Allinea Performance Reports User Guide

Version 6.1

Contents

1 Introduction

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

Allinea 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 well-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 we make to improve performance further?

It is based on Allinea 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 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 http://www.allinea.com/knowledge-center/training

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

Support http://www.allinea.com/knowledge-center/get-support

2 Installation

A release of Allinea Performance Reports may be downloaded from the Allinea website: http://www.allinea.com.

Both a graphical and text-based installer are provided—see the sections below for details.

2.1 Linux/Unix Installation

2.1.1 Graphical Install

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

```
tar xf allinea-reports-6.1-ARCH.tar
cd allinea-reports-6.1-ARCH
./installer
```

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

The *Install Type* page lets you choose which user(s) to install Allinea Performance Reports for. If you are an administrator (`root`) you may install Allinea Performance Reports for *All Users* in a common directory such as `/opt` or `/usr/local`, otherwise only the *Just For Me* option is enabled.
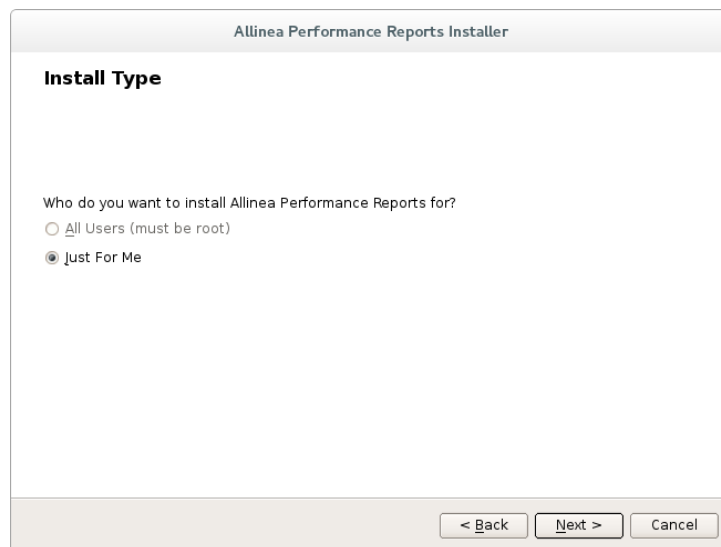
Figure 1: *Allinea Performance Reports Installer—Installation type*

Once you have selected the installation type, you will be asked which directory you would like to install Allinea Performance Reports in. If you are installing on a cluster, make sure you choose a directory that is shared between the cluster login node / frontend and the cluster nodes. Otherwise you must install or copy it to the same location on each node.

Allinea Performance Reports 6.1
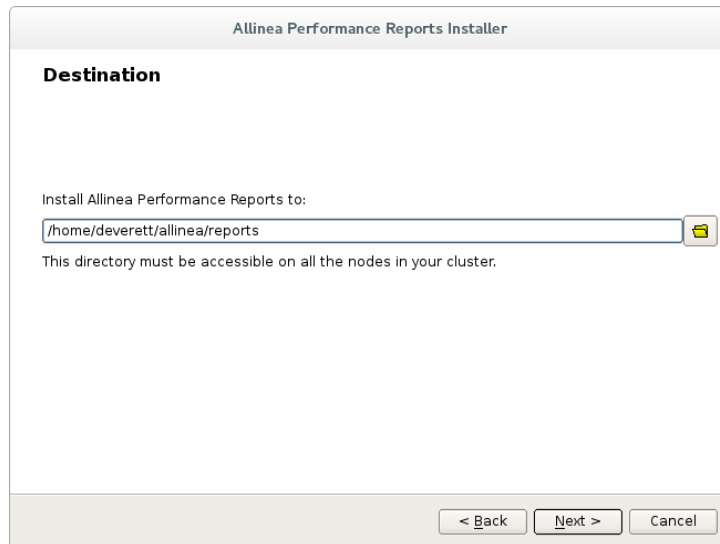
Figure 2: *Allinea Performance Reports Installer—Installation directory*

