Principles for automated and reproducible benchmarking

Koskela, Tuomas ORCID logoORCID: https://orcid.org/0000-0002-5813-6539, Christidi, Ilektra ORCID logoORCID: https://orcid.org/0000-0002-5045-7987, Giordano, Mosè ORCID logoORCID: https://orcid.org/0000-0002-7218-2873, Dubrovska, Emily ORCID logoORCID: https://orcid.org/0009-0003-8066-5458, Quinn, Jamie ORCID logoORCID: https://orcid.org/0000-0002-0268-7032, Maynard, Christopher ORCID logoORCID: https://orcid.org/0000-0002-6253-9154, Case, Dave ORCID logoORCID: https://orcid.org/0009-0001-3735-5687, Olgu, Kaan ORCID logoORCID: https://orcid.org/0000-0003-0351-2055 and Deakin, Tom ORCID logoORCID: https://orcid.org/0000-0002-6439-4171
(2023) Principles for automated and reproducible benchmarking. In: SC-W 2023: Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, 12-17 Nov 2023, Denver, Colorado, pp. 609-618. doi: https://doi.org/10.1145/3624062.3624133 (ISBN: 9798400707858) Available at https://centaur.reading.ac.uk/114121/

Principles for Automated and Reproducible Benchmarking

Tuomas Koskela
Ilektra Christidi
Mosè Giordano
Emily Dubrovska
University College, London
Advanced Research Computing
London, UK

Jamie Quinn
University College, London
Advanced Research Computing
London, UK

Christopher Maynard
Met Office
Exeter, UK
University of Reading
Department of Computer Science
Reading, UK

Dave Case
National Centre for Atmospheric Science
University of Reading
Department of Meteorology
Reading, UK

Kaan Olgu
University of Bristol
School of Computer Science
Bristol, UK

Tom Deakin
tom.deakin@bristol.ac.uk
University of Bristol
School of Computer Science
Bristol, UK

ABSTRACT

The diversity in processor technology used by High Performance Computing (HPC) facilities is growing, and so applications must be written in such a way that they can attain high levels of performance across a range of different CPUs, GPUs, and other accelerators. Measuring application performance across this wide range of platforms becomes crucial, but there are significant challenges to do this rigorously, in a time efficient way, whilst assuring results are scientifically meaningful, reproducible, and actionable. This paper presents a methodology for measuring and analysing the performance portability of a parallel application and shares a software framework which combines and extends adopted technologies to provide a usable benchmarking tool. We demonstrate the flexibility and effectiveness of the methodology and benchmarking framework by showcasing a variety of benchmarking case studies which utilise a stable of supercomputing resources at a national scale.

ACM Reference Format:
Tuomas Koskela, Ilektra Christidi, Mosè Giordano, Emily Dubrovska, Jamie Quinn, Christopher Maynard, Dave Case, Kaan Olgu, and Tom Deakin. 2023. Principles for Automated and Reproducible Benchmarking. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3624062.3624133

1 INTRODUCTION

Hoefler and Belli [17] eloquently summarised sensible ways to present performance results in response to the satire presented by Bailey [3]. Bailey presents a humorous vignette on sharing performance results which are dubious, such as comparing to unoptimised serial code, not comparing like-for-like, and "if all else fails, show pretty pictures and animated videos" [3]. Hoefler and Belli provided advice on how to prepare and present performance results that we could interpret and trust [17]. They express the importance of documenting and controlling the environment in which the benchmark is run but do not elaborate on how this can be tangibly achieved in a practical, repeatable way.

As supercomputing systems are deploying different processor technologies to reach Exascale, the need to understand the performance of codes running on different systems, perhaps using different architectures, is becoming vital. This information is crucial when assessing the performance portability of an application [25]; or in other words, the ability of an application to make the most of all computational platforms available to the research group and the protection this offers through assurances of productive longevity in the investments made in the software.

Our work here addresses this challenge. This paper offers a practical and rigorous methodology for avoiding the pitfalls of poor benchmarking practice, providing a framework for conducting rigorous benchmarking for performance portability. We will show how our approach provides a robust and rigorous way to collect, and repeat, performance portability benchmarking studies efficiently. We build on widely adopted recent technologies and present a framework and a number of case studies highlighting how the methodology can be put into practice in a variety of normally complex and onerous benchmarking studies.

Previous studies by Deakin et al., such as [7], surveyed the performance portability of mini-apps across a range of different processors. The mini-apps themselves were written in different programming models, with the study providing a snapshot on the state of the compilers and runtimes on the systems needed to test all combinations. Collecting the performance results for that study is estimated to have taken around 18–24 months FTE (full-time equivalent), noting the study ran in a less elapsed time with the authors working in parallel and was therefore a highly labour intensive exercise. Whilst such time capsules are useful in evaluating the current state of the union of programming models on advanced architectures, and provide a historical record from which we can

SC-W 2023, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0785-8/23/11.
https://doi.org/10.1145/3624062.3624133

identify trends [8], it is clear that conducting sweeps of the performance portability of application implementations across several systems is hugely expensive. It is our ambition to instead invest in deeper analysis of the codes, implementations, runtimes and compilers on the different systems, however the burden of benchmarking restricts the depth whilst maintaining breadth (noting that the "Bristol Definition" of performance portability seeks to consider an extreme range of platforms [8]). We will demonstrate in Section 3.1 how the benchmarking framework presented allows such performance portability surveys to be more easily obtained.

Benchmarking results from any high-performance system intrinsically have a temporal challenge in that they can only be reproduced whilst the system is still available. The survey by Plale et al. [28] gave insights that sharing the code is not enough to reproduce results, and in line with "Rule 9" from Hoefler and Belli [17], documenting the process helps others reproduce the results on similar systems, or adapt to newer ones sharing a technological lineage (i.e., the next revision of a similar technology). Whilst the documentation of a benchmarking process allows for archaeology, such approaches in our experience can often be developed in an ad hoc manner and may lose intrinsic knowledge about the state of system defaults. Attempts to capture this detail, such as that in the Supercomputing Reproducibility Initiative Author Kit[1], may capture too much detail around irrelevant aspects of the experimental setup. The focus of these approaches is on the documenting of what happened for later audit, and not *a priori* collecting the results with a view that they should be reproducible.

The methodology set out in Section 2, along with a workflow building on well-known tools, will provide a practical solution for benchmarking activity. In Section 3 we showcase the use of the framework to ask different research questions which require benchmarking to provide the evidence to answer. In doing this, we are able to provide appropriate configurations for a range of benchmarks and supercomputing systems and testbeds in the UK HPC provision (Tier-1, Tier-2, etc.) providing good coverage of system and architecture diversity.

