A novel performance analysis workshop series concept, developed at Durham University under the umbrella of the ExCALIBUR programme

Alastair Basden Marion Weinzierl Tobias Weinzierl * Brian J. N. Wylie

August 2, 2021

Abstract

From January 2021 until July 2021, a group of members of the POP consortium, DiRAC, the N8 CIR, Durham University's Institute for Computational Cosmology and members of Computer Science organised a series of seven workshops around performance analysis and optimisation. The series covered a variety of tools, featured different invited speakers, and addressed inter-node, intra-node and core-level runtime efficiency. In contrast to established workshops that usually run en bloc over multiple days with individual participants, we organised a **series of one-day workshops** and invited **teams of participants**. Each team consisted of multiple (3–7) scientific software developers working together on one code base. This report summarises the workshops as well as feedback gathered at the end and throughout the series. It discusses interesting lessons learned w.r.t. the status quo of the codes/teams that signed up for the series, it summarises impressions regarding the training status and needs of the participants, and it relates these take-aways to current discussions around the training of high-performance computing application developers in the UK.

Disclaimer: All opinions and impressions presented here are those of the authors. They might not reflect the opinions of the workshop participants. While some of our statements are supported by quantiative data from our participant questionnaires, many statements are anecdotal and biased.

1 Introduction

Build-measure-learn [Rie11] is a sequence of three activities that we find in many scientific code development processes. We build software, we measure its behaviour, we we learn from these data. The lessons learned yield scientific insight and feed into subsequent development iterations. Unfortunately, build-measure-learn also recounts the story of a triad of fundamental problems that we face on the road to exascale: (i) Teams build software with features, i.e. functionality in mind, and then are surprised by its non-functional properties, i.e. performance. Good performance consequently has to be recovered in subsequent releases while the functional software core remains in place. (ii) Teams struggle to measure performance data. It is easy to time programs, program phases or even algorithmic steps, and it is relatively straightforward to build home-grown tools that illustrate runtime behaviour (and often illustrate what the builders want them to show). It is hard to relate runtime behaviour to the system software and the hardware, i.e. to explain how and why machine properties cause certain behaviour. (iii) Teams find it hard to interpret performance data, as performance data typically is voluminous, is noisy, is unstructured, is distributed. While some fundamental and revolutionary changes to the software development processes might be necessary to master these challenges—in particular a performance-first and concurrency-first engineering approach seems to be promising—performance analysis and performance analysis tools promise to ride to our rescue on the support tool side.

ExCALIBUR (The Exascale Computing ALgorithms & Infrastructures Benefiting UK Research) is the UK's UKRI Strategic Priorities Fund [Web21a] to prepare codes for exascale. The programme [Web21b, Web21c] is co-delivered by the Met Office on behalf of PSREs and EPSRC on behalf of UKRI partners, NERC, MRC and STFC. One out of its four programmatic pillars is the investment into

*Corresponding author; email:tobias.weinzierl@durham.ac.uk

people: ExCALIBUR wants to contribute towards the "interdisciplinary Research Software Engineer (RSE) career development driven by forward-looking scientific software design" [Web21c, Overview page]. As we consider performance analysis skills to be essential to deliver forward-looking scientific software design in High-Performance Computing (HPC), the Design and Development Working Group (DDWG) ExaClaw [Web21d] built a consortium with members from the N8 Centre of Excellence for Computationally Intensive Research (N8 CIR) [Web21e], Durham's ExCALIBUR Hardware & Enabling Software project [Bas21] and the VI-HPS [Web21f] and d,,id,[Web21g]d,,id,,id,[Web21g]Performance Optimisation and Productivity Centre of Excellence in HPC (POP) [Web21g], to run a series of workshops around performance analysis techniques and tools.

Five constraints shaped our workshop series design: (i) A wide variety of different tools has to be covered; (ii) performance tools can be laborious to install and trial out; (iii) an in-depth understanding of how to use the tools and what they (can) present is essential to benefit from them; (iv) an ExCALIBUR-funded workshop has to make direct contributions to potential exascale software; and (v) the potential participants are all immersed in (other) full time day jobs. Therefore, we decided to run a whole workshop series where multiple tools are covered, and we decided to make each workshop session comprise of a tutorial and lecture element plus a hands-on, hackathon-style element where participants apply acquired skills to codes they have to work with anyway. Durham reserved a local supercomputer and provided high priority access to all workshop participants over the seven months, while we ensured that all presented tools are available and ready-to-use on this cluster. Furthermore, the tool developers themselves were asked to present their performance analysis tools. We had the expert knowledge in the classroom. Different to existing workshops, our workshop did not call for individual participants. Instead, we asked for whole teams to sign up. Each team was encouraged to bring one dedicated research code with them. As there is no one-fits-all performance analysis tool, our intention was that each team benefits at least from a few tools and that research codes improve over the course of the workshop series. Finally, we decided to schedule roughly one day per month for our workshop such that participants can fit the workshop into their calendars. Collaboration tools (slack and email) ensured that participants remained engaged and provided the opportunity to follow-up with the tool developers, while continuous cluster access gave participants the opportunity to digest and deepen their understanding of the tools between the sessions.

The remainder of this document gives an overview of the workshop organisation and topics (Section 2) before we provide a high-level overview of the participants, their codes and their technology backbone (Section 3). In Section 4, we discuss the most dominant themes of the participants' feedback, before we summarise our own impressions and lessons learned (Section 5). A brief outlook closes the discussion.

2 Workshops organisation

We organised a series of seven workshops all committing to the same structure.

2.1 Format

We had a brief *kick off* (around 30 minutes) where two volunteering team demonstrated how the previous workshop has helped them with understanding their code, and which challenges they faced. In the first session, this kick off spanned two hours and allowed each team to present their code base, challenges and expectations. After the kick-off, the morning sessions hosted *tutorials*, i.e. lectures and brief hands-on sessions with prepared examples. In the afternoon, all teams had two up to four hours time in breakout rooms to try out the tools with their respective research code. There was no formal taught content, but the lecturers walked around (virtually) and were available for hands-on support. These *hands-on sessions* allowed the participants to transfer new skills and insight to their respective domain or code of interest. In three workshop sessions, we also offered optional 20–40 minute presentations about upcoming new features within the OpenMP standard, or some revision classes around topics like MPI progression. The seven workshop sessions were followed by a final wrap-up meeting, where the participants first reported on their lessons learned, then were asked to give in-depth feedback (Appendix D), and finally attended a brief wrap-up where we presented follow-up training and access to further (experimental) machines.

Table 1: Overview of workshop sessions. The detailed program as well as video recordings are available from [Web21h].

Date	Topic	Tools	Lecturers (lead)
21/1/2021	Introduction	Intel performance snapshot, ARM performance reports	Wylie (Jülich), T. Weinzierl (Durham)
18/2/2021	Parallel profiling	Score-P, CUBE	Schlütter & Wylie (Jülich)
13/3/2021	Execution trace visualisation	Vampir	Williams (Dresden) Wylie (Jülich)
8/4/2021	Automated trace analysis	Scalasca	Geimer (Jülich)
20/5/2021	Correctness checking MPI, multithreading and memory	Archer, MUST, valgrind, LLVM extensions	Protze (RWTH Aachen)
24/6/2021	Node/loop analysis	MAQAO	Valensi (Versailles-Saint-Quentin-en-Yvelines)
15/7/2021	Single core analysis	Likwid	Hager (Erlangen)

2.2 Content

The choice of topics d,,id,1)d,,id,,id,1)ranged from multi-node analysis to single core and loop analysis (Table 1): We kicked off with general performance overview analyses, and then discussed multi-node profiling, tracing (two sessions), (parallel) correctness, and single core behaviour. There are two directions of travel in the area of performance analysis tools that we decided to leave out: We skipped the usage of GPUs, and we omitted (apart from the first session) commercial tools.

The decision against GPUs resulted from the assumption that the majority of large-scale science codes still have no GPU support yet. We assumed that is is too early to discuss GPU analysis. At the same time, open source performance analysis tools just start to catch up with GPUs and to support the accelerators. Despite recent changes in the market, a discussion of GPU machines would likely have been NVIDIA-centric, whereas all other tools had been rather generic and applicable to a wide variety of architectures. The decision against GPUs had to be validated and reviewed throughout the course and its evaluation.

We decided to discuss (primarily) free tools. There are four reasons for this decision: First, Europe is in the lucky position to have a well-established workshop/tutorial ecosystem around European performance analysis tools with the POP Centre of Excellence. Colleagues from this centre were happy to buy into our concept and to commit to give lectures and to attend the workshops and to provide support. Second, we wanted to discuss a set of tools that are not focused on a particular vendor and can, in theory, be installed on the participants' home system if they find the tools beneficial for a broader audience at home. Third, we considered it to be essential to have the tool developers themselves giving the presentations, as this gives the participants the opportunity to provide feedback about missing and new features. It also makes it easier to dive into code, tool and machine details. In several cases, we even had the opportunity to try out pre-release features of new tool releases.

d,,id,B. We continue with a brief overview of the participants, before we focus on the feedback, lessons learned and our evaluation. d,,id,,id,B. We continue with a brief overview of the participants, before we focus on the feedback, lessons learned and our evaluation. We brief overview of the individual tools can be found in Appendix B. We continue with a brief overview of the participants, before we focus on the feedback, lessons learned and our evaluation.

