Effective Performance Portability

Stephen Lien Harrell[†*], Joy Kitson[‡*], Robert Bird[*x], Simon John Pennycook[§], Jason Sewall[§], Doug Jacobsen[§], David Neill Asanza[¶*], Abigail Hsu[‖*], Hector Carrillo Cabada[††*], Heesoo Kim[‡‡*], and Robert Robey[*x]
[*] Los Alamos National Laboratory
[x] Email: {bird, brobey}@lanl.gov
[†] Purdue University, Email: sharrell@purdue.edu
[‡] University of Delaware, Email: tkitson@udel.edu
[§] Intel Corporation, Email: {john.pennycook, jason.sewall, douglas.w.jacobsen}@intel.com
[¶] Grinnell College, Email: neillasa@grinnell.edu
[‖] Stonybrook University, Email: abigail.hsu@stonybrook.edu
[††] University of New Mexico, Email: hcarrillo@unm.edu
[‡‡] Brown University, Email: heesoo_kim@brown.edu

***Abstract*—Exascale computing brings with it diverse machine architectures and programming approaches which challenge application developers. Applications need to perform well on a wide range of architectures while simultaneously minimizing development and maintenance overheads. In order to alleviate these costs, developers have begun leveraging portability frameworks to maximize both the code shared between platforms and the performance of the application. We explore the effectiveness of several such frameworks through applying them to small production codes. Throughout the process, we apply a logging tool to gather data on the development process. We use this information to develop metrics of application development productivity, which can be used to holistically assess how productively a performance-portable application was developed.**

***Index Terms*—productivity, developer metrics, performance, portability, HPC, VPIC, Kokkos**

I. Introduction

Computing is intrinsically based on resource constraints; the same holds for the scientific fields that build upon computation. In the past, performance on a single system was the primary metric used to measure how efficiently a scientific application used available resources. However, as the range of hardware found in modern supercomputers increases, so too does the importance of having application code that can effectively make use of different underlying compute hardware. Hardware is not the only resource of concern, however: the process of porting applications to new platforms can be incredibly expensive in terms of both time and effort, not just in initial costs to produce functional code, but also in terms of maintenance costs that are required over time in support of the code. If not done well, ports can vastly reduce developer productivity by forcing them to maintain several divergent versions of an application, as shown in Fig. 1(a). Thus, to truly capture the efficiency of an application, we need a holistic metric that captures not only the performance of an application, but also its portability across different platforms and how productive developers are as they work on the underlying code for the application.

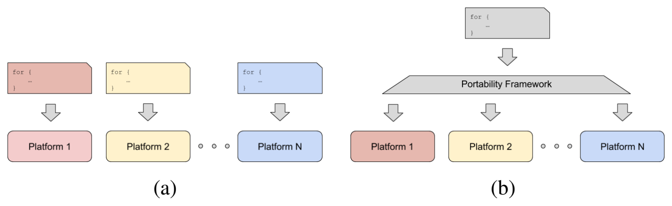

Fig. 1: Portability frameworks reduce the number of platform-specific codepaths.

Hardware Diversity Computing hardware has evolved in many dimensions as hardware manufacturers have sought to improve computing efficiency as well as peak levels of performance and address increasingly diverse needs. This has had dramatic effects on software ecosystems.

Large-scale systems consisting of "traditional" CPUs are still popular, but so too are systems that combine CPUs with accelerators [1]. These accelerators are frequently throughput-oriented GPUs, and there is growing interest in custom silicon—such as that found in the Sunway TaihuLight—and in FPGAs. The CPUs themselves are variegated from earlier iterations, with dramatic increases in compute density in the form of vector units, superscalar execution, and many cores on a single die.

While porting and performance optimization has always been a formidable undertaking, the myriad of hardware options present a bigger challenge than ever before. In addition to the question of what implementation achieves the best efficiency for a given piece of hardware, it has become increasingly difficult to reconcile these manifold codes.

System purchasers, in light of these trends, are keenly aware of the possibility of "vendor lock-in"; that all of their codes will be deeply adapted to a narrow set of hardware and unable to effectively utilize new platforms save at great expense.

Programming Models These concerns have led to demand for programming models and frameworks that allow for a single-source solution to be run on multiple hardware backends, as in Fig. 1(b). Examples of these efforts include: OpenMP* (4.5) [2], OpenACC* [3], OpenCL* [4], HIP [5], Kokkos [6], RAJA [7], and FleCSI [8]. While the claims of these individual efforts differ, the overall themes are the same: the desire to achieve both *performance* and *portability* with a single code base, and to do so *productively*.

Contributions In this work, we introduce a candidate methodology for tracking the combination of these "three Ps" during application development. Specifically:

1) We discuss a number of metrics for assessing performance, portability and developer productivity (PPP), highlighting the significant amount of data required to compute them.
2) We develop a candidate methodology and associated tools for tracking application performance and portability alongside developer productivity.
3) We evaluate our methodology by applying it to the development of performance-portable implementations of production codes: VPIC, Truchas and SpectralBTE.

II. Related Work

The DARPA High Productivity Computer Systems program [9] addressed many aspects of high performance computing (HPC), including programmer productivity. While it introduced important concerns specific to the HPC community, at the time it dealt with ports to the Cell processor and was only beginning to consider GPUs as general-purpose computation devices. Many of the languages and tools available today did not exist or were in their infancy. Kepner [10] compared productivity for MPI, OpenMP, HPF and Java* implementations of the NAS parallel benchmarks and found that MPI required 70% more lines than the serial baseline while HPF and OpenMP only required 10% more. In other representative works at the time, Funk et al. [11], [12] present a relative productivity metric targeted specifically at parallel code development.

Today, these concerns have shifted to code-bases that may be created and maintained with performance portability across diverse architectures; this was the focus of the DOE Centers of Excellence Performance Meeting in 2016 [13].

In the area of performance portability, a number of previous efforts have evaluated different programming languages for the development of performance-portable applications [14], [15], [16]. Studies such as these often draw positive conclusions about the ability to achieve portability across different hardware designs, but lack a formal methodology for evaluating their success. While no definition for performance portability has yet been widely accepted, our previous work [17] attempted to develop a shared lexicon in order to foster effective discussion on the topic. This paper builds upon this previous work, examining how the concept of performance portability relates to productivity.

Wienke et al. have published several studies [18] of productivity in HPC, examining both the relationship between developer effort and performance and the relationship between a system's total cost of ownership and the number of applications it will run in its lifetime [19]. The EffortLog [20] tool produced as part of their work collects performance and effort information from application developers at regular intervals, similarly to the tool presented here: the key differences are our focus on applications targeting multiple architectures, and tighter integration of effort logging into developer workflows (via git hooks).

The *ninja gap* discussed by Satish el al. [21] embodies the different levels of performance achieved by traditional languages that predate widespread parallel hardware as compared to code written in languages augmented by libraries/frameworks that take advantage of parallelism. Performance results are compared against a baseline peak performance achievable by an expert ("ninja"). Productivity is qualitatively presented with example code as opposed to quantitative metrics such as source lines of code (SLOC).

The breadth of ways to even count lines of code in a meaningful way has been the subject of research; Nguyen et al. [22] present a standardized method for counting lines of code, giving examples for Perl, Javascript, and SQL. They offer justifications for each choice and introduce precedence rules for determining how constructs with multi-line physical representations may be mapped to "logical" lines of code.

