

Profiling Firedrake for the APinTA-PDEs project

David Acreman

August 25, 2021

Contents

1	Introduction	2
2	Target platforms	2
3	Test cases	3
4	Profiling tools	5
4.1	Intel Parallel Studio	5
4.2	ARM Forge	12
4.2.1	Performance Reports	12
4.2.2	MAP	17
4.3	Cray PAT	21
5	Builds	22
6	Profiling Firedrake	23
6.1	Intel Trace Analyzer and Collector	23
6.2	ARM Performance Reports and MAP	24
6.3	Cray PAT	24
7	Conclusions	24

1 Introduction

In order to understand the performance characteristics of numerical schemes and the impact of potential optimisations it is necessary to gather information beyond simple measures of elapsed run times. A number of profiling tools exist which are capable of gathering performance information from applications parallelised using MPI, for example the time spent in MPI calls and the identification of performance problems such as load imbalance. This document gives an overview of the profiling tools available to the APinTA-PDEs project with a focus on tools which can provide information about MPI communication and related performance issues.

The Firedrake will be used extensively for development and rapid prototyping work in the course of the project. Firedrake is python-based domain specific language which generates code at run time. These characteristics make integrating Firedrake with profilers more complex than for traditional HPC applications written using C/C++/Fortran which are compiled then run. Although this report primarily focusses on Firedrake the profiling tools evaluated could be used for profiling other codes used by the project even if those tools are not currently suitable for use with Firedrake.

Section 2 presents the target platforms being considered in this report and reviews the hardware and software available on these systems. Section 3 describes the test cases which will be used to demonstrate the profilers. The profiling tools available to the project are discussed in section 4 and their capabilities demonstrated with the test cases from section 3. Section 5 discusses how to build Firedrake on the target platforms and which profilers the successful builds can be used with. Proof-of-concept results from these profilers are shown in section 6. Finally section 7 presents conclusions and recommendations.

2 Target platforms

An overview of the hardware in the target platforms for this project is shown in Table 1. The

System	Processor	Cores/node	Memory/node	Interconnect
Archer-2	AMD x86_64	128	256 GB DRAM	HPE Cray Slingshot
Isambard XCI	Thunder X2 ARM64	64	256 GB DRAM	Cray Aries
Isca	Intel x86_64	16/20	128 GB DRAM	Mellanox Infiniband
Server	AMD x86_64	128	256 GB DRAM	None

Table 1: Target platform hardware specifications.

target HPC platforms are Archer-2 (national tier-1), Isambard (national tier-2) and Isca (local tier-3). Isambard comprises multiple systems and for this project we will focus on the the XCI system which is an established production system based on the ARM64 processor architecture. The project has purchased a local server which is designed to be similar to an Archer compute node, although it will run Ubuntu which will make it easier to build Firedrake. The server can provide a reference installation as it will use an unmodified Firedrake installation, rather than the more customised installations on the HPC platforms which will use non-standard software components, for example MPI libraries and maths libraries.

An overview of the software stacks on the target platforms is shown in Table 2. Each HPC platform has the GNU compilers, compilers from the processor vendor (AMD, Intel, or ARM) and Cray systems also have the Cray compiler. The GNU compilers are the one compiler family which is available on all the target platforms.

On the Cray systems compilation is handled by wrapper scripts from the Cray Programming Environment. The wrapper script can run different compilers depending on which programming environment module is loaded at compile time (e.g. if `PrgEnv-cray` is loaded then `cc` calls the Cray C compiler and if `PrgEnv-gnu` is loaded `cc` calls the GNU C compiler). The MPI library and maths library (Cray libsci) are then linked by the wrapper script. On Cray systems the standard profiling tool is Cray Performance Analysis Tools (PAT) with the profiling tools from the ARM

System	Compilers	MPI libraries	Maths lib.	Profilers
Archer-2	GNU, Cray, AOCC	Cray MPICH2	Cray libsci	Cray
Isambard XCI	GNU, Cray, ARM	Cray MPICH2	Cray libsci	Cray, ARM
Isca (GCC-foss)	GNU	OpenMPI	OpenBLAS	None
Isca (Intel)	Intel	Intel	Intel MKL	Intel
Server	GNU	MPICH	OpenBLAS	None

Table 2: Software stacks on target platforms. AOCC is the AMD Optimizing Compiler Collection. Intel MPI and MVAPICH are MPICH derivatives which support an Infiniband interconnect. The Cray profiler is Cray Performance Analysis Tools (PAT), the ARM profilers are part of the ARM Forge tool suite and the Intel profilers are part of the Intel Parallel Studio suite.

Forge suite¹ also available on Isambard.

The software environment on Isca is managed using Easybuild² which has the concept of a toolchain. There are two toolchains on Isca: the GCC-foss toolchain and the Intel toolchain. Each toolchain has a different MPI library and maths library. Intel Parallel Studio has an MPI profiling tool called Intel Trace Analyzer and Collector (ITAC) which works with Intel MPI.

The server does not currently have any profiling software. However the profiling tools from Intel's Parallel Studio product are now available for free as part of the HPC toolkit as part of the newer OneAPI product³ and these could be installed on the server. There is also the option of obtaining an evaluation licence for ARM Forge to test these tools on the server. A server licence for the ARM Forge product costs \$USD 425 for a one year academic licence which allows profiling with up to 64 MPI processes⁴.

3 Test cases

The profilers will be tested using three different versions of a Mandelbrot set calculation. The different versions parallelise the calculation in different ways, have different performance characteristics and display different performance issues:

- Version 1: divides the computational domain into equal sized chunks along the real axis. Dividing the workload in this manner causes a load imbalance as some sub-domains will complete significantly more iterations than others (see figure 1)
- Version 2: similar to version 1 but interleaves the iterations on the real axis to improve the load balance. Both version 2 and version 1 pack a 2D array into a 1D buffer which is communicated using an `MPI_Reduce`. Even when the program is run on a single compute node (where MPI communication is quick) there will be an overhead from packing and unpacking the buffer.
- Version 3: implements a manager-worker pattern for distributing the iterations to ensure good load balancing. This version also replaces the `MPI_Reduce` with point-to-point communication between the manager process and the worker processes. The manager process does not perform any calculations which adversely affects performance with small numbers of MPI processes. However this version avoids calling `MPI_Reduce` and scales better to larger numbers of MPI processes than the other versions .

The following tests will be used to demonstrate the profiling tools:

- Test 1: version 1, no I/O, 4 MPI processes
- Test 2: version 1, with I/O, 4 MPI processes

¹ARM Forge was previously known as Allinea Forge

²<https://docs.easybuild.io/en/latest/>

³<https://software.intel.com/content/www/us/en/develop/tools/oneapi/all-toolkits.html>

⁴<https://store.developer.arm.com/store/high-performance-computing-hpc-tools/arm-forge>

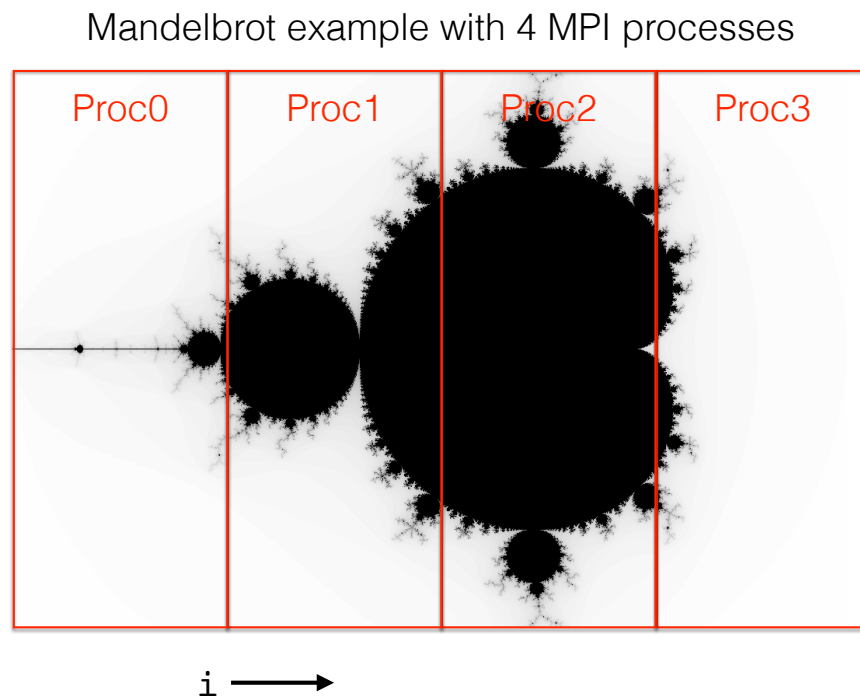


Figure 1: Domain decomposition in version 1 of the Mandelbrot program with four MPI processes. Darker colours show regions of the domain where more iterations are required illustrating the load-imbalance from this decomposition (e.g. process 2 will carry out many more iterations than process 0 or 3).

- Test 3: version 2, no I/O, 4 MPI processes
- Test 4: version 3, no I/O, 4 MPI processes
- Test 5: version 3, no I/O, 16 MPI processes

Comparing test 1 and test 2 shows the impact of I/O. The I/O comprises process 0 writing values to an ASCII file which adds a significant overhead which will show up in the profiling output. Comparing tests 1 and test 3 and shows the impact of load imbalance, as the only difference between these tests is the distribution of the workload. Tests 4 and 5 have been designed to illustrate a different communication pattern (manager-worker) and a different type of MPI call (point-to-point rather than collectives). The manager-worker pattern is relatively inefficient with 4 MPI processes (test 4) as one process is the manager which does not participate in the calculation but shows much better efficiency with 16 MPI processes (test 5).

4 Profiling tools

Some performance information can be obtained by using functionality already available in Firedrake. Performance information can be obtained from PETSc (by adding the `-log_view` flag when running Firedrake⁵) and there is also the ability to time sections of code by adding a PyOP2 timed stage⁶. These options will be available on all platforms when Firedrake is the target application but the rest of this report will look at how the external profilers already installed on the target platforms can be used to obtain extra information.

There are three different profilers installed on the target platforms with different platforms having different profiling tools available:

- Intel Parallel Studio: Isca only (requires Intel MPI)
- ARM Forge: Isambard only (currently only licensed on Isambard)
- Cray PAT: Isambard and Archer (Cray systems only)

Each of these products is a suite of tools with a range of capabilities. This report will focus on the tools appropriate for understanding the performance of parallel applications which use MPI.

