Department: Head
Editor: Name, xxxx@email

Navigating Performance, Portability and Productivity

S. J. Pennycook
Intel Corporation

J. D. Sewall
Intel Corporation

D. W. Jacobsen
Intel Corporation

T. Deakin
University of Bristol

S. McIntosh-Smith
University of Bristol

***Abstract*—The phrase "performance portability" is commonly used, but may mean different things to different people. Developing a better appreciation of the needs of different software developers and a framework for talking about these needs improves our ability to define goals, design experiments and make forward progress. This article discusses a methodology for quantifying, summarizing, visualizing, and understanding application performance portability and programmer productivity.**

"PERFORMANCE PORTABILITY" has long been a nebulous term. Notes from a meeting on the subject held in 2016 highlight the lack of a "universally accepted definition of performance portability", observing that "several attempts were made by various speakers to take a crack at it" [1]. How could so many researchers be said to be working towards a common goal, if they could not agree upon what it was?

The attendees all ascribed *importance* to the term performance portability even if no precise meaning was agreed upon. Increasing micro-architectural diversity and specialization had created challenges to address in software, impacting the performance and portability of applications and the productivity of the programmers creating them. An ecosystem was beginning to develop around frameworks promising to improve performance portability and maintainability [2]. In the absence of precise definitions, subjectivity prevailed, with attendees imposing their own weighting to the many facets of software development and maintenance—two words was too brief a prompt. In the years since this initial meeting,

 Published by the IEEE Computer Society

the "Performance, Portability and Productivity in HPC (P3HPC)" community has made significant progress towards shared terminology, and we are closer to a universally adopted methodology for assessing performance portability and programmer productivity than ever before.

This article seeks to prepare readers to safely navigate the perilous waters surrounding the "three Ps" (i.e. performance, portability and productivity). We will briefly revisit the issues that complicate discussions in this space, and summarize continuing efforts to minimize misunderstandings via a combination of precise definitions and quantitative metrics. The techniques and best practices discussed here are the result of multiple years of research, and represent the state-of-the-art in assessing and comparing the behaviors of performance-portable applications.

GROUNDWORK

Quantitative performance characterization is a well-studied area, but this does not mean clear sailing to precise and objective discussion of *performance portability*, *performance productivity*, and other combinatorial considerations of these topics. The sorts of discussions that prevail rely on vague claims such as "good performance portability", or "productive, portable environments". Subjective statements like these have little useful meaning, and can stymie discussion. Here, we explore some of the headwinds that beset this topic.

Measurement

Performance measurement is well-studied, but few established metrics for productivity and portability exist. Productivity in particular is notoriously difficult to measure; notions such as "programmer time" and "cost" are hard to compare across developers and environments, and little can be done to quantify these values for code that has already been developed. See Wienke et al. [3] for an exploration of some of these considerations and approaches for making useful measurements.

Aggregation

Aggregating performance, portability, and productivity measurements meaningfully is difficult. Qualitative relationships to aggregated concepts can be identified (e.g. better performance should mean better performance portability, and more effort should lead to less productive portability), but the details of how these should translate into quantitative techniques are fraught. While it is tempting to apply familiar statistical tools when confronted with bulk data, we have observed that these tools rarely provide insight or align with intuition [4].

Subjectivity

Even given agreement on the individual meanings of performance, portability, and productivity, their combination assumes a distinct quality. Developers are likely to ascribe meaning according to their own priorities and needs. To some, performance portability means that a single source code runs everywhere, and that it does so with maximally efficient performance. To others, it means that a software package may contain target-specific source code, but that the absolute performance numbers on each target of interest are very close together. To still others, it means that some minimum threshold of absolute performance (e.g. a required number of simulation seconds/day) is attained across all targets of interest.

Definitions

While there is consensus that striving for performance, portability, and productivity is "good", there is ample space for disagreement about the details. The first step toward productive discussion is for every developer to understand and communicate what they value. To that end, we introduced a list of questions for developers in a recent paper [4]; answering these questions requires developers to confront assumptions that may lead to misunderstandings.

To foster productive discussion, we proposed definitions for various concepts related to the study of performance, portability, and productivity. These definitions are intentionally both precise and broad to provide an objective foundation for future work.

- *Problem*—An input to an *application*, with a correctness test and observable performance.
- *Application*—A collection of software that can run *problems* on one or more *platforms*.
- *Platform*—A collection of hardware and software on which an *application* runs *problems*.

These definitions provide a common basis for description that practitioners build upon by specifying further details. Consider a study investigating the performance portability of a linear algebra package. This package—all of the code needed to run on the platforms of interest—forms the application. A specific set of input matrices and their expected product forms the problem. The platforms are formed by the various hardware systems that are the subject of the study.