3 Participants

With 79 registrations, the uptake of our workshop announcement was very good. The first sessions kicked off with more than 50 participants, with the number of attendants shrinking over the seven months to 20+ people who actively participated in the lectures and workshops. The 79 original registrations formed 13 teams, i.e. were grouped around 13 research codes, with each team hosting between three and seven participants in practice. Prior to our first session, we ran a questionnaire (Appendix C) to identify the background and needs of our audience.

We found that C/C++ was the predominant language—we had eight teams using C++ and seven C, though teams were allowed to tick both and the data thus might exaggerate the dominance of the C language family—while Fortran also is popular among our audience (seven teams). In contrast to the popular perception, Python was not dominant among the HPC codes at all, and only one team actually used Python as core programming platform. Three more used Python for preparatory code generation and for pre- and postprocessing.

Among our teams, the multicore era has not begun yet for all. Only a minority of six teams used OpenMP, three had support for native pthreads. Again, multiple technologies could be mentioned by every single team. We conclude that more than half of the projects rely only on MPI to parallelise. Even more interesting, more projects had plans to add GPU support than we had teams interested in adding multicore parallelism. This suggests that several teams plan to skip the multicore step and instead migrate to accelerators straightaway. However, no team said that they have GPUs in production use on a daily base.

Around half of the participating teams had experience with tool-based debugging, automated testing and the usage of performance tools. This is surprising given that we had been asking for mature codes which are used in production or extreme-scale scenarios.

Half of our participants claimed to have a good understanding how their codes perform and where bottlenecks are. The minority of the teams found themselves currently in a performance crisis, i.e. wanted better performance for better science. Instead, most teams signed up as they wanted to acquire better skills to be prepared for new machines to come. Few did already suffer from an increase in hardware concurrency on current systems. Both statements are in line with the observation that multicore programming is not high up on the priority list, but teams are interested in GPU ports and support. When the participants had been asked where they see major showstoppers that cause frequent problems, a majority mentioned that they suffer from a large memory footprint. Insufficient memory per node or excessive memory requirements are a hot topic.

The participating teams' codes span a wide variety of science domains: biology, particle physics, astrophysics, engineering, etc. If there is a predominant application domain then it would be physics with a slight bias towards astrophysics and wave propagation and phenomena. On the technology side, we saw lots of different algorithmic and numerical approaches: stencil codes, unstructured meshes with DG, Lattice Boltzmann, Lattice QCD, and Monte Carlo-based algorithms. No team worked on core Linear Algebra.

Three teams reported that their code of interest has framework character, i.e. is a generic solution to a whole problem class. The remaining groups all had one particular application problem in front of them. Only one team declared that they can work with a miniapp throughout the hands-on sessions. The remaining teams all worked with a "full-blown" simulation code base.

4 Participant feedback and instructor impressions

We gathered three types of feedback: Throughout the seven sessions, the coordinators aka authors of this report gathered anecdotal remarks and exchanged impressions and lessons "spotted" in a recap session after each workshop. In a dedicated feedback session after the last workshop , we collected quantitative feedback on a set of questions which naturally arose from the workshops and the lecturers' impressions. Finally, we gave participants the opportunity to share non-quantitative feedback publicly, or to send it confidentially directly to the organisers. All data here is anonymized.

4.1 Quantitative final feedback

The quantitative final feedback was gathered via Mentimeter in an interactive session, i.e. we talked the participants through each question. Participants who did not participate in the final feedback session were sent the evaluation questionnaire afterwards. Detailed data can be found in Appendix D.

The quantitative data is delicate to analyse, as we lack comparison data or a historical track record. We however see that we had a strong core participant group, which is a subset of the people giving feedback and makes up the bullk of 20+ participants that we saw in each individual session. For our type of workshop format, we conclude that an attendance of 25% of the number of registrations is a realistic estimate though around 2/3 of the registrations might actually show up for one or the other session—in particular the first one. We have no further quantitative information on particular

reasons why people that signed up decided to stay away. Time and a lack of commitment to an all-virtual workshop are likely reasons. Two participants mentioned that they recognised after the first workshop that their background in programming—both were Python users only—and computer science is insufficient. Two teams dropped out in the middle of the series, as they felt that they were unable to use the tools for their particular software; either due to missing tool capabilities or a lack of expertise with or usability of their own code base. We follow up on both points below.

We received a very positive feedback about our workshop format, and attendees where happy with both the length of the workshop and the interval of the workshop sessions. Seven out of 24 participants could apply the lessons learned directly to an ExCALIBUR code, further seven said they planned to apply it to an ExCALIBUR code. The majority felt that it was first of all themself as an individual who benefited from the workshop, with their team or institution at second place. Few thought the (ExCALIBUR) PI who eventually is in charge to promote the code benefits from the skills acquired. The workshop played had a strong personal development role, participants clearly saw a pathway why their home institutions benefits from the skills acquired, but the direct link between performance analysis and the research agenda behind a (DDWG ExCALIBUR) code evolution was less clear to the participants.

While the workshop organisers were extremely happy about the experience reports presented by the teams—acted both as feedback mechanism and as icebreaker—the feedback of the participants on this element of the programme was mixed. Some rated it as very valuable, others gave lower ratings. At the same time, participants repeatedly asked for performance analysis success stories. We conclude that the experience reports only partially fulfilled the expectations of what participants consider to be a "success story". This could be due to the lack of success, due to diversity of the codes which make it harder to transfer presented insight to our own codes, or due to the missing link from performance insight to code evolution. While the tool demonstrations received positive feedback, also the theory and programming recap was found valuable. This was picked up in more detail in the formative feedback.

Most quantitative feedback has been on the positive side. There were four outliers:

1. The insight generated via high level overview tools was considered to be too shallow by the participants;
2. a significant number of participants struggled with the complexity of Scalasca/Score-P (though others found it straightforward) and the complexity of the data presented;
3. MUST and the other correctness checkers did not yield particular insights for our participants;
4. The perception of MAQAO with its loop-level analysis split the participant group into two worlds—either enthusiastic or skeptical.

All four outliers have to be contextualised through formative feedback. Non-quantitative further comments allow us to understand them better and to see that they address result from fundamental misconceptions and problems. They do not downrate particular tools in the first place. The same argument has to be made for the mixed perception of the experience reports. Our formative feedback helps to understand which elements the success stories did lack from a participants' point of view. Again, they suggest there are structural and organisational reasons for this lack:

4.2 Formative feedback

The formative feedback at the end of the course can be grouped into the following topics:

- *Code maturity.* All teams considered their own codes to be sufficiently mature. For our performance analysis workshop, this turned out to be a double-edged sword: The participants appreciated the potential of correctness checking tools (MUST, Archer, correctness validation within the Clang compiler family), but the direct impact of the hands-on session with these tools was limited. The perception also had been challenged by the fact that some analysis tools required the users to migrate to particular toolchains (Clang, particular MPI versions). Not all the codes of the teams were portable and could manage a switch.

- *Code size.* The majority of our participants brought real-world scientific codes with them, i.e. codes which are used for production runs to obtain new scientific insight through supercomputing. All participants appreciated the opportunity to obtain a high-level performance characterisation through tools like the Intel Performance Snapshots. However, for a large-scale code with many different compute phases, an averaged total over the entire execution is of limited use. Furthermore, the participants reported that they suffer from the fact that large-scale simulation runs tend to require a significant startup time (see next item) and changes cannot be made "quickly" but often require in-depth code understanding, significant refactoring, and apply to many code locations. Consequently, quick analysis turnarounds are made difficult.

- *Temporal and spatial filters.* Almost all tools offer filter files or APIs to control what events and time intervals are included in measurements. These are valuable and allowed the participants to eliminate irrelevant data (e.g. from the aforementioned startup phases). However, inserting filter instructions into the source code is laborious and intrusive. Tailoring a filter file manually and letting a tool instrument the code also remains time consuming. Several participants thus asked for a sliding time window approach, where they can delay data gathering or can study data over certain time windows—even though it might be aggregated over these windows. Even if fully supported, filtering was perceived to be a major showstopper: incremental filter refinement, e.g., is great for codes where one has a clear goal. For codes where the performance bottlenecks are unknown, it is tedious—in particular if a user is not fully familiar with all aspects of the code base.

- *Hybrid programming models.* Some tools struggled to support the `MPI_THREAD_MULTIPLE` mode where multiple threads concurrently do MPI operations. This was frustrating for the affected participants. d,,id,[id=BW]insufficient multithreading support.d,,id,,id,[id=BW]insufficient multithreading support.For two teams, a lack of this MPI mode became a showstopper and they eventually gave up on tools with insufficient multithreading support.

- *Runtime focus.* Right from the start, participants emphasised that the focus on runtime characteristics (MPI wait times, inefficient CPU usage, memory bus congestion, ...) is important yet not the most pressing issue for some of the bigger codes. They suffer from severe memory usage constraints. As a consequence, it is not possible to discuss better load balancing without a knowledge of the memory demand including a break-down over ranks and program phases. Some participants had hoped for a more memory-centric tool support.