You will be shown the progress of the installation on the *Install* page.
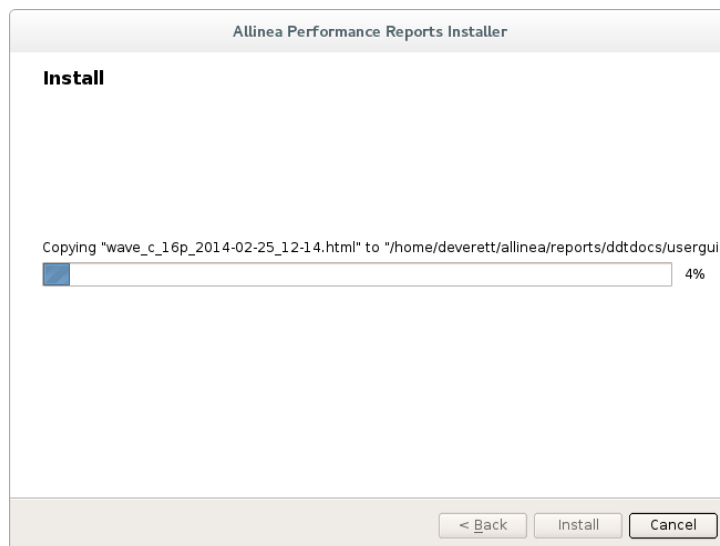
Figure 3: *Install in progress*

Performance Reports does not have a GUI and will 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 will need a valid licence file. You can obtain an evaluation licence by completing the form at http://www.allinea.com/products/performance-reports/free-trial.

Due to the vast number of different site configurations and MPI distributions that are supported by Allinea Performance Reports, it is inevitable that sometimes you may need to take further steps to get the 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.

```
tar xf allinea-reports-<unknown>-ARCH.tar
cd allinea-reports-<unknown>-ARCH
./text-install.sh
```

Press *Return* to read the licence when prompted and then enter the directory where you would like to install Allinea Performance Reports. The directory must be accessible on all the nodes in your cluster.

2.2 Licence Files

Allinea Performance Reports requires a licence file for its operation.

Time-limited evaluation licences are available from the Allinea website: http://www.allinea.com.

2.3 Workstation and Evaluation Licences

Workstation and Evaluation licence files for Allinea Performance Reports do not need a licence server and should be copied directly to {`installation-directory`}`/licences` (e.g. `/home/user/allinea/reports/licences/Licence.ddt`). Do not edit the files as this will prevent them working.

You may specify an alternative location of the licence 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 Licences

For users with Supercomputing and other floating licences, the Allinea Licence Server must be running on the designated licence server machine prior to running Allinea Performance Reports.

The Allinea Licence Server and instructions for its installation and usage may be downloaded from http://www.allinea.com/downloads.

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

2.5 Architecture Licensing

Licences issued after the release of Allinea Performance Reports 6.1 specify the compute node architectures that they may be used with. Licences issued prior to this release will enable the x86_64 and i686 architectures by default. Existing users for other architectures will be supplied with new licences that will enable their architectures. If there is any problem then contact support@allinea.com .

3 Running with an Example Program

This section will take you through compiling and running one of the the provided example programs.

3.1 Overview of the Example Source Code

3.2 Compiling

Allinea provides a simple 1-D wave equation solver that's 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:

```
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.

Depending on the default compiler on your system you may see some errors when running this, for example:

```
pgf90-Error-Unknown switch: -fno-inline
```

Our example makefile is set up for the GNU compilers by default. There are lines in `examples/wave.makefile` that you can uncomment to enable support for other compilers. In the above case, to enable PGI compiler support you can simply switch the commented lines:

```
# gnu
#        ${MPICC} -g -O3 -fno-inline wave.c -o wave_c -lm -lrt
#        ${MPIF90} -g -O3 -fno-inline wave.f90 -o wave_f -lm -lrt
# intel
#        ${MPICC} -g -fno-inline-functions -O3 wave.c -o wave_c -lm
    -lrt
#        ${MPIF90} -g -fno-inline-functions -O3 wave.f90 -o wave_f
   -lm -lrt
# pgi
         ${MPICC} -g -O3 wave.c -o wave_c -lm -lrt -Meh_frame
         ${MPIF90} -g -O3 wave.f90 -o wave_f -lm -lrt -Meh_frame
```

Note that although these example Makefiles include the `-g` flag, Performance Reports does *not* need this and you should not use them in your own Makefiles. In most cases 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 Allinea 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 Allinea 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 Allinea 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/allinea/forge/doc/.
```

Then follow the instructions in the output to link the example program with the Allinea 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 will need 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, like this:

```
$ qsub -I
qsub: waiting for job 31337 to start
qsub: job 31337 ready
$ cd allinea/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 Allinea Performance Reports module for your system has been loaded:

```
$ perf-report --version
Allinea Performance Reports
Part of Allinea Performance Reports.
(c) Allinea Software Ltd 2002-2015
...
```

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 Allinea 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 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.

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 re-linked—this is explained in section 4.1.2 Linking below.

4.1.1   .eh-frame-hdr section

For statically-linked programs, you may need to compile with extra flags to ensure that the executable still has all the information Performance Reports needs to record the call path and gather the data needed for the Parallel Stack View. For the GNU linker this means adding `--Wl,--eh-frame-hdr` to the compile line, or just `--eh-frame-hdr` to the link line:

```
mpicc hello.c -o hello -g -Wl,--eh-frame-hdr
```

4.1.2   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 us to place an extra library into your program when starting it.

Note: Although these libraries contain the word 'map' they are used for both Performance Reports and 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 F.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 4.1.3 Dynamic Linking on Cray X-Series Systems, or 4.1.4 Static Linking and 4.1.5 Static Linking on Cray X-Series Systems.

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

4.1.3   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 Allinea MPI Wrapper Library

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

4.1.4   Static Linking

If you compile your program statically (i.e. your MPI uses a static library or you pass the -static option to the compiler) then you must explicitly link your program with the Allinea sampler and MPI wrapper libraries.

Compiling the Allinea MPI Wrapper Library

First you must compile the Allinea MPI wrapper library for your system using the make-profiler-libraries --lib-type=static command. Note that 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 Allinea 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/allinea/forge/doc/.
```

Linking with the Allinea 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 Allinea sampler. Usually this is sufficient, but not in all cases. The rest of this section explains how to manually add the Allinea 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 an 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 Allinea MPI wrapper as a shared library as described in (ommitting the `--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 (e.g. `-lmylibrary`).

3. *After* the path to the profiler libraries (`-L/path/to/profiler-libraries`).

4. *Before* the MPI's Fortran wrapper library, if any (e.g. `-lmpichf`).

5. *Before* the MPI's implementation library (usually `-lmpi`).

6. *Before* the Allinea sampler library (`-lmap-sampler`).

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

1. *After* the Allinea MPI wrapper library.

2. *After* your program's object (`.o`) files.

3. *After* your program's own static libraries (e.g. `-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
```

MVAPICH 1

You must add `-lfmpich` after `-lmap-sampler-pmpi` (MVAPICH must be compiled with Fortran support).

If you get a linker error about multiple definitions of `mpi_init_`, you need to specify additional linker flags:

```
-Wl,--allow-multiple-definition
```

4.1.5 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 Allinea 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/allinea/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
```

4.2 Express Launch Mode

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 a 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 Allinea 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:

- BlueGene/Q
- 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 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 (e.g. using `ssh`).

To use Compatibility Mode replace the `mpiexec` command with the `perf-report` command. For example:

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

would become:

```
perf-report --np=256 ./wave_c 20
```

Only a small number of `mpiexec` arguments are supported by perf-report (e.g. `-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 Allinea Performance Reports module for your system has been loaded:

```
$ perf-report --version
Allinea Performance Reports
Part of Allinea Performance Reports.
(c) Allinea Software Ltd 2002-2015
...
```

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 Allinea 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 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 text-format report in the file `my-report.txt` in the current directory

- `--output=/home/mark/public/my-report.html` will create a HTML report in the file `/home/mark/public/my-report.html`

- `--output=my-report` will create a text-format report in the file `my-report.txt` and a HTML report in the file `my-report.html`, both in the current directory

- `--output=/tmp` will create automatically-named reports based on the application executable name in /tmp/, e.g. `/tmp/wave_f_16p_2013-11-18_2330.txt` and `/tmp/wave_f_16p_-2013-11-18_2330.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 will be called and the collected metrics will be sent to Ganglia (a System Monitoring tool).

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

4.6.1 Customising 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 `ALLINE_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) will specify that the metric `METRIC` has the value `VALUE`

- `-U{METRIC}={UNITS}` (optional) will specify that the metric `METRIC` is expressed in `UNITS`

- `-T{METRIC}={TITLE}` (optional) will specify that the metric `METRIC` has title `TITLE`

- `-t{METRIC}={TYPE}` (optional) will specify 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) will specify the name of the metric (starts with `com.allinea`)

- `-t` {TYPE} (mandatory) will specify the type of the metric (i.e. `double` or `int32`)

- `-v` {VALUE} (mandatory) will specify the value of the metric

- `-g` {GROUP} (optional) will specify which groups the metric belongs to (i.e. `allinea`)

- `-u` {UNIT} (optional) will specify the unit of the metric (e.g. `%`, `Watts`, `Seconds`, etc.)