4.1 Intel Parallel Studio

The MPI profiler in Intel Parallel Studio is called Intel Trace Analyzer and Collector (ITAC). Other profiling tools are available, which provide information on node-level performance for example, but the focus of this report is on MPI performance so ITAC is the tool under consideration.

In the following examples the Mandelbrot executable will be build with MPI tracing enabled as follows. Firstly load the modules for the Intel compilers and ITAC:

```
> module load intel/2017b
> module load itac/2017.1.024
```

Then compile the executable with `-trace` flag:

```
> mpicc -trace -o mandelbrot mandelbrot_mpi.c
```

(including the `-g` flag enables function level information to be obtained but it not required). The `itac` module needs to be loaded in the job script (in addition to the `intel` module) but no other modifications are required to the job script. When the job runs a number of additional files are generated, including a file with a `.stf` extension which is the trace file. The profiling output can be viewed opening the trace file with the Trace Analyzer graphical interface:

```
> traceanalyzer.bin mandelbrot_mpi.stf &
```

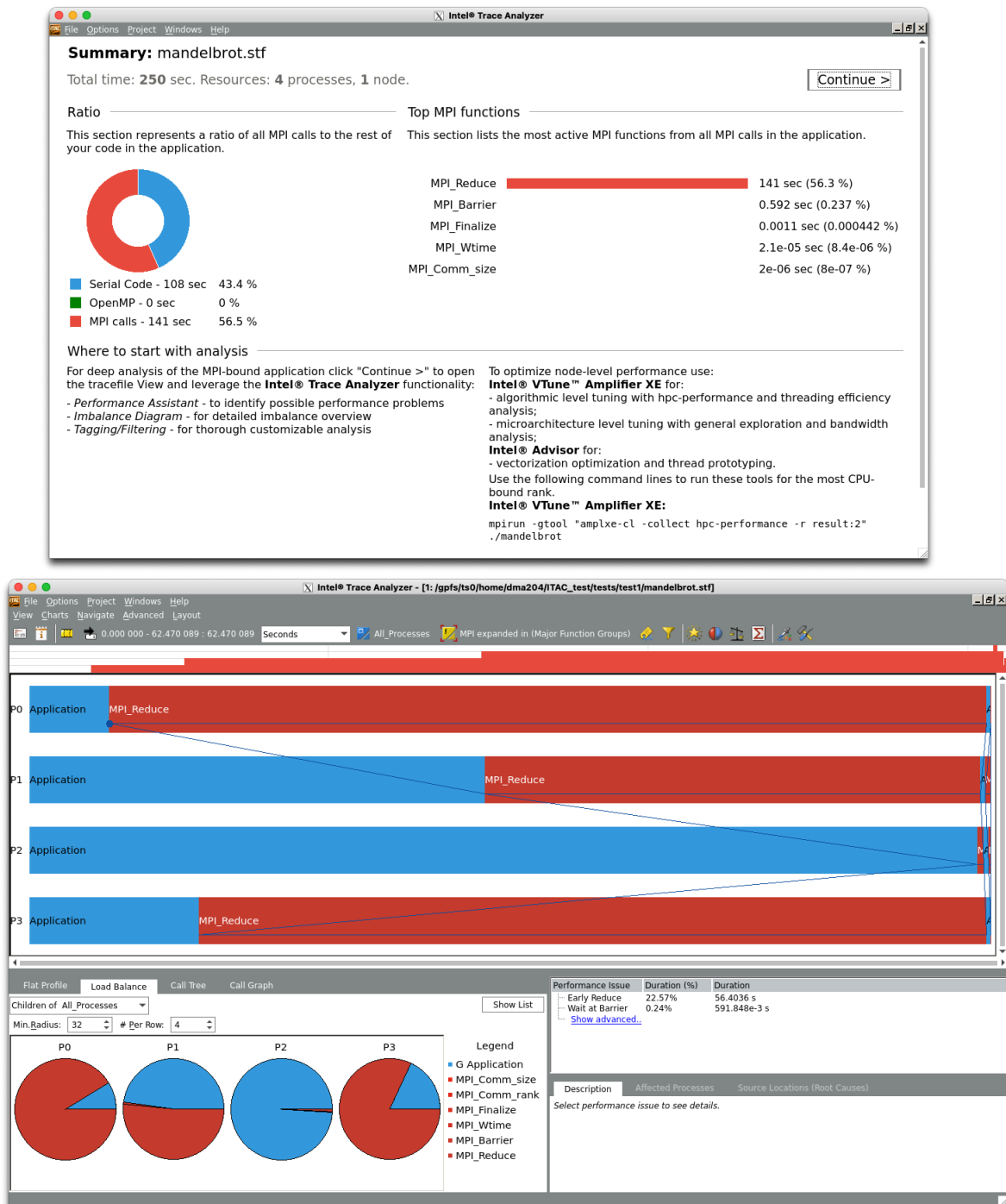


Figure 2: Intel Trace Analyzer plots for test 1. On the summary page (top) the load imbalance shows up as a large fraction of time spent in MPI.Reduce. The event timeline (bottom) shows that process 2 takes longer than the other processes to complete its iterations which results in the other processes waiting at the MPI.Reduce call.

Test 1: Figure 2 (top) shows the opening summary page for test 1 (Mandelbrot version 1, no I/O, 4 MPI processes). This test has a load imbalance which is seen as a large fraction (56%) of time spent in `MPI_Reduce`. The reduce itself is not time consuming (all communication takes place on the same compute node) but this is the blocking collective where the MPI processes synchronise. This can be seen in the lower panel of figure 2 which shows the event timeline. To view the event timeline as seen in this figure:

- Click the continue button on the summary page
- Select **Charts** → **Event Timeline** to open a new pane showing process activity over time.
- In the lower left pane select the **Load Balance** tab and click the **Show pies** button.
- Right click on a red MPI section in the timeline and choose **Ungroup MPI** from the drop down menu

The timeline shows that process 2 is taking much longer to complete its iterations than the other processes which has resulted in processes 0, 1 and 3 waiting at the `MPI_Reduce` call.

Test 2: this test is similar to test 1 but has I/O enabled. The summary page (see figure 3) shows a significantly longer runtime (450s vs. 250s) and significant time spent in `MPI_Barrier`. The I/O is carried by process 0 which causes all the other processes to wait at an `MPI_barrier` at the end of the program (processes are synchronised at a barrier before a final run time is reported by process 0). The effect of switching on I/O has been to introduce another load imbalance, this time with process 0 working and all other processes waiting. This can be seen clearly in the timeline where there are two distinct phases both of which have a load imbalance.

Test 3: this test uses version 2 of the program which has corrected the load imbalance and the profiling results are shown in figure 4. The run time is now significantly reduced (108 seconds vs. 250 seconds) and the time line shows the MPI processes reaching the `MPI_Reduce` together. At the end of the time line we can see some time spent carrying out the `MPI_Reduce` and some application time (blue) as the buffer is unpacked (if required the time line can be zoomed-in to show more detail in a selected region).

Test 4: this test uses version 3 of the Mandelbrot program which parallelises the workload using a manager-worker pattern to ensures good load balancing by distributing work at run-time. The summary page shows a large fraction of time spent in `vMPI_Recv` (24.8%) due to the manager process waiting to hand out work to worker processes. In this test there are only 4 MPI processes so this constitutes a large fraction of the total run time. The time line shows the worker processes (processes 1–3) completing together with negligible time spent in the final `MPI_Barrier`. The communication in this version of the program takes place with a large number of point-to-point operations instead of a single collective operation and the time line shows black lines connecting processes when they communicate. The lower right panel of the bottom figure is a message profile which shows which processes communicate using point-to-point operations (To view the message profile select: **Charts** → **message profile**). In the manager-worker pattern all the workers communicate only with the manger process so the communication is seen in the first row and first column of the message profile diagram.

Test 5: this test uses version 3 of the Mandelbrot program (manager-worker) as for the previous test but uses 16 MPI processes instead of 4 MPI processes. The time spent in `MPI_Reduce` is now much smaller (6.24%) consistent with slightly less than 1/16 of the run time (i.e. one manager process out of a total of 16 processes). The point-to-point communication between processes is seen to be most intense at the start of the calculation, when each unit of work is completed quickly, with less communication in the middle of the time line when each point takes longer to calculate.

