UNIVERSITY of York

This is a repository copy of *Developing Performance Portable Plasma Edge Simulations: A Survey*.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/208741/

Version: Accepted Version

Article:

Wright, Steven A. orcid.org/0000-0001-7133-8533, Ridgers, Christopher Paul orcid.org/0000-0002-4078-0887, Mudalige, Gihan R. et al. (5 more authors) (2024) Developing Performance Portable Plasma Edge Simulations: A Survey. Computer Physics Communications. 109123. ISSN 0010-4655

https://doi.org/10.1016/j.cpc.2024.109123

Reuse
This article is distributed under the terms of the Creative Commons Attribution (CC BY) licence. This licence allows you to distribute, remix, tweak, and build upon the work, even commercially, as long as you credit the authors for the original work. More information and the full terms of the licence here: https://creativecommons.org/licenses/

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

White Rose
university consortium
Universities of Leeds, Sheffield & York

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

Developing Performance Portable Plasma Edge Simulations: A Survey

Steven A. Wright[a,*], Christopher Ridgers[b], Gihan Mudalige[c], Zaman Lantra[c], Josh Williams[d], Andrew Sunderland[d], Sue Thorne[d], Wayne Arter[e]

[a] *Department of Computer Science, University of York, York YO10 5GH, UK*
[b] *York Plasma Institute, University of York, York YO10 5DQ, UK*
[c] *Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK*
[d] *Hartree Centre, STFC Daresbury Laboratory, Sci-Tech Daresbury, Keckwick, Daresbury, Warrington, WA4 4AD, UK*
[e] *UK Atomic Energy Authority, Culham Science Centre, Abingdon OX14 3DB, UK*

Abstract

Heterogeneous architectures are increasingly common in modern High-Performance Computing (HPC) systems. Achieving high-performance on such heterogeneous systems requires new approaches to application development that are able to achieve the three Ps: Performance, Portability, and Productivity.

In this paper, we provide an overview of the state-of-the-art for developing high-performance, portable and productive multi-physics applications with particular focus on the simulation of a plasma fusion reactor. Simulating such a complex system relies on both fluid- and particle-based simulations, and coupling interfaces between these two domains. We also review the current state-of-the-art in reasoning about the performance, portability and productivity of HPC applications.

Keywords: High-Performance, Parallel Programming, Portability, Coupling, Plasma Simulation, Reactor Design

---

1. Introduction

Numerical simulation allows scientists and engineers to rapidly develop physical understanding and prototype new designs in domains where physical experimentation may be costly, impractical, or dangerous. As a result, computational methods have joined theory and experiment as central pillars of scientific investigation. Maximising the performance of these numerical simulations means that more calculations can be carried out, allowing scientists to increase the size, complexity, or resolution of their experiments. The field of High Performance Computing (HPC) exists to develop and improve the performance of scientific applications and the supercomputers running

*Corresponding author:
Email address: steven.wright@york.ac.uk (Steven A. Wright)

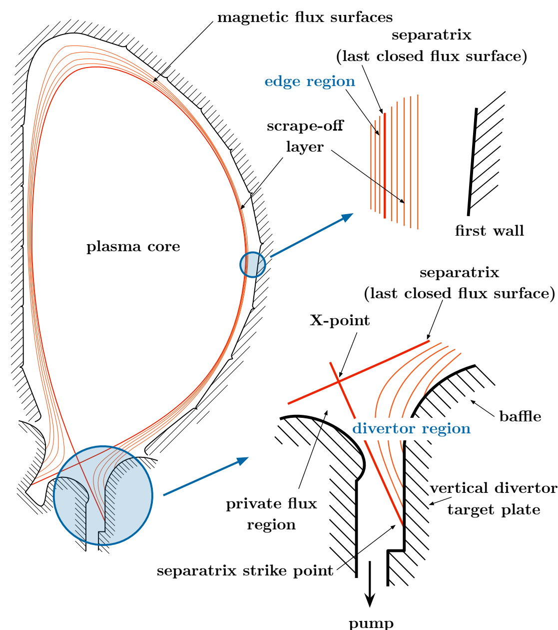


Figure 1: A schematic of a generic tokamak "poloidal cross section", highlighting the targeted regions of plasma and machine, namely the main chamber between core plasma scrape off layer and wall (upper circle) and the so called "plasma exhaust" or "divertor" region (lower circle) where heat and particles come into direct contact with material surfaces [7]

them. In turn, increasingly powerful supercomputers allow us to tackle new scientific grand challenges [1, 2]. One such grand challenge is the economic delivery of nuclear fusion energy in the coming decades.

Project NEPTUNE (NEutrals and Plasma TUrbulence Numerics for the Exascale) aims to develop new modelling software to treat the complex dynamics of high temperature fusion plasma for the design of a nuclear fusion reactor, and is funded as part of the UK's ExCALIBUR programme [3]. It is one of a number of efforts in developing new numerical simulation software focused on nuclear fusion. Other notable examples include the Whole Device Model Application (WDMApp) project [4], funded by the Department of Energy's Exascale Computing Project (ECP), and the Plasma-PEPSC project [5], funded by EuroHPC.

The aim of Project NEPTUNE is to deliver expertise in, and tools for, "in-silico" reactor interpretation and design, initially with a focus upon the "edge" region of the tokamak plasma where hot plasma comes into contact with the material walls of the machine (see Figure 1). This challenging, multi-physics, multi-scale intersection between plasma physics and engineering has long been heralded as at least an "exascale" modelling and simulation problem, and its solution is widely regarded as critical to the success of commercial fusion energy (e.g., see Mission 2 of the EUROfusion Road Map [6]).

One of the biggest challenges in developing such a complex simulation applica-

tion is in ensuring that it is able to achieve high performance on current- and future-generation supercomputers, adapting to new advancements in hardware and software. This is especially important currently as we transition towards heterogeneous systems comprising of CPU hosts with specialised accelerator architectures, where achieving high performance often requires the adoption of vendor-specific programming models and tools, and appropriate algorithms for the accelerator architecture.

For most large scientific simulation applications, maintaining multiple versions of a code-base is simply not a reasonable option given the significant time and effort required, not to mention the expertise needed in the many different associated technologies. Even holding multiple versions does not guarantee a future-proof application where the next innovation in hardware may well require yet another parallel programming model or a different parallelisation strategy to obtain best performance for the new device. These challenges are now general and applicable equally to any scientific domain that relies on numerical simulation software using HPC systems. As a recent review for applications in the computational fluid dynamics (CFD) domain [8] elucidates, three key factors can be identified when considering the development and maintenance of large-scale simulation software, particularly aimed at production:

1. **Performance:** running at a reasonable/good fraction of peak performance on given hardware.
2. **Portability:** being able to run the code on different hardware platforms/architectures with minimum manual modifications.
3. **Productivity:** the ability to quickly implement new applications and features, and maintain existing ones.

The last of these is particularly challenging for nuclear fusion, where new experimental results have on several occasions suggested radical revision of models and designs.

As more and more parallelisation features have been added to general purpose programming languages, achieving performance, portability and productivity in production code bases has become increasingly difficult – with significant application rewrites required to achieve high performance and portability. Fully automated parallelisation by a compiler for general purpose languages has consistently failed, requiring manual intervention to achieve a satisfactory result [9–12]. Compilers for imperative languages such as C/C++ or Fortran, the dominant languages in HPC, have struggled to extract sufficient semantic information, enabling them to safely parallelise a program from all, but the simplest structures. Consequently, the programmer has been forced to carry the burden of "instructing" the compiler to exploit available parallelism in applications, targeting the latest, and purportedly greatest, hardware.

In many cases, the use of very low-level techniques, some only exposed by a particular programming model/language extension, are required with careful orchestration of computation and communications to obtain the best performance. Such a deep understanding of hardware is difficult to gain and, even more so, unreasonable for domain scientist/engineers to be proficient in – especially given that the expertise required rapidly changes with the technology of the moment following hardware trends. A good

example is the many-core path originally touted by Intel with accelerators such as the Xeon Phi, which has been discontinued – the first US Exascale systems are all GPU based, with two systems containing AMD GPUs, and one containing Intel GPUs.

As such, it is near impossible to keep re-implementing large science codes for various architectures. This has led to a *separation of concerns* approach where the description of what to compute is separated from how the computation is implemented. This is in direct contrast to languages such as C/C++ or Fortran, which explicitly describe the computation.

In this paper, we provide a review of the state-of-the-art in achieving the goal of developing software that is performant, portable and productive. Specifically we outline the key approaches and tools currently used to develop numerical simulation applications targeting modern HPC architectures and systems, including methods of re-engineering existing codes appropriately. A number of these approaches and tools have been covered in previous surveys [8, 13–15], however this paper provides a broader picture of this landscape, considering low-level architecture-specific techniques, through to high-level domain-specific languages that allow scientists to represent their problems in a mathematical form. This paper additionally includes consideration of coupling frameworks, that allow developers to couple many different solvers together.

Our review focuses on applications and algorithms from the plasma fusion domain and related supporting applications from engineering. Our aim is to survey and present the state-of-the-art in achieving "performance portability" for fusion, where an application can achieve efficient execution across a wide range of HPC architectures without significant manual modifications. For completeness, we focus on both fluid- and particle-based computational models of plasma, and on frameworks for coupling these approaches.

The remainder of this paper is structured as follows: Section 2 highlights the challenges in developing future-proofed scientific applications; Section 3 provides a review of the predominant general purpose programming languages used in scientific application development; Section 4 outlines some of the main programming models that provide parallelism to developers; Section 5 describes of a range of important software libraries that provide common functionality to parallel scientific applications; Section 6 gives an overview of domain specific languages specifically targeted at fluid- and particle-based simulations; Section 7 outlines a number of coupling interfaces that allow multiple applications to be coupled together; Section 8 gives an overview of how performance, portability and productivity may be evaluated for a project such as NEPTUNE; finally, Section 9 concludes the paper.

2. Challenges in Developing Modern Parallel Applications

The end of CPU clock frequency scaling in 2004 gave rise to multi-core designs for mainstream processor architectures. The turning point came about as the current CMOS-based microprocessor technology reached its physical limits, reaching the threshold postulated by Dennard in 1974 [16]. The end of Dennard scaling has meant that further increases in clock frequency would result in unsustainably large power consumption, effectively halting a CPUs ability to operate within the same power envelope at higher frequencies.

More than a decade and a half has passed since the switch to multi-core, where we now see a golden age of processor architecture design with increasingly complex and innovative designs used to continue delivering performance improvements [17]. The primary trend continues to be the development of designs that use more and more discrete processor "cores" with the assumption that more units can do more work in parallel to deliver higher performance by way of increased throughput. This has aligned well with the hardware industries' ambition to see the continuation of Moore's Law – exponentially increasing the number of transistors on a silicon processor.

As a result, on the one hand, we see traditional CPU architectures gaining more cores, currently over 40 cores for high-end processors, and increasing vector widths (e.g., Intel's 512-bit vector units) per core, widening their ability to do more work in parallel. On the other hand, we see the widespread adoption of separate devices, called accelerators, such as GPUs that contain much larger numbers (over 1024) of low-frequency (power) cores, targeted at speeding up specific workloads.

More cores on a processor has effectively resulted in making calculations on a processor, usually measured by floating-point operations per second (FLOP/s), cheap. However, feeding the many processors with data to carry out the calculations, measured by bandwidth (bits/sec), has become a bottleneck. As the growth in the speed of memory units has lagged that of computational units, multiple levels of memory hierarchy have been designed, with significant chunks of silicon dedicated to caches to bridge the bandwidth/core-count gap.

Memory technologies, such as High Bandwidth Memory (HBM), has produced "stacked memory" designs where embedded DRAM is integrated on to CPU chips. The memory hierarchy has been further extended off-node, with burst buffers and I/O nodes serving as staging areas for scientific data en route to a parallel file system. Larger and more heterogeneous machines have also necessitated more complex interconnection strategies. Technologies such as NVLink, Infinity Fabric or Compute eXpress Link allow GPUs to communicate point-to-point without requiring data to travel through the CPU. New high-speed interconnects have been developed that seek to minimise the number of *hops* required to move data between nodes and devices, potentially benefiting both inter-node communications and file system operations.

A decade ago, the vast majority of the fastest HPC systems in the world were homogeneous clusters based around the x86-64 architecture, with a few notable exceptions such as the IBM BlueGene architectures. Now, there is a diverse range of multi-core CPUs on offer, supported by an array of manycore co-processor architectures, complex high-speed interconnects, and multi-level parallel file systems.

The underpinning expectation of the switch to multi-core and the subsequent proliferation of complex, massively parallel hardware was that performance improvements could be maintained at historical rates. As a recent review by Leiserson et al. elicits, improvements in performance are now likely to come from the top of the computing stack in the form of algorithms, architecture and "big components" (e.g., software with millions of lines of code, GPUs, CPUs, compilers) [18].

However, this has led to the need for highly skilled parallel programming know-how to fully exploit the full potential of these devices and systems. The switch to parallelism and its consequences was aptly described by David Patterson in 2010 as a "Hail-Mary pass", an act done in desperation by the hardware vendors "without any

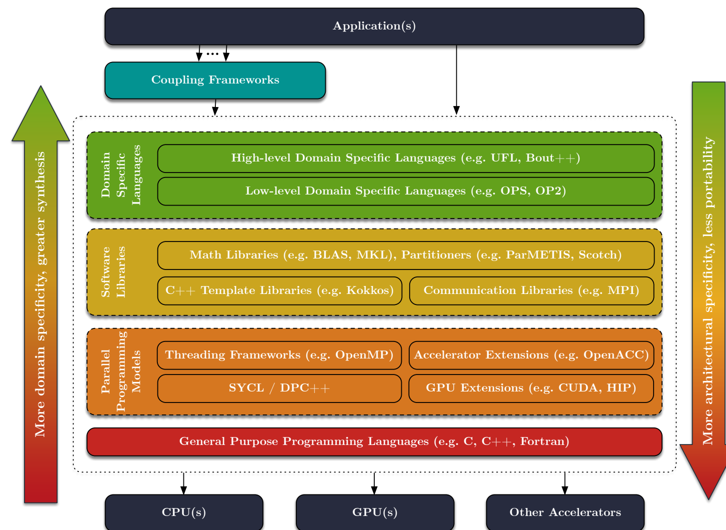


Figure 2: Overview of the potential layers in a software stack for a multi-physics simulation application.

clear notion of how such devices would in general be programmed" [19].

