Benchmark guidelines

SHAREing consortium

December 5, 2025

Abstract

Benchmarks form a basis for characterising software and/or hardware performance. In the context of performance critical software, benchmarks are a core component of assessing performance as they allow computational researchers to distill features of significant research software into smaller applications. Therefore, a well designed benchmark can be repeatedly executed to aggregate performance data on the software without having to execute the entire code base, as these runs may take days and large proportions of compute resources which becomes infeasible. In this document we define guidelines for computational researchers and developers to design benchmarks of their software. A key concern for this work is to ensure that benchmarks are truly indicative of typical research software workflows, because otherwise performance reports run the risk of being dismissed as not indicative of *true* use cases.

1 Introduction

There is no one standardised set of guidelines for creating a benchmark and this document does not aim to be this. However, these guidelines are to aid researchers in defining benchmarks of their research software — in some cases for the first time — for submitting to the developing performance assessment service of SHAREing.

Benchmarks can be broken largely into synthetic and application benchmarks. Synthetic benchmarks tend to be useful for measurements of lower level system performance, e.g., STREAM [1] is often used for measuring memory bandwidth. However, for SHAREing we are typically interested in benchmarks that behave as reduced versions of full research applications, e.g., miniBUDE [2] which is based on the molecular docking code, BUDE [3], or XSBench [4] and RSBench [5] which are neutron scattering benchmarks based on OpenMC. These original codebases are significant so it is not feasible for a performance analyst to run large, repeated experiments of the *full* software; we can instead use these benchmarks to capture performance features of the overarching software.

In this document we layout some advice for defining benchmarks to be submitted to the future SHAREing performance assessment service. We begin in Section 2 by laying out our key considerations for creating a benchmark and the associated materials that should be packaged with the benchmark. Then in Section 3 we touch on the compute resources that benchmarks may use and how to factor this into your benchmark design. Section 4 motivates why a benchmark should be reasonably configurable for a performance assessment service. Finally, Section 5 concludes some of the main points and defines a reduced checklist for ease of use when writing a benchmark.

2 Defining a benchmark

Reader's guide

If the reader has already designed their own benchmark, then this section can be skipped. However, the reader may find it useful to read this section to *review* their benchmark.

Our first recommendations are on designing a benchmark from a larger research software example, i.e., which components of the research software are necessary and what other materials are needed to support the benchmark other than the source code.

2.1 Writing a benchmark

We want benchmarks to represent the performance of larger research software, to capture features of the performance for analysis. As a starting point, developers may wish to know the functions that consume significant runtime, and the `gprof` profiler can be used on a serial run to give the percentage of total runtime for each function. The reason for this coarse level profile is not to force developers to begin their own analysis, it is to guide their benchmark design.

Multiple kernels: Similarly once the critical kernels are known, it is best to have more than just a single kernel or routine. Most research software contains more than just a single, performance critical routine. Hence, it is advised that a benchmark includes multiple kernels that are of interest. For example, if the code contains some computationally heavy matrix-vector equation solver, as well as other computationally intensive routines, it is advised to include some of these other kernels so that we are not just observing the performance of the matrix-vector implementation. Instead we are capturing performance that is more indicative of the 'true' application workflow.

Reduced algorithm: Once a subset of the compute kernels have been selected, we recommend building this into a small, viable algorithmic runtime, i.e., likely to include:

- Initialisation - configuration, setup of data structures, reading in data, etc.
- Iteration steps - repeated executions, typically of the computationally heavy routines
- Finalisation - cleaning up, perhaps some data postprocessing and writing out data

Typically the initialisation and finalisation times are small compared to the iteration steps. The iteration steps, by definition, will likely be iterated multiple times and so a subset of the typical number of iterations can be used. The key performance questions will likely be asked of these iteration steps. This outlines a 'minimal viable product' or 'algorithm' as we have selected a functioning algorithm that is a subset of the typical research software workflow. Likewise by a 'minimal viable product' or 'algorithm', we mean that the benchmark result can be validated. This is a similar process to how developers build tests, i.e., the benchmark may solve a reduced but known problem that is indicative of the broader software performance but is simpler and can be checked, e.g., against an analytic result.