III. Background

A. Programming for Performance, Portability & Productivity

Development teams looking to write performance-portable codes find themselves with several options, each a trade-off between performance, portability and productivity. Intuitively, codes written to a high-level abstraction are likely to be easier to develop and to move between platforms, but may achieve a lower level of absolute performance than an implementation highly optimized for a single platform. Conversely, codes written to a low-level abstraction (or to directly target a specific machine) may be able to achieve a very high level of absolute performance, but at the cost of portability and developer productivity.

Striking the right balance between the 'three Ps' depends on the development team's goals and the end-use of the code – for example, a standalone application intended to be run only a few times on a fixed dataset has very different requirements than an (optimized) library intended to be integrated into multiple scientific codes. Other influencing factors include the code's size/complexity and the development team's familiarity with different programming languages.

Novel codes unburdened by expectant users and an existing code base may choose to develop (or prototype) the new code at a high level of abstraction, facilitating the rapid exploration of different algorithms and providing portability across different hardware platforms from the outset. In some domains, domain-specific languages (or *DSL*s, e.g. Halide [23], OP2 [24], Devito [25]) provide a natural way for scientists to express applications without consideration of the underlying hardware; in other domains, it is possible to program to abstract representations of parallel machines using a framework

Programming Model	**C**	**C++**	**Fortran**
OpenMP	✓	✓	✓
OpenACC	✓	✓	✓
OpenCL	✓	✓	
Kokkos		✓	
RAJA		✓	
SYCL		✓	

TABLE I: Programming model and language compatibility.

(e.g. Kokkos [6], RAJA [7], FleCSI [8], SYCL [26], HIP [5]) or a modern parallel language (e.g. OpenCL [4], C++17).

Complete rewrites of code in a new language or framework may not be desirable or realistic for developers looking to port large existing codes with many users, and incremental porting of legacy codes may be complicated by language compatibility issues (e.g. between Fortran and C++-based frameworks, see Table I). Such developers are more likely to adopt a directive-based solution (e.g. OpenMP [2], OpenACC [3]) that can be more easily integrated into the existing code base. To initiate a discussion on these topics, we begin by defining the terminology and metrics needed to objectively measure the development of a scientific application in the exascale era.

B. Performance Portability

We take our definition of performance portability (PP) from Pennycook, et al. [17] as "A measurement of an application's performance efficiency for a given problem that can be executed correctly on all platforms in a given set.", along with the metric to quantify PP using the harmonic mean as shown in Equation (1).

$$\Phi(a,p,H) = \begin{cases} \dfrac{|H|}{\sum_{i \in H} \dfrac{1}{e_i(a,p)}}, & \begin{matrix}\text{if } i \text{ is supported} \\ \forall i \in H\end{matrix} \\ 0, & \text{otherwise} \end{cases} \quad 0 \tag{1}$$

This equation states that on a given set of platforms, H, the Performance Portability, $\Phi(a,p,H)$, of an application a running a problem p is the harmonic mean of the performance efficiencies $e_i(a,p)$ on each platform i. $\Phi(a,p,H)$ is 0, if any platform in H is unsupported by a running p.

A common criticism of this definition is that it does not penalize "heroic" development efforts for individual platforms: it is possible (and perhaps necessary) to maintain completely separate highly-optimized code paths for each platform in order to maximize PP. Although the definition is useful for assessing an application, it does not address the desire to reason about which of the myriad approaches to developing performance portable code is most effective. For this, we need to have some ways of measuring the effort expended in achieving performance across different architectures.

C. Productivity and Effectiveness

The definition of productivity used by Wienke [27] is:

$$\text{productivity} = \frac{\text{output}}{\text{input}}$$

This ratio is intuitive, and allows for a single definition/metric to be used to measure productivity in different contexts: an individual's productivity may be measured as the ratio of application performance to development effort; while an HPC site's productivity may be measured as the ratio of scientific output to monetary cost. A more specific measure of productivity is *relative development time productivity (RDTP)*, introduced by Funk et al. [12] for parallel code development and shown in Equation (2).

$$\Psi_{\text{relative}} = \frac{\text{speedup}}{\text{relative effort}} \tag{2}$$

Calculating productivity at a fixed point in time this way is useful, but using it to compare different development approaches may be misleading. The relationship between input and output is unlikely to be a linear function, so projections (e.g. the level of performance reached for a fixed effort) given a productivity score are likely to be inaccurate.

D. Developer Effort

The measurement of effort can be broken into two different approaches: direct methods based on effort logging; and indirect methods that approximate effort from other quantities (e.g. lines of code or number of code changes).

The direct methods require substantial involvement from developers and often a subjective input from them. They require either a daily "diary" entry as in EffortLog [20] or detailed input for each commit.

The indirect methods measure input effort through the number of added SLOC. For example, Wheeler's SLOCCount [28] converts the objective measure of SLOC from an abstract number into more useful values such as man-months and dollars using the Constructive Cost Model (COCOMO) [29] or even a custom estimation formula.

Function points have also emerged as an alternative measure to SLOC [30], but have not seen much use in scientific codes. Simply, it is difficult to break up the core of a scientific application into small functional operations. This approach may still have some value for higher-level functionality of a code (e.g. operations related to input/output).

Often, just SLOC or function points do not adequately describe the process of developing a highly-parallel code. A more detailed view can be obtained by looking at the lines added and removed through the use of diff utilities. These utilities first appeared in the mid-1970s and were formalized by Hunt and MacIlroy [31]. Many variants have since emerged and have become intertwined with code revision systems.

IV. Measuring Performance, Portability and Productivity

Developing PPP codes rests on a paradox: the application must be specific enough to take advantage of the peculiarities of each system it runs on while being broad enough to do so on all of them, and it must accomplish this in a way that does not significantly reduce developer productivity.

One potential solution is to implement separate code paths for critical functions on all platforms. However, this increases maintenance overhead with every platform added, as developers are forced to fix every bug and implement every feature separately for each platform. An alternative solution is to integrate optimizations for a new platform into an existing "single-source" code base. This decreases maintenance costs long-term, but having to maintain compatibility with the myriad other platforms already supported may make initial porting and optimization more difficult and time consuming.

It is difficult to say which of these (or other) approaches delivers the highest performance portability for the lowest *total* effort. In the remainder of this section, we detail a number of metrics that we believe may provide insight into this relationship.

A. Platform Divergence and Maintenance Cost

Before an application can be optimized for a particular platform, it must first be capable of producing valid answers to relevant problems on that platform. This *porting* process can take several forms: rewriting the application in a different language or framework that supports the platform (e.g. CUDA* for NVIDIA* GPUs); adding support for the new platform to some underlying framework (e.g. Kokkos); or simply debugging the application, such as when the language in use already supports the new platform but the application has not yet been tested.

Whatever the approach, a port requires modifying an existing application that solves a given problem, p, on a given platform, h, so that it solves p on a new platform, h'. Thus a port requires applying a transformation, defined in terms of both the tool(s) used in the port and the techniques which define their use, t, to the code base: $t : A_{p,h} \to A_{p,h'}$, where $A_{p,h}$ and $A_{p,h'}$ are the set of applications that solve p on platform h and h', respectively.

