Introduction

This document will guide you through the process of using Scalasca on HPC Wales.

What is Scalasca?

Scalasca (SCalable performance Analysis of LArge SCale Applications) is a software tool that supports the performance optimisation of parallel programs by measuring and analysing their runtime behaviour. The analysis identifies potential performance bottlenecks – in particular those concerning communication and synchronisation – and offers guidance in exploring their causes.

Further details on accessing HPC Wales systems can be found in the User Guide and help can be obtained through the Support Desk:

- E-mail support@hpcwales.co.uk
- Telephone 08452 572 207
- Website https://hpcwprod.service-now.com/

Table of Contents

Ewrop & Chymru: Buddsoddi yn eich dyfodol
Cronfa Datblygu Rhanbarthol Ewrop
Europe & Wales: Investing in your future
European Regional Development Fund
ERDF
Llywodraeth Cymru
Welsh Government

Profiling and Scalasca

Profiling is a vital step in developing efficient, scalable applications. The key parts to profiling any application are looking at the time spent executing code, analysing the time spent waiting for data to be delivered from the memory hierarchy to the CPU, and quantifying the time spent in parallel communication calls. By understanding the performance of the application it enables you to target any optimisation work at the right areas to improve both serial execution and scaling.

Do remember that a very inefficient serial algorithm will probably give better parallel scaling because the parallel overheads are a much smaller proportion of the time. However, when doing optimisation always concentrate on improving the actual execution of your code as this gives you the answers in a shorter amount of computational time overall.

Profiling works by analysing a particular run of your code and either (1) recording where it is every, say, microsecond, or (2) recording the time at which each instruction was executed. The former is known as a *sampling experiment*, the latter as a *tracing experiment*.

Serial execution

Basic profiling of any code will enable the user to discover which routines have the highest overhead in terms of work. If your code spends 90% of its execution time in one routine and the other 10% split equally between five others, there is little to be gained (at this stage) from looking at reducing the cost of anything but the largest.

Profiling can give more fine-grained information that just which functions are taking a long time. By using line-level profiling data it is possible to see which lines are the most computationally demanding. These are typically loops and sometimes it is possible to rewrite these to make significant performance increases.

To go beyond saying an individual line is taking a long time, it is necessary to actually look at what is going on inside the CPU. Various *hardware counters* are available to allow measurements of how much time is being spent waiting for memory to be delivered from the various levels of cache, for example. This is discussed further in relation to *PAPI*.

Parallel execution

When running in parallel, typically using MPI or OpenMP, additional overheads are necessarily incurred in the code. Every message that is sent needs to be received, and hence may need waiting for if computation and communication have not been overlapped sufficiently. Every collective operation will be waiting for either the message to be sent or received, or in the worst case of an MPI_Alltoall, getting a message from every other processor. Barriers are always waiting points where increasing numbers of processes are idle until the last one arrives at the synchronisation point.

Avoiding such problems is often easier said than done. By profiling, though, it is possible to diagnose the biggest offenders in your code, and the barrier to scaling that they present. On top of using a profiler on a single multi-process run, it is important to consider how the scaling has changed by increasing the processor count. For instance, consider a routine-based profiling of a code with two major functions:

Routine	16 processors	64 processors
`bigger`	200s	50s
`smaller`	50s	45s

In this case we can see that the function `bigger` may be costing more time than `smaller`,

Ewrop & Chymru: Buddsoddi yn eich dyfodol
Cronfa Datblygu Rhanbarthol Ewrop
Europe & Wales: Investing in your future
European Regional Development Fund
ERDF
Llywodraeth Cymru
Welsh Government

but it is the latter that does not scale at all well whereas `bigger` scales perfectly. Optimisation work would be focussed differently on the two routines: serial optimisations may improve `bigger`, but analysing the cause of the lack of scaling in `smaller` is vital for enabling parallel scalability of the code.

Using a profiler: 1. Instrumentation

Typically, profilers work by *instrumentation*. This is where timing or logging information gets placed into objects at compile time in order to record events. Libraries such as MPI are often provided in a pre-instrumented manner. Your own code will not be.

This means that to profile more than just the MPI library calls, you need to recompile your software with the appropriate timing information included. Each profiler provides tools to do this instrumentation automatically. In addition, they may also provide an API to allow developers to include their own instrumentation.

For Scalasca, this step is explained below in the section '*Instrumenting with Scalasca*'.

