Profiling for the APinTA-PDEs project

David Acreman

July 20, 2021

Contents

1 Introduction **2**

2 Target platforms **2**

3 Profiling tools **3**

3.1 Test cases 3

3.2 ARM Forge 3

3.2.1 Performance Reports 4

3.2.2 MAP 4

3.2.3 Python 4

3.3 Intel Parallel Studio 4

3.4 Cray PAT 10

3.5 Profilers summary 10

4 Builds **10**

5 Conclusions and recommendations **10**

1 Introduction

This document gives an overview of the profiling tools available to the APinTA-PDEs project. Although this report primarily focusses on Firedrake the profiling tools evaluated could be used for profiling other codes used by the project. Section 2 presents the target platforms being considered in this report and reviews the hardware and software available on these systems.

2 Target platforms

The target HPC platforms for the project are Archer-2 (national tier-1), Isambard (national tier-2) and Isca (local tier-3). Isambard has two separate systems based on the ARM64 architecture: the XCI system which is an established production system, and the A64FX system which is a newer system. The A64FX system should be considered a stretch objective as the hardware and software are less well known to us. We have also 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 function as a reference installation as it will use an unmodified Firedrake install script, rather than the more customised installations on the HPC platforms which will use non-standard components, for example MPI libraries and maths libraries. An overview of the hardware in the target platforms is shown in Table 1.

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
Isambard A64FX	Fujitsu A64FX ARM64	48	32 GB HBM	Mellanox Infiniband
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. The Isambard A64FX platform has high bandwidth memory (HBM) which should give better performance than DRAM but has a smaller capacity.

An overview of the software stacks on the target platforms is shown in Table 2. Each HPC platform has the GNU compilers, a compiler from the processor vendor (AMD, Intel, ARM and Fujitsu) and Cray systems also have the Cray compiler[1]. The GNU compilers are the one compiler

System	Compilers	MPI libraries	Maths lib.	Profilers
Archer-2	Cray, GNU, AOCC	Cray MPICH2	Cray libsci	Cray
Isambard XCI	Cray, GNU, ARM	Cray MPICH2	Cray libsci	Cray, ARM
Isambard A64FX	Cray, GNU, ARM, Fujitsu	Cray MVAPICH2	Cray libsci	ARM
Isca	GNU	OpenMPI	OpenBLAS	None
Isca	Intel	Intel	Intel MKL	Intel
Server	GNU	MPICH?	???	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. The A64FX system did not appear to have a perftools module.

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. `PrgEnv-cray` calls the Cray compiler and

[1]On Isambard A64FX run `module use /lustre/software/aarch64/modulefiles` to make extra modules available

`PrgEnv-gnu` calls the GNU compiler). The MPI library and maths library are then linked by the wrapper script. On Cray systems the standard profiling tool is Cray Performance Analysis Tools (PAT) and on Isambard there is also the ARM Forge profiler[2].

The software environment on Isca is managed using Easybuild 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 needs to use Intel MPI.

The server does not currently have any profiling software but an evaluation licence for ARM Forge and/or Intel Parallel Studio could be used to evaluate these tools on the server. There is also the option of installing open source profiling tools on the server.

Recommendation: standardise on the GNU compilers for all platforms. This avoids having to manage too many different builds and takes the compiler out of the performance equation (different GCC versions notwithstanding).

3 Profiling tools

Within Firedrake there are two ways to obtain profiling information:

- Add a PyOP2 timed stage
- Request profiling information from PETSc

These options will be available on all platforms when Firedrake is the target application.

There are also external profilers which are already installed on the target platforms. Different platforms have different profilers available:

- ARM Forge (currently only licensed on Isambard)
- Cray PAT (Cray systems only)
- Intel Parallel Studio (Intel MPI only)

3.1 Test cases

Plan for testing external profilers:

1. Test with the Mandelbrot examples
2. Test with Firedrake if possible

Use the Mandelbrot examples to show whether the profiler can spot load imbalance vs. overhead (version 1 vs. version 2) and can show the difference between collectives and point-to-point communication (version 2 vs version 3).

3.2 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[3].

