Allinea Performance Reports Quick Start Guide

Installation to enlightenment in just 15 minutes

Allinea Performance Reports runs your unmodified application and produces a .txt and .html report file describing the run in the current working directory.

This guide takes you through downloading, installing, running and understanding the output from Performance Reports in the time it takes to enjoy a good coffee.

The Very Quick Version

If you want to get started on a system with **dynamically-linked MPI programs on non-Cray systems**, this is all you need to do. If you are running on a Cray or are using statically-linked programs, check the section “Manual Linking for Cray Systems and Static Binaries” to see the extra steps required.

1. Installation: extract the tarball and run the installer. You can install Performance Reports into your own user directory; there is no need for root access. The installation directory *does* need to be available on the nodes, however:

```
tar xf allinea-tools-*.tar
allinea-tools-*/textinstall.sh
```

2. Licence: copy the Licence file sent to you into <INSTALL_DIR>:

```
cp Licence ~/allinea/tools/
```

3. Compile the example program: load any module files you normally use for MPI compilation. If your system uses “mpicc” and has a dynamically-linked MPI then this will work:

```
cd ~/allinea/tools/examples
make -f wave.makefile
```

Statically-linked MPIs require that you also link Allinea's performance sampling libraries with the application. These are shared with Allinea MAP. To create these libraries, run:

```
~/allinea/tools/bin/make-map-static-libraries
```

from your application's working directory and follow the instructions displayed. For Cray systems, there is the separate make-map-static-cray-libraries script to use instead.

4a. Run the example program: You may need to request an interactive session on the cluster first, or even create a queue submission file that runs wave_c and then submit that via e.g. llsubmit or qsub. At some point you execute an mpirun / mpiexec command, like this:

```
qsub -I

...

mpiexec -n 4 ./wave_c
```

4b. Run with Performance-Reports: put "<INSTALL_DIR>/bin/perf-report" in front of your mpiexec / mpirun command, e.g:

```
~/allinea/tools/bin/perf-report mpiexec -n 4 wave_c
```

If you needed to create a submission file, find the mpiexec/mpirun line in that and put <INSTALL_DIR>/perf-report in front of it, then submit the file again.

5a. View test report in beautiful HTML: The report is placed in the current working directory. The name is prefixed with the program name - "ls -lrt" will show it on the last line of output. Open the HTML version in a browser (you can also copy it to your laptop first):

```
firefox wave_c_*.html
```

5b. View test report in text mode: This can be handy if you don't have a shared filesystem visible from your desktop or laptop and just want a quick look at the data:

```
less wave_c_*.txt
```

A More Detailed Step-by-Step Overview

Step 1: Download and install Performance Reports

This Performance Reports release is only available for RHEL-6 and SLES-11 compatible x86_64 (AMD / Intel) systems. Make sure you download and install the correct tarball. The easiest way to do this is to check which OS version is running on your machine:

```
$ lsb_release -d || cat /etc/redhat-release
lsb_release: Command not found.
Scientific Linux SL release 5.5 (Boron)
```

In addition to the text-based "textinstall.sh" there is a graphical installer, but as you're probably running this on a cluster with slow and laggy X forwarding you're best sticking to the text-based one.

Step 2: Install a licence

The email you received with your trial should have contained an attachment called "Licence". Put this into the installation directory, e.g. ~/allinea/tools/. If you have any problems with this, contact support@allinea.com for assistance.

Step 3: Compile the example program

Now we're going to build the example programs. If you need to load any MPI or compiler modules, do so now.

```
$ cd ~/allinea/tools/examples/
$ make -f wave.makefile
```

Depending on the default compiler on your system, you may see some errors here, like these:

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

Our examples makefiles are set up for the GNU compilers by default. There are lines in wave.makefile that you can uncomment to enable support for other compilers, but switching modules is often quickest. On this system I modified examples/wave.makefile to use the PGI options by swapping the comment characters to this:

```
# 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 -Mprof=func -O3 wave.c -o wave_c -lm -lrt
        ${MPIF90} -g -Mprof=func -O3 wave.f90 -o wave_f -lm -lrt
$ make -f wave.makefile
mpicc -g -Mprof=func -O3 wave.c -o wave_c -lm -lrt
mpif90 -g -Mprof=func -O3 wave.f90 -o wave_f -lm -lrt
```

All done!

Step 4a: Run the example program (without Performance Reports)

This checks that everything is set up correctly; it only takes 30 seconds.

When running an MPI program you usually need to request access to a node. The simplest way to do this is to ask for an interactive session, if your site has these available:

```
$ qsub -I
qsub: waiting for job 392232 to start
qsub: job 392232 ready
$ cd allinea/tools/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
```

Check that you see something similar to this before running Performance Reports.

