance report for the wave_openmp.c example*

Your report may differ from this one depending on the performance and network architecture of the machine it is run on, but the basic structure of these reports is always the same. This makes comparisons between reports simple, direct and intuitive. Each section of the report is described in the following sections.

6.2 Report summary

This characterizes how the application's wallclock time was spent, broken down into compute, MPI and I/O.

In this example file you see that Arm Performance Reports has identified the program as being compute-bound, which simply means that most of its time is spent inside application code rather than communicating or using the filesystem.

The snippets of advice, such as "this code may benefit from running at larger scales" are good starting points for guiding future investigations and are designed to be meaningful to scientific users with no previous MPI tuning experience.

The triangular radar chart in the top-right corner of the report reflects the values of these three key measurements: compute, MPI and I/O. It is helpful to recognize and compare these triangular shapes when switching between multiple reports.

6.2.1 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.

6.2.2 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`.

6.2.3 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 may also be performed by the MPI library. `MPI_File_close` *is treated as time spent writing, which is often but not always correct.*

6.3 CPU breakdown

Note: All of the metrics described in this section are only available on x86_64 systems.

This section breaks down the time spent in application and library code further by analyzing the kinds of instructions that this time was spent on.

Note that 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.

6.3.1 Single core code

The percentage of wall-clock in which the application executed using only one core per process, as opposed to 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.

6.3.2 OpenMP code

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.

6.3.3 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.

6.3.4 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, as that is the only way to achieve anything close to the peak performance of modern processors.

If this value is low it is worth checking the compiler's vectorization report to understand why the most time consuming loops are not using these operations. Compilers need a good deal of help to efficiently vectorize non-trivial loops and the investment in time is often rewarded with 2x–4x performance improvements.

6.3.5 Memory accesses

The percentage of time spent in memory access operations, such as mov, load, store. A portion of the time spent in instructions using indirect addressing is also included here. A high figure here shows the application is memory-bound and is not able to make full use of the CPU resources. Often it is possible to reduce this figure by analyzing loops for poor cache performance and problematic memory access patterns, boosting performance significantly.

A high percentage of time spent in memory accesses in an OpenMP program is often a scalability problem. If each core is spending most of its time waiting for memory, even 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.

6.3.6 Waiting for accelerators

The percentage of time that the CPU is waiting for the accelerator.

6.4 CPU metrics breakdown

This section presents key CPU performance measurements gathered using the Linux perf event subsystem.

Note: All of the metrics described in this section are only available on Armv8 systems. These metrics are not available on virtual machines. Linux perf events performance events counters will need to be accessible on all systems on which the target program will run. See section F.3.1 for details.

6.4.1 Cycles per instruction

The average amount of CPU cycles lapsed per retired instruction. This metric is affected by CPU frequency scaling and various issues, most notably hardware interrupt counts.

6.4.2 Stalled cycles

The percentage of CPU cycles lapsed on which no operation instructions were issued.

6.4.3 L2 cache misses

The rate of level 2 data cache refills.

6.4.4 Mispredicted branch instructions

The rate of mispredicted branch instructions. This counts the number of incorrectly predicted retired branches that are conditional, unconditional, branch and link, return or eret.

6.5 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.

6.5.1 Computation

The percentage of time threads in OpenMP regions spent computing as opposed to waiting or sleeping. Keeping this high is one important way to ensure OpenMP codes scale well. If this is high then look at the CPU breakdown to see whether that time is being well spent on, for example, floating-point operations, or whether the cores are mostly waiting for memory accesses.

6.5.2 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 are finishing sooner and waiting then this value will increase. Equally, there is some overhead associated with entering and leaving OpenMP regions and a high synchronization time may suggest that the threading is too fine-grained. In general, OpenMP performance is better when outer loops are parallelized rather than inner loops.

6.5.3 Physical core utilization

Modern CPUs often have multiple *logical* cores for each *physical* cores. This is often referred to as hyper-threading. These logical cores may 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 then `OMP_NUM_THREADS` is set to a larger number of threads than physical cores are available and performance may be impacted, usually showing up as a larger percentage of time in OpenMP synchronization or memory accesses.

6.5.4 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 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% may indicate your program is not taking full advantage of the CPU resources available on a compute node.

6.6 Threads breakdown

This section breaks down 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.

6.6.1 Computation

The percentage of time worker threads spent computing as opposed to waiting in locks and synchronization primitives. If this is high then look at the CPU breakdown to see whether that time is being well spent on, for example floating-point operations, or whether the cores are mostly waiting for memory accesses.

