Performance Analysis and Assessment—A Developer Workbook

Tobias Weinzierl and Thomas Flynn

June 4, 2025

Preface

April 2025

We organised a series of performance analysis workshops in spring 2025, funded through the Digital Research Infrastructure (DRI) programme under the umbrella of our HAI-End project. These workshops differed from previous installments in that we did not focus on particular tools. Instead, we conducted brainstorming and discussion exercises centered around a fundamental question: How do you begin performance assessment when you know little to nothing about your code?

While some tools such as Intel's Application Performance Snapshots (APS) or Linaro's MAP provide high-level overviews, we approached this challenge not from a tools perspective but from an observational point of view. We asked: What are the essential properties we must understand before diving deeper into code analysis?

This inquiry led to a high-level, top-down approach, documented through mindmaps, flowcharts, and case studies. The raising approach is fundamentally different from the existing POP methodology, which collects fine-grained metrics and successively aggregates them into high-level metrics and flaws. It also differs from approaches offered by and supported through tools like Scalasca, which require a certain level of code maturity and system knowledge from the user.

Both direction of travel are important and very valuable, and we think they might be more sophisticated and insightful compared to what we came up with. However, our technique is very simple and does require next to no specialised software and tools knowledge. We developed a straightforward, accessible starting point to initiating performance analysis. This first version of the document summarises these foundational ideas.

Some of the sections are not yet written, and a lot more are yet to come. However, we decided to share them early, to be able to gather feedback and polish the ideas. This is only the first version of a living document.

Thomas Flynn, Tobias Weinzierl
Durham, June 4, 2025

Contents

Part I

Introduction

Chapter 1

How to read

As with any workbook — i.e. a book that's there to guide actual work not to teach a big subject — there are numerous ways to read the text. You can read the manuscript chapter by chapter, but this is unlikely to be the most fun experience. Below, we collect some ideas on how to substructure the reading:

Context

This document is part of an ongoing body work of designing and implementing a structured performance analysis methodology with the future aim of building a performance assessment service.

The work presented here is a precursor to a far more detailed text which will include a structured methodology for conducting performance assessments. In this iteration we introduce: relevant terminology for High-Performance Computing (HPC) and performance analysis; preparing for an assessment; and finally how to begin an assessment from a high-level.

Current state of manuscript

Both Chapter 2 and 3 are predominantly complete though with notable exceptions such as Section 3.4 which are yet to be completed, and are labelled as 't.b.d'.

Within Chapter 4, the first three sections are largely finished and have been written in conjunction with implementing these methodologies during recent performance analysis workshops. Section 4.4 has been started but the ending of this section still needs significant work. We want to implement more of this 'inter-node' assessment before making recommendations that are unfounded. Sections 4.5 and 4.6 are yet to be worked through, though Section 4.7 is mostly complete.

Clearly sections labelled as 'complete' may also need to be extended as further work is done, as the content may become insufficient. Note however that some forward references may link to topics to come in future versions of this manuscript, e.g. discussions of tools, performance assessment service, etc.

Finally, Chapter 5 will not be part of the final document (as with this section and the previous) but recaps the current progress of this document whilst also discussing the content to come in the next iteration of this document.

First-time assessment

We recommend to run through the preparatory remarks in Chapter 3 first and to complete all the checkboxes (dos) there. Once completed, it is time to run through the high-level assessment remarks in Chapter 4. With most codes, it is highly unlikely that a code will only show one kind of flaw in the high-level analysis.

We propose to run through the detailed analysis step by step afterwards. Per step, it makes sense to consult the tools summary (Chapter **??**) and notably to try out different things. Where appropriate, the reader might prefer to read through one of the case studies in Part **??**, just to get some inspiration on what tools and approaches to try out and how to report outcomes.

Assessment

If you have already completed numerous assessments, then the manuscript is mainly a reference document. You'd start with the high-level assessment in Chapter 4, before you dig into the various subchapters depending on the initial assessment outcome. The case studies are not of particular relevance (anymore), as you have already completed you own case studies.

Feedback

The quintessential objective of any assessment is to give feedback to developers. Once they make changes to their source code, it is important to re-start the assessment from scratch. There is always this temptation to return to the particular characteristics of interest that have led to a report, to see how the code changes by the development team have altered these characteristics, and then to continue from there.

It is important to reassess the "flaw" metrics. However, this should only be a validation that something has changed. After that, we have to start from scratch, as performance optimisation is a complicated process where one change might introduce flaws, improvement or character changes in a completely different part of the code or affect a completely different flavour of the performance. If we alter a particular code part that makes the scaling deteriorate for example, this might have a severe knock on effect on the core performance. If we improve the core performance, the GPU offloading might all of a sudden run into data transfer issues. There are numerous examples for these side-effects.

How it works

- ⊞ Conduct a performance assessment which both covers all aspects of the code (cmp. Chapter 4) plus goes in-depth all the way through for all flaws identified.
- ⊞ Hand this completed assessment back to the developers to address the flaws.
- ⊞ Restart the assessment from scratch top-down, i.e. starting with the high-level assessment covering all code runtime nuances.

How it does not work

- ⊟ Mix code optimisation and performance assessment workflows.
- ⊟ Restart the assessment from the place where you stopped when you reported a flaw and continue digging through the assessment decision metrics without redoing a high-level, overarching assessment.

Chapter 2

Terminology

Performance and efficiency, i.e. the speed of a code, are critical non-functional properties of scientific codes. Measuring, assessing and interpreting them is challenging; without even considering tuning. Most annoyingly, there is no standardised way how to analyse and assess performance. There is no vanilla workflow. Every team, every project, every scientist create their own analysis workflow or process, and this processes often changes from paper to paper or experiment to experiment.

Despite the vagueness in the metrics and the processes, there are some well-defined, common terms in the community that we should use. There are also some key concepts and some jargon that we use throughout the write-up. So we better introduce them first.

2.1 What we do

From hereon, we rely on a few key assumptions:

1. We assume that the person studying the code has *limited knowledge* of the code's internals, the application domain, and the algorithms used.
2. We evaluate *how* a code performs and whether certain characteristics deserve closer examination.
3. We are interested in the code's behavior on *one particular machine* for *one particular experimental setup* or a closely defined set of experiments.
4. Whenever we explain *why* a code performs as it does, this explanation comes from a machine's point of view: It does not argue about the algorithm or science case.

Reader's guide

Feel free to skip the remainder of this chapter if you are a first-time reader.

In scientific computing, we typically work with an iterative approach: First, we *measure* or *benchmark* how a code performs. Second, we use these data to inform our *performance assessment*, i.e. our reporting. Third, we conduct a *performance analysis* to explain why the code exhibits the observed behaviour. Finally, this analysis feeds into program tuning or optimisation, and we repeat the workflow.

If we take our list of activities and integrate it within the definition of the performance analysis workflow, it becomes clear that the tuning step is out-of-scope here. Another related technique is out-of-scope, too: The most advanced papers in scientific computing, particularly within high-performance computing, deliver a performance model; essentially a pen-and-paper assessment of what a code should be able to deliver. This is impossible if we adhere to the principles above: having limited knowledge and only explaining observed behavior rather than relating it to external factors (such as input data or algorithmic choices) or algorithmic context. We commit to a data-driven approach. Obviously, such an approach might complement an analytic, theoretical assessment of code, but we recommend considering this as an entirely separate cup of tea.