More than a decade later, industry, academia and stakeholders of HPC have still not been able to provide an acceptable and agile software solution to this issue. The problem has become even more significant with the current deployment of heterogeneous Exascale-capable HPC systems, limiting their use for real-world applications for continued scientific delivery. On the one hand, open standards have been slow to catch up with supporting new hardware and, for many real applications, have not provided the best performance achievable from these devices. On the other hand, proprietary solutions have only targeted narrow vendor-specific devices resulting in a proliferation of parallel programming models and technologies.

On heterogeneous platforms, a significant proportion of the available performance comes from the accelerators, with the host CPU primarily providing problem setup, synchronisation, and I/O operations. Each of the major GPU manufacturers provide a different programming model to interact with their accelerators and, so, application developers must consider their approach when targeting a heterogeneous system. Further consideration must also be given to vendor-supported approaches that may lead to vendor lock-in.

Figure 2 gives a broad outline of the various components that may be involved when developing multi-physics simulation applications for execution on heterogeneous systems. Higher-level representations of physics problems (such as DSLs) allows an application to better synthesise machine-code representations for various hardware, and potentially enables more developer productivity (in many DSLs partial differential equations can be represented directly in code). Lower-level representations are more likely to be able to exploit available parallelism on various platforms, but may limit portability between systems.

Over the next five sections of this paper, we will focus on each of the boxes in

Figure 2 in turn, looking at the current state-of-the-art in (i) general purpose programming languages, (ii) parallel programming models, (iii) software libraries, (iv) domain specific languages, and (v) coupling frameworks. Our survey follows Reguly and Mudalige [8] together with specific considerations for algorithms of interest to the fusion domain (in particular both fluid- and particle-representations of plasma fusion).

3. General Purpose Programming Languages

In this class we consider traditional programming languages with long history of usage and support in scientific computing. These languages typically allow fine control over every aspect of an algorithms implementation.

Scientific computing is dominated by the **Fortran**, **C** and **C++** programming languages. In the UK, Fortran-based applications currently dwarf C/C++ applications. On ARCHER, the UK's Tier-1 resource between 2014 and 2020, Fortran applications accounted for almost 70% of the machine's core hours, while C/C++ applications accounted for around 15% [20]. This trend has somewhat continued on ARCHER2, with usage data for April 2023 showing that Fortran applications still account for over half of the identifiable core hours, while C/C++ accounts for around a third[1]. This skew towards Fortran is in part due to a number of mature applications with large user bases, such as VASP (24% of identified usage on ARCHER2), and its longevity in HPC, meaning that it benefits from mature compiler support more than most other languages.

Within the US Department of Energy's Exascale Computing Project (ECP), the opposite trend can be seen, where C/C++ is being actively pursued for around 70% of their applications, compared to just 10% in Fortran [21]. This is motivated by a perceived risk of relying on Fortran for mission critical codes in the future [22]. A number of other studies have similarly concluded that Fortran may harm portability and productivity due to limitations in the language [23] and the development of tooling [24]. However, there are also ongoing efforts to bridge some of the gaps between C/C++ and Fortran, with the introduction of the Fortran Lang community, a Fortran standard library and a Fortran package manager [25].

Another language growing in popularity in HPC is **Python**. While not traditionally a "high performance" language, it provides interfaces to many external libraries, often written using languages such as C/C++ and Fortran. This has meant that Python can provide an easy interface for developers to write their applications at a high-level, leaving the implementation and execution to optimised libraries (see Section 6). Due to Python's use in a wide range of fields, by large corporations such as Alphabet, the community has invested significant effort into improving the performance of pure Python. The flexibility of the language and dynamic type system limits opportunities for static analysis and optimisation; instead Just-In-Time (JIT) compilers have been developed, both as libraries to target particular code hotspots (e.g., Numba [26]), and whole programs (e.g., PyPy [27]). However, threading within Python, and thus its parallel performance, has historically been poor, limited by the Global Interpreter Lock (GIL) present

[1] `https://github.com/ARCHER2-HPC/usage-data/tree/main/allusers/2023/04`

in the reference CPython implementation, PyPy and Stackless Python [28]. Proposals to remove this lock, or make it optional, are on-going (see PEP 703), and until this happens, Python's use in HPC will continue to be limited to being primarily a "glue" language, coordinating work done in components implemented in higher-performance languages.

There is a long history of research and development of languages for scientific and high performance computing, including those such as **Chapel** [29], **Fortress** [30] and **X10** [31] which target parallel computation. These have tended to remain niche languages and have not been widely adopted. A promising language, which is general purpose but designed in particular for scientific computing, is the **Julia** language [32]. This has a syntax which is familiar to Matlab or Fortran programmers, but is built on a sophisticated type system and language design, and uses LLVM to perform JIT compilation for CPU and GPU hardware. It is a relatively new language (version 1.0 was released in August 2018), but is seeing rapid adoption in scientific and machine-learning communities, and already has some libraries which are recognised as best in class [33]. It aims to combine the flexibility and high productivity of Python, with high performance.

Developing applications in these general purpose programming languages presents a number of challenges:

1. The languages are very prescriptive, and optimising an application for one system may harm performance on another system. In fact, optimising for one architecture can obfuscate the science source so much so that future maintenance and addition of new features becomes difficult.

2. Applications developed with multiple code paths may provide portable performance, but requires duplicated effort keeping each code path up to date.

3. Parallelism must be explicitly written into the application, almost always using parallel programming extensions to the languages (as discussed in the next section), significantly increasing the complexity of development.

4. Parallel Programming Models

In this class we consider the programming models that extend from traditional general purpose programming languages to provide parallelisation both on-node (e.g., vectorisation, threading) and off-node (e.g., message passing). We also consider programming models that are designed specifically for heterogeneous computation with accelerator devices.

The parallelism available on modern supercomputers is hierarchical in nature. Vector operations (e.g., **SSE**, **AVX**, **APX**, **SVE**) provide instruction-level parallelism within a core (using a single instruction, multiple data (SIMD) model), while threading provides a form of shared memory parallelism within a node. Symmetric Multithreading (SMT) allows two or more threads to co-exist (with an architecture dependent degree of concurrency) on a single compute core, potentially increasing compute unit utilisation. Parallelism across a system is usually achieved using message passing or shared global memory techniques.

Vectorised code can be achieved during the compilation phase, if there are no data dependencies present in the code. All modern compilers attempt to generate vectorised code through auto-vectorisation, usually when higher optimisation levels are specified (e.g., with compiler flags such as `-O2` and above). However, the compiler will only produce vectorised code when it is absolutely certain that no dependencies exist. In almost all non-trivial (especially real-world) codes a conclusive determination cannot be made and auto-vectorisation fails [34].

A developer can aid the compiler with the use of **compiler directives** or **vector intrinsics**. Compiler directives (or pragmas) allow a developer to indicate that an assumed dependency can be ignored, potentially resulting in the compiler being able to generate vectorisable code that is portable across architectures. However, the compiler may still believe there is a dependency present and, in this case, the developer must use a lower-level interface (e.g., **Intel Intrinsics**) to directly manipulate the vector registers [35]. This is likely to result in higher performance, but the instruction set specific code is likely to significantly harm both portability and productivity [36].

Distributing execution across all cores in a node can be achieved through threading and shared memory, or through message passing.

In HPC applications threading is often achieved through **OpenMP** [37–40], while message passing is usually implemented using the **Message Passing Interface (MPI)** [41]. Both threading and message passing can be achieved through the **POSIX Threads (pthreads)** library; however, like OpenMP, this approach does not extend to distributed memory systems. Consequently, many implementations of the OpenMP standard rely on a lightweight threading library, such as pthreads [42].

In OpenMP, parallelism is achieved at the loop-level by annotating loop structures with compiler directives (e.g., `#pragma omp parallel for`), such that the compiler can thread each iteration for execution in parallel. In MPI, parallelisation must be implemented explicitly, with each process initiating inter-process communications directly through the application programming interface (API).

Parallelisation beyond a single node requires inter-node communications. The de facto standard in HPC is MPI. MPI provides a number of functions for distributed computation, including point-to-point communications, one-sided communications, collective operations and reduction operations. In an MPI-parallelised program, each process operates on its own data, and communicates edge values to surrounding processes where a dependency exists.

There are also a number of programming models that treat the distributed memory space as a single homogeneous block. This partitioned global address space (PGAS) approach is taken by **Coarray Fortran** [43] and **Unified Parallel C** [44], among others. In this model, communications are hidden to the application developer, but are typically implemented using MPI in the backend library.

An alternative to the data parallelism approaches described above, is task parallelism. In a task parallel approach, the work of the application is divided into tasks and then scheduled for execution across a distributed architecture. Tasking was introduced in version 3.0 of the OpenMP standard, and the standard now includes functionality for standard tasks, single tasks, task dependencies and task groups [40]. Tasking has been developed further by a number of asynchronous many-tasking (AMT) frameworks. Notable examples include **Charm++** [45], **Legion** [46], **HPX** [47], and **DARMA/vt** [48].

4.1. Accelerator Extensions

For heterogeneous systems, host code is usually written using the programming languages and models mentioned previously to coordinate between compute nodes; however, the accelerators themselves often require a different approach. This is a consequence of the significant differences in accelerator architectures compared to traditional CPUs; indeed their simplified architecture often restricts developers to a subset of language features.

Each vendor typically offers their own platform-specific programming model, such as **CUDA** from NVIDIA and **HIP/ROCm** from AMD. Moving between hardware vendors therefore requires moving between programming models. In some cases, this may be as simple as finding matching API calls, and in some cases this may require significant re-engineering [49, 50].

Although proprietary, CUDA has been the most dominant accelerator programming extension and has maintained a high level of adoption in HPC given the widespread use of NVIDIA GPU hardware and the maturity and support that NVIDIA put into the numerical solver libraries based on CUDA (e.g., cuBLAS, cuFFT) [51]. The HIP programming model provides a near like-for-like API, and a similar number of numerical solver packages [52]. Both follow a single instruction, multiple thread (SIMT) programming model where large numbers of threads are launched simultaneously (in contrast to SIMD, this allows for thread divergence and additional complexity).

OpenCL [53] largely mirrors the SIMT model of CUDA, having a similar API, but is developed as an open standard. Much like with CUDA and ROCm/HIP, in OpenCL the programmer is given the opportunity to write explicit computational kernels for devices, with significant control over the orchestration of parallelism. OpenCL is supported by all major vendors (Intel, AMD, NVIDIA) to some extent, and has been promoted as a vendor agnostic model. However, the same OpenCL application will not necessarily give the best performance on all architectures, where some level of device specific optimisations are required to obtain best performance.

While offering much less control, **OpenACC** [54] directives can be used to indicate/instruct a compiler where code can be parallelised for execution on an accelerator. OpenACC also provides directives to indicate whether memory should be allocated on the host or the device, and when to move data between the two. Memory management, such as when data is moved to/from the device, and how often, are key considerations to achieving good performance. If not handled correctly, directives can lead to frequent data movement to/from a device and lead to significant slowdowns. Currently OpenACC is provided in commercial compilers from NVIDIA (previously PGI) and Cray, with the latter only supporting Cray-supplied hardware. GCC also offers nearly complete support for OpenACC 2.5, targeting both NVIDIA and AMD devices.

OpenMP added support similar to OpenACC for offloading computation to accelerators in version 4.0 of the standard [38]. Similar to its counterpart, data locality is controlled through compiler directives, with parallelisable loops being specified using the `#pragma omp target` directive. OpenMP 4.0 is a good example of standards attempting to catch up with evolving hardware, where support for accelerator directives (which were introduced as a proprietary solution first in 2011 with OpenACC with the adoption of NVIDIA GPUs in HPC) were only added to the OpenMP standard in 2013.

Even then OpenMP supporting compilers took several years more to fully implement the standard for working code [55].

Subsequent OpenMP standards (e.g., OpenMP 4.5 [39] and OpenMP 5.0 [40]) have further improved support for accelerator devices, making target offload a viable path to producing performant GPU applications. Support for target offload can be found in commercial compilers from Intel, IBM, AMD and Cray, with a variety of target architectures. Support also exists in the Clang/LLVM [56] and GCC open-source compilers[2].

While the explicit device control provided by the CUDA, ROCm/HIP, and OpenCL programming models may be more powerful than directive-based approaches, it may also significantly increase developer effort. More recently, the Khronos Group released **SYCL** [57], a new high-level cross-platform abstraction layer, which can be viewed as a data-parallel version of C++ inspired by OpenCL. Many of the concepts remain the same, but the significant amount of "boiler-plate" code required to setup parallelism in OpenCL applications is now not required where SYCL uses a heavily-templated C++ API. In this way, SYCL is also able to offer a *single-source* solution, similar to OpenMP and OpenACC, where kernel code and host code can co-exist within an application (unlike in OpenCL, where application code is typically split between host and device code).

In SYCL, there is typically a queue that work items can be submitted to. Loop-level parallelisation is achieved using constructs such as the `parallel_for` function, while task-parallelism can be achieved using the `single_task` function.

