# Performance on High-performance Computing Platforms is Possible without C++

Anshu Dubey, *Argonne National Laboratory, U.S.A*

Tal Ben-Nun, *Lawrence Livermore National Laboratory, U.S.A*

Bradford L. Chamberlain, *Hewlett Packard Enterprise, U.S.A*

Bronis R. de Supinski, *Lawrence Livermore National Laboratory, U.S.A*

Damian Rouson, *Lawrence Berkeley National Laboratory, U.S.A*

*Abstract—Computing at large scales has become extremely challenging due to increasing heterogeneity in both hardware and software. More and more scientific workflows must tackle a range of scales and use machine learning (ML) and artificial intelligence (AI) intertwined with more traditional numerical modeling methods, placing more demands on computational platforms. These constraints indicate a need to fundamentally rethink the way computational science is done and the tools that are needed to enable these complex workflows. The current set of C++ based solutions may not suffice and relying exclusively upon C++ may not be the best option, especially because several newer languages and boutique solutions offer more robust design features to tackle the challenges of heterogeneity. In June 2023, we held a minisymposium that explored the use of newer languages and heterogeneity solutions that are not tied to C++ and that offer options beyond template metaprogramming and parallel-for for performance and portability. We describe some of the presentations and discussion from the minisymposium in this article.*

## INTRODUCTION

Computing at large scales has become extremely challenging due to increasing heterogeneity in both hardware and software. A positive feedback loop exists where more scientific insight leads to more complex solvers which in turn need more computational resources. More and more scientific workflows need to tackle a range of scales and use machine learning (ML) and artificial intelligence (AI) intertwined with more traditional numerical modeling methods, placing more demands on computational platforms. These constraints indicate a need to fundamentally rethink the way computational science is done and the tools that are needed to enable these complex workflows.

Hardware is becoming more complex because Dennard scaling, which permitted faster chips by increasing clock speeds, has ended. Now we obtain greater computing power through different architectures in different computational units, such as CPUs and GPUs. GPU programming does not respect a unified programming paradigm in the same way that MPI did for distributed memory parallelism. Too many design choices are available for software development and choices are typically not universally applicable across platforms in the same generation, let alone from one generation to the next. A different cycle plays out for scientific software. One starts with simplified models for initial insights. As understanding and insights grow, models are refined, which in turn need more diverse solvers, thereby increasing the complexity of the software.

Abstractions have long been recognized as a key mechanism to tame this complexity on multiple axes. Maximum success in this arena has been obtained by the tools that rely on C++ template meta-programming. These tools hide the specialization for specific devices from the code developer, letting them express their computation through a common API. However, these solutions have their own shortcomings. The most obvious one is that they work only for C++ codes, but a few others are also worth observing. Debugging with templates is extremely difficult and tedious. Further, since the tools must account for all corner cases, they often themselves become cumbersome to manage. It


Digital Object Identifier 10.1109/MCSE.2023.3329330

is also not obvious how C++ based solutions will fit with workflows that involve interfacing with AI/ML tools.

In June 2023 we organized a minisymposium at PASC-23[1] to explore portability solutions that do not rely upon C++. We considered newer HPC languages, extensions to languages that are attempting to provide HPC solutions, tools developed to exploit the Python ecosystem, and tools that came out as a result of lack of choices, such as legacy Fortran. In this article we feature some of the solutions that were presented in the minisymposium, and also distill the discussions with the attendees.

## Portability Challenges

In this article we are deliberately avoiding the use of the term "performance portability" because we agreed that the term was misleading. What the scientific community really cares about is obtaining acceptable performance across platforms. In today's landscape, therefore, we can redefine portability to imply that the code is able to use all available hardware resources adequately. With that definition, portability suffices for the purpose of this discussion.

It has always been true that target-specific code optimization is needed for improving performance on any platform. In the past, a common parallel programming model, that of distributed memory, made it easy to separate scaling optimization from arithmetic and computational optimization. Further, scaling optimizations typically worked well across platforms without needing significant code modification. While this portability remains valid for MPI-based scaling optimizations, much more parallelism is available on the node, and no widely avialable single programming model currently works well across the existing variety of node architecture. Data locality, critical for on-node performance, has become extremely difficult to understand, express and manage. Mapping computation to resources, and effecting data movement to execute the mapping are difficult to determine, and impossible to express in conventional programming languages. To circumvent this challenge, abstraction tools have largely aimed at providing a common interface with multiple backends. They optimize performance by inferring the map and movement through code analysis, and as a result end up being very complex.