We will inevitably face difficulties if we truly do not know the code we are evaluating: Explaining certain effects is so much easier with some (rudimentary) understanding of the code. However, there are three compelling reasons to treat code as a (logical) black box: First, performance assessment is often delegated to non-developers: colleagues from computing centers, centralised HPC specialists, or external service providers. If we want to discuss performance analysis methodology, it is counterproductive to assume in-depth code knowledge. Many people conducting assessments cannot be experts in every code they examine. Second, requiring code expertise within assessment teams would imply that core developers are always available. This might be an ideal in the spirit of eXtreme Programming [1], where the customer must be accessible at all times. It is unrealistic in science, where PIs are busy with various tasks, developers frequently leave projects—as they graduate, accept new positions, or are reassigned—and where code components are often written by individuals and are so complex that only a few people understand the core parts. Finally (and most importantly), knowledge about a codebase tends to bias the assessment.

Developers inevitably make assumptions about reasons for code behaviour when working on performance analysis. (Senior) Developers tend to "defend" their code, explaining any flaws as natural consequences of their domain's challenging nature, their awareness of future requirements, or their in-depth knowledge of other setups not covered by the present configuration. Simultaneously, we have frequently observed junior developers downplaying or challenging existing code—likely to prove themselves by pointing out "how things really should work"—and consequently interpreting every piece of data through this lens. This list is not comprehensive. The whole team dynamics dimension deserves further discussion (in Section **??**). For the time being, we conclude that having emotionally invested developers involved in early assessment is counterproductive due to their inherent bias.

Work in progress/todo The link to team dynamics is broken as this part is work in progress and not yet written.

Assessment crime 1 *People conduct assessments as code experts. Due to this, they deliver biased assessments since they think they already know the code's performance characteristics and its flaws without an objective assessment of its "real" behaviour.*

For a high-quality assessment, it helps to step back and (pretend to) know nothing. This way, we avoid entering performance analysis with bias. Adhere strictly to the observational perspective. Even if it is your own code, try to detach yourself and pretend you have no prior knowledge. Abstract.

How it works

- ⊞ Pretend you know nothing about the code. Blend out your expert knowledge and start on a fresh sheet.
- ⊞ Stick to the data you measure.
- ⊞ Ignore what others pretend to know or have found.

How it does not work

- ⊟ Start with the attitude "I know what this code does and where there are weaknesses".
- ⊟ Skip assessment steps, as you already know what's going on.
- ⊟ Involve someone who's emotionally involved in the code's development into the reporting. They will always defend their brain child.

2.2 Fundamental HPC and assessment terms

We try to keep the HPC jargon in this write-up to a minimum. You might argue that this is a bad idea as we want to run scientifically sound assessments. However, we have to acknowledge that most scientific code today is written by people without a computer science background, and that most scientists do not have an extensive HPC training. It is therefore not only a matter of accessibility of the performance analysis and assessment itself to try to avoid too much jargon—if we use too much of it, we run the risk that our outputs, i.e. the assessments themselves, cannot be read and interpreted appropriately by our target audience. After all, most of the assessments are not done for ourselves, but for other scientists.

At the same time, performance analysis tools are written by computer scientists, software stacks are written by computer scientists, and computer hardware is built by computer scientists. It is hence clear that there's a lot of jargon buried in any conversation around performance. We have to find a healthy middle ground to make precise, in-depth assessments and analyses, without immersing ourselves in technical fuzz.

Some of the most important performance analysis terms When we speak of *profiling*, we mean recording and presenting (accumulated) metrics. The classic example is measuring how much time we spend in a function or waiting for a message to arrive.

Tracing is the counterpart to profiling. Here, we record performance data of individual events. We are interested in the sequence of operations occurring within the system. Profiling focuses on the effect of program execution while tracing focuses on the source of behaviour (Figure 2.1). Many developers confuse these definitions, which we took from the VI-HPS community.

Tracing can certainly be used to generate profiles, provided the recorded performance data encompasses all quantities required for a particular metric. Conversely, the reverse problem is ill-defined: it is generally impossible to reconstruct a trace from a profile without specialized knowledge of the code's internal workings. One might immediately ask why we don't trace all the time. The answer is equally simple: It is too expensive. Tracing generates enormous amounts of data. Therefore, all real-world tracing relies heavily on filtering, where you determine beforehand which types of events to trace, which quantities to measure, and which program parts to monitor. Consequently, we revise our introductory statement: most traces are *not* well-suited for deriving program metrics, as they are, by definition, incomplete.

To create a profile, we distinguish between two techniques: We can *sample* program execution. Essentially, we run the code, pause it at regular intervals, and each time examine our metric of interest at that point. Over time, we gather a good impression of which functions are called most frequently, how many MFlops/s we achieve, and so forth. Some people refer to this process as monitoring. As with all statistical methods, the data quality improves the longer an application runs. However, the risk of missing important but brief effects remains.

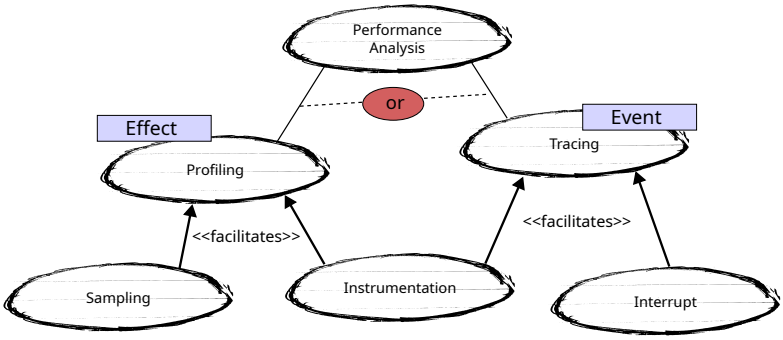


Figure 2.1: Overview of some fundamental performance analysis concepts.

Alternatively, we can *instrument* an application. Instrumentation involves inserting measurement code into our actual code which then measures quantities of interest whenever these insertion points are reached. We can insert either at compile time, or in a postprocessing step after the compile, or alter the code as we go. Instrumentation yields very accurate data (provided we instrument all relevant code sections), but it can become prohibitively expensive.

Most functions in our applications call other functions. Therefore, we must always clearly state whether any metric we present (e.g., runtime) is *inclusive*, comprising all contributions from called routines, or *exclusive*, representing only the function itself.

When tracking metrics, measuring time and call frequency is straightforward: we either measure system time or simply count occurrences. Other metrics are more complicated to obtain. Fortunately, modern systems offer performance counters. They are special registers within the chip that automatically count certain events. For example, they can be configured to count every floating-point operation or every main memory access, allowing us to sample them or read their values upon completion. Performance counter registers are expensive to manufacture, so they are limited in number, and not every vendor tracks all possible events. Consequently, some analyses might not be possible on every system, and complex analyses might require multiple code executions, as only a few event types can be measured per run.

As we know about the existence of hardware counters, it is clear how sampling and tracing can obtain machine data beyond mere timing and call counts: They simply monitor (read out) the counters. Through this, they can make statements about the number of floating point operations completed, the number of cache misses, and so forth. For many of these concepts, we can invert the control flow: Rather than polling a counter or quantity, we can define *interrupts*, i.e. situations in which the program flow is stopped and we call our profiler or tracer. Interrupts can be realised through hardware or software. Their beauty is that the performance analysis sits somewhere outside of the core program logic—most of the time, our code runs totally unaware of the analysis in the background.

Hardware jargon We revise the bare minimum of hardware jargon here to get started, and then refer to further resources to revise details.

Supercomputers consist of many different individual computers linked through a high-performance interconnect (network). These computers are called *nodes*.

Each node is a full-grown computer and hence hosts many individual *cores*. All the cores on a node share their memory (but cores from different nodes do typically not share memory). Technically, many vendors call their cores *hardware threads*, as a core is a technical building block and might host multiple threads. We use hardware thread and core as synonyms.