In particular, we make the following contributions:

- Define a methodology and set of principles for reproducibly collecting benchmarking data across diverse systems.
- Build on and extend existing tools in the benchmarking pipeline into a cohesive workflow.
- Showcase the flexibility of the methodology through case studies seeking to benchmark different aspects of applications, systems, or numerical algorithms.
- Share an open-source collated suite of benchmarks and system configurations to enable easy reproduction of the results in the paper, along with a path to adapt them to new applications and systems.[2]

2 BENCHMARKING METHODOLOGY

Benchmarking superficially involves simply running a code on a system and reporting its runtime. We might extend this to survey a range of different processors to determine which system to buy, or test different software stacks to find the best optimizing compiler, or any number of experiments. At this point, the benchmarking activity becomes significantly more complex, and it is no longer straightforward to carefully collect and interpret these results.

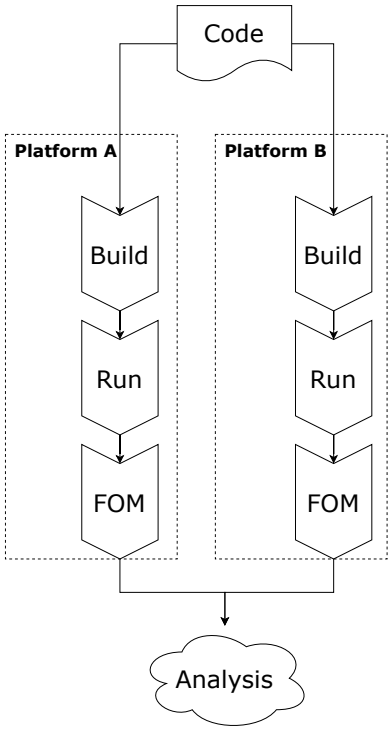

Figure 1: A typical workflow for benchmarking the performance portability of an application on different systems, adapted from a diagram from Pennycook et al. [26]

The illustration of a workflow is abstractly represented in Figure 1, based on a figure by Pennycook [26]. A benchmarking workflow takes a code, compiles and runs a chosen problem size on different platforms, and extracts a Figure of Merit (FOM), a comparable numeric value that measures the performance of the code on that platform. The collection of FOMs can then be analysed as a set. Under this model the problem size is part of the "Code" in so far as that it is fixed across all platforms. The platforms are abstract, following the definition of a Platform from Pennycook et al. [27], whereby it represents the hardware, software, compilers, and runtimes needed to run the benchmark code (or application). Here lies the challenge, in particular for following Hoefler and Belli's advice [17] on keeping as many of the varying factors constant as possible when comparing one platform to another.

This section explores in detail the best practices for benchmarking across diverse systems and benchmark applications, and defines a methodology through a series of guiding Principles for reproducible benchmarking. We will also show how we have built a Benchmarking Framework using Spack [15] and ReFrame [18] that follows that methodology.

[1]See the `collect_environment.sh` script available at https://github.com/SC-Tech-Program/Author-Kit.

[2]The Benchmarking Framework is available under the Apache 2.0 license at https://github.com/ukri-excalibur/excalibur-tests.

2.1 Choosing and Writing Benchmarks

Although any scientific application can be benchmarked, typically by just recording the runtime of the application (or of each component part), it is often more instructive to be able to reason about its performance by understanding whether it is making good use of the underlying hardware. This is important as we move to HPC ecosystems where supercomputing provision (at a national scale) is diverse, and applications should be able to use any resources; for example, this is a focus of the Exascale Computing ALgorithms & Infrastructures Benefiting UK Research (ExCALIBUR) project in the UK, which aims to "future proof the UK against the fast-moving changes in supercomputer designs" and is funding improvements to software and creating diverse resources through testbeds [13].

A benchmark then needs a way to report the performance on different platforms (architectures, software stacks, or both). The United States Exascale Computing Project (ECP) Exascale Proxy Applications Project recommends proxy apps have a Figure of Merit (FOM) that enables this comparison [12]. Initially this can be useful, however collecting such a benchmark result does not provide a clear understanding of how good the performance is on a particular platform. The focus is on comparing systems and not primarily whether the application is portable, and performance portable, across different systems with perhaps different architectures.

A better approach is one described by Pennycook et al. as *efficiency* [25, 27]. When efficiency of the application running on a specified platform (as a percentage of theoretical peak performance, or relative to some unportable highly-optimized code, or other such ratios) is used, the measure of performance becomes far more directly comparable over changes in the platform.

Measuring efficiency gives more relevant information. For example, the HPCG benchmark was designed to capture the performance of memory subsystems, both to better capture how scientific applications utilise large supercomputers, but also to motivate improvements to the relevant parts of computer architecture [10]. Considering runtime alone would not yield such nuanced analyses.

We note that this is not in conflict with Bailey's humorous "Ninth Way" of fooling readers by presenting processor utilization [3]. An efficiency metric does not capture such "busy work", rather by definition would penalise wasting the resources. A common example of this is that memory bandwidth benchmarks such as STREAM [22] do not count the data moved for a Read-for-Ownership for write instructions on some micro-architectures, and therefore performance models might understate the amount of data moved in reality, but capture the rate at which data is used *usefully*. Presenting a hardware measurement of the data moved therefore also captures the wasteful movement, just as reporting this for codes with unoptimised data layouts, due to inefficient use of cache lines for instance, can indicate a more favourable picture of the utilisation of precious hardware resources than is otherwise the case—a problem that can occur with tools that calculate Arithmetic Intensity under the Roofline model through application measurement [29].

For our purposes here, a key motivation is the exploration of applications across diverse systems. Which hardware is faster than another for a given application is not the principal focus, although of course this data can be gleaned from benchmarking studies and is more relevant when considering procurement of a system for a collection of applications. In line with the "Bristol Definition" of performance portability which we mentioned in Section 1, the analysis of performance results on different platforms (see again Figure 1) should be formulated as efficiency (effective use of the computational resources).

> **Principle 1**—A benchmark application should have a Figure of Merit which can measure (directly or indirectly) the *efficiency* of the application on a given platform.

2.2 Building Benchmarks

Building applications on one system can be a significant challenge, let alone building it many times to test different compilers, numerical libraries, etc., and further repeat all this on a different system or architecture. In our experience [7], this is a challenge even on benchmark codes which don't have many dependencies beyond a single parallel programming model, and is thus extremely hard for more complex applications.

We have previously attempted to encode all the information to build a particular code on a given platform using Bash scripts.[3] This attempt highlights the complex integration between the benchmark and the system on which it runs. To address this, we developed more rigorous CMake scripts for our benchmarks that help reduce the platform specialisation needed for the parallel programming model runtime libraries.[4]

Details aside, the fundamental principle is to encode information in the build system of the benchmark so that it can easily be built the correct way on each platform under the assumption that the dependencies are made available. This ensures that the benchmark is consistently built using the authors's knowledge of the code, reducing the likelihood of it being built incorrectly by others (curating the "Wisdom of the Crowd" as described by Gamblin and Katz [14]). Importantly, it also captures the nuances of building the benchmarks with different libraries, ensuring the knowledge required to build the codes is shared.

