# Profiling Firedrake for the APinTA-PDEs project

David Acreman

August 5, 2021

# Contents

# 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 suite[1] also available on Isambard.

The software environment on Isca is managed using Easybuild[2] 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 product[3] 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 processes[4].

# 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

[1] ARM 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`

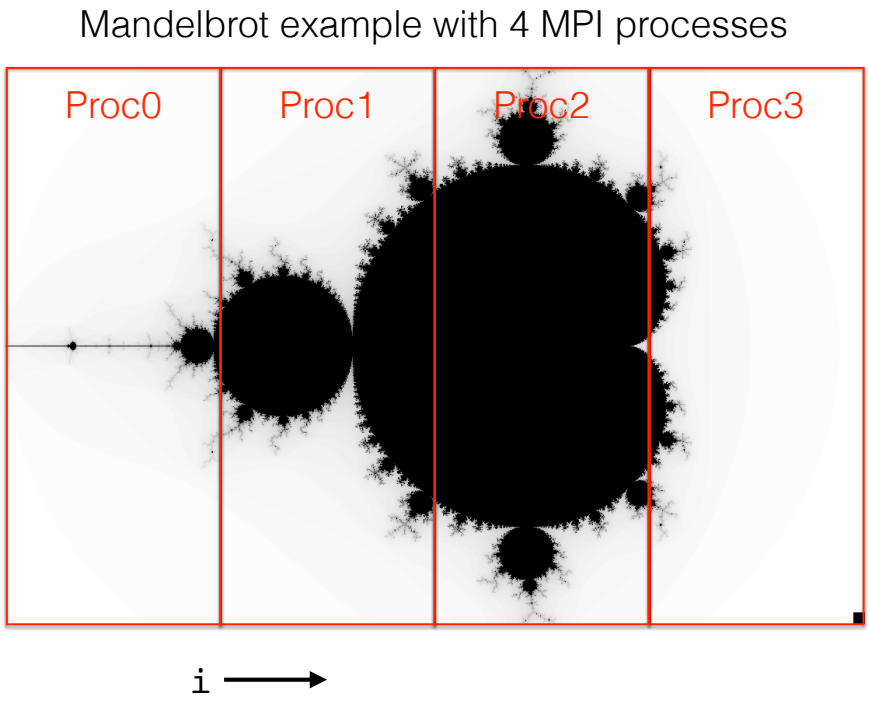



Figure 1: Domain decomposition in version 1 of the Mandelbrot program with four MPI processes. Darker colours show regions of the domain where more iterations are required illustrating the load-imbalance from this decomposition (e.g. process 2 will carry out many more iterations than process 0 or 3).

- 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 Firedrake[5]) and there is also the ability to time sections of code by adding a PyOP2 timed stage[6]. 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
```

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 &
```