⁵<https://petsc.org/release/documentation/manual/profiling/>

⁶<https://op2.github.io/PyOP2/profiling.html>

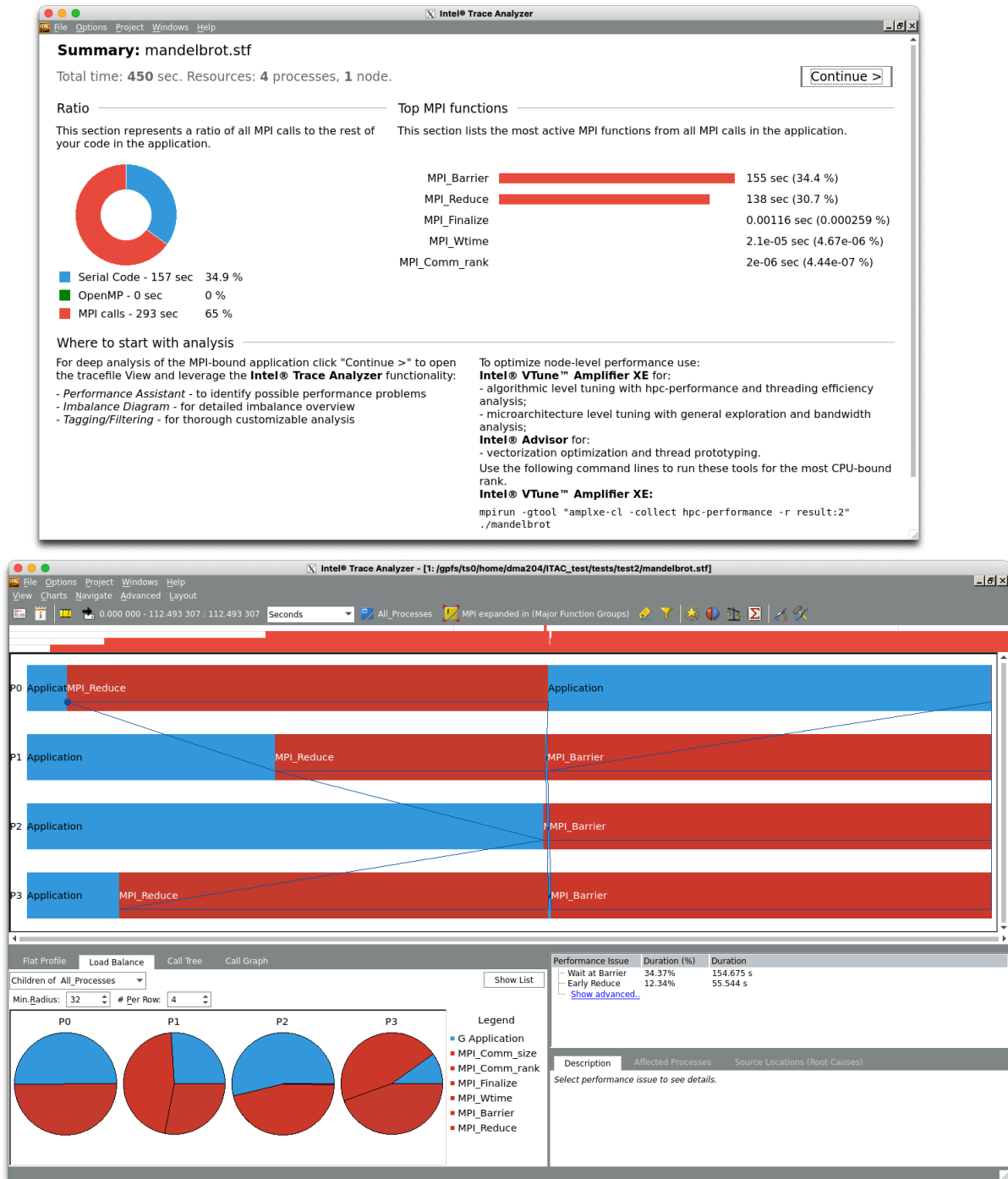


Figure 3: Intel Trace Analyzer plots for test 2 which is like test 1 but with I/O enabled. There are two distinct phases to the time line: firstly the calculation and secondly the I/O. Both phases have a load imbalance with the second phase showing processes waiting at a barrier as the rank zero process performs I/O.

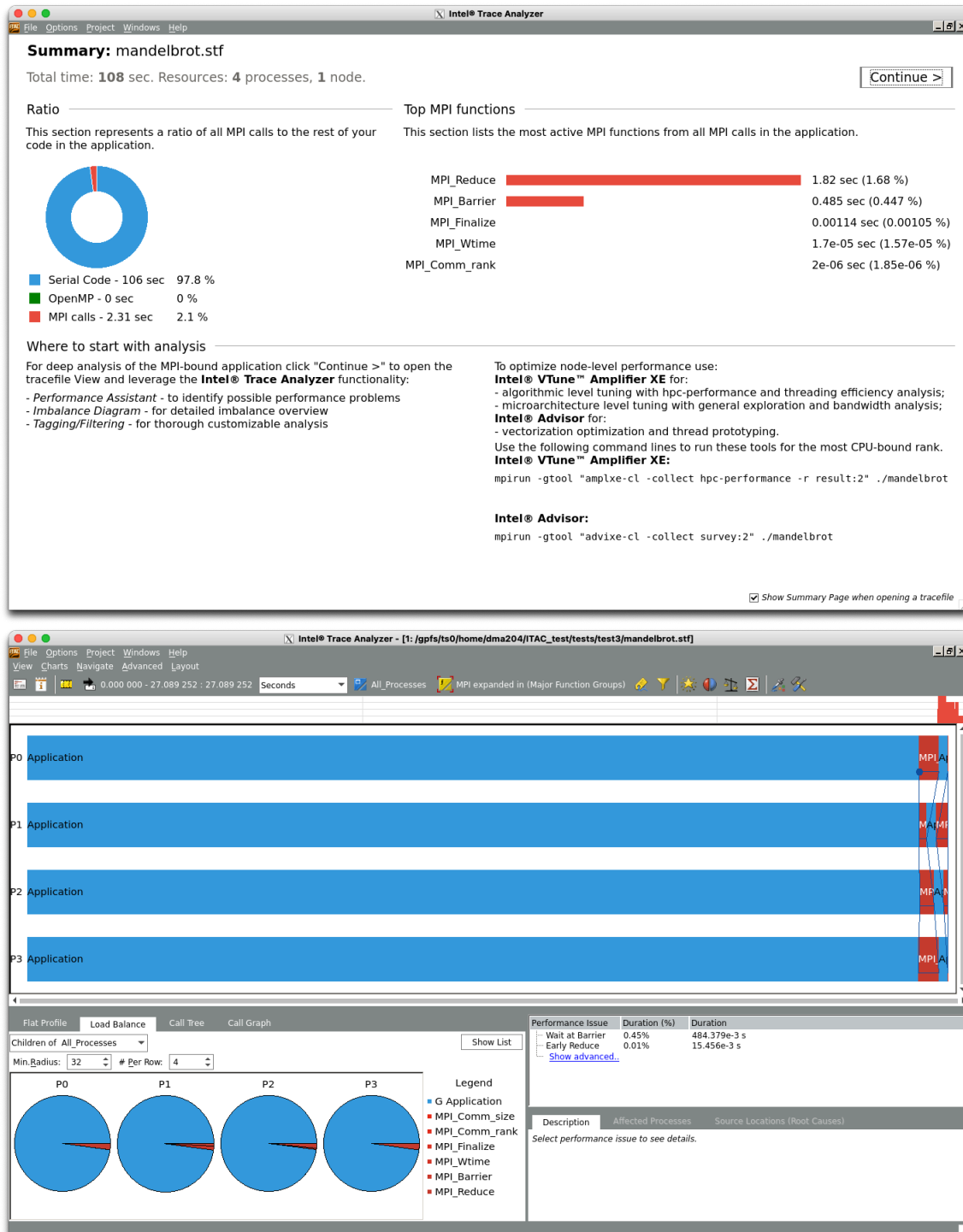


Figure 4: Intel Trace Analyzer plots for test 3. This test uses version 2 of the program which has corrected the load imbalance present in version 1 resulting in all MPI processes reaching the MPI_Reduce together.

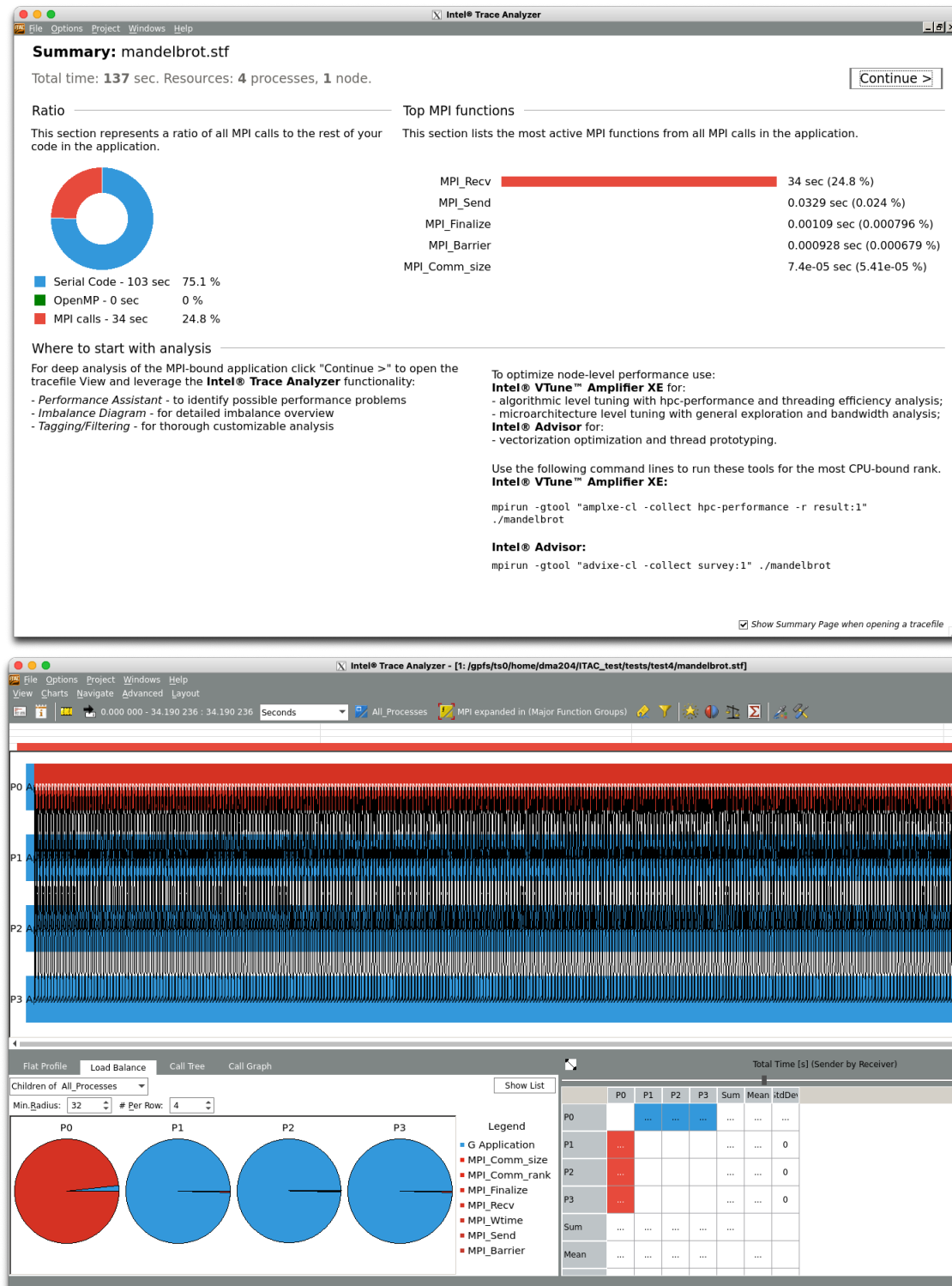


Figure 5: Intel Trace Analyzer plots for test 4. This test uses version 3 of the Mandelbrot program which parallelises the workload using a manager-worker pattern. In this example the communication takes place with a larger number of point-to-point operations instead of a single collective operation.