Example I/O: A final consideration of performance is to capture indicative I/O behaviour. For example, if the software typically relies on repeated I/O, then it is advised to include examples of this I/O (such as writing simulation data to files during iteration steps) but with the ability for this to be switched off by the performance analyst.

Many algorithms and research software codebases have their own unique features, so we advise using the benchmark writing stage as time to reflect on which software features you wish to capture.

2.2 Reproducibility

Documentation is an integral component of reproducibility [6]. Likewise, much of performance analysis involves running experiments to investigate software performance issues, and reproducibility of these experiments is vital [7]. We recommend that a user of the assessment service take the opportunity of writing a benchmark — or revisiting a preexisting benchmark — to document all steps from beginning to end of a benchmark run, in keeping with **Rule 9** of Ref. [8]. This will likely include:

- libraries and dependencies, preferably with suggested versions used by the researcher or developer
- build instructions including use of compilers and the recommended build system, e.g., Make or CMake
- instructions for running an example test case.

Version control is ubiquitous now throughout software design but when compiling documentation, this can be an excellent time to check that this documentation is up-to-date and also relates well to the code repository (likely hosted on GitHub or GitLab). A clean, structured repository helps the analyst be able to easily clone the code and get setup running benchmarks quickly.

Beyond the necessary information for getting a new user from cloning the repository to running a test case, documentation can also be used to aid the performance analyst. For example, it can be useful to inform the analyst of typical job configurations that the researcher or developer uses. This could include, e.g., the distribution of MPI ranks across GPUs. This naturally raises questions about the resource and hardware usage of benchmarks.

3 Resource considerations

Akin to defining an indicative algorithm of the 'full' software, resource use should also be somewhat indicative. There are two main topics relating to resources that we consider here: (1) memory footprint; (2) runtime.

Memory footprint: Many modern compute nodes have of order 100GB of memory, so we recommend using a benchmark that can be configured to occupy $\sim 20\%$ of this. This may seem arbitrary, but we are looking to ensure that the memory of the benchmark does not just simply sit within the caches. Memory footprints that are 'too small' can also suffer from poor scaling as the work sharing across cores becomes too diffuse, i.e., there is not enough work to share. This could give inaccurate results from a strong scaling analysis for the *full* research software.

Runtime: Clearly when scaling up the memory footprint, runtime can become an issue. It is vital for the viability of running frequent analyses that the runtime should not be excessive, e.g., on the order of multiple hours. However, we also want to avoid benchmarks that are trivially short, e.g., less than one minute. For example, if a benchmark is running in parallel (and is maximally performant) then a runtime of approximately 10 minutes is good. However, this means that if we run this benchmark configuration in serial it may take significant time, e.g., over an hour. The serial run does not need to be run frequently so this is fine.

The key point of resource use is that we want a 'goldilocks zone' of large enough such that the benchmark is indicative of the typical research workloads, but small enough that the benchmarks can be executed repeatedly. If we configure our benchmark such that the memory footprint is considerable, for many simulation codes we can simply set the number of time steps to limit the total runtime. Therefore, configurablility of the benchmark is integral.

4 Benchmark configuration

A key piece of advice for our benchmarks is that they must be clearly configurable by a performance analyst including instructions for how the configuration changes or 'scales' the problem. For example, SHAREing's performance analysis methodology includes both strong and weak scaling analyses.

For weak scaling, the problem size must be scaled with the allocated resources and so the performance analyst must therefore have an idea of how to scale the problem size both from a job configuration standpoint — e.g, how to vary input flags — but also from an algorithmic perspective. Job configuration should be relatively simple if the benchmark is well documented. However, understand how the algorithm scales with input parameters can be subtle. For a three-dimensional, Cartesian grid, doubling the number of grid points in one direction scales the number of grid points by $2^3 = 8$. This is a relatively simple scaling, but some algorithms have less obvious scalings and this can be challenging. If the researchers or developers know the algorithmic scaling, it is incredible helpful to document this information for the performance analyst.