6.6.2 Synchronization

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 should be tracked down with a multicore profiler such as Arm MAP.

6.6.3 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 may 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.

A program may have dozens of helper threads that do little except sleep and these will not be shown here. Only threads actively and simultaneously consuming CPU time are included in this metric.

6.6.4 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 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% may indicate your program is not taking full advantage of the CPU resources available on a compute node.

6.7 MPI breakdown

This section breaks down the time spent in MPI calls reported in the summary. It is only of interest if the program is spending a significant amount of its time in MPI calls in the first place.

All the rates quoted here are inbound + outbound rates. That is, the rate of communication from the process to the MPI API, and not of the underlying hardware directly, is being measured.

This application-perspective is found throughout Arm Performance Reports and in this case allows the results to capture effects such as faster intra-node performance, zero-copy transfers and so on.

Note that for programs that make MPI calls from multiple threads (MPI is in `MPI_THREAD_SERIALIZED` or `MPI_THREAD_MULTIPLE` mode) Arm Performance Reports will only display metrics for MPI calls made on the main thread.

6.7.1 Time in collective calls

The percentage of time spent in collective MPI operations such as `MPI_Scatter`, `MPI_Reduce` and `MPI_Barrier`.

6.7.2 Time in point-to-point calls

The percentage of time spent in point-to-point MPI operations such as `MPI_Send` and `MPI_Recv`.

6.7.3 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. That is, 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.

6.7.4 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` are able to achieve much higher effective transfer rates than synchronous calls.

Overlapping communication and computation is often a good strategy to improve application performance and scalability.

6.8 I/O breakdown

This section breaks down the amount of time spent in library and system calls relating to I/O, such as read, write and close. I/O due to 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.

6.8.1 Time in reads

The percentage of time spent on average in read operations from the application's perspective, not the filesystem's perspective. Time spent in the `stat` system call is also included here.

6.8.2 Time in writes

The percentage of time spent on average in write and sync operations from the application's perspective, not the filesystem's perspective.

Opening and closing files is also included here, as measurements have shown that the latest networked filesystems can spend significant amounts of time opening files with create or write permissions.

6.8.3 Effective process read rate

The average transfer rate during read operations from the application's perspective. A cached read will have a much higher read rate than one that has to hit a physical disk. This is particularly important to optimize for as current clusters often have complex storage hierarchies with multiple levels of caching.

6.8.4 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, but 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.

6.8.5 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 running on each node. Therefore, the data presented gives the information gathered on a per-node basis. The data presented is also cumulative over all of the processes run on a node, not only the application being profiled. Therefore, there may be some data reported to be read and written even if the application itself does not perform file I/O through Lustre. However, an assumption is made that the majority of data read and written through the Lustre client will be from an I/O intensive application, not from background processes. This assumption has been observed to be reasonable. For generated application profiles with more than a few megabytes of data read or written, almost all of the data reported in Arm 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, as well as a count of some metadata operations. Lustre does not just store pure data, but associates this data with metadata, which describes where data is stored on the parallel file system and how to access it. This metadata is stored separately from data, and needs to be accessed whenever new files are opened, closed, or files are resized. Metadata operations consume time and add to the latency in accessing the data. Therefore, frequent metadata operations can slow down the performance of I/O to Lustre. Arm Performance Reports reports on the total number of metadata operations, as well as the total number of file opens that are encountered by a Lustre client. With the information provided in Arm Performance Reports you can observe the rate at which data is read and written to Lustre through the Lustre client, as well as be able to identify whether a slow read or write rate can be correlated to a high rate of expensive metadata operations.

Notes:

- *For jobs run on multiple nodes, the reported values are the mean across the nodes.*
- *If you have more than one Lustre filesystem mounted on the compute nodes the values are summed across all Lustre filesystems.*
- *Metadata metrics are only available if you have the Advanced Metrics Pack add-on for Arm 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' may count as multiple operations, for example, when it creates a new file or truncates an existing one.

6.9 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, that is physical RAM usage, and not virtual memory usage. Most HPC jobs attempt to stay within the physical RAM of their node for performance reasons.

6.9.1 Mean process memory usage

The average amount of memory used per-process across the entire length of the job.

6.9.2 Peak process memory usage

The peak memory usage seen by one process at any moment during the job. If this varies greatly from the mean process memory usage then it may 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 as rapid allocation and deallocation of large amounts of memory is universally avoided for performance reasons.