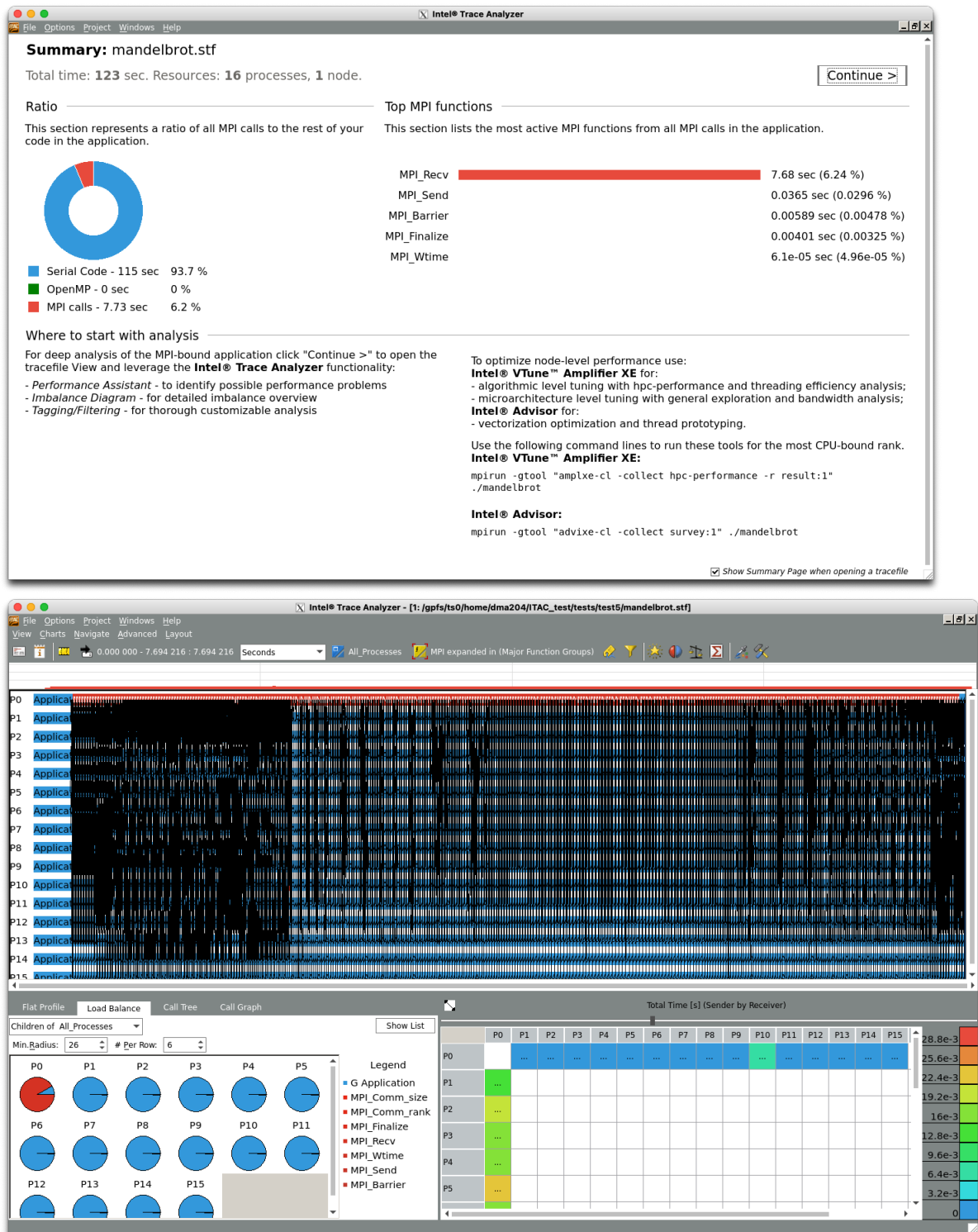


Figure 6: Intel Trace Analyzer plots for test 5. This test uses version 3 of the Mandelbrot program (manager-worker) as for the previous test but uses 16 MPI processes instead of 4 MPI processes.

4.2 ARM Forge

The ARM Forge suite of development tools is available on the Isambard system. ARM Forge includes two performance tools: “Performance Reports” and “MAP”. Performance Reports gives a one page summary of an application’s performance. MAP is a profiler for more in-depth analysis which can profile C++, C, Fortran and Python⁷.

4.2.1 Performance Reports

The documentation for Performance Reports says that on Cray systems dynamic linking or explicit linking with the profiling libraries is required⁸. However in the latest Cray compilers dynamic linking is the default so no changes are required to the build process. The following shows an example session and output from Performance Reports on the Isambard XCI system.

No changes are required at compile time so to compile an executable using the GCC compiler for example:

```
> module switch PrgEnv-cray PrgEnv-gnu
> cc -o mandelbrot mandelbrot_mpi.c
```

Two changes are required in the job script in order to activate performance reports. The first change is to load the `arm-forge` module:

```
module load tools/arm-forge
```

and the second change is to add the `perf-report` command before the `aprun` command e.g. replace

```
aprun -n ${nprocs} ./mandelbrot
```

with

```
perf-report aprun -n ${nprocs} ./mandelbrot
```

This method of launching a job under `perf-report` is referred to as Express Launch mode in the documentation. When the job runs two extra files are generated which contain the output from the performance report in text and HTML formats.

Test 1: the output from running a performance report on test 1 is shown in figure 7. Performance Reports correctly identifies that the application is MPI bound but without the time line it is not clear that load imbalance is responsible. A similar fraction of time is spent in MPI calls to that shown in the ITAC example (the corresponding ITAC output is shown in figure 2). Performance Reports also includes summaries of other statistics, for example CPU, I/O and memory metrics.

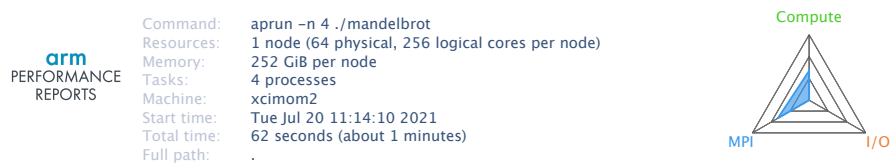
Test 2: results from this test are shown in figure 8. This test is like test 1 but with I/O enabled and time spent in I/O is now seen in the Performance Report. However the main performance issue caused by the I/O is actually an exacerbated load imbalance due to I/O being funnelled through the rank zero process causing all other processes to wait at an MPI barrier. Although slow I/O is the underlying cause the time spent in I/O is only a small fraction of the time breakdown and the main impact is additional time spend in MPI collectives.

Test 3: results from this test are shown in 9 The fraction of time spent in MPI is now low (around 2%) as the load imbalance has been corrected. The Performance Report now directs the reader to the CPU metrics section as the calculation is considered compute bound.

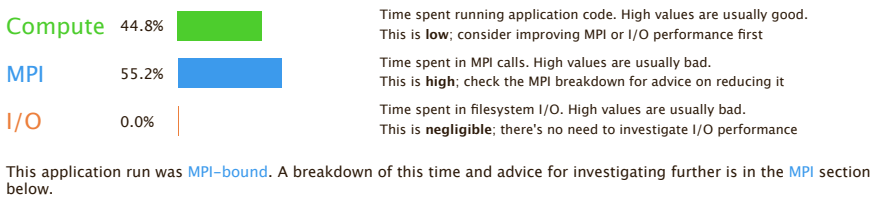
Test 4: Figure 10 shows the Performance Report from test 4 (manager-worker with 4 MPI processes). With this test only 3 out of 4 MPI processes carry out the calculation and the fourth process is the manager which does not compute any points. The MPI time of 25% is consistent with the manager process waiting for results from the worker processes. In the MPI section of

⁷<https://www.arm.com/products/development-tools/server-and-hpc/forgemap>

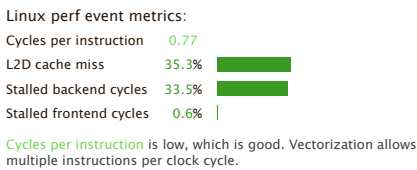
⁸<https://developer.arm.com/documentation/101136/2102/Performance-Reports/Get-started-with-Performance-Reports/Compile-on-Cray-X-series-systems>



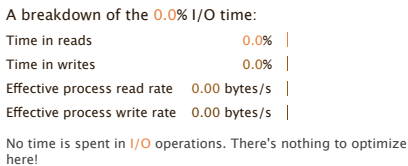
Summary: mandelbrot is **MPI-bound** in this configuration



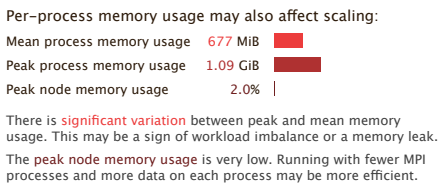
CPU Metrics



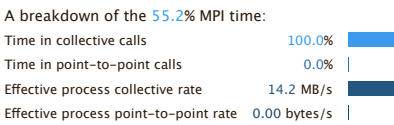
I/O



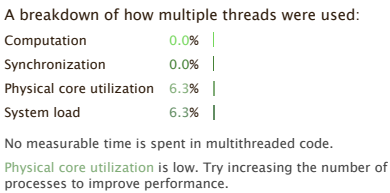
Memory



MPI



Threads



Energy

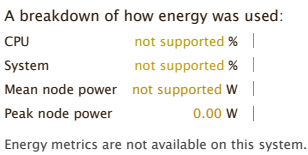
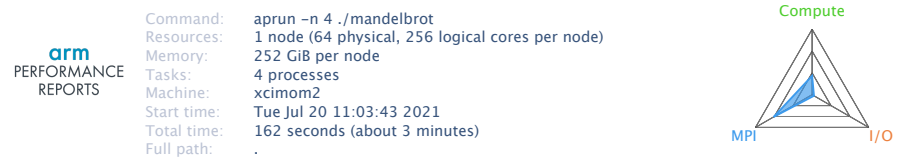
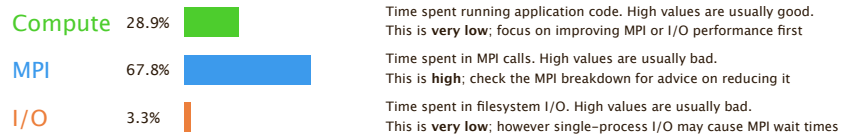


Figure 7: ARM Performance Report from test 1. The tool correctly identifies that there is excessive time spent in MPI calls but does not identify load imbalance as the performance issue.



Summary: mandelbrot is **MPI-bound** in this configuration



This application run was **MPI-bound**. A breakdown of this time and advice for investigating further is in the **MPI** section below.

CPU Metrics

Linux perf event metrics:

Cycles per instruction	1.39	<div></div>
L2D cache miss	27.3%	<div></div>
Stalled backend cycles	36.0%	<div></div>
Stalled frontend cycles	4.1%	<div></div>

Cycles per instruction is moderate. Lower values are better but are application-dependent. High values may indicate memory latency or branch mispredictions.