Using a profiler: 2. Running

Once you have an instrumented executable it is time to run it. For Scalasca the normal run command is modified slightly, as explained in the section '*Gathering Performance Data with Scalasca*'. The different ways of executing sampling and tracing experiments are explained here.

When running a parallel profiling experiment, adding all the instrumentation into the code may add a significant overhead to execution time. As such there are ways to limit the parts of the code from where data may be gathered. This is typically known as a *filter* as it selects which parts of the code may be recorded. Filter files for Scalasca are explained in the section '*Filter Files for Scalasca*'.

The use of hardware counters, also known as *metrics*, is independent of the compilation of the code, and so the recording of them may be enabled at run time. They are hardware dependent and use particular registers to record these events. As such, you are limited to how many counters you may record on a single run. Some metrics are derived, meaning that the counter is worked out from other metrics, and some combinations of counters may be incompatible. The library *PAPI* is used to interface with these counters. More information on using these counters with Scalasca is given in the section '*PAPI Hardware Counters in Scalasca*'.

Using a profiler: 3. Analysing the data

Once the instrumented application has been run, data will have been written to disk describing the execution profile. Now it is possible to start to examine this detail to understand where the inefficiencies in the code are.

Most profiling tools, including Scalasca, offer the ability to have a textual summary printed out of routine level costs. The power of the profiler comes through the GUI interface and any suggestions it may be making about inefficiencies it thinks it can identify. '*Analysing Profiles in Scalasca*' explains the Scalasca GUI and how it may be used to diagnose problems effectively.

When a tracing experiment has been conducted, it is possible to compare the timelines of execution across processes, enabling a visual representation of the messages being sent and periods waiting for synchronisation. Unfortunately Scalasca does not have its own trace analyser, and so on HPC Wales it is recommended to run a trace using the TAU

instrumentation and visualise using Jumpshot. This is explained in the guide to TAU.

Over the rest of this document the steps to profiling a parallel application using Scalasca are explained. This is not a straightforward process to explain, so this document is best read while actually running a profiling case on your own code at the same time.

When doing a real profiling experiment, the sections on 'Filtering' and 'Hardware counters' would usually follow doing initial visualisations using the GUI in the 'Analysing' section. Similarly, tracing experiments should only be performed after understanding the results of the sampling experiments.

Please note this guide is intended to be only a quick-start guide for using Scalasca on the HPC Wales systems. For full instructions on using Scalasca and the other software it includes, please see the documentation linked to at the end.

Preliminaries

On the HPC Wales systems both versions 1.4 and 2.0 of Scalasca have been installed. The latter is the latest release and features some major changes from the former. If you have not used Scalasca before it is recommended to use the 2.0 release. For existing v1.4 users, there may be some differences, most notably for in-source instrumentation as EPIK is no longer supported, having been replaced by Score-P which provides similar functionality. The rest of this documentation will focus on version 2.0.

Before compiling or running a job to be profiled using Scalasca, you must ensure that you have loaded the necessary modules. These include not only the modules for Scalasca, but also the appropriate compiler and MPI modules. This is because Scalasca requires that any executable to be analysed must be generated using the same compiler and MPI combination that was used to build Scalasca itself. On HPC Wales systems, Scalasca was installed using the default compilation environment at the time. Thus to use Scalasca on the Westmere nodes (i.e. machines with a hostname of the form ??-log-00?) you must load the following modules:

```
[user.name@cf-log-001 ~]$ module load compiler/intel-11.1
[user.name@cf-log-001 ~]$ module load mpi/intel-4.0
```

Similarly, on the Sandy Bridge nodes (i.e. ??-sb-log-00?):

```
[user.name@cf-sb-log-001 ~]$ module load compiler/intel/13.0
[user.name@cf-sb-log-001 ~]$ module load mpi/intel/4.1
```

In order to make all the Scalasca tools available, its module is loaded using:

```
[user.name@cf-log-001 ~]$ module load scalasca
```

Instrumenting with Scalasca

In order to use Scalasca, it is necessary to recompile your code to allow Scalasca to instrument the executables or libraries that are created. Note that this means any external libraries you use will not be analysed by default.