- *Code sophistication.* Several teams ran into issues if they used generated code, highly templated code, or sophisticated iterators. These modern programming paradigms often are poorly supported by tools, or tools struggle to map executed instructions back to source code locations. Several tools relied on Python as front- or back-end. A lot of tools already provide Python support. Still, projects struggled whenever Python was not used a pre- or post-processing step, but man-in-the-middle yielding C output, e.g., or invoking MPI calls.

- *Lack of theory and knowledge transfer.* Many participants reported a lack of formal theory or programming background, and many participants also asked for more examples how to translate performance analysis insight into better code. There is a need for more methodology education and also formal computer science education (to understand the behaviour of MPI codes, people require a certain understanding of MPI communication modes; to understand MFlop rates, people need a certain understanding of chip architectures); notably as many developers of scientific software do not have a degree in computer science, or the curriculum did not cover these topics.

4.3 Further (anecdotal) continuous feedback

A lot of the feedback that we gathered throughout the sessions was eventually repeated at the final wrap-up. Besides the wish for a even broader spectrum of tools and a coverage of GPUs, participants raised a few additional, particular interesting issues:

1. The participants appreciated the availability of a dedicated workshop cluster with a well-defined and prepared toolchain. They however observed that a lot of tools prefer one particular toolchain and it is not always clear that this toolchain is available on a normal cluster with the respective

extensions, or can be installed quickly. Some codes also were (over-)sensitive to particular MPI versions, e.g. (see remark above). As system administrators and tool developers cannot foresee any potential required combination, teams ended up rebuilding Python and compiling MPI from scratch—certainly not an ideal situation for the standard user or performance analyst.

2. A lot of participants observed that runtime data can become noisy. This raised the question to which degree it is reasonable to define small, short-running benchmarks or to write miniapps.

3. Several teams identified severe issues early throughout the workshop series such as dysfunctional load balancing. As the teams struggled to fix these issues on time, the benefit from follow-up sessions was limited: One team, for example, saw limited sense in tuning the loop level parallelism. Their load imbalance was so severe that any investment and investigation into further runtime aspects was clear not to yield a significant (and, relative to the work required, reasonable) payoff.

4. One team articulated that they struggle to implement performance improvements, as functional extensions always are given higher priority by the decision makers of the underlying project, while several teams could, at some point, not move forward with the necessary changes as they were not the code owners or main code developers. This meant that they did not know the underlying code or the application domain in enough detail, or simply could not to crawl through 1000s of source files.

5. Participants were asked to bring along setups which complete within a short time frame such that data produced by the tools remains manageable and there are reasonable turn-around times. This implied that groups had to "cook down" setups that they usually tackle with their codes. However, alterations such as a change of the problem size have the potential to skew a program's runtime characteristics (or even make it crash), if one does not know all settings necessary to consider. The change of fundamental equations (such as adding additional physics) can make an originally fast code perform badly.

Item 3–5 were particular severe in cases where RSE efforts are bought in from central university Research Software Engineering teams "to fix these performance flaws" or if a group extends a code which was originally written by someone else and was tailored to another problem domain. . The issues were amplified by situations where there was no significant commitment by the "sponsor" or original developer to interact and collaborate with the performance analysis team (one participant articulated frustration that they were required to re-engineer and re-construct what a particular code does rather than to actively improve its performance), were there was no opportunity, permission and funding to change a code base once flaws are identified, or where code owners do not consider it to be critical to deliver code which is portable over a range of toolchains. A reason for this overall constellation could be either of these two, or both: (i) The performance tools do generally not work on a black box code, but require in-depth knowledge of the analysed code base. (ii) The HPC codes that were analysed do not work as a black box.

All lecturers had a strong computer science background and decades of experience with HPC systems and HPC software. Despite the fact that all teams worked with large-scale HPC software, it became clear that some terminology that a computer scientist takes for granted were unfamiliar to the audience. Questions throughout the workshops ranged from "May I ask what a NUMA domain is?" over "what is strong or weak scaling" to "I've never heard of an eager send". All our participants worked in the area of high performance software engineering. We observe that this does not automatically imply that they have had a formalised HPC training – in fact, more than one participant thanked us for "finally explaining what these terms meant that everyone just uses without explanation".

4.4 Three success stories

Even though the majority of the participants said they were lacking time (as well as recipes and methods) to translate the analysis insights into better code, there were outstanding success stories, too. Three of them shall be sketched:

One team brought along a well-balanced simulation code which, however, had poor peak performance even though the underlying algorithm was floating-point heavy. The Scalasca/Vampir analyses eventually identified that the code performance degraded because of a lot of small MPI messages. The

original authors had written their code base with maximum overlap and asynchronicity in mind. It turned out that the overhead per message in MPI made these efforts counterproductive. After consolidating (packing) multiple messages into fewer, larger messages, and after replacing many individual wait instructions with few `MPI_Waitall`s, the workshop participants obtained significantly faster code.

A second team rediscovered their old, homemade load balancing analysis tool after the discussed tools had flagged some severe load balancing issues. Through a combination of these bespoke tools with the discussed ones, the participants found a flaw in their optimisation strategy: They had previously used profilers and analyses to guide them to improve their code performance. Unfortunately, the code's runtime characteristics changed once the problem was strongly upscaled. Tweaks that payed off for smaller setups suddenly became counterproductive. A rollback of these ill-considered optimisations made the code run faster for more realistic, bigger setups. Obviously, these improvements and rollbacks are problematic for a code where most users run small- and medium-scale simulations, while funding agencies want to see large-scale runs.

The third and final code suffered from a very expensive data initialisation phase dominated by calls to a third-party library. An in-depth analysis with MAQAO identified a few loops as hotspots. These loops and the nested function calls had been implemented efficiently following the Numerical Recipes [PTVF07]. Unfortunately, the respective Chebychev polynomials yield good runtimes if the evaluation points change. In the present code, the evaluation points however remained invariant over different polynomials. The wrong optimisation had been applied. A loop reordering and simplification facilitated an 11x speedup. The prohibitively long initialisation phase was eliminated and facilitated more in-depth analyses of further compute routines.

5 Evaluation

5.1 Lessons learned

Take away 1 *The workshop format has been a great success.*

Due to the fact that we wanted to reach the whole UK community (as well as projects from all over the EU) and primarily targeted Research Software Engineers, i.e. colleagues whose day-to-day job is the development and improvement of (various) software projects, we spread out the workshops over multiple days with roughly one month between each session. We also did not advertise the course as training for individuals, but asked whole teams to participate. These teams were asked to bring one piece of functional parallel software with them. Each workshop day consisted of a tutorial-style morning and a hands-on afternoon, where tool developers and experts mentored and advised the teams.

The intervals in-between the workshops allowed the participants to digest and try out the features of the tools in great detail. This was a key requirement for having participants who can report on success stories throughout the workshop series. As the sessions were spread out, it was possible for attendants to squeeze them into their own schedule. Timetabling otherwise quickly becomes challenging for full-time RSEs serving many projects. The team-plus-code concept ensured that the afternoon sessions were successful: We assigned the teams individual breakout rooms and the instructors visited and consulted them in these rooms. As a whole team collaborated on each code, we saw some real team spirit evolving, and we avoided the situation where participants get isolated and detached; with a team, the team often discusses and solves problems proactively, and the hurdle to ask questions is reduced. The tool developers thus met an enthusiastic crowd and constructive discussions resulted. It was also extremely valuable that many tool developers agreed to sign up to our Slack workspace and hence had been able to follow-up after the actual workshop.

Our expectation that success stories of the participants serve as icebreakers and create additional momentum has not been fully met. Participants gave mixed feedback about the experience reports of other participants. We think that the idea of an experience report in principle is good, however lacks to unfold its full potential if participants cannot translate performance insight into refactoring (see comment on programming).

We also acknowledge that the team approach runs risk to exclude individual RSEs who do not work in a larger team as well as embedded RSEs who are the sole main developer of a code base.

Take away 2 *The workshop scope was too wide.*

We organised a tour-de-force through MPI, shared memory and single node optimisation. Our (naive) assumption had been that this triad is of relevance for all (ExCALIBUR) codes, and that a holistic performance analysis of state-of-the-art simulation codes requires skills in all of these areas.

Only few codes however use all three levels of parallelism, and most participants were interested in one particular aspect. It might thus be better to split the workshop series into a series of series and to focus on one type of parallelism per mini-workshop. Code correctness however does not fit into this set of contents (see comments below). Distinguishing workshop sets by parallelisation level also ensures that the participants get most out of it, i.e. it accepts that a lack of load balancing on the MPI side for example can become a real showstopper for single core tuning for some teams. In such a case, it would be better to give the teams enough time to fix the load balancing first before they return to core optimisation.

Take away 3 *The workshop scope was too narrow.*

There is a need for similar workshops around GPUs, and potentially about tools for other programming languages such as Python or Julia. At the moment, a focus on C/C++ and Fortran only is appropriate as these languages are predominant in HPC. Our questionnaire also suggests that GPUs are not widely used even though they are omnipresent on development roadmaps, PR material and grant applications. This is likely to change over the upcoming years.