Building on SYCL, Intel announced their new programming model, **OneAPI**, in 2018. OneAPI is a unified programming model, that combines several libraries (e.g., the Math Kernel Library), with **Thread Building Blocks (TBB)** and **Data Parallel C++ (DPC++)**. DPC++ is a cross-architecture language built upon the C++ and SYCL standards, providing some extensions to SYCL. Support for SYCL and DPC++ is provided in a number of compilers from vendors such as AMD, Intel, and Codeplay, and can target a number of device types directly, or via existing OpenCL runtimes[3]. In the case of the Intel and AMD compilers, it is even possible to use SYCL to target FPGA devices. However, the question of whether one code written in SYCL is able to obtain the best performance on all supported hardware remains to be answered [58–61].

In addition to the aforementioned programming models, there are a number of software ecosystems that target both host and accelerator platforms, that present themselves as parallel programming models to developers. Notable examples are OCCA [62], Kokkos [63], RAJA [64] and Alpaka [65]. These are discussed in more detail in Section 5.

Parallelisation based on OpenMP and MPI have a long history in HPC application development. CUDA also now has over a decade of development, with OpenACC, and OpenCL following close behind. SYCL/DPC++ is the latest addition to the parallel programming extensions available. While CUDA, OpenMP, and OpenACC all support C/C++ and Fortran, OpenCL and SYCL only support C/C++. If, indeed, C/C++-based

[2] `https://www.openmp.org/resources/openmp-compilers-tools/`
[3] `https://www.khronos.org/sycl/`

extensions and frameworks dominate the parallel programming landscape for emerging hardware, there could well be a need for porting existing Fortran-based applications to C/C++.

The key considerations and challenges when using the above programming models and extensions to general purpose languages include:

1. Open standards lagging hardware development – especially when the standard is developed by a large number of organisations.
2. The *complete* implementation of these standards into many compilers can be slow [55].
3. Some of these programming models offer low-level fine control over parallelism and therefore may lead to overly complex code. In some cases, different code-paths are required to get the best performance on different architectures [59], for example, to handle the different memory layouts required to optimise for CPUs vs. GPUs.

5. Software Libraries

In this class we consider libraries that facilitate scheduling and execution of data parallel or task-parallel algorithms, and classical software libraries that target scientific application development, implementing a diverse set of common algorithms (often numerical).

5.1. Parallel Programming Abstraction Libraries

There are a number of software libraries that present themselves similarly to the parallel programming models mentioned previously. They typically offer parallel functionality through various methods such as macro preprocessing, JIT compilation or template metaprogramming.

OCCA [62] is one such vendor neutral, open source parallel programming framework focused on portability. OCCA supports both C/C++ and Fortran, and can target platforms through a range of backends (e.g., CUDA, HIP, DPC++, OpenMP, OpenCL).

An approach, exclusive to C++, is the use of template libraries, that enable developers to write a generic "template" to express an operation such as a parallel-loop iteration, but at compile-time select a specific implementation of a method or function (known as static dispatch). This allows users to express algorithms as a sequence of parallel primitives executing user-defined code at each iteration, e.g., providing a loop-level abstraction. These libraries follow the design philosophy of the C++ Standard Template Library [66] – indeed, their specification and implementation is often considered as a precursor towards inclusion in the C++ STL. The largest such projects are **Boost** [67], **Eigen** [68], **Thrust** [69] and **HPX** [47]. While there are countless such libraries, here we focus on ones that also target performance portability in HPC.

Three notable examples of this approach are **Kokkos** [63] and **RAJA** [64], developed as part of the US Department of Energy's Exascale Computing Project, and **Alpaka** [65], developed as part of the PIConGPU Particle-in-Cell application [70].

Each library supports execution on shared-memory parallel platforms, such as CPUs and GPUs, through a variety of approaches. They do not provide parallelism on distributed memory systems, rather they are designed to be used in conjunction with MPI (or another off-node communication library).

Kokkos is a C++ performance portability layer, developed by Sandia National Laboratories, that provides data containers, data accessors, and a number of parallel execution patterns [63]. Its data structures can describe where data should be stored (CPU memory, GPU memory, non-volatile, etc.), how memory should be laid out (row/column-major, etc.), and how it should be accessed. Similarly, one can specify where algorithms should be executed (CPU/GPU), what algorithmic pattern should be used (parallel for, reduction, tasks), and how parallelism is to be organised. It is a highly versatile and general tool capable of addressing a wide set of needs, but as a result is more restricted in what types of optimisations it can apply compared to a tool that focuses on a narrower application domain. Kokkos is able to target CUDA, OpenMP, pthreads, HIP or SYCL, meaning it can target all of the post-Exascale platforms currently deployed or in development.

RAJA is a similar abstraction developed by Lawrence Livermore National Laboratory [64]. It is similar to Kokkos in many respects, but offers more flexibility for manipulating loop scheduling, particularly for complex nested loops. It supports CPUs (with OpenMP and TBB), as well as NVIDIA GPUs with CUDA.

Alpaka is a header-only C++17 abstraction library developed initially at TU Dresden [65]. It provides similar semantics to both Kokkos and RAJA, and can achieve parallelism through OpenMP, std::thread, TBB, CUDA, HIP and SYCL.

Both Kokkos and RAJA were designed by Department of Energy laboratories to help move existing software to new heterogeneous hardware, and this is very much apparent in their design and capabilities – they can be used in an iterative process to port an application, loop-by-loop, to support shared-memory parallelism. Of course, for practical applications, one needs to convert a substantial chunk of an application; on the CPU that is because non-multithreaded parts of the application can become a bottleneck, and on the GPU because of the cost of moving data to/from the device. Kokkos and RAJA are used heavily within the Exascale Computing Project (ECP) [21, 71], and due to their reliance on template meta programming, can be used alongside almost any modern C++ compiler.

Using parallel programming libraries comes with the following considerations:

1. Future support for, and maintenance of, these libraries is not guaranteed.
2. Some libraries may restrict application development to modern C++.
3. Development time may be high due to the compilation times associated with heavily templated code.
4. Debugging heavily templated code can be difficult, with errors obfuscated by numerous templates. This can be particularly problematic for inexperienced programmers.
5. Platform specific code can be easily integrated into templated code to achieve higher performance on some platforms, provided that the abstraction used is carefully designed and at a sufficiently high level.

5.2. Numerical Algorithm Libraries

Beyond the programming models mentioned previously, portability can also be achieved using libraries provided by various vendors. These software libraries typically provide common functionality, and are often highly optimised for particular architectures.

The basis of many mathematical libraries is the **Basic Linear Algebra Subprograms (BLAS)** [72], first developed in 1979. BLAS provides vector operations, matrix-vector operations and matrix-matrix operations. The **Linear Algebra Package (LAPACK)** [73] builds on BLAS, providing routines for solving systems of linear equations. The **FFTW** [74] library provides functions for computing discrete Fourier transforms, and is known to be the fastest free software implementation of the FFT.

Architecture-tuned implementations of BLAS, LAPACK and FFTW are often available, with notable examples being AMD Optimized CPU Libraries, ARM Performance Libraries, Intel Math Kernel Library, cuBLAS, clBLAS, OpenBLAS, and Boost.uBLAS. Similarly, **MAGMA** [75] provides dense linear algebra kernels for multicore and accelerator architectures.

The **Portable, Extensible Toolkit for Scientific Computation (PETSc)** [76] provides a number of data structures and routines for solving PDEs. It was developed by Argonne National Laboratory and employs MPI for distributing algorithms across an HPC system. Recently, PETSc has implemented an abstraction layer for scalable communications over MPI and between host and GPU devices, PetscSF [77].

Similarly, **HYPRE** [78] is a library of data structures, preconditioners and solvers developed at Lawrence Livermore National Laboratory. It can be built with support for GPU devices through CUDA, OpenMP target offload, or using RAJA or Kokkos.

Trilinos [79] is an extensive collection of open-source libraries that can be used to build scientific software, developed by Sandia National Laboratories. It provides over 50 self-contained, independent packages for solving linear and non-linear systems, preconditioning, and using sparse graphs and matrices. It supports distributed memory computation through MPI and also provides shared memory computation through the Kokkos package.

Achieving high performance with many solvers requires that a problem is well partitioned and load balanced. Notable examples are **METIS** [80] (and its parallel incarnation **ParMETIS** [81]), **Scotch** [82] (and the parallel **PT-Scotch** [83]), and **KaHIP** [84]. Trilinos provides the **Zoltan2** [85] package that can perform partitioning through ParMETIS, PT-Scotch, or its own GPU partitioner, Sphynx.

Using these libraries introduces a number of key considerations and challenges:

1. While the standard interfaces to these libraries may restrict their usefulness to some applications, it does encourage vendors to produce optimised *and portable* versions of performance critical functions.

2. Library functions often operate in lock-step, meaning operations cannot typically be fused. This may necessitate a number of unnecessary CPU-GPU transfers.

6. Domain Specific Languages

In this class we consider a wide range of languages and libraries – the key commonality is that their scope is limited to a particular application or algorithmic domain.

Domain Specific Languages (DSLs) and approaches by definition restrict their scope to a narrower problem domain, set of algorithms, or computation/communication patterns. By sacrificing generality, it becomes feasible to attempt and address challenges in gaining all three of performance, portability and productivity. A wide range of approaches exist, at different levels of abstractions starting from libraries focusing on specific numerical methods (e.g., Finite Element method) to low-level parallel computation patterns and loop abstractions. Some are embedded in general purpose languages (eDSLs) such as C/C++/Fortran or Python allowing them to make use of the compiler and development infrastructure (debuggers and profilers) of these languages. Others develop an entirely new language of their own.

Restricting to a specific domain allows DSLs to apply more powerful optimisations to help deliver performance as well as portability. The key reason being that a lot of assumptions are already built into the programming interface (the domain specific API). As such, explicit description of the problem need not occur when programming with DSLs, significantly improving productivity. Conversely, the key deficiency of DSLs then is their limited applicability – if they cannot develop a considerable userbase, they will lack the support required to maintain them. As such two of the key challenges to building a successful DSL or framework are: designing an abstraction wide enough to cover a range of applications, but narrow enough such that powerful optimisations can be applied; and, ensuring there is a feasible approach to long-term support.

DSLs can be categorised based on their level of abstraction. In some respects, many of the solutions discussed previously in this paper could be thought of as low-level DSLs, providing abstractions for parallel computation (e.g., parallel for loops, parallel reductions, etc.); SYCL, Kokkos, and RAJA could be thought of as such low-level DSLs.

However, in this review, we consider low-level DSLs to be those that are designed for a particular class of computational pattern (e.g., mesh-based computations, particle-based interactions), and high-level DSLs to be those that allow mathematical descriptions of problems to be directly expressed in code, leaving the problem discretisation and computational method to the solver/framework.

6.1. Low-level Domain Specific Languages

6.1.1. Mesh-based Computations

This class of DSLs are, for the most part, oblivious to the numerical methods being implemented, which in turn allows them to be used for a wider range of algorithms, e.g., finite differences, finite volumes, or finite elements. The key goal here is to create an abstraction that allows the description of parallel computations over either structured or unstructured meshes (or hybrid meshes), with neighbourhood-based access patterns.

There are a number of notable and currently active DSLs at this level of abstractions. **Halide** [86] is a DSL intended for image processing pipelines, but generic enough to target structured-mesh computations [87]; it has its own language, but is

also embedded into C++, and it can target both CPUs and GPUs, as well as distributed memory systems. **YASK** [88] is a C++ library for automating advanced optimisations in stencil computations, such as cache blocking and vector folding. It targets CPU vector units, multiple cores with OpenMP, as well as distributed-memory parallelism with MPI. **OPS** [89] is a multi-block structured-mesh DSL embedded in both Fortran and C/C++, targeting CPUs, GPUs and clusters of CPUs/GPUs – it uses a source-to-source translation strategy to generate code for a variety of parallelisations. **ExaSlang** [90] is part of a larger European project, ExaStencils [91], which allows the description of PDE computations at many levels – including at the level of structured-mesh stencil algorithms. It is embedded in Scala and targets MPI and CPUs, with limited GPU support. Another DSL for stencil computations, **Bricks** [92], gives transparent access to advanced data layouts using C++, which are particularly optimised for wide stencils, and is available on both CPUs and GPUs.

OP2 [93] and its Python derivative, **PyOP2** [94], give an abstraction to describe neighbourhood computations for unstructured meshes. They are embedded in C/Fortran and Python respectively, and can target CPUs, GPUs, and distributed memory systems. Unlike the structured-mesh motif (which uses a regular stencil), unstructured-mesh computations are based on explicit connectivity information between mesh elements, leading to indirect increments. Indirect increments need to be carefully handled when parallelising, given the existence of data dependencies, and as such need different code-paths to obtain the best performance on different architectures [59]. OP2 generates parallel code targeting CPU and GPU clusters making use of a range of parallel programming models (e.g., SIMD, OpenMP, CUDA, SYCL, and their combinations with MPI).

A number of DSLs have emerged from the weather prediction domain, such as **STELLA** [95] and **PSyclone** [96]. STELLA is a C++ template library for stencil computations that is used in the COSMO dynamical core [97], and supports structured-mesh stencil computations on CPUs and GPUs. PSyclone is part of the effort in modernising the UK Met Office's weather code; it works with LFRic, which is the Met Office' replacement for its Unified Model. PSyclone has code-transformation and automatic code generation functionalities, and as well as LFRic, it has been used with the NEMO and CROCO ocean models [98]. PSyclone is capable of generating OpenACC, OpenMP and (in limited cases) OpenCL [99]. Its transformation functionality allows for CPU optimisations, as well as targetting GPUs. **CLAW-DSL** [100], used for the ICON model [101], targets Fortran applications and generates CPU and GPU parallelisations – mainly for structured-mesh codes, but it is a generic tool based on source-to-source translation using preprocessor directives. It is worth noting that these DSLs are closely tied to larger software projects (weather models in this case), developed by state-funded entities, greatly helping their long-term survival. At the same time, it is unclear if there are any other applications using these DSLs.