Practitioners have considerable discretion in the level of detail that must be specified; details of the software stack may not be particularly relevant in a study where each platform uses a different micro-architecture, but become necessary for studies using a single micro-architecture with different system configurations. Therefore, a study should clearly state what among the software stack is part of the application and what is part of the system; an application that links to vendor-provided libraries for the Fast Fourier Transform (FFT) on each platform is likely to count these libraries as part of the platforms rather than as part of the application proper. In another study, the FFT library itself could be an application. Whether libraries are categorized as part of an application or part of a platform is ultimately dependent on the nature and goals of the study.

MEASURING PERFORMANCE PORTABILITY

If an application is developed with the goal of being performance portable, it follows that there should be a way to track progress towards that goal. Measuring performance portability is necessary to show the extent to which an application is (or is not) performance portable, and to provide insight into how the performance portability of an application might be improved.

Measuring portability

The *portability* of an application is binary—either the application runs correctly on all of the platforms it needs to, or it doesn't. This property follows from our focus on the portability of applications, rather than the portability of constituent source code; the difference is subtle, and demonstrates the role of clear definitions in productive discussions.

Measuring performance

The performance of an application can be measured in many ways: time to solution, sustained memory bandwidth, the rate of floating-point operations (FLOP/s), etc. However, platforms have intrinsic differences in peak performance, and knowing which platform is fastest overall for an application is only part of the picture. Normalizing the performance that an application achieves on one platform against the performance it *could* have achieved provides insight into how well the application utilizes different kinds of platforms, and thus into its performance portability.

In previous work [5], we proposed two complementary efficiencies with which to normalize performance. The *architectural* efficiency measures how close the application runs to the performance of the processor specified on its specification sheet (e.g. how many GFLOP/s the application manages to complete relative to the peak GFLOP/s of the processor). A variant that measures performance relative to a Roofline model [6] has become popular because it allows for the calculation to be largely automated. Such measures are naturally extended to other performance models.

Calculating the architectural efficiency for a large application is not always tractable—different parts of an application may exercise different capabilities of a processor, and capturing such behavior accurately in an efficiency measurement often requires a complex analytical performance model. To address this, *application* efficiency can be used to represent performance as a percentage of the best known performance previously demonstrated for the problem on a given platform. This is similar in spirit to comparing against a "world record", although due diligence is required as there is rarely a centralized list of application performance results.

Calculating both efficiencies is often useful: comparing a portable application to the fastest known (potentially non-portable) way to solve the same problem on the same platform provides insight into whether an application can still be improved even if its architectural efficiency is high (e.g. by using a different algorithm).

A metric for performance portability

Our proposed performance portability metric (Φ) [5] is shown in Equation (1).

$$\Phi(a,p,H) = \begin{cases} \dfrac{|H|}{\sum\limits_{i \in H} \dfrac{1}{e_i(a,p)}} & \text{if, } \forall i \in H \\ & e_i(a,p) \neq 0 \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

The metric is the harmonic mean of application a's performance efficiency $e(a,p)$ when executing problem p on a set of platforms H. If any efficiency is zero, we define $\Phi = 0$; this reflects that a non-portable application is not performance portable, and is consistent with the limit of the harmonic mean.

This metric provides an overall picture of an application's performance portability, and has proved to be a useful way to summarize performance portability data. Aspects of the metric, such as normalizing performance across platforms and accounting for unsupported platforms are widely accepted, but discussion continues about ways to further improve and extend the metric [7], [8], [9].

Interpreting performance portability

After measuring the efficiency of contrasting implementations across a variety of platforms, Φ can be calculated and used as the basis for comparison.

Most commonly, we see comparisons of implementations using different parallel programming languages and frameworks (such as our previous study in Deakin et al. [10]). Such studies seek either to track the development of programming environments themselves (i.e. the ability of each implementation to deliver good performance on all platforms for one benchmark), or to identify the most promising approach (i.e. whether one implementation always delivers the highest scores for a large suite of benchmarks).

Alternatively, one can compare different implementations in the same programming language or framework. Both high-level choices (e.g. the algorithms and data structures used) and low-level choices (e.g. the parallelization strategy or loop structure) will have different impacts on performance on different platforms.

Visualizing performance portability

As with all averages, summarizing an entire data set with a single number may hide some useful information. Comparing different values of Φ allows developers to reason about which application is "best", but provides no insight into why that is the case. Some performance portability analyses require developers to form a more holistic view of their data set, and for such analyses we have found it convenient to use visualizations [4].

We have developed the *Cascade Plot*, an example of which is shown in **Figure 1**. Cascade plots are designed to help identify trends in the performance portability data: applications extending further to the right support more platforms (portability), and if they do so with higher efficiency, are more performance portable.

To construct the cascade plot, the efficiencies for each application are sorted from high to low. These efficiencies, shown as solid lines, are then plotted against the platform ranking. The plot is a min-max function, showing the minimum efficiency among platforms for each increasing subset of the N best platforms. In concrete terms, consider the points plotted where $N = 5$: for each application, the efficiency plotted is the lowest efficiency among the five most efficient platforms. The Φ, shown as dashed lines, is calculated on that same set of platforms.