6.9.3 Peak node memory usage

The peak percentage of memory seen used on any single node during the entire run. If this is close to 100% then swapping may be occurring, or the job may be likely to hit hard system-imposed limits. If this is low then it may be more efficient in CPU hours to run with a smaller number of nodes and a larger workload per node.

6.10 Accelerator breakdown

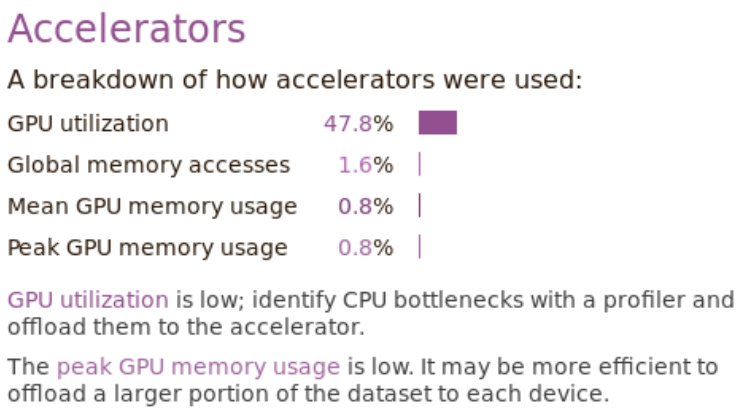


Figure 5: *Accelerator metrics report*

This section shows the utilization of NVIDIA CUDA accelerators by the job.

6.10.1 GPU utilization

The average percentage of the GPU cards working when at least one CUDA kernel is running.

6.10.2 Global memory accesses

The average percentage of time that the GPU cards were reading or writing to global (device) memory.

6.10.3 Mean GPU memory usage

The average amount of memory in use on the GPU cards.

6.10.4 Peak GPU memory usage

The maximum amount of memory in use on the GPU cards.

6.11 Energy breakdown

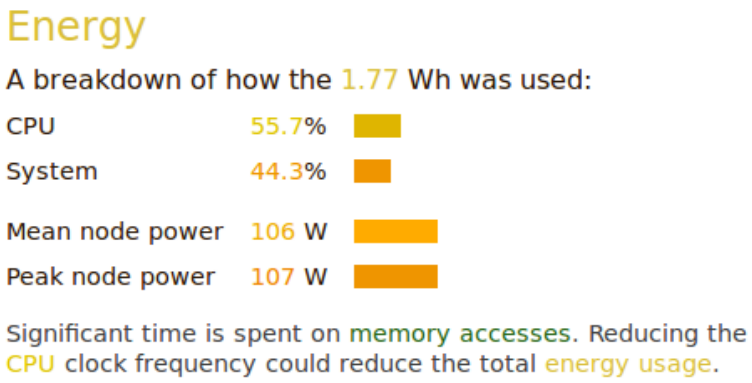


Figure 6: *Energy metrics report*

This section shows the energy used by the job, broken down by component, for example CPU and accelerators.

6.11.1 CPU

The percentage of the total energy used by the CPUs.

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.

6.11.2 Accelerator

The percentage of energy used by the accelerators. This metric is only shown when a CUDA card is present.

6.11.3 System

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%.

6.11.4 Mean node power

The average of the mean power consumption of all the nodes in Watts.

6.11.5 Peak node power

The node with the highest peak of power consumption in Watts.

6.11.6 Requirements

CPU power measurement requires an Intel CPU with RAPL support, for example Sandy Bridge or newer, and the intel_rapl powercap kernel module to be loaded.

Node power monitoring is implemented via one of two methods: the Arm IPMI energy agent which can read IPMI power sensors, or the Cray HSS energy counters.

For more information on how to install the Arm IPMI energy agent please see G.4 Arm IPMI Energy Agent. The Cray HSS energy counters are known to be available on Cray XK6 and XC30 machines.

Accelerator power measurement requires a NVIDIA GPU that supports power monitoring. This can be checked on the command-line with `nvidia-smi -q -d power`. If the reported power values are reported as "N/A", power monitoring is not supported.