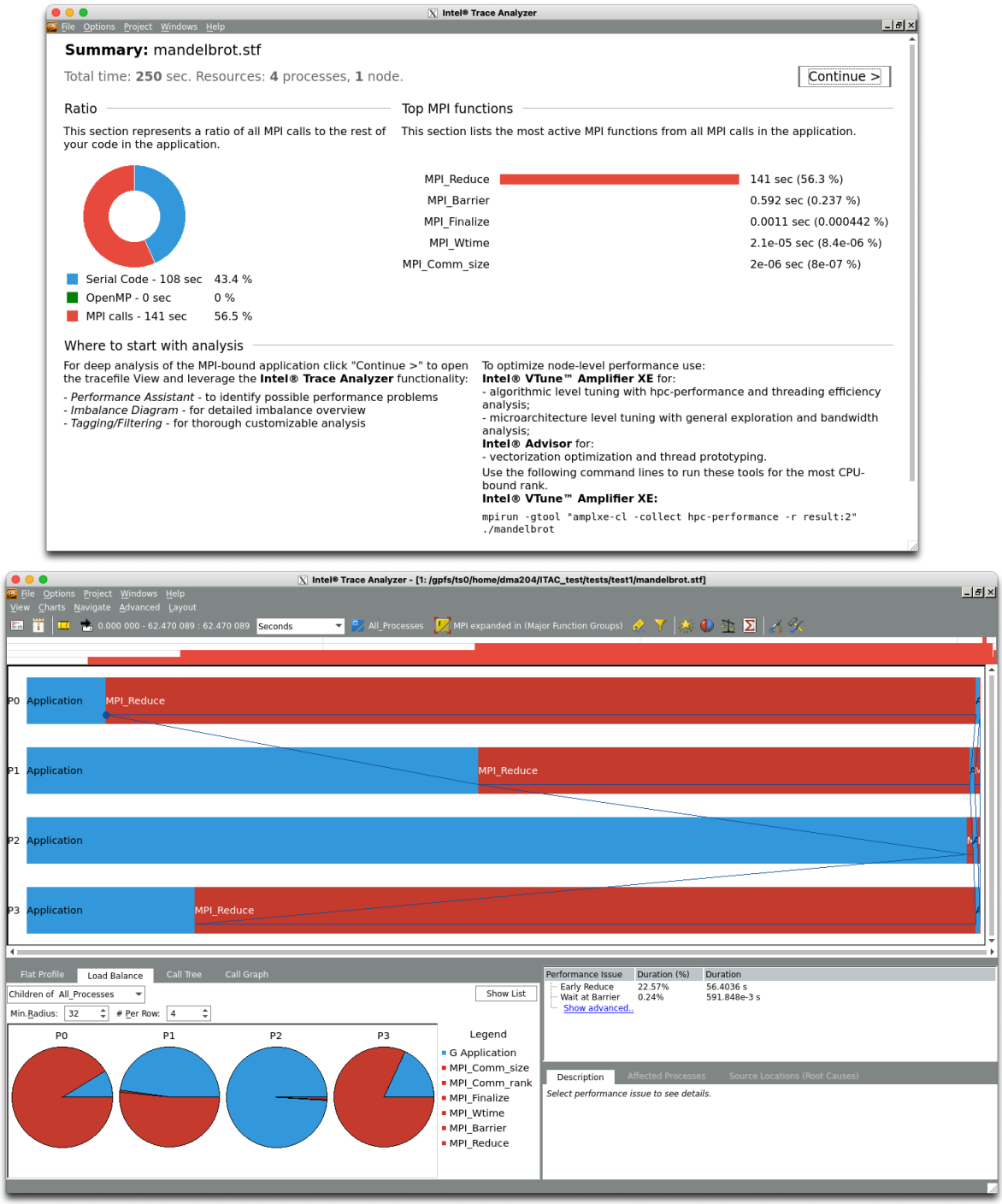



Figure 2: Intel Trace Analyzer plots for test 1. On the summary page (top) the load imbalance shows up as a large fraction of time spent in `MPI_Reduce`. The event timeline (bottom) shows that process 2 takes longer than the other processes to complete its iterations which results in the other processes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

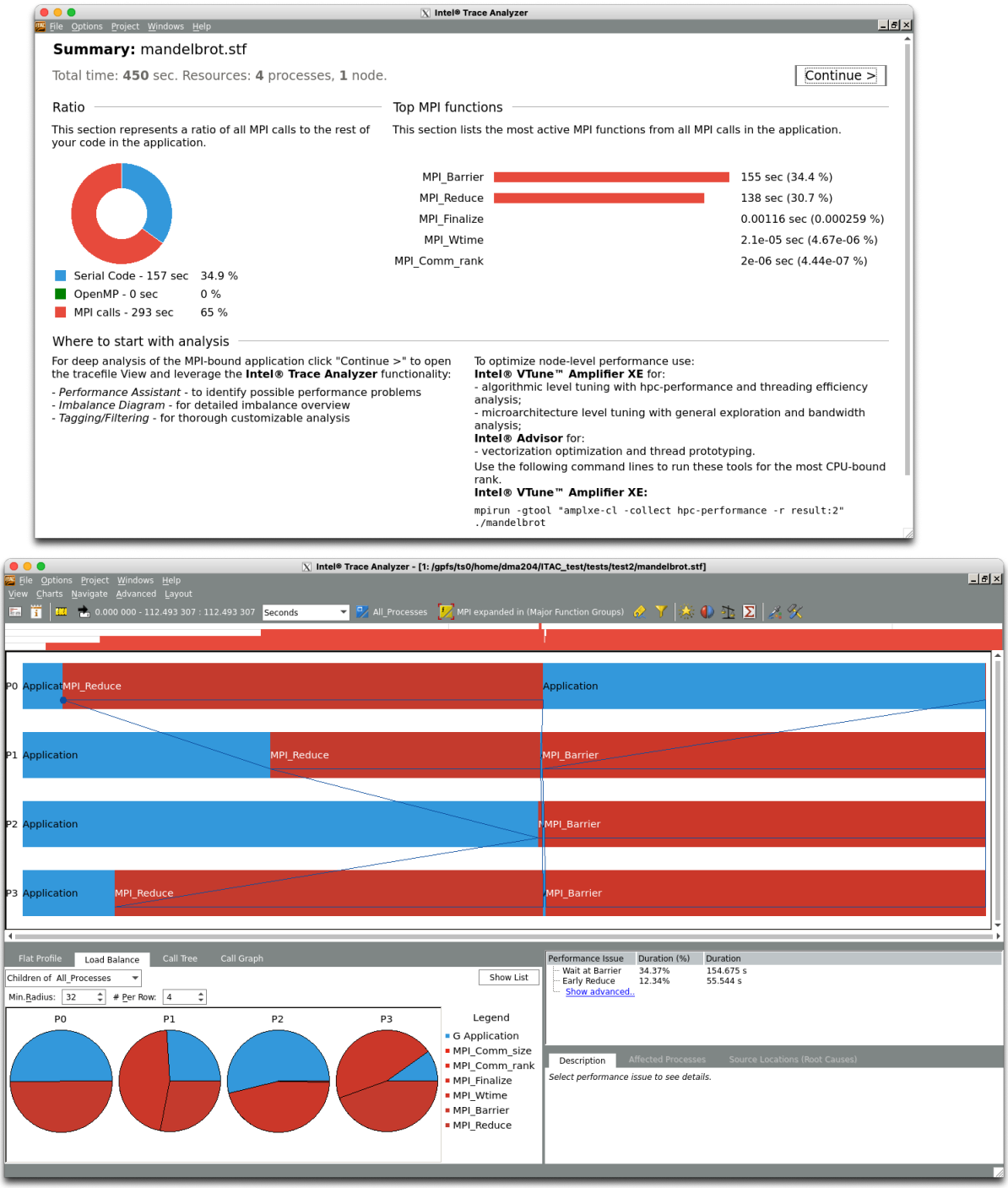



Figure 3: Intel Trace Analyzer plots for test 2 which is like test 1 but with I/O enabled. There are two distinct phases to the time line: firstly the calculation and secondly the I/O. Both phases have a load imbalance with the second phase showing processes waiting at a barrier as the rank zero process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