On larger systems, you do not use the compute nodes directly. Instead, you log into a *login node*, you compile your code there, and then you hand your code over to a *scheduler*. The most popular scheduler today is Slurm. Slurm takes your request, i.e. how many nodes you want to have and for how long, as well as all the other requests from all other users and puzzles out a fair and efficient way to assign the resources. For performance analysis, it makes sense to investigate if your system of choice offers a test or benchmarking queue with a high priority. These queues usually offer only a few nodes and restrict the maximum application runtime. In return, jobs handed over to these queues are scheduled with high priority.

Compilation should be on the login node, as the actual compute nodes might not provide all the required software, and many production nodes are also shielded from the Internet, i.e. even updating you repository is a pain. Benchmarking should not be done on the login nodes, as you share them with all other users. In contrast, you can tell the scheduler that you want to have nodes *exclusive*, so no other user interferes with your measurements. This is particularly important for all data stemming from performance

counters, as these counters are a hardware feature that we read out, i.e. our profiler or tracer cannot distinguish if a hardware counter's data stems from your code or another code if multiple users coexist on one machine.

Core performance quantities The runtime of a code is written here as $t(p)$ where the p is either the number of nodes or number of cores. It depends on the context. The *speedup* of a program is defined as

$$S(p) = \frac{t(1)}{t(p)}, \tag{2.1}$$

which is typically smaller than or equal to one and tells us how much faster we made an application by using p compute units (cores or nodes) rather than only one.

The *parallel efficiency* is given by

$$E(p) = \frac{S(p)}{p}. \tag{2.2}$$

It quantifies to which degree adding further compute resources was worth it.

2.3 Bottom-up vs. top-down assessment

Reader's guide

It might be better to skip this section and to read it after the first high-level assessment is completed.

Work in progress/todo This part is yet to be written and will basically point out how the POP methodology differs from what we do here.

Chapter 3

Preparation

Reader's guide

This section is absolutely key before you start an assessment!

In this chapter, we discuss various steps that we might want to do *before* we dive into any benchmarking or assessment. It summarises activities that ensure that we are well-prepared to run assessments, but it also comprises steps that ensure that we understand the code sufficiently to come up with unbiased and valid statements.

3.1 Set up the benchmarks

Before we start a performance assessment, we need to agree on what type of measurements to perform: There has to be one well-defined benchmark. If we are given a set of setups and try to compare them, this will make our task complicated. We run the risk of comparing different characteristics produced by different code parts.

- This one benchmark has to run without any user interaction. In the normal case, the sponsor has to provide a script which builds and runs the whole benchmark "as a black-box".
- It has to be very easy to scale that benchmark up and down, i.e. to make the problem solved slightly bigger or smaller.
- The benchmark has to be a simulation run which is representative of a real-world run.
- The benchmark does not run for too long, and it does not terminate too early. "Good" runs are often 3–5 minutes long, so we get a good coverage of performance characteristics, but we do not wait for ages.

None of the properties are surprising: We need a well-defined toy setup, and we have to be able to modify its size to make statements on its scalability. In this context,

it is important that we know exactly how changes of input sizes affect the compute workload: We have to know or have to be told the computational complexity of the underlying algorithms. If we have a pure Monte Carlo setup, then doing 100 times more trials should last around 100 times longer. If we have a parabolic Partial Differential Equation (PDE) solver with explicit time stepping and a regular grid, we have to know that the problem sizes increases by a factor of 2^d if we half the mesh spacing. However, these algorithms also have to reduce the time step size, so if we measure over a fixed time span, we actually will have four times more time steps and therefore increase the workload by a factor of 2^{d+2} once we half the mesh size. Finally, if we run an Adaptive Mesh Refinement (AMR) code for a hyperbolic setup with explicit time stepping or a Smoothed-Particle Hydrodynamics (SPH) code, we need some kind of output that tells us exactly how many particles or mesh cells we have and what time step sizes we have picked. In this case, time step sizes go down linearly with mesh spacing. At the end of the day, we are interested in the *throughput* of the system, i.e. the time we need to update one quantity of interest. It is this metric that we will start from in the next steps.

However, if domain experts tell us afterwards, i.e. after we have completed our assessment—they will always do that afterwards when they are unhappy with the outcome—that this particular run is not characteristic for larger runs, as it lacks a feature or does not really use a particular routine heavily, our whole assessment exercise is corrupted. And with it the whole assessment will be a big disappointment. It is their job to pick a setup that represents what we are interested in and is nevertheless reasonably small.

Having a "small" setup that completes in reasonable time is important since some performance analysis tools induce significant overheads, i.e. make the code significantly slower. If our vanilla run already needs more than an hour, we will never get an answer on time. Furthermore, a lot of tools gather significant amounts of data per second. Long-running simulations thus run the risk that our analysis tools fail to handle the amount of data collected.

Checklist

- [1] Your benchmark compiles as a black box, i.e. you can give it to another person together with a `README` file and there might be a few instructions that they have to copy-n-paste into their terminal, but other than that there's no other preparatory work to be completed.
- [2] Your benchmark runs without any user interaction.
- [3] Your benchmark finishes within less than ten minutes on a single core.

No matter how we create this setup, we have to be able to run all experiments with absolutely minimal user interaction. If we have to run through a complex pipeline for each individual experiment, we will not be able to measure anything productively.

How it works

- ⊞ Ask the code owner ("assessment customer") to provide you with a ready-to-use benchmark.

⊞ Validate that this benchmark meets the criteria outlined in this section. Otherwise, return it immediately.

How it does not work

⊟ Try to make a code run and design a benchmark yourself. This is the job of a domain/code expert.
⊟ Expect insight from benchmarks that somebody without domain expertise has chosen.

3.2 Working environment

Work in progress/todo Yet has to be written, but will cover

- Access to some queues with quick turnaround times (recommended);
- A couple of standard assessment tools. We refer to them later (forward link). Reader recommendations: Study them as you go along.
- Exclusive access to nodes.
- Some higher user priviliges (advanced).
- A good relationship to your sysad.

3.3 Compiler setup

Before we start, it is important to double-check whether we have used the correct compiler settings. It makes no sense to benchmark a code which is not producing tailored code for our particularly machinery (therefore it is important to benchmark on the right system), and it makes no sense to benchmark a code that is not using the latest compiler features.

First of all, we work with the the most aggressive optimisation that our compiler can offer. The minimum choice is to compile with `-O3`, although some other compilers might offer further granularity. Also consider using `-ffast-math` which enables aggressive floating-point optimisation.

Second, we create code that is tailored towards our particular machinery. The correct setting for this depends once again on your compiler. With LLVM and GNU, you can use flags starting with `-mXXXX` to specify machinery-specific optimisations, i.e. you can exactly say for which processor generation, instruction set, ... you want for the compiler to create binary code. If your compile node is of the same type as your test node, then simply use `-xhost` and the compiler will pick the highest instruction set available. The flag is called `-mhost` with GNU.

Intel compiler

With the Intel compiler, you sometimes can pick between the generic LLVM (community) versions of a feature and one optimised by Intel itself (with the latter obviously performing better on bespoke Intel architecture). For example, `-fopenmp` enables the general OpenMP implementation whereas `-fiopenmp` gives you an in-house version of the same.

And while `-xhost` uses Intel-specific optimisations, `-mhost` will use LLVM's optimisation stack, i.e. Intel has replaced some optimisation steps with bespoke variants, which are enabled if and only if you go down the `-xhost` route. Be aware that Intel's optimisation passes typically will be disabled if you use the Intel toolchain on other vendor's hardware, i.e. you'll get unoptimised code.