[1] https://pasc23.pasc-conference.org/, slides for most presentations are available at the website through the time-table

## Language/Compiler Based Solutions

Chapel[2] is a programming language defined from first principles to support scalable parallel computing from the desktop to the supercomputer [1]. Chapel aims to support code that is similarly readable and writeable as Python while providing the performance, scalability, and control that users are accustomed to from traditional HPC technologies like Fortran/C/C++, MPI, OpenMP, and CUDA.

Beyond standard desktop language features like procedures, objects, and iterators, Chapel has direct support for expressing the two main concerns that are crucial to scalable computing: *parallelism*—what computations should execute simultaneously—and *locality*—where data should be allocated and where computations should execute. These features can be combined in various ways to express parallel computations for multicore processors, distributed memory systems, and GPUs in a vendor-neutral manner.

The key feature supporting this portability is the *locale* which can represent the memory and processor cores of a network-attached compute node, a socket on that compute node, or a GPU. By using Chapel's *on-clauses* to target a given locale or sub-locale, computation and memory allocation can be directed to a specific unit of the target system with the compiler generating the appropriate instructions. Meanwhile Chapel supports a partitioned global namespace in which variables can be accessed using traditional lexical scoping whether they are local, remote, or in GPU memory. These core features are then used to build higher-level abstractions like distributed arrays or parallel iterators to support higher-level parallel programming without the need to manage every detail.

In practice, users are making use of these features to write applications being actively used in production and research in fields as diverse as unstructured 3D CFD for aircraft design, massive-scale interactive data science, exact diagonalization for quantum many-body physics, and image analysis to study coral reef biodiversity. Chapel has also been applied to AI/ML problems where the desire for writing innovative new codes and algorithms portably and quickly can overcome traditional barriers to trying new languages such as the need to maintain long-lived legacy codes.

### Fortran

The world's first widely used, high-level programming language invented in 1957, Fortran has evolved

[2] https://chapel-lang.org

through the publication of standards offering array programming in Fortran 90; object-oriented programming and C-interoperability in Fortran 2003; modular, parallel and GPU programming in Fortran 2008; and expanded parallelism and C-interoperability in Fortran 2018. As of this writing, a 2023 standard is nearing publication, and the Fortran standard committee is developing type-safe generic programming and task-based asynchrony for the next standard after 2023. While the abstractions supporting these new features offer improved programmability, the underlying runtime libraries promise improved performance.

On the programmability front, *coarray* distributed data structures offer a simple syntactical extension to traditional arrays and array statements and array intrinsic functions facilitate compact expression of unordered calculations without requiring compiler directives. Additional parallel features include atomic operations, collective procedures, image-failure detection (an *image* is an instance of a program analogous to an MPI rank), and *teams* (groupings) of images with intra-team synchronizationa and communication mechanisms.

On the performance front, `do concurrent` loops offer automatic GPU offloading with some compilers and compiler developers can support coarray features with communication libraries that outperform MPI. For example, the Caffeine parallel runtime library uses the GASNEt-Ex exascale networking middleware [2].

Owing partly to the ongoing development of legacy applications in fields that embraced computing early — fields such as nuclear energy; automotive and aerospace engineeering; weather forecasting and climate prediction — Fortran occupies a substantial fraction of HPC centers' workloads [3]. A growing community of open-source, modern Fortran developers are also pushing the language into non-traditional domains such as package management, deep learning, and task scheduling.

### OpenMP

OpenMP [4] extends base languages through directives to support a wide-range of parallelization strategies. This language began as an application programming interface (API) that unified directive-based extensions available in multiple compilers to support loop-level parallelization. The API has evolved during its over 25 year history to support a nearly full-range of parallelization idioms, including task-based parallelism, SIMD/vector parallelism and accelerator/offload parallelism. The API is supported by all HPC compilers and so offers significant portability within the differences and limitations of different implementations.