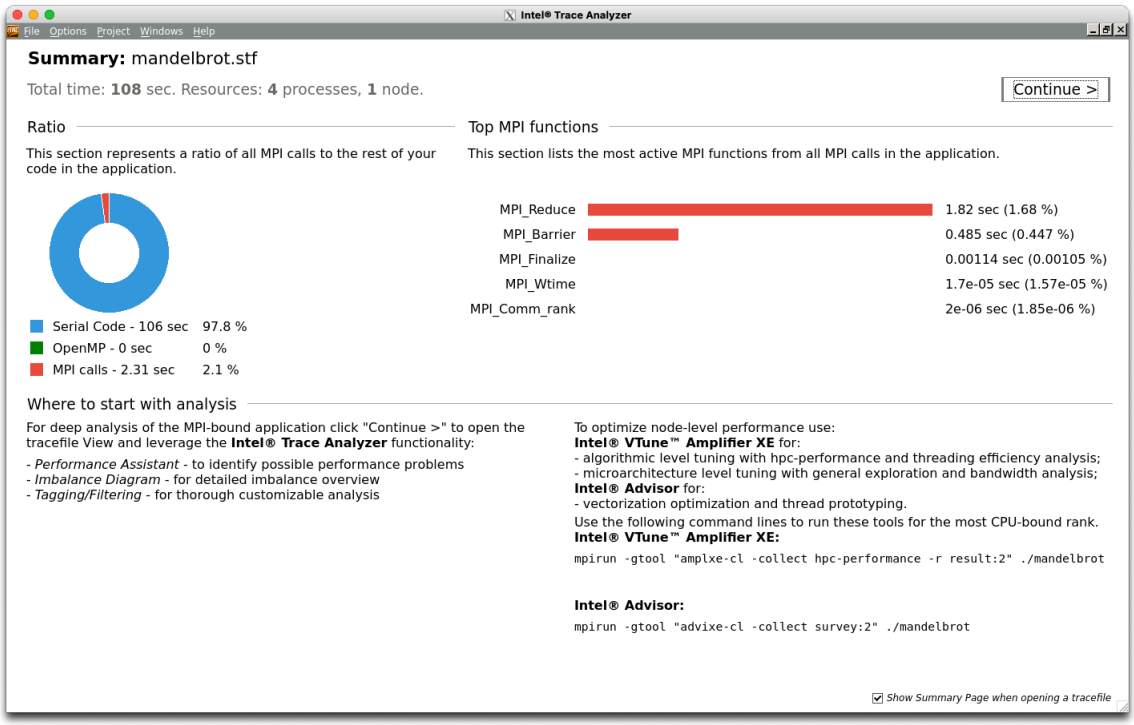



Figure 4: Intel Trace Analyzer plots for test 3. This test uses version 2 of the program which has corrected the load imbalance present in version 1 resulting in all MPI processes reaching the `MPI_Reduce` together.

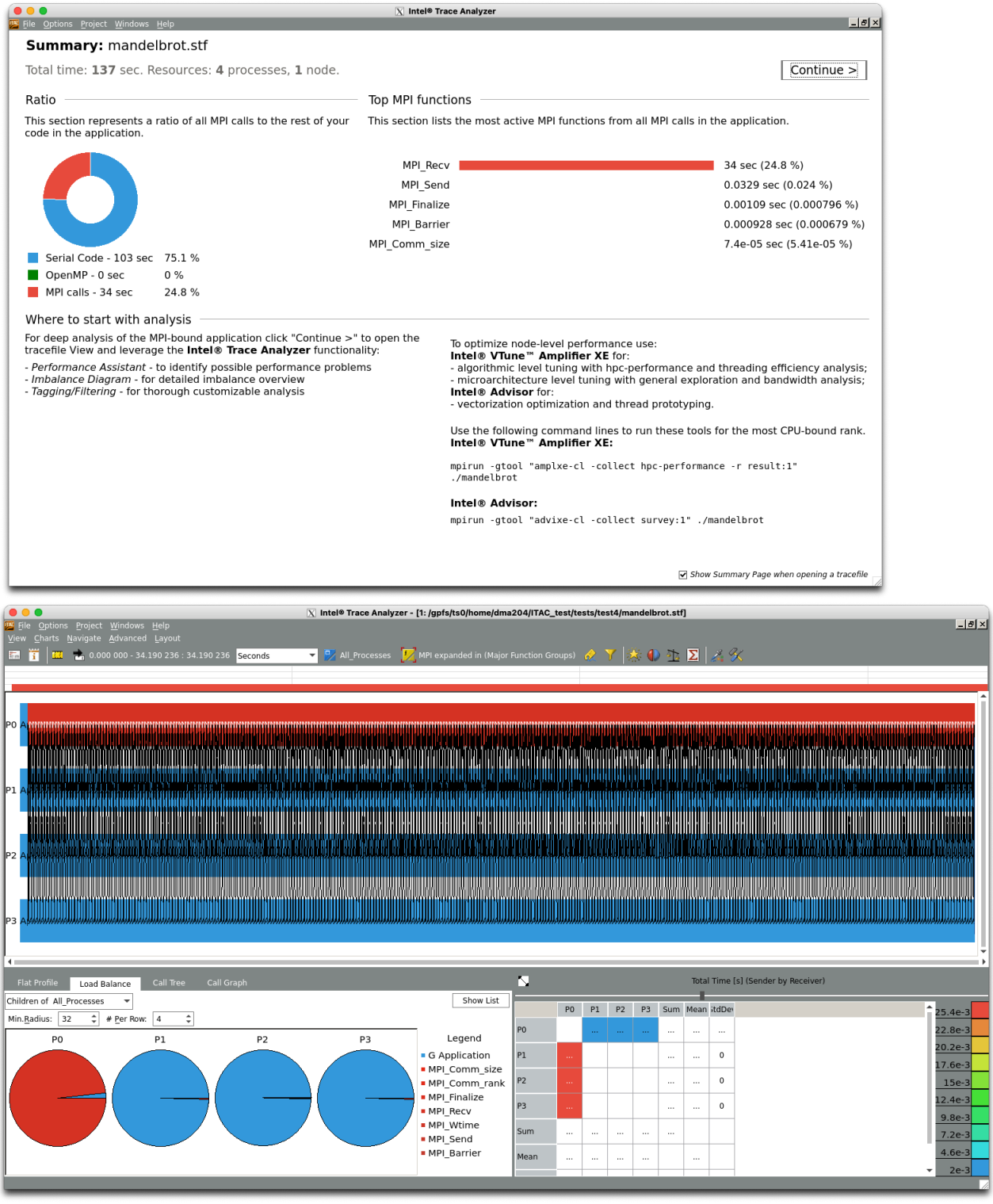
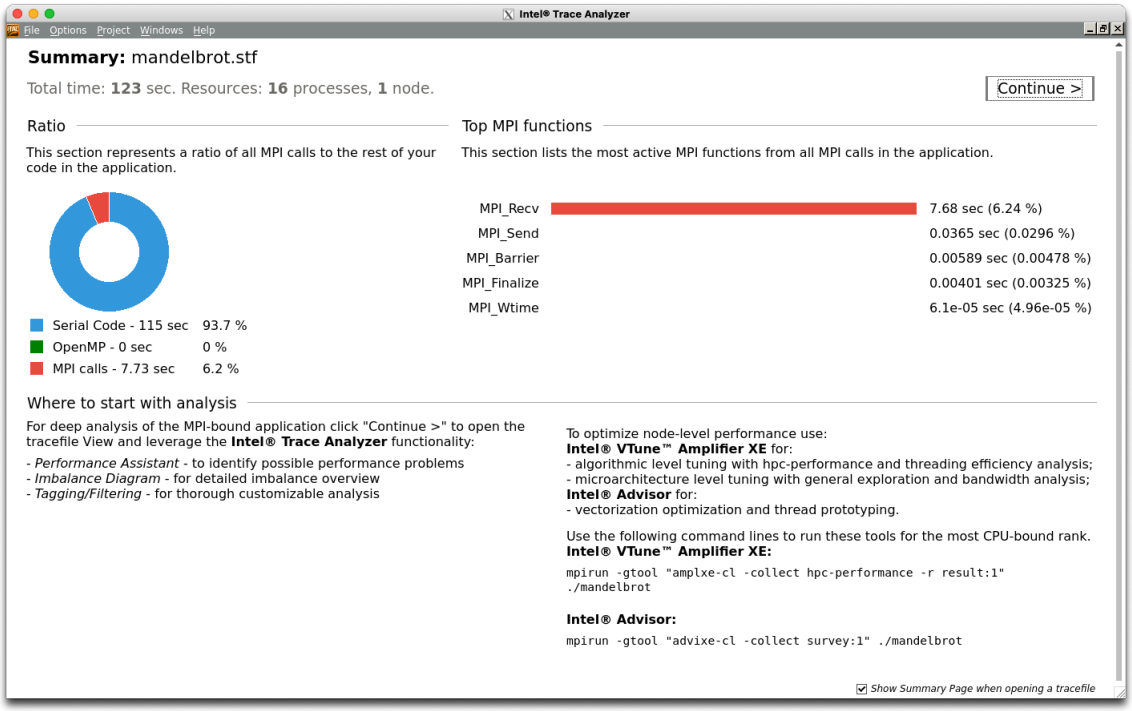