MPI

A breakdown of the **67.8%** MPI time:

Time in collective calls	100.0%	<div></div>
Time in point-to-point calls	0.0%	<div></div>
Effective process collective rate	4.34 MB/s	<div></div>
Effective process point-to-point rate	0.00 bytes/s	<div></div>

I/O

A breakdown of the **3.3%** I/O time:

Time in reads	0.0%	<div></div>
Time in writes	100.0%	<div></div>
Effective process read rate	0.00 bytes/s	<div></div>
Effective process write rate	101 MB/s	<div></div>

Most of the time is spent in **write operations** with an average effective transfer rate. It may be possible to achieve faster effective transfer rates using asynchronous file operations.

Threads

A breakdown of how multiple threads were used:

Computation	0.0%	<div></div>
Synchronization	0.0%	<div></div>
Physical core utilization	6.0%	<div></div>
System load	7.0%	<div></div>

No measurable time is spent in multithreaded code.

Physical core utilization is low. Try increasing the number of processes to improve performance.

Memory

Per-process memory usage may also affect scaling:

Mean process memory usage	783 MiB	<div></div>
Peak process memory usage	1.09 GiB	<div></div>
Peak node memory usage	2.0%	<div></div>

The **peak node memory usage** is very low. Running with fewer MPI processes and more data on each process may be more efficient.

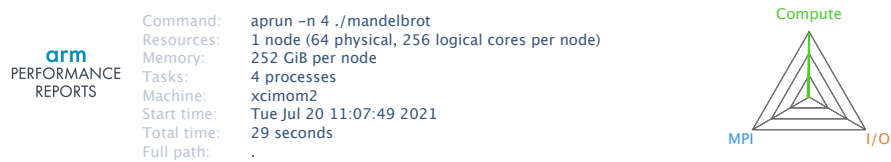
Energy

A breakdown of how energy was used:

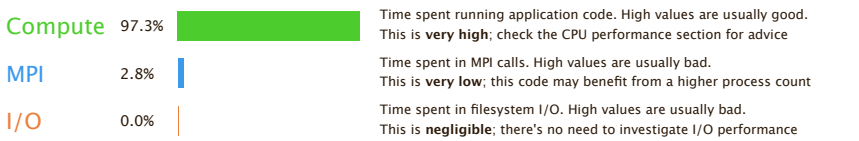
CPU	not supported %	<div></div>
System	not supported %	<div></div>
Mean node power	not supported W	<div></div>
Peak node power	0.00 W	<div></div>

Energy metrics are not available on this system.

Figure 8: ARM Performance Report from test 2 which is like test 1 but with I/O enabled. The additional time spent in I/O causes an exacerbated load imbalance due to I/O being funnelled through the rank zero process which has increased the fraction of time spent in MPI calls.



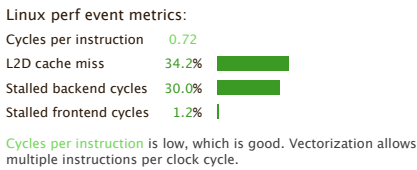
Summary: mandelbrot is **Compute-bound** in this configuration



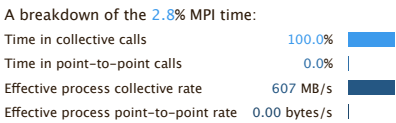
This application run was **Compute-bound**. A breakdown of this time and advice for investigating further is in the **CPU Metrics** section below.

As very little time is spent in **MPI** calls, this code may also benefit from running at larger scales.

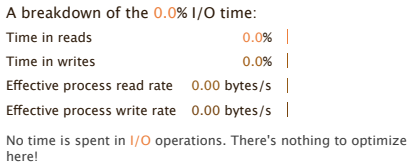
CPU Metrics



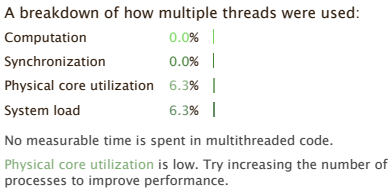
MPI



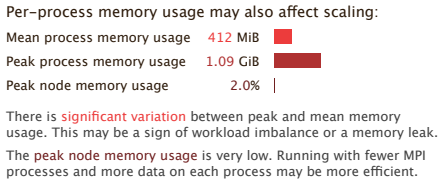
I/O



Threads



Memory



Energy

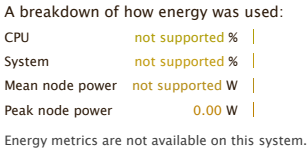
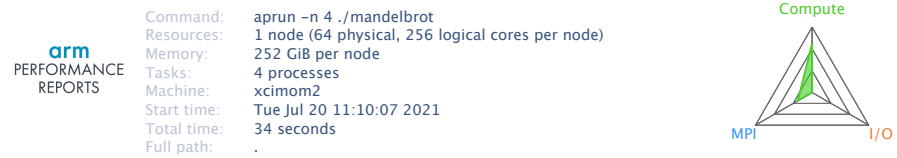
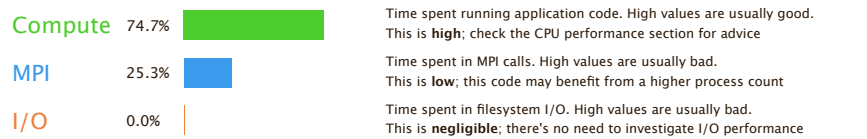


Figure 9: ARM Performance Report from test 3. This version has fixed the load imbalance by interleaving iterations on the real axis.



Summary: mandelbrot is **Compute-bound** in this configuration

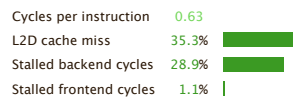


This application run was **Compute-bound**. A breakdown of this time and advice for investigating further is in the **CPU Metrics** section below.

As little time is spent in **MPI** calls, this code may also benefit from running at larger scales.

CPU Metrics

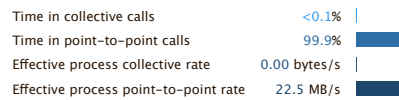
Linux perf event metrics:



Cycles per instruction is low, which is good. Vectorization allows multiple instructions per clock cycle.

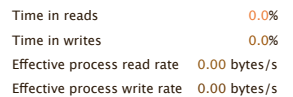
MPI

A breakdown of the **25.3%** MPI time:



I/O

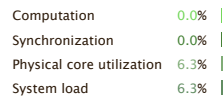
A breakdown of the **0.0%** I/O time:



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

Threads

A breakdown of how multiple threads were used:

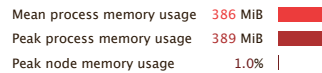


No measurable time is spent in multithreaded code.

Physical core utilization is low. Try increasing the number of processes to improve performance.

Memory

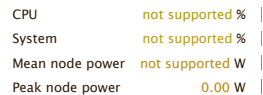
Per-process memory usage may also affect scaling:



The **peak node memory usage** is very low. Running with fewer MPI processes and more data on each process may be more efficient.

Energy

A breakdown of how energy was used:



Energy metrics are not available on this system.

Figure 10: ARM Performance Report from test 4. In the manager-worker pattern only 3 out of 4 MPI processes carry out the calculation and the fourth process is the manager which does not compute any points.

the Performance Report we see that almost all time is spent in point-to-point calls instead of collectives, as expected. We also see a reduction in the peak memory usage as there is no need to populate a large buffer for carrying out communication.

Test 5: in this test the number of MPI processes is increased to 16 (see figure 11) and the fraction of MPI time decreases as there are now 15 out of 16 processes actively working on the calculation instead of 3 out of 4.

4.2.2 MAP

MAP provides more comprehensive information than Performance Reports and if the application is compiled with a `-g` flag then function-level information can be obtained (although this is not required for MAP to work). When profiling an application with MAP modify the job script to load the Forge module (as for performance reports) and modify the `aprun` command by prepending `map --profile` e.g. replace

```
aprun -n ${nprocs} ./mandelbrot
```

with

```
map --profile aprun -n ${nprocs} ./mandelbrot
```

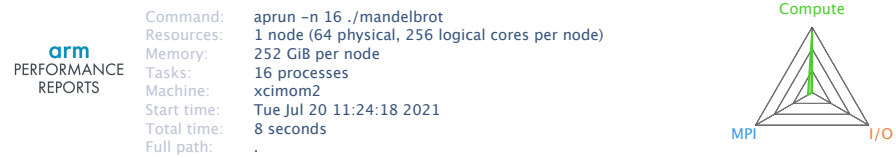
When the application runs a file with a `.map` extension is generated which can be loaded into the map graphical interface:

```
> module load tools/arm-forge
> map mandelbrot_4p_1n_1t_2021-08-05_17-23.map &
```

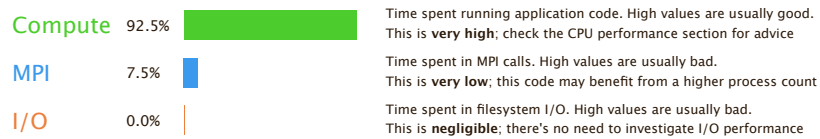
Example output from two of the tests (tests 1 and 5) will be shown to demonstrate the MAP tool. Output from profiling test 1 with MAP is shown in figure 12. This view was set up by selecting **Metrics** → **Preset: MPI** to load the preset view of MPI events. The point-to-point metrics were then switched off (under the **Metrics** menu) as there are no point-to-point calls in this particular example. The “Main thread activity” panel shows the load imbalance. The green bars show application activity and these complete at different times and the MPI call duration ramps up as processes reach the `MPI_Reduce` and are forced to wait⁹. The executable has been compiled with a `-g` flag, which enables mapping back to line numbers on the source code, and in the middle panel we see the line number where the `MPI_Reduce` is called.

Output from profiling test 5 is shown in figure 13 (this time all the MPI metrics have been left switched on). The MPI sent and MPI received time lines show the number of bytes per second passed to MPI send and receive functions respectively. These plots show higher levels of activity at the beginning and end of the calculation when the workers are completing work units more rapidly. At the end of the application we see some time spent in a collective which transfers no data and this is the `MPI_Barrier` at the end of the program.

⁹See <https://developer.arm.com/documentation/101136/2103/MAP/Metrics-view/MPI-calls> for more information on interpreting MAP MPI plots



Summary: mandelbrot is **Compute-bound** in this configuration

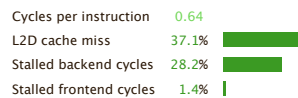


This application run was **Compute-bound**. A breakdown of this time and advice for investigating further is in the **CPU Metrics** section below.

As very little time is spent in **MPI** calls, this code may also benefit from running at larger scales.