- `-T` {TITLE} (optional) will specify the title of the metric

5 Summarizing an Existing MAP File

Performance Reports can be used to summarize an application profile generated by Allinea MAP. To produce a performance report from an exisiting 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 (e.g. Performance Reports 6.1 with MAP 6.1). Performance Reports can use MAP files from versions of MAP as old as 5.0.

6   Interpreting Performance Reports

This section will take you interpreting the reports produced by 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; the HTML version is easier to read and visually compare while the textual version is better suited to quick checks from the terminal. If you wish to combine Performance Reports with other tools, consider using the CSV output format—see 6.12 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:allinea/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:

Command: /shared_scratch/xavier/code/ddt/examples
/wave_openmp 60
Resources: 1 node (8 physical, 8 logical cores per node)
Memory: 15 GB per node
Tasks: 8 processes, OMP_NUM_THREADS was 2
Machine: mars
Start time: Tue Jun 23 14:36:05 2015
Total time: 61 seconds (1 minute)
Full path: /shared_scratch/xavier/code/ddt/examples
Input file:
Notes:

Compute

MPI   I/O
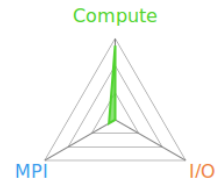
Summary: wave_openmp is Compute-bound in this configuration

Compute   91.2%   Time spent running application code. High values are usually good.
This is **very high**; check the CPU performance section for advice.

MPI   8.8%   Time spent in MPI calls. High values are usually bad.
This is **very 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 very little time is spent in MPI calls, this code may also benefit from running at larger scales.

CPU

A breakdown of the 91.2% CPU time:

Single-core code	30.6%
OpenMP regions	69.4%
Scalar numeric ops	9.5%
Vector numeric ops	0.0%
Memory accesses	78.1%

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.
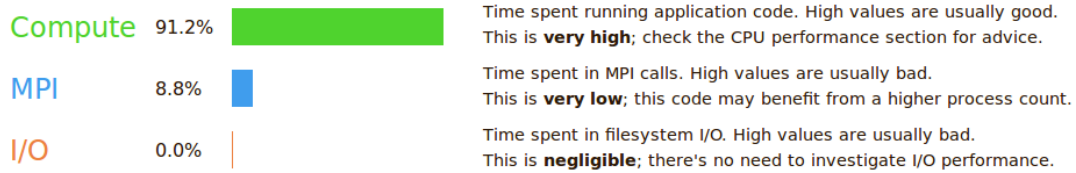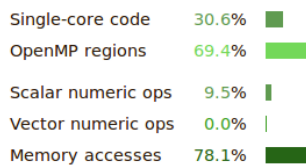
MPI

A breakdown of the 8.8% MPI time:

Time in collective calls	4.0%
Time in point-to-point calls	96.0%
Effective process collective rate	66.3 kB/s
Effective process point-to-point rate	312 kB/s
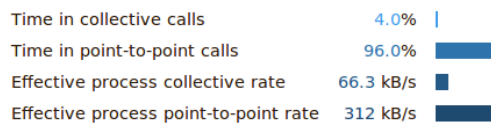
Most of the time is spent in point-to-point calls with a very low transfer rate. This suggests load imbalance is causing synchronization overhead; use an MPI profiler to investigate.

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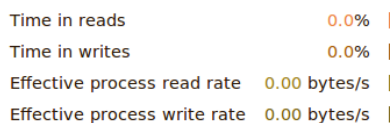
No time is spent in I/O operations. There's nothing to optimize here!

OpenMP

A breakdown of the 69.4% time in OpenMP regions:

Computation	32.7%
Synchronization	67.3%
Physical core utilization	100.0%
System load	116.3%
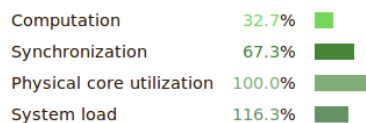
Significant time is spent synchronizing threads in parallel regions. Check the affected regions with a profiler.

This may be a sign of overly fine-grained parallelism (OpenMP regions in tight loops) or workload imbalance.

Memory

Per-process memory usage may also affect scaling:

Mean process memory usage	39.9 MB
Peak process memory usage	46.3 MB
Peak node memory usage	7.0%
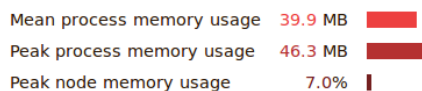
The peak node memory usage is very low. Running with fewer MPI processes and more data on each process may be more efficient.