6.12 Textual performance reports

The same information is presented as in 6.1 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:   /global/users/mark/arm/reports/examples
Notes:

Summary: wave_c is compute-bound in this configuration
Compute:                      82.4% |=======|
MPI:                          17.6% |=|
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.
As little time is spent in MPI calls, this code may also benefit
   from running at larger scales.
...
```

A combination of `grep` and `sed` can be useful for extracting and comparing values between multiple runs, or for automatically placing such data into a centralized database.

6.13 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.

6.14 Worked examples

The best way to understand how to use and interpret performance reports is by example. You can download several sets of real-world reports with analysis and commentary from the Arm Developer website.

At the time of writing there are three collections available, which are described in the following sections.

6.14.1 Code characterization and run size comparison

A set of runs from well-known HPC codes at different scales showing different problems:

Characterization of HPC codes and problems

6.14.2 Deeper CPU metric analysis

A look at the impact of hyper-threading on the performance of a code as seen through the CPU instructions breakdown:

Exploring hyperthreading

6.14.3 I/O performance bottlenecks

The open source MAD-bench I/O benchmark is run in several different configurations, including on a laptop, and the performance implications analyzed:

Understanding I/O behavior

7 Configuration

Arm Performance Reports generally requires no configuration before use.

If you only intend to use Arm Performance Reports and have checked that it works on your system without extra setup then you can safely ignore the rest of this section.

7.1 Compute node access

When Arm Performance Reports needs to access another machine as part of starting one of *MPICH 1–3*, *Intel MPI*, and *SGI MPT*, it attempts to use the secure shell, `ssh`, by default.

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

Arm Performance Reports checks for the script at `~/.allinea/remote-exec`, and it will be executed as follows:

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

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

SSH based remote-exec

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

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

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

See the `ssh-keygen` manual page for more information.

Testing

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

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

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

If you are having trouble setting up `remote-exec`, please contact Arm support at Arm support for assistance.

Windows

The functionality described above is also provided by the Windows remote client. 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 Arm Performance Reports.

A Getting support

While this document attempts to cover as many parts of the installation, features and uses of our tool as possible, there will be scenarios or configurations that are not covered, or are only briefly mentioned, or you may on occasion experience a problem using the product.

You can contact Arm support at Arm support.

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

- Version number of Arm Performance Reports (for example, `perf-report --version`) and your operating system and the distribution, for example Red Hat Enterprise Linux 6.4.

  This information is all available by using the `--version` option:

  ```
  bash$ perf-report --version

  Arm Performance Reports
  Copyright (c) 2002-2018 Arm Limited (or its affiliates). All
     rights reserved.

  Version: 18.0.2
  Build ID: 556f23c4895e
  Build Platform: Ubuntu 16.04 x86_64
  Build Date: Jan 25 2018 22:42:00