> **Principle 2**—Teach the build system how to build the benchmark using the best known parameters on each platform.

With such knowledge encoded in the build system, it then follows that this should become the only way the benchmark is built, and the build procedure is exercised regularly. While we see scientific software increasingly adopt Continuous Integration (CI) to test performance through efforts such as the Extreme-Scale Scientific Software Stack (E4S) project under ECP [11], we should adopt a similar methodology for performance benchmarking. As such, the benchmark binary should be rebuilt as regularly as possible, ideally every time it is run, in order to uphold **Principle 2**. Insofar as this is not the case, the binary is no longer reproducible, and performance results could be curated using forgotten knowledge: magic compiler options, environment variables, locally installed packages, secrets in a user's `.bashrc`, or other unhygienic software practices. In our view, it becomes impossible for someone else to reproduce our work if we ourselves do not reproduce it.

[3] See https://github.com/UoB-HPC/performance-portability associated with studies including [7, 8].
[4] See the build system for BabelStream for example [9]: https://github.com/UoB-HPC/BabelStream.

Principle 3—Rebuild the benchmark every time it runs to guarantee the steps to reproduce the binary are known.

We use Spack to drive the builds of our benchmarking applications. Spack is a popular package manager for HPC programs [15] which can be installed and updated in user-space if not already available in the system. It has a built-in dependency solver and it allows users to build software with multiple variants, allowing benchmarking experiments such as testing different compilers, etc., to be driven in a productive way. Spack uses the existing build systems (e.g. CMake, Autotools) of applications, rather than solving the problem of finding and installing the compilers and libraries the application needs; Spack is therefore agnostic of the language and build system of the original application. Furthermore, Spack supports virtual environments, which can be used to ensure reproducibility of builds. Spack's concretization mechanism records these steps so that they can be inspected later ("archaeological reproducibility"). All these features are instrumental for reliably building and comparing performance of applications across different platforms (using the definition of "platform" from Figure 1).

Spack comes with a large collection of recipes to build thousands of packages, one file for each package. Spack centralises the knowledge for building scientific software, while being flexible and enabling users to build multiple versions of a specific package, and/or with different options. However, it is also possible to create custom repositories of recipes for packages not included in Spack.

We create a Spack environment detailing the compilers and relevant packages available in all the systems we run benchmarks on, to reuse as many existing packages as possible, while allowing users to compile different versions if needed. We make these system-level Spack configurations available as part of the Benchmarking Framework. If the benchmarks are run on a system not yet supported by our framework, a basic Spack environment will be automatically created, but no system packages will be added.

In undertaking this study, we contributed to the centralised Spack repository recipes for new benchmark applications, and improved some existing ones, so as to share best practices. At the same time we keep a local repository of recipes for building applications not generally relevant for upstream Spack.

Principle 4—Capture all steps taken to build the benchmark on a given platform so it can be reproduced by anyone else using the system default environment.

Spack makes upholding **Principle 4** easy, with shareable configuration files capturing nuance on different systems. In combination with a robust build system under **Principle 2**, always building benchmarks with Spack helps ensure the benchmarks are built in a systematic way and that they can be built by someone else on the same system in the default environment.

2.3 Running Benchmarks

When running a benchmark on a system, there are a number of properties that need to be defined:

(1) the runtime configuration of the benchmark, such as problem size, decomposition scheme, etc.
(2) system job scheduler properties to distribute processes on the system, including SLURM/PBS parameters, MPI distribution and affinity, etc.
(3) changes to the default software environment to provide the necessary run-time libraries, access to drivers, etc.

Some of this information is intrinsic to the benchmark itself, such as the problem size and input data, and needs to be kept fixed across different platforms. The remainder is system-specific, and may be specialised to the platform, either to provide an equivalent resource on a different system, a different set of runtime libraries (for example, where you want to test various numerical libraries such as in the study by McIntosh-Smith et al. [23]), or to provide an entirely new architecture.

Principle 5—Capture all steps to run the built benchmark so it can be run by anyone on the same system using the default environment.

There is a clear similarity with **Principle 4** and **Principle 5**; the benchmark should be able to run on the same system independent of the user, just as the benchmark should be build-able under the same. Just as with building, this means the procedure for running the benchmark with the correct input files and parameters needs to be scripted. Further, the system specific information, such as job scheduler parameters, needs to be included in a way that can be easily changed or adapted separately to the information needed to run the benchmark (such as problem size, etc.), which is independent of the system specifics.

ReFrame is a Python-based framework for developing system regression tests, tailored to HPC facilities [18]. ReFrame helps control the benchmark execution by providing ways to adapt regression tests to new systems without changing the tests themselves through separating the environment and building of the test from the launching of the test.

One of the main features of ReFrame is that it separates the description of the benchmarks from the system-specific details for compiling and running it. A benchmark is defined by implementing a Python class that specifies how to build the software, which executable to run, the inputs and the parallel execution layout. System-specific details are recorded in a configuration file which includes, among other things, the job scheduler used, the MPI launcher, the topology of the nodes, etc. Each system can have multiple partitions, to accommodate cases where sets of nodes differ by one or more of the above properties. Note that the first abstraction conflates some system-specific information with the description of the benchmark application, however ReFrame provides abstractions to alleviate this. The benchmark definitions can access some information about the system. This means that, for example, it is possible to automatically run a benchmark on a fixed fraction of the cores available on a node. Hardcoding this number in the benchmark script instead would make the benchmark unportable. This allows the development of portable benchmarks that can be written on one system and subsequently run on any other system, as long as the benchmark can be compiled and the required computing resources are available, and for all the existing compatible benchmarks to be run on a newly added system.

ReFrame has built-in support for multiple build systems, including Spack. We extended this functionality with a ReFrame class to streamline the integration with the Spack environments provided by our framework: when the user selects the system where the benchmarks are being run, this class automatically identifies the

corresponding Spack environment and copies the relevant files to the stage directory where the benchmarks will run. This addresses challenge (3) in the list presented at the beginning of this section, bringing up the necessary system specific environment in a well defined and transparent manner to all benchmarks incorporated into the framework.

2.4 Interpreting Benchmarks

The output of benchmarks is not usually in a format readily available for the user to interpret. When a large collection of benchmarking results are generated, they need to be processed to extract the Figures of Merit, perhaps used to calculate an efficiency (such as dividing by a theoretical peak value based on the underlying architecture), and collected together in a single place in order to begin the analysis step of Figure 1. The curation of disparate data sets can result in unreproducible processing, or even mistakes, in the final presentation of data.