6.1.2. Particle Interactions

Within the plasma domain, particle simulations are typically performed using the particle-in-cell method [102], where individual particles (or macro particles) are tracked across a mesh representing the electromagnetic (or electrostatic) fields. Particles may interact only with the mesh (known as particle-mesh methods, or **PM**), with other par-

ticles (known as particle-particle, or **PP**), or with both (known as **PP-PM**, or **P^{3}M**). There are a number of DSLs that exist at this level of abstraction, proving support for one or more of these methods.

PPML [103] is a DSL for numerical simulations based on particle methods and hybrid particle-mesh methods. PPML provides a concise set of high-level abstractions through its own language, targeting reduced implementation times, and uses a source-to-source compiler to generate plain Fortran code, which is then linked with the PPML back-end. However, the use of new language constructs limits the standard optimisation/debugging routines and is hard to extend and maintain. The **Parallel Particle-Mesh Environment (PPME)** [104] overcomes some of these issues by leveraging a meta programming system to enhance the programmer's experience, which is developed as an Integrated Development Environment (IDE) for particle-mesh methods. **OpenFPM** [105] is a framework that provides an abstraction layer for mixed mesh-particle, and particle methods, embedded in C++. It provides a comprehensive library that targets CPUs, GPUs, and supercomputers. The **HartreeParticleDSL**[4] is a DSL for particle methods implemented using the Regent programming language. It currently only supports a limited feature set, catering for short-range pairwise interactions and per-particle operations.

While many of the particle DSLs listed here are general to PM, PP or P^{3}M methods, some are designed with a specific scientific domain in mind, in particular N-body and molecular dynamics applications. **HOOMD-blue** [106] is a general-purpose molecular dynamics and hard particle Monte Carlo simulation toolkit. It provides a Python-based domain specific API and runs over a high-performance C++/CUDA back-end with MPI, enabling the creation of simulation and analysis workflows. **MDL** [107] is a molecular-dynamics DSL designed to allow rapid prototyping, testing and debugging of efficient propagation algorithms. **PPMD** [108] is a portable framework for molecular dynamics applications, providing a Python interface and parallelised computation using OpenMP, MPI and CUDA.

At a level lower, the **Cabana** [109] library provides a number of data structures, algorithms and utilities specifically for particle-based simulations. Parallel execution of particle kernels is achieved through Kokkos for on-node parallelism and MPI for off-node communication. Each of these libraries can be used to abstract away some of the mathematical operations and data storage requirements needed by scientific applications.

6.2. High-Level DSLs

Domain specificity can be at a higher level, where the DSL focuses on the declaration and solution of particular numerical problems. The most widely implemented DSLs at such a high level are frameworks for the solution of PDEs. The problem is specified starting at the symbolic expression of the problem (e.g., in Einstein notation). An interpreter or a compiler then (semi-) automatically discretises the problem and generates a solution. Most are focused on a particular set of equations and discretisa-

[4] `https://github.com/stfc/RegentParticleDSL`

tion methods, and they can offer excellent productivity – assuming the problem to be solved matches the focus of the library.

Many of these libraries, particularly ones where portability is important, are built with a layered abstractions approach; the high-level symbolic expressions are transformed, and then passed to a layer that maps them to a discretisation, then this is given to a layer that arranges parallel execution – the exact layering of course depends on the library. This approach allows the developers to work on well-defined and well-separated layers, without having to gain a deeper understanding of the whole system. These libraries are most commonly embedded in the Python language, which has the most commonly used tools for symbolic manipulation in this field – although functional languages are arguably better suited for this, they still have little use in HPC. Due to the poor performance of interpreted Python, these libraries ultimately generate low-level C/C++/Fortran code to deliver high performance.

One of the most established such libraries is **FEniCS** [110], which targets the finite element method. However it only supports CPUs and distributed memory cluster execution with MPI, with some support for GPUs with the integration of PETSc. **Firedrake** [111] is a similar project with a different feature set, that uses the aforementioned PyOP2 library for parallelising and executing generated code. A feature of Firedrake is that it generates code at runtime to exploit further optimisation opportunities, for example based on the mesh being available/input at runtime.

The **ExaStencils** project [91] uses four layers of abstraction to create code running on CPUs or GPUs from the continuous description of the problem – its particular focus is structured meshes and multigrid. **OpenSBLI** [112] is a DSL embedded in Python, focused on resolving shock-boundary layer interactions and uses finite differences and structured meshes – it generates C code using the OPS library which provides the stencil abstraction. As noted before, OPS then generates parallel code targeting distributed memory machines with both CPUs and GPUs. **Devito** [113] is a DSL embedded in Python which allows the symbolic description of PDEs, and focuses on high-order finite difference methods, with the key target being seismic inversion applications. Devito also supports CPU and GPU parallelisation, where GPU acceleration is obtained by generating OpenACC directives.

In fusion research, the **BOUT++** [114, 115] framework has been developed as a flexible toolbox for solving a wide range of PDEs. Its design was in large part driven by the need for physicist users to modify and customise the model equations being solved. BOUT++ therefore uses C++ features to implement models in a way which closely mimics their mathematical form. The BOUT++ framework then solves these equations, and allows the user runtime control over the finite difference methods and stencils used, as well as time integration solver, Laplacian inversions, and so on.

BOUT++'s physics model implementation language is an example of an eDSL, in this case C++ is the host language. eDSLs have the advantage of the user/developer being able to easily "break out" of the DSL and write generic code for situations not handled by the DSL, for example to handle complicated boundary conditions. The cost of this approach is that certain transformations of the code are harder to achieve. For example, each physics and arithmetic operator in BOUT++ contains a loop over the whole domain for its own kernel. To achieve the full performance with OpenMP or accelerators requires merging these loops into a single loop. This in turn necessi-

tates rewriting the top-level set of equations to include this loop explicitly, or to use something akin to expression templates (as is done in libraries such as Eigen [68] or Blitz++ [116]), which have their own downsides.

BOUT++ currently only supports execution on CPUs with OpenMP for multi-threading and MPI for distributed memory execution. Experimental branches exist with ongoing development to support GPU execution. These include (1) using Hypre [78] with GPU support for the Laplacian inversion parts of the problem (which in practice can take about half the total time) and (2) with RAJA for putting the user physics model on GPUs, with Umpire [117] to handle memory. This requires modifying the physics DSL to enable operations to be fused together, reducing the number of separate kernels which need to be launched.

Similar to BOUT++, the **Unified Form Language (UFL)**, used in FEniCS and Firedrake provides a high-level language to describe variational forms. The problem to be solved is specified at a high level, which corresponds closely to the mathematical form.

Firedrake uses the FEniCS Form Compiler (FFC) to convert UFL to an intermediate representation, and then uses PyOP2 to generate code for target architectures, aiming to be performance portable on both CPUs and GPUs.

The most common challenges when using DSLs include:

1. Difficulties in debugging due to the extra hidden layers of software between user code and code executing on the hardware. However, DSLs generating low-level C/C++/Fortran codes can use standard debuggers or profilers.

2. Extensibility – implementing algorithms that fall slightly outside of the abstraction defined by the DSL can be an issue.

3. Customisability – it is often difficult to modify the implementation of high-level constructs generated automatically.

To mitigate some of these issues, systems can be provided with "escape hatches", which provide ways for users to implement components of the problem which cannot be expressed in the high-level DSL. An example is custom flux-limiters, which cannot currently be expressed in UFL; instead a user needs to be able to implement their own kernels, and integrate these into the remainder of the system in an elegant way. Firedrake provides such escape hatches for direct access to linear algebra operators (PETSc), and allows implementation of custom PyOP2 kernels. However it should be noted that such modifications may not deliver the best performance on all hardware and should be used only sparingly, or for prototyping. As is the case with many complex performance issues there is no silver-bullet to solve all cases.

7. Coupling Frameworks

In this class, we consider libraries acting as interfaces to enable communication between several applications to perform multiscale simulations.

The multiscale problem of fusion modelling tackled by NEPTUNE requires the coupling of physics at various length- and time-scales to predict and control instabilities in the edge-region that influence the plasma core (see Figure 1). This requires efficient coupling between various solvers that represent physics at different scales/regimes (continuum-kinetic coupling), such as the **SOLPS-ITER** [118] code for simulating tokamak edge physics. Coupling libraries must be flexible and lightweight to provide developers the ability to optimise the performance and portability of each separate code. Classically, developers have needed to hard-code various solvers together to perform coupled simulations [119], or use file-based coupling (which may introduce an increased I/O overhead that usually hinders performance but may be useful for debugging) [120]. However, several third-party libraries are now available to couple a potentially arbitrary number of solvers used to model various physical systems. Common problems driving these developments include fluid-structure interaction [121], conjugate heat-transfer [119], and aeroacoustics [122]. The suite of modules developed as part of the **Multiphysics Object-Orientated Simulation Environment (MOOSE)** framework [123] have been used for coupled problems in nuclear fusion such as the breeder blanket [124]. With the provision that data provided to the coupling middleware remains consistent, each solver can be developed in isolation of the other; this could be thought of as implementing a *horizontal* separation of concerns approach, akin to the vertical separation of concerns achieved by aforementioned DSLs [125].

One strongly developed coupling library is **preCICE** [121, 126]. PreCICE aims to couple existing solvers together, creating what is known as 'partitioned' simulations. Users couple simulation codes through adapters which interface to libprecice. These adapters are standalone software packages that may either be provided by preCICE, the preCICE community, or may be user-defined for in-house solver methods. These community contributions are important to enable developer productivity. Recently, preCICE was used to couple a GPU lattice-Boltzmann method solver coupled to OpenFOAM (CPU-based CFD code), showing it can be used to couple heterogeneous simulation codes [127]. The communication between simulations coupled by preCICE is recommended to be done based on TCP/IP sockets by the developers. Although MPI is an available option for communication, Rubin [128] attempted to configure their simulations to communicate with MPI but were unsuccessful. Uekermann [129] showed that MPI-based communication is between 5 and 10 times faster than TCP/IP sockets.

The **Coupling With Interpolation Parallel Interface (CWIPI)** [130] and the **Multiscale Universal Interface (MUI)** [119] libraries perform coupling using the multiple-programme multiple-data (MPMD) MPI model. A primary benefit of MUI is the availability of examples and documentation provided by the developers, compared to a lack of documentation or examples for CWIPI (aside from open-source contributions such as Moratilla-Vega et al. [122] and, within the NEPTUNE project, the successful demonstration of using CWIPI for the coupling of fluid and particle regions for examples of interest to the fusion community [131, 132]). Additionally, MUI represents data as a cloud of points whereas CWIPI represents data as a mesh. The additional information on mesh connectivity may not be necessary in particle-based solvers that are necessary for fusion reactor modelling, making the cloud-based representation preferred for its generality [133]. The MUI software is written in C++11 with wrappers for C, Fortran and Python. A header-based approach is used for the en-

tire library and the only external library is MPI. As such, it can be used in the same way any other C++ standard library would be used, without the need for pre-compilation.

Another potential option for code coupling is the **Adaptive I/O System (ADIOS)** [134, 135], developed as part of ECP, which is primarily concerned with high-performance I/O. ADIOS yields minimal computational overhead, and also allows seemless switching between file-based coupling and in-memory coupling for debugging and full-scale runs, respectively [120]. ADIOS has been used for coupling the edge physics code **XGC (X-point Included Gyrokinetic Code)** [136] with several core plasma and stability analysis codes [137, 138] as part of the ECP-funded **Whole Device Model Application (WDMApp)** [4, 139]. WDMApp aims to develop easy coupling for core and edge plasma codes, improve the performance of the XGC edge physics code, and allow for integration of in-situ processing. ADIOS has also been used in the CFD community for *in-situ* post-processing and analysis [140], suggesting it would be a suitable tool for developers looking to integrate their software with machine learning approaches such as reduced-order models, without writing to disk.

Through the use of third-party coupling frameworks, developers can optimise their code for maximal performance on HPC with minimal intrusion to the codebase of each application. Specific advantages include advanced coupling approaches such as implicit coupling, high-order data mapping (e.g., interpolation via splines) and non-matching time-stepping without implementing these approaches in their own software. This also results in increased potential for scalability optimisations using varying domain decomposition strategies. For example, load-balancing algorithms can be applied separately to each application domain, individual computational grids can be redefined, coarsened or refined, and the different applications can even be run on separate platforms such as GPUs if desirable.

Using coupling libraries for multiscale simulation introduces the following considerations:

1. Performance and scalability of the communication and coupling numerics when coupling various applications.

2. The ability of the coupling framework to easily be incorporated with existing codes with minimal intrusion is crucial to maximise developer productivity.

3. Coupling features available with the chosen library (various time-stepping or interpolation/mapping schemes may be desirable). Offloading the implementation of these features to coupling library developers is key for productivity.

8. Evaluating Performance, Portability and Productivity

When considering the development of a new simulation application, we are typically interested in maximising performance, portability and productivity [8]. In terms of performance we are usually concerned with metrics that directly measure or affect the "time-to-science", while for portability we are usually concerned with an application's ability to run correctly on different HPC systems and architectures. Productivity, on the other hand, is usually a measure of the time and expertise required to develop and maintain the application.

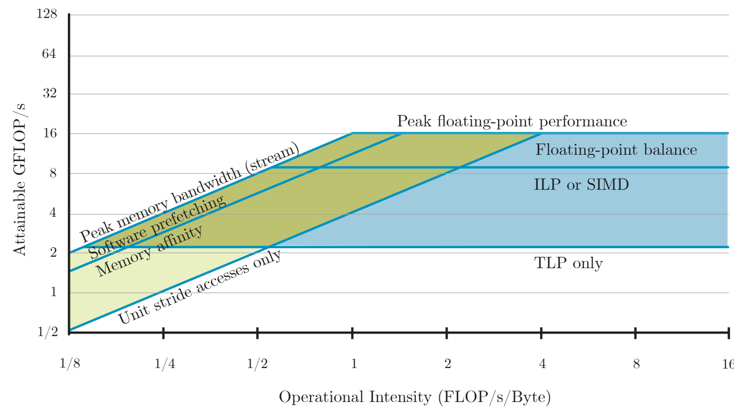


Figure 3: A Roofline plot for an AMD Opteron X2 processor [142].