Figure 5: Intel Trace Analyzer plots for test 4. This test uses version 3 of the Mandelbrot program which parallelises the workload using a manager-worker pattern. In this example the communication takes place with a larger number of point-to-point operations instead of a single collective operation.



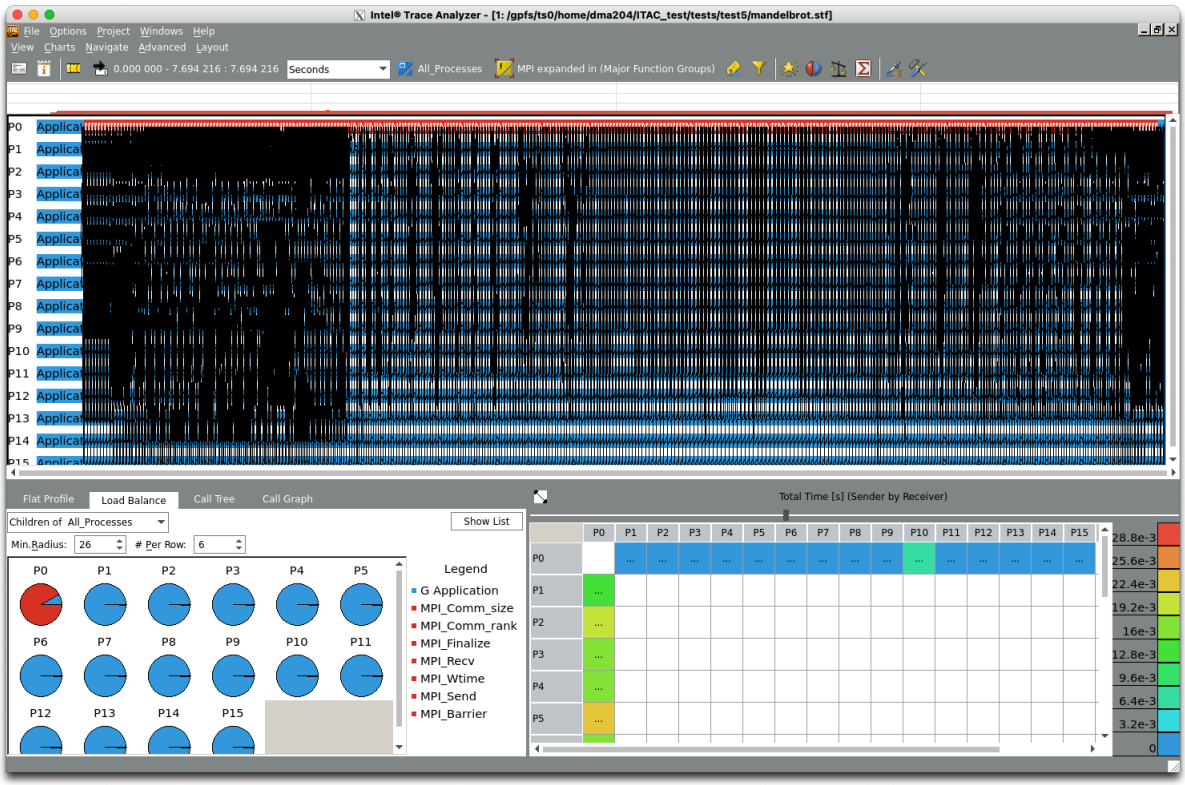




Figure 6: Intel Trace Analyzer plots for test 5. This test uses version 3 of the Mandelbrot program (manager-worker) as for the previous test but uses 16 MPI processes instead of 4 MPI processes.