ReFrame helps significantly here by providing mechanisms for extracting the Figures of Merit for application outputs. When defining a benchmark in ReFrame, it can automatically collect a dictionary of Figures of Merit by parsing the output with user-provided regular expressions. A similar mechanism is used to check that the benchmark ran correctly (produced valid output). Benchmark output data is appended to a performance log (also known as a "perflog") associated with the benchmark on each system, and these logs can be collated directly and post-processed to extract information for graphing without manual manipulation or extraction.

Principle 6—Assimilate and post-process the data in a programmable manner so as to make extraction and presentation of Figures of Merit transparent and error-free.

We have developed a set of post-processing scripts to be included with the benchmarking framework which allow the user to visualise the results, reading data directly from the perflog output generated by ReFrame. The components of these scripts can also be used as a library, and imported into user-made scripts to provide natural extensions for more complex graphing and analysis.

The post-processing scripts parse the ReFrame output into a Pandas DataFrame [24]. This representation can efficiently handle large volumes of data, and Pandas has many useful inbuilt functions that simplify data modification. If more than one perflog is used for plotting, DataFrames from individual perflogs are concatenated together into one DataFrame—this feature is crucial for cross-platform data assimilation in a predictable manner where perflogs are generated on isolated systems.

The post-processing scripts also provide a unified way to filter the perflog and pass selected data to sample plotting scripts, all controlled via a YAML configuration file. We have a proof-of-concept way to visualise the plots as a bar chart, developed using Bokeh [4], that supports multiple data series. There is also ongoing work to provide simplified configurations that can be used to produce scaling and time-series regression plots. Including post-processing capabilities in the methodology aids the reproducibility of benchmarking results. In particular, the steps to assimilate, filter and present graphical plots of the benchmarking data (the ultimate aim of the workflow we showed in Figure 1) becomes automated and reproducible and provides a step forward for a fully integrated performance portability CI pipeline.

3 BENCHMARKING IN PRACTICE

The act of benchmarking can take many forms. People often put together benchmark suites in order to test various aspects of systems at key inflection points in supercomputing technology, for example: the HPC Challenge Benchmark Suite for Petascale-sized systems [21]; Mantevo, to motivate application-hardware co-design [6]; SPEChpc (and previous suites like SPECaccel) for ever growing heterogeneous systems [5]. In each case, benchmark codes were created and curated to provide informative measurements on current and new systems. Whilst benchmarking as a process was crucial to these studies, it was the data they produced and the timely impact they had on future developments of software and hardware that was of most value.

As such, the act of conducting a benchmarking study using an automated framework needs that tool to be flexible to a variety of different research questions, whilst supporting a shared underlying methodology such as the one we have outlined in this study. Here, we share some case-study style benchmarking activities to demonstrate how the principles we outline are followed by using the benchmarking framework. These indicative vignettes capture some common benchmarking patterns, and by providing the tools and configurations, we enable other communities to do the same.

3.1 Single-node memory bandwidth

BabelStream is a widely-used benchmarking tool designed to assess modern computer systems' memory bandwidth and performance [9]. Its primary purpose is to measure the data transfer rate between different levels of a memory hierarchy by measuring the sustained rate of data transfers to and from (main) memory (i.e., higher is better). Therefore, it provides a benchmark for the best case scenario for expected performance of an application bound by the movement of data. The output metric Triad, expressed in gigabytes per second (GB/s), is a fundamental figure that represents the memory bandwidth achieved during the benchmark. Further, BabelStream is written in many different parallel programming models and abstractions, and as such, can be used to explore **how performance portable are different programming models across a wide range of CPUs and GPUs?**

Figure 2 demonstrates the ratio of measured bandwidth values from the Triad kernel to the theoretical peak memory bandwidth values obtained from Table 1 for a number of CPUs and GPUs. The rows name the programming models, followed by signs that signifiy important choices required by that model: the "+" sign indicates any backend used where the programming model is an abstraction over another (such as Kokkos over OpenMP); the "%" signals the compiler name, and the "@" sign is used for the compiler version.

For a fair comparison between different architectures, the array size should be set such that it forces the data to go beyond the L3 cache size and be read from the main memory. The array size from the run command is set to be 2^{29} to have a processed data size larger than the cache sizes. For instance, the AMD Milan 7763 CPU used in the University of Paderborn system has a 256 MB per socket L3 cache size, equating to 512 MB with two sockets. The run command

Table 1: Information about Processors Used for BabelStream Benchmarks

Vendor	Processor	Cores/ Compute Units	Peak Memory Bandwidth (GB/s)
Intel	Cascade Lake	2x20	$2 \times 140.784 = 282$
Marvell	ThunderX2	2x32	288
AMD	Milan	2x64	2×204.8
NVIDIA	V100	80	900

with 2^{29} array size generates an array size of 4295.0 MB (=4.3 GB) and a total size of 12884.9 MB (=12.9 GB). For the rest, the array size is kept at 2^{25} since the 4 GB data is unnecessary to process with, for example, the Intel Cascade Lake CPU, whose cache size is 27.5 MB. The GCC compiler version used for "Isambard-MACS:Volta" is 9.2.0 since the build system has conflicts with newer versions; for the rest of the environments it is GCC 10.3.0.

Some tests do not work on some environments, either through incompatibilities (CUDA on CPUs,Intel-TBB on Thunder) or error messages that need to be addressed in the future: these tests are highlighted with a "*" sign with white boxes in the figure. The NVIDIA Volta GPU is close to the peak maximum bandwidth available when executing benchmarks with OpenCL and CUDA. OpenMP works on all devices and the performance results for OpenMP with GCC compiler show us that better utilisation is achieved with Intel and AMD CPUs.

The throughput of ReFrame execution matches with earlier research [20]. There is a disparity between throughput results of std-data & std-indices and std-ranges. This was an expected behaviour since the multicore version of std-ranges is a work in progress, and it only executes in a single thread. Also, some systems do not support using Intel TBB (Thread Building Blocks) for configuring multicore execution, which results in performance degradation. This behaviour is evident between paderborn-milan and isambard-macs:cascadelake TBB execution results, and also with the std-data and std-indices execution performance differences between isambard-macs and isambard-xci. We have not observed any specific degredation in runtime performance between building BabelStream via Spack, which invokes the normal CMake-based build system, from invoking the CMake manually.

It is essential to emphasize the positive contributions of ReFrame. One significant advantage is the marked productivity improvement in running the tests. In contrast to the previous work [20] that employed the CMake build system directly and required an extensive multi-month endeavour to accumulate comprehensive test run data, ReFrame streamlines the process significantly and reduces it to around a day of work (FTE). This remarkable efficiency in data collection highlights the potential of ReFrame to boost research efforts and enhance productivity substantially. Secondly, ReFrame allows users to configure their test runs with simple command line tools, and thus easily test different compilers on the same system (creating a new platform under our definition). Further, once a system is added to the configuration in the Benchmarking Framework, it can be shared with others and new benchmarks in the suite added without any alterations.