Finally, we might want to add debug symbols. These instruct the compiler to insert additional information into the source code which tools can then later use to draw inference from, e.g. which source code line a certain instruction stems from. Usually, you cannot know from a single instruction in a machine code, where this comes from in the original code. With the debug systems, we inject exactly this information into the executable. Therefore, we should add `-g`. Some compilers support more detailed information (e.g. using `-g3`). It requires studying the compiler's feedback whether these flags disable optimisations, in which case we might have to balance insight vs. best-case speed.

Checklist

1. Your benchmark is compiled with debug symbols (`-g`).
2. Your benchmark is compiled with the highest optimisation level (`-O3` plus a hardware-specific instruction set such as picked by `-xhost`).

MAQAO

If you run MAQAO over your code, the tool will report back to you what optimisations you might have missed out throughout the compilation. The top-level rubric *Global* provides you with information on the optimisation chosen, but it also makes recommendations which options might lead to better runtime.

The nice thing about MAQAO is that the tool can handle situations where you translate different translation units with different options. Also, the recommendation aspect goes beyond pure reporting and is helpful.

How it works

- ⊞ Use the most aggressive compiler optimisation that preserve the code semantics (aka still return valid results) and nevertheless are tailored towards your assessment machinery.

⊞ Check what optimisations the users' build system uses by default. Often, this is a low-hanging fruit.

How it does not work

⊟ Use a low optimisation level and machine-generic optimisations and then draw conclusions on code behaviour on a particular machine.

3.4 Understand the code's complexity

Reader's guide

The complexity discussion is mainly relevant when we conduct an MPI (weak scaling) assessment. In many cases, we might be able to skip it initially.

Work in progress/todo This part is missing but we need some discussion of the $O(N^2)$ vs. $O(N)$ behaviour that we see frequently.

3.5 I/O

Work in progress/todo This section is only partially complete. We cover the terminal outputs but miss out on the "real" I/O.

Assessment crime 2 *The code writes a lot of information to the terminal.*

String operations are excessively expensive.

Checklist

1. Your benchmark does **not** write (a lot of) files.
2. Your benchmark does **not** write a lot of information to the terminal.

In this context, it is important that there's no I/O. We don't want to study complex outputs; we are busy enough with handling the output of performance analysis tools.

Assessment crime 3 *The code reads or write to files excessively.*

More importantly, if our code writes excessive amounts of data, we almost certainly profile the I/O efficiency rather than the runtime behaviour. Unless we want to explicitly study I/O, we are well-advised to switch off any file outputs, database outputs, but also excessive file reads. A log file alone can become problematic when these files become suddenly huge or are updated frequently. In the best case, a performance assessment benchmark writes and reads barely any data at all, i.e. it largely hard-coded and silent.

How it works

- ⊞ Disable I/O, exclude reading and writing files from any assessment as far as possible (and as long as you don't want to benchmark exactly this I/O).
- ⊞ Remove terminal dumps. If the developers want to preserve the output (e.g. for validation or debugging), suggest to embed them into macros which can be compiled out of the code prior to production runs.

How it does not work

- ⊟ Start with a high-level assessment straightaway. In particular any shared memory measurement will be corrupted by excessive terminal dumps.

3.6 Machine details

Modern supercomputers are very complex and it is virtually impossible to memorise all the hardware details of a testbed given the zoo of processor types, generations and flavours. While it is possible to consult your compute centre's webpages or to consult your system administrator, it might be more reasonable to obtain that data directly from the machine.

Assessment crime 4 *Never grab the specification from the login node. Instead, ask for an interactive terminal on a compute node and get the data there. Many supercomputers' login nodes are slightly different to the compute nodes, i.e. slightly different make, more memory, more cores, . . . So you might end up with the wrong hardware specification if you analyse a login node.*

Linux command line

On every Linux system, information about the CPU on the system is held in a central file. You can read it out by typing `cat /proc/cpuinfo` into the terminal.

Likwid

If you have Likwid installed, you can invoke `likwid-topology` to obtain in-depth information about your CPU.

MAQAO

In MAQAO, you find some topology information in the menu rubric *Topology*.

How it works

⊞ Gather the exact spec of your testbed under realistic load. If this information is already available (e.g. due to previous assessment exercises), it makes sense to reuse it.

How it does not work

⊟ Throw advanced tools onto your code and expect that they report the theoretical machine capabilities themselves.

3.7 Labelling of code parts

Reader's guide

You might want to skip this section initially and return to it later, once you need information about what certain code parts do.

Software typically contains quite a lot of boilerplate code: predominantly administrative routines that enable calculations but don't deliver actual science flops. For several follow-on exercises, it will be very useful to know which code parts deliver science and which code parts do not directly add scientific value. This is domain knowledge. Using it contradicts, to some degree, our ambition to run a performance assessment in a black-box manner.

However, we do not ask an assessor to understand what a code does. We require the assessor to know which code parts contribute towards the science case. For this reason, it is absolutely sufficient to hand a first profile back to a domain expert and to ask them to label those routines that do the actual calculations.

As an example: In a linear algebra code, the actual matrix-vector products, scalar products, ... all deliver science. The allocations for temporary block matrices, the exchange routines for rows of the matrices, any indexing over the sparsity pattern, and so forth do not contribute calculations. They are absolutely essential, but they do not give us the MFlops/s.

As your analysis progresses, you might run into situations where functions suddenly appear to be hot that you have not seen before. In this case, you can kind of safely assume that these functions do *not* contribute to the science directly. If they didn't show up for a small-scale, few-core run, then they likely do not feed directly into the result.

VTune

Start a multithreaded run and mark the actual computational phase in the timeline at the bottom. Switch to the *Bottom up* view. *Zoom in and filter by selection* will narrow down the profile to the region of interest. Sort by *Effective Time* and change

to *Show Data As . . . Percentage*. Make notes of all the entries above roughly 1% (rule of thumb) and/or track the top 10 routines.

How it works

⊞ Upon your very first profiling, ask a code expert which code parts (functions) actually perform calculations, i.e. contribute towards the science.

How it does not work

⊟ Assume that all routines deliver science (even though you may assume that all are necessary).

3.8 Existing optimisations

Reader's guide

Playing around with (user-driven) optimisations can become tricky and might require the assessor to communicate with developers (if you are not the same person). For an initial assessment, it might be very reasonable to skip this section.

It is rare that we work with a code that hasn't gone through a series of optimisation steps already. Most developers think at one point about how to make a realisation fast. However, optimisation steps can backfire:

First, many code developers have no systematic training in HPC and/or do not apply rigorous performance analysis. Consequently, their realisation does not tackle hotspots. In the worst case, the resulting optimisation steps make the code more complicated than it actually has to be.

Second, many scientific codes are old. They contain legacy code parts. If these code parts had been optimised, the optimisation steps might not be valid for today's computer architectures anymore. For example, compute time used to be expensive and performance engineers hence used to introduce precomputed data tables in the past. Today, memory bandwidth is likely the most precious resource. Having extensive lookup tables can actually impose additional stress on the memory interconnect, and a recomputation of values might be advantageous.

Finally, we have to be frank with our community: A lot of people who consider themselves to be outstanding computational scientists actually have a very limited understanding of computer architecture. That's not a problem per se, as long as people are aware of their lack of knowledge. It is problematic when we see developers introducing or pushing for "optimisations" which actually are not supported by data or insight, but rather individual "genius" and "expert knowledge".