[2]ARM Forge was previously known as Allinea Forge

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

3.2.1 Performance Reports

The documentation for Performance Reports says that on Cray systems dynamic linking or explicit linking with the profiling libraries is required[4]. 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 this example the GCC compiler will be used:

```
> 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:

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

This method of launching a job under perf-report is referred to as Express Launch mode. When the job runs two extra files are generated which contain the output from the performance report in text and HTML formats. The output from running a performance report on the Mandelbrot version 1 example using 4 MPI processes is shown in figure 1. In this example the I/O has been switched off to highlight the performance issue caused by the load imbalance. The tool correctly identifies that there is excessive time spent in MPI calls but does not identify load imbalance as the performance issue.

For the next test the I/O was switched on and the results are shown in figure 2. The additional time spent in I/O shows up as an exacerbated load imbalance due to I/O being funnelled through the rank zero process which causes 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.

Figure 4 shows the Performance Report from the Mandelbrot version 3 example with 4 MPI processes. This version of the program uses a manager-worker pattern which means that 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. If the number of MPI processes is increases to 16 (see figure 5) then the fraction of MPI time decreases as there are now 15 processes actively working on the calculation.

The Performance Report tool has provided a clear overview of the application performance, which highlighted the presence of performance issues, without needing to change the build process. However was not able to distinguish between load imbalance and overhead which would be clearer when viewing a time line.