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 turn below.

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 we see that 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 generally 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. We've found it helpful to recognize and compare these triangular shapes when flicking 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 I/O

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 done under the covers by the MPI library. `MPI_File_close` is treated as time spent writing, which is often but not always correct.

6.3 CPU Breakdown

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/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, mull, 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's 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 attempt to access the same cache lines and the over-use of the `atomic` pragma will 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   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.

6.4.1   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 e.g. floating-point operations or whether the cores are mostly waiting for memory accesses.

6.4.2   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.

6.4.3   Physical core utilisation

Modern CPUs often have multiple *logical* cores for each *physical* cores; this is often referred to as hyperthreading. 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.

6.4.4   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 hyperthreading, if the cores are *oversubscribed*, or if other system processes and daemons start running and take CPU resources away from your program. A value consistently less than 100% may indicate your program is not taking full advantage of the CPU resources available on a compute node.

6.5   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.

6.5.1   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 e.g. floating-point operations or whether the cores are mostly waiting for memory accesses.

6.5.2   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 Allinea MAP.

6.5.3   Physical core utilisation

Modern CPUs often have multiple *logical* cores for each *physical* cores; this is often referred to as hyperthreading. 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 utilisation of physical cores - a value over 100% indicates that more threads are executing than there are physical cores, e.g. that hyperthreading is in use. A program may have dozens of helper threads that do little except sleeping and these will not be shown here. Only threads actively and simultaneously consuming CPU time are included in this metric.

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 hyperthreading, if the cores are *oversubscribed*, or if other system processes and daemons start running and take CPU resources away from your program. A value consistently less than 100% may indicate your program is not taking full advantage of the CPU resources available on a compute node.

6.6   MPI Breakdown

This section breaks down the time spent in MPI calls reported in the summary. It's 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—we are measuring the rate of communication from the process to the MPI API and not of the underlying hardware directly. This application-perspective is found throughout Performance Reports and in this case allows the results to capture effects such as faster intra-node performance, zero-copy transfers and so on.

6.6.1   Time in collective calls

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

6.6.2   Time in point-to-point calls

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

6.6.3 Estimated 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.

6.6.4 Estimated 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 communicaton and computation is often a good strategy to improve application performance and scalability.

6.7 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.

6.7.1 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.

6.7.2 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 our measurements have shown that current-generation networked filesystems can spend significant amounts of time opening files with create or write permissions.

6.7.3 Estimated 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.

6.7.4   Estimated 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   Memory Breakdown

Unlike the other sections, the memory section does not refer to one particular portion of the job. Rather, it summarizes memory usage across all processes and nodes over the entire duration. All of these metrics refer to RSS, i.e. physical RAM usage and not virtual memory usage. Most HPC jobs try very hard to stay within the physical RAM of their node for performance reasons.

6.8.1   Mean process memory usage

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

6.8.2   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.

6.8.3   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 occuring, 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.

6.9   Accelerator Breakdown

Figure 5: *Accelerator metrics report*

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

6.9.1   GPU utilization

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

6.9.2   Global memory accesses

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

6.9.3   Mean GPU memory usage

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

6.9.4   Peak GPU memory usage

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

6.10   Energy Breakdown
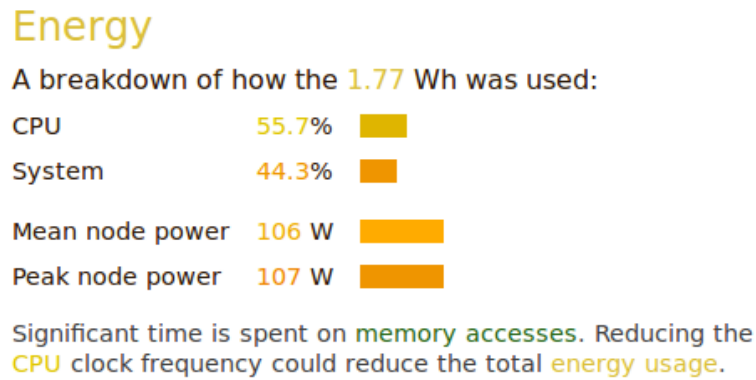
Figure 6: *Energy metrics report*

This section shows the energy used by the job, broken down by component (e.g. CPU and accelerators).

6.10.1   CPU

The percentage of the total energy used by the CPUs.