Underneath the cascade plot, each platform is assigned a unique color. Since the platforms are sorted independently for each application, a given platform may be found at different positions along the horizontal axis for different applications. Each row indicates the platform ordering for the corresponding application. This data is useful to highlight precisely which platforms introduce the most significant changes in the performance portability of each application.

A case study with BabelStream

The cascade plot for BabelStream, shown in Figure 1, contains data for six applications on fifteen platforms and allows for many comparisons. We consider two specific examples here.

The OpenACC implementation (in green) achieves nearly peak efficiency on approximately four platforms, but a range of lower efficiencies on other platforms. Note that there are four platforms where it does not run at all. The cascade

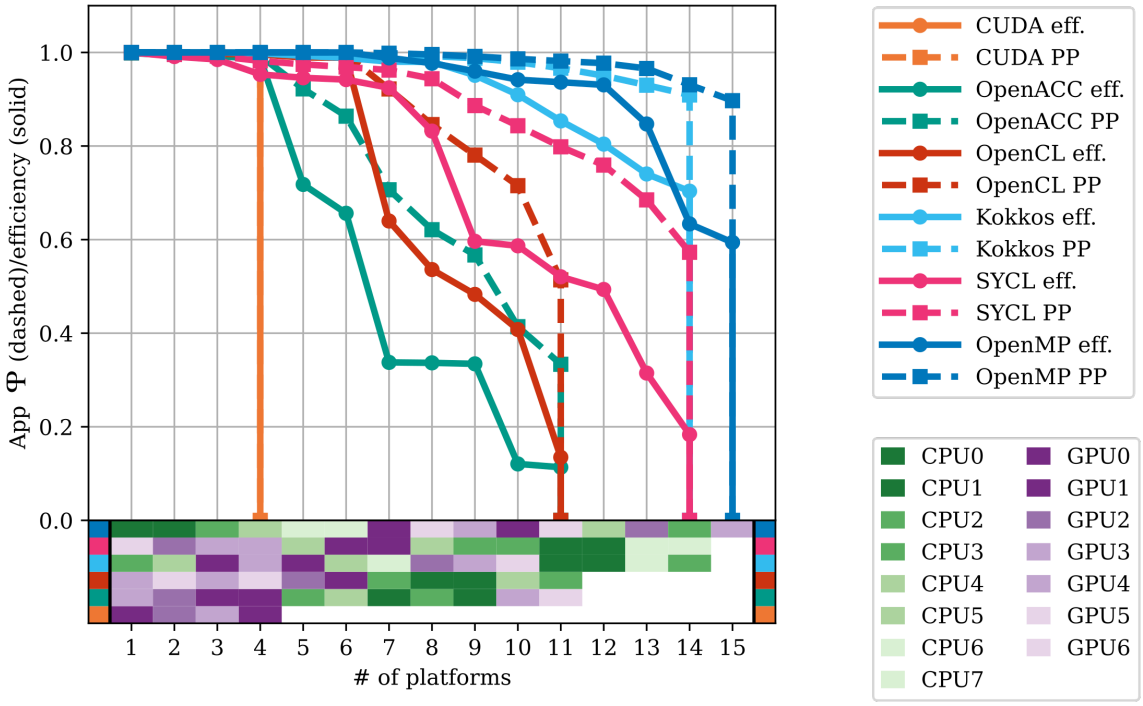


Figure 1. A cascade plot for several implementations of BabelStream (each implementation is considered an application). The lines show sorted Φ and efficiency data, and the chart beneath indicates an application's platform ordering. Different platform types have been colored differently: CPUs in purple and GPUs in green.

plot shows this as a steep decline in efficiency (solid line) and performance portability (dashed line), before intersecting the x axis at platform 11. The corresponding (second from bottom) row of the platform ordering provides some explanation: the model works well on four GPU platforms, but reduces in efficiency as the set is expanded to include CPU platforms. Said another way, the OpenACC implementation of BabelStream achieves high performance portability across a small subset of the platforms, but no performance portability across the complete set of platforms. If we limit our analysis to the 11 platforms with non-zero results, the performance portability is still lower than some of the alternative implementations.

The OpenMP implementation (in blue) shows high efficiency (and thus high Φ) on all of the platforms in this study. Critically, it is the only implementation to support all platforms. The cascade plot also shows that the implementation exhibits fairly consistent efficiency across 80% of the platforms—something which cannot be deduced from the Φ value alone. Observe that the Φ (dashed line) always equals or exceeds the efficiency at a given number of platforms, and that, as an average, it smooths out sharp drops in efficiency.

MEASURING PRODUCTIVITY

The Φ metric provides developers with a tool to reason about whether an application meets a stated performance portability goal. However, it does not show *how* those goals are met—it is possible for two applications to achieve the same score with drastically different development approaches. It is desirable to further distinguish between "simple" and "complicated" solutions to performance portability, which have very different implications for programmer productivity.