CPU Metrics

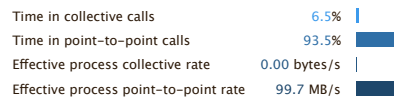
Linux perf event metrics:



Cycles per instruction is low, which is good. Vectorization allows multiple instructions per clock cycle.

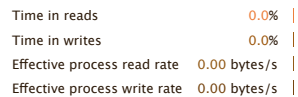
MPI

A breakdown of the 7.5% MPI time:



I/O

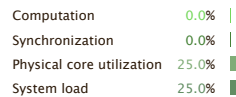
A breakdown of the 0.0% I/O time:



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

Threads

A breakdown of how multiple threads were used:

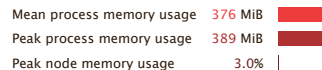


No measurable time is spent in multithreaded code.

Physical core utilization is low. Try increasing the number of processes to improve performance.

Memory

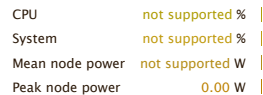
Per-process memory usage may also affect scaling:



The **peak node memory usage** is very low. Running with fewer MPI processes and more data on each process may be more efficient.

Energy

A breakdown of how energy was used:



Energy metrics are not available on this system.

Figure 11: ARM Performance Report from test 5. This test uses the manager-worker version of the program (like test 4) but with 16 processes instead of 4. The MPI fraction is now reduced as the manager process is only one out of sixteen processes instead of one out of three.

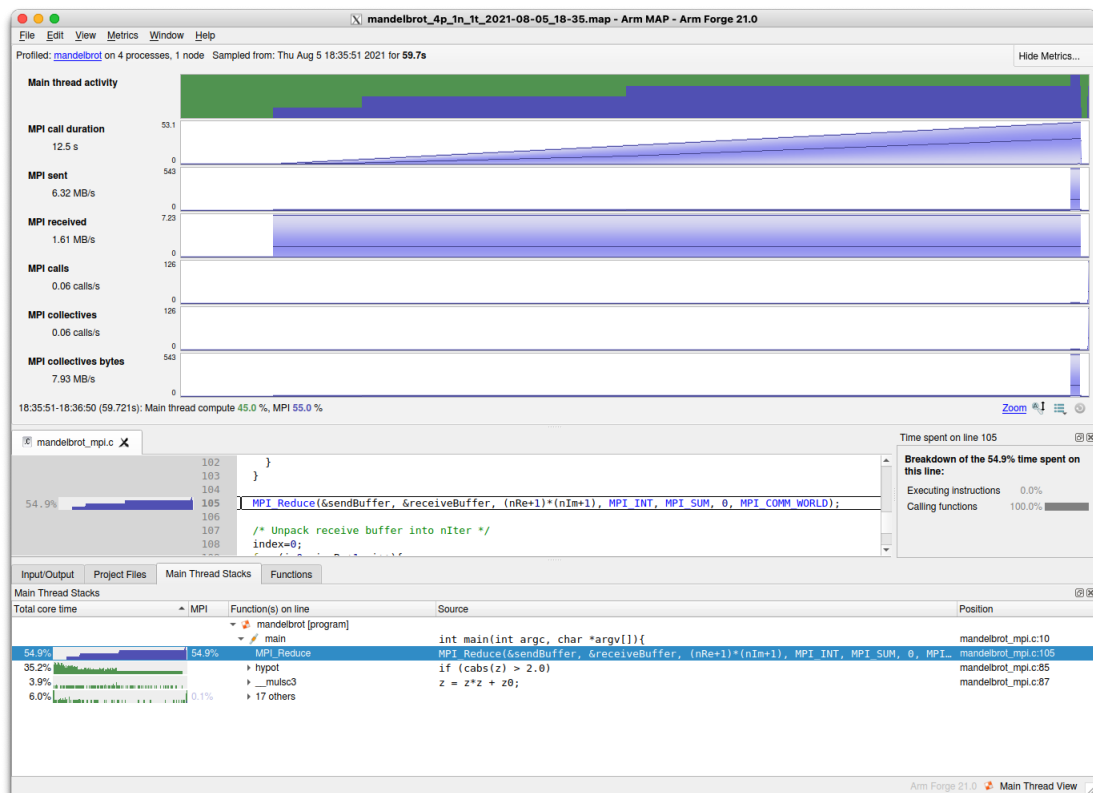


Figure 12: Results from profiling test 1 using MAP. The “Main thread activity” panel shows the load imbalance and we can see the specific line which calls the MPI_Reduce as the executable has been compiled with a `-g` flag.

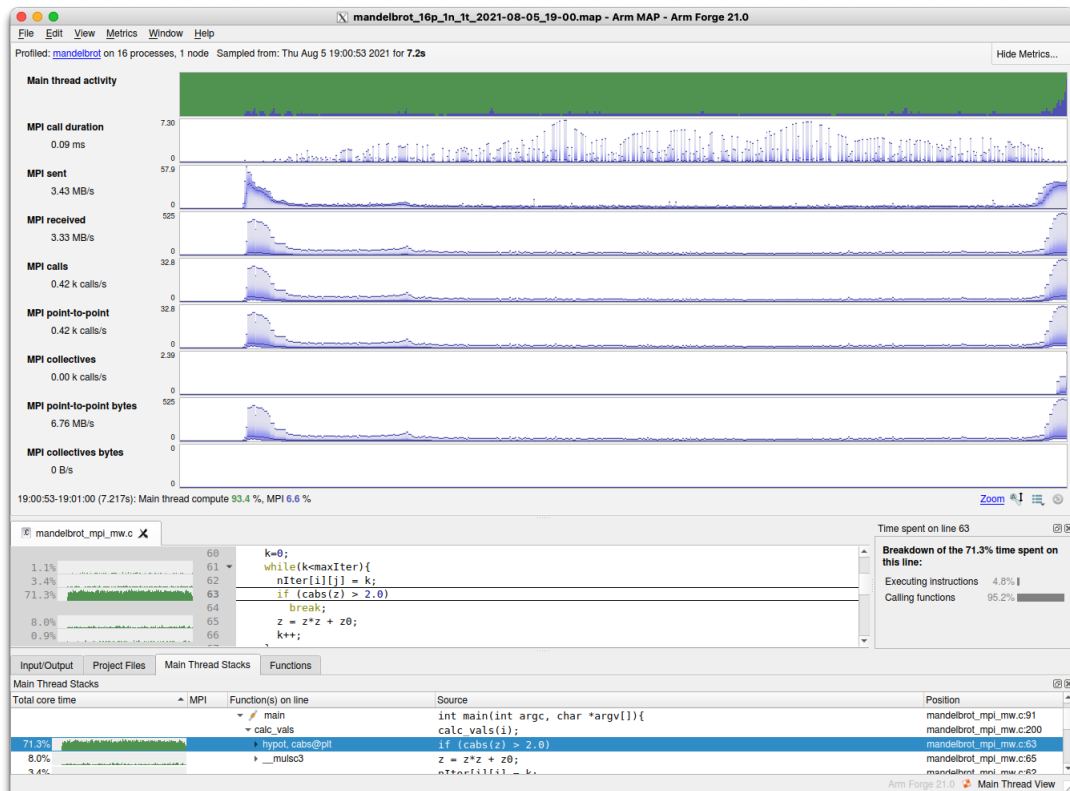


Figure 13: Results from profiling test 5 using MAP. This is a manager-worker pattern which carries out a large number of point-to-point communications.

4.3 Cray PAT

Cray PAT (Performance Analysis Tools) is a suite of profiling tools which are available on Cray systems. Cray PAT has a large number of features¹⁰ but also has a simplified mode called `perftools-lite`. In section tests 1 and 5 will be run using `perftools-lite` to demonstrate this mode of operation.

To enable profiling with `perftools-lite` the `perftools-lite` module is loaded and the executable compiled using as usual. For example to build with the GCC compiler and profiling:

```
> module switch PrgEnv-cray PrgEnv-gnu
> module load perftools-lite
> cc -o mandelbrot mandelbrot_mpi.c
WARNING: PerfTools is saving object files from a temporary directory into directory
'/home/username/.craypat/mandelbrot/178976'
INFO: creating the PerfTools-instrumented executable 'mandelbrot' (lite-samples) ...OK
```

When the `cc` wrapper script is run it reports that that PerfTools is instrumenting the executable as expected. The job can then be submitted to the queue using an unmodified job script (the `perftool-base` module is loaded as part of the programming environment and does not need to be explicitly loaded in the job script). When the job runs profiling information is written into a new directory (e.g. `mandelbrot10863-42s+`) with a summary written to the `stdout` file (see Appendix A for example output). There is a graphical interface called “Apprentice 2” which can be used to view the output in the profiling directory e.g.

```
> app2 mandelbrot+10863-42s &
```

Figure 14 shows the Apprentice 2 overview window from `perftools-lite` profiling of test 1. There is a clear indication of a performance problem shown by a large fraction of time is spent in the Programming Model which in this case is MPI. Looking at the sample profile pie chart (figure 15 we see that around 55% of the samples were in `MPI_Reduce`. The activity window (see figure 16) shows a histogram of activity for each MPI process and we see significant variation in the time spent in collectives which is indicative of the load imbalance in this program. If a full trace is requested then the activity window can switch to a timeline view but this is not available with `perftools-lite`.

The full version of the profiler has many more capabilities than `perftools-lite` including viewing a time line and measuring load imbalance. To use the full capabilities of the profiler the executable is build using the `pat_build` command, the executable is run, and the profiling information generated is processed using the `pat_report` command. One feature of particular interest to the project is the ability to re-order MPI ranks to minimise network traffic by placing frequently communicating processes on the same node which could be of particular relevance when combining spatial and temporal decompositions.