In order to compare applications, it is useful to have a notion of *distance* between them: $d : A_p \times A_p \to \mathbb{R}_{\geq 0}$, where A_p is the set of all applications which solve p. In particular, d should define a metric on A_p, i.e. it should be symmetric, should come out to zero for identical applications, and satisfy the triangle inequality.

For a given set of applications, $A \subset A_p$, we can extend d to give a measure of how different they are from each other. We call this measure of *code divergence* D:

$$D(A) = \binom{|A|}{2}^{-1} \sum_{\{a_i,a_j\}\subset A} d(a_i, a_j) \quad (3)$$

Thus $D(A)$ is the average of the pairwise distances between all of the applications in A. The set of current ports is used for the divergence because we are looking at the cost of maintaining these different ports. On the other hand, the distance from the original application, whether serial or parallel, is important to understand the cost of porting an application.

As an example, we use the change in the number of source lines of code, normalized to the size of the smaller application:

$$d(a_{t_1,h_1}, a_{t_2,h_2}) = \frac{|\text{SLOC}(a_{t_1,h_1}) - \text{SLOC}(a_{t_2,h_2})|}{\min(\text{SLOC}(a_{t_1,h_1}), \text{SLOC}(a_{t_2,h_2}))} \quad (4)$$

Minimizing the distance between the code paths for each supported platform reflects the typical motivation behind "single source" programming frameworks like Kokkos.

B. Development Cost

Another common area of interest is the cost associated with porting an application to a target platform, expressed as:

$$c_{d,t}(a, a_{t,h}) = \alpha_{d,t}(d(a, a_{t,h})) \quad (5)$$

where $\alpha_{d,t}$ represents a transformation-dependent conversion function that converts distance into cost.

We extend this idea to describe the cost of porting an application to a set of platforms, H:

$$C_{d,t}(a, A_H) = \alpha_{d,t}\left(\frac{1}{|A_H|}\sum_{a_h\in A_H} d(a, a_h)\right) + \beta_{d,t}\left(D(A_H), |A_H|\right) \quad (6)$$

where A_H is the set of ports to the platforms in H, $\alpha_{d,t}$ is the same as in Equation (5), and $\beta_{d,t}$, like $\alpha_{d,t}$, is a function which relates distance to cost, although $\beta_{d,t}$ is also impacted by the number of ports (i.e. the cardinality of A_H, as seen above). This encapsulates the idea that the cost of the port depends not just on how far the ported versions are from the original version, but how far they are from each other. It should be noted that $\beta_{d,t}$ is not necessarily monotonically increasing. In fact, for many transformations there is expected to be a significant cost to minimizing the code divergence, especially for large sets of platforms. We anticipate transformations which use PP frameworks well to have an associated $\beta_{d,t}$ which allows low divergence ports to be much less expensive. The initial cost is typically the amount of developer time required to complete the port. Much like distance, this is often thought of in terms of lines of code, especially when the intent is to *predict* the cost of a project. One software cost estimation method is that used by the COCOMO II project:

$$\text{effort [person-months]} = A \cdot M \cdot (\text{SIZE})^F \quad (7)$$

where $M = \prod_{i=1}^{h} EM$ (each EM is an effort multiplier), $F = B + 0.01 \cdot \Sigma_{j=1}^{5}\text{SF}_j$ (each SF is a scale factor), SIZE is in thousands of lines of code, and A, B are calibration constants [18].

The constants for common programming languages have been gathered from studies of developer effort on various projects, but constants for parallel programming languages and code transformations are not established. This is one area in which we hope development logging can make a contribution.

C. Application Performance

The final consideration that developers are usually concerned with is application performance. This is best understood in terms of how well the version of the application uses the available hardware to solve a problem; recalling Equation (1), Φ provides a means of combining these *performance efficiencies* into a single value.

Productively carrying out performance portable ports of an application to a set of platforms H amounts to maximizing Equation (1) while minimizing Equations (3) and (6). That is, in porting an application, we need to find which techniques allow us to minimize both the total cost of porting the application and the divergence of the ported versions, while maximizing performance portability.

D. Churn

In highly-parallel computing application development, it is common that changes result in very small increases in lines of code, but lots of addition and removals of lines to accomplish the result. We define a measure called *churn* to quantify this:

$$\text{churn} = \frac{\text{\# lines add/del.}}{l}, l = \begin{cases} 1, & \Delta\text{SLOC} = 0 \\ \Delta\text{SLOC}, & \Delta\text{SLOC} \neq 0 \end{cases} \quad (8)$$

It should be noted that the sign of churn is only positive when $\Delta\text{SLOC} = 0$ by convention.

There could be many variants of churn that try to account for actual "lines edited" or to avoid counting white space or changes to comments. In tracking churn, we hope to capture the refactoring process as well as simply moving directives around or optimizing MPI communication. We also want to differentiate between development approaches that are verbose but productive and those that are compact but difficult to use.

V. Data Collection Methodology

One goal of this project was to allow software developers the ability to capture and store data related to productivity and performance in a way that satisfied the following requirements:

1) Easily parseable.
2) Separable from the actual software source code.
3) Traversable in relation to software development process.

These requirements allow an analysis system to extract information over arbitrary development windows that can define small feature developments, or larger project milestones. Additionally, it allows productivity and performance logs to be shared separately from the actual source code (assuming access controls are different for each of them). Finally, we can put together tools to aggregate the information into productivity statistics.

We also had two overarching concerns regarding the data collection itself:

1) How to keep the data as objective as possible; and
2) How to keep the process as painless as possible.

Prior efforts to collect productivity data have been plagued by subjectivity; the notion itself is so nebulous as it is hard to capture in the first place, let alone capture in a consistent, reproducible, and credible manner. Above all else, this is due to the necessity of accounting for the human factor in the data. Ideally, we would minimize that factor as much as possible. Additionally, keeping the process as painless as possible helps to improve the quality of data by reducing user frustration; the fewer questions they have to answer, the less frustrated they will be; while the more frustrated respondents get, the lower the quality of their answers.

A. Automation

There is a lot of relevant data that can collected automatically during development: files modified, lines changed, branch name, time stamps between commits, and more. Collecting such data automatically will likely be far more consistent (and objective) than anything a user could provide; they might forget a file that they edited, misspell a branch name, or underestimate development time.

B. Developer Surveys

Free-form responses to developer surveys pose inherent problems. First, it makes the data hard to analyze, likely requiring some form of postprocessing by either humans or sophisticated natural language interpretation tools. Second, it makes the data highly subjective and unlikely to be comparable between different respondents. To help avoid these issues, our tools present several clear-cut questions with a clearly-delimited set of answers. This ensures that the same areas of interest will be covered in every record, while also making the answers simple enough to be analyzed easily.

Of course, different types of answers are appropriate for different questions. For questions where the desired information is described well by a spectrum between two extremes, we use a 7 point scale as set by NASA's precedent [32]. For other types of questions, we restrict answers to a simple 'yes'/'no', or to a single value (e.g. a length of time) requested in specific units in order to keep answers unambiguous.

There are two methods to impose such specific restrictions on the form and content of responses. One is validating the entire log entry after all the responses are completed, and rejecting any entries that do not fit the desired format. Unfortunately, this is cumbersome at best and likely to irritate users. The other option is to validate on a per-question basis, immediately after respondents submit each answer. This has the benefit of offering real-time feedback, making it easier to correct answers, avoid future mistakes, and overall make the process less frustrating. However, the ease of use provided by immediate validation comes at the cost of maintaining an interactive system to provide that feedback. Relative to the danger of corrupted or incomplete data, though, this cost is easily managed.