There are a wide variety of metrics for assessing the performance of an application, common examples include application runtime, floating-point operations per second, or memory bandwidth. Portability could be thought of as a binary decision – an application either does or does not run correctly. Productivity is perhaps the most difficult to assess objectively, but common proxies include lines-of-code written, development time (in person-hours), and code complexity metrics [141].

These metrics (and combinations of them) allow us to analyse and reason about an application, in order to evaluate development and optimisation strategies. One of the most notable methods of performance analysis is the Roofline model [142]. In a roofline model, numerous rooflines are drawn that correspond to the various peaks of floating point performance (horizontal lines) and the maximum memory bandwidth (diagonal lines).

Figure 3 (reproduced from Williams et al. [142]) shows the data for an AMD Opteron X2. The performance of an application, or an individual computational kernel, can be placed on the plot, showing if an application is compute-bound (blue region), memory-bound (yellow region), or both (green region), and can therefore suggest possible routes for performance optimisation (e.g., focus on improving use of ILP, improving memory cache behaviour). Rooflines can be calculated using published data (from processor specifications), or benchmarked empirically using tools such as Intel Advisor, or the Empirical Roofline Toolkit [143]. An application's performance can then be placed on the plot using data gathered from performance counters.

Although portability itself is a binary measure, we are usually concerned with how *performant* a portable application is – arguably an application that runs correctly but with significantly degraded performance on an alternative architecture is not truly portable. This has led to an effort to develop a multi-objective optimisation figure of merit to assess *performance portability*.

One example is the metric introduced by Pennycook et al. [144].

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

In Equation (1), the performance portability ($\mathbf{\Phi}$) of an application a, solving problem p, on a given set of platforms H, is calculated by finding the harmonic mean of an application's performance efficiency ($e_i(a, p)$). The performance efficiency for each platform can be calculated by comparing achieved performance against the best recorded (possibly non-portable) performance on each individual target platform (i.e. *the application efficiency*), or by comparing the achieved performance against the theoretical maximum performance achievable on each individual platform (i.e. *the architectural efficiency*). Should the application fail to run on one of the target platforms, a performance portability score of 0 is awarded.

Harrell et al. propose a code divergence metric as a measure of developer productivity [145], and this measure has been adopted by Code Base Investigator [146] and the P3 Analysis Library [147].

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

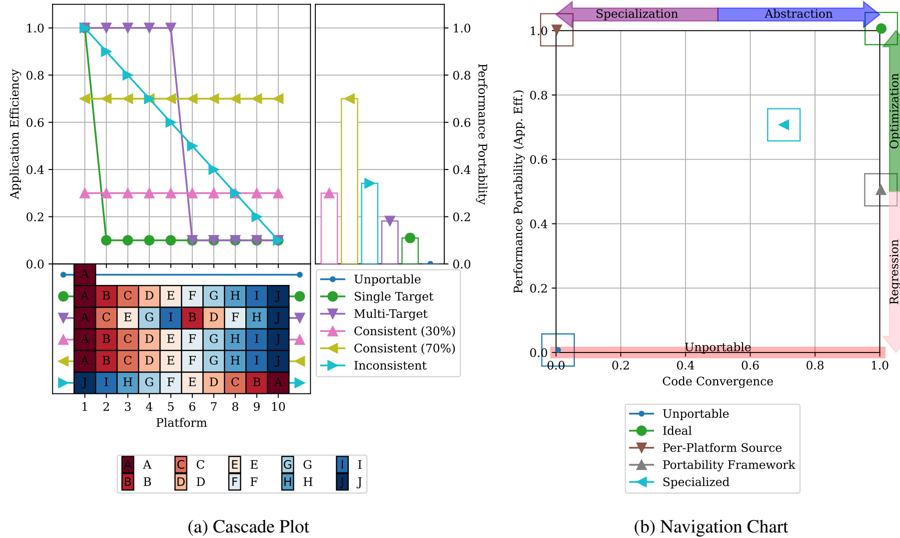
In Equation (2), code divergence is a measure of the average "distance" between the source code required to compile an application a, and execute problem p for each pair of platforms in H, where $d_{i,j}(a, p)$ is any distance metric between two source codes. Pennycook et al. [148] suggest using the Jaccard distance (shown in Equation (3)), where $c_i(a, p)$ represents the set of source lines required to compile application a and execute problem p on a given platform i.

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

Similarly to $\mathbf{\Phi}$, this metric provides a single value in the range [0, 1], where 0 means a single source code is used for every platform, and 1 means an entirely separate source code is required with no shared code.

While Equations (1) and (2) provide formal definitions for performance portability and productivity, these single value metrics may not answer all questions a developer might have about their application. In recognising this a series of visualisations of performance portability and productivity have been proposed by Sewall et al. [149] and Pennycook et al. [148]. These visualisations are best explained with examples.

Figure 4(a) shows a cascade plot of 6 hypothetical application implementations, across 10 platforms. The implementations are: **unportable** with high performance on a single platform, but not portable to any other platform; **single target** with high performance on a single platform, and low performance on all others; **multi target** achieving high performance on some platforms, and low performance on others; **inconsistent**



(a) Cascade Plot

(b) Navigation Chart

Figure 4: Example plots showing how the performance, portability and productivity of an implementation may be visualised [147, 148]

showing a range of performance across all platforms; and **consistent** showing consistent low (30%) or high (70%) performance across all platforms.

While we could simply apply the Φ metric in Equation (1) to this synthetic data, doing so would mean that we lose some information about how the performance portability is spread across platforms, and how the application efficiency changes as we add and remove platforms from the evaluation set.

Figure 4(b) shows a performance-portability code-convergence (Φ-CC) navigation chart with 5 hypothetical application implementations. The implementations correspond to: an **unportable** implementation entirely tailored for a single hardware target; an **ideal** implementation that achieves $\Phi = 1.0$, from a single codebase; a **per platform source** implementation that achieves $\Phi = 1.0$, but does so with specialised code for each hardware target; a **portability framework** implementation using a parallel programming model aimed at portability, but only achieving $\Phi = 0.5$; and a **specialized** implementation that strikes a balance between portability and specialisation, achieving 0.7 in each.

Metrics and visualisations such as these provide vital information when planning for the development of a new code, and for evaluating progress in developing a code that maximises performance, portability and productivity.

8.1. Previous Studies of Performance, Portability and Productivity

There are currently a large number of projects focused on preparing scientific applications for the complexities of post-Exascale computation. With many of the largest Supercomputers edging towards heterogeneity and hierarchical parallelism, many of these efforts are in ensuring that applications are performant *and portable* between dif-

ferent architectures. The previous five sections of this paper have outlined a wide number of options available for developing performance portable applications, and each approach comes with various advantages and disadvantages.

In this paper we are primarily concerned with applications and algorithms that can be used to simulate the behaviour of plasma, using either a fluid- or a particle-based scheme. Table 1 lists a number of applications implementing common algorithms used for the simulation of plasma that have been the subject of performance and portability studies.

8.2. General Purpose Programming Languages

In many cases, the applications listed in Table 1 have a "reference" implementation written only using a general purpose programming language. These reference versions serve as a baseline for performance and portability studies, often only exhibiting good single core performance on CPUs, with no portability to heterogeneous architectures. This is certainly typical in the mini-application space, where developers are encouraged to demonstrate portable performance with miniaturised representative applications, prior to larger porting efforts. Notable examples of this are BookLeaf, CloverLeaf, and TeaLeaf from the UK Mini-App Consortium (UK-MAC) [151, 152, 180], and miniFE, Nekbone, and Laghos from the ECP Proxy Apps [168, 171, 175].

In the case of large applications such as EPOCH and nek5000, the applications only exist written in these general purpose programming languages, parallelised using the MPI programming model. These typically have no portability to heterogeneous systems, and instead are subject to significant porting efforts[5].

8.3. Parallel Programming Models

At perhaps the lowest level of parallel programming, the legacy VPIC (Vector Particle-in-Cell) code uses hand-coded SIMD intrinsics to achieve the highest possible performance on modern CPU architectures [188]. Specialised code is required for each new instruction set, and a particular codepath is chosen using pre-processing directives at compile time. The maintainability and portability of such an approach has subsequently led to the development of VPIC 2.0, which instead uses Kokkos for portable performance [186], at the expense of performance on some platforms.

The two pragma-based approaches of OpenMP and OpenACC are perhaps the easiest to implement into an existing application and require only minimal code changes. There are various studies that show that across CPUs, OpenMP can provide good performance [61, 182], and that across GPUs, OpenACC can be competitive with native solutions such as CUDA [154]. However, applications using OpenMP with target offload may require different directives for host and accelerator architectures, in order to achieve the best performance on each [59, 183], potentially leading to code divergence and target-specific compilation. The descriptive *loop* construct (introduced in the OpenMP 5.0 standard [40]) aims to address this concern, instead allowing the compiler to generate different code paths based on architecture within a single binary [196].

[5] See: EPOC++ [194], NekRS [195]

Table 1: A selection of performance studies performed using applications of interest to the Plasma fusion community.

Application	Computational Methods	Programming Models	Performance metrics	References
BookLeaf	Unstructured Arbitrary Lagrangian-Eulerian	Fortran, C++, OpenMP, CUDA, MPI, Kokkos, RAJA	Time, **Ψ**	[150, 151]
CloverLeaf	Finite Volume	Fortran, C++, OpenMP, OpenACC, OpenCL, CUDA, SYCL, Coarray Fortran, MPI, OPS	Time, Energy, **Ψ**	[58, 60, 152–156]
EMPIRE-PIC	Finite-Element, Particle-in-Cell	C++, Kokkos, Trilinos	Time, FLOP/s, GB/s	[157–159]
EPOCH	Structured Particle-in-Cell	Fortran, MPI	Time	[160–162]
GENE	Finite Difference	Fortran, C++, OpenMP, MPI	Time, FLOP/s, GB/s	[163, 164]
Heat	Finite Difference	C, OpenMP, OpenCL, CUDA, HIP, SYCL	Time, GB/s, **Ψ**	[58, 61]
HERMES-3	Finite Volume	C++, Bout++, PETSc, Hypre	Time	[165]
hipBone	High-order Finite Element	C++, OpenMP, OpenCL, CUDA, HIP, SYCL, OCCA	FLOP/s, GB/s	[166]
Laghos	High-order Finite Element	C++, OpenMP, CUDA, OCCA, RAJA, MFEM, METIS, hypre	Time	[167, 168]
MG-CFD	Multi-grid Finite Volume	C/C++, OpenMP, OpenACC, MPI, CUDA, OP2	Time, FLOP/s, GB/s	[59, 169]
miniFE	Finite Element	C++, OpenMP, CUDA, HIP, SYCL, MPI, Kokkos, MKL	Time, Energy, **Ψ**	[61, 156, 170, 171]
nek5000	Spectral Element	Fortran, C, MPI	Time, FLOP/s	[172, 173]
Nekbone	Spectral Element	Fortran, C, OpenMP, OpenACC, MPI	Time, FLOP/s, GB/s,	[174–176]
Nektar++	Spectral Element	C++, MPI, BLAS, LAPACK	Time, Energy	[177]
PIConGPU	Structured Particle-in-Cell	C++, MPI, OpenMP, CUDA, HIP, SYCL, Alpaka	Time, FLOP/s	[70, 178]
PUMIPic	Unstructured Particle-in-Cell	C++, Kokkos	Time	[179]
TeaLeaf	Finite Difference	Fortran, C, OpenMP, OpenACC, CUDA, MPI, Kokkos, RAJA, OPS	Time, Energy, GB/s, **Ψ**	[156, 180–183]
vlp4d	Semi-Lagrangian Scheme	C++, OpenMP, OpenACC, OpenMP4.5, CUDA, HIP, stdpar, Kokkos, Thrust	Time, FLOP/s, **Ψ**	[184, 185]
VPIC	Structured Particle-in-Cell	C++, SIMD Intrinsics, Kokkos	Time, FLOP/s	[186–188]
WarpX	Structured Particle-in-Cell, Adaptive Mesh Refinement	C++, OpenMP, CUDA, HIP, SYCL, MPI, AMReX	Time, FLOP/s	[189–191]
XGC/XGCm	Unstructured Particle-in-Cell	Fortran, C++, OpenMP, CUDA, MPI, Kokkos	Time	[136, 192, 193]

Many mini-apps are also implemented using native GPU parallel programming models such as CUDA and HIP/ROCm. These typically achieve the highest performance on their respective target platforms, but they are subject to significant vendor lock-in [152, 181, 182].

Although significantly more portable, many studies with OpenCL highlight degraded performance on CPUs [58, 197] and productivity challenges (with application kernels potentially requiring twice as much code as the equivalent CUDA kernel [154]).

SYCL addresses these productivity issues, being a single-source solution embedded in C++. The focus of many SYCL performance studies has been on the maturity of the programming model and its implementing compilers [58, 60, 61]. In the general case it is able to offer portability and productivity at the expense of some performance. However, the performance gap has been reducing as the compilers improve.

8.4. Software Libraries

Both EMPIRE-PIC and Laghos make use of numerical algorithm libraries for a significant proportion of their compute; Laghos through MFEM, and EMPIRE-PIC through Trilinos (see Section 5). These libraries in turn use other numerical algorithm libraries through portable interfaces such as BLAS and LAPACK. MFEM targets heterogeneous architectures through RAJA, while Trilinos targets heterogeneous architectures through Kokkos. Since these applications have been designed with portability in mind using a single portable programming model, it is difficult to evaluate their performance portability, since there are no alternative implementations for comparison.

However, their parallel programming models, Kokkos and RAJA, have been evaluated more widely as part of many mini-application portability studies. Studies have shown that both are typically able to deliver good and portable performance from a single-source code base [150, 180, 182, 183]. The Kokkos programming model has also been adopted by a new implementation of VPIC, enabling portability to heterogeneous architectures at the expense of performance on CPU systems [186].