  Frontend OS: Ubuntu 16.04.2 LTS
  Nodes' OS: unknown
  Last connected ddt-debugger: unknown
  ```

- The compiler used and its version number.
- The MPI library and version if appropriate.
- A description of the issue : what you expected to happen and what actually happened.
- An exact copy of any warning or error messages that you may have encountered.

B Supported platforms

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

B.1 Performance Reports

See Arm Performance Reports supported platforms

Platform	Operating Systems	MPI	Compilers
x86_64	Red Hat Enterprise Linux and derivatives 6 and 7, SUSE Linux Enterprise Server 11 and 12, Ubuntu 14.04 and 16.04	Bullx MPI 1.2.7 and 1.2.8, Cray MPT (MPI/SHMEM), Intel MPI 4.1.x and 5.0.x, MPICH 2.x.x and 3.x.x, MVAPICH 2.0 and 2.1, Open MPI 1.6.x, 1.8.x (MPI/SHMEM), 1.10.x and 2.0.x, Platform MPI 9.x, SGI MPT 2.10 and 2.11	Cray, GNU 4.3.2+, Intel 13+, PGI 14+
Arm®v8 (AArch64)	Ubuntu 16.04, SUSE Linux Enterprise Server 12.2, and Red Hat Enterprise Linux 7.4	Open MPI 1.8.x, 1.10.x and 2.0.x	Arm Compiler for HPC, GNU
NVIDIA CUDA Toolkit 7.0/7.5/8.0	Linux	-	

The Arm profiling libraries must be explicitly linked with statically linked programs which mostly applies to the Cray X-Series.

Batch schedulers: SLURM 2.6.3+ and 14.03+ (srun only).

C Known issues

The most significant known issues for the latest release are summarized here:

- I/O metrics are not fully available on some systems, including Cray systems.
- CPU instruction metrics are only available on x86_64 systems.
- Thread activity is not sampled whilst a process is inside an MPI call with a duration spanning multiple samples.
- Xeon Phi systems using many MPI ranks per card should set ALLINEA_REDUCE_MEMORY_USAGE=1.
- Version 14.9 or later of the PGI compilers is required to compile the Arm Performance Reports MPI wrappers as a static library.
- MPICH 3.0.3 and 3.0.4 do not work with MAP, DDT or Performance Reports due to a defect in MPICH 3.0.3/4. MPICH 3.1 addressed this and is fully supported.
- Open MPI 2.1.3 works with Arm Performance Reports. Previous versions of Open MPI 2.1.x do not work due to a bug in the Open MPI debug interface.
- On Cray X-series systems only *native* SLURM is supported, *hybrid* mode is not supported.
- Performance Reports may fail to finalize a profiling session if the cores are oversubscribed on AArch64 architectures. For example when attempting to profile a 64 process MPI program on a machine with only 8 cores. This will appear as a hang after finishing a profile.
-

See also additional known issues here:

Category	**Known Issues**
MPI Distribution	D MPI distribution notes
Compiler	E Compiler notes
Platform	F Platform notes
General	G General troubleshooting

D MPI distribution notes

This appendix has brief notes on many of the MPI distributions supported by Arm Performance Reports.

Advice on settings and problems particular to a distribution are given here.

D.1 Bull MPI

Bull X-MPI is supported.

D.2 Cray MPT

Arm Performance Reports users may wish to read 4.1.3 Static linking on Cray X-Series Systems.

Arm Performance Reports has been tested with Cray XK7 and XC30 systems.

Arm Performance Reports requires Arm's sampling libraries to be linked with the application before running on this platform.

See 4.1.1 Linking for a set-by-step guide.

Arm supplies module files in `REPORTS_INSTALLATION_PATH/share/modules/cray`.

See 4.1.5 Dynamic and static linking on Cray X-Series systems using the modules environment to simplify linking with the sampling libraries.

Known Issues:

- By default scripts wrapping Cray MPT will not be detected, but you can force the detection by setting the `ALLINEA_DETECT_APRUN_VERSION` environment variable to "yes" before starting Performance Reports.

D.3 Intel MPI

Arm Performance Reports has been tested with Intel MPI 4.1.x, 5.0.x and onwards.

Known Issue: If you use Spectrum LSF as workload manager in combination with Intel MPI and you get for example one of the following errors:

- <*target program*> exited before it finished starting up. One or more processes were killed or died without warning
- <*target program*> encountered an error before it initialised the MPI environment. Thread 0 terminated with signal SIGKILL

or the job is killed otherwise during launching then you may need to set/export `I_MPI_LSF_USE_COLLECTIVE_LAUNCH=1` before executing Arm Performance Reports. See Using IntelMPI under LSF quick guide and Resolve the problem of the Intel MPI job . . . hang in the cluster for more details.

D.4 MPICH 2

If you see the error `undefined reference to MPI_Status_c2f` during initialization or if manually building the sampling libraries as described in 4.1.1 Linking, then you need to rebuild MPICH 2 with Fortran support.