3.2.2 MAP

3.2.3 Python

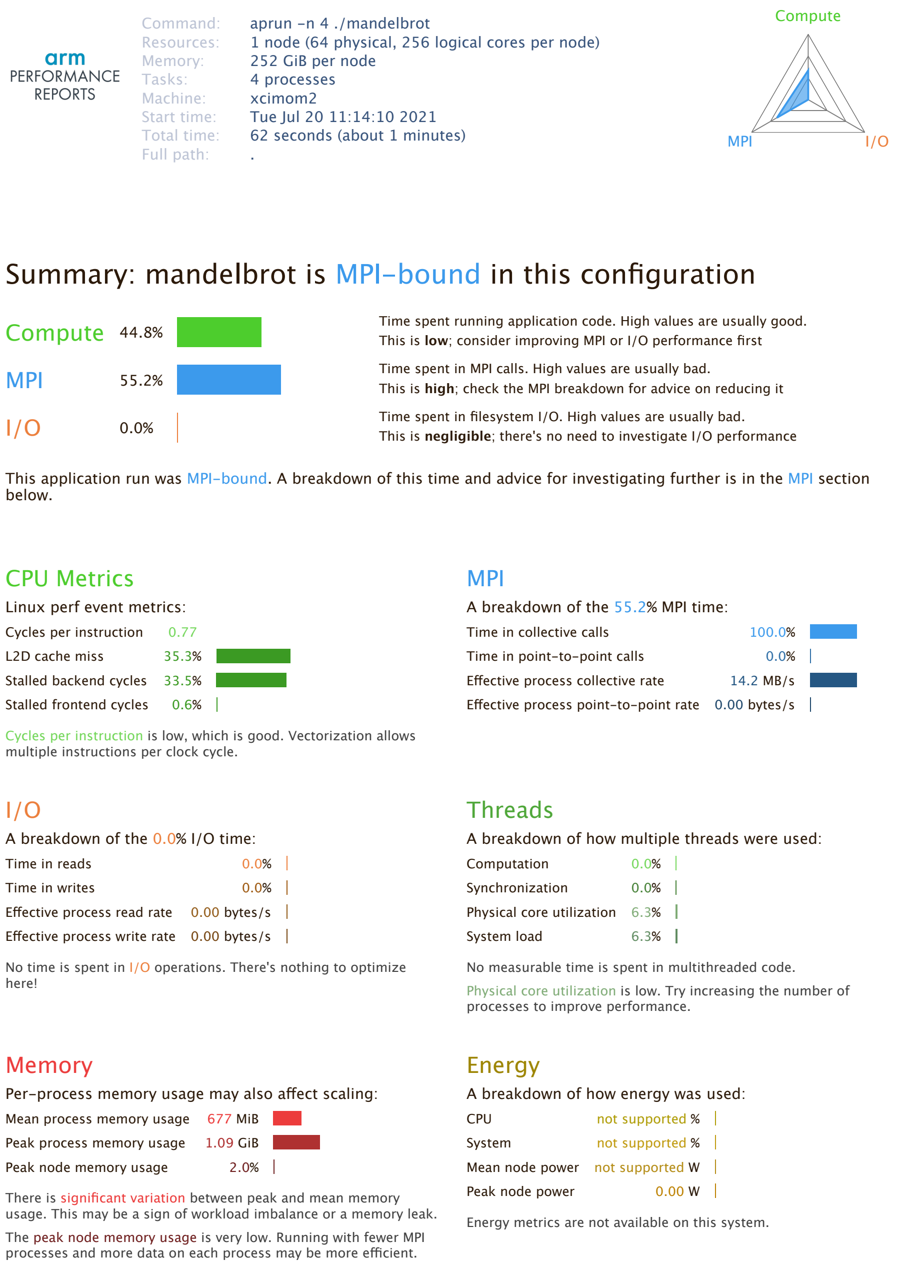
ARM Forge can be used to profile python applications. Make sure to add the `-j 1` flag when using ARM profilers on Isambard otherwise there can be a hang on exit[5] (this appears to be specific to the ARM64 architecture).

3.3 Intel Parallel Studio

Previously working on Isca with the Intel build. Intel MPI is required for the Trace Analyzer and Collector profiler to work. We should be able to use Intel MPI with the GCC compilers.

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

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



Command:	aprun -n 4 ./mandelbrot
Resources:	1 node (64 physical, 256 logical cores per node)
Memory:	252 GiB per node
Tasks:	4 processes
Machine:	xcimom2
Start time:	Tue Jul 20 11:14:10 2021
Total time:	62 seconds (about 1 minutes)
Full path:	.

Summary: mandelbrot is MPI-bound in this configuration

Compute	44.8%	Time spent running application code. High values are usually good. This is **low**; consider improving MPI or I/O performance first
MPI	55.2%	Time spent in MPI calls. High values are usually bad. This is **high**; check the MPI breakdown for advice on reducing it
I/O	0.0%	Time spent in filesystem I/O. High values are usually bad. This is **negligible**; there's no need to investigate I/O performance

This application run was 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	0.77
L2D cache miss	35.3%
Stalled backend cycles	33.5%
Stalled frontend cycles	0.6%

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

MPI

A breakdown of the 55.2% MPI time:

Time in collective calls	100.0%
Time in point-to-point calls	0.0%
Effective process collective rate	14.2 MB/s
Effective process point-to-point rate	0.00 bytes/s

I/O

A breakdown of the 0.0% I/O time:

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