C. Work-flow Integration

In order to minimize the burden on respondents, we chose to embed our interactive logging tool into a developer's normal workflow as seamlessly as possible. Since we sought to record data about the development process, this meant that we wanted to tie our tools to commits in the revision control system. Ideally, the developer wouldn't even need to remember to use our tools; they would be automatically triggered whenever someone commits work on a project that uses our tools.

When entering productivity information, the first prompt asks the user whether or not they want to answer questions for the effort log for the commit. This serves two main purposes. First, not all commits represent a milestone that a developer would like to capture productivity information for. While we could prescribe which commits require human input ourselves, the developers are in a far better position to evaluate the importance of any given commit, and so we leave the choice to them. Second, by asking the user if they want to answer questions for a commit, we avoid the irritation of forcing them to answer questions they feel are unnecessary. This choice to have our users consciously volunteer their time to aid in our logging is intended to make it seem minimally invasive, and has no bearing on data that are collected automatically.

Once the collection process is complete, a log file is generated in a parseable format, and connected to each commit using git-notes. This allows the productivity log to track to the git history but remain unobtrusive to the developer. Additionally, this provides a mechanism to easily share logs across multiple machines, update logs as new information is collected (e.g. performance tests on a new platform are executed), extract information separately from the actual source code, and interact with the collected data using existing git tools.

D. Productivity

There are two different ways that we collect productivity data: some automatically and others based on survey answers provided by developers. The automatic collection is focused on the files changed – including both files added to the git tree and those not added – and captures the name, size and last modified attributes of each file. This data can give us insight into the total workload of a commit, regardless of whether the work is represented in the git tree.

NASA TLX perceived workload

- mental demand
- physical demand (omitted)
- temporal demand
- overall performance
- effort
- frustration

Fig. 2: The perceived workload from the NASA TLX survey

The other data are generated from survey responses during the git commit hook. The survey responses capture: the type(s) of activity performed (planning, coding, refactoring, debugging or optimizing); the hours spent on each type of activity; what programming language(s) or framework(s) were used; and a self-reported difficulty metric. The difficulty metric is derived from a modified NASA TLX [32] survey which gauges the perceived workload for each commit; the components of the perceived workload are shown in Fig. 2.

E. Performance

Whether performance data is generated online (e.g. as part of a test carried out before making a commit) or offline (e.g. as part of an automated nightly build system), it is always associated with a particular commit. Combining collected performance data with collected productivity data enables us to track derived metrics such as performance as a function of the hours of development time, or performance as a function of change in the number of lines of code.

VI. Case Studies

During the yearly Parallel Computing Research Summer Internship at Los Alamos National Laboratory [33] teams of students work to optimize and parallelize real scientific codes. During this year's internship three teams volunteered to report their performance, portability and productivity metrics. Each team took a different approach to a different problem and together they give an outline of what can be shown by looking at these metrics. It is important to note the outcomes of each case study should not be generalized, but the tools and methods used are widely applicable.

The reader is reminded that the focus of this paper is on *effort* and *productivity*; the codes detailed here are each at an early stage of development, and each student team was focused on evaluating different languages and platforms. The performance numbers in this section are often not reflective of the peak performance achievable on any one platform, but this strengthens our argument for tracking PPP data throughout development and reacting accordingly.

A. Case: VPIC

VPIC is a particle-in-cell (PIC) plasma physics model that has traditionally used hand-tuned intrinsics to achieve high levels of performance. The code tracks particles and electric and magnetic fields through a structured grid according to basic principles. VPIC is also relatively small for a production code, with only around 40,000 - 60,0000 lines. The application runs at large scales and on many CPU platforms, operating with upwards of 2 million MPI ranks and 7 trillion particles [34], as well as leveraging threads. Despite this, VPIC currently lacks a method to offload work to GPUs.

Together, these characteristics make it a perfect candidate to port to a PP framework. The current scalability and the range of systems that VPIC can run on provide a good baseline for comparison; if a VPIC port can compare to the mainline version, we will know that the overhead involved in the framework is not a real obstruction to getting good performance out of a code. The small size of VPIC makes a port much more feasible, reducing the required time from years to months, while still offering interesting comparisons to other production codes. The lack of GPU support is a clear motivation to explore portability frameworks, and to explore possible trade-offs between portability and performance on individual platforms.

For the portability framework we chose to use Kokkos [6], a C++ library developed at Sandia National Labs that can compile down to a variety of backends. We use Kokkos' OpenMP and CUDA backends, allowing it to switch between CPU and GPU builds at compile time. Kokkos is amongst the most mature of the PP frameworks, and offers advanced features such as scatter-add views and automatic device-aware particle sorting.

For the case study, the performance portability and the effort (productivity) of porting VPIC into Kokkos was measured. Two kernels are converted, `advance_p` and `advance_b` (which handles the magnetic field updates), these are analyzed

TABLE II: Application efficiency of the `advance_b` and `advance_p` Kernels in VPIC

	`advance_b`		**`advance_p`**	
Platform [1]	**Original**	**Kokkos**	**Original**	**Kokkos**
Intel® Xeon® 8176 Processor	100%	13%	100%	62%
IBM* Power 9	100%	32%	100%	46%
Cavium* ThunderX2	100%	54%	100%	50%
NVIDIA* V100	0%	100%	0%	100%

TABLE III: Architectural efficiency of the `advance_b` and `advance_p` Kernels in VPIC

	`advance_b`		**`advance_p`**	
Platform [1]	**Original**	**Kokkos**	**Original**	**Kokkos**
Intel® Xeon® 8176 Processor	12%	2%	18%	11%
IBM* Power 9	14%	5%	8%	4%
Cavium* ThunderX2	11%	6%	10%	5%
NVIDIA* V100	0%	93%	0%	5%

[1] Hardware and software configurations available in Appendix A

in Table II. Further, we note that as part of this effort we developed an initial port of 5 of the 11 core kernels, while eliminating the need for a direct implementation for 3 kernels which are related to handling replicated data – a direct CPU optimization with Kokkos has the functionality to replicate. Performance analysis of these kernels is omitted for brevity, and will be covered in a future study.

It is important to note that the code used on each platform is the portable code variant and has not received hand optimization for the given platform. We expect that with tuning the performance on a given platform could be significantly increased, particularly in the case of vectorization. This performance tuning will be addressed directly in future work.

As seen in Table II, VPIC was originally portable on 3 of the 4 target platforms. When application efficencies are compared across code bases on the original set of platforms using the harmonic mean, Φ is 100% which tells us that the original code base is more optimized for CPU than the Kokkos version. However, when the NVIDIA V100 platform is introduced the Φ of all the platforms in the original code base goes to 0. The Φ for the Kokkos port of the `advance_b` and `advance_p` kernels are 29% and 59% respectively.