Assessment crime 5 *We work with a code that the developers consider to be "highly optimised". Due to these optimisations, we start from a code base that is actually overly complicated (thus hiding the "real" performance flaws) and might actual underperform* due to *these optimisation steps.*

Many scientific codes have not been co-designed as HPC codes. They have been written first and foremost with functional requirements ("the science") in mind. The HPC came later. This is reflected in the code design: Optimisations are realised as add-ons wrapping around the core science (yet sometimes, unfortunately, also penetrating it). Therefore, we always have to keep in mind that performance has been seen as a second-class citizen in most development projects. By the time a project has reached a certain maturity and people suffer from performance flaws, they will ask for a performance assessment (and then want to be told that they have already done the heavy lifting or that their science is so challenging and completely different to everything else that it is very hard to get good performance). They won't ask for it prior to that.

If possible, it is helpful to disable all of these historic optimisations. We can then start with the performance assessment on a green field and re-inject the optimisations one by one as we go along. In an ideal world, they should make the code faster, but this has to be validated!

How it works

- ⊞ Disable existing user-driven tuning options.
- ⊞ Independently validate that they actually make the code faster.

How it does not work

- ⊟ Assume historical performance optimisation steps to pay off on current hardware.
- ⊟ Rely on domain scientists that their optimisation has been driven by measurements (formal performance analysis) and an in-depth insight into the machine.

Part II

Performance Assessment

Chapter 4

High-level first glance

There are various ways or strategies to obtain a first high-level overview and how to get started if we want to understand a code's performance better. Every team, every project, every scientist have their own approach, and it often changes from project to project. In this chapter, we propose a workflow running along few very simple principles:

1. We study the performance characteristics for *different machine part/aspects separately*.
2. We use a simple *blackbox performance metric* per dimension to find out first if there are issues or not.
3. We *do not use any tools* (yet).
4. We are not interested in explanations of *why* we see any performance flaws. We simply document *what flaws* we observe.

Per dimension, we later break all performance flaws found down into subrubrics, and dig deeper and deeper into the code. Along this way, we construct working hypotheses of why the code performs poorly. But this comes later. For the time being, we focus in a bird's eye view, which is an analysis which we can conduct, for example, with a simple electronic spreadsheet, a few tools and mere timings.

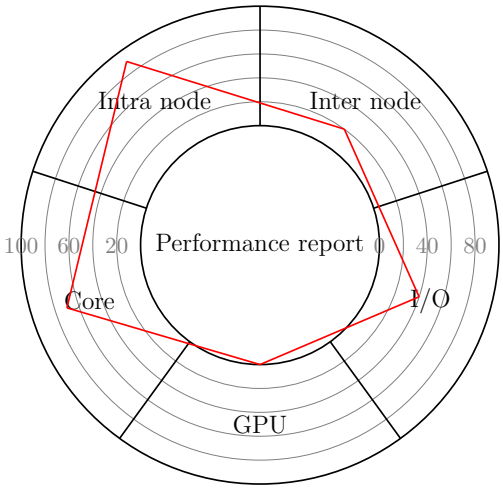
Assessment crime 6 *Modern preformance analysis tools (e.g. Intel Performance Application Snapshot (APS) and Linaro MAP) and methodologies offer a high-level overview as kind of an automatic service. This is a very valuable feature, as it provides you with a quick overview of your code's characteristics. While we appreciate these opportunities, we are however not sure if this is really the right starting point for performance assessment. In our work approach, we do not use one single run and its characterisation to get started, but we use many different runs (with different configurations) to zoom into particular feature dimensions such as intra-node scaling and see if there are issues there. Instead of one global overview, we obtain a set of metrics which we combine into an application profile. We not rely on a* **one run uncovers it all** *approach.*

4.1 The assessment rubrics

We classify codes along five different metrics:

1. inter node,
2. intra node,
3. core performance,
4. GPUs, and
5. input/output.

For each of these rubrics, we initially aim for a single three-valued: All is fine (green), there might be issues (yellow) or there are obvious flaws (red). In line with our overall presentation approach, we use thresholds of 60% and 80% to distinguish these classes. That is, if the GPU performance is somewhere close to 100%, all is fine. However, if it drops below 80%, we flag it as yellow. Once the metrics we define for GPUs underrun 60%, we claim that there are definitely major issues. It is convenient to present the overall data via a spiderweb diagram (below for a code that has I/O issues, doesn't scale particularly well between nodes, has no GPU yet reasonably good core performance and node scaling):



The idea to look at the five different performance dimensions separately is natural yet does come along without shortcomings. Many performance issues within a code affect multiple dimensions. Think about a poor GPU utilisation that materialises in bad load balancing as well. When we dig down recursively into finer and finer metrics later on, we have to take such cross-influence into account. Eventually, we might have to return to our highest level of assessment again and again.

We propose to use different benchmarks for the individual rubrics. Each benchmark will follow the recommendations from Section 3.1, but we will tailor the benchmark

characteristics to the effect we want to study, such that the effects of interest are most pronounced while others are less dominant. Notably, we will disable I/O for all studies unless we study I/O itself (cmp. Section 3.5). This implies that the final picture might not reflect the real behaviour of a science run.

As we analyse the dimensions of performance characteristics independently, the order does not really make a difference. In theory. In practice, we have to start somewhere. We have personally had the experience that it is typically favourable to start with the node-level analysis (intra-node): If the individual cores are not used properly, MPI often does not make that much of a difference and we struggle to keep GPUs busy. Often, the next dimension is naturally identified once we look at the node-level performance, i.e. it tells us what to analyse next.

How it works

- ⊞ Ignore most things you know about tools to avoid that you are not able to see obvious shortcomings.
- ⊞ When you report on your findings, focus exclusively on what you observe. Avoid any attempt to interpret things.

How it does not work

- ⊟ Throw a huge, powerful tool onto your code and expect it to flag problems immediately.
- ⊟ Start to measure without a clear plan of what you want to measure.

4.2 Core performance

For the core-level analysis, we examine how efficiently your code harnesses the chip's microarchitecture. Does your code leverage the most advanced instruction sets available? Are we maybe even compute-bound? Does it properly utilise the memory interconnect? Are we memory-bound? Today's chips come equipped with powerful vector computing capabilities and sophisticated cache and streaming features designed to move data smoothly between memory and compute registers. Our analysis evaluates how effectively your implementation takes advantage of these capabilities.

To capture core behaviour at a high-level we simply want to measure the code's bandwidth and peak performance relative to the maxima of the hardware. Tools can tempt us to reach immediately for a roofline model, or potentially even more detailed metrics around cache misses, vectorisation ratios, etc. These topics are relevant to a core performance analysis, but we want to start high-level or we otherwise risk focusing on an individual metric which obfuscates or distractes our analysis. By focusing simply on bandwidth and peak performance, our initial core performance investigation should sit us somewhere on the 'traffic light' matrix below.

Assessment crime 7 *We assess a setup that fits completely into one of the caches.*

Assessment crime 8 *We study a code that is not translated with the most aggressive compiler optimisation for the particular architecture that we study.*

Both properties are easy to check: Measure the total memory footprint of your code and ensure that the working set exceeds the largest cache you find on your system (cmp. remarks on caches below). Further to that, check your compiler's manual and ensure you use a bespoke code version for your target system—something that the `-march` or `-xarch` as well as the highest compiler optimisations (`-O3`) for example give you.

To some degree, the argument holds the other way round for the cores: Modern chips are often not capable of cooling all of their cores if all of them run at full compute capacity. They therefore downclock the system when they encounter lots of compute-intense operations. As we only assess scalar peak performance in this first exercise, this effect should not arise or be negligible. You might however want to double-check.