OpenMP continues to evolve and to add functionality. While some complain about the wide range of features available in the language, its original concepts remain and the advanced features offer mechanisms to increase parallelization control only if required. Important recent additions reflect that parallelization control can inhibit traditional compiler optimizations. Thus, direct support to specify key optimizations such as loop transformations are now included. These emerging directions in OpenMP offer the programmer or tool implementer the ability to "program the compiler" and the use of OpenMP to implement autotuning tools and domain specific languages is becoming common.

## Tools Based Solutions

### DaCe

Over the past decade, Python has become one of the most popular programming languages. Python is supported by an ecosystem of packages, with NumPy as a central provider for scientific computing. Although the NumPy syntax and semantics bear similarities to Fortran's array programming, Python and its quirks are challenging to accelerate in the general case.

The Data-Centric (DaCe) parallel programming framework [5] is a Python compiler focused on data movement minimization. To balance productivity and performance, DaCe defines and compiles a subset of Python/NumPy in which data containers and their accesses can be analyzed. The subset of Python is complemented by augmenting the language with optimization hints (e.g., symbolic array shapes, parallel loop annotation), allowing programmers to compile codes ahead-of-time and guide optimization.

Internally, DaCe uses an intermediate representation that exposes and transforms parametric data movement and hierarchical parallelism, promoting portability across its supported targets — CPU, GPU, and FPGA. Data-centric transformations such as memory allocation scheduling, inter-component streaming (i.e., array-to-FIFO), and global dataflow orchestration have proven to be crucial in transforming imperative NumPy code to efficiently utilize GPUs and FPGAs. The parametric data movement analysis guides optimization decisions via performance models (e.g., work-depth analysis) or through automated instrumentation and tuning. DaCe is actively used in atmospheric modeling, running the finite-volume cubed-sphere dynamical Core[3] entirely on the GPU for the first time;

[3] https://www.gfdl.noaa.gov/fv3/

quantum transport simulation; distributed multilinear algebra; and machine learning; proving that Python can be used to write portable HPC codes productively.

### Boutique Tool-chain

The boutique tool-chain described in this section was built for enabling portability in a multiphysics application software largely written in Fortran, Flash-X [4]. The design of the tool-chain is based on the observation that node heterogeneity requires applications to address three main requirements.

- Abstraction for Unification: Different computational hardware components often require different data layouts and orders of operation, while the base arithmetic of the computation remains the same. It is desirable to unify such variants into a single expression for ease of maintenance.
- Mapping Computations and Data: Applications need to map computations and data to different computational compoments within the node for optimizing performance and resource utilization.
- Dynamic Data and Computation Movement: A mechanism needs to be in place to dynamically move data and computation to the target computational components based on the established mapping

In response to these requirements, The Flash-X team built three distinct sets of tools where each one addresses a specific action mentioned above [6]:

**Macro-based Code Compression** where macros, with a custom processor, are used to condense source code. Macros are allowed to have multiple alternative definitions, including null definitions. The expansion of macros is managed by a Python tool called "macro-processor" which also arbitrates on which definition to apply where.

**CGKit** is a specialized tool designed to process recipes expressed in a high-level pseudocode-like format. It assumes availability of templates that describe the logical control flow of algorithm for which the code is being generated. The recipes define concurrent regions and high-level dependencies among functional components, specifying the execution location of each component. The tool translates these dependencies into a graph, optimizing it to minimize data movement and maximize load balance. It then uses specified control flow template to convert the optimized graph into compilable code.

[4] https://flash-x.org

**Milhoja** Runtime functions as a runtime that executes an extended finite state machine. It assigns thread teams to manage data movement to specified destinations and launches computations accordingly. Application-specific helper tools translate the data needs of computations into a Json description of the required data packet. A data packet generator then emits code capable of packaging the data and associated pointers into data packets, facilitating their movement by Milhoja. It is expected that Miljoha is accompanied by data movement pipelines for target platforms based on performance analysis. This may need to be a separate activity for every application using Milhoha.

Together these tools are able to provide an end-to-end portability solution that is expected to be easy to adapt for new architectures. In the best case scenario, a new platform will only require generation of new pipelines for Milhoja, and perhaps new templates for CGKit. Some platforms may need additional alternative definitions of macros. In the worst case, new sections of the source may need to have variants, and new macros may need to be introduced, or a completely new algorithm may be needed. However, the design philosophy of the tool-chain ensures that modifications needed in the maintained source code are as unintrusive as possible for a given platform architecture.