For the calculation of architectural efficiency in Table III we compare to number of bytes moved to the maximum achievable memory bandwidth. This is only entirely representative of kernels which are entirely memory bound, but serves as a good first order approximation of our expected performance bound. The efficiencies in Table III show that the Φ on the CPU platforms solely (`advance_p`: 10%) is higher on the original code base than the Kokkos port (`advance_p`: 5%), but as before we see that once you add the the GPU platform

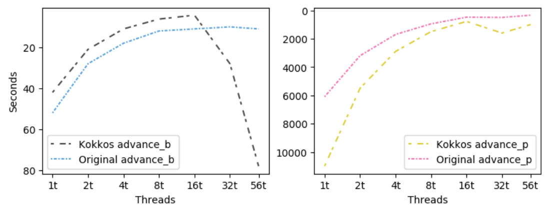

Fig. 3: Thread Scaling of VPIC[1]

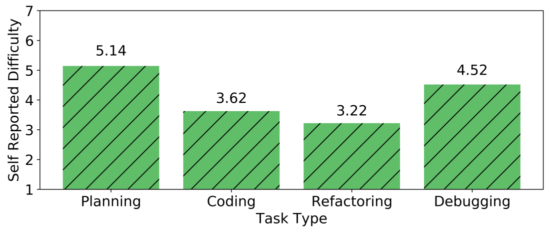

Fig. 4: Difficulty Per Task for VPIC Team

the Φ of the original code base is 0. For the Kokkos version of advance_b and advance_p the Φ is 4% and 5%.

It is interesting to note that the Kokkos version increases performance-portability while maintaining a low-divergence single-source solution which could also offer support for future architectures. The Kokkos port of VPIC has a two line difference between the GPU and CPU versions, from a total line count of 18198. When using the distance metric in Equation (4) the divergence of these two code paths is 0.01%, with the expected maintenance cost being very close to the cost of a single code path.

During the port of VPIC to Kokkos, we tracked development progress with productivity monitoring tools. As seen in Fig. 4, Planning and Debugging were found to be some of the hardest tasks, with Coding and Refactoring being some of the easiest. The rate of code change over time with specific milestones can be seen in Fig. 5. We can see the rate of change holding steady before the completion of `advance_b` and a slowly increasing rate until a few smaller kernels were converted and a well-understood pattern is completed. The rate is then fairly constant again until right before the completion of `advance_p`, the most complex kernel completed.

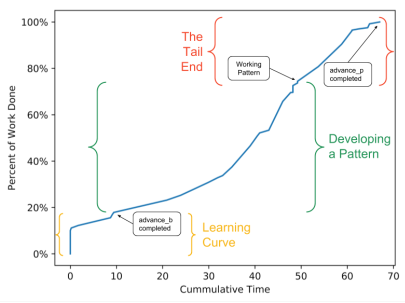

Fig. 5: Completion of `advance_b` and `advance_p` over time

B. Case: Truchas

As a second case study, we optimized two computational kernels from the open source Truchas code [35]. Truchas is a 3D multi-physics simulation tool developed by Los Alamos National Laboratory for metal casting and other applications, and includes physics models for heat transfer, phase change, incompressible free-surface fluid flow, and several others. It uses unstructured meshes for modeling complex geometries and uses finite volume, finite element, and mimetic finite difference spatial discretizations. Truchas is also written in modern Fortran, making heavy use of object-oriented language features introduced in the Fortran 2003 standard.

The performance portability study looked at porting key computational kernels to OpenMP CPU, OpenMP GPU offload, and CUDA. The kernel shown here is the mimetic finite difference kernel, essentially a stencil operation on an unstructured mesh that preserves important geometric properties. OpenMP on CPU is a directive-based API for multi-threaded parallel processing on shared-memory multi-processor (core) computers. OpenMP for the GPU is a set of new directives available in OpenMP 4.0+ enabling execution on offload devices such as GPUs. CUDA (Compute Unified Device Architecture) is a GPU-specific parallel API that runs on NVIDIA hardware.Table IV compares the net line changes and

TABLE IV: Effort Summary of Mimetic Finite Difference kernel

Approaches	Time to Adopt (in Hours)	Net Line Changes	Cumulative Frustration
OpenMP CPU	18	28	8
OpenMP GPU	21	151	10
CUDA GPU	41	284	12

implementation hours across the three approaches, and shows that CUDA requires the largest number of line changes, incurs the highest frustration level, and involved the most developer hours for planning, implementing and debugging. Meanwhile, OpenMP CPU requires fewer line changes, and relatively little effort in terms of developer hours and frustration. The divergence metric in Equation (3) for all three ports and the combinations of two ports that cover all the architectures is shown in Table VI. The distance between ports is determined using a *git diff* between branches established for the port and then divided by a line count in the source part of the repository. Because there is a lot of mesh setup code, the percentage expressed in the divergence metric is relatively small. The results show that the divergence of an OpenMP CPU and GPU approach is smaller and will likely have lower maintenance costs, approaching the goal of a single-source ideal for a Fortran code. This must be weighed against the performance results of each port and whether the project goals stress maintenance or performance.

TABLE V: Distance matrix comparing three ports with different languages against each other.

	OpenMP CPU	OpenMP GPU	CUDA
OpenMP CPU	0	148	503
OpenMP GPU	148	0	357
CUDA	503	357	0

TABLE VI: Code divergence metric evaluated using different maintenance options from Table V

Ports to maintain	Divergence
OpenMP CPU & OpenMP GPU	2.02%
OpenMP CPU & CUDA	6.86%
OpenMP CPU, OpenMP GPU, & CUDA	4.58%

For the same kernel, the performance portability metric was calculated based on Architectural Efficiency, relative to STREAM bandwidth from main memory, as shown in Table VII. If we calculate the metric for the two subsets as in Table VI, we get 6.94% and 5.85% respectively. The efficiency of the OpenMP code is low on both the CPU and GPU platforms, highlighting a need to optimize it further; future work will explore improvements to the code's non-contiguous memory access pattern, which are expected to require fewer changes (in terms of lines of code) than were needed by the CUDA port.

TABLE VII: Performance portability based on architectural efficiency for the Mimetic Finite Difference Kernel

Platform[1]	Version	Arch. Eff.
Intel® Xeon® E5-2698 processor	OpenMP CPU	4.19%
Intel® Xeon Phi™ 7250 processor	OpenMP CPU	7.76%
Intel® Xeon® Platinum 8176 processor	OpenMP CPU	5.49%
Power9	OpenMP CPU	3.44%
Volta + Power9	OpenMP GPU	5.41%
Volta + Power9	CUDA	77.71%
Volta + Intel® Xeon® E5-2683 processor	CUDA	90.04%
Performance Portability	6.7%	

[1] Hardware and software configurations available in Appendix B

C. Case: Spectral BTE

In this case study, we focused on making the simulation of the Boltzmann Transport Equation (BTE) faster and scalable. The simulation of BTE is used to model molecules that are not in equilibrium and has applications in hypersonic flows, fluid micro-flows, plasma physics. This code uses a spectral method to compute a large sum solving for the collision operator of the BTE. This sum contains a constant weight term that scales as O(N^6), where N is the number of velocity grid points. A large N is needed for higher accuracy but this can result in a weight term that is gigabytes in size.

The existing version of the code was parallelized using MPI and OpenMP. In this implementation, grid points in physical space are evenly distributed across MPI ranks. This requires every individual MPI rank to hold its own copy of the convolutional weights, limiting both the number of MPI ranks that can be placed on a single node and the maximum N that we can solve for to the amount of memory on a single computing node. We attempted to remove this bottleneck by first refactoring the existing code and then reconfiguring and implementing two different workflows.