Likwid

With Likwid it is only a few lines

```
likwid-bench -t triad -W S0:2GB:1
likwid-bench -t triad -W N:2GB:128
likwid-bench -t peakflops -w S0:10000MB:1
likwid-bench -t peakflops -w N:10000MB:128
```

The Stream Triad [2] benchmark is a brilliant starting point to obtain realistic throughput bounds. As we are not (yet) interested in explanations of how well caches are used, it is important to pick a reasonably big setup that exceeds all caches. 2GByte should be plenty. It depends on our measurement strategy if we assess the single core bandwidth or take the whole node's (here with 128 threads) throughput and re-calibrate it accordingly.

The instruction set usage is something we can evaluate per core. Depending on the character of the code, we might want to pick a more advanced instruction set (e.g. `peakflops_sse` or `peakflops_avx_fma`). As long as we argue about the overall code performance, we find the plain peak performance to be absolutely sufficient. Being within 80% of that one for the total program execution is typically a good sign.

Once we have the thresholds, the code's performance itself can again be used by multiple tools. Again, likwid is our personal favourite:

```
likwid-perfctr -f -C 0 -g MEM ./mycode
likwid-perfctr -f -C 0 -g FLOPS_DP ./mycode
```

provides you with the information relevant, although the exact names of the groups might differ from system to system.

Amplifier

t.b.d.

Linaro MAP

t.b.d.

Evaluation Once we know the maximum bandwidth and the obtained bandwidth, we can compute the ratio (or normalised measured bandwidth) C_{bw}. Along the same lines, we determine the normalised peak C_{peak}. The latter is a little bit weird (and we will revise it later), as we typically normalise against the scalar peak performance. This is surprising given that all mainstream CPUs support vectorisation and hence can deliver several times more than that. However, we empirically found that starting with a comparison against the baseline is a good starting point: Few codes spend all of their runtime exclusively in very compute-heavy routines. Therefore, there will always be code parts which do not exploit vectorisation. If those that are compute-heavy do so and hence lift the average total peak into the regime of the scalar peak, this is already an overall efficient code.

Performance metric (rule of thumb) 1 *The delivered MFlops/s normalised against the scalar peak is a good high-level indicator for the compute efficiency.*

Performance metric (rule of thumb) 2 *The used bandwidth normalised against the memory bandwidth is a good high-level indicator for the compute efficiency.*

It is not clear whether normalising against the memory bandwidth is reasonable given that modern CPUs feature vast caches. The comparison furthermore is biased as we measure a single core but compare it to a value taken from the whole system divided by the core count. We however think that this normalisation nevertheless delivers a good initial high-level metric for a code, even though it favours the real code.

Following Section 2.3, we might introduce a multiplicative or additive metric over the two quantities to come up with a first assessment. As a matter of simplicity (and as we use both quantities later on), we use a third variant here: an and-based correlation of the two metrics.

	$C_{\text{bw}} \geq 0.8$	$0.8 > C_{\text{bw}} \geq 0.6$	$0.6 > C_{\text{bw}}$
$C_{\text{peak}} \geq 0.8$		perfect	
$0.8 > C_{\text{peak}} \geq 0.6$			flaw
$0.6 > C_{\text{peak}}$	severe flaw		

If $C_{\text{peak}} \geq 0.8$, the code uses the compute capabilities (very) efficiently. However, if the bandwidth $C_{\text{bw}} < 0.6$, there might still be additional space to improve the throughput

and to speed up the code. If $0.6 \leq C_{\text{peak}} < 0.8$, the code does not use the compute units particularly efficiently and we have to investigate further. However, if $C_{\text{bw}} \geq 0.8$, the problem might be inherently memory-bound (to be checked), in which case there might be limited potential to optimise further.

How it works

- ⊞ Run a standard benchmark to determine a realistic maximum throughput (bandwidth) and peak performance for a single core.
- ⊞ Assess your code to find out if it makes sufficient use of the available bandwidth and compute units.

How it does not work

- ⊟ Apply some performance counters as a black box without a focus on bandwidth and peak performance.
- ⊟ Jump straight for a tool with gives significant data on vectorisation, cache misses, etc.
- ⊟ Immediately dig into any performance flaw without evaluating the other four rubrics first.

4.3 Intra-node (node-level) performance

On an individual node, our first goal has to be that we use all the hardware threads efficiently. Some people prefer the term cores of threads. The term intra-node hence means all those cores sharing one memory. Hereby it does not matter if we have a code using MPI or a genuine multithreading language: We assess the runtime characteristics for a code physically sharing its memory. It might be written logically in a distributed memory fashion.

Our method of choice to argue about node-level performance is strong scaling. In principle, we ask ourselves "what would happen if we had only one core on our node" (we use the term core and hardware thread as synonyms from hereon). If our code needs 80 cores on one node, then we would expect that it only needs 40 cores on two, and 20 cores on four. This is obviously an artificial question to ask: Why would we not use all the node? Despite the fact that we have the cores available anyway and therefore might want to use them, finding an answer to the "what if" questions helps us to uncover inefficiencies. How much performance do we leave on the way as we use more and more cores? How much more computation should we be able to squeeze out of the machine?

The underlying performance model is called Amdahl's law, and we will return to this law in Chapter **??**. For the time being, we can abstract from any performance model or law, and simply focus on the scaling behaviour over cores, as we have defined it in Section 2.2: We alter the number of cores available to our code until we have reached all the cores available, i.e. use the whole machine, and track the time to solution dependence on core counts.

Benchmark preparation Throughout this assessment, we use one fixed problem setup. The setup has to be chosen such that we can compute it on a single core if necessary. However, it also has to be reasonably big such that it makes sense to exploit all threads of a node.

Assessment crime 9 *We start an intra-node assessment with a code which only uses a tiny fraction of the main memory.*

We may assume that a node and its memory are somehow balanced. If we don't use a reasonable amount of this memory for a test run, then there is simply not enough work to do to utilise all cores. We can barely blame the code. As a rule of thumb, we should have a memory footprint which is at least 20% of the total RAM available on the whole node.

Data collection For the data collection, we first run the simulation without any parallelisation, which is, trivially, a single-core run. After that, we switch to the parallel code version and vary the core count p_{intra}.

Assessment crime 10 *Use a code base that is relatively inefficient on a single core and use this as baseline for any scaling claims.*

It is important that we question the code developers on different code variants prior to the assessment: If they offer, for example, quite sophisticated task-based parallelism for shared memory, we still should calibrate all runtime against a serial code without these tasks. That is a fair benchmark that avoids any tasking overhead where it is not needed.

OpenMP

In OpenMP, we can simply set `OMP_NUM_THREADS` to achieve this. After that, we run our code and should automatically get the correct core count. Depending on your system's configuration, you might want to set some additional command line variables such as

```
export OMP_PROC_BIND=close
```

or similar to ensure that the operating system does not swap your threads around, i.e. decide at one point to migrate a thread to a different core. This never gives good performance. Be careful as well with setting the OpenMP threads manually while using another core specification technique such as Slurm's configs or tasksets (both below): If you do so, you might force all the OpenMP threads onto a few cores only, i.e. accidentally over- or undersubscribing the testbed.

Slurm

Using Slurm, we do not have set environment variables ourselves, instead

Slurmmanages resources and so we can request a quantity of cores with the `-c` flag, e.g.,

```
#!/bin/bash

#SBATCH -c 2

./my_exec
```

or equivalently use `#SBATCH --cpus-per-task 2`.

Linux command line

The one thing that works always is using

```
taskset -c 0-3 ./myexec
```

to employ, e.g., the first four cores only. However, you should be careful when you use this in combination with Slurm or OpenMP which also make choices on cores. You might end up with competing or contradictory settings and end up with a suboptimal system usage.