8.5. Domain Specific Languages

In the case of many high-level DSLs, it is again difficult to objectively measure the performance portability without alternative implementations available. Rathgeber et al. evaluate the Firedrake and FEniCS solvers, both of which use UFL. Their study shows that Firedrake, which uses the low-level PyOP2 DSL for performance portability, typically outperforms FEniCS [198].

The lower-level OP DSLs, OPS and OP2, have been more widely examined as part of a number of mini-application studies [155, 169, 182]. Typically they show good performance on a range of heterogeneous hardware, but they are arguably not as productive as higher-level DSLs, where mathematics can be represented more directly. Higher-level DSL thus can provide a productivity advantage if an application is to be developed from scratch. However, when converting an existing application or legacy code to a DSL, the lower-level OPS/OP2 style DSLs are better suited and compatible for a step-by-step (e.g., loop-by-loop) conversion co-existing with the original application [199]. Such a conversion might not be possible with a high-level DSL, where the full application might need to be expressed in the higher-level DSL notation before any validation can begin.

Applications implemented in high-level DSLs, such as Firedrake, often contain multiple levels of DSL with a high-level DSL for scientists and a lower-level DSL providing the portability. These applications fully embody the separation of concerns paradigm, improving productivity for domain experts who can represent their problems directly. If the code generated at each layer is transparent to the user, this helps with debugging and end-to-end validation. Users more often than not, do not like to have black-box or hard to understand intermediate representations. However, changing user requirements that demand components that break the abstraction most likely will be difficult to support with these DSLs. Users asking for “escape hatches” to support future numerics could lead to a considerable impact on performance.

9. Conclusion

Newly developed simulation applications may employ DSLs and software at different levels of the software development stack (see Figure 2). High-level DSLs may allow scientists to express equations directly, while low-level DSLs and parallel programming models will allow applications to target different parallel architectures.

In this paper we have provided an overview of many of the approaches that are available at each level of this software development stack, highlighting many of the advantages and disadvantages in each case. Furthermore, we have outlined some of the key metrics and methods used to evaluate the performance, portability, and productivity of scientific simulation applications.

The focus of this paper has been on approaches and applications from the plasma physics domain, motivated by Project NEPTUNE – a UK effort to develop a new tokamak edge code to treat the complex dynamics of fusion plasma. However, the discussion in this paper is more generally applicable to the development of any new simulation application focused on portability with high parallel performance.

Any modern HPC application is likely to use approaches from at least the three lowest levels of the software stack, typically being implemented in a general purpose programming language, using a parallel programming model, and a distributed memory model.

Targeting portability from a single code base, and productivity for domain specialists often requires further abstractions, at the top levels of our development stack, allowing greater scope for code synthesis. For multi-science/multi-scale applications, a further level of abstraction may be required to couple multiple simulations/applications together.

Acknowledgements

The ExCALIBUR programme (`https://excalibur.ac.uk/`) is supported by the UKRI Strategic Priorities Fund. The programme is co-delivered by the Met Office and EPSRC in partnership with the Public Sector Research Establishment, the UK Atomic Energy Authority (UKAEA) and UKRI research councils, including NERC, MRC and STFC.

References

[1] C. Chang, V. L. Deringer, K. S. Katti, V. Van Speybroeck, C. M. Wolverton, Simulations in the Era of Exascale Computing, Nature Reviews Materials 8 (2023) 309–313.

[2] J. Dongarra, et al., The International Exascale Software Project Roadmap, The International Journal of High Performance Computing Applications 25 (2011) 3–60.

[3] W. Arter, L. Anton, D. Samaddar, R. Akers, ExCALIBUR Fusion Modelling System Science Plan, Technical Report CD/EXCALIBUR-FMS/0001, UKAEA, 2019. https://www.metoffice.gov.uk/binaries/content/assets/metofficegovuk/pdf/research/spf/ukaea-excalibur-fms-scienceplan.pdf.

[4] A. Bhattacharjee, ECP WDMApp Team, High-fidelity whole device model of magnetically confined fusion plasma, in: APS Division of Plasma Physics Meeting Abstracts, volume 2019, 2019, pp. NM9–002.

[5] S. Markidis, J. J. Williams, T. Dannert, V. Papaefstathiou, U. Ganse, L. Kos, I. B. Peng, D. Tskhakaya, Plasma-PEPSC - Plasma Exascale-Performance Simulations Centre of Excellence, Horizon Europe Grant Reference: 101093261, 2023.

[6] Eurofusion, European Research Roadmap to the Realisation of Fusion Energy, Technical Report SOFT 2018 Version, Eurofusion, 2018. https://euro-fusion.org/wp-content/uploads/2022/10/2018_Research_roadmap_long_version_01.pdf.

[7] G. Federici, W. Biel, M. R. Gilbert, R. Kemp, N. Taylor, R. Wenninger, European DEMO design strategy and consequences for materials, Nuclear Fusion 57 (2017) 092002.

[8] I. Z. Reguly, G. R. Mudalige, Productivity, performance, and portability for computational fluid dynamics applications, Computers & Fluids 199 (2020) 1–10.

[9] J. P. Singh, J. L. Hennessy, An empirical investigation of the effectiveness and limitations of automatic parallelization, Shared memory multiprocessing (1992) 203–207.

[10] R. Harel, Y. Pinter, G. Oren, Learning to Parallelize in a Shared-Memory Environment with Transformers, in: Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, PPoPP '23, AMC, New York, NY, USA, 2023, p. 450–452. doi:10.1145/3572848.3582565.

[11] S. Prema, R. Jehadeesan, B. K. Panigrahi, Identifying pitfalls in automatic parallelization of NAS parallel benchmarks, in: 2017 National Conference on Parallel Computing Technologies (PARCOMPTECH), 2017, pp. 1–6. doi:10.1109/PARCOMPTECH.2017.8068329.

[12] K. Kennedy, C. Koelbel, H. Zima, The Rise and Fall of High Performance Fortran: An Historical Object Lesson, in: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, HOPL III, AMC, New York, NY, USA, 2007, pp. 7–1–7–22. doi:10.1145/1238844.1238851.

[13] E. Belikov, P. Deligiannis, P. Totoo, M. Aljabri, H.-W. Loidl, A survey of high-level parallel programming models, Technical Report HW-MACS-TR-0103, Heriot-Watt University, Edinburgh, UK, 2013.

[14] H. Kasim, V. March, R. Zhang, S. See, Survey on Parallel Programming Model, in: J. Cao, M. Li, M.-Y. Wu, J. Chen (Eds.), Network and Parallel Computing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 266–275.

[15] J. Diaz, C. Muñoz Caro, A. Niño, A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era, IEEE Transactions on Parallel and Distributed Systems 23 (2012) 1369–1386.

[16] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, A. R. LeBlanc, Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions, IEEE Journal of Solid-State Circuits 9 (1974) 256–268.

[17] J. L. Hennessy, D. A. Patterson, A New Golden Age for Computer Architecture, Commun. ACM 62 (2019) 48–60.

[18] C. E. Leiserson, N. C. Thompson, J. S. Emer, B. C. Kuszmaul, B. W. Lampson, D. Sanchez, T. B. Schardl, There's plenty of room at the Top: What will drive computer performance after Moore's law?, Science 368 (2020) eaam9744.

[19] D. Patterson, The trouble with multi-core, IEEE Spectrum 47 (2010) 28–32, 53.

[20] A. Turner, Parallel Software usage on UK National HPC Facilities 2009-2015: How well have applications kept up with increasingly parallel hardware?, Technical Report, Edinburgh Parallel Computing Centre, 2015.

[21] T. M. Evans, A. Siegel, E. W. Draeger, J. Deslippe, M. M. Francois, T. C. Germann, W. E. Hart, D. F. Martin, A survey of software implementations used by application codes in the Exascale Computing Project, The International Journal of High Performance Computing Applications 36 (2022) 5–12.

[22] G. M. Shipman, T. C. Randles, An evaluation of risks associated with relying on Fortran for mission critical codes for the next 15 years, Technical Report LA-UR-23-23992, Los Alamos National Laboratory, 2023. doi:10.2172/1970284.

[23] J. Marks, E. Medwedeff, O. Čertík, R. Bird, R. W. Robey, Improving fortran performance portability, in: B. Chapman, J. Moreira (Eds.), Languages and Compilers for Parallel Computing, Springer International Publishing, Cham, 2022, pp. 74–83.

[24] A. Hsu, D. N. Asanza, J. A. Schoonover, Z. Jibben, N. N. Carlson, R. Robey, Performance Portability Challenges for Fortran Applications, in: 2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2018, pp. 47–58. doi:10.1109/P3HPC.2018.00008.

[25] L. J. Kedward, B. Aradi, O. Čertík, M. Curcic, S. Ehlert, P. Engel, R. Goswami, M. Hirsch, A. Lozada-Blanco, V. Magnin, A. Markus, E. Pagone, I. Pribec, B. Richardson, H. Snyder, J. Urban, J. Vandenplas, The State of Fortran, Computing in Science & Engineering 24 (2022) 63–72.

[26] S. K. Lam, A. Pitrou, S. Seibert, Numba: A LLVM-Based Python JIT Compiler, in: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM '15, Association for Computing Machinery, New York, NY, USA, 2015, pp. 1–6. doi:10.1145/2833157.2833162.

[27] W. T. Lavrijsen, A. Dutta, High-Performance Python-C++ Bindings with PyPy and Cling, in: 2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC), 2016, pp. 27–35. doi:10.1109/PyHPC.2016.008.

[28] D. Beazley, Understanding the Python GIL, in: PyCON Python Conference. Atlanta, Georgia, 2010, pp. 1–62.

[29] B. L. Chamberlain, D. Callahan, H. P. Zima, Parallel Programmability and the Chapel Language, The International Journal of High Performance Computing Applications 21 (2007) 291–312.

[30] G. L. Steele, Parallel Programming and Code Selection in Fortress, in: Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '06, Association for Computing Machinery, New York, NY, USA, 2006, p. 1. doi:10.1145/1122971.1122972.

[31] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, V. Sarkar, X10: An Object-Oriented Approach to Non-Uniform Cluster Computing, SIGPLAN Not. 40 (2005) 519–538.

[32] J. Bezanson, A. Edelman, S. Karpinski, V. B. Shah, Julia: A Fresh Approach to Numerical Computing, SIAM Review 59 (2017) 65–98.

[33] C. Rackauckas, Q. Nie, Differentialequations.jl–a performant and feature-rich ecosystem for solving differential equations in julia, Journal of Open Research Software 5 (2017).

[34] S. Maleki, Y. Gao, M. J. Garzar´n, T. Wong, D. A. Padua, An Evaluation of Vectorizing Compilers, in: 2011 International Conference on Parallel Architectures and Compilation Techniques, 2011, pp. 372–382. doi:10.1109/PACT.2011.68.

[35] H. Amiri, A. Shahbahrami, SIMD programming using Intel vector extensions, Journal of Parallel and Distributed Computing 135 (2020) 83–100.

[36] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, S. A. Jarvis, Exploring SIMD for Molecular Dynamics, Using Intel Xeon Processors and Intel Xeon Phi Coprocessors, in: Parallel and Distributed Processing Symposium, International, IEEE Computer Society, Los Alamitos, CA, USA, 2013, pp. 1085–1097. doi:10.1109/IPDPS.2013.44.

[37] L. Dagum, R. Menon, OpenMP: An Industry Standard API for Shared-Memory Programming, IEEE Computational Science & Engineering 5 (1998) 46–55.

[38] OpenMP Architecture Review Board, OpenMP API Version 4.0, 2013. URL: https://openmp.org/wp-content/uploads/OpenMP4.0.0.pdf.

[39] OpenMP Architecture Review Board, OpenMP API Version 4.5, 2015. URL: https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[40] OpenMP Architecture Review Board, OpenMP API Version 5.0, 2015. URL: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

[41] Message Passing Interface Forum, MPI: A Message Passing Interface Standard Version 2.2, High Performance Computing Applications 12 (2009) 1–647.

[42] A. Castelló, R. M. Gual, S. Seo, P. Balaji, E. S. Quintana-Ortí, A. J. Peña, Analysis of Threading Libraries for High Performance Computing, IEEE Transactions on Computers 69 (2020) 1279–1292.

[43] R. W. Numrich, J. Reid, Co-Array Fortran for Parallel Programming, SIGPLAN Fortran Forum 17 (1998) 1–31.

[44] T. El-Ghazawi, L. Smith, UPC: Unified Parallel C, in: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC '06, Association for Computing Machinery, New York, NY, USA, 2006, p. 27–es. doi:10.1145/1188455.1188483.

[45] L. V. Kale, S. Krishnan, CHARM++: A Portable Concurrent Object Oriented System Based on C++, SIGPLAN Not. 28 (1993) 91–108.

[46] Bauer, Michael and Treichler, Sean and Slaughter, Elliott and Aiken, Alex, Legion: Expressing locality and independence with logical regions, in: SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2012, pp. 1–11. doi:10.1109/SC.2012.71.

[47] H. Kaiser, P. Diehl, A. S. Lemoine, B. A. Lelbach, P. Amini, A. Berge, J. Biddiscombe, S. R. Brandt, N. Gupta, T. Heller, K. Huck, Z. Khatami, A. Kheirkhahan, A. Reverdell, S. Shirzad, M. Simberg, B. Wagle, W. Wei, T. Zhang, HPX - The C++ Standard Library for Parallelism and Concurrency, Journal of Open Source Software 5 (2020) 2352.

[48] J. Lifflander, P. Miller, N. L. Slattengren, N. Morales, P. Stickney, P. P. Pébaÿ, Design and Implementation Techniques for an MPI-Oriented AMT Runtime, in: 2020 Workshop on Exascale MPI (ExaMPI), 2020, pp. 31–40. doi:10.1109/ExaMPI52011.2020.00009.