Scalasca instrumentation of every object is done through prefixing the compiler command with "`scalasca -instrument`". For example, to compile a C program use:

Instrumenting with Scalasca

```
[user.name@cf-log-001 ~]$ scalasca -instrument icc sampleprog.c
```

Similarly, a Fortran program could compiled using

```
[user.name@cf-log-001 ~]$ scalasca -instrument ifort sampleprog.f90
```

Therefore, `Makefiles` should be modified appropriately. Full details of the instrumentation options are given by "`scalasca -instrument --help`".

Gathering Performance Data with Scalasca

Once an instrumented executable has been obtained, it needs to be run, again using Scalasca. The command for doing this is "`scalasca -analyze`". In a job script the `mpirun` line thus becomes:

scalasca -analyze mpirun -np 4 ./exec

As an example, a code running on four processes each with two OpenMP threads would give output that would look something like:

```
S=C=A=N: Scalasca 2.0 runtime summarization
S=C=A=N: ./scorep_exec_4x2_sum experiment archive
S=C=A=N: Thu Sep 13 18:05:17 2012: Collect start
mpirun -np 4 ./exec

 [... application output ...]
S=C=A=N: Thu Sep 13 18:05:39 2012: Collect done (status=0) 22s
S=C=A=N: ./scorep_exec_4x2_sum complete.
```

Note that execution with instrumentation will take longer than the unmodified optimised version.

Upon completion, a directory containing the profiling data will have been created in the directory from which the executable was launched. In the above case this directory would be called `scorep_exec_4x2_sum` which would typically contain files `*.cubex`, `scorep.*` and `scout.*`. Note that by default the experiment directory is named according to the name of the executable, the number of MPI processes and the number of OpenMP threads. This means that re-running an experiment will fail if the previous directory still exists, so either move or rename it.

As before, the command-line options can be examined using "`scalasca -analyze --help`". Most notable amongst these is the "`-t`" flag which turns on tracing experiments. This means that rather than sampling every, say, microsecond, and producing composite data at the end of an execution, the run records every instruction in a time-stamped format which can then be examined afterwards. Tracing experiments produce much larger output files, in a directory called, say, `scorep_exec_4x2_trace`, and hence may be best done either on short executions or with the use of a filtering file (see the next section). Either way it is necessary to perform a preparatory step before running a trace.

A tracing experiment needs a suitably large buffer size to be pre-specified. An estimate of this size is therefore needed. Once a sampling experiment has been performed the `cube_score` program can be used to provide this estimate.

Ewrop & Chymru: Buddsoddi yn eich dyfodol
Cronfa Datblygu Rhanbarthol Ewrop
Europe & Wales: Investing in your future
European Regional Development Fund
ERDF
Llywodraeth Cymru
Welsh Government

Gathering Performance Data with Scalasca

```
[user.name@cf-log-001 ~]$ cube_score -r scorep_exec_4x2_sum/profile.cubex
Reading scorep_exec_4x2_sum/summary.cubex... done.
Estimated aggregate size of event trace (total_tbc): 1112694576 bytes
Estimated size of largest process trace (max_tbc):   428006758 bytes
(Hint: When tracing set ELG_BUFFER_SIZE > max_tbc to avoid intermediate
flushes or reduce requirements using file listing names of USR regions to
be filtered.)

flt  type       max_tbc         time       % region
      ANY     428006758       107.77  100.00 (summary) ALL
      USR     428006758       107.77  100.00 (summary) USR

      USR     137954976         3.68    3.42 findtnconn
      USR     137265840         7.10    6.58 FindConn
      USR      12349368         0.64    0.59 MPI_Test
      USR       9801984         0.33    0.31 findlegid
      USR       5716248         0.46    0.42 tensor_delete
      USR       5656704         0.63    0.58 tnode_free
      USR       5654592         0.58    0.54 tensor_fuse
     ..........
```

The output `max_tbc` (output line 3) is the estimate of the size of the trace. Therefore, in your job script set the environment variable `SCOREP_TOTAL_MEMORY` (rather than the v1.4 variable `ELG_BUFFER_SIZE`) to be this reported figure (or a bit bigger for safety). For example, in this case the job script will include:

```
export SCOREP_TOTAL_MEMORY=450000000
scalasca -analyze -t mpirun -np 4 ./exec
```

which, when run, will produce additional `trace.*` files in the output directory.