D.5 MPICH 3

MPICH 3.0.3 and 3.0.4 do not work with Arm Performance Reports due to an MPICH bug. MPICH 3.1 addresses this and is supported.

D.6 Open MPI

Arm Performance Reports products have been tested with Open MPI 1.6.x, 1.8.x, 1.10.x and 2.0.x.

Open MPI 2.1.3 works with Arm Performance Reports. Previous versions of Open MPI 2.1.x do not work due to a bug in the Open MPI debug interface.

Known issue: If you are using the 1.6.x series of Open MPI configured with the `--enable-orterun-prefix-by-default` flag then Arm Performance Reports requires patch release 1.6.3 or later due to a defect in earlier versions of the 1.6.x series.

D.7 Platform MPI

Platform MPI 9.x is supported, but only with the `mpirun` command. Currently `mpiexec` is not supported.

D.8 SGI MPT / SGI Altix

SGI MPT 2.10+ is supported.

Some SGI systems can not compile programs on the batch nodes, for example because the `gcc` package is not installed.

If this applies to your system you must explicitly compile the Arm MPI wrapper library using the `make-profiler-libraries` command and then explicitly link your programs against the Arm profiler and sampler libraries.

The `mpio.h` header file shipped with SGI MPT 2.10 contains a mismatch between the declaration of `MPI_File_set_view` and some other similar functions and their PMPI equivalents, for example `PMPI_File_set_view`. This prevents Arm Performance Reports from generating the MPI wrapper library. Please contact SGI for a fix.

If you are using SGI MPT with SLURM and would normally use `mpiexec_mpt` to launch your program you will need to use `srun --mpi=pmi2` directly.

Preloading the Arm profiler and MPI wrapper libraries is not supported in Express Launch mode. Arm recommends you explicitly link your programs against these libraries to work around this problem. If this is not possible you can manually compile the MPI wrapper, and explicitly set `LD_PRELOAD` in the launch line.

D.9 SLURM

The use of the `--export` argument to `srun` (SLURM 14.11 or newer) is not supported. In this case you can avoid using `--export` by exporting the necessary environment variables before running Arm Performance Reports.

The use of the `--task-prolog` argument to `srun` (SLURM 14.03 or older) is also not supported, as the necessary libraries cannot be preloaded. You will either need to avoid using this argument, or explicitly link to the libraries.

E Compiler notes

E.1 AMD OpenCL compiler

Not supported by Arm Performance Reports.

E.2 Berkeley UPC compiler

Not supported by Arm Performance Reports.

E.3 Cray compiler environment

The Cray UPC compiler is not supported by Arm Performance Reports.

E.4 GNU

The `-foptimize-sibling-calls` optimization (used in `-O2`, `-O3` and `-Os`) interfere with the detection of some OpenMP regions.

If your code is affected with this issue add `-fno-optimize-sibling-calls` to disable it and allow Arm Performance Reports to detect all the OpenMP regions in your code.

E.4.1 GNU UPC

Arm Performance Reports does not support this.

E.5 Intel compilers

Arm Performance Reports has been tested with versions 13 and 14.

E.6 Portland Group compilers

Arm Performance Reports has been tested with Portland Tools 14 onwards.

F Platform notes

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

F.1 Intel Xeon

Intel Xeon processors starting with Sandy Bridge include Running Average Power Limit (RAPL) counters. Performance Reports can use the RAPL counters to provide energy and power consumption information for your programs.

F.1.1 Enabling RAPL energy and power counters when profiling

To enable the RAPL counters to be read by Performance Reports you must load the `intel_rapl` kernel module.

The `intel_rapl` module is included in Linux kernel releases 3.13 and later.

For testing purposes Arm have backported the `powercap` and `intel_rapl` modules for older kernel releases. You may download the backported modules from:

Download backported modules

Note: These backported modules are unsupported and should be used for testing purposes only. No support is provided by Arm, your system vendor or the Linux kernel team for the backported modules.

F.2 NVIDIA CUDA

- CUDA metrics are not available for statically-linked programs.
- CUDA metrics are measured at the node level, not the card level.

F.3 Arm

F.3.1 Arm®v8 (AArch64) known issues

There are a number of issues you should be aware of:

- Performance Reports does not support CPU time metrics on this platform. Linux perf event metrics are available instead. To ensure access to performance counters is not restricted, use `sysctl -w kernel.perf_event_paranoid=0`.