## Discussion

We reserved the last slot of the minisymposium for open discussion with audience participation. Not very surprisingly, a great deal of discussion revolved around Fortran's language features and semantics and the prospects of its future. It's evident that Fortran's trajectory is a significant point of debate, with a focus on its continued relevance in the world of HPC. Some HPC centers have a substantial presence of Fortran codes, sometimes constituting up to 80% of computational cycles. However, when these codes undergo refactoring, especially for upgrades or GPU offloading, they often transition to C++. The economics of the situation play a crucial role; C++ gains adoption due to substantial investment from major tech players, securing its future in ways that other languages cannot.

An idea was proposed to freeze feature growth in Fortran to maintain its viability, given the cost-effectiveness of fewer features for writing compilers. The discussion pointed to efforts towards enhancing compilers being more useful for the scientific community rather than addition of new unsupported features. Additionally, it was noted that some large facilities might not require compliance with the latest standard,

as their users might not utilize the advanced features. Not all participants agreed with this. Developments in both the open source and commercial world are not widely known and may lead to better sustainment of the Fortran ecosystem. As a counterargument to the expectation that open source would prove a panacea, others noted that the broader Fortran community is too small to sustain the development tools in the way that LLVM has for C and C++. Another key aspect of the discussion involved exploring ways to constrain the language's semantics in the standard, with advanced features potentially being made available as add-ons. However, concerns were expressed about the sustainability of these features if they were not incorporated into the standard, potentially dissuading users from adopting them due to fear of losing support.

Despite the growth of C++, the conversation acknowledged the rising popularity of languages like Julia and Rust in general, and Chapel in HPC, which strive to amalgamate the best features of high productivity languages like Python and MATLAB while preserving the performance characteristics of traditional languages. However, their long term viability was noted to be uncertain given that they still have modest levels of adoption. They are used by smaller projects, and show a great deal of promise, while large-scale codes are more conservative in adopting these newer solutions, as they should be. Tools based solutions are by design adaptable to the changes in hardware landscape more readily at the cost of requiring more direct input from the users and creating their own portability challenges.

## ACKNOWLEDGMENTS

This work was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

Authors would like to acknowledge contributions from attendees, and in particular, Emil Vatai and Valentin Churavy who also gave presentations.

## REFERENCES

1. B. L. Chamberlain. Chapel. In P. Balaji, editor, *Programming Models for Parallel Computing*, chapter 6, pages 129–159. MIT Press, November 2015.
2. D. Rouson and D. Bonachea. Caffeine: Coarray fortran framework of efficient interfaces to network environments. In *2022 IEEE/ACM Eighth Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 34–42. IEEE, 2022.
3. B. Austin et al. *NERSC-10 Workload Analysis*, 2020. doi:10.25344/S4N30W.
4. OpenMP ARB. OpenMP 5.2 Specification. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf, Nov. 2021.
5. T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.
6. A. Dubey, Y. Lee, T. Klosterman, and E. Vatai. A tool and a methodology to use macros for abstracting variations in code for different computational demands. *Future Generation Computer Systems*, 2023.

**Anshu Dubey** is a Senior Computational Scientist in the Mathematics and Computer Science Division at Argonne National Laboratory and a Senior Scientist (CASE) at the University of Chicago. Her research interests include all aspects of scientific software development.

**Tal Ben-Nun** is a Computer Scientist in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. His research interests lie in the intersections between programming languages, scientific computing, and machine learning.

**Brad Chamberlain** is a Distinguished Technologist at Hewlett Packard Enterprise (formerly Cray Inc.) whose career has focused on user productivity for HPC systems, particularly through the design and development of the Chapel parallel programming language (https://chapel-lang.org).

**Bronis R. de Supinski** is Chief Technology Officer (CTO) for Livermore Computing (LC) at Lawrence Livermore National Laboratory (LLNL), a role in which he formulates LLNL's large-scale computing strategy and oversees its implementation. He is a Fellow of the ACM and the IEEE.

**Damian Rouson** is a Scientist at Berkeley Lab, where he leads the Computer Languages and Systems Software Group, researches language-based parallel and GPU programming and deep learning for HPC applications and teaches parallel programming model tutorials. He also founded and leads the research software engineering firms Archeaologic Inc. and Sourcery Institute.