The refactoring step of this process involved the deletion of dead code and turning repeated code into functions. This resulted in many lines of code added (i.e. turning code into functions) and removed (i.e. deleting dead code and removing repeated code) and the decrease of SLOC. This is captured by the churn of -3.47.

In one of the implementations, we restructured the code in such a way that the MPI ranks work on solving portions of collision operator for every spatial grid point instead of solving for it entirely for only a couple of spatial grid points. This allows MPI ranks to hold only a fraction of the convolutional weights. Because of these changes, much of the MPI communication and related code in the original implementation was removed. Additionally, temporary code was added to support the new workflow as well as debugging. This code was later deleted or reworked and moved to more relevant files. Overall, the new implementation saw a slight increase in lines of code over the refactored code but a large number of changes in terms of addition, deletion, and movement of code, resulting in the computed churn of 25.60.

The other implementation involved forgoing the precomputation of the convolutional weights and instead recomputing the weights at each time step on the GPU using CUDA. This implementation gets rid of the problem of weights storage. The computation of the convolutional weights are series of big sums that are well-suited for porting to GPU, but a problem of code replication arises because the GPU cannot access host (CPU) memory or instructional code and vice versa. Thus, porting the desired code over to CUDA initially required copying and pasting large chunks of code with small changes to make them suitable for CUDA, which resulted in a large net gain in line numbers with little effort. However, later changes to the code were made more difficult as the result of this code duplication, since the same changes needed to be replicated across both CPU and GPU versions of the code. As such, the GPU implementation saw a large increase in the lines of code in comparison to the refactored code, but the actual changes in terms of code modification were small, resulting in the computed churn of 6.09.

The churn scores are very different for MPI implementation versus CUDA implementation because while the MPI implementation required a complete change in code workflow (e.g. distributing the convolutional weights rather than the physical space across the nodes) but had a small net change in the line numbers, the CUDA implementation only required small changes regarding the actual code to ensure the code instruction can be executed in GPU but had a large net change in the line numbers due to code duplication in writing for a GPU version.

VII. Discussion

We evaluated the performance portability metric for kernels from two real application codes, using both application efficiency and architectural efficiency. For VPIC, application efficiency gives a comparative view of the Kokkos port relative to other implementations and shows its performance is generally poorer. For Truchas, architectural efficiency identifies an imbalance in the performance achieved on different platforms: the CPU performs worse than the GPU and also worse than expected, suggesting that the OpenMP code is in need of further optimization. Both efficiencies clearly identify focus areas for future optimization efforts, highlighting the value of using our tools to track Φ during development.

The productivity measurement efforts gave mixed results. The identification of churn as a potential measure of interest is confirmed by the work on SpectralBTE, where net lines of code decreased in some aspects of the development work: in cases such as this, reporting final SLOC alone would be misleading. The code divergence metric is similarly confirmed by the work on VPIC, where it accurately represents progress towards the ideal goal of a single-source code (i.e. a divergence of nearly 1). However, the work on Truchas highlights potential difficulties in interpreting the metric; although divergence of the code as a whole is arguably the most relevant metric for any development team, calculating divergence with some standardization of the basis may be necessary in order to compare the result to other efforts or to evaluate a programming approach in a small-scale study (e.g. a single kernel).

Our analysis of the coding approaches in each case study is limited by a lack of data quantifying different aspects of parallelization effort (e.g. hours required to port to OpenMP or MPI). In most cases, there is simply no data and in others there are very limited sample sizes that give a high uncertainty and really no basis to extrapolate far from the original study.

VIII. Conclusions

It is apparent that even basic data on the processes for creating parallel code from a serial code are difficult to find. When the scope is expanded to consider multiple parallelization approaches and the effort expended on each, the absence of this data becomes more pronounced. Without such data, we cannot hope to gain useful insight into the relationship between performance, portability and productivity and how they behave over time for different individuals, code bases and programming languages.

Improving developer understanding of the three Ps is necessary to develop scientific applications in the exascale era. Recent advances in both hardware and software have given developers many more options to consider when deciding what direction their future code development should take, but their resources have not increased accordingly. There is great danger here: going down the wrong path risks creating a degree of

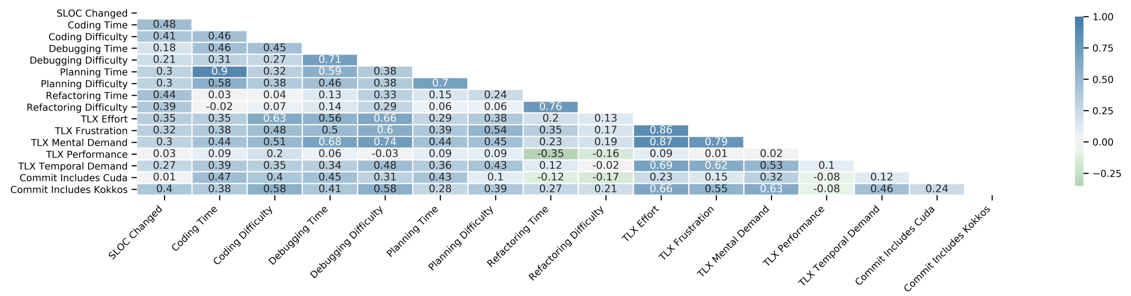

Fig. 6: The cross-correlation matrix of logged data shows some of the comparisons that future studies could make.

technical debt from which it would be difficult to recover, but not going forward at all guarantees that an application's capabilities fall behind. The babel of parallel languages and the lack of portability may have profound consequences for the scientific community and could cause a stagnation in progress if performance portability and productivity are not addressed properly. This is a problem that the whole community needs to address: from hardware designers, vendor software developers through to scientific programmers. Each plays a role in making scientific applications more capable while best utilizing the scarce resources available.

The methodology presented in this paper is an important first step towards understanding and modeling the three Ps. Our tools integrate directly into a developer's existing (git) workflow in order to facilitate the calculation of Φ and complementary productivity metrics (which differentiate between initial development/porting effort and maintenance effort) while minimizing the burden of data collection. We acknowledge that the sample sizes in this work are too small, and the case studies too limited in scope, to make any broad definitive statements on the costs of adopting different parallelization frameworks; however, our results nonetheless demonstrate the value of individual development teams using our tools to guide their optimization efforts.

A. *Future Work*

The work has just begun on trying to quantify software development effort as well as the performance portability of a highly-parallel scientific application. The methodology proposed here enables developers to track their effort and produces metrics that provide a starting point to objectively evaluate parallelization efforts in a variety of ways. There is still a need to deploy the tools and techniques to classroom research, hackathons, and real development environments to gather some fundamental parallel software development data.

IX. Acknowledgements

The authors wish to thank Sandra Wienke and Julian Miller, the attendees of the Dagstuhl seminar on "Performance Portability in Extreme Scale Computing: Metrics, Challenges, Solutions", and the attendees of the Department of Energy Performance Portability workshops, for fruitful discussions about performance, portability and productivity.

Disclaimers

References

[1] V. Kindratenko and P. Trancoso, "Trends in high-performance computing," *Computing in Science & Engineering*, vol. 13, no. 3, pp. 92–95, 2011.

[2] OpenMP Architecture Review Board, "OpenMP application program interface version 4.0," 2013.

[3] OpenACC Working Group and others, "The OpenACC Application Programming Interface Version 2.5," October 2015.