## 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". ARM MAP is a profiler which is part of the ARM Forge suite and it can profile C++, C, Fortran and Python[7]. The documentation for Performance Reports says that on Cray systems dynamic linking or explicit linking with the profiling libraries is required[8]. 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 simply compile the executable as normal, for example using the GCC compiler:

```
> 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 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.

[7] `https://www.arm.com/products/development-tools/server-and-hpc/forge/map`

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

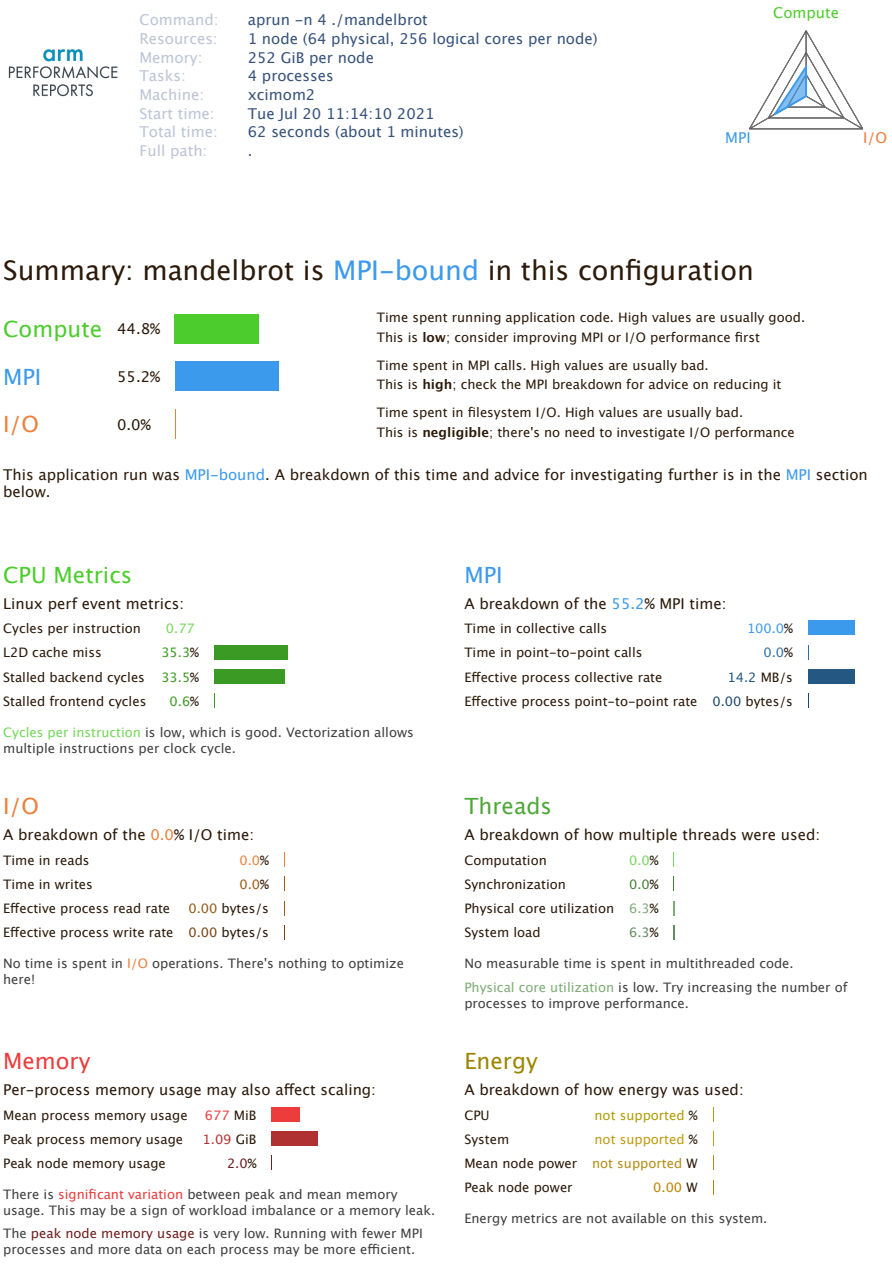



Figure 7: ARM Performance Report from test 1. The tool correctly identifies that there is excessive time spent in MPI calls but does not identify load imbalance as the performance issue.

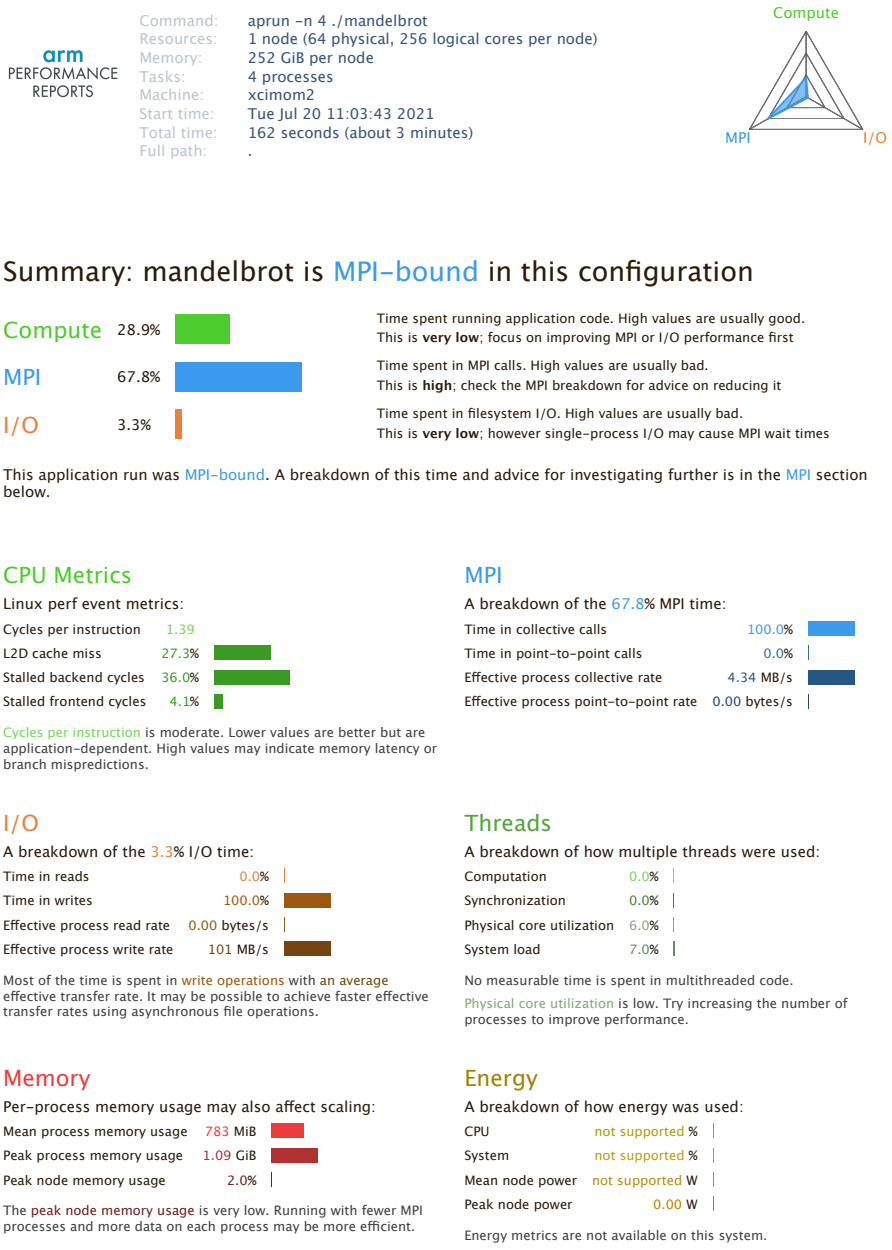



Figure 8: ARM Performance Report from test 2 which is like test 1 but with I/O enabled. The additional time spent in I/O causes an exacerbated load imbalance due to I/O being funnelled through the rank zero process which has increased the fraction of time spent in MPI calls.

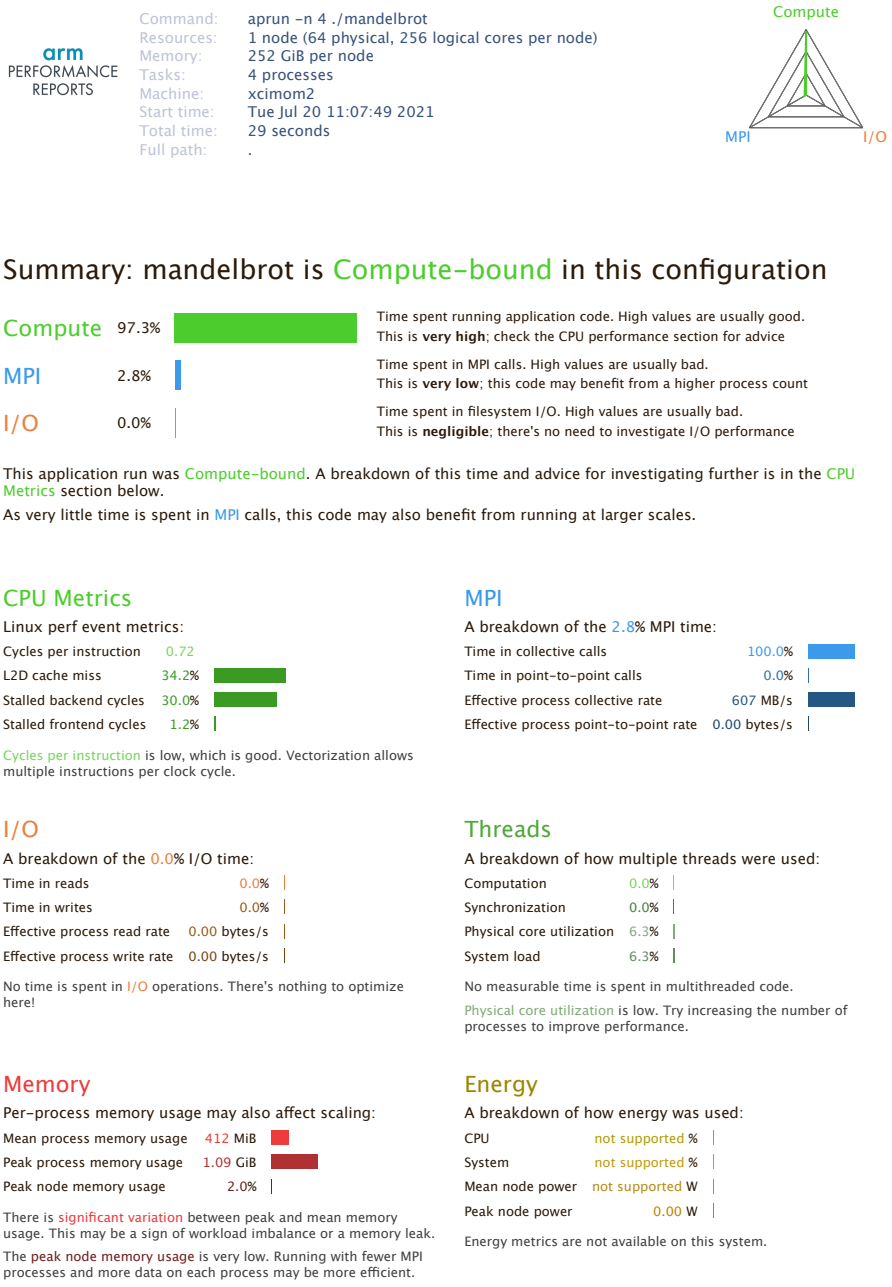