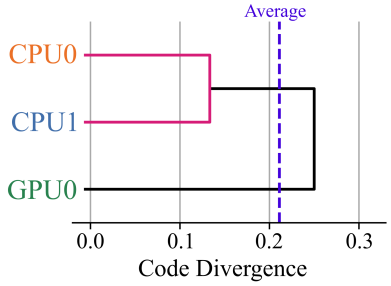
Measuring programmer productivity in general and absolute terms is likely to be impossible, and obtaining a complete picture of programmer productivity requires attention to all aspects of writing code, accounting for developer experience and choice of programming model [3]. Our experience suggests that performance portability practitioners are often satisfied by approximations

```
#if defined(CPU0)
#define SIMDLEN 4
#elif defined(CPU1)
#define SIMDLEN 8
#elif defined(GPU0)
#define SIMDLEN 16
#endif
void vector_add(float *a, float *b, float *c, int N) {
#if !defined(OFFLOAD)
#pragma omp parallel for simd simdlen(SIMDLEN)
#else
#pragma omp target teams distribute parallel for simd simdlen(SIMDLEN)
#endif
  for (int i = 0; i < N; ++i) {
    c[i] = a[i] + b[i];
  }
}
```

(a) Code snippet with C preprocessor-based specialization. Orange lines are specific to CPU0, blue lines are specific to CPU1, and green lines are specific to GPU0. Pink lines are used only by CPU0 and CPU1.

		CPUs		Offload
		CPU0	CPU1	GPU0
CPUs	CPU0	0.0	0.13	0.25
	CPU1	0.13	0.0	0.25
Offload	GPU0	0.25	0.25	0.0

(b) Code distances as computed using Equation (3).



(c) Dendrogram showing clustering of code distances.

Figure 2. Computing divergence provides insight into how platforms employ specialization.

relating to factors that are known (or assumed) to impact programmer productivity.

One such factor is the amount of platform-specific effort, which can be approximated by the degree to which an application employs platform-specific code (or code *specialization*). The presence of platform-specific code indicates two sources of additional effort (and hence reduced productivity): firstly, the aforementioned need to write new code for each new platform that may be introduced in the future; and secondly, the burden of maintaining multiple representations of the same code for existing platforms.

The ideal solution to performance portability, then, is a programming language that is simple to use, inherently portable to all platforms, and which delivers the highest level of performance with no platform-specific tuning whatsoever. This is the aim of both domain-specific languages and performance portability frameworks (such as Kokkos [2] and RAJA). Conversely, the most negatively perceived solution to performance portability is to write disparate code bases that are highly tuned for individual platforms, potentially using different programming languages.

Code divergence

The "code divergence" metric we proposed [11] can be used to establish a spectrum between the least and most productive solutions to performance portability. The metric builds on the terminology used by Φ, as shown in Equation (2).

$$\mathrm{CD}(a,p,H) = \binom{|H|}{2}^{-1} \sum_{\{i,j\} \in H \times H} d_{i,j}(a,p) \quad (2)$$

Code divergence represents the average "distance" between the source code required to compile application a and execute problem p for each

pair of platforms in H, where $d_{i,j}(a,p)$ can be any metric that represents the distance between two source codes.

We have found it most useful to use the Jaccard distance, which is shown in Equation (3). The Jaccard distance is a dissimilarity metric for sets; we define $c_i(a,p)$ to represent the set of source lines required to compile application a and execute problem p for platform i.

$$d_{i,j}(a,p) = 1 - \frac{|c_i(a,p) \cap c_j(a,p)|}{|c_i(a,p) \cup c_j(a,p)|} \quad (3)$$

By construction, values of this distance metric fall in the range $[0,1]$: a value of 0 means a "single-source" code where everything is shared by both platforms; and a value of 1 means the application source is separate for each platform and nothing is shared. Taking the average of these distances to compute $\mathrm{CD}(a,p,H)$ extends this intuitive metric to cases with more than two platforms, while remaining in the range $[0,1]$.

A case study with Code Base Investigator

To simplify the computation of code divergence for real code bases, we have written a tool called Code Base Investigator (CBI). Given a list of source files and a set of platform configuration options, CBI identifies which source code is compiled for each platform, and uses that information to compute divergence. **Figure 2** demonstrates how CBI computes and reports divergence for a simple example.

First, CBI preprocesses the file for each platform and records the sets of lines required to compile the file. It is important to note that our methodology counts preprocessor directives themselves as required for compilation; this ensures fair comparisons between preprocessor and non-preprocessor based solutions to specialization, and reflects that preprocessor logic itself may be re-used across multiple platforms. For our simple example, we consider three platforms: 1) a CPU platform that defines the `CPU0` macro; 2) a CPU platform that defines the `CPU1` macro; and 3) a GPU platform that defines the `GPU0` and `OFFLOAD` macros.