¹⁰See <https://docs.nersc.gov/tools/performance/craypat/>

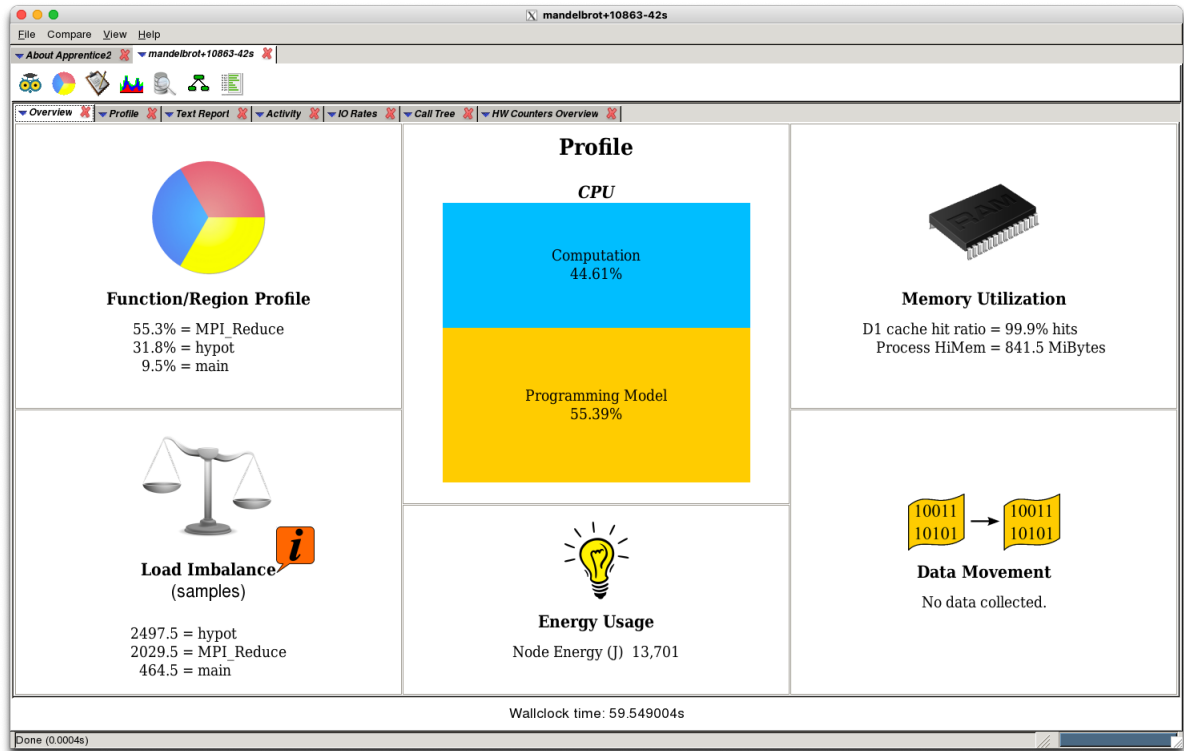


Figure 14: Apprentice 2 overview window from `pertools-lite` profiling of test 1. A large fraction of time is spent in the Programming Model which indicates a performance problem.

5 Builds

There are three primary HPC platforms for the project: Isca, Isambard, Archer-2. Work on the ARCHER-2 build of Firedrake is supported by an Archer-2 eCSE project (ARCHER2-eCSE04-5 PI: Dr David A Ham (Imperial College) “Scalable and robust Firedrake deployment on ARCHER2 and beyond”) and will not be considered further here to avoid duplication of effort.

For each target platform there is a build using the reference BLAS/Lapack implementation (fblaslapack) and optimised builds which use optimised maths libraries appropriate for the platform. The status of the builds is shown in table 3 To force the build to use the reference

Platform	Build	Status	Profiler
Isca	GCC-OpenMPI-fblaslapack	✓	-
Isca	GCC-OpenMPI-OpenBLAS	✓	-
Isca	Intel-IMPI-MKL	X	ITAC
Isambard XCI	GCC-CrayMPI-fblaslapack	✓	perf-report
Isambard XCI	GCC-CrayMPI-CrayLibsci	✓	perf-report

Table 3: Status of Firedrake builds and profiling for the Isca and Isambard

BLAS/lapack implementation an environment variable is set which is used when PETSc is built:

```
export PETSC_CONFIGURE_OPTIONS="--download-fblaslapack=1"
```

The PETSc configuration is reported at the end of the file PETSc configuration log file found at

`firedrake/src/petsc/configure.log`

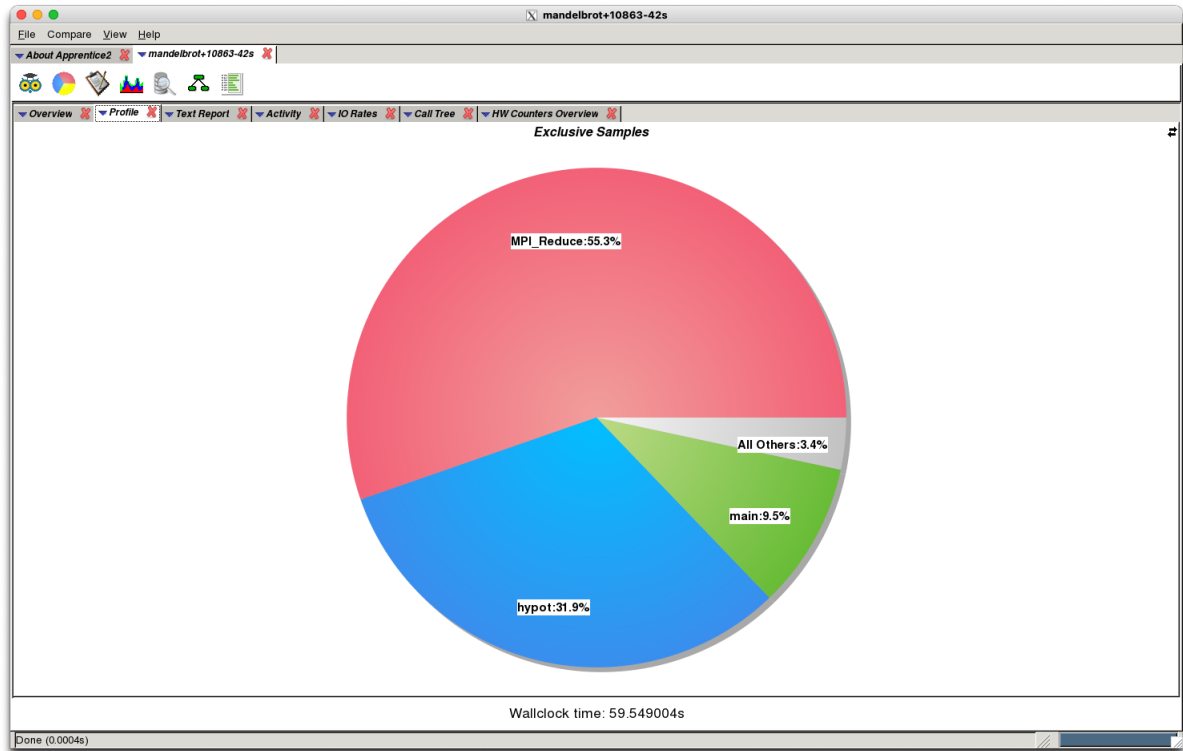


Figure 15: Apprentice 2 sample profile window from `pertools-lite` profiling of test 1. In this window we can see that the time spent in the profiling model is due to an `MPI.Reduce`.

which will confirm the impact of setting the `PETSC_CONFIGURE_OPTIONS` environment variable.

The vendor supplied maths libraries are optimised for the platform under consideration so should offer better performance than the reference implementations. Vendor supplied maths libraries often have support for multi-threading which needs to be switched off when using Firedrake (e.g. by setting `OMP_NUM_THREADS=1`). However newer Firedrake versions issue a warning at run time if threading is not disabled.

Building Firedrake on Isca with the Intel compilers is not currently working because one component of the build (pybind11) requires a newer version of the Intel C++ compiler (v18 or newer). Building Firedrake with the Intel toolchain was previously successful (before the requirement for a newer C++ compiler) and could potentially be fixed by installing newer versions of the compilers (which are now available for free download).

6 Profiling Firedrake

All three tools suites previously discussed can be used to profile Firedrake, however some modifications to usage are required which are described below.

6.1 Intel Trace Analyzer and Collector

ITAC was previously used on Isca with the Intel toolchain (before the requirement for newer compilers). Tracing can be enabled by adding the `-trace` flag to the `mpirun` command which enables Firedrake to be profiled without any modifications to the build.



Figure 16: Apprentice 2 activity window from `pertools-lite` profiling of test 1. In this window we see significant variation in the time spent in collectives which is indicative of a load imbalance.

6.2 ARM Performance Reports and MAP

ARM Forge can be used to profile python applications including Firedrake. It was necessary to add the `-j 1` flag when using ARM profilers on Isambard otherwise the job will hang on exit¹¹ (this appears to be specific to the ARM64 architecture).

The ARM Forge tools were tested using a higher resolution version of the DG advection example from the Firedrake tutorials. Results from Performance Reports and MAP are shown in figures 17 and 18 respectively. In the main thread stacks pane we see where the most time is spent broken down by source code line. The top source code lines are the three linear solvers associated with different stages of the Runge-Kutta time stepping scheme. The next largest contributions are from setting up Firedrake itself and setting up the mesh.

6.3 Cray PAT

Dynamically linked executables can be profiled by adding a `pat_run` command before the executable. For Firedrake this requires adding `pat_run` before the `python` command and appending the full path to the `python` executable e.g.

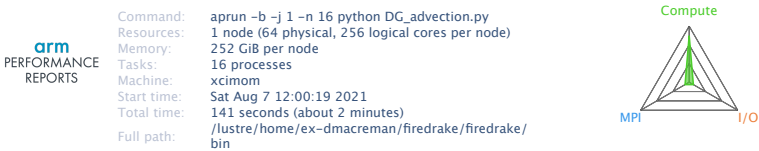
Also mention loading pre-load module to avoid runtime warning

Add results from `/home/ex-dmacreman/fd test/DG big PAT` as proof of concept

7 Conclusions

The build with the Intel toolchain could potentially be fixed by installing newer versions of the compilers which are now available for free download. This would enable the ITAC profiler to be used on Isca. However Isambard is a more suitable platform for the development work required

¹¹<https://developer.arm.com/documentation/101136/2102/Appendix/Known-issues/Arm-MAP>