Figure 9: ARM Performance Report from test 3. This version has fixed the load imbalance by interleaving iterations on the real axis.

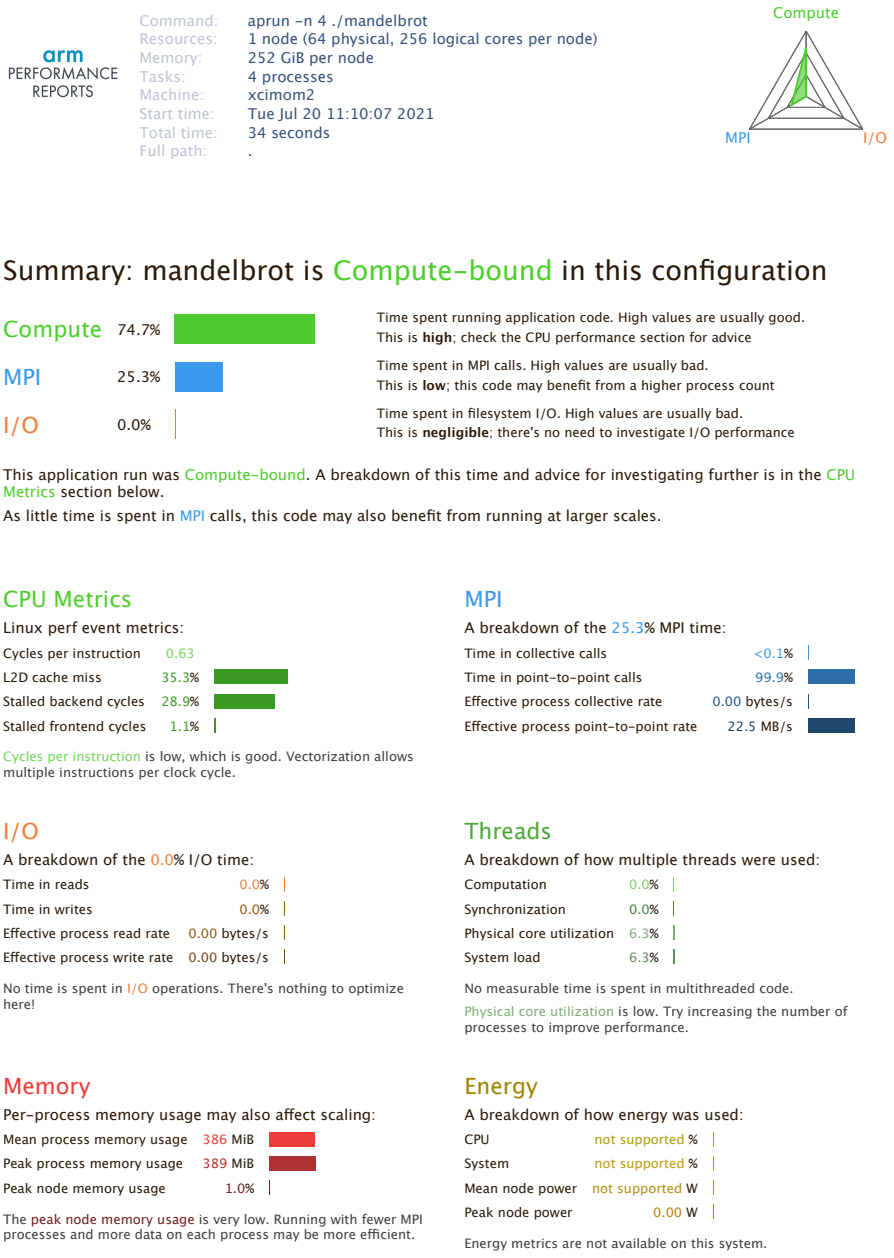



Figure 10: ARM Performance Report from test 4. In the manager-worker pattern only 3 out of 4 MPI processes carry out the calculation and the fourth process is the manager which does not compute any points.

**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.

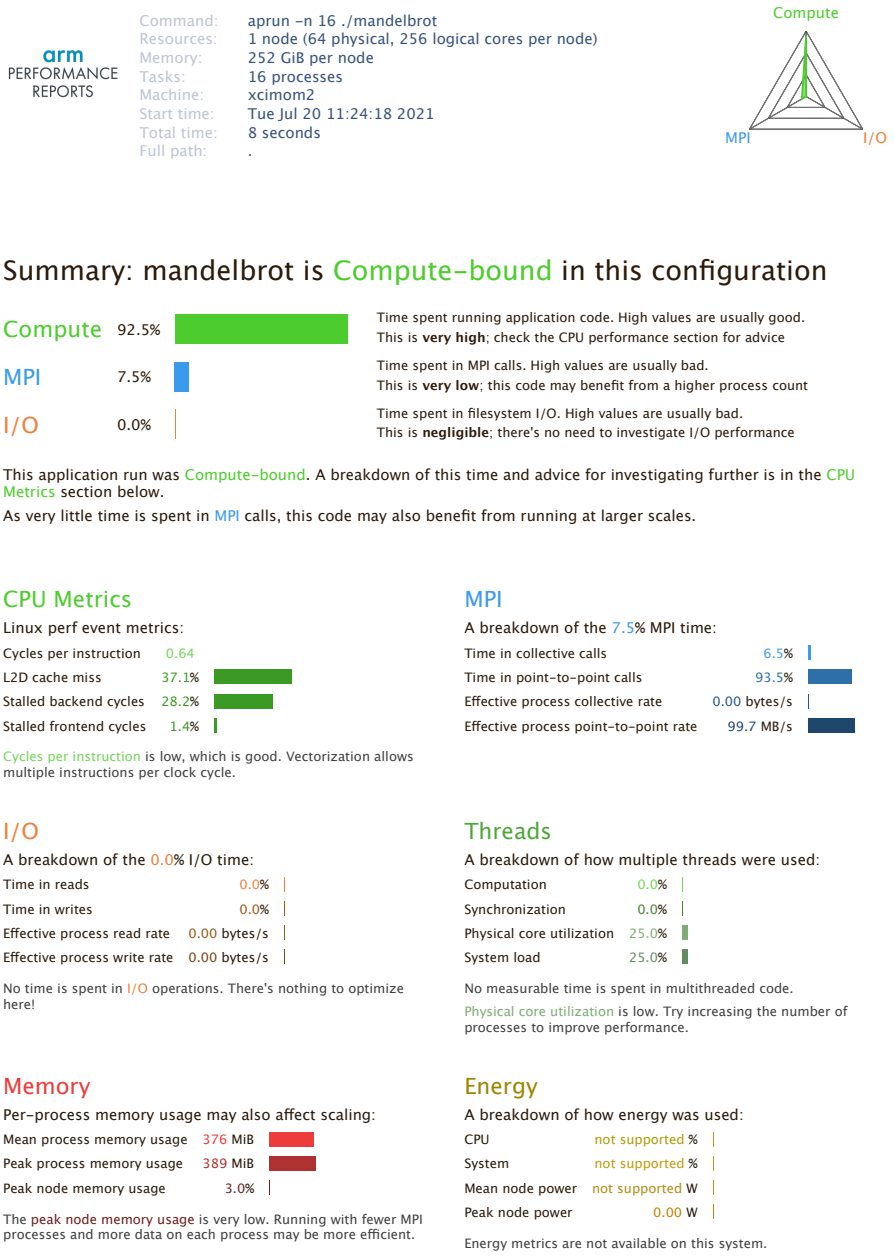



Figure 11: ARM Performance Report from test 5. This test uses the manager-worker version of the program (like test 4) but with 16 processes instead of 4. The MPI fraction is now reduced as the manager process is only one out of sixteen processes instead of one out of three.

### 4.3 Cray PAT

Although this is a Cray tool it does work with compilers other than Cray (the other PrgEnv modules load perftools-base). Experience building SWIFT showed that the `perftools-lite` module can confuse autogen and configure so only load the module prior to the make command.

- Cray PAT has two modes: standard and "lite"
- There is a graphical viewer (Apprentice 2) which reads the profiling output
- Can view a time line from a full trace

See also `https://docs.nersc.gov/tools/performance/craypat/`.