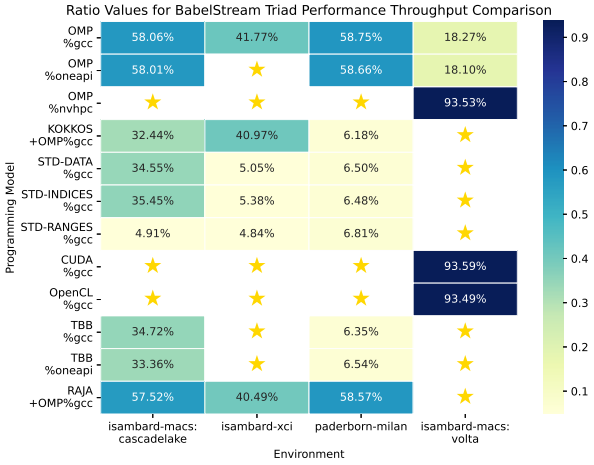

Figure 2: Figure of Merit for Different Vendor CPU & GPU results with the GCC v9.2.0 for GPU tests and GCC v12.1.0 compiler and oneAPI 2023.1.0 . The array size for Milan is 2^{29} and 2^{25} for the others.

3.2 Comparing variations/extensions from standard benchmarks

In science applications there is another dimension or axis to the space of performance and portability, that of the algorithm. Indeed, the implementation and algorithm are not independent. In this section, the framework is used to measure performance of both aspects to answer the question: **how does the implementation and algorithm effect the performance on different architectures?**

The HPCG benchmark[5] is a sparse linear algebra benchmark [10]. For many scientific applications this can be a better representation of application performance than other established benchmarks. Hardware optimised versions are available for several processor architectures, which allows for effective comparison of different processors. However, the relatively simple mathematical problem (Poisson's Equation in three dimensions) with a Finite-Difference, 27-point stencil and the Compressed Sparse Row (CSR) representation of the matrix results in specific memory access patterns which limit how representative the benchmark can be for other scientific problems. The equation, dimensionality, discretisation method (Spectral, Finite Element, Volume or Difference) and order, as well as choice matrix storage, will all influence the memory access patterns which are crucial to the performance of the HPCG benchmark.

Optimisations for specific processor architectures are provided by some vendors. This is useful in understanding the performance on these architectures and the HPCG benchmark provides a set of rules for how to report these. However, comparing a different mathematical problem to the standard one and vendor optimised standard problem can assist application developers in understanding how their application could perform on that platform by providing a richer set of information. The framework can be used to explore the algorithmic and implementation space in a rigorous way. Moreover,

[5]https://www.hpcg-benchmark.org

Table 2: Results for different HPCG variants on different architectures in GFlop/s. All results are for MPI only on a single node. The Intel Xeon Cascade Lake results were for 20-core, dual socket (40 MPI ranks) on Isambard. The AMD EPYC Rome results were for 64-core, dual socket (128 MPI ranks) on ARCHER2.

HPCG Variant	Intel Cascade Lake	AMD Rome
Original (CSR)	24.0	39.2
Intel-avx2 (CSR)	39.0	N/A
Matrix-free	51.0	124.2
LFRic	18.5	56.0

it mitigates against one of Bailey's twelve tricks—"secretly optimised code on one platform" [3]—which may inadvertently arise when using vendor optimised binaries.

Matrix storage methods such as CSR used by HPCG allow for any matrix arising from the equation to be solved. This generality is useful and this approach is often used by linear algebra libraries to support solution of generic systems. However, due to the indirect memory addressing in the method, this may not be fast enough for some applications. Typically, these applications use the matrix-free method where the matrix is not assembled, but the operation of the matrix on a vector is explicitly encoded for given values of the matrix. Each application requires its own implementation of the matrix-vector operation, but this can be much more memory and cache efficient than a general method.

A matrix-free implementation of the 27-point stencil of the original problem and a symmetrised version of the Helmholtz operator from the Met Office weather and climate model, known as LFRic [2], have been implemented. These can then be compared to the standard, CSR, version. Shown in Table 2 are the results of running the HPCG benchmark in the framework on various HPC machines for the different variations. The result labelled "Intel-avx2" is the best performing of the three binaries provided by Intel as part of the Intel oneAPI MKL library. The technical details of the LFRic Helmholtz operator are relevant for the application developer but not for the purposes of this paper, only that an application specific representation of the matrix can be employed in the framework and compared to the other implementations. Even with the limited set of comparisons shown here, the size of changes from processor, algorithm and implementation allow the application developer to estimate what expected maximum performance could be achieved with their application and an optimised implementation of the relevant algorithm.

Comparing the ratios of different aspects allows the notion of efficiency and thus satisfies **Principle 2** defined in Section 2. For example, comparing the FOM for the vendor optimised implementation to the algorithmic variation with the standard implementation:

$$E = \frac{\text{VAR}}{\text{ORIG}} \tag{1}$$

gives $E_I = 1.625$ for the (Intel) implementation optimisation but the algorithmic variation of *CSR* to *matrix-free*, $E_A = 2.125$ is greater. For the AMD processor the algorithmic gain is even larger, $E_A = 3.168$. This allows some quantification for how much more efficient algorithmic optimisation is, than optimising the implementation; echoing observations in the 2010 SCALES report [19]. The initial results presented here are limited in scope but demonstrate the principle. Using this approach, a comprehensive study across many processor architectures, with implementation or vendor optimisations and algorithmic choices, could be performed. This collated information would allow a comprehensive map of the algorithm/implementation/processor architecture landscape to be determined.

Table 3: Concretized build dependencies of the HPGMG-FV benchmark using the hpgmg%gcc spec

System	gcc	Python	MPI library
ARCHER2	11.2.0	3.10.12	cray-mpich 8.1.23
COSMA8	11.1.0	2.7.15	mvapich 2.3.6
CSD3	11.2.0	3.8.2	openmpi 4.0.4
Isambard-macs	9.2.0	3.7.5	openmpi 4.0.3

3.3 Supercomputing provision survey

We have configured the framework on a selection of the UK HPC systems used by the scientific community. As a proof of point, we installed the framework on four of these systems: ARCHER2, CSD3, COSMA8 and Isambard's Multi-Architecture Comparison System, and thus the framework provides a rubric for others to build and run the benchmarks on any of those systems using a single workflow. With this case study, we **explore the performance portability of a benchmark across systems with roughly similar architectures and programming environments**, with a longer term view that this data is crucial for ensuring systems and applications are (or become) performance portable between similar systems.