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

6.10.2   Accelerator

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

6.10.3   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%.

6.10.4   Mean node power

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

6.10.5   Peak node power

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

6.10.6 Requirements

CPU power measurement requires an Intel CPU with RAPL support, e.g. 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 Allinea IPMI energy agent which can read IPMI power sensors; or the Cray HSS energy counters. For more information on how to install the Allinea IPMI energy agent please see F.4 Allinea 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.

6.11 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/allinea/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 very useful for quickly extracting and comparing values between multiple runs, or for automatically placing such data into a centralized database.

6.12 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 well suited for feeding to an automated analysis tool, such as a plotting program. Also can be imported into a spreadsheet for analysing values among executions.

6.13   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 our website.

At the time of writing there are three collections available:

6.13.1   Code characterization and run size comparison

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

http://allinea.com/products/performance/characterization-of-hpc-codes-and-problems/

6.13.2   Deeper CPU metric analysis

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

http://allinea.com/products/performance/exploring-hyperthreading/

6.13.3   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:

http://allinea.com/products/performance/understanding-i-o-behavior/

7 Configuration

Allinea Performance Reports generally requires no configuration before use. If you only intend to use 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 Allinea Performance Reports needs to access another machine as part of starting one of *MPICH 1–3*, *Intel MPI*, and *SGI MPT*, it will attempt to use the secure shell, ssh, by default.

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

Allinea Performance Reports will use look for the script at ~/.allinea/remote-exec, and it will be executed as follows:

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

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

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
```

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

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

Windows The functionality described above is also provided by the Windows remote client. There are however two differences:

- The script is named remote-exec.cmd rather than remote-exec.
- The default implementation uses the plink.exe executable supplied with Allinea Performance Reports.

A   Getting Support

Whilst 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. In any event, the support team at Allinea will be able to help and will look forward to assist in ensuring that you can get the most out of the Allinea Performance Reports products.

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

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

- Version number of Allinea Performance Reports (e.g. `perf-report --version`) and your operating system and the distribution (example: Red Hat Enterprise Linux 6.4). This information is all available by using the `--version` option on the command line of any Allinea tool:

  ```
  bash$ perf-report --version

  Allinea Performance Reports
  Part of Allinea Performance Reports.
  (c) Allinea Software 2002-2015

  Version: 5.0
  Build: Ubuntu 12.04 x86_64
  Build Date: Jan 5  2015

  Licence Serial Number: see About window

  Frontend OS: Ubuntu 14.04 x86_64
  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 Allinea website. It is likely that MPI distributions supported on one platform will work immediately on other platforms.

B.1   Performance Reports

See http://www.allinea.com/products/performance/platforms/

Platform	Operating Systems	MPI	Compilers
x86_64	Red Hat Enterprise Linux and derivatives 5, 6 and 7, SUSE Linux Enterprise 11 and 12, Ubuntu 12.04 and 14.04	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 and 1.8.x (MPI/SHMEM), Platform MPI 9.x, SGI MPT 2.10 and 2.11	Cray, GNU 4.3.2+, Intel 13+, PGI 14+
NVIDIA CUDA Toolkit 6.0/6.5/7.0/7.5/8.0	Linux	-	

The Allinea 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 MPI Distribution Notes

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

C.1 Bull MPI

Bull X-MPI is supported.

C.2 Cray MPT

Performance Reports users may wish to read 4.1.4 Static Linking on Cray X-Series Systems.

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

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.

C.3 Intel MPI

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

C.4 MPICH 2

If you see the error `undefined reference to MPI_Status_c2f` during initialization or if manually building the sampling libraries (4.1.2 Linking) then you need to rebuild MPICH 2 with Fortran support.

C.5 MPICH 3

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

C.6 Open MPI

Allinea Performance Reports products have been tested with Open MPI 1.4.x, 1.6.x and 1.8.x. Select *Open MPI* from the list of MPI implementations.

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

C.7 Platform MPI

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

C.8 SGI MPT / SGI Altix

SGI MPT 2.10+ is supported.

Note that support for SGI MPT scalable start-up was removed in version 4.2.1-38188 and will be re-added in a future release.

Some SGI systems can not compile programs on the batch nodes (e.g. because the `gcc` package is not installed). If this applies to your system you must explicitly compile the Allinea MPI wrapper library using the `make-profiler-libraries` command and then explicitly link your programs against the Allinea 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, e.g. `PMPI_File_set_view`. This prevents 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 Allinea profiler and MPI wrapper libraries is not supported in express launch mode. We recommend 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.