Second, the distance between each pair of platforms is computed and recorded in a distance matrix. The CPU0 and CPU1 platforms require 15 lines of code in total (i.e. everything except the lines guarded by the `GPU0` and `OFFLOAD` macros), sharing 13 lines (i.e. because the definition of `SIMDLEN` differs) giving a distance of $1 - 13/15 = 0.13$. Similarly, the CPU0 and GPU0 platforms require 16 lines of code in total (i.e. everything except the line guarded by the `CPU1` macro), and share 12 lines (i.e. because the definition of `SIMDLEN` and the selected OpenMP pragma differ) giving a distance of $1 - 12/16 = 0.25$. Note that the calculations use different denominators, reflecting differing amounts of "unused" code; no combination of two platforms in H use all 17 lines of code. Computing the average over all unique pairs of platforms gives the final code divergence metric: $(0.13 + 0.25 + 0.25)/3 = 0.21$.

Third, the computed distances can be used to produce visualizations of code divergence that provide insight into the structure of a code base and its use of specialization. The dendrogram shown summarizes the distance matrix via clustering, highlighting for our simple example that the CPU platforms share the most code.

This example demonstrates that tracking divergence can be an effective way to provide actionable insight for application developers, identifying dissimilar code paths that could benefit from abstractions, or platforms with a higher-than-average maintenance burden. Work is underway to extend CBI further, to help developers identify which areas of a code contribute to divergence, and the extent to which divergence is necessary for performance and/or portability.

PUTTING IT ALL TOGETHER

Given ways to measure performance portability and code divergence, how can we form a complete picture of an application's performance, portability, and productivity?

Φ–CC charts

At the 2020 P3HPC Forum we introduced performance-portability code-convergence (Φ–CC) charts as a means to visualize this holistic analysis by plotting performance portability and code convergence in a bounded 2D space. See **Figure 3** for an example.

Note that these charts use code *convergence* rather than code *divergence* as in previous sections. The convergence is simply the divergence