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

Threads

A breakdown of how multiple threads were used:

Computation	0.0%
Synchronization	0.0%
Physical core utilization	6.3%
System load	6.3%

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	677 MiB
Peak process memory usage	1.09 GiB
Peak node memory usage	2.0%

There is significant variation between peak and mean memory usage. This may be a sign of workload imbalance or a memory leak.

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

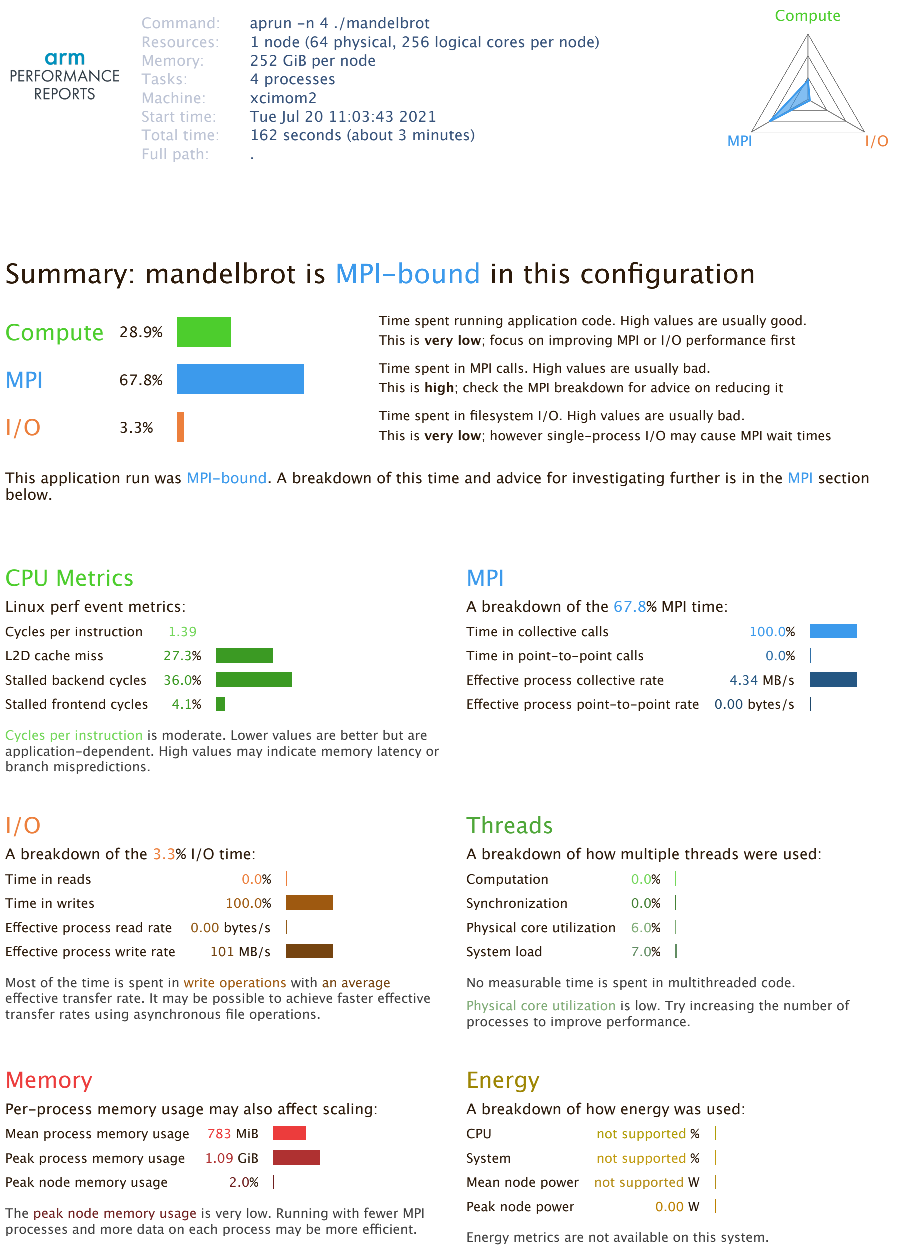
Energy

A breakdown of how energy was used:

CPU	not supported %
System	not supported %
Mean node power	not supported W
Peak node power	0.00 W

Energy metrics are not available on this system.

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



Command:	aprun -n 4 ./mandelbrot
Resources:	1 node (64 physical, 256 logical cores per node)
Memory:	252 GiB per node
Tasks:	4 processes
Machine:	xcimom2
Start time:	Tue Jul 20 11:03:43 2021
Total time:	162 seconds (about 3 minutes)
Full path:	.

Summary: mandelbrot is MPI-bound in this configuration

Compute	28.9%	Time spent running application code. High values are usually good. This is **very low**; focus on improving MPI or I/O performance first
MPI	67.8%	Time spent in MPI calls. High values are usually bad. This is **high**; check the MPI breakdown for advice on reducing it
I/O	3.3%	Time spent in filesystem I/O. High values are usually bad. This is **very low**; however single-process I/O may cause MPI wait times

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
L2D cache miss	27.3%
Stalled backend cycles	36.0%
Stalled frontend cycles	4.1%

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

I/O

A breakdown of the 3.3% I/O time:

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

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%
Synchronization	0.0%
Physical core utilization	6.0%
System load	7.0%

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