Take away 4 *Few tool extensions would improve the usability and participants' productivity greatly.*

Throughout our feedback sessions, four major areas materialised where the participants felt left out by the tools discussed. Obviously, some tools already support these areas, others not. First, users suffered from a lack of support for `MPI_THREAD_MULTIPLE`. Second, the support of a Python integration and on-the-fly code generation becomes more important and has to be supported by the performance analysis toolchain. Third, memory footprint tracking and measuring play an important role, and, finally, temporal context is important.

The four items are not a generic wishlist. We are now in a position to explain where these requirements come from: With the omnipresence of multicore architectures, more codes will allow multiple threads to make MPI calls. It is not possible or straightforward to rewrite software such that only one thread makes calls. The participants appreciated that adding support for multithreaded MPI on the tool side is not trivial. Yet, they found it not acceptable that an analysis tool constrains how a code has to be written.

Support for on-the-fly code generation and the embedding into Python-esque tool landscapes becomes more important as projects mature. This notably is a direct result from the popularity of the approach to generate bespoke, highly optimised compute kernels for codes, as well as the rise of domain-specific languages. Both trends anticipate the need for a higher developer productivity, the increasing complexity of hardware, and the ambition to deliver performance-portable code.

Memory requirements serve as a natural constraint for many code changes, and it was interesting to see that memory is already a major issue when we run large-scale calculations. It is difficult to ask participants to cook up better load balancing without providing them with insight into the actual memory footprints tied to such refactorings.

Finally, all participants suffered from initialisation or burn-in phases with their "realistic" codes (cmp. remark on miniapps). It became clear that tools that provide a holistic overview of the runtime behaviour would be even more advantageous if they split up this information into windows/slices over time. The participants' codes were non-trivial with different physics involved and different program phases. Once the attendees commit to a particular experimental setup, they have a good feeling about the code behaviour throughout the run. It is therefore natural that they want to be able to tell a tool that they are interested only in certain time windows. Asking them to use a marker API, e.g., instead is not user-friendly and sometimes infeasible as it requires additional synchronisation, message exchange, or thread communication.

Overall, our participants met the tools with great enthusiasm and were impressed by the tool capabilities as well as usability. While the feature request list from above would improve the tools, the major showstoppers we identified are not tied to the tools but result from organisational and educational constraints:

5.2 Open questions and controversial impressions

Take away 5 *Individual participants and those who did not belong to the core developer team of the code(s) they worked with (for example RSEs that were bought in by a research team to "just make this code run faster or tell us where the bottlenecks are") struggled to get most out of the present course format.*

To understand how performance analysis tools work and what they can do, it is beneficial if participants are familiar with the internals of the used code base, and it is beneficial to work in a team. No individual ("isolated") participant that had signed up completed the full workshop series. Where teams used codes by other groups and the corresponding core developers were not members of the team, they had a hard time and had to jump over many hurdles—too many design decisions and internals tend to be unknown. Where teams consisted of "techies" and domain specialists were absent, the teams struggled to design and alter meaningful benchmarks, and later struggled to relate (raw) performance data to scientific impact.

These observations do not come as a surprise. The organisational and procedural implications, however, are far-reaching: Performance analysis has to be a first-class activity for high-performance code development. Current software development projects still start with a functionality-first attitude: Developers first ensure that the code computes the correct thing, teams then send individuals to a performance analysis course, and after that expect that those individuals can bring the code up to speed. This development model is not sustainable. Whole teams have to be educated in performance analysis and high performance computing aspects, in the best case right from the first line of code. Buying-in performance analysis workforce requires commitment beyond grants. Many institutions are starting to offer RSE expertise as buy-in option offered by a central RSE team: Colleagues with performance analysis skills can be "bought in" into projects. While this is a valid business model, our experience shows that more than a sole monetary/budgetary commitment is required: If teams hire tool expertise, they still have to be willing to make the core developers (see first argument) join the team, too. Otherwise, they fail to exploit the complete potential of the analyst. This requires significantly more commitment of the umbrella institution or the PI, as the additional time for domain specialists adds on top of the analyst cost.

To some degree, the need to have a developer nearby indicates a lack of good software engineering practice and low software quality. However, it is not clear what a performance engineering approach or design patterns yielding code that can be analysed and tuned without domain konwledge looks like. It is not clear whether there is "an industry standard" for performance-optimisable code.

Take away 6 *There is a lack of high performance machinery and system stack education among the people using and developing HPC software.*

High performance computing is a challenging domain. Successful players have to move within the classic triad of computer science, mathematics and domain expertise. Besides the algorithmic and programming flavour of computer science, they need a fair amount of system-level, hardware and even electrical engineering knowledge on top. This whole breadth of knowledge is not covered by undergrad courses, while many tutorials have a strong hands-on, trial-and-error character.

We appreciate that HPC software development driven by people who are autodidacts in at least a couple of areas has delivered success stories over the past decades. However, bare-metal courses disappear from many (UK) undergrad courses to make space for more popular content, while we witness an increasing hardware complexity. At the same time, HPC software development has always been driven by application domains which abstract further and further from the actual system software and hardware. This implies that it becomes difficult to understand performance characteristics (see comments on meaning of the acronym NUMA). It is thus important that the HPC community either establishes more academic hardware-centric courses, or actively include electrical engineering as a fourth computational science & engineering subdiscipline.

Take away 7 *There is a lack of in-depth HPC programming training.*

Our participants failed to translate performance insight into code modifications. This is mainly due to a lack of time and commitment to other tasks (see comment on feature-first policy in projects above). However, we also witness a certain insecurity w.r.t. programming.

We came to the conclusion that there is a potential problem arising for HPC software developments: Many participants from domain disciplines and mathematics are C, C++ and Fortran autodidacts. At the same time, Python becomes omnipresent in many curricula. Despite all Python success stories in HPC, Fortran and C remain the two dominant languages. However, we slide into a situation where there is a lack of developers in these languages with an academic background: Programmers who can translate the concept of call-by-value and call-by-reference semantics onto machine behaviour such as memory movements or indirect memory addresses. Programmers who understand basic compiler optimisation, i.e. where can it be automated (and where manual optimisation thus is economical nonsense and even runs risk to harm performance portability), where does a compiler have to assume data dependencies and thus requires hints, where does a compiler produce slower code after applying an "optimisation" transformation. Programmers who can quickly integrate novel concepts such as SYCL or familiarise themselves with new pragmas. Programmers who deliver fast and correct code. The latter includes applying concepts such as unit testing. In an HPC context, it also implies that developers are familiar with tools like MUST or Archer or compiler validation flags. There seems to be a need for C/C++/Fortran courses along these lines which go beyond the creation of a sophisticated Hello World program.

Take away 8 *The multicore revolution still has not arrived in software.*

The free lunch is over [Sut05] for more than 15 years. However, OpenMP or shared memory programming have not been omnipresent in the participants' codes. For some multithreaded features (`MPI_THREAD_MULTIPLE`) it was disappointing that some tools struggle to support it, though all participants appreciated the engagement of the lecturers to apply the tools nevertheless, as well as the explanations why a support of the MPI+X paradigm with multithreaded MPI is challenging.

d,,id,[id=BW]and similarly multithreadedet al codes overall.d,,id,,id,[id=BW]and similarly multithreadedet al codes overall.Given the small number of codes using multithreaded MPI plus the low uptake of shared memory parallelism overall, a lack of support of this mode maybe is not surprising. It definitely however is not a reason for the small number of OpenMP and similarly multithreaded codes overall. The organisers of the workshop series struggle to distill a coherent story explaining the observation. Skipping genuine multicore support plus the focus on GPUs left us with three hypotheses:

1. Are the workloads in our biggest software projects so stationary that they would not benefit from the flexibility of shared memory parallelism over distributed memory? If projects struggle to employ dynamic adaptive mesh refinement, local p-adaptivity, etc., they are probably better off without dynamic load balancing and sole Bulk Synchronous Parallel (BSP).

2. Have programmers/PIs made disappointing experiences with OpenMP (or another shared memory approach) and therefore avoid it? A strategic decision not to use OpenMP might result from satisfactory MPI performance plus disappointments with OpenMP extensions. The latter often results from Amdahl's law: if few pragmas do not yield a significant speedup, a team might decide not to follow-up on shared memory parallelisation.

3. Are the software roadmap designers aware of intrinsic challenges behind accelerator programming? In the authors' opinion, multicore programming is easier than GPU programming, and it is also more promising performance-wisely unless an algorithm fits directly to an accelerator. In our experience, accelerator codes require a lot of investment and work until they can match the performance of a multicore chip or go beyond it (data transfer, kernel orchestration, loop rewrites). This makes it difficult to understand the GPU focus.