We demonstrate running the HPGMG-FV benchmark [1] on the four systems. HPGMG-FV solves elliptic PDEs using the finite volume method (FV) and a Full Multigrid (FMG) algorithm, making it a useful proxy for scientific applications built on PDE solvers such as [16]. We specified the Spack specification `hpgmg%gcc` to use the gcc compiler, and used eight tasks, two tasks per node, eight CPUs per task in ReFrame; although each system has a different number of cores than this, we use this to demonstrate running the benchmark in a fixed configuration on all systems. The default FV variant of the HPGMG benchmark has two build dependencies, MPI and Python. We let Spack determine the versions of the dependencies, and report what was used in Table 3. We use the command line arguments `7 8` to set the box size and number of boxes per process in HPGMG-FV.[6]

This example is not meant to be a comprehensive comparison of the four systems, but rather a demonstration of the capabilities of the benchmarking framework for a benchmarking experiment such as this. We report the three Figures of Merit reported by HPGMG-FV as captured by ReFrame in Table 4.

The results are presented here to demonstrate that comparable results are straightforward to collect using the framework. We do not comment specifically on the performance of the benchmark on the different systems, however the results indicate that specifics of

[6]The inputs to HPGMG follow the example in https://bitbucket.org/hpgmg/hpgmg/src/master/README.md

Table 4: Figures of Merit of HPGMG-FV benchmark, where results are reported as a compute rate of 10^6 DOF/s

System	l_0	l_1	l_2
ARCHER2 (Rome)	95.36	83.43	62.18
COSMA8 (Rome)	81.67	72.96	75.09
CSD3 (Cascade Lake)	126.10	94.39	49.40
Isambard (Cascade Lake)	30.59	25.55	17.55

the platform can impact the performance of a benchmark significantly beyond changes in the underlying architecture. As shown here, the benchmark is not necessarily performance portable out of the box, even with consistent approaches to building and running the code (under **Principle 4** and **5**), and this is vital information for further developments to improve performance, portability, and performance portability. Indeed, these preliminary results should be used as exemplar motivation that cross-system performance regression testing is now a fundamental necessity of scientific software development, and with the framework as presented we provide an integrable approach which can form the basis of a CI pipeline to test the stable of supercomputers available to scientific communities on a regular basis.

4 CONCLUSION

Benchmarking in a rigorous and reproducible manner is a significant challenge. This is further exacerbated by the diversity in supercomputer architectures that an application needs to run on. This means testing that it will build, run, and run with good efficiency, across all systems of interest is a crucial part of scientific software development today.

Testing the performance of codes—benchmarking them—is therefore a significant undertaking of effort when multiple platforms are involved. Understanding and recording the changes in programming environments, compilers and other factors are vital to ensure that the performance results can be reproduced, and compared fairly between results of the same code prepared on other systems.

The methodology presented here shares some guiding principles for the art of benchmarking. By following these principles, the benchmarking results can be generated in a way that is reproducible both by the benchmark-er, and by others wishing to repeat or build on the initial experiments. The principles outline the best practices for undertaking a benchmarking activity, and we encourage future benchmarking studies to adopt these principles for ensuring the performance data presented doesn't "Fool the Masses" [3].

The benchmarking principles apply to a wide variety of benchmarking studies, as we shared with the case studies in Section 3. As with all performance results, these snapshots are presented as case studies for how following the methodology proposed can yield interesting datasets for further analysis and optimisation.

By automating the design and collection of benchmarking data in a framework, in our case built on Spack and ReFrame, the way is paved for making changes in performance as important as changes in answers for scientific applications. Going forward, we plan to enhance the framework with more analysis capability so that a sweep of performance data across diverse computer systems in the UK ecosystem can be run as part of a CI pipeline, and enable researchers to measure and track the performance portability of their applications over time. In addition, we are planning to add functionality to capture relevant parameters of the system state during the runtime of the benchmarks, such as network or filesystem usage levels or energy consumption.

ACKNOWLEDGMENTS

This work was supported by the Engineering and Physical Sciences Research Council [EP/X031829/1].

This work used the Isambard 2 UK National Tier-2 HPC Service (http://gw4.ac.uk/isambard/) operated by GW4 and the UK Met Office, and funded by EPSRC (EP/T022078/1).

This work used the ARCHER2 UK National Supercomputing Service (https://www.archer2.ac.uk).

This work was performed using resources provided by the Cambridge Service for Data Driven Discovery (CSD3) operated by the University of Cambridge Research Computing Service (www.csd3.cam.ac.uk), provided by Dell EMC and Intel using Tier-2 funding from the Engineering and Physical Sciences Research Council (capital grant EP/T022159/1), and DiRAC funding from the Science and Technology Facilities Council (www.dirac.ac.uk).

This work used the DiRAC@Durham facility managed by the Institute for Computational Cosmology on behalf of the STFC DiRAC HPC Facility (www.dirac.ac.uk). The equipment was funded by BEIS capital funding via STFC capital grants ST/P002293/1, ST/R002371/1 and ST/S002502/1, Durham University and STFC operations grant ST/R000832/1. DiRAC is part of the National e-Infrastructure.

The authors gratefully acknowledge the computing time provided to them on the high-performance computers Noctua2 at the NHR Center PC2. These are funded by the Federal Ministry of Education and Research and the state governments participating on the basis of the resolutions of the GWK for the national high-performance computing at universities (www.nhr-verein.de/unsere-partner).

REFERENCES

[1] Mark F. Adams, Jed Brown, John Shalf, Brian Van Straalen, Erich Strohmaier, and Samuel Williams. 2014. *HPGMG 1.0: A Benchmark for Ranking High Performance Computing Systems.* Technical Report. LBNL 6630E.

[2] S.V. Adams, R.W. Ford, M. Hambley, J.M. Hobson, I. Kavčič, C.M. Maynard, T. Melvin, E.H. Müller, S. Mullerworth, A.R. Porter, M. Rezny, B.J. Shipway, and R. Wong. 2019. LFRic: Meeting the challenges of scalability and performance portability in Weather and Climate models. *J. Parallel and Distrib. Comput.* 132 (2019), 383–396. https://doi.org/10.1016/j.jpdc.2019.02.007

[3] David H. Bailey. 1991. Twelve ways to fool the masses when giving performance results on parallel computers. *Supercomputing Review* 4, 8 (August 1991), 54–55.