Filter Files for Scalasca

Performing profiling, especially tracing experiments, on whole applications may be very inefficient in terms of overall usefulness. This is because adding the instrumentation to, for example, commonly encountered but lightweight functions may give a big overhead in terms of execution, without helping to profile the code. As was explained in the introduction, it is necessary to focus any optimisation work on the parts with significant scope for improvement, only looking at the most costly functions.

By performing an initial sampling experiment, the highest cost functions are identified. The program cube_score lists these in decreasing order of costs, as shown in the section above. A filter file will typically be in order to tell the profiler which routines it should be profiling and which not.

For Scalasca there are currently two different formats of filter file that need to be constructed. First, cube_score needs a list that details the functions to be excluded, i.e. those you are ***not*** interested in profiling. For a large application this may be extensive, and so can be generated automatically from running it on a previously performed sampling experiment:

Filter Files for Scalasca

[user.name@cf-log-001 ~]$ cube_score -r scorep_exec_4x2_sum/profile.cubex
| awk -F" " '{print $5 " " $6 " " $7 " " $8}' > filter.file

then editing the filter.file to remove the first few lines and those functions ***you wish to keep***.

In order to update the trace estimated cost to reflect the filtering requested, the `cube_score` command is changed to:

```
[user.name@cf-log-001 ~]$ cube_score -f filter.file -r
                                  scorep_exec_4x2_sum/summary.cubex
```

Unfortunately Scalasca itself no longer uses this simple format. The Score-P format is used instead, which does give much greater flexibility, and their documentation should be consulted for achieving fine-grained filtering. A simple example, though, can be made by adding a couple of lines to the top and bottom of the filter file made for `cube_score`:

```
SCOREP_REGION_NAMES_BEGIN
  INCLUDE

[…. Function list here ….]

SCOREP_REGION_NAMES_END
```

This filtering file can then be passed to Scalasca for either sampling or tracing experiments, by using a "`-f <file>`" flag.

```
scalasca -analyze -t -f filter.file mpirun -np 4 ./exec
```

PAPI Hardware Counters in Scalasca

By default, Scalasca counts MPI and OpenMP calls and execution time in each function. Additional metrics are available through the use of hardware counters. These enable the user to discover exactly how many times the CPU is having to perform which type of operations, access which part of the memory hierarchy of the machine, or be stalled waiting for something to occur.

These are accessed through the PAPI library. Note that PAPI is not included on the Westmere systems due to it being incompatible with the kernel version that is installed on those machines, meaning hardware counters are not available.

When using PAPI you can find a list of available counters by loading the module `papi` and doing

```
[user.name@cf-sb-log-001 ~]$ module load papi
[user.name@cf-sb-log-001 ~]$ papi_avail
```

This gives a list of, currently, 108 different events PAPI knows about of which 50 are available on the HPC Wales machine. From this list one can select compatible ones up to the maximum allowed on the machine. It appears that 11 may be recorded simultaneously on this system. The chosen events are then selected by setting the environment variable `SCOREP_METRIC_PAPI`, e.g.

PAPI Hardware Counters in Scalasca

```
[user.name@cf-sb-log-001 ~]$ export SCOREP_METRIC_PAPI="PAPI_FP_OPS,PAPI_L1_DCM"
```

When the next tracing experiment occurs there will be extra output files listing these results.

Analysing Profiles in Scalasca

After your code has run, profiling data will have been generated (by default) in the directory where your `mpirun` command was executed.

As seen above, summary text output of the profiling experiment is given by running `cube_score`. For parallel runs this text summary does not give much detail. Instead Scalasca provides a GUI interface to the profiling data. Having an X11-enabled shell on the HPC Wales login node ("`ssh -Y …`" from your Linux or Mac machine; using XMing or MobaXTerm from a Windows machine) is necessary. Then simply run "`scalasca -examine`" on one of the `.cubex` files in your experiment directory:

```
[user.name@cf-log-001 ~]$ scalasca -examine scorep_exec_4x4_sum/profile.cubex
```