This highlights a final point: **we do not want there to be too high a barrier of entry for using the performance service**. Put another way, we recommend the user of the performance assessment to give us as much information as they can. The more information given during the application, the quicker the analyst can start digging into the details. For example, understanding algorithmic scaling is very useful but we appreciate that this can be complex. Hence, at the beginning on the assessment, the performance analyst can work closely with the user of the assessment service to build up any information that is missing upon submission.

5 Overview

We have here outlined some of the main features that the SHAREing consortium propose make a *useful* benchmark. The main ideas are that a benchmark captures behaviour that is indicative of the full software's performance, and is configurable for a performance analyst. For brevity we list some of the most important ideas from this document in considering benchmark design.

Checklist

In brief, we recommend a benchmark should:

1. capture several components of the mathematical or algorithmic features of the software
2. be representative of a typical job run, e.g., including realistic I/O calls
3. be configurable by the performance analyst, i.e., the example job should be easily scalable and should allow for features such as I/O to be switched off or on
4. include documentation which explains how to build and run the benchmark, including details on compilers, libraries, etc. and their versions
5. not be too small, as this will behave more like a synthetic benchmark. It is likely easier for a performance analyst to reduce the size of the benchmark, than increase it in any meaningful way

Writing a benchmark can be time consuming, however, we think that benchmarks are not only important for the researchers and developers on a particular codebase, but benchmarks can also be a research output. Benchmarks can become adopted by computational communities, and help compute centres build suites of benchmarks for procuring new systems [9].

References

[1] J. D. McCalpin *et al.*, "Memory bandwidth and machine balance in current high performance computers," *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, vol. 2, no. 19-25, 1995.

[2] A. Poenaru, W.-C. Lin, and S. McIntosh-Smith, "A performance analysis of modern parallel programming models using a compute-bound application," in *International Conference on High Performance Computing*, pp. 332–350, Springer, 2021.

[3] S. McIntosh-Smith, J. Price, R. B. Sessions, and A. A. Ibarra, "High performance in silico virtual drug screening on many-core processors," *The International Journal of High Performance Computing Applications*, vol. 29, no. 2, pp. 119–134, 2015. PMID: 25972727.

[4] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSBench - the development and verification of a performance abstraction for Monte Carlo reactor analysis," in *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, (Kyoto), 2014.

[5] J. R. Tramm, A. R. Siegel, B. Forget, and C. Josey, "Performance analysis of a reduced data movement algorithm for neutron cross section data in monte carlo simulations," in *EASC 2014 - Solving Software Challenges for Exascale*, (Stockholm), 2014.

[6] J. Fehr, J. Heiland, C. Himpe, and J. Saak, "Best practices for replicability, reproducibility and reusability of computer-based experiments exemplified by model reduction software," *arXiv preprint arXiv:1607.01191*, 2016.

[7] T. Koskela, I. Christidi, M. Giordano, E. Dubrovska, J. Quinn, C. Maynard, D. Case, K. Olgu, and T. Deakin, "Principles for automated and reproducible benchmarking," in *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, (New York, NY, USA), p. 609–618, Association for Computing Machinery, 2023.

[8] T. Hoefler and R. Belli, "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*, SC '15, (New York, NY, USA), Association for Computing Machinery, 2015.

[9] A. Herten, S. Achilles, D. Alvarez, J. Badwaik, E. Behle, M. Bode, T. Breuer, D. Caviedes-Voullième, M. Cherti, A. Dabah, S. E. Sayed, W. Frings, A. Gonzalez-Nicolas, E. B. Gregory, K. H. Mood, T. Hater, J. Jitsev, C. M. John, J. H. Meinke, C. I. Meyer, P. Mezentsev, J.-O. Mirus, S. Nassyr, C. Penke, M. Römmer, U. Sinha, B. v. S. Vieth, O. Stein, E. Suarez, D. Willsch, and I. Zhukov, "Application-driven exascale: The jupiter benchmark suite," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–45, 2024.