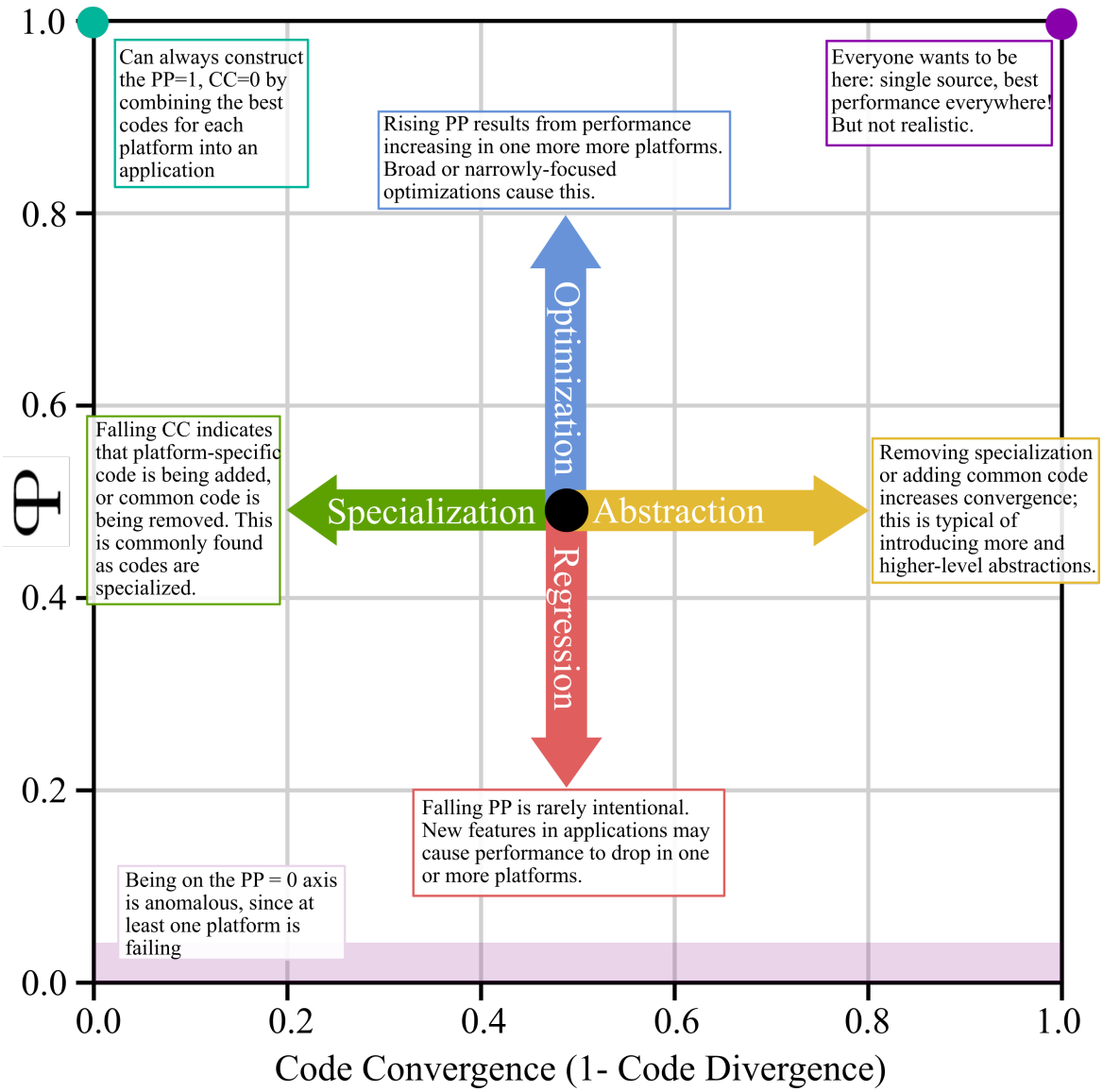


Figure 3. Ƥ-CC chart; plotting performance portability against code convergence can help developers to identify which actions to take next in order to move them closer to an ideal solution (top right).

subtracted from 1—whether working with divergence or convergence is easier depends on the analysis being performed, and when combining with Ƥ it is convenient to choose metrics such that higher is always better.

There is a great deal of insight to be had from understanding both absolute points in this space as well as what it means to traverse it through code changes. While we are free to explore this in abstract terms to help our understanding, it is important to note that any concrete consideration of these plots must involve a fixed platform set. So too must we decide which variety of performance efficiency we wish to use for the performance portability axis; for the purposes for exposition, we assume application efficiency simply because it makes the extreme of 1 attainable.

Navigating the plane

We can characterize a given state of an application using its Ƥ and CC, and plot these results on a 2D plane. We can set goals as to where we would like the application to be. How do we get there?

Figure 3 is our guide, featuring a compass rose that ascribes software engineering terms to

the usual cardinal and ordinal arrows. This tool can be used to identify the broad classes of transformations required by an application to realize our aspirations in performance, portability, and productivity. Note that it is rare that an application would move in a strictly axis-aligned fashion, and real observations show diagonal translations in the space.

Every developer understands that there is a relationship between abstraction and performance. It would be unusual for a developer to specialize their code (decreasing CC) and to see it regress in performance (thereby decreasing Φ); rather we would expect specializations to be introduced as part of an optimization to help improve performance on certain platforms.

Similarly, we are all familiar with the potential performance costs of introducing high-level abstractions (increasing CC), and sometimes that is a trade-off we are willing to accept for better maintainability. It is somewhat rare to see adoption of abstractions also increase Φ; the most common example of this sort of movement would be from the adoption of external libraries in favor of specialized implementations inside the application itself.

A case study with Hydro2D

In **Figure 4**, we show a real-world example of an application's course through the Φ-CC space over time. The application (Hydro2D) is a shock hydrodynamic benchmark code—a proxy application consisting of approximately 5 000 lines of code—from CEA that was the subject of an optimization study [12].

The initial code targeted multiple CPUs with different instructions and features using the compiler's auto-vectorization capabilities. This resulted in low Φ but—with no specializations in the code—a CC of 1.

In pursuit of performance, manual vectorization was used for multiple instruction sets to optimize for the target CPU micro-architectures as specifically as possible, using a modified algorithm incompatible with auto-vectorization. The result was new levels of performance (and thus, Φ) while introducing significant code divergence. This sort of trade-off between increasing Φ for decreasing CC is typical of specialization optimizations.

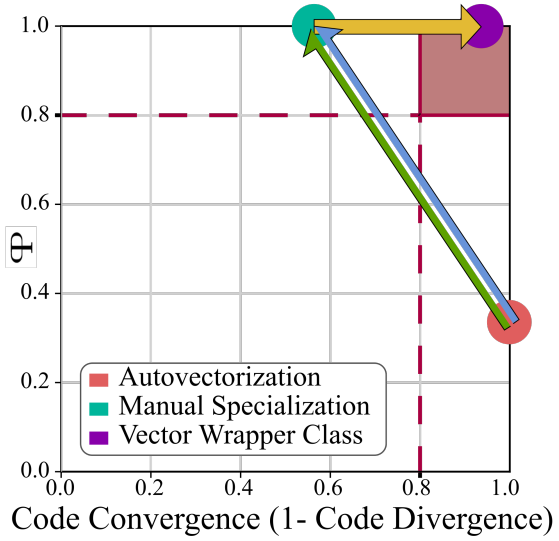


Figure 4. Φ-CC chart of Hydro2D's course. The initial Hydro2D code relied on auto-vectorization. Applying manual specialization decreased convergence while improving performance, corresponding to movement in the specialization and optimization directions. Encapsulating the specialization into wrapper classes resulted in higher convergence while maintaining performance, corresponding to the abstraction direction. The dashed red lines at $\Phi = 0.8$ and $\mathrm{CC} = 0.8$ show how one might define a hypothetical goal region when developing a code.

For the final tack, a vectorization abstraction was introduced to factor out the diverging, CPU-specific code paths. The design of this abstraction allowed for performance and Φ to remain unchanged while CC increased back to almost 1.