Summary: python is **Compute-bound** in this configuration

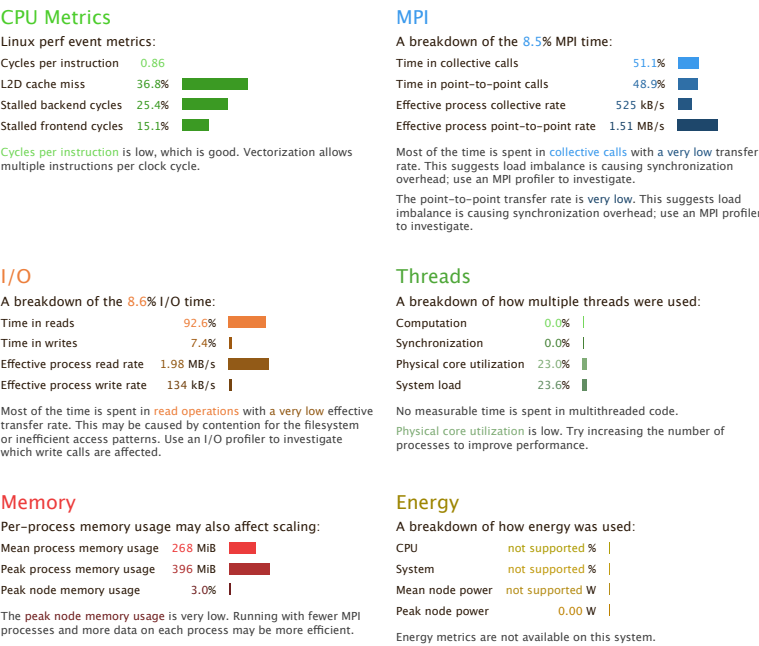
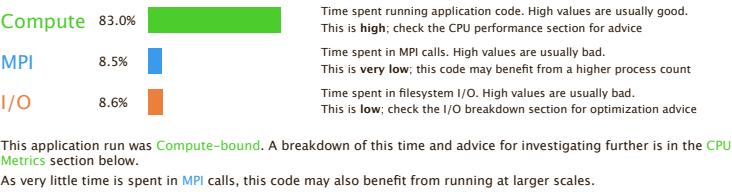


Figure 17: Performance report from the Firedrake DG advection example

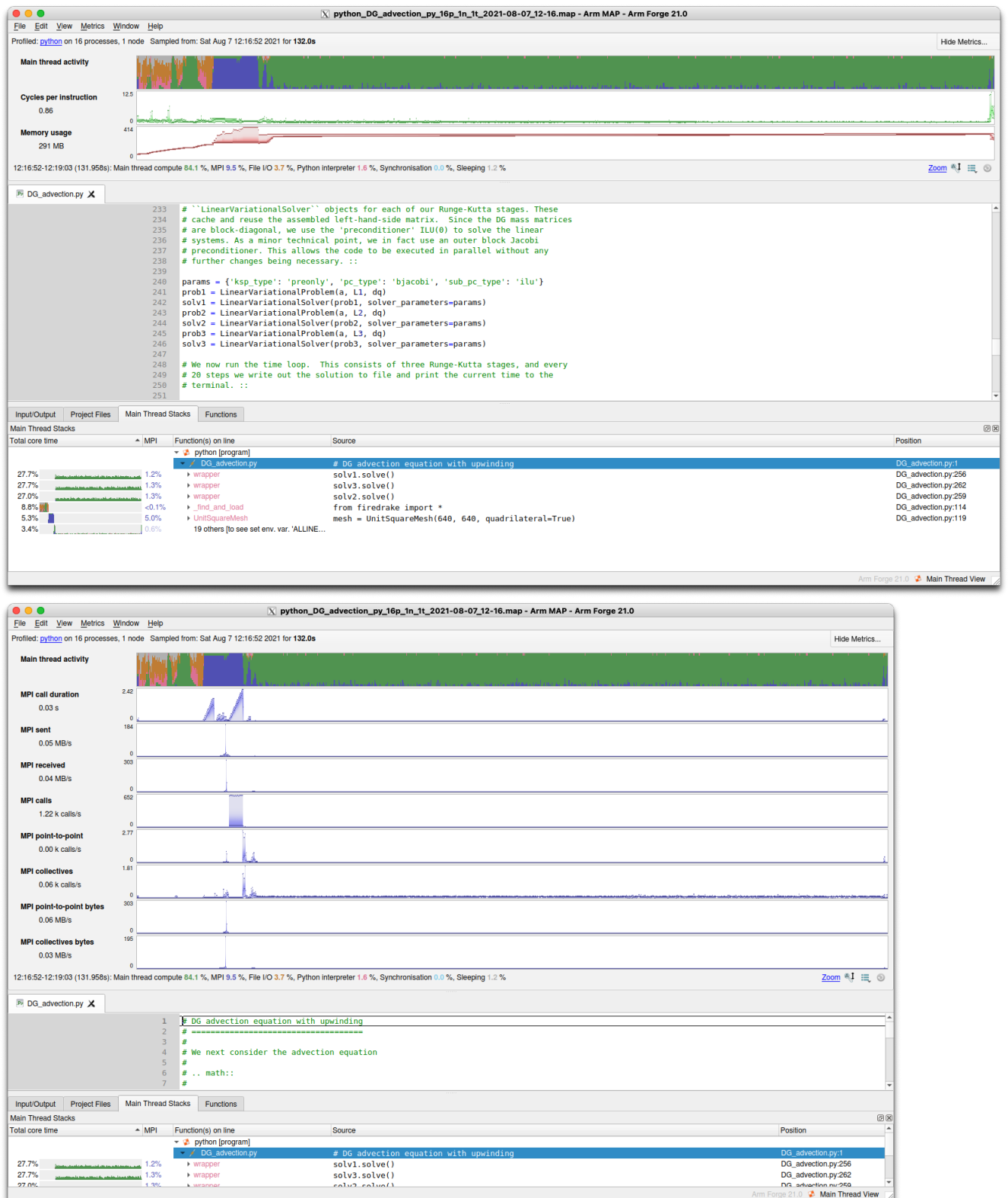


Figure 18: MAP profiling results from the Firedrake DG advection example. Top: main page; bottom: MPI metrics

for the project. Isambard has a shorter maximum wall time than Isca which allows for faster turnaround of multi-node jobs which benefits the workflows required for this project. Profiling on Isambard can use either the ARM Forge tools or the Cray PAT tools, and the tools have complimentary strengths. The ARM tools can profile to the level of individual lines of source code; the Cray tools can generate heat map plots of MPI traffic and load imbalance for specific MPI calls. When scaling up runs for Archer the Cray PAT tools provide shared profiling tool between Isambard and Archer.

Appendix A

```
#####
#
#          CrayPat-lite Performance Statistics
#
#
#####
```

```
CrayPat/X:  Version 21.05.0 Revision 89579db89  04/08/21 23:35:55
Experiment:          lite  lite-samples
Number of PEs (MPI ranks):      4
Numbers of PEs per Node:       4
Numbers of Threads per PE:     1
Number of Cores per Socket:    32
Execution start time:  Thu Aug  5 19:51:20 2021
System name and speed:  nid00042  2.500 GHz (nominal)
ARM   ThunderX2 (new)      CPU
```

```
Avg Process Time:      59.55 secs
High Memory:           3,365.8 MiBytes      841.5 MiBytes per PE
Percent cycles stalled: 33.3 %
Instr per Cycle:       1.33
I/O Write Rate:        1.164873 MiBytes/sec
```

Notes for table 1:

This table shows functions that have significant exclusive sample hits, averaged across ranks.

For further explanation, see the "General table notes" below,
or use: `pat_report -v -O samp_profile ...`

Table 1: Profile by Function

Samp%	Samp	Imb.	Imb.	Group
		Samp	Samp%	Function=[MAX10]
				PE=HIDE
100.0%	5,940.0	--	--	Total

55.4%	3,290.2	--	--	MPI

55.3%	3,284.8	2,029.2	50.9%	MPI_Reduce
=====				
31.8%	1,891.0	2,497.0	75.9%	MATH

31.8%	1,891.0	2,497.0	75.9%	hypot
=====				
9.5%	562.8	464.2	60.3%	USER

9.5%	562.8	464.2	60.3%	main
=====				
3.3%	196.0	--	--	ETC

1.8%	107.5	143.5	76.2%	__mulsc3
1.0%	60.2	76.8	74.7%	__hypot_finite

```
|=====
```

Notes for table 2:

This table shows functions, and line numbers within functions, that have significant exclusive sample hits, averaged across ranks. For further explanation, see the "General table notes" below, or use: `pat_report -v -O samp_profile+src ...`

Table 2: Profile by Group, Function, and Line

Samp%	Samp	Imb.	Imb.	Group
		Samp	Samp%	Function=[MAX10]
				Source
				Line
				PE=HIDE
100.0%	5,940.0	--	--	Total
55.4%	3,290.2	--	--	MPI
55.3%	3,284.8	2,029.2	50.9%	MPI_Reduce
31.8%	1,891.0	2,497.0	75.9%	MATH
31.8%	1,891.0	2,497.0	75.9%	hypot
9.5%	562.8	--	--	USER
9.5%	562.8	--	--	main
3				PAT_test/runs/test1/mandelbrot_mpi.c
4	2.4%	144.0	155.0	69.1% line.85
4	2.0%	120.8	166.2	77.2% line.87
3.3%	196.0	--	--	ETC
1.8%	107.5	--	--	__mulsc3
3				../cray-gcc-9.3.0-202003310324.b29d417121fbe/libgcc/libgcc.so.1
1.0%	60.2	76.8	74.7%	__hypot_finite

Observation: MPI utilization

No suggestions were made because all ranks are on one node.

Notes for table 3:

This table shows energy and power usage for the nodes with the maximum, mean, and minimum usage, as well as the sum of usage over all nodes.

Energy and power for accelerators is also shown, if applicable.

For further explanation, see the "General table notes" below,

or use: `pat_report -v -O program_energy ...`

Table 3: Program energy and power usage (from Cray PM)

Node	Node	Process	PE=HIDE
Energy	Power	Time	
(J)	(W)		

13,701	230.085	59.547615	Total
=====			

Notes for table 4:

This table show the average time and number of bytes written to each output file, taking the average over the number of ranks that wrote to the file. It also shows the number of write operations, and average rates.

For further explanation, see the "General table notes" below, or use: `pat_report -v -O write_stats ...`

Table 4: File Output Stats by Filename

Avg	Avg	Write Rate	Number	Avg	Bytes/	File Name=!x/~/(proc sys)/
Write	Write	MiBytes/sec	of	Writes	Call	PE=HIDE
Time per	MiBytes		Writer	per		
Writer	per		Ranks	Writer		
Rank	Writer			Rank		
	Rank					

0.000051	0.000059	1.164873	1	2.0	31.00	stdout
=====						

Program invocation: `./mandelbrot`

For a complete report with expanded tables and notes, run:

`pat_report /lustre/home/user/PAT_test/runs/test1/mandelbrot+10863-42s`

For help identifying callers of particular functions:

`pat_report -O callers+src /lustre/home/user/PAT_test/runs/test1/mandelbrot+10863-42s`

To see the entire call tree:

`pat_report -O calltree+src /lustre/home/user/PAT_test/runs/test1/mandelbrot+10863-42s`

For interactive, graphical performance analysis, run:

`app2 /lustre/home/user/PAT_test/runs/test1/mandelbrot+10863-42s`

===== End of CrayPat-lite output =====