[4] Khronos OpenCL Working Group, *The OpenCL Specification Version 2.2*, March 2016.

[5] AMD, "HIP-datasheet." [Online]. Available: https://www.amd.com/Documents/HIP-Datasheet.pdf

[6] 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, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[7] R. D. Hornung and J. A. Keasler, “The RAJA portability layer: overview and status,” Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep. LLNL-TR-661403, September 2014.

[8] B. Bergen, N. Moss, and M. R. J. Charest, “Flexible Computer Science Infrastructure (FleCSI), version 00,” 4 2016. [Online]. Available: https://www.osti.gov//servlets/purl/1311634

[9] J. Dongarra, R. Graybill, W. Harrod, R. Lucas, E. Lusk, P. Luszczek, J. McMahon, A. Snavely, J. Vetter, K. Yelick *et al.*, “DARPA’s HPCS program: History, models, tools, languages,” in *Advances in Computers*. Elsevier, 2008, vol. 72, pp. 1–100.

[10] J. Kepner, “High performance computing productivity model synthesis,” *The International Journal of High Performance Computing Applications*, vol. 18, no. 4, pp. 505–516, 2004.

[11] A. Funk, V. Basili, L. Hochstein, and J. Kepner, “Application of a development time productivity metric to parallel software development,” in *Proceedings of the second international workshop on Software engineering for high performance computing system applications*. ACM, 2005, pp. 8–12.

[12] ——, “Analysis of parallel software development using the relative development time productivity metric,” *CTWatch Quarterly*, vol. 2, no. 4A, 2006.

[13] J. Neely, “DOE centers of excellence performance portability meeting,” Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2016.

[14] S. J. Pennycook and S. A. Jarvis, “Developing performance-portable molecular dynamics kernels in OpenCL,” in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, Nov 2012, pp. 386–395.

[15] S. McIntosh-Smith, M. Boulton, D. Curran, and J. Price, “On the performance portability of structured grid codes on many-core computer architectures,” in *International Supercomputing Conference*. Springer, 2014, pp. 53–75.

[16] M. Martineau, S. McIntosh-Smith, and W. Gaudin, “Assessing the performance portability of modern parallel programming models using TeaLeaf,” *Concurrency and Computation: Practice and Experience*, vol. 29, no. 15, p. e4117, 2017.

[17] S. J. Pennycook, J. Sewall, and V. Lee, “A metric for performance portability,” *arXiv preprint arXiv:1611.07409*, 2016.

[18] S. Wienke, *Productivity and Software Development Effort Estimation in High-Performance Computing*. Apprimus Verlag, 2017.

[19] S. Wienke, D. an Mey, and M. S. Müller, “Accelerators for technical computing: Is it worth the pain? a TCO perspective,” in *International Supercomputing Conference*. Springer, 2013, pp. 330–342.

[20] S. Wienke, J. Miller, M. Schulz, and M. S. Müller, “Development effort estimation in HPC,” in *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 2016, pp. 107–118.

[21] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey, “Can traditional programming bridge the ninja performance gap for parallel computing applications?” in *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*. IEEE, 2012, pp. 440–451.

[22] V. Nguyen, S. Deeds-Rubin, T. Tan, and F. B. Boehm, “A SLOC counting standard,” in *COCOMO II Forum*, vol. 2007. Citeseer, 2007, pp. 1–16.

[23] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, “Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines,” *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[24] G. Mudalige, M. Giles, I. Reguly, C. Bertolli, and P. Kelly, “Op2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures,” in *Innovative Parallel Computing (InPar), 2012*. IEEE, 2012, pp. 1–12.

[25] M. Lange, N. Kukreja, M. Louboutin, F. Luporini, F. Vieira, V. Pandolfo, P. Velesko, P. Kazakas, and G. Gorman, “Devito: towards a generic finite difference DSL using symbolic Python,” in *Python for High-Performance and Scientific Computing (PyHPC), Workshop on*. IEEE, 2016, pp. 67–75.

[26] Khronos Group, “SYCL,” 2016. [Online]. Available: https://www.khronos.org/sycl

[27] S. Wienke, “Productivity and Software Development Effort Estimation in High-Performance Computing; 1. Auflage,” Dissertation, RWTH Aachen University, Aachen, 2017, veröffentlicht auf dem Publikationsserver der RWTH Aachen University 2018; Dissertation, RWTH Aachen University, 2017. [Online]. Available: https://publications.rwth-aachen.de/record/711110

[28] D. Wheeler, “SLOCCount,” 2001. [Online]. Available: http://www.dwheeler.com/sloccount/

[29] B. W. Boehm *et al.*, *Software engineering economics*. Prentice-hall Englewood Cliffs (NJ), 1981, vol. 197.

[30] J. E. Matson, B. E. Barrett, and J. M. Mellichamp, “Software development cost estimation using function points,” *IEEE Transactions on Software Engineering*, vol. 20, no. 4, pp. 275–287, 1994.

[31] J. W. Hunt and M. D. MacIlroy, *An algorithm for differential file comparison*. Bell Laboratories Murray Hill, 1976.

[32] S. G. Hart and L. E. Staveland, “Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research,” in *Human Mental Workload*, ser. Advances in Psychology, P. A. Hancock and N. Meshkati, Eds. North-Holland, 1988, vol. 52, pp. 139 – 183.

[33] R. Robey, H. A. Nam, K. Garrett, E. Koo, and L. V. Roekel. Parallel Computing Summer Research Internship. Los Alamos National Laboratory, Operated by Los Alamos National Security, LLC, for the U.S. Department of Energy. [Online]. Available: https://www.lanl.gov/projects/national-security-education-center/information-science-technology/summer-schools/parallelcomputing/

[34] S. Byna, R. Sisneros, K. Chadalavada, and Q. Koziol, “Tuning parallel i/o on blue waters for writing 10 trillion particles,” *Cray User Group (CUG)*, 2015.

[35] D. A. Korzekwa, “Truchas – a multi-physics tool for casting simulation,” *International Journal of Cast Metals Research*, vol. 22, no. 1-4, pp. 187–191, 2009, the Truchas software is available at https://gitlab.com/truchas. [Online]. Available: https://doi.org/10.1179/136404609X367641

[36] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, “Evaluating attainable memory bandwidth of parallel programming models via BabelStream,” *International Journal of Computational Science and Engineering (special issue)*, 2017.

Appendix A
Artifact Description Appendix: VPIC Performance Experiments - Table II, Table III and Fig. 3

A. Abstract

The methods used for the VPIC benchmarks are provided below. The Kokkos version of the code is not released at the time of the writing of this paper. However, the build environment, hardware platform and execution methods of testing are provided.

B. Description

1) Check-list (artifact meta information):

- **Program:** VPIC
- **Compilation:** CUDA, Intel® C Compiler, GCC, ARMCLANG and XL
- **Data set:** VPIC-provided harris input deck
- **Run-time environment:** Intel® MPI Library, OpenMPI
- **Hardware:** Intel® Xeon® Platinum 8176 processor, Power9, ThunderX2, Volta
- **Output:** Per-kernel timings reported by VPIC

2) How software can be obtained: The current VPIC codebase can be obtained at https://github.com/lanl/vpic. The Kokkos version of the VPIC software has not yet been released publicly.

3) Hardware: For these experiments the datsets were run on a variety of hardware detailed in Table VIII.