[4] Bokeh Development Team. 2018. *Bokeh: Python library for interactive visualization.* Bokeh Contributors. https://bokeh.pydata.org/en/latest/

[5] Holger Brunst, Sunita Chandrasekaran, Florina M. Ciorba, Nick Hagerty, Robert Henschel, Guido Juckeland, Junjie Li, Verónica G. Melesse Vergara, Sandra Wienke, and Miguel Zavala. 2022. First Experiences in Performance Benchmarking with the New SPEChpc 2021 Suites. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid).* IEEE, Taormina, Italy, 675–684. https://doi.org/10.1109/CCGrid54584.2022.00077

[6] Paul Stewart Crozier, Heidi K Thornquist, Robert W Numrich, Alan B Williams, Harold Carter Edwards, Eric Richard Keiter, Mahesh Rajan, James M Willenbring, Douglas W Doerfler, and Michael Allen Heroux. 2009. *Improving performance via mini-applications.* Technical Report. Sandia National Laboratories (SNL), Albuquerque, NM, and Livermore, CA ….

[7] Tom Deakin, Simon McIntosh-Smith, James Price, Andrei Poenaru, Patrick Atkinson, Codrin Popa, and Justin Salmon. 2019. Performance Portability across Diverse Computer Architectures. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC).* IEEE, Denver, CO, 1–13. https://doi.org/10.1109/P3HPC49587.2019.00006

[8] Tom Deakin, Andrei Poenaru, Tom Lin, and Simon McIntosh-Smith. 2020. Tracking Performance Portability on the Yellow Brick Road to Exascale. In *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, GA, USA, 1–13. https://doi.org/10.1109/P3HPC51967.2020.00006

[9] Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. 2018. Evaluating attainable memory bandwidth of parallel programming models via BabelStream. *International Journal of Computational Science and Engineering* 17, 3 (2018), 247–262. https://doi.org/10.1504/IJCSE.2018.095847 Special Issue on Novel Strategies for Programming Accelerators.

[10] Jack Dongarra, Michael Heroux, and Piotr Luszczek. 2015. High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems. *International Journal of High Performance Computing Applications* 30 (08 2015), 3–10. Issue 1. https://doi.org/10.1177/1094342015593158

[11] Exascale Computing Project. 2019. *Software Technology Update*. Technical Report. United States Department of Energy Office of Science and National Nuclear Security Administration.

[12] Exascale Proxy Applications Project. 2022. Proxy App Quality Standards and Best Practices. Available at https://proxyapps.exascaleproject.org/standards/ (2023/07/31).

[13] ExCALIBUR. 2022. About ExCALIBUR. Available at https://excalibur.ac.uk/about-excalibur/ (2023/07/31).

[14] Todd Gamblin and Daniel S. Katz. 2022. Overcoming Challenges to Continuous Integration in HPC. *Computing in Science & Engineering* 24, 6 (2022), 54–59. https://doi.org/10.1109/MCSE.2023.3263458

[15] Todd Gamblin, Matthew LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and Scott Futral. 2015. The Spack Package Manager: Bringing Order to HPC Software Chaos. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Association for Computing Machinery, Austin, Texas, USA, 1–12. https://doi.org/10.1145/2807591.2807623 LLNL-CONF-669890.

[16] David A. Ham, Paul H. J. Kelly, Lawrence Mitchell, Colin J. Cotter, Robert C. Kirby, Koki Sagiyama, Nacime Bouziani, Sophia Vorderwuelbecke, Thomas J. Gregory, Jack Betteridge, Daniel R. Shapero, Reuben W. Nixon-Hill, Connor J. Ward, Patrick E. Farrell, Pablo D. Brubeck, India Marsden, Thomas H. Gibson, Miklós Homolya, Tianjiao Sun, Andrew T. T. McRae, Fabio Luporini, Alastair Gregory, Michael Lange, Simon W. Funke, Florian Rathgeber, Gheorghe-Teodor Bercea, and Graham R. Markall. 2023. *Firedrake User Manual* (first edition ed.). Imperial College London and University of Oxford and Baylor University and University of Washington. https://doi.org/10.25561/104839

[17] Torsten Hoefler and Roberto Belli. 2015. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses When Reporting Performance Results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) *(SC '15)*. Association for Computing Machinery, New York, NY, USA, Article 73, 12 pages. https://doi.org/10.1145/2807591.2807644

[18] Vasileios Karakasis, Theofilos Manitaras, Victor Holanda Rusu, Rafael Sarmiento-Pérez, Christopher Bignamini, Matthias Kraushaar, Andreas Jocksch, Samuel Omlin, Guilherme Peretti-Pezzi, João P. S. C. Augusto, Brian Friesen, Yun He, Lisa Gerhardt, Brandon Cook, Zhi-Qiang You, Samuel Khuvis, and Karen Tomko. 2020. Enabling Continuous Testing of HPC Systems Using ReFrame. In *Tools and Techniques for High Performance Computing*, Guido Juckeland and Sunita Chandrasekaran (Eds.). Springer International Publishing, Cham, 49–68.

[19] David E. Keyes and Office of Science. 2004. *A Science-based Case for Large-scale Simulation Volume 2*. U.S. Department of Energy, Washington, DC, Chapter 10 Plasma Science: Taming a Star, 123–134.

[20] Wei-Chen Lin, Tom Deakin, and Simon McIntosh-Smith. 2022. Evaluating ISO C++ Parallel Algorithms on Heterogeneous HPC Systems. In *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, Dallas, TX, 36–47. https://doi.org/10.1109/PMBS56514.2022.00009

[21] Piotr Luszczek, Jack J. Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John McCalpin, David Bailey, and Daisuke Takahashi11. 2005. Introduction to the HPC Challenge Benchmark Suite. In *Supercomputing*. 12 pages.

[22] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (December 1995), 19–25.

[23] Simon McIntosh-Smith, James Price, Andrei Poenaru, and Tom Deakin. 2019. Benchmarking the first generation of production quality Arm-based supercomputers. *Concurrency and Computation: Practice and Experience* 32 (November 2019), 12 pages. Issue 20. https://doi.org/10.1002/cpe.5569 special issue.

[24] The pandas development team. 2020. *pandas-dev/pandas: Pandas*. pandas-dev. https://doi.org/10.5281/zenodo.3509134

[25] S.J. Pennycook, J.D. Sewall, and V.W. Lee. 2019. Implications of a metric for performance portability. *Future Generation Computer Systems* 92 (2019), 947–958. https://doi.org/10.1016/j.future.2017.08.007

[26] S. John Pennycook, Jason Sewall, Douglas Jacobsen, Tom Deakin, Yuliana Zamora, and Kin Long Kelvin Lee. 2023. *Performance, Portability and Productivity Analysis Library*. Intel. https://doi.org/10.5281/zenodo.7733678

[27] S. John Pennycook, Jason D. Sewall, Douglas W. Jacobsen, Tom Deakin, and Simon McIntosh-Smith. 2021. Navigating Performance, Portability, and Productivity. *Computing in Science & Engineering* 23, 5 (2021), 28–38. https://doi.org/10.1109/MCSE.2021.3097276