It has been our experience that intermediate steps in the development of an application may lead to temporary decreases in Φ or CC, but that these transformations can eventually lead to a more performant and maintainable code.

Aggregate analysis

Thus far, our discussion has examined how one might analyze a single application, or perhaps a family of applications that solve the same problems. This is a good perspective to work from for an application developer, but there are cases where the performance, portability, and productivity of many disparate applications are of interest. Consider the role of a compiler de-

veloper (who seeks to provide high-quality tools for application developers), or a computing center administrator (who seeks to understand how their systems are being used as a whole).

Analysis of P and CC can provide useful insight into these cases as well. The first step is to select the set of platforms of interest H. Then, for each application of interest, we can compute the P and CC scores for H. The results for all applications can then be plotted onto a single P-CC plane for a visualization of the aggregate.

While it would be a challenge to attempt to catalogue the many different ways that an aggregate set of data may appear in a P-CC chart, we can identify general qualities. Data points clustered near the $\mathrm{CC} = 1$ line indicate a proclivity for single-source programming languages or frameworks, while clustering near $\mathrm{CC} = 0$ indicates that there are few tools for low-overhead specialization available. Applications near $\mathrm{P} = 1$ and $\mathrm{CC} = 0$ may share the priority to get performance portability over productivity concerns; so called "duct tape" codes are not uncommon.

In recognition of the tension between P and CC, one possible strategy is to define a metric that combines Equations (1) and (2)—transformed to code convergence—with a minimum operator:

$$M(a, p, H) = \min \mathrm{P}(a, p, H), \mathrm{CC}(a, p, H) \quad (4)$$

The intention is to use this in aggregate P-CC analysis to enable simple one-dimensional analysis of application distributions. For example, the dashed red lines in Figure 4 indicate how one might depict the requirement that $M \geq 0.8$.

Another option is to compute the median M across all applications; compiler developers at Intel are using a related approach to evaluate the progress of the oneAPI initiative.

VOYAGE TO SUCCESS

Despite the inherent subjectivity in all aspects of performance, portability, and productivity discussions, the terminology summarized in this article can be used to accurately communicate goals and best practices. Coupled with quantitative data, developers can start to reason about specific actions that they can take to improve their applications.

As demonstrated in Figure 4, improving an application's performance, portability and productivity is an iterative process. We recommend tracking performance portability and code convergence over time, beginning as early as possible during code development, to identify improvements before they become too painful to employ. We view this as a four step process, which we outline below.

Step 1: Identify the destination

Determine your platforms of interest, acceptable values of performance portability, and code convergence for the application, and plot the intersection of these lines (as in Figure 4). This initial step accounts for subjectivity, and enables development teams to set application-specific goals.

Step 2: Get your bearings

Compute and plot the performance portability and code convergence for each kernel of interest (or the application as a whole). Using methodologies like Roofline and tools like Code Base Investigator can simplify this process, and it is straightforward to automate this step for a given application.

Step 3: Chart the course

Use the compass guides from Figure 3 to determine how to improve the application: whether optimization or abstraction is required, and the degree to which additional specialization can be employed in pursuit of performance.

Step 4: Set sail

Take action to implement the changes identified in Step 3. The precise nature of this step will be application- and platform-specific, depending on the availability of language support for specialization and the generality of relevant optimizations.

CONCLUSION

The methodology demonstrated in this article is the culmination of significant progress as a community towards a robust way of analyzing the three Ps. There is consensus that all three Ps are desirable, but without the rigorous metrics, tools, and methods detailed here, the concepts are difficult to discuss in a quantitative way.

Measuring the performance, portability, and productivity of applications using P and code convergence enables developers to understand their applications via three new analyses. First

and foremost, the measurements themselves provide an objective snapshot of the current state of an application. Second, cascade plots provide an ability to dig deeper into the behavior of an application across multiple platforms, helping to identify trends in both performance and portability. Finally, a combined Φ-CC chart fulfills a similar role to a Roofline model [6]: helping developers to track progress relative to their performance, portability, and productivity goals.

We are already seeing the benefits of following such a methodology. It has impacted the way that many challenges in the space are discussed, and is informing the future directions of performance portable, productive, standardized parallel programming languages and frameworks—the OpenMP and SYCL standards, to name two. Although this article uses simple applications to demonstrate our approach, we are close to a reality where production-grade applications can be implemented in portable languages and run across a wide variety of architectures. With careful effort, we believe developers will shortly be able to demonstrate productive performance portability.

Exploring the ideas discussed in this article using larger and more complex applications is the next step, and we encourage all application developers to test our methodology by measuring performance portability and code convergence for their largest and most important codes. Together, we can continue to work on refining and improving the metrics, methods and visualization techniques for analysing the three Ps: performance, portability and productivity.