C.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 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.

D Compiler Notes

D.1 AMD OpenCL compiler

Not supported by Performance Reports.

D.2 Berkeley UPC Compiler

Not supported by Performance Reports.

D.3 Cray Compiler Environment

The Cray UPC compiler is not supported by Performance Reports.

D.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 Performance Reports to detect all the OpenMP regions in your code.

D.4.1 GNU UPC

Performance Reports do not support this.

D.5 Intel Compilers

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

D.6 Portland Group Compilers

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

E   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.

E.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.

E.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 Allinea have backported the `powercap` and `intel_rapl` modules for older kernel releases. You may download the backported modules from:

http://content.allinea.com/downloads/allinea-powercap-backport-20150601.tar.bz2

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

E.2   Intel Xeon Phi

Performance Reports does not presently support Intel Xeon Phi. However, the host side of applications that use offload mode can still use Performance Reports.

E.3   NVIDIA CUDA

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

F   General Troubleshooting

If you have problems with any of the Allinea Performance Reports products, please take a look at the topics in this section—you might just find the answer you're looking for. Equally, it's worth checking the support pages on http://www.allinea.com and making sure you have the latest version.

F.1   Starting a Program

F.1.1   Problems Starting Scalar Programs

There are a number of possible sources for problems. The most common is—for users with a multi-process licence—that the *Run Without MPI Support* check box has not been checked. If 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 we would very much like to hear from you!

Other potential problems are:

- A previous Allinea session is still running, or has not released resources required for the new session. Usually this can be resolved by killing stale processes. The most obvious symptom of this is a delay of approximately 60 seconds and a message stating that not all processes connected. You may also see, in the terminal, a `QServerSocket` message

- The target program does not exist or is not executable

- Allinea Performance Reports products' backend daemon—`ddt-debugger`—is missing from the `bin` directory—in this case you should check your installation, and contact Allinea for further assistance.

F.1.2   Problems Starting Multi-Process Programs

If you encounter problems whilst 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 Allinea 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 whilst starting a job. To a large extent, the solution to these problems depend on the MPI implementation that is being used. In the simplest case, for rsh based systems such as a default MPICH 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 Allinea for advice.

If a large number of processes are reported by the status bar to have connected, then it is possible that some have failed to start due to resource exhaustion, timing out, or, unusually, an unexplained crash. You should verify again that MPI is still working, as some MPI distributions do not release all semaphore resources correctly (for example MPICH 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 Allinea for further long-term advice.

F.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.

F.2   Performance Reports specific issues

F.2.1   My compiler is inlining functions

Yes, they do that. Unfortunately their abilities to include sufficient information to reconstruct the original call tree vary between vendors. We've found that the following flags work best:

- Intel: -g -O3 -fno-inline-functions
- PGI: -g -O3 -Meh_frame
- GNU: -g -O3 -fno-inline

Be aware that some compilers may still inline functions even when explicitly asked not to.

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

Alternatively, you can let the compiler inline the functions and just compile with `-g -O3`. Or `-g -O5` or whatever your preferred performance flags are. Performance Reports will work just fine, but you will often see time inside an inlined function being attributed to its parent in the Stacks view. The Source Code view should be largely unaffected.

Performance Reports should not be affected by function inlining.

F.2.2   Tail Recursion Optimization

If a function returns the result of calling another function, for example:

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

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 pasing the `-fno-optimize-sibling-calls` argument to `gcc`.

F.2.3   MPI Wrapper Libraries

Performance Reports wrap MPI calls in a custom shared library. We build one, just for your system, each time you run Performance Reports. Sometimes it won't work. If it doesn't, please tell us. It should work on every system we've ever seen, first time, every time. In the meantime, you can also try setting MPICC directly:

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

F.2.4   Thread support restrictions

Performance Reports requires that all MPI calls in the program being profiled are made on the same thread as the one that called `MPI_Init` or `MPI_Init_thread`. To enforce this restriction, Performance Reports will abort if `MPI_Init_thread` is called with either `MPI_THREAD_SERIALIZED` or `MPI_THREAD_MULTIPLE` specified as the required thread support. The error displayed is:

The Allinea Sampler does not support the use of MPI with thread support set to MPI_THREAD_SERIALIZED or MPI_THREAD_MULTIPLE (as specified in the call to MPI_Init_thread). All MPI communication must take place on the main thread for profiling to be successful.