Take away 9 *The expressiveness and usefulness of miniapps has, from a performance analysis point of view, to be carefully assessed.*

d,,id,[id=BW]componentingredients. Once stripped of components or scaled down, they also lose their key characteristics. d,,id,,id,[id=BW]componentingredients. Once stripped of components or scaled down, they also lose their key characteristics. Defining benchmarks and distilling miniapps from larger codes that preserve key characteristics is challenging. The design of characteristic miniapps, reproducers or benchmarks and benchmarks is an art in itself. If we do not know the code properties that cause characteristic behaviour, i.e. if we do not know what to search for, what to focus on, what to preserve if we cook codes down, designing small benchmarks can become a frustrating trial-and-error

job. If miniapps are properly designed, their value is however: They help developers and analysts alike to digest key properties without a significant overhead, and allow us to try out code modifications with short turnaround cycles. Throughout our workshop, the question however did arise if there are codes where no proper miniapp does exist, i.e. codes that have to run on a large scale with many different component. Once stripped of components or scaled down, they also lose their key characteristics.

Take away 10 *It is almost sarcastic that a core computer-science subdiscipline (runtime behaviour analysis) does not yet benefit from the rapid advance in automatic processing of large unstructured data.*

Our participants spent significant time and effort on filtering measurement data, stripping codes down to the essential steps, and reducing execution runtimes. In most cases, this was laborious and discouraging yet successful eventually. In some cases, it turned out to be impossible within reasonable time to preserve characteristic behaviour with short-running benchmarks (see also remark that domain scientists have to be in the team).

The dilemma is surprising given that there is the assumption that a lot of filter identification and processing could be automated with modern machine learning and data analysis techniques. It is not clear why filters have to be developed to allow for pattern and flaw searching, while the filter design could be subject to supervised learning.

Take away 11 *Many participants had no resources to translate performance insight into faster code.*

This observation is a surprise given the scope and agenda behind ExCALIBUR, and it requires some deeper analysis why performance analysis, studies and characterisation are not at the heart of supercomputing code development.

5.3 ExCALIBUR's RSE training roadmap

Our performance analysis course has been organised under the umbrella of the ExCALIBUR Design & Development Working Group ExaClaw. ExCALIBUR puts a strong emphasis on (RSE) training and dissemination and has published a landscape review [P+21]. The review discusses the HPC skills available in the UK plus development needs. It is thus timely to compare our insights and take-aways to this landscape review.

Among the top four essential skills required for HPC, the review lists performance analysis skills on number two. Though given in no particular order, the first item refers to software engineering predominantly for CPU+GPUs, while item three and four mention the development of novel algorithms. The review confirms the statement that most HPC software developers on the RSE track (more than 73%) do not have a formal computer science (or mathematics) background. An awareness of compilers and and machine properties are listed among systematic performance analysis as "desirables" for an RSE working in HPC, while the report emphasises the role of linear algebra as fundamental (black-box) building block. Finally, the debugging and profiling of AI applications is listed as crucial skill, though we have not seen AI-driven codes among our participants (this might be because of our decision not to include GPU performance analysis in our course).

The report comes to the conclusion that there are five major areas where there is a skills gap in modern HPC code development through research software engineers: Parallel paradigms, data and application workflows including their optimisation, HPC and AI benchmarking, heterogeneous architecture programming, and containerisation. On the parallel paradigm side, the authors ask for "better training courses—focused on teaching by example—and linked to benchmarking and optimisation strategies and including a focus on specific system types." The benchmarking section lists the need of "learning how to look at both sides of a problem. Given a machine, how does a kernel or code perform on it. Given a set of kernels or codes, what kind of machine is best suited to them."

The landscape review thoroughly supports the need for our workshop series. Comparing the document to our experience, we however make the following observations:

- The review document seems to have a *bias towards accelerator-based programming.* Multicore programming is not prominently featured. This bias has to be challenged given the our course participants did not yet exploit the full potential of homogeneous hardware. It is not clear why software developers then should master GPUs which are more restrictive.

- The review document emphasises the importance of algorithms. While this is reasonable (and in line with PI's personal research interest), our feedback suggests that this *emphasis on algorithms and functional extensions is one reason behind the lack of a performance-first approach.* It is a valid concern that software development without a performance-first approach becomes non-sustainable.
- The document supports our observations that a lot of people working in our field would appreciate *more formal and in-depth training* both around tools (skills) but also covering core computer science topics which many PIs and workshop lecturers assume to be known already.
- The document emphasises a co-design aspect that it is important for qualified staff to recognise which algorithms work best on which architecture. Our experience adds a second dimension to this question: which algorithm and *which problem size (setup)* and which architecture are the best match?

6 Conclusion

We consider our workshop series to be a huge success. Consequently, our team currently searches for opportunities and funding to re-run and extend the workshops. The feedback received and observations suggest an evolutionary approach to improve the series, i.e. to tweak details and maybe split up/diversify the course.

d,,id,[id=TW]to identify the capabilities that are most valued or lacking functionality to be prioritised for future versions. The workshop itself also provided a valuable opportunity to observe the tools being used and difficulties which ensued. The extended workshop sessions and periods to follow-up between sessions facilitated deeper interaction and more advanced usage than typically otherwise available, while reducing overload on participants.d,,id,,id,[id=TW]to identify the capabilities that are most valued or lacking functionality to be prioritised for future versions. The workshop itself also provided a valuable opportunity to observe the tools being used and difficulties which ensued. The extended workshop sessions and periods to follow-up between sessions facilitated deeper interaction and more advanced usage than typically otherwise available, while reducing overload on participants.The developers of the presented tools appreciated the candid feedback from the workshop participants who applied the tools to their application codes, which helps to identify the capabilities that are most valued or lacking functionality to be prioritised for future versions. The workshop itself also provided a valuable opportunity to observe the tools being used and difficulties which ensued. The extended workshop sessions and periods to follow-up between sessions facilitated deeper interaction and more advanced usage than typically otherwise available, while reducing overload on participants.

Our lessons learned suggest that any future workshop activities should be paired up with training in areas that we took for granted when we designed the 2021 activities: Principles of computer architectures, C/C++ programming, performance models, etc. Such training should have a strong academic/computer science flavour but target colleagues that entered the RSE role from an application domain or mathematics.

Our experience confirms that the code-/team-centric approach is valid and reasonable. To make future events more beneficial to a broad community and to make the team idea work, we however require strong commitment by the home institutions of the participants and their projects (i.e. PIs): Domain scientists that drive the code evolution have to be encouraged to join colleagues working more on the technology/code side in one team. This implies further buy-in than just sending one person to a course. It also implies that centralised RSEs need the opportunity to work over a long time span on a project and to actively engage into the core coding; in particular it implies that black-box performance analysis along the lines "here's the code, just run it as it is, tell us the bottlenecks, and we will take it from there" is difficult to realise.

References

[Bas21] A. Basden, visited July 2021. https://www.dur.ac.uk/icc/cosma/excalibur/.

[P+21] M. Parsons et al. Excalibur research software engineer knowledge integration landscape review, 2021. https://zenodo.org/record/4986062#.YPqD8_Yo_mg.

[PTVF07] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, 2007.

[Rie11] E. Ries. *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. Vikin, 2011.

[Sut05] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[Web21a] visited July 2021. https://www.ukri.org/our-work/our-main-funds/strategic-priorities-fund.

[Web21b] visited July 2021. https://excalibur-portal.uk.

[Web21c] visited July 2021. https://excalibur.ac.uk.

[Web21d] visited July 2021. http://www.peano-framework.org/index.php/projects/exaclaw-clawpack-enabled-exahype-for-heterogeneous-hardware.

[Web21e] visited July 2021. https://n8cir.org.uk/.

[Web21f] visited July 2021. https://www.vi-hps.org.

[Web21g] visited July 2021. https://pop-coe.eu.

[Web21h] visited July 2021. https://tinyurl.com/performanceanalysis2021.

A Acknowledgements

Marion's and Tobias' work is sponsored by EPSRC under the ExCALIBUR Phase I call through the grants EP/V00154X/1 (ExaClaw) and EP/V001523/1 (Massively Parallel Particle Hydrodynamics for Engineering and Astrophysics). Both appreciate the support from ExCALIBUR's cross-cutting tasking theme (grant ESA 10 CDEL). The Exascale Computing ALgorithms & Infrastructures Benefiting UK Research (ExCALIBUR) programme is supported by the UKRI Strategic Priorities Fund. The programme is co-delivered by the Met Office on behalf of PSREs and EPSRC on behalf of UKRI partners, NERC, MRC and STFC.

The support of the N8 Centre of Excellence for Computationally Intensive Research (N8 CIR) funded by the N8 research partnership (Universities of Durham, Lancaster, Leeds, Liverpool, Manchester, Newcastle, Sheffield and York) is gratefully acknowledged.

We acknowledge the support of the Society of Research Software Engineering for providing access to Mentimeter for the feedback session.

Our work made use of the facilities of the DINE HPC Service at Durham University funded by strategic investment funds of Durham's Department of Computer Science, and it also made use of the facilities provided by the ExCALIBUR Hardware and Enabling Software programme, funded by BEIS via STFC grants ST/V001140/1 and ST/V002724/1, and hosted by the DiRAC@Durham Memory Intensive 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 UK's National e-Infrastructure.