RESOURCES

- *Scripts and Notebooks*—Tools to calculate Φ and draw cascade plots are available from the University of Bristol's GitHub: github.com/UoB-HPC/performance-portability, doi: 10.5281/zenodo.5012530
- *Code Base Investigator*—Tools for calculating and analyzing code divergence/convergence are available from Intel's GitHub: github.com/intel/code-base-investigator, doi: 10.5281/zenodo.5018973

ACKNOWLEDGMENT

The authors wish to thank the Performance, Portability and Productivity in HPC (P3HPC) community and attendees of Dagstuhl Seminar 17431 "Performance Portability in Extreme Scale Computing: Metrics, Challenges, Solutions" for their feedback on the ideas outlined in this article.

This work was in part funded by EPSRC through the Strategic Partnership in Computational Science for Advanced Simulation and Modelling of Engineering Systems (ASiMoV) project, grant number: EP/S005072/1.

NOTICES & DISCLAIMERS

REFERENCES

1. J. R. Neely, "DOE Centers of Excellence Performance Portability Meeting," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-700962, 4 2016, doi: 10.2172/1332474.
2. H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 12 2014, doi:10.1016/j.jpdc.2014.07.003.
3. S. Wienke, J. Miller, M. Schulz, and M. S. Müller, "Development effort estimation in HPC," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 107–118, doi: 10.1109/SC.2016.9.
4. J. Sewall, S. J. Pennycook, D. Jacobsen, T. Deakin, and S. McIntosh-Smith, "Interpreting and visualizing performance portability metrics," in *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2020, pp. 14–24, doi: 10.1109/P3HPC51967.2020.00007.
5. S. J. Pennycook, J. D. Sewall, and V. W. Lee, "Implications of a metric for performance portability," *Future Generation Computer Systems*, vol. 92, pp. 947–958, 2019, doi: 10.1016/j.future.2017.08.007.
6. C. Yang, R. Gayatri, T. Kurth, P. Basu, Z. Ronaghi, A. Adetokunbo, B. Friesen, B. Cook, D. Doerfler, L. Oliker, J. Deslippe, and S. Williams, "An empirical roofline methodology for quantitatively assess-

ing performance portability," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2018, pp. 14–23, doi: 10.1109/P3HPC.2018.00005.

7. A. Sedova, J. D. Eblen, R. Budiardja, A. Tharrington, and J. C. Smith, "High-performance molecular dynamics simulation for biological and materials sciences: Challenges of performance portability," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2018, pp. 1–13, doi: 10.1109/P3HPC.2018.00004.
8. D. F. Daniel and J. Panetta, "On applying performance portability metrics," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 50–59, doi: 10.1109/P3HPC49587.2019.00010.
9. A. Marowka, "Toward a better performance portability metric," in *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2021, pp. 181–184, doi: 10.1109/PDP52278.2021.00036.
10. T. Deakin, A. Poenaru, T. Lin, and S. McIntosh-Smith, "Tracking performance portability on the yellow brick road to exascale," in *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2020, pp. 1–13, doi: 10.1109/P3HPC51967.2020.00006.
11. S. L. Harrell, J. Kitson, R. Bird, S. J. Pennycook, J. Sewall, D. Jacobsen, D. N. Asanza, A. Hsu, H. C. Carrillo, H. Kim, and R. Robey, "Effective performance portability," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2018, pp. 24–36, doi: 10.1109/P3HPC.2018.00006.
12. J. D. Sewall and G. C. de Verdière, "Chapter 2 - From "correct" to "correct & efficient": A Hydro2D case study with Godunov's scheme," in *High Performance Parallelism Pearls*, J. Reinders and J. Jeffers, Eds. Boston: Morgan Kaufmann, 2015, pp. 7–42, doi: 10.1016/B978-0-12-802118-7.00002-9.

John Pennycook is an HPC Application Engineer at Intel Corporation. He received a Ph.D. in Computer Science from the University of Warwick in 2013. His research is focused on improving application performance portability and programmer productivity. Contact him at john.pennycook@intel.com.

Jason Sewall is a Senior HPC Application Engineer at Intel Corporation. He received a Ph.D. in Computer Science from the University of North Carolina at Chapel Hill in 2010. Jason's research interests include parallel algorithms, numerical analysis, and practices for improving performance, portability, and productivity in software. He can be reached at jason.sewall@intel.com.

Douglas Jacobsen is now a Senior HPC Software Engineer at Google LLC (previously at Intel Corporation). He received a Ph.D. in Scientific Computing from Florida State University in 2011. His research is focused on improving and understanding performance of HPC applications. He can be reached at dwjacobsen@google.com.

Tom Deakin is a Senior Research Associate at the University of Bristol. He received a Ph.D. in Computer Science from the University of Bristol in 2018. His research focuses on measuring and attaining the highest possible performance on existing and future many-core processors in a performance portable way. Contact him at tom.deakin@bristol.ac.uk.

Simon McIntosh-Smith is a Professor of High Performance Computing at the University of Bristol, where he leads the HPC research group. His research interests include simulating advanced computer architectures, and performance portability across diverse computer systems. He can be reached at S.McIntosh-Smith@bristol.ac.uk.