If we have a code base which employs MPI only, we set the number of ranks per node through the `mpirun` call or the scheduler (Slurm) environment.

Assessment crime 11 *If you have the opportunity to mask out the initialisation time and to focus only on the "real" runtime of your benchmark code, use it at this point. Notably when you work with very short execution times, the initialisation overhead might otherwise dominate your measurements.*

From the set of measurements (Figure 4.1), we first compare the two timings for a single core (the red dot vs. the green dot above it). This gives us a t_{serial} and a $t_{\text{intra}}(1)$ given in red and green, respectively. As we use different executables—once compiled with support for shared memory and once without—these times will differ and we'd assume $t_{\text{serial}} \leq t_{\text{intra}}(1)$. If the code is not available without any parallelisation, assume $t_{\text{serial}} = t_{\text{intra}}(1)$. After that, we collect the remaining $t(p_{\text{intra}})$.

It is not unheard of that codes yield data where the runtime starts to raise again once we exceed a certain core count (yellow dots). This means that the speedup deteriorates completely. In this case, it is perfectly reasonable to constrain the maximum p_{threads} such that the we do not run into this "problematic" range of threads and to neglect the measurements further on. The performance flaws will already be documented for the smaller core counts where we at least continued to see some speedup.

For hybrid MPI+X codes, we have a certain freedom: We can use an arbitrary number of MPI ranks—indeed our analysis also works for pure MPI codes as long as we stick to one node—and use threads on top such that we exploit all p_{threads} hardware threads of interest. This means, we have a certain degree of freedom. Picking one rank

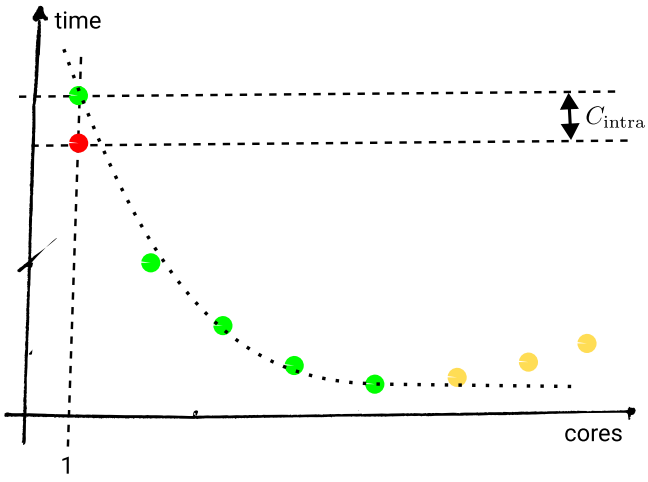


Figure 4.1: A scatter plot tracking the (strong) scaling data feeding into our intra-node analysis. The red node is the code's runtime without any parallelisation enabled, while the green and yellow measurements stem from the parallelised code over various cores.

and four threads is, logically, equivalent to picking two ranks and two threads each. It is part of the assessment to play around with these degrees of freedom. Once again: If we spot sudden runtime jumps (yellow dots in example sketch) and if these jumps coincide with a NUMA domain or socket, then this might be a strong indicator that we should use MPI instead of threading. However, one might argue that this is already part of an in-depth analysis rather than a high-level overview.

Evaluation With the speedups at hand, we can compute the code's parallel efficiency. We use a slightly modified version as compared to Equations (2.1) and (2.2), where we calibrate the runtimes against the serial code runtime:

Performance metric (rule of thumb) 3 *The modified parallel core efficiency*

$$E_{intra}(p_{intra}) = \frac{t_{serial}}{t(p_{intra}) \cdot p_{intra}}$$

normalised against the serial runtime without any parallelism support is a good indicator for inter-node performance flaws.

We use this quantity to determine if we consider the code to have an intra-node flaw. Obviously, we will obtain a series of these quantities. The assessor might decide to compute an average, to argue over different regimes of values, or to study only the value for the maximum core count.

Efficiency	Description
$E_{\text{intra}}(p_{\text{intra}}) \geq 0.8$	Shared memory scaling is good.
$0.6 \leq E_{\text{intra}}(p_{\text{intra}}) < 0.8$	Shared memory scaling is not particularly good, and we notably might run into problems with the next generation of chips where the core count will increase.
$E_{\text{intra}}(p_{\text{intra}}) < 0.6$	Shared memory scaling is poor.

How it works

- ⊞ Compile your code without multithreading/parallel support and compare its performance to a code without any parallelisation to determine the overhead of the parallel implementation.
- ⊞ Run a strong scaling study and calibrate the constants of an extended Amdahl model against these measurements.

How it does not work

- ⊟ Skip intra-node scalability studies, as the code is "only" MPI.
- ⊟ Perform a sole strong scalability study without taking the implementation penalty into account.
- ⊟ Jump straight into a further "in-depth" analysis without evaluating the other high-level metrics.

4.4 Inter-node performance

Inter-node behaviour means that code runs over multiple physically separated nodes. In our case, this implies that it uses MPI. We work with a programming model which treats memory physically and logically as separate. There are software solutions that pretend to the code to have one large shared memory. Such virtually shared memory solutions would fall into this assessment category as well, so long as we use more than one physical processor.

Our method of choice to argue about inter-node performance is weak scaling. We apply basically the vanilla version of Gustafson's law [3], where we assume that the runtime of the code is determined by

$$t(1) = t(p_{\text{ranks}}) \left(f_{\text{inter}} p_{\text{ranks}} + (1 - f_{\text{inter}}) \right) . \tag{4.1}$$

In this model, we reconstruct the time $t(1)$ for a code that we would need to run an experiment on one rank only. Implicitly, we assume that f_{inter} stays constant for all setups. This deserves a brief excursus: If we took one setup and distributed it among multiple nodes, we typically would introduce more and more overhead as we use more ranks. That is, we increase f_{inter}. To keep it constant, we have to increase the problem size appropriately, such that the f_{inter} remains roughly the same for all setups, i.e. no matter how many ranks we use.

Given said "we alter the problem size to keep f roughly constant" discussion, it is formally better to write $t(1, p_{\text{ranks}})$ rather than only $t(1)$—even though almost no

write-up does so. When we compare our model to Equation (**??**), we observe that Equation (4.1) lacks a penalty parameter. Indeed, we introduce another parameter of interest here: We assume that each data point in (4.1) contains two types

$$\begin{aligned} t(p_{\text{ranks}}) &= t_{\text{useful}}(p_{\text{ranks}}) + t_{\text{overhead}}(p_{\text{ranks}}) \\ &= (C_{\text{useful}} + (1 - C_{\text{useful}}))\ t(p_{\text{ranks}}) \end{aligned} \tag{4.2}$$

of runtime.

Benchmark preparation For weak scaling, we increase the problem size as we add additional nodes. Therefore, it is important that we understand the cost function $T^{(0)}$.

Assessment crime 12 *We scale up the problem such that the individual work per nodes becomes very small.*

As a rule of thumb, we can simply track the memory footprint per rank and ensure that it stays within a certain regime such as 60% of the available memory per node.

Data collection The data collection is very similar to the strong scaling studies in Section 4.3: We track the cost of the computation (time-to-solution) over multiple nodes for each problem size. For the bigger setups, we won't be able to evaluate the problem on only one node, but that's, in general, not necessary. If we have a problem of size N on one node, a problem of size $2N$ does only have to be evaluated on two nodes. There is no need to run this one on two nodes as well.

t.b.d.

This is where the strong scaling curves (Section 4.3) plateau or even start to rise again over the core count.

4.5 GPU performance

t.b.d.