Peak process memory usage	1.09 GiB
Peak node memory usage	2.0%

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

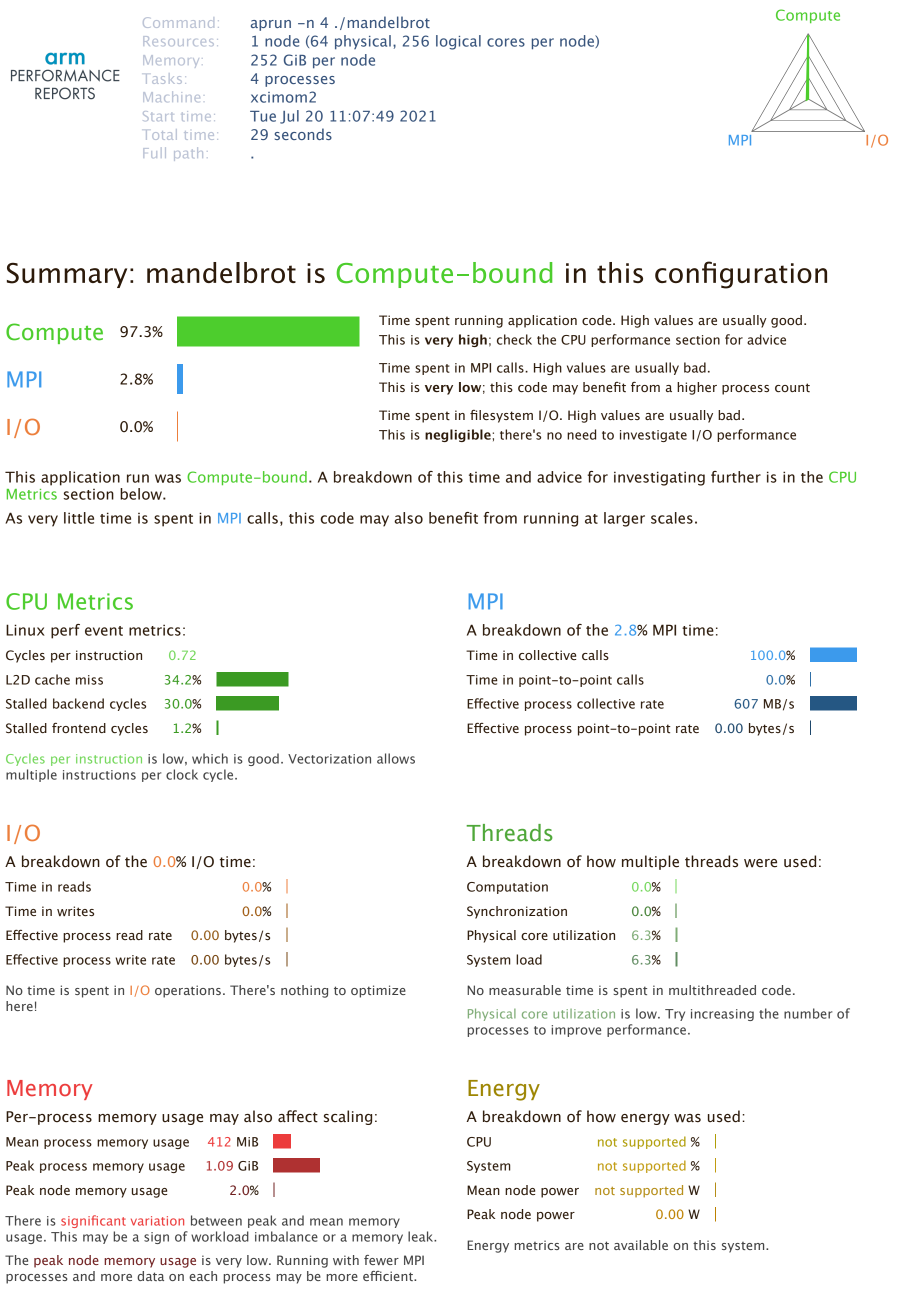
Energy

A breakdown of how energy was used:

CPU	not supported %
System	not supported %
Mean node power	not supported W
Peak node power	0.00 W

Energy metrics are not available on this system.

Figure 2: ARM Performance Report from the Mandelbrot version 1 example with I/O switched on. The additional time spent in I/O shows up as an exacerbated load imbalance due to I/O being funnelled through the rank zero process.



Command:	aprun -n 4 ./mandelbrot
Resources:	1 node (64 physical, 256 logical cores per node)
Memory:	252 GiB per node
Tasks:	4 processes
Machine:	xcimom2
Start time:	Tue Jul 20 11:07:49 2021
Total time:	29 seconds
Full path:	.

Summary: mandelbrot is Compute-bound in this configuration

Compute	97.3%	Time spent running application code. High values are usually good. This is **very high**; check the CPU performance section for advice
MPI	2.8%	Time spent in MPI calls. High values are usually bad. This is **very low**; this code may benefit from a higher process count
I/O	0.0%	Time spent in filesystem I/O. High values are usually bad. This is **negligible**; there's no need to investigate I/O performance

This application run was Compute-bound. A breakdown of this time and advice for investigating further is in the CPU 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	0.72
L2D cache miss	34.2%
Stalled backend cycles	30.0%
Stalled frontend cycles	1.2%

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

MPI

A breakdown of the 2.8% MPI time:

Time in collective calls	100.0%
Time in point-to-point calls	0.0%
Effective process collective rate	607 MB/s
Effective process point-to-point rate	0.00 bytes/s

I/O

A breakdown of the 0.0% I/O time:

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

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