[49] Z. Jin, J. Vetter, Evaluating CUDA Portability with HIPCL and DPCT, in: 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2021, pp. 371–376. doi:10.1109/IPDPSW52791.2021.00065.

[50] J. Fang, A. L. Varbanescu, H. Sips, A Comprehensive Performance Comparison of CUDA and OpenCL, in: 2011 International Conference on Parallel Processing, 2011, pp. 216–225. doi:10.1109/ICPP.2011.45.

[51] M. Fatica, CUDA toolkit and libraries, in: 2008 IEEE Hot Chips 20 Symposium (HCS), 2008, pp. 1–22. doi:10.1109/HOTCHIPS.2008.7476520.

[52] C. Brown, A. Abdelfattah, S. Tomov, J. Dongarra, Design, Optimization, and Benchmarking of Dense Linear Algebra Algorithms on AMD GPUs, in: 2020 IEEE High Performance Extreme Computing Conference (HPEC), 2020, pp. 1–7. doi:10.1109/HPEC43674.2020.9286214.

[53] J. E. Stone, D. Gohara, G. Shi, OpenCL: A parallel programming standard for heterogeneous computing systems, Computing in Science & Engineering 12 (2010) 66.

[54] OpenACC-Standard.org, The OpenACC Application Program Interface Version 3.3, 2022. URL: https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.3-final.pdf.

[55] J. Kelling, S. Bastrakov, A. Debus, T. Kluge, M. Leinhauser, R. Pausch, K. Steiniger, J. Stephan, R. Widera, J. Young, M. Bussmann, S. Chandrasekaran, G. Juckeland, Challenges Porting a C++ Template-Metaprogramming Abstraction Layer to Directive-Based Offloading, in: S. Bhalachandra, C. Daley, V. Melesse Vergara (Eds.), Accelerator Programming Using Directives, Springer International Publishing, Cham, 2022, pp. 92–111.

[56] D. Truby, C. Bertolli, S. A. Wright, G.-T. Bercea, K. O'Brien, S. A. Jarvis, Pointers Inside Lambda Closure Objects in OpenMP Target Offload Regions, in: 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC), 2018, pp. 10–17. doi:10.1109/LLVM-HPC.2018.8639410.

[57] The Khronos SYCL Working Group, SYCL 2020 Specification, 2023. URL: https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf.

[58] T. Deakin, S. McIntosh-Smith, Evaluating the Performance of HPC-Style SYCL Applications, in: Proceedings of the International Workshop on OpenCL, IWOCL '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 1–11. doi:10.1145/3388333.3388643.

[59] I. Z. Reguly, A. M. B. Owenson, A. Powell, S. A. Jarvis, G. R. Mudalige, Under the Hood of SYCL – An Initial Performance Analysis With an Unstructured-mesh CFD Application, in: B. L. Chamberlain, A.-L. Varbanescu, H. Ltaief, P. Luszczek (Eds.), Proceedings of the International Supercomputing Conference (ISC 2021), Springer International Publishing, 2021, pp. 391–410.

[60] W.-C. Lin, T. Deakin, S. McIntosh-Smith, On Measuring the Maturity of SYCL Implementations by Tracking Historical Performance Improvements, in: International Workshop on OpenCL, IWOCL'21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 1–13. doi:10.1145/3456669.3456701.

[61] W. Shilpage, S. A. Wright, An Investigation into the Performance and Portability of SYCL Compiler Implementations, Lecture Notes in Computer Science (LNCS) 13999 (2023).

[62] D. S. Medina, A. St-Cyr, T. Warburton, OCCA: A unified approach to multi-threading languages, 2014. doi:10.48550/ARXIV.1403.0968.

[63] H. C. Edwards, C. R. Trott, D. Sunderland, Kokkos: Enabling manycore performance portability through polymorphic memory access patterns, Journal of Parallel and Distributed Computing 74 (2014) 3202–3216.

[64] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, T. R. Scogland, RAJA: Portable Performance for Large-Scale Scientific Applications, in: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2019, pp. 71–81. doi:10.1109/P3HPC49587.2019.00012.

[65] E. Zenker, B. Worpitz, R. Widera, A. Huebl, G. Juckeland, A. Knüpfer, W. E. Nagel, M. Bussmann, Alpaka - An Abstraction Library for Parallel Kernel Acceleration, in: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE Computer Society, 2016, pp. 631–640. doi:10.1109/IPDPSW.2016.50.

[66] P. Plauger, M. Lee, D. Musser, A. A. Stepanov, C++ Standard Template Library, 1st ed., Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[67] B. Schling, The Boost C++ Libraries, XML Press, 2011.

[68] G. Guennebaud, B. Jacob, et al., Eigen v3, http://eigen.tuxfamily.org, 2010.

[69] N. Bell, J. Hoberock, Chapter 26 - Thrust: A Productivity-Oriented Library for CUDA, in: W. mei W. Hwu (Ed.), GPU Computing Gems Jade Edition, Applications of GPU Computing Series, Morgan Kaufmann, Boston, 2012, pp. 359–371. doi:10.1016/B978-0-12-385963-1.00026-5.

[70] H. Burau, R. Widera, W. Hönig, G. Juckeland, A. Debus, T. Kluge, U. Schramm, T. E. Cowan, R. Sauerbrey, M. Bussmann, PIConGPU: A Fully Relativistic Particle-in-Cell Code for a GPU Cluster, IEEE Transactions on Plasma Science 38 (2010) 2831–2839.

[71] Exascale Computing Project, ECP Proxy Applications, https://proxyapps.exascaleproject.org/ (accessed April 20, 2021), 2021.

[72] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al., An updated set of basic linear algebra subprograms (BLAS), ACM Transactions on Mathematical Software 28 (2002) 135–151.

[73] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, D. Sorensen, LAPACK: A portable linear algebra library for high-performance computers, in: Supercomputing '90:Proceedings of the 1990 ACM/IEEE Conference on Supercomputing, 1990, pp. 2–11. doi:10.1109/SUPERC.1990.129995.

[74] M. Frigo, S. Johnson, FFTW: an adaptive software architecture for the FFT, in: Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181), volume 3, 1998, pp. 1381–1384 vol.3. doi:10.1109/ICASSP.1998.681704.

[75] S. Tomov, J. Dongarra, M. Baboulin, Towards dense linear algebra for hybrid GPU accelerated manycore systems, Parallel Computing 36 (2010) 232–240.

[76] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. Gropp, D. Karpeyev, D. Kaushik, M. Knepley, D. May, L. McInnes, R. Mills, T. Munson, K. Rupp, P. Sanan, B. Smith, S. Zampini, H. Zhang, H. Zhang, PETSc Users Manual, 2019.

[77] J. Zhang, et al., The PetscSF Scalable Communication Layer, arXiv (2021) 2102.13018.

[78] R. D. Falgout, J. E. Jones, U. M. Yang, The design and implementation of hypre, a library of parallel high performance preconditioners, in: Numerical solution of partial differential equations on parallel computers, Springer, 2006, pp. 267–294.

[79] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, K. S. Stanley, An Overview of the Trilinos Project, ACM Trans. Math. Softw. 31 (2005) 397–423.

[80] G. Karypis, V. Kumar, METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Technical Report TR 97-061, University of Minnesota, 1997.

[81] G. Karypis, K. Schloegel, V. Kumar, PARMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library, Technical Report TR 97-060, University of Minnesota, 1997.

[82] F. Pellegrini, J. Roman, Sparse matrix ordering with Scotch, in: B. Hertzberger, P. Sloot (Eds.), High-Performance Computing and Networking, Springer Berlin Heidelberg, Berlin, Heidelberg, 1997, pp. 370–378.

[83] C. Chevalier, F. Pellegrini, PT-Scotch: A tool for efficient parallel graph ordering, Parallel Computing 34 (2008) 318–331. Parallel Matrix Algorithms and Applications.

[84] P. Sanders, C. Schulz, Think Locally, Act Globally: Highly Balanced Graph Partitioning, in: Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings, volume 7933, Springer, 2013, pp. 164–175.

[85] E. G. Boman, K. D. Devine, V. J. Leung, S. Rajamanickam, L. A. Riesen, M. Deveci, U. Catalyurek, Zoltan2: Next-Generation Combinatorial Toolkit, in: Trilinos Users Group Meeting, 2012.

[86] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, S. Amarasinghe, Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines, in: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, ACM, New York, NY, USA, 2013, pp. 519–530. doi:10.1145/2491956.2462176.

[87] B. Mostafazadeh, F. Marti, F. Liu, A. Chandramowlishwaran, Roofline Guided Design and Analysis of a Multi-stencil CFD Solver for Multicore Performance, in: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2018, pp. 753–762.

[88] C. Yount, J. Tobin, A. Breuer, A. Duran, YASK—Yet Another Stencil Kernel: A Framework for HPC Stencil Code-Generation and Tuning, in: 2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), 2016, pp. 30–39.

[89] I. Z. Reguly, G. R. Mudalige, M. B. Giles, D. Curran, S. McIntosh-Smith, The OPS Domain Specific Abstraction for Multi-block Structured Grid Computations, in: Proceedings of the 2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, WOLFHPC '14, IEEE Computer Society, Washington, DC, USA, 2014, pp. 58–67.

[90] S. Kuckuk, G. Haase, D. A. Vasco, H. Köstler, Towards generating efficient flow solvers with the ExaStencils approach, Concurrency and Computation: Practice and Experience 29 (2017) e4062.

[91] C. Lengauer, S. Apel, M. Bolten, A. Größlinger, F. Hannig, H. Köstler, U. Rüde, J. Teich, A. Grebhahn, S. Kronawitter, S. Kuckuk, H. Rittich, C. Schmitt, ExaStencils: Advanced Stencil-Code Engineering, in: L. Lopes, J. Žilinskas, A. Costan, R. G. Cascella, G. Kecskemeti, E. Jeannot, M. Cannataro, L. Ricci,

S. Benkner, S. Petit, V. Scarano, J. Gracia, S. Hunold, S. L. Scott, S. Lankes, C. Lengauer, J. Carretero, J. Breitbart, M. Alexander (Eds.), Euro-Par 2014: Parallel Processing Workshops, Springer International Publishing, Cham, 2014, pp. 553–564.

[92] T. Zhao, S. Williams, M. Hall, H. Johansen, Delivering Performance-Portable Stencil Computations on CPUs and GPUs Using Bricks, in: 2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2018, pp. 59–70.

[93] G. R. Mudalige, M. B. Giles, I. Reguly, C. Bertolli, P. H. J. Kelly, OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures, in: 2012 Innovative Parallel Computing (InPar), 2012, pp. 1–12.

[94] F. Rathgeber, G. R. Markall, L. Mitchell, N. Loriant, D. A. Ham, C. Bertolli, P. H. J. Kelly, PyOP2: A high-level framework for performance-portable simulations on unstructured meshes, in: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, IEEE, 2012, pp. 1116–1123.

[95] O. Fuhrer, C. Osuna, X. Lapillonne, T. Gysi, B. Cumming, M. Bianco, A. Arteaga, T. Schulthess, Towards a performance portable, architecture agnostic implementation strategy for weather and climate models, Supercomputing Frontiers and Innovations 1 (2014).

[96] S. Adams, R. Ford, M. Hambley, J. Hobson, I. Kavčič, C. Maynard, T. Melvin, E. Müller, S. Mullerworth, A. Porter, M. Rezny, B. Shipway, R. Wong, LFRic: Meeting the challenges of scalability and performance portability in Weather and Climate models, Journal of Parallel and Distributed Computing 132 (2019) 383–396.

[97] M. Baldauf, A. Seifert, J. Förstner, D. Majewski, M. Raschendorfer, T. Reinhardt, Operational convective-scale numerical weather prediction with the COSMO model: description and sensitivities, Monthly Weather Review 139 (2011) 3887–3905.

[98] S. Siso, A. R. Porter, R. W. Ford, Transforming Fortran Weather and Climate Applications to OpenCL Using PSyclone, in: Proceedings of the 2023 International Workshop on OpenCL, IWOCL '23, Association for Computing Machinery, New York, NY, USA, 2023, pp. 1–8. doi:10.1145/3585341.3585360.

[99] S. Siso, A. R. Porter, R. W. Ford, Transforming Fortran weather and climate applications to OpenCL using PSyclone, in: 2023 International Workshop on OpenCL (IWOCL '23), 2023, pp. 1–8. doi:10.1145/3585341.3585360.

[100] V. Clément, S. Ferrachat, O. Fuhrer, X. Lapillonne, C. E. Osuna, R. Pincus, J. Rood, W. Sawyer, The CLAW DSL: Abstractions for Performance Portable Weather and Climate Models, in: Proceedings of the Platform for Advanced Scientific Computing Conference, PASC '18, ACM, New York, NY, USA, 2018, pp. 2:1–2:10. doi:10.1145/3218176.3218226.

[101] V. Clément, P. Marti, O. Fuhrer, W. Sawyer, Performance portability on GPU and CPU with the ICON global climate model, in: EGU General Assembly Conference Abstracts, volume 20 of *EGU General Assembly Conference Abstracts*, 2018, p. 13435.

[102] J. M. Dawson, Particle simulation of plasmas, Rev. Mod. Phys. 55 (1983) 403–447.

[103] O. Awile, M. Mitrovic, S. Reboux, I. F. Sbalzarini, A domain-specific programming language for particle simulations on distributed-memory parallel computers, CIMNE, 2013, pp. 436–447.

[104] S. Karol, T. Nett, J. Castrillon, I. F. Sbalzarini, A Domain-Specific Language and Editor for Parallel Particle Methods, ACM Trans. Math. Softw. 44 (2018).

[105] P. Incardona, A. Leo, Y. Zaluzhnyi, R. Ramaswamy, I. F. Sbalzarini, OpenFPM: A scalable open framework for particle and particle-mesh codes on parallel computers, Computer Physics Communications 241 (2019) 155–177.