B Performance analysis tools

The following tools were presented and used in our workshop series. Where appropriate, we also made references to tools and facilities that are built into modern compilers and MPI environments.

B.1 Intel Application Performance Snapshots (APS)

Low-overhead tool summarizing and characterising CPU utilization (including memory accesses and vectorization), file I/O, memory footprint, OpenMP multithreading and MPI communication. Runs with optimised application executables to produce single-page text and HTML reports.

Part of the Intel VTune profiler bundle particularly supporting Intel processors, compilers and MPI libraries (but with limited support for others): https://software.intel.com/content/www/us/en/develop/documentation/get-started-with-application-performance-snapshot/top.html

B.2 Arm Performance Reports

Low-overhead tool summarizing and characterizing CPU utilization (including vectorization), file I/O, memory footprint, OpenMP multithreading, MPI communication, and GPU usage. Runs with optimised application executables to produce single-page text and HTML reports.

Part of the commercially licensed FORGE toolset from Arm Ltd available for a variety of architectures and systems: https://developer.arm.com/tools-and-software/server-and-hpc/debug-and-profile/arm-forge/arm-performance-reports

B.3 Score-P

Instrumentation and measurement infrastructure for profiling and recording event traces of MPI, OpenMP, hybrid (MPI+OpenMP) and other programming models. Supports a wide range of HPC systems (as well as smaller clusters), also including adapters for file I/O, memory usage and hardware counters.

Open-source developed and maintained by a consortium of tools developers: http://www.score-p.org

B.4 CUBE

Graphical performance analysis report explorer and utilities for profiles produced by Score-P and Scalasca comprising a multi-dimensional space of performance metrics, execution call-paths and system resources (process/threads). Coupled tree browsers for each dimension are complemented with plug-ins providing statistical and topological presentations of the distributions of metric values.

Open-source developed by Forschungszentrum Jülich and others: https://www.scalasca.org/software/cube-4.x/

B.5 Scalasca

Toolset for automated execution trace analysis to quantify communication and synchronization wait states and other inefficiences of MPI and OpenMP application executions. Scalability derives from parallel replay of Score-P traces exploiting the same computational resources as used for the traced application execution.

Open-source developed by Forschungszentrum Jülich and others: https://www.scalasca.org/

B.6 Vampir

Tool for interactive visualisation and exploration of parallel application execution traces generated by Score-P. Zoomable timelines of events on each process/thread showing interactions via messages and thread synchronizations.

Commercially licensed and developed by Technical University Dresden: http://www.vampir.eu

B.7 MUST

Runtime checker of the correctness of MPI usage, including detection of deadlocks and resource leaks, datatype mismatches and communication buffer overlaps. Produces an HTML report.

Open-source developed by RWTH Aachen University and Technical University Dresden: https://itc.rwth-aachen.de/must/

B.8 Archer

Low-overhead runtime data race detector for OpenMP programs, incorporated within LLVM ThreadSanitzer. Produces a textual report.

Open-source developed by RWTH Aachen University and others: https://github.com/PRUNERS/archer

B.9 MAQAO

Performance analysis and optimisation framework operating at binary level with a focus on processor core performance. Assesses code quality of the most time-consuming loops, providing estimates of the performance that could be reached along with hints on how to achieve it via compiler flags, pragmas and source code transformations. Agnostic to programming model, but mostly useful for single-node performance.

Open-source developed by University of Versailles St-Quentin-en-Yvelines: http://maqao.org

B.10 LIKWID

Command-line tools reporting system topology, controlling CPU/task affinity, and hardware performance monitoring with validated event sets and derived metrics.

Open-source developed by Friedrich-Alexander-University Erlangen-Nürnberg: https://hpc.fau.de/research/tools/likwid

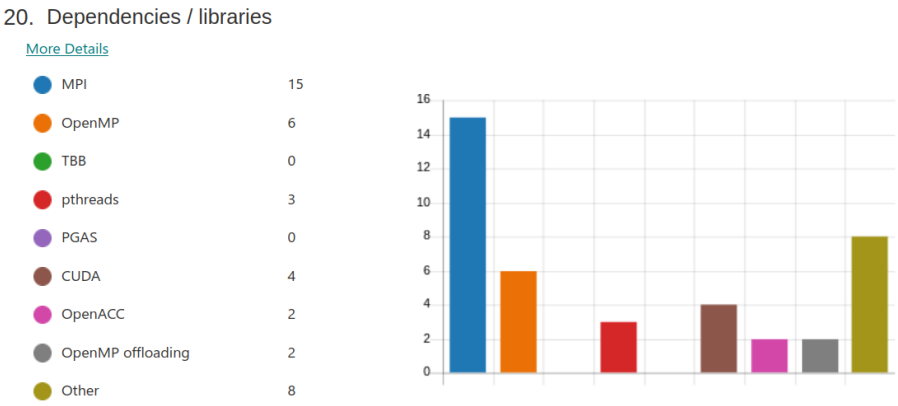
C Participants

Thirteen teams with a dedicated research code registered for the workshop series (SUSY LATTICE, Oxford RAMSES, FLASH and burn, Sherpa, Devito, Firedrake, preCICE, SeisSol, Peano/ExaHyPE, SWIFT, HemeLB, BOUT++, Zeltron). Two additional teams from two universities attended. They were made up of centralised RSEs, i.e. they did not bring one dedicated code with them but had access to a set of codes they had to work on as part of their day-to-day work. One of them had access to a high priority miniapp. Two individuals signed up for the course as well.

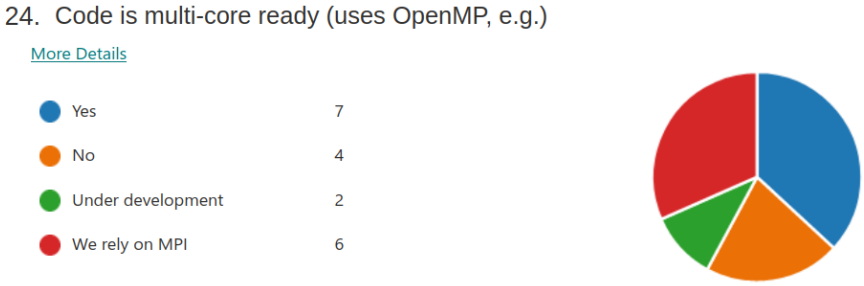
Over the seven months, we had 79 registrations, including seven instructors and three EPSRC representatives. Three out of the 69 "proper" registrations were PhD students, all others worked as PDRAs (postdoctoral researchers), for a research institute, or at a university in a Research Software Engineering role.

All codes have been open source and free in principle, though two software projects require a registration before they release the sources. The teams reported between 10 to several hundreds of users on a regular base, though the majority quantified their user base with around 50–100. The three predominant programming languages used by the codes were C++ (8), Fortran (7) and C (7), though four projects also rely heavily on Python.

While three projects listed more than three different funding agencies ranging from EU programmes and DoE support over UKRI agencies to industry, the remaining projects were only supported by one funding agency. Besides the UKEA and MetOffice support, this was mainly EPSRC, typically under the umbrella of one research grant. Several projects—in particular the single grant ones—acknowledged further institutional support.



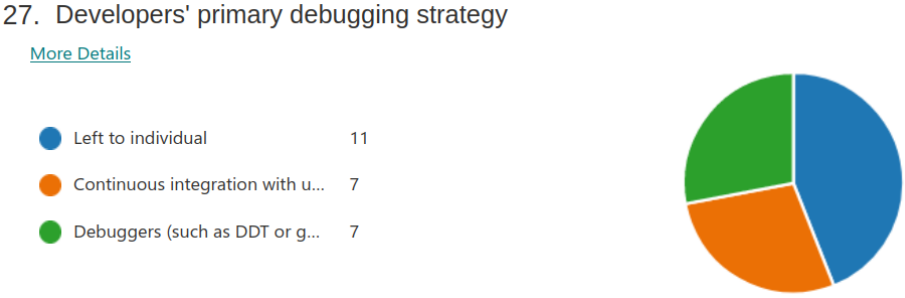
All teams relied on MPI, while the two RSE groups participating also work most frequently with MPI. All projects labelled their MPI support as mature and production-ready.



Half of the teams used OpenMP, three further teams used pthreads directly, i.e. a bespoke, homemade multicore parallelisation. However, only seven teams labeled their shared memory parallelisation as production-ready. Six have no intention to introduce any shared memory parallelisation paradigm and rely solely on message passing.

Eight teams either had mature GPU support or have started to investigate accelerator programming. While half of the GPU teams (4) use CUDA, the other half employs OpenACC (2) or OpenMP offloading (2). Overall, half of the projects have accelerators on their roadmap, while the other half has no intention to switch to GPGPUs. While three projects claimed that their GPU-port were "ready to go", no team claimed throughout the kick-off session that their GPU code variant were in production use on a regular basis.

Four projects had run their codes on a PowerPC platform before the workshop series, while five had gathered previous experience with ARM. All projects routinely ran on x86 systems.



Half of the teams used continuous integration in a systematic way to identify bugs, and half of the teams had access to debuggers and used them routinely. However, eleven teams said that the choice of correctness validation and debugging environment is primarily down to the programmers. Anecdotal evidence suggests that this means a heavy usage of `print` statements.