Threads

A breakdown of how multiple threads were used:

Computation	0.0%
Synchronization	0.0%
Physical core utilization	6.3%
System load	6.3%

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	412 MiB
Peak process memory usage	1.09 GiB
Peak node memory usage	2.0%

There is significant variation between peak and mean memory usage. This may be a sign of workload imbalance or a memory leak.
The peak node memory usage is very low. Running with fewer MPI processes and more data on each process may be more efficient.

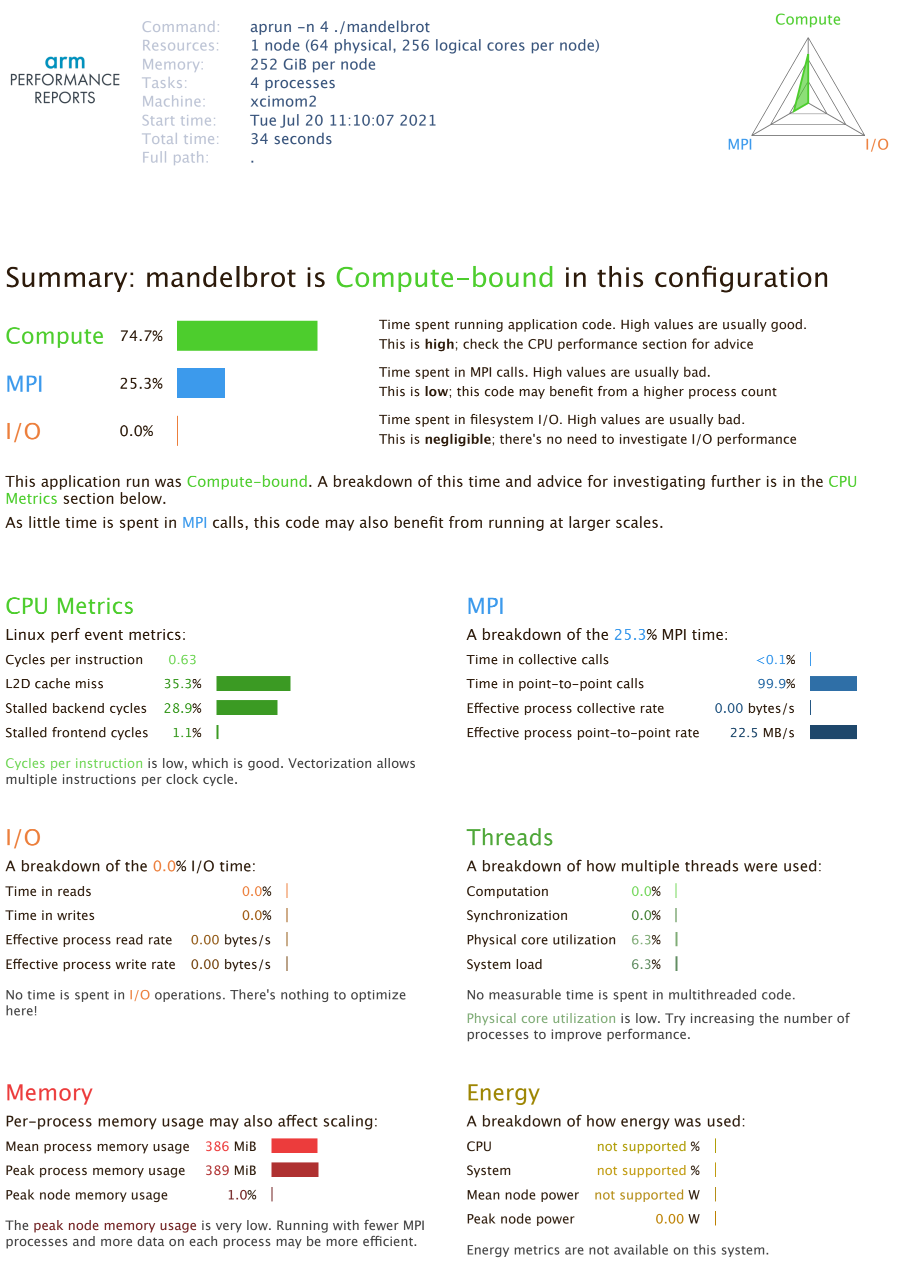
Energy

A breakdown of how energy was used:

CPU	not supported %
System	not supported %
Mean node power	not supported W
Peak node power	0.00 W

Energy metrics are not available on this system.

Figure 3: ARM Performance Report from the Mandelbrot version 2 example. This version has fixed the load imbalance by interleaving iterations on the real axis.



Command:	aprun -n 4 ./mandelbrot
Resources:	1 node (64 physical, 256 logical cores per node)
Memory:	252 GiB per node
Tasks:	4 processes
Machine:	xcimom2
Start time:	Tue Jul 20 11:10:07 2021
Total time:	34 seconds
Full path:	.

Summary: mandelbrot is Compute-bound in this configuration

Compute	74.7%	Time spent running application code. High values are usually good. This is **high**; check the CPU performance section for advice
MPI	25.3%	Time spent in MPI calls. High values are usually bad. This is **low**; this code may benefit from a higher process count
I/O	0.0%	Time spent in filesystem I/O. High values are usually bad. This is **negligible**; there's no need to investigate I/O performance

This application run was Compute-bound. A breakdown of this time and advice for investigating further is in the CPU 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	0.63
L2D cache miss	35.3%
Stalled backend cycles	28.9%
Stalled frontend cycles	1.1%

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

MPI

A breakdown of the 25.3% MPI time:

Time in collective calls	<0.1%
Time in point-to-point calls	99.9%
Effective process collective rate	0.00 bytes/s
Effective process point-to-point rate	22.5 MB/s

I/O

A breakdown of the 0.0% I/O time:

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

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

Threads

A breakdown of how multiple threads were used:

Computation	0.0%
Synchronization	0.0%
Physical core utilization	6.3%
System load	6.3%

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	386 MiB
Peak process memory usage	389 MiB
Peak node memory usage	1.0%

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

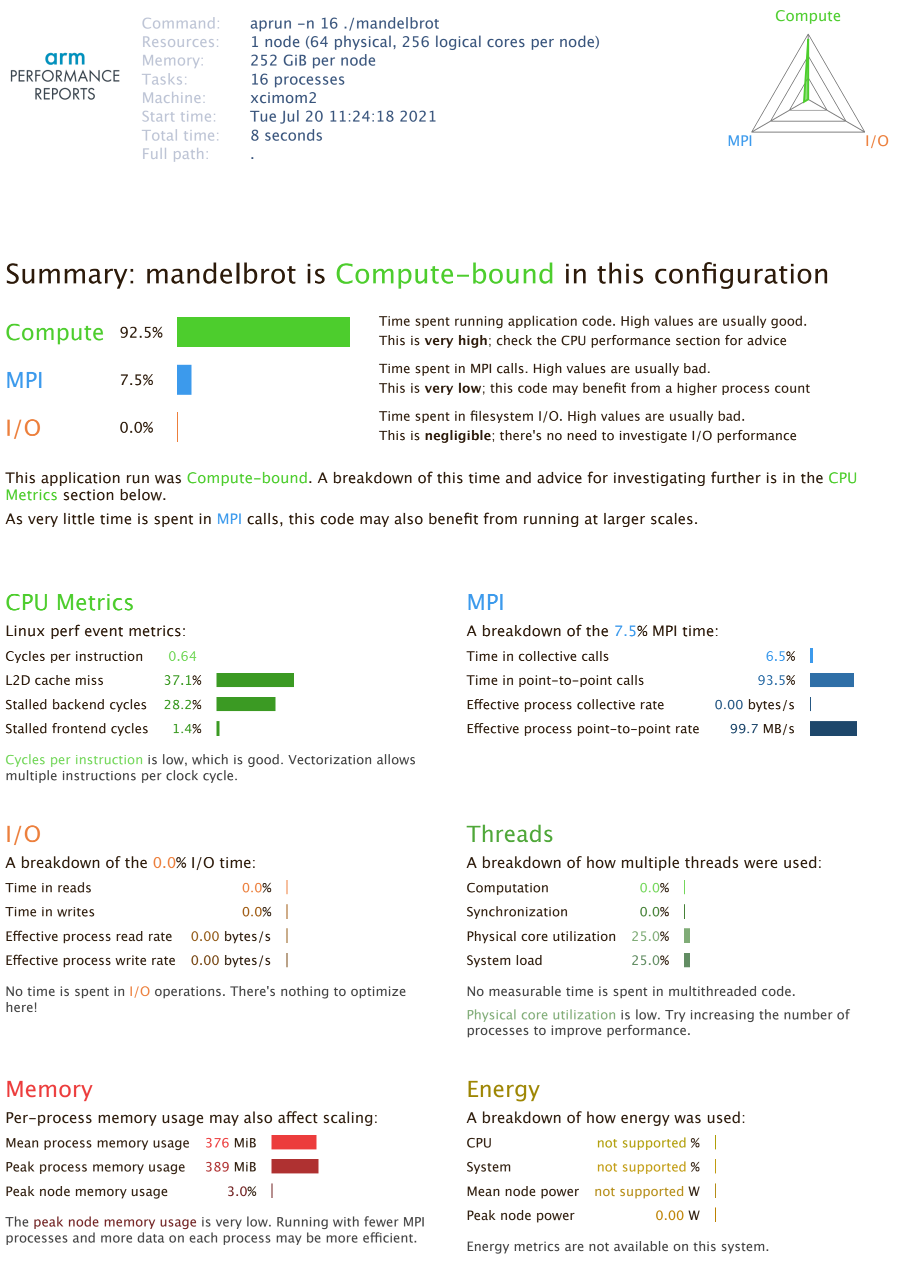
Energy

A breakdown of how energy was used:

CPU	not supported %
System	not supported %
Mean node power	not supported W
Peak node power	0.00 W

Energy metrics are not available on this system.

Figure 4: ARM Performance Report from the Mandelbrot version 3 example with 4 MPI processes. 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.



arm PERFORMANCE REPORTS

Command:	aprun -n 16 ./mandelbrot
Resources:	1 node (64 physical, 256 logical cores per node)
Memory:	252 GiB per node
Tasks:	16 processes
Machine:	xcimom2
Start time:	Tue Jul 20 11:24:18 2021
Total time:	8 seconds
Full path:	.

Summary: mandelbrot is Compute-bound in this configuration

Compute	92.5%	Time spent running application code. High values are usually good. This is **very high**; check the CPU performance section for advice
MPI	7.5%	Time spent in MPI calls. High values are usually bad. This is **very low**; this code may benefit from a higher process count
I/O	0.0%	Time spent in filesystem I/O. High values are usually bad. This is **negligible**; there's no need to investigate I/O performance

This application run was Compute-bound. A breakdown of this time and advice for investigating further is in the CPU 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	0.64
L2D cache miss	37.1%