TABLE VIII: VPIC Performance: Hardware Specifications

Hardware Platform	Processor SKU	Cores per socket	Sockets per node	Memory per node (GB)
Intel® Xeon® Platinum 8176 Processor	Platinum 8176	28	2	376
IBM* Power 9	8335-GTG	20	2	285
Cavium* ThunderX2	CN9980	32	2	255
NVIDIA* Volta	V100 SXM2	5120	1	16

4) Software: The "original" version of VPIC used is available as commit c188cca6ad692a3f03865362c6480f223d870692 on the official VPIC github site: https://github.com/lanl/vpic/commit/c188cca6ad692a3f03865362c6480f223d870692

Compiler and MPI version per platform:

- **Intel® Xeon® Platinum 8176 Processor:** Intel® C Compiler 19.0.0 / Intel® MPI Library 19.0.0
- **IBM Power 9:** IBM XL 16.1.0 / OpenMPI 2.1.3
- **Cavium ThunderX2:** ARM HPC Compiler 18.4.0 / OpenMPI 2.1.3
- **NVIDIA Volta:** Cuda 9.2

5) Datasets: The "harris" data set from the sample folder in the official VPIC distribution was used. The nx, ny and nz were changed to 128, 128 and 128 respectively. The last change was to turn off check-pointing.

C. Installation

Starting from the standard VPIC CmakeLists.txt the following was changed for each platform. On all installations, both Kokkos and Original, -DCMAKE_BUILD_TYPE=RELEASE was used while configuring. The primary function of this flag is to add the -O3 flag and remove debug code from the code path.

On the Kokkos builds ENABLE_KOKKOS_AGGRESSIVE_VECTORIZATION was used as well as ENABLE_KOKKOS_OPENMP on the CPU builds and ENABLE_KOKKOS_CUDA on the GPU build. Additionally, for each platform KOKKOS_ARCH was set appropriately.

D. Experiment workflow

For each data point in Table II and Fig. 3 the specified build and data-set was run 5 times. From that data the lowest timing is shown.

Each experiment was run with one MPI process per socket, with each MPI process pinned to one of the available sockets. One thread per core on a specific socket was used (no HyperThreading or hardware threads). The Kokkos version used OpenMP exclusively while the original version used PThreads. OpenMP settings OMP_PLACES=threads and OMP_PROC_BIND=spread were used. All experiments were done using a single node.

The thread scaling experiment used the Intel® Xeon® Platinum 8176 processor platform listed in Table VIII. During the experiment data points from Fig. 3 the 1,2,4,8 and 16 thread experiments were executed on a single socket without MPI. The 32 and 56 thread data points use two MPI processes as described above with an equal number of threads on each socket.

Appendix B
Artifact Description Appendix: Truchas Performance Experiments - Table VII

A. Abstract

The ports to OpenMP for the CPU, OpenMP for the GPU and to CUDA with C kernels were the focus of this effort and for the performance studies.

B. Description

1) Check-list (artifact meta information):

- **Program:** Truchas
- **Compilation:** Nvidia CUDA compiler, Intel® Fortran Compiler, GNU Fortran compiler, and IBM XLF compiler
- **Run-time environment:** OpenMP environment variables set to best performing for each platform
- **Hardware:** Intel® Xeon® E5-2698 processor, Intel® Xeon Phi™ 7250 processor, Intel® Xeon® Platinum 8176 processor, Power9, Volta
- **Output:** Kernel averaged timings reported by kernel driver programs

2) How software can be obtained : The Truchas application is available at the official Truchas GitLab repository (https://gitlab.com/truchas/truchas-release).

The computational kernels were extracted from Truchas into a separate Truchas Kernels repository, which is publicly available at the web address https://gitlab.com/truchas/pcsri2018/truchas-kernels. Each of the three optimization approaches was implemented in its own git branch, as described in Table IX.

TABLE IX: Truchas Kernels Optimization Approaches

Optimization Approach	Git Branch Name
OpenMP CPU	`openmp_orig`
OpenMP GPU	`openmp_orig_offload_one_memcpy`
CUDA	`cuda_orig`

3) Hardware: For this case study we run on a variety of hardware detailed in Table X.

TABLE X: Truchas Performance: Hardware Specifications

Hardware Platform	Processor SKU	Cores per socket	Sockets per node	Threads per node	Memory per node (*GB*)
Intel® Xeon® E5-2698 processor	Xeon E5-2698	16	2	64	125
Intel® Xeon Phi™ 7250 processor	Xeon Phi 7250	68	1	272	94
Intel® Xeon® Platinum 8176 processor	Xeon Platnium 8176	28	2	112	376
IBM Power 9	8335-GTG	20	2	160	285
NVIDIA Volta	V100 SXM2	5120	1	N/A	16
NVIDIA TITAN V	V100 PCIE	5120	1	N/A	12

4) Software: The compiler version used on each platform:

- **Intel® Xeon® E5-2698 processor:** Intel® Fortran Compiler 18.0.2 or GNU Fortran 7.3.0
- **Intel® Xeon® E5-2683 processor:** Intel® Fortran Compiler 18.0.2 or GNU Fortran 7.3.0
- **Intel® Xeon Phi™ 7250 processor:** Intel® Fortran Compiler 18.0.2 or GNU Fortran 7.3.0
- **Intel® Xeon® Platinum 8176 processor:** Intel® Fortran Compiler 18.0.2 or GNU Fortran 7.3.0
- **IBM Power 9:** IBM XLF 16.1.0
- **NVIDIA Volta:** CUDA 9.2 or IBM XLF 16.1.0
- **NCIDIA Titan V:** CUDA 9.2

5) Datasets: The "puck-cast" mesh file located in the meshes folder of the Truchas Kernels repository was used for all experiments.

C. Installation

See the current build instructions on the Truchas Kernels GitLab repository. The repository's config folder contains a cmake configuration file for each of the compilers used in the experiment. For each installation, the -DCMAKE_BUILD_TYPE=RELEASE cmake flag was used to enable compiler optimizations and disable debugging code.

D. Experiment workflow

The existing Gradient and Mimetic Finite Difference Kernel were extracted from the Truchas repository and placed into a timing harness to experiment with the different ports. The driver programs that execute and time the extracted kernels are available in the Truchas Kernels repository.

Each experiment was run on a single node. OpenMP settings OMP_PLACES=cores and OMP_PROC_BIND=spread were used for all OpenMP CPU experiments. The number of OpenMP threads was set to the maximum number of threads per node, as shown in Table X.

Architectural Efficiency was computed as the ratio of main memory bandwidth of the computational kernel to the main memory bandwidth of a STREAM benchmark. The STREAM benchmarks for Intel® CPUs were obtained from the roof line plots generated by Intel® Advisor 2018. The STREAM benchmarks for IBM CPUs and Nvidia GPUs were calculated using the different STREAM implementations provided by the BabelStream [36] suite. BabelStream's OpenMP CPU, OpenMP GPU offloading, and CUDA implementations were compared against the corresponding optimization approaches.

E. Evaluation and expected result

Similar changes to the code can be made as is shown in the source code excerpts in the paper. Similar performance results should be obtained.

F. Experiment customization

Ports to other parallel frameworks and hardware can be made to see what performance might be obtained as well as the amount of code that needs to be changed. The hours required to make the port can be compared to those presented in the paper. An evaluation of a single-source pathway can be made along with a code divergence assessment.