- Performance Reports may fail to finalize a profiling session if the cores are oversubscribed on AArch64 platforms. For example, this issue is likely to occur when attempting to profile a 64 process MPI program on a machine with only 8 cores. This issue will appear as a hang after finishing a profile.

G General troubleshooting

If you have problems with any of the Arm Performance Reports products, please read this section carefully.

Additionally, check the support pages on the Arm Developer website, and make sure you have the latest version of the product.

G.1 Starting a program

G.1.1 Problems starting scalar programs

There are a number of possible sources for problems. The most common for users with a multi-process license is that the *Run Without MPI Support* check box has not been checked.

If the software reports a problem with MPI and you know your program is not using MPI, then this is usually the cause.

If you have checked this box and the software still mentions MPI then please contact Arm support at Arm support.

Other potential problems are:

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

G.1.2 Problems starting multi-process programs

If you encounter problems while starting an MPI program, the first step is to establish that it is possible to run a single-process (non-MPI) program such as a trivial "Hello, World!", and resolve such issues that may arise. After this, attempt to run a multi-process job and the symptoms will often allow a reasonable diagnosis to be made.

In the first instance verify that MPI is working correctly by running a job, without Arm Performance Reports products applied, such as the example in the `examples` directory.

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

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

Sometimes problems are caused by environment variables not propagating to the remote nodes while starting a job. The solution to these problems depend on the MPI implementation that is being used.

In the simplest case, for rsh-based systems such as a default MPICH 1 installation, correct configuration can be verified by rsh-ing to a node and examining the environment. It is worthwhile rsh-ing with the `env` command to the node as this will not see any environment variables set inside the `.profile` command.

For example, if your nodes use a `.profile` instead of a `.bashrc` for each user then you may well see a different output when running `rsh node env` than when you `run rsh` node and then run `env` inside the new shell.

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

If a large number of processes are reported by the status bar to have connected, then it is possible that some have failed to start due to resource exhaustion, timing out, or, unusually, an unexplained crash. You should verify again that MPI is still working, as some MPI distributions do not release all semaphore resources correctly, for example MPICH 1 on Redhat with SMP support built in.

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

G.1.3 No shared home directory

If your home directory is not accessible by all the nodes in your cluster then your jobs may fail to start.

To resolve the problem open the file `~/.allinea/system.config` in a text editor. Change the `shared directory` option in the `[startup]` section so it points to a directory that is available and shared by all the nodes. If no such directory exists, change the `use session cookies` option to `no` instead.

G.2 Performance Reports specific issues

G.2.1 My compiler is inlining functions

While compilers may inline functions, their ability to include sufficient information to reconstruct the original call tree vary between vendors. Arm has found that the following flags work best:

- Intel: `-g -O3 -fno-inline-functions`
- Intel 17+: `-g -fno-inline -no-ip -no-ipo -fno-omit-frame-pointer -O3`
- PGI: `-g -O3 -Meh_frame`
- GNU: `-g -O3 -fno-inline`
- Cray: `-G2 -O3 -h ipa0`

Note: Some compilers may still inline functions even when explicitly asked not to.

There is typically a small performance penalty for disabling function inlining or enabling profiling information.

Alternatively, you can let the compiler inline the functions and compile with `-g -O3`, or `-g -O5`, or whatever your preferred performance flags are.

Arm Performance Reports will work correctly, but you will often see time inside an inlined function being attributed to its parent in the Stacks view. The Source Code view will be largely unaffected.

Arm Performance Reports will not be affected by function inlining.

G.2.2 Tail recursion optimization

A function may return the result of calling another function, for example:

```
int someFunction()
{
    ...
    return otherFunction();
}
```

In this case the compiler may change the call to `otherFunction` into a jump. This means that, when inside `otherFunction`, the calling function, `someFunction`, no longer appears on the stack.

This optimization is called *tail recursion optimization*. It may be disabled for the GNU C compiler by passing the `-fno-optimize-sibling-calls` argument to `gcc`.

G.2.3 MPI wrapper libraries

Arm Performance Reports wrap MPI calls in a custom shared library. One is built for your system each time you run Arm Performance Reports.

If this does not work please contact Arm support at Arm support.

You can also try setting MPICC directly:

```
$ MPICC=my-mpicc-command bin/perf-report --np=16 ./wave_c
```

G.2.4 Thread support limitations

Performance Reports provides limited support for programs when threading support is set to `MPI_THREAD_SERIALIZED` or `MPI_THREAD_MULTIPLE` in the call to `MPI_Init_thread`.

MPI activity on non-main threads will contribute towards the MPI-time of the program, but not the more detailed MPI metrics.

MPI activity on a non-main thread may result in additional profiling overhead due to the mechanism employed by Performance Reports for detecting MPI activity.

Warnings are displayed when the user initiates and completes profiling a program which sets `MPI_THREAD_SERIALIZED` or `MPI_THREAD_MULTIPLE` as the required thread support.

Performance Reports does support calling `MPI_Init_thread` with either `MPI_THREAD_SINGLE` or `MPI_THREAD_FUNNELED` specified as the required thread support.

It should be noted that the requirements that the MPI specification make on programs using `MPI_THREAD_FUNNELED` are the same as made by Performance Reports: *all MPI calls must be made on the thread that called `MPI_Init_thread`.*

In many cases, multi-threaded MPI programs can be refactored such that they comply with this restriction.

G.2.5 No thread activity while blocking on an MPI call

Unfortunately Arm Performance Reports is currently unable to record thread activity on a process where a long duration MPI call is in progress.

If you have an MPI call that takes a significant amount of time (multiple samples) to complete then Arm Performance Reports will record no thread activity for the process executing that call for most of that MPI call's duration.

G.2.6 I'm not getting enough samples

By default sampling takes place every 20ms initially, but if you get warnings about too few samples on a fast run, or want more detail in the results, you can change that.

To increase the frequency of sampling to every 10ms set environment variable `ALLINEA_SAMPLER_INTERVAL=10`.

Note: Sampling frequency is automatically decreased over time to ensure a manageable amount of data is collected whatever the length of the run.

Note: Whether OpenMP is enabled or disabled in Arm MAP, the final script or scheduler values set for `OMP_NUM_THREADS` will be used to calculate the sampling interval per thread. When configuring your job for submission, check whether your final submission script, scheduler or the Arm MAP GUI has a default value for `OMP_NUM_THREADS`.

Note: Custom values for `ALLINEA_SAMPLER_INTERVAL` will be overwritten by values set from the combination of `ALLINEA_SAMPLER_INTERVAL_PER_THREAD` and the expected number of threads (from `OMP_NUM_THREADS`).

Increasing the sampling frequency is not recommended if there are lots of threads or there are deep stacks in the target program as this may not leave sufficient time to complete one sample before the next sample is started.

G.2.7 Performance Reports is reporting time spent in a function definition

Any overheads involved in setting up a function call (pushing arguments to the stack and so on) are usually assigned to the function definition.

Some compilers may assign them to the opening brace '{' and closing brace '}' instead. If this function has been inlined, the situation becomes further complicated and any setup time (for example, allocating space for arrays) is often assigned to the definition line of the enclosing function.

G.2.8 Performance Reports is not correctly identifying vectorized instructions

The instructions identified as vectorized (packed) are enumerated below. Arm also identifies the AVX-2 variants of these instructions (with a "`v`" prefix).

Contact Arm support at Arm support if you believe your code contains vectorized instructions that have not been listed and are not being identified in the *CPU floating-point/integer vector* metrics.

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`

G.2.9 Performance Reports takes a 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 Arm Performance Reports and `objdump` to see invalid function lengths and addresses.

This causes Arm Performance Reports to take an extremely long time disassembling and analyzing apparently overlapping functions containing millions of instructions.

A fix for this was accepted into the OpenBLAS codebase on October 8th 2013 and versions 0.2.9 and above should not be affected.

To work around this problem without updating OpenBLAS, simply run “strip libopenblas*.so”—this removes the incomplete `.symtab` section without affecting the operation or linkage of the library.

G.2.10 Performance Reports over-reports MPI, I/O, accelerator or synchronization time

Arm Performance Reports employs a heuristic to determine which function calls should be considered as MPI operations.

If your code defines any function that starts with `MPI_` (case insensitive) those functions will be treated as part of the MPI library resulting in the time spent in MPI calls to be over-reported.

Starting your functions names with the prefix `MPI_` should be avoided and is in fact explicitly forbidden by the MPI specification (page 19 sections 2.6.2 and 2.6.3 of the MPI 3 specification document http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf#page=49):

> All MPI names have an MPI_ prefix, and all characters are capitals. Programs must not declare names, for example, for variables, subroutines, functions, parameters, derived types, abstract interfaces, or modules, beginning with the prefix MPI_.

Similarly Arm Performance Reports categorizes I/O functions and accelerator functions by name.

Other prefixes to avoid starting your function names with include `PMPI_`, `_PMI_`, `OMPI_`, `omp_`, `GOMP_`, `shmem_`, `cuda_`, `__cuda`, `cu[A-Z][a-z]` and `allinea_`. All of these prefixes are case-insensitive.

Also avoid naming a function `start_pes` or any name also used by a standard I/O or synchronisation function (`write`, `open`, `pthread_join`, `sem_wait` etc).

G.3 Obtaining support

To receive additional support, contact Arm support at Arm support with a detailed report of the problem you are having.

If possible, you should obtain a log file for the problem and contact Arm support at Arm support. When describing your issue, state that you have obtained a log file and the support team will be in contact.

To generate a log file, start Performance Reports with the `--debug` and `--log` arguments:

```
$ perf-report --debug --log=<log>
```

Where `<log>` is the name of the log file to generate.

Next, reproduce the problem using as few processors as possible. Once finished, close the program as usual.

On some systems this file may be quite large. If so, please compress it using a program such as `gzip` or `bzip2` before sending it to support.

If your problem can only be replicated on large process counts, then omit the `--debug` argument as this will generate very large log files.

G.4 Arm IPMI Energy Agent

The Arm IPMI Energy Agent allows Arm MAP and Arm Performance Reports to measure the total energy consumed by the compute nodes in a job in conjunction with the Arm Advanced Metrics Pack add-on.

The IPMI Energy Agent is a separate download from our website: IPMI Energy Agent.

G.4.1 Requirements

- The compute nodes must support IPMI.
- The compute nodes must have an IPMI exposed power sensor.
- The compute nodes must have an OpenIPMI compatible kernel module installed, such as `ipmi_devintf`.
- The compute nodes must have the corresponding device node in `/dev`, for example `/dev/ipmi0`.
- The compute nodes must run a supported operating system.
- The IPMI Energy Agent must be run as root.

To list the names of possible IPMI power sensors on a compute node use the following command:

```
ipmitool sdr | grep 'Watts'
```


Index