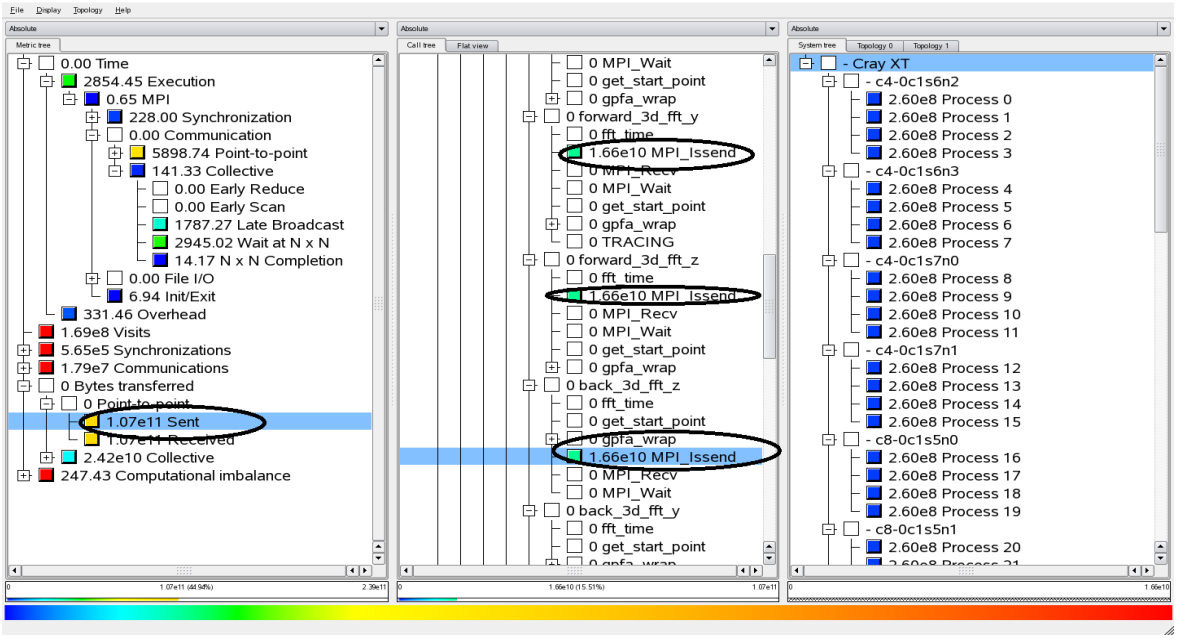
It is important to understand that there are differences between the `.cubex` files that can be loaded. Generally, for sampling experiments use `profile.cubex` and for tracing experiments use `summary.cubex`. The latter includes the information from the other `.cubex` files that have been recorded for each of the chosen metrics.

The Scalasca GUI starts with the interface having a default view showing three panels, each containing tree views of the profiling of the application. The left hand panel shows the *metrics* recorded. The second panel is the functions visited, grouped by *backtrace*, i.e. the ones at the top were the functions encountered first. The third window is the partitioning of the application over the computational resources used.

Selecting a node in the left hand pane sets the colour of the boxes in the second and third panes. For example, if 'Time' is the selected metric then execution time in each function is coloured blue to red for increasing amounts. Selecting a function in the middle pane then colours the individual resources in the right hand pane by the split of work between the processes.

Ewrop & Chymru:
Buddsoddi yn eich dyfodol
Cronfa Datblygu Rhanbarthol Ewrop
Europe & Wales:
Investing in your future
European Regional Development Fund
ERDF
Llywodraeth Cymru
Welsh Government

Analysing Profiles in Scalasca



By examining the time spent in the MPI functions, in the *Time* metric, and looking at information in the *Communications* and *Computational Imbalance* metrics, it can help identify bottlenecks. More information is provided when tracing is enabled, including identifying messages sent late and collective operations taking too much time.

In the picture above the counts of sent messages are highlighted, and towards the top the 'Time' metric tree the cost of these MPI operations is detailed. For this case it can be seen that there is 2854.45 s actual computation taking place but almost 6000 s spent doing point-to-point communication and over 4000 s spent in collective operations, mainly waiting for other processors to send data.

When you have performed hardware counter metric profiling using PAPI (see above), if these seem not to be available in the list of metrics, make sure you have loaded the 'profile.cubex' or the 'summary.cubex' file.

Online Documentation

The full Scalasca 2.0 user guide is still being developed at this time. This will be available from

http://www.scalasca.org/software/scalasca-2.x/documentation.html

which already has links to their current resources. As a starting point the User guide v1.4 is far more comprehensive and many things have not changed radically.

Ewrop & Chymru:
Buddsoddi yn eich dyfodol
Cronfa Datblygu Rhanbarthol Ewrop
Europe & Wales:
Investing in your future
European Regional Development Fund
ERDF
Llywodraeth Cymru
Welsh Government