4.6 I/O performance

t.b.d.

4.7 Localisation

So far, we have assessed our code as one big black box.

- We did not ask *which part of the code* causes a problem.
- We did not ask *when* it arises throughout the execution.

- We did not ask *where*, i.e. on which rank (if we run over multiple nodes) a problem arises.
- We distinguished, to some degree, *what* we want to study in our code, i.e. which quantity of interest.

Before we continue to assess the code and dive deeper into details, it makes sense to gather the information from the first three bullet points above or at least to discuss various approaches for gathering said information. Hereby, we must consider the type of flaw we are investigating: Single-core flaws imply that the "where" question does not play a significant role. Inter-node flaws might suggest that the "where" question is the primary concern to address.

Finally, the most frequent quantity of interest (metric) will be time spent in routines. In this case, the routines consuming most time are also called *hotspots* of a code. However, there are many other metrics that affect runtime, and we might want to study them separately—even though all of them eventually affect the time-to-solution[1]. Not all tools can provide answers for all metrics. We therefore must carefully select our repertoire of tools.

How it works

- ⊞ Once you have a first high-level overview, run a profiler to confirm that effect is either localised or uniformly found over all compute units.
- ⊞ If the flaw is found on few units only or only for a certain compute time or within one function which is not responsible for the majority of the runtime, study the flaw for this program phase, compute unit, function separatedly from all others.
- ⊞ Upon your very first profiling, ask a code expert which code parts (functions) actually perform calculations (cmp. Section 3.7).

How it does not work

- ⊟ Jump straight into some traces without spending time to think what metrics and program parts are relevant to answer your hypotheses regarding performance flaws.
- ⊟ Draw conclusions from global observations on all code parts and compute resources (cores or ranks).

[1] Some codes are interested in other metrics such as energy, but they are still rare.

Chapter 5

Summary and outlook

This document is still early into its preparation. However, it demonstrates some key points around our main focus of building a performance analysis methodology that seeks to avoid jumping straight for a performance tool. Instead we want a methodology that starts high-level and only reaches for tool, when appropriate. Rather than being lead by the tool, we want to be lead *towards* a tool by the methodology.

5.1 Recap

Though this iteration of the document is just a brief introduction, it covers some of the necessities for establishing a rigorous performance assessment service.

We believe that beginning with an overview of terminology in Chapter 2, including the fundamentals of HPC, is a vital starting point. Many users and developers come to HPC without a technical background in computing, and it is important now more than ever to set out an agreed set of terms for this performance analysis methodology. Beyond acting as a glossary of HPC and performance terms, Chapter 2 also establishes some of the key concepts of how we are to conduct a performance assessment such as treating a code as a blackbox and noting measurements of runtime performance objectively, without jumping quickly to deductions.

Preparation for a performance assessment is key to reproducibility. Chapter 3 lays out some details of our 'experimental reproducibility', i.e. before carrying out a performance assessment, we need to establish the fundamentals of setting up our benchmark. Once the benchmark example has been agreed upon, setting up the compiler and other variables such as I/O, etc. is a crucial part of how to make sure that varying, e.g. the number of cores gives reliable data. It should be noted that within this preparation, we may also see performance gains in tuning compiler options. This preparation stage is a key point of the performance assessment in which engagement with the service user is crucial. In order to get a useful benchmark example which is indicative of a real simulation run, the domain specialist needs to set this up.

This current document finishes with a high-level approach to an assessment in Chapter 4. One reason for this is we find often without this initial high-level approach,

HPC users will immediately jump to a tool which can give a miriad of data and metrics. This sometimes sends an HPC user down a wrong path as they may not be following a structure methodology to rigorously analyse performance; or, we have even found that some people are put off by the overwhelming data and the user gives up on the analysis as they get a sensation of: "where do I begin?".

The high-level approach can be a stepping stone into a more detailed analysis as it simply breaks down performance into five topics: core, intra-node, inter-node, GPU, I/O. For some tool developers, the tools direct the user towards an almost 'Matryoshka doll' model of performance, i.e. core performance sits within intra-node performance which sits within inter-node performance. We believe this to be an over simplification, and in reality, these performance topics can never be truly de-coupled from one another. The high-level approach presented here is not seeking to focus the analyst down one particular avenue or another, it simply gives an overview of the performance landscape for a particular code.

5.2 Outlook

Having established that this document is not the finished work, it is worthwhile detailing what is to come in future versions. In particular: 1. what is left out of the current methodology; 2. what is to be discussed about the implementation of this methodology into a performance assessment service.

Firstly, performance topic decision trees. We believe there is value in beginning with the high-level performance assessment, and we indeed recommend beginning with this for any performance assessment. However, this is by definition not a complete analysis in and of itself. Through a series of performance analysis workshops we have begun to design a series of decision trees.

Each performance topic will have a decision tree, i.e. having conducted the high-level assessment the analyst will have to make a decision as to which topic to begin with and the decision trees will direct the analyst through a series of binary outcomes to isolate individual performance issues. For example, for intra-node performance we can begin by asking if the code is dominantly parallelised, if not then our performance engineering should be directed towards increasing the scope of parallelisation (though again we note that said engineering falls outside the scope of this work). If there is significant parallelism, then we can ask if there is good load-balancing. If there is good load balance then we can explore synchronisation, if not we turn to checking if there are enough tasks, etc. as we work down the tree. We plan to have decision trees for each performance topic with good progress made on core, intra-node and inter-node with our focus next turning to GPU and I/O.

Secondly, once we have introduced the decision trees we can describe our entire performance assessment methodology including case-studies. As we work through the creation of this manuscript, we are employing this methodology on existing codes whilst also engaging with other users/developers through performance analysis workshops. We include points on building the performance analysis results into a finished report alongside these case studies.

Lastly, we will give details as to the implementation of the assessment service from

an organisational stand point. Beyond setting out the assessment methodology, there are also a number of points around how do you actually build a functional assessment service from the interactions with scientists and developers through to the interface of the service.

One component of the service interface is planned to be a reproducibility analysis, i.e. if an HPC developer is deciding to engage with a performance assessment then it is worth auditing the reproducibility of their software. For example, is the code documented, using version control, implementing CI/CD pipeline, etc. A performance assessment can mean a developer is already giving their time to stop and reflect on their code and so introducing some time to also implement some good software practices can have two effects: 1. it helps the sustainability of their software going forward; 2. it helps the job of the analyst if the building and running of the software is well documented and hosted on a well maintained repository.

In summary, this document lays out just beginning of delivering a performance assessment, and the future iteration of this document will expand upon this methodology whilst also discussing the infrastructure of how to implement this methodology as a functional performance assessment service.

Part III

Appendix

Appendix A

Acknowledgements

A.1 Funding councils and supporting initiatives

This work was supported by the Science and Technology Facilities Council [grant number UKRI/ST/B000293/1]. This work also was supported by the Engineering and Physical Sciences Research Council [grant number UKRI1801]. The underlying projects HAI-End and SHAREing (Skills Hub for Accelerated Research Environments Inspiring the Next Generation) is part of the cross-UKRI Digital Research Infrastructure initiative.

A.2 Partners

We thank the companies AMD, Intel and Linaro for their continuing support of these activities by sending delegates to our workshops and contributing to our document.

Durham University is a proud partner of VI-HPS (https://www.vi-hps.org/) and appreciates the input of our colleagues and friends of the research consortium. Without their input, none of this would have been possible.

Bibliography

[1] Kent. Beck. *Extreme programming explained : embrace change*. Addison-Wesley, Reading, Mass, 1999.

[2] John D McCalpin et al. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, 2(19-25), 1995.

[3] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

Index