[28] Beth A. Plale, Tanu Malik, and Line C. Pouchard. 2021. Reproducibility Practice in High-Performance Computing: Community Survey Results. *Computing in Science & Engineering* 23, 5 (2021), 55–60. https://doi.org/10.1109/MCSE.2021.3096678

[29] Charlene Yang, Rahulkumar Gayatri, Thorsten Kurth, Protonu Basu, Zahra Ronaghi, Adedoyin Adetokunbo, Brian Friesen, Brandon Cook, Douglas Doerfler, Leonid Oliker, Jack Deslippe, and Samuel Williams. 2018. An Empirical Roofline Methodology for Quantitatively Assessing Performance Portability. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, Dallas, TX, 14–23. https://doi.org/10.1109/P3HPC.2018.00005

A ARTIFACT DESCRIPTION APPENDIX: PRINCIPLES FOR AUTOMATED AND REPRODUCIBLE BENCHMARKING

This study shared a set of Principles for reproducible benchmarking, and provides a methodology for collecting benchmarking data across different platforms, including supercomputing systems with different architectures. A Benchmarking Framework was developed following the Principles we outlined using Spack [15] and ReFrame [18].

We use the framework to collected performance results in three case-study experiments, and presented the preliminary results in Section 3. The framework by design allows the results to be reproduced on the different systems we used (ARCHER2, CSD3, COSMA8, Isambard, and Noctua2). Details of the processors used is shown in Table 5 The abstractions in the framework are such that it is straightforward to adapt the system configurations to a new system and reproduce the benchmarking there.

A.1 Reproducibility of Experiments

The Benchmarking Framework is available from https://github.com/ukri-excalibur/excalibur-tests. The installation workflow on each system is identical, once a compatible Python module has been loaded into the user's `$PATH` environment variable and a suitable working directory has been selected. The Framework depends on Spack, which should be installed prior to the installation of the Framework, noting that ReFrame is installed automatically with the Framework installation. The installation instructions can be found in the project README, and an example setup script can be found in the `excalibur-tests` repository.[7] The setup workflow can be summarized as:

(1) Create a Python virtual environment
(2) Clone the `excalibur-tests` repository
(3) Run `pip install excalibur-tests` (with `-editable`)
(4) Install the Spack package manager, and make it available in your `$PATH` environment variable
(5) Set the `RFM_CONFIG_FILES` environment variable so ReFrame can locate the benchmark configurations included in the Benchmarking Framework

After the setup is done, a benchmark is run by invoking ReFrame. Key points to be aware of when running ReFrame are:

- Accounting varies between HPC systems. On most systems, the account with the access to a time allocation has to be

[7] https://github.com/ukri-excalibur/excalibur-tests/blob/tk-portability-demo/demo/setup.sh

Table 5: Details of the processors used in this study

System	Processor	Core count
Isambard	Marvell ThunderX2 @ 2.5 GHz	32 cores/socket, dual-socket
Isambard MACS	Intel Xeon Gold 6230 @ 2.1 GHz (Cascade Lake)	20 cores/socket, dual-socket
Isambard MACS	NVIDIA Tesla V100 PCIe 16 GB	-
COSMA8	AMD EPYC 7H12 (Rome) @ 2.6 GHz	64 codes/socket, dual-socket
ARCHER2	AMD EPYC 7742 (Rome) @ 2.25 GHz	64 codes/socket, dual-socket
CSD3	Intel Xeon Platinum 8276 (Cascade Lake) @ 2.2 GHz	28 cores/socket, dual-socket
Noctua2	AMD EPYC 7763 (Milan) @ 2.45 GHz	64 cores/socket, dual-socket

passed to ReFrame with the `-J'-account'` command line option.
- The framework will attempt to identify the system but due to ambiguity of login node names, explicitly naming the system with the –system command line option helps avoid some errors.
- The Spack specification chosen should be specified on the command line with the `-S spack_spec` option to ensure the same toolchain is used on all systems. Most benchmarks will use a default specification. The Spack specification that was used by ReFrame will be recorded in the log files after the benchmark run has completed.
- The hardware requirements of the benchmark can be specified on the command line with the `–setvar` command line option. The size of a job is determined by the combination of a number of options: `num_tasks`, `num_cpus_per_task`, and `num_tasks_per_node`. Some of the applications, such as BabelStream, that are included as part of the Benchmarking Framework will use the maximum number of CPUs per task unless overridden on the invocation of ReFrame on the command line.

In the following sections we provide links to the ReFrame invocations used to collect the results presented in Section 3. In all cases, ReFrame will check the benchmarks have been executed correctly, and produce the performance in the performance log ("perflog") file. The results presented in the tables in the main paper were extracted from the ReFrame output, which is also saved in the perflog. All the benchmarks have a short runtime of at most a few minutes on the hardware we tested.

A.1.1 Running BabelStream with the Benchmarking Framework. The invocations of the Benchmarking Framework using ReFrame for the BabelStream experiment in Section 3.1 are available online.[8]

An example ReFrame command for running OpenMP using the GCC compiler on a Cascade Lake processor in the Isambard supercomputer is as follows:

```
reframe -c benchmarks/apps/babelstream -r --tag omp
↪ --system=isambard-macs:cascadelake -S
↪ build_locally=false -S
↪ spack_spec='babelstream%gcc@9.2.0 +omp'
```

Note: the std-data and std-indices parallel versions are in "option_for_vec" branch

[8]https://gist.github.com/tomdeakin/258867a79a363f2007e356c49f14f3b3

A.1.2 Running HPCG with the Benchmarking Framework. The invocations of the Benchmarking Framework using ReFrame for the HPCG experiment in Section 3.2 are available online.[9]

An example ReFrame command for running our modified HPCG benchmark on a Cascade Lake processor in the Isambard supercomputer is as follows:

```
reframe -c benchmarks/apps/hpcg -r -n HPCG_ -x
↪ HPCG_Intel --system isambard-macs:cascadelake
↪ --performance-report -S build_locally=false
```

A.1.3 Running HPGMG-FV with the Benchmarking Framework. The invocations of the Benchmarking Framework using ReFrame for the HPGMG-FV experiment in Section 3.3 are available online, and were invoked similarly to `./run.sh hpgmg gcc archer2`.[10]

An example ReFrame command for running the HPGMG-FV benchmark on the AMD Rome processor in the ARCHER2 supercomputer, as evaluated in the `run.sh` script, is as follows:

```
reframe -c excalibur-tests/benchmarks/apps/hpgmg -r
↪ -J'--qos=standard' --system archer2 -S
↪ spack_spec=hpgmg%gcc --setvar=num_cpus_per_task=8
↪ --setvar=num_tasks_per_node=2 --setvar=num_tasks=8
```

[9]https://github.com/ukri-excalibur/excalibur-tests/pull/196
[10]https://github.com/ukri-excalibur/excalibur-tests/blob/tk-portability-demo/demo/run.sh