Stalled backend cycles	28.2%
Stalled frontend cycles	1.4%

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

MPI

A breakdown of the 7.5% MPI time:

Time in collective calls	6.5%
Time in point-to-point calls	93.5%
Effective process collective rate	0.00 bytes/s
Effective process point-to-point rate	99.7 MB/s

I/O

A breakdown of the 0.0% I/O time:

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

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

Threads

A breakdown of how multiple threads were used:

Computation	0.0%
Synchronization	0.0%
Physical core utilization	25.0%
System load	25.0%

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	376 MiB
Peak process memory usage	389 MiB
Peak node memory usage	3.0%

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 %
System	not supported %
Mean node power	not supported W
Peak node power	0.00 W

Energy metrics are not available on this system.

Figure 5: ARM Performance Report from the Mandelbrot version 3 example with 16 MPI processes. The MPI fraction is now reduced as the manager process is only one out of 16 processes.

3.4 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.

3.5 Profilers summary

Table 3 summarises the capabilities of the profiling tools reviewed in this section.

Profiler	Max MPI procs	C/C++	Fortran	Python
ARM Forge	??	✓	✓	✓
Intel Parallel Studio	??	✓	✓	??
Cray PAT	Unlimited	✓	✓	

Table 3: Summary of profiling tools

4 Builds

Plan:

1. Working builds on main target platforms
2. Optimised builds on main target platforms: use optimised maths libraries but beware threading

Platform	Build status	Profilers
Archer2	??	
Isambard XCI	working	`perf-report`
Isambard A64FX	??	
Isca	??	
Server	??	

Table 4: Status of Firedrake builds and profiling for the target platforms

- Isambard build of Firedrake: what's happening with BLAS/Lapack etc? Scalapack gets built as part of the PETSc build but is also available from `libsci`

5 Conclusions and recommendations