Step 4b: Run with Performance Reports

Now comes the good bit! However you ran the example program in step 4a, repeat the same procedure but put “perf-report” in front of the mpiexec / mpirun command, e.g.:

```
$ ~/allinea/tools/bin/perf-report mpiexec -n 4 ./wave_c 5
Wave solution running with 4 processes

0: points = 1000000, running for 5 seconds
points / second: 63.7M (15.9M per process)
compute / communicate efficiency: 94% | 95% | 98%

Points for validation:
0:0.00  200000:0.95  400000:0.59  600000:-0.59  800000:-0.95
999999:0.00
wave finished
```

Excellent, it looks just the same as before! The eagle-eyed will have noticed the performance of the application has dropped by 0.2M points per second, a decrease of just 0.29%. This is fairly typical of the impact Allinea Performance Reports has on most codes.

The only difference with this run is that a performance report has been 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
```

Step 5: Understanding the Performance Reports

Two reports are generated – both text-based and an HTML-based. The same information is presented in both, but the text-file is easier to automatically parse or take a quick look at from the terminal:

```
$ less wave_c_4p*.txt
Executable:     wave_c
Resources:      4 processes, 1 node
```

```
Machine:        node208
Started on:     Thu Nov 14 03:27:55 2013
Total time:     5 seconds (0 minutes)
Full path:      /global/users/mark/allinea/tools/examples
Notes:

Summary: wave_c is CPU-bound in this configuration
CPU:      94.3% |========|
MPI:       5.7% ||
I/O:       0.0% |
This application run was CPU-bound. A breakdown of this time and advice
for investigating further is found in the CPU section below.
As very little time is spent in MPI calls, this code may also benefit
from running at larger scales.
…
```

You will see something similar to this on your first run – at 4 processes the example program is massively CPU bound. There's a lot more in the CPU breakdown section, too, but before we dig into it let's put this in a browser and enjoy the high-definition HTML version.

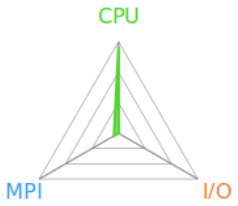
Viewing HTML files is best done on your local machine first. Many larger sites have places you can put HTML files to be viewed from the intranet - these directories are a good place to automatically send your Performance Reports to. Alternatively, you can use scp or even the excellent "sshfs" to make the reports available to your laptop / desktop:

```
$ scp login1:allinea/tools/examples/wave_c_4p*.html .
$ firefox wave_c_4p*.html
```

A sample report is shown on the next page, followed by a detailed description of each section.

allinea Performance Reports

Executable:	wave_c
Resources:	4 processes, 1 node
Machine:	node208
Start time:	Thu Nov 14 03:27:55 2013
Total time:	5 seconds (0 minutes)
Full path:	/global/users/mark/allinea/tools/examples
Notes:	



Summary: wave_c is CPU-bound in this configuration

The total wallclock time was spent as follows:

CPU	94.3%	Time spent running application code. High values are usually good. This is **very high**; check the CPU performance section for optimization advice.
MPI	5.7%	Time spent in MPI calls. High values are usually bad. This is **very low**; this code may benefit from increasing the 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 CPU-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 how the 94.3% total CPU time was spent:

Scalar numeric ops	21.4%
Vector numeric ops	0.0%
Memory accesses	67.6%
Other	10.9%

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

Of the 5.7% total time spent in MPI calls:

Time in collective calls	7.1%
Time in point-to-point calls	92.9%
Estimated collective rate	82.5 bytes/s
Estimated point-to-point rate	93.5 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 synchonization overhead; use an MPI profiler to investigate further.

I/O

A breakdown of how the 0.0% total I/O time was spent:

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

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

Memory

Per-process memory usage may also affect scaling:

Mean process memory usage	133 Mb
Peak process memory usage	133 Mb
Peak node memory usage	13.8%

The peak node memory usage is very low. You may be able to reduce the total number of CPU hours used by running with fewer MPI processes and more data on each process.

Summary Section

This shows how the application's wallclock time was spent, broken down into three areas:

CPU 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.

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. This does not include time spent in MPI-IO calls.

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. This includes all time spent in MPI-IO calls.

In this example file we see that Performance Reports has identified the program as being CPU-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 – CPU, MPI and I/O. We've found it helpful to recognize and compare these triangular shapes when flicking between multiple reports.

CPU Section

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 CPU time, not to the entire application run. Time spent in MPI and I/O calls is not represented inside this section.

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.

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.

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's resources. Often it is possible to reduce this figure by analyzing loops for poor cache performance and problematic memory access patterns, boosting performance significantly.
Other	A catch-all for all other housekeeping operations such as branches, jumps, comparisons and so on. This should be low for all scientific codes. A high value may be caused by busy-waiting, but values in the range of 10-20% are usually not of interest.

MPI Section

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.

Time in collective calls	The percentage of time spent in collective MPI operations such as MPI_Scatter, MPI_Reduce and MPI_Barrier.
Time in point-to-point calls	The percentage of time spent in point-to-point MPI operations such as MPI_Send and MPI_Recv.
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.
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.

I/O Section

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.

All MPI-IO calls and associated reads/writes are counted here and not in the MPI section, even though implementations may perform some MPI communication under the covers.

Time in reads The percentage of time spent on average in read operations from the application's perspective, not the filesystem's perspective.

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.

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.

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.

Memory Section

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.

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

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 that 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.
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.

Manual Linking for Cray Systems and Static Binaries

Performance Reports normally uses LD_PRELOAD to insert its instrumentation libraries into the application before running it. This is not currently supported in two cases:

1. Cray systems, using either dynamic or static linking
2. Statically-linked programs on any system

Extra steps are required to generate performance reports for these cases.

What is dynamic / static linking?

Executables are called *dynamically-linked* if they depend on external library files that are automatically loaded into memory by the system at run-time.

By contrast, a *statically-linked* executable contains all the libraries and code it requires to run within it.

On most systems you can see the dynamic libraries (if any) used by an application using the `ldd` command:

```
$ ldd examples/wave_c
        linux-vdso.so.1 =>  (0x00007fff9b5ff000)
        libhugetlbfs.so => /usr/lib64/libhugetlbfs.so
(0x00007fc67ba0d000)
...
        libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00007fc6743bd000)
```

For a statically-linked application, the following is printed:

```
$ ldd examples/wave_c
        not a dynamic executable
```

Performance Reports on Cray or Statically-linked applications

For both of these cases the LD_PRELOAD mechanism is not currently supported, so you must link the instrumentation libraries into your application directly:

1. Move into the working directory for your application and compile the instrumentation libraries there:

   ```
   $ cd ~/allinea/tools/examples
   $ ~/allinea/tools/bin/make-map-cray-libraries
   ```

   There are several variants of this command. The above is for dynamically-linked programs on Cray systems. If you have a statically-linked program on a Cray, use:

   ```
   $ ~/allinea/tools/bin/make-map-static-cray-libraries
   ```

   And for all non-Cray statically-linked programs:

   ```
   $ ~/allinea/tools/bin/make-map-static-libraries
   ```

   Each of these commands produces libmap-sampler.so and libmap-sampler-pmpi.so files in the current directory. These are the libraries that must be linked with your application. Relevant instructions are also printed with compile/link flags that you can copy and paste. For example, for a statically-linked program on a Cray system the output might be:

   ```
   Created the MAP libraries in /users/mark/allinea/tools/examples:

      libmap-sampler.a

      libmap-sampler-pmpi.a

   To instrument a program, add these compiler options:

      compilation : -g (or '-G2' for native Cray fortran) (and -O3 etc.)

          linking : -L/users/mark/allinea/tools/examples -lmap-sampler-
   pmpi -Wl,--undefined,allinea_init_sampler_now -lmap-sampler -lstdc++
   -lgcc_eh -Wl,--whole-archive -lpthread -Wl,--no-whole-archive -Wl,--eh-
   frame-hdr ... EXISTING_MPI_LIBRARIES

   If your link line specifies EXISTING_MPI_LIBRARIES e.g. -lopenmpi,
   these must appear *after* the MAP libraries in the link line. There's a
   comprehensive description of the link ordering requirements in section
   17.1.4 of /users/mark/allinea/tools/doc/userguide.pdf
   ```

2. Compile / link your program with the extra link flags shown above. You may need to edit your Makefile and look for a LFLAGS line. For our example code, we can compile and link it in one step on a Cray as follows:

   ```
   $ cc -O3 wave.c -o wave_c -L/users/mark/allinea/tools/examples -lmap-
   sampler-pmpi -Wl,--undefined,allinea_init_sampler_now -lmap-sampler
   -lstdc++ -lgcc_eh -Wl,--whole-archive -lpthread -Wl,--no-whole-archive
   -Wl,--eh-frame-hdr
   ```

The copy-pasted part is in italics. Note that I did not include the recommended compile flags (-g / -G2) – these are recommended for Allinea MAP but are not required for Allinea Performance Reports.

3. There is no step three – the program is now ready to run with or without Performance Reports! You can now continue from step 4 in the “Detailed Step-by-Step Overview” section, e.g.:

```
$ ~/allinea/tools/bin/perf-report aprun -n 16 ./wave_c
Wave solution running with 16 processes

0: points = 1000000, running for 30 seconds
points / second: 1326.3M (82.9M per process)
compute / communicate efficiency: 66% | 81% | 98%

Points for validation:
0:0.00  200000:0.95  400000:0.59  600000:-0.59  800000:-0.95
999999:0.00
wave finished
```

Troubleshooting

Some MPIs, most notably MVAPICH, are not yet supported by Allinea's Express Launch mode (in which you can just put “perf-report” in front of an existing mpirun/mpiexec line). These can still be measured using the Compatibility Launch mode.

Instead of this Express Launch command:

```
perf-report mpiexec <mpi args> <program> <program args>
```

Use this Compatibility Launch version:

```
perf-report -n <num procs> --mpiargs=”<mpi args>” <program> <program
args>
```

The `mpiexec` command is omitted, the number of processes to launch is passed explicitly to `perf-report` and the MPI arguments are wrapped in quotes and passed with the `--mpiargs=` parameter.

If you write a script to wrap this for your users then be careful to ensure that quotes are correctly escaped inside the MPI arguments.

Further Reading

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:

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/

Deeper CPU metric analysis

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

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

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/