It's not clear how to use Cray PAT with python/Firedrake. Loading the perftools-lite module and using the python from the cray-python module does not produce any profiling output.

## 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

| Platform | Build | Status | Profiler |
|---|---|---|---|
| Isca | GCC-OpenMPI-fblaslapack | ✓ | - |
| Isca | GCC-OpenMPI-OpenBLAS | ✓ | - |
| 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

- Flag to PETSc can be used to switch to reference version of BLAS/Lapack
- To see how PETSc was configured see the end of the file `firedrake/src/petsc/configure.log`
- Vendor supplied maths libraries often have support for multi-threading which needs to be switched off when using Firedrake. Newer Firedrake versions warn about this.
- Building Firedrake on Isca with the Intel compilers is not working because pybind11 requires Intel C++ compiler v18 or newer.
- The Isca build scripts are named `install_firedrake_isca_COMPILER_MPI_MATHLIB.sh`

## 6 Profiling Firedrake

ITAC was previously working on Isca with the Intel **Say how to modify the mpirun command. Can build problems be fixed with newer compiler versions?**

ARM Forge can be used to profile python applications. Make sure to add the `-j 1` flag when using ARM profilers on Isambard otherwise there can be a hang on exit[9] (this appears to be specific to the ARM64 architecture).

**Test with a Firedrake example e.g. high resolution DG advection. This will show point-to-point communication pattern from domain decomposition.**

[9] `https://developer.arm.com/documentation/101136/2102/Appendix/Known-issues/Arm-MAP`

# 7 Conclusions and recommendations

Carry out initial performance studies using Firedrake for rapid prototyping. Use ARM Forge to profile Firedrake on Isambard which enables profiling up to the 20 992 cores. Once the scalability of the algorithm has been established on Isambard implement the algorithm in FRric and run on Archer-2 with profiling using Cray PAT. This will enable performance studies beyond 20 000 cores.
**To do:**

1. Add examples using ARM MAP
2. Builds on Isca: GCC reference, platform optimised Intel toolchain. Describe how to use Easybuild to set up the Intel build if this works.
3. Firedrake examples: Performance Reports/MAP, old ITAC results?
4. Describe other capabilities of Cray PAT in addition to perftools-lite including rank re-ordering
5. Add examples using Cray PAT perftools-lite
6. Write conclusions and recommendations section and describe the next steps with reference to the proposal document