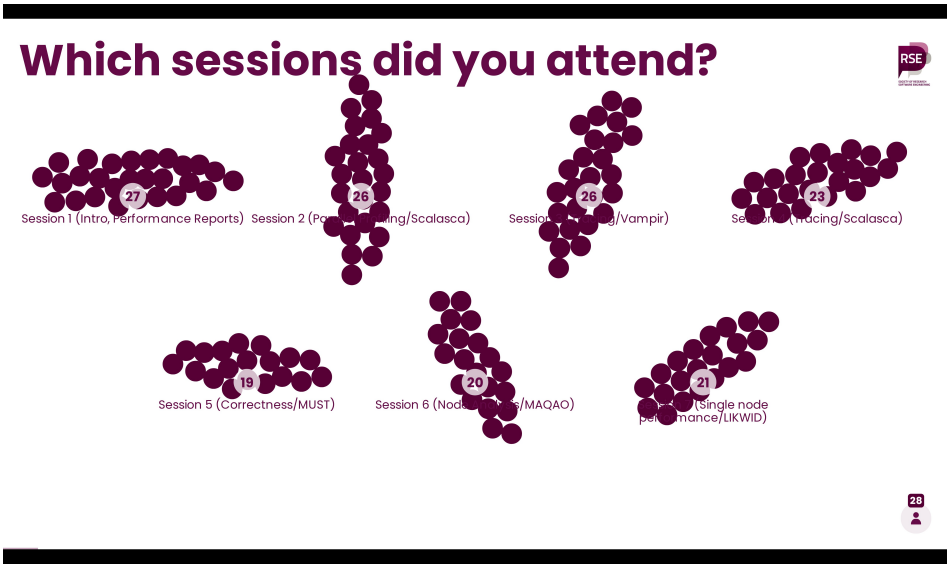
All teams (and the individual participants) reported that they have a systematic route to identify performance flaws, or have developed some home-grown analysis tools. Five teams claimed that the major performance flaws were well-known within their user community. Six reported that they had no idea where performance "is lost". Eight teams have used profilers before, while four teams could rely on performance regression testing.

While five teams participated because their current project required faster codes, six had a generic interest in runtime improvements which was not driven by a particular research project. Eleven teams reported that they would appreciate a broader overview of what is possible with modern tools and what tools are on the market, while twelve projects said they wanted to learn about new tools before they integrate new features and extend the code base. Five teams finally mentioned that they required a better understanding of the performance bottlenecks due to an increase in hardware parallelism.

The free text responses ("other") were dominated by three types of comments: First, participants had performance engineering on the agenda, but had not found time and resources for it so far. Second, a lot of participants knew that one particular aspect of their code (MPI or vectorisation) was heavily optimised (due to a research project, e.g.), while they assumed that the other aspects of their code had never really been studied systematically. Finally, several participants explained that they lack any education in performance analysis and encountered a very steep learning curve when they tried out performance analysis themselves.

D Wrap-up questionnaire

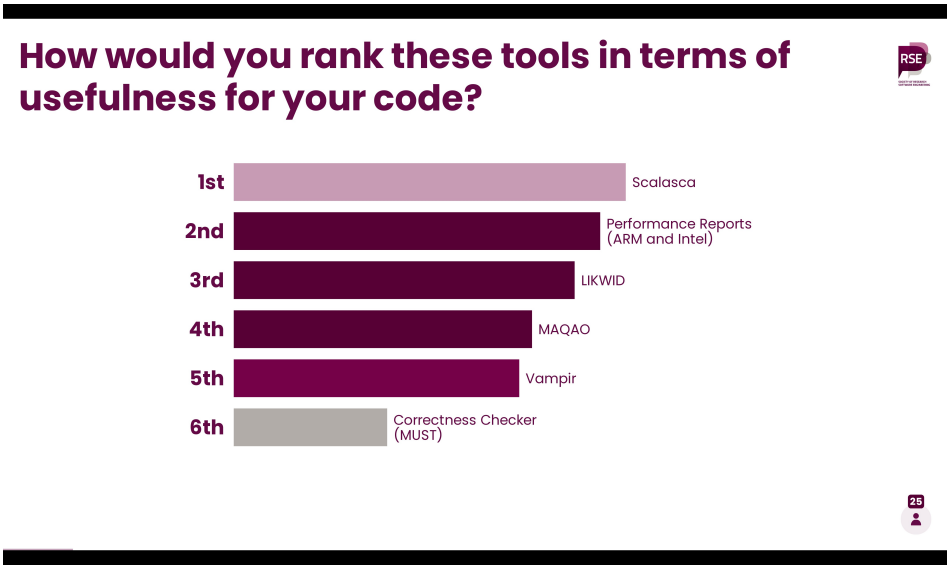
The quantitative final feedback was gathered via Mentimeter in an interactive session, i.e. we talked the participants through each individual question. Participants who did not participate in this session were sent the evaluation questionnaire separately. We had 16 participants answering the questions in the interactive session, and received answers from further 12 workshop participants offline. Note that not every participant answered every question. The total number of answers for a question is displayed in the bottom right for each Mentimeter graph. All questions about the tools were rated on a scale of 1–5, whereas 3 is neutral and higher is better/more positive. The personal experience is rated on a scale from 1–10.



Each single workshop day was attended by more than 20 participants. The slight drop for session 5 and 6 results from the fact that we had to move the workshops on short notice due to other commitments of the lecturers and it then unfortunately fell into UK vacation or pre-vacation weeks.

The questions were divided into two parts: First, we asked about the individual performance analysis tools, their ease of use, insight generated and quality of instructions, as well as what was positive about them and what could be improved. We do not provide all graphs and free-text answers here, but just summarise the gist of the answers. The second part of the questions was about the workshop format.

The participants first rated the tools according to the usefulness to their project.



It is difficult to digest any pattern from this ranking, but we observe that the top three tools are those that are able to provide users immediately with a high-level overview without diving deeper into the code details. We also observe that the correctness checks have not been considered particularly beneficial for the participating codes.

All participants gave positive feedback regarding the high-level overview tools studied. However, the level of insight was perceived as limited. A temporal breakdown of program phases (a simple timeline) was missing.

Scalasca was very positively received, and many teams obtained valuable insight from the tool. Special praise was due to the Cube visualisation concept which distinguishes the what—where—which code part dimensions. The two showstoppers for the participants were the enormous amount of data

generated such that they felt lost in spending significant time to tailor filter files, and the lack of support of the `MPI_THREAD_MULTIPLE` mode. Due to these showstoppers only a few teams succeeded in benefiting from Scalasca's performance flaw identification.

The feedback for Vampir was very similar to Scalasca with the same two major showstoppers. An additional interesting remark (likely regarding both tools) targeted the tool focus: Both focus on imbalances and compute time, while the actual memory consumption turned out to be a problem for some teams who wanted to track memory consumption per process.

The correctness checkers MUST and Archer received high praise for their ease of use (as well as for the accompanying instructions), but the impact on the present projects was limited. Most codes did not find any (severe) errors. Some participants asked for an integration of shared memory and distributed memory correctness checks.

MAQAO received mixed feedback which is worth a deeper analysis: The two major points of criticism were its GCC compiler focus (as some of the participating projects had switched to Clang), and its struggle with complicated loop macros, iterators and templates. The projects that did not suffer from these hurdles articulated positive or even enthusiastic feedback.

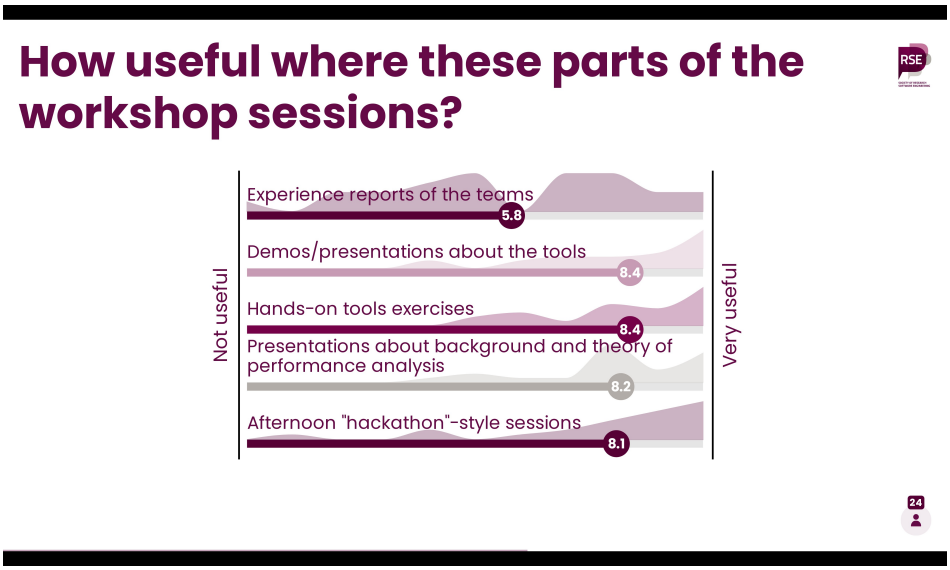
Likwid was very positively received though the participants noted that a high level of technical expertise is required to digest the data obtained. Some teams struggled to use Likwid's marker API, while the major point of criticism was the integration of the tool into MPI-based simulations.