The supported arguments to MPI_Init_thread are MPI_THREAD_SINGLE (only one thread will execute) and MPI_THREAD_FUNNELED (if the process is multithreaded, only the main thread will make MPI calls). If in doubt use MPI_THREAD_FUNNELED.

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.

F.2.5   No thread activity whilst blocking on an MPI call

Unfortunately 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 Preformance Reports will record no thread activity for the process executing that call for most of that MPI call's duration.

F.2.6   I'm not getting enough samples

By default we start sampling every 20ms, but if you get warnings about too few samples on a fast run, or want more detail in the results, you can change that. To increase the frequency to every 10ms set

environment variable `ALLINEA_SAMPLER_INTERVAL=10`. Note that the sampling frequency is automatically decreased over time to ensure a manageable amount of data is collected whatever the length of the run. Increasing the sampling frequency is not recommended if there are lots of threads and/or very deep stacks in the target program as this may not leave sufficient time to complete one sample before the next sample is started.

F.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 etc) 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 (e.g. allocating space for arrays) is often assigned to the definition line of the enclosing function.

We're looking for ways to unravel this and present a more intuitive picture; any ideas or suggestions are much appreciated!

F.2.8 Performance Reports is not correctly identifying vectorized instructions

The instructions identified as vectorized (packed) are enumerated below. We also identify the AVX-2 variants of these instructions (with a "v" prefix). Contact support@allinea.com 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`

F.2.9 MAP harmless linker warnings on Xeon Phi

When explicitly linking with `libmap-sampler-pmpi.so` generated using `make-profiler-libraries --platform=xeon-phi` you may see the following compiler warnings:

```
x86_64-k1om-linux-ld: warning: libimf.so, needed by ./libmap-
    sampler-pmpi.so, not found (try using -rpath or -rpath-link)
x86_64-k1om-linux-ld: warning: libsvml.so, needed by ./libmap-
    sampler-pmpi.so, not found (try using -rpath or -rpath-link)
x86_64-k1om-linux-ld: warning: libirng.so, needed by ./libmap-
    sampler-pmpi.so, not found (try using -rpath or -rpath-link)
x86_64-k1om-linux-ld: warning: libintlc.so.5, needed by ./libmap-
    sampler-pmpi.so, not found (try using -rpath or -rpath-link)
```

These warnings are harmless and may be ignored but you must ensure that the Xeon Phi Intel runtime libraries are in your LD_LIBRARY_PATH when running your program.

F.2.10 Performance Reports harmless error messages on Xeon Phi

When running Performance Reports on a Xeon Phi host, where the Performance Reports installation has been configured for E.2 Intel Xeon Phi heterogeneous support, but your MPI program was compiled without MIC options, you may see harmless 'ERROR' messages similar to the following:

```
Other: ERROR: ld.so: object '/home/user/.allinea/wrapper/libmap-sampler-
pmpi-mic3-mic-115427.so' from LD_PRELOAD cannot be preloaded:  ig-
nored.
```

These may be safely ignored.

F.2.11 Performance Reports takes an extremely long time to gather and analyze my OpenBLAS-linked application

OpenBLAS versions 0.2.8 and earlier incorrectly stripped symbols from the .symtab section of the library, causing binary analysis tools such as Allinea Performance Reports and objdump to see invalid function lengths and addresses.

This causes 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.

F.2.12 MAP over-reports MPI, I/O, accelerator or synchronisation time

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, e.g., for variables, subroutines, functions, parameters, derived types, abstract interfaces, or modules, beginning with the prefix MPI_.

Similarly Performance Reports categorises 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).

F.3   Obtaining Support

If this guide hasn't helped you, then the most effective way to get support is to email us with a detailed report. If possible, you should obtain a log file for the problem and email this to support@allinea.com .

You can generate a log file by starting Performance Reports with the `--debug` and `--log` arguments:

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

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

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

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

F.4   Allinea IPMI Energy Agent

The Allinea IPMI Energy Agent allows Allinea MAP and Allinea Performance Reports to measure the total energy consumed by the compute nodes in a job in conjunction with the Allinea Energy Pack add-on. The IPMI Energy Agent is a separate download from our website: http://www.allinea.com/ipmi-energy-agent.

F.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 (i.e. `ipmi_-devintf`).

- The compute nodes must have the corresponding device node in `/dev` (e.g. `/dev/ipmi0`).

- The compute nodes must run a supported operating system.

- The IPMI Energy Agent must be run as root.

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

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

Index