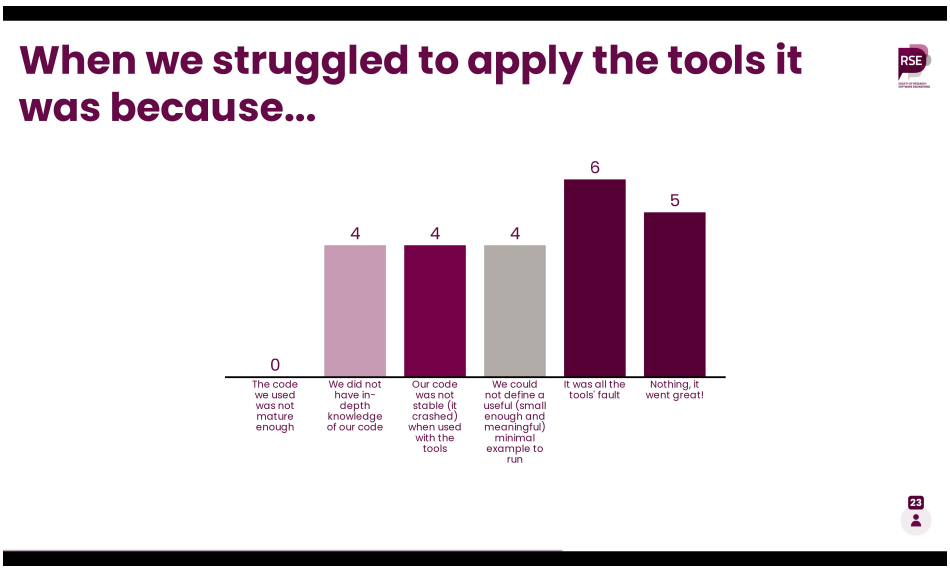
A challenge that arose for several tools was the fact that most tools assume that there is a fixed C or Fortran executable. Some tools provide a Python integration nowadays, but many still struggle if codes are controlled by a Python front-end and eventually yield machine code output, i.e. in situations where a performance analysis likely has to squeeze in-between a Python front-end and a compute back-end.

In the workshop-format part of the questions we received overall very positive feedback.

There were slightly mixed feelings, though, about the experience reports by other teams. At the same time, participants had asked multiple times for success stories and beyond-toy problem use cases for the individual tools.

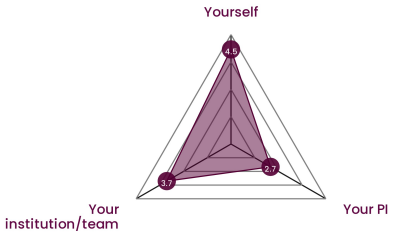
While the tool demonstrations received positive feedback, also the theory and programming recap was found valuable.





All participants felt that the codes they used were sufficiently mature, and that problems in applying the tools were either down to the tools themselves, or one of the following reasons: Participants suffered from a lack of insight into the code they were given or wanted to study; participants found that the codes became unstable if combined with tools (or yield suddenly different results); participants struggled to construct a meaningful and representative test that terminates within reasonable time.

Who benefitted (ultimately) the most from this workshop?

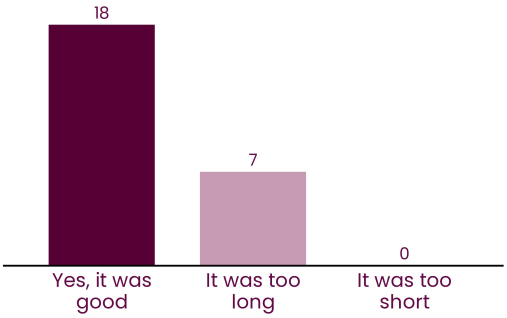


24

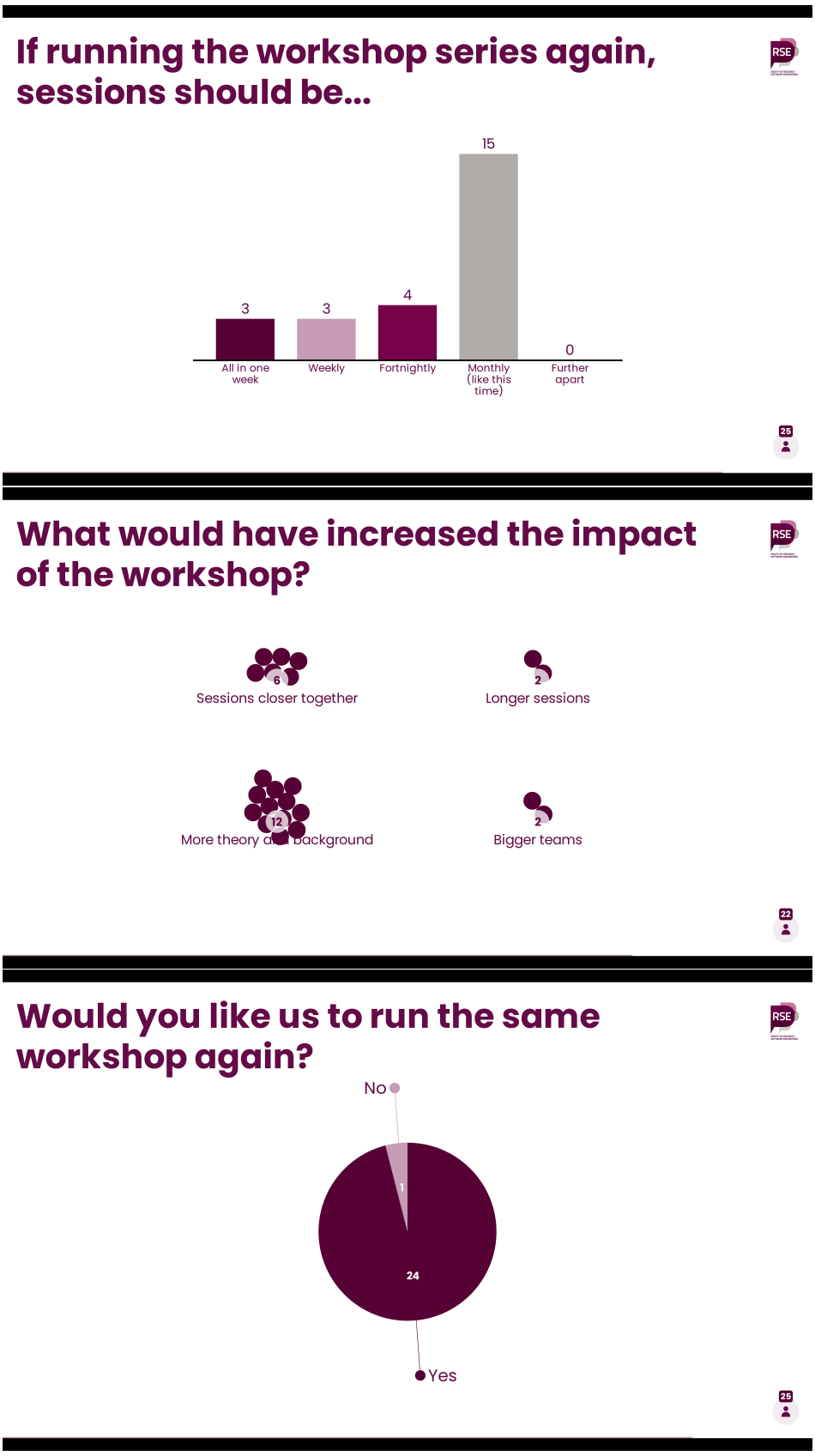
The majority of the participants saw this workshop as a way to improve their personal knowledge and skill set, rather than to introduce particular performance tweaks immediately. Seven participants nevertheless have been able to improve an ExCALIBUR code directly, while further seven plan to do so. The majority felt that it was first of all them personally who benefited from the workshop, with their team or institution at second place, and only then the (ExCALIBUR) PI who eventually is in charge to promote the code.

In line with the positive feedback on the (few) recap and theory sessions that we had offered, and in line with the observation that some teams struggled with the more sophisticated in-depth analyses (Scalasca analyses, MAQAO, Likwid), participants would have appreciated more theory and background information, i.e. information that might best fit into a classical computer science education.

Was the length (7 months) of the workshop series right?



25



The majority of the participants was very happy with the overall format, and the survey suggests that it would be appreciated if it were run again in a very similar way.

Asking for suggestions for improvements the following were raised by multiple participants: Ob-

viously, GPU support was high up on the list. Many asked for more in-depth information and more theory, though one team raised the issue that performance analysis as a service is an important skill, too—in particular if RSEs are "bought in" from centralised university divisions. A third and final remark was that many participants missed the opportunity to translate performance analysis insight into code modifications; though it remains an open question why this has not been possible given that there had been roughly four weeks in-between the sessions.