[106] J. A. Anderson, J. Glaser, S. C. Glotzer, HOOMD-blue: A Python package for high-performance molecular dynamics and hard particle Monte Carlo simulations, Computational Materials Science 173 (2020) 109363.

[107] T. Cickovski, C. Sweet, J. A. Izaguirre, MDL, A Domain-Specific Language for Molecular Dynamics, in: 40th Annual Simulation Symposium (ANSS'07), 2007, pp. 256–266. doi:`10.1109/ANSS.2007.26`.

[108] W. R. Saunders, J. Grant, E. H. Müller, A domain specific language for performance portable molecular dynamics algorithms, Computer Physics Communications 224 (2018) 119–135.

[109] S. Slattery, S. T. Reeve, C. Junghans, D. Lebrun-Grandié, R. Bird, G. Chen, S. Fogerty, Y. Qiu, S. Schulz, A. Scheinberg, A. Isner, K. Chong, S. Moore, T. Germann, J. Belak, S. Mniszewski, Cabana: A Performance Portable Library for Particle-Based Simulations, Journal of Open Source Software 7 (2022) 4115.

[110] M. S. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, G. N. Wells, The FEniCS Project Version 1.5, Archive of Numerical Software 3 (2015).

[111] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, P. H. J. Kelly, Firedrake: Automating the Finite Element Method by Composing Abstractions, ACM Trans. Math. Softw. 43 (2016) 24:1–24:27.

[112] D. J. Lusher, S. P. Jammy, N. D. Sandham, Shock-wave/boundary-layer interactions in the automatic source-code generation framework OpenSBLI, Computers & Fluids 173 (2018) 17–21.

[113] M. Lange, N. Kukreja, M. Louboutin, F. Luporini, F. Vieira, V. Pandolfo, P. Velesko, P. Kazakas, G. Gorman, Devito: Towards a Generic Finite Difference DSL Using Symbolic Python, in: 2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC), 2016, pp. 67–75.

[114] B. D. Dudson, M. V. Umansky, X. Q. Xu, P. B. Snyder, H. R. Wilson, BOUT++: A framework for parallel plasma fluid simulations, Computer Physics Communications 180 (2009) 1467–1480.

[115] B. D. Dudson, P. A. Hill, D. Dickinson, J. Parker, A. Dempsey, A. Allen, A. Bokshi, B. Shanahan, B. Friedman, C. Ma, D. Schwörer, D. Meyerson, E. Grinaker, G. Breyiannia, H. Muhammed, H. Seto, H. Zhang, I. Joseph, J. Leddy, J. Brown, J. Madsen, J. Omotani, J. Sauppe, K. Savage, L. Wang, L. Easy, M. Estarellas, M. Thomas, M. Umansky, M. Løiten, M. Kim, M. Leconte, N. Walkden, O. Izacard, P. Xi, P. Naylor, F. Riva, S. Tiwari, S. Farley, S. Myers, T. Xia, T. Rhee, X. Liu, X. Xu, Z. Wang, BOUT++, 2020. doi:`10.5281/zenodo.4046792`.

[116] T. L. Veldhuizen, Arrays in Blitz++, in: D. Caromel, R. R. Oldehoeft, M. Tholburn (Eds.), Computing in Object-Oriented Parallel Environments, Springer Berlin Heidelberg, Berlin, Heidelberg, 1998, pp. 223–230.

[117] D. Beckingsale, M. Mcfadden, J. Dahm, R. Pankajakshan, R. Hornung, Umpire: Application-Focused Management and Coordination of Complex Hierarchical Memory, IBM Journal of Research and Development (2019).

[118] S. Wiesen, D. Reiter, V. Kotov, M. Baelmans, W. Dekeyser, A. Kukushkin, S. Lisgo, R. Pitts, V. Rozhansky, G. Saibene, et al., The new SOLPS-ITER code package, Journal of Nuclear Materials 463 (2015) 480–484.

[119] Y.-H. Tang, S. Kudo, X. Bian, Z. Li, G. E. Karniadakis, Multiscale universal interface: a concurrent framework for coupling heterogeneous solvers, Journal of Computational Physics 297 (2015) 13–31.

[120] J. Y. Choi, C.-S. Chang, J. Dominski, S. Klasky, G. Merlo, E. Suchyta, M. Ainsworth, B. Allen, F. Cappello, M. Churchill, et al., Coupling exascale multiphysics applications: Methods and lessons learned, in: 2018 IEEE 14th International Conference on e-Science (e-Science), IEEE, 2018, pp. 442–452.

[121] B. Gatzhammer, Efficient and flexible partitioned simulation of fluid-structure interactions, Ph.D. thesis, Technische Universität München, 2014.

[122] M. A. Moratilla-Vega, M. Angelino, H. Xia, G. J. Page, An open-source coupled method for aeroacoustics modelling, Computer Physics Communications 278 (2022) 108420.

[123] C. J. Permann, D. R. Gaston, D. Andrš, R. W. Carlsen, F. Kong, A. D. Lindsay, J. M. Miller, J. W. Peterson, A. E. Slaughter, R. H. Stogner, et al., MOOSE: Enabling massively parallel multiphysics simulation, SoftwareX 11 (2020) 100430.

[124] H. Brooks, A. Davis, Scalable multi-physics for fusion reactors with AURORA, Plasma Physics and Controlled Fusion 65 (2022) 024002.

[125] A. Powell, K. Choudry, A. Prabhakar, I. Reguly, D. Amirante, S. Jarvis, G. Mudalige, Predictive Analysis of Large-Scale Coupled CFD Simulations with the CPX Mini-App, in: 2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC), 2021, pp. 141–151. doi:10.1109/HiPC53243.2021.00028.

[126] G. Chourdakis, K. Davis, B. Rodenberg, M. Schulte, F. Simonis, B. Uekermann, G. Abrams, H.-J. Bungartz, L. C. Yau, I. Desai, et al., preCICE v2: A sustainable and user-friendly coupling library, arXiv preprint arXiv:2109.14470 (2021).

[127] M. Camps Santasmasas, Hybrid GPU/CPU Navier-Stokes lattice Boltzmann method for urban wind flow, Ph.D. thesis, The University of Manchester, School of Mechanical, Aerospace and Civil Engineering, 2021.

[128] P. Rubin, Comparison of code coupling libraries for high performance multiphysics simulation, Technical Report DL-TR-2022-001, STFC, 2022.

[129] B. W. Uekermann, Partitioned fluid-structure interaction on massively parallel systems, Ph.D. thesis, Technische Universität München, 2016.

[130] E. Quemerais, B. Frisuli, A. Meyniel, S. Beyou, The CWIPI coupling library, https://w3.onera.fr/cwipi/bibliotheque-couplage-cwipi, 2022.

[131] H. Al Daas, N. Bootland, T. Rees, A. S. P. Rubin, S. Thorne, J. Williams, Techniques and software relevant to the coupling of continuum (fluid) and particle models of plasma for NEPTUNE, Technical Report 2068625-TN-05, ExCALIBUR-NEPTUNE Programme, 2023. URL: https://excalibur-neptune.github.io/Documents/.

[132] S. Thorne, J. Williams, Implementation and scalability analysis of coupling continuum (fluid) and particle models of plasma for NEPTUNE, Technical Report 2068625-TN-06, ExCALIBUR-NEPTUNE Programme, 2023. URL: https://excalibur-neptune.github.io/Documents/.

[133] S. Longshaw, R. Pillai, L. Gibelli, D. Emerson, D. Lockerby, Coupling Molecular Dynamics and Direct Simulation Monte Carlo using a general and high-performance code coupling library, Computers & Fluids 213 (2020) 104726.

[134] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, C. Jin, Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS), in: Proceedings of the 6th international workshop on Challenges of large applications in distributed environments, 2008, pp. 15–24.

[135] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck, et al., ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management, SoftwareX 12 (2020) 100561.

[136] S. Ku, R. Hager, C.-S. Chang, J. Kwon, S. E. Parker, A new hybrid-Lagrangian numerical scheme for gyrokinetic simulation of tokamak edge plasma, Journal of Computational Physics 315 (2016) 467–475.

[137] C. Chang, S. Klasky, J. Cummings, R. Samtaney, A. Shoshani, L. Sugiyama, D. Keyes, S. Ku, G. Park, S. Parker, et al., Toward a first-principles integrated simulation of tokamak edge plasmas, Journal of Physics: Conference Series 125 (2008) 012042.

[138] J. Dominski, J. Cheng, G. Merlo, V. Carey, R. Hager, L. Ricketson, J. Choi, S. Ethier, K. Germaschewski, S. Ku, et al., Spatial coupling of gyrokinetic simulations, a generalized scheme based on first-principles, Physics of Plasmas 28 (2021).

[139] E. Suchyta, S. Klasky, N. Podhorszki, M. Wolf, A. Adesoji, C. Chang, J. Choi, P. E. Davis, J. Dominski, S. Ethier, et al., The exascale framework for high fidelity coupled simulations (effis): Enabling whole device modeling in fusion science, The International Journal of High Performance Computing Applications 36 (2022) 106–128.

[140] Y. Ju, A. Perez, S. Markidis, P. Schlatter, E. Laure, Understanding the impact of synchronous, asynchronous, and hybrid in-situ techniques in computational fluid dynamics applications, in: 2022 IEEE 18th International Conference on e-Science (e-Science), IEEE, 2022, pp. 295–305.

[141] S. Wienke, J. Miller, M. Schulz, 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.

[142] S. Williams, A. Waterman, D. Patterson, Roofline: An Insightful Visual Performance Model for Multicore Architectures, Commun. ACM 52 (2009) 65–76.

[143] Y. J. Lo, S. Williams, B. Van Straalen, T. J. Ligocki, M. J. Cordery, N. J. Wright, M. W. Hall, L. Oliker, Roofline Model Toolkit: A Practical Tool for Architectural and Program Analysis, in: S. A. Jarvis, S. A. Wright, S. D. Hammond (Eds.), High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation, Springer International Publishing, 2015, pp. 129–148.

[144] S. Pennycook, J. Sewall, V. Lee, Implications of a metric for performance portability, Future Generation Computer Systems 92 (2019) 947–958.

[145] S. L. Harrell, J. Kitson, R. Bird, S. J. Pennycook, J. Sewall, D. Jacobsen, D. N. Asanza, A. Hsu, H. C. Carrillo, H. Kim, 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.

[146] J. Sewall, S. J. Pennycook, D. Jacobsen, Code Base Investigator, 10.5281/zenodo.5019024, 2022. doi:10.5281/zenodo.5019024.

[147] S. J. Pennycook, J. Sewall, D. Jacobsen, T. Deakin, Y. Zamora, K. L. K. Lee, Performance, Portability and Productivity Analysis Library, 10.5281/zenodo.7733678, 2023. doi:10.5281/zenodo.7733678.

[148] S. J. Pennycook, J. D. Sewall, D. W. Jacobsen, T. Deakin, S. McIntosh-Smith, Navigating Performance, Portability, and Productivity, Computing in Science & Engineering 23 (2021) 28–38.

[149] J. Sewall, S. J. Pennycook, D. Jacobsen, T. Deakin, 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.

[150] T. R. Law, R. Kevis, S. Powell, J. Dickson, S. Maheswaran, J. A. Herdman, S. A. Jarvis, Performance Portability of an Unstructured Hydrodynamics Mini-application, in: 2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2018, pp. 0–12.

[151] D. Truby, S. A. Wright, R. Kevis, S. Maheswaran, J. A. Herdman, S. A. Jarvis, BookLeaf: An Unstructured Hydrodynamics Mini-Application, in: 2018 IEEE International Conference on Cluster Computing (CLUSTER), 2018, pp. 615–622. doi:10.1109/CLUSTER.2018.00078.

[152] A. Mallinson, D. Beckingsale, W. Gaudin, A. Herdman, J. Levesque, S. Jarvis, CloverLeaf: Preparing Hydrodynamics Codes for Exascale, in: Proceedings of the Cray User Group (CUG), 2013, pp. 1–15.

[153] T. Deakin, A. Poenaru, T. Lin, 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.

[154] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, S. A. Jarvis, Accelerating Hydrocodes with OpenACC, OpenCL and CUDA, in: Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, SCC '12, IEEE Computer Society, USA, 2012, p. 465–471. doi:10.1109/SC.Companion.2012.66.

[155] I. Z. Reguly, G. R. Mudalige, M. B. Giles, Design and Development of Domain Specific Active Libraries with Proxy Applications, in: 2015 IEEE International Conference on Cluster Computing, 2015, pp. 738–745. doi:10.1109/CLUSTER.2015.128.

[156] S. I. Roberts, S. A. Wright, S. A. Fahmy, S. A. Jarvis, The Power-Optimised Software Envelope, ACM Transactions on Architecture and Code Optimization 16 (2019).

[157] M. T. Bettencourt, D. A. S. Brown, K. L. Cartwright, E. C. Cyr, C. A. Glusa, P. T. Lin, S. G. Moore, D. A. O. McGregor, R. P. Pawlowski, E. G. Phillips, N. V.

Roberts, S. A. Wright, S. Maheswaran, J. P. Jones, S. A. Jarvis, EMPIRE-PIC: A Performance Portable Unstructured Particle-in-Cell Code, Communications in Computational Physics 30 (2021) 1–37.

[158] D. A. S. Brown, M. T. Bettencourt, S. A. Wright, S. Maheswaran, J. P. Jones, S. A. Jarvis, Higher-order particle representation for particle-in-cell simulations, Journal of Computational Physics 435 (2021) 110255.

[159] D. A. S. Brown, S. A. Wright, S. A. Jarvis, Performance of a Second Order Electrostatic Particle-in-Cell Algorithm on Modern Many-Core Architectures, Electronic Notes in Theoretical Computer Science 340 (2018) 67–84. The proceedings of UKPEW 2017, the thirty third Annual UK Performance Engineering Workshops (UKPEW).

[160] J. R. Smith, C. Orban, N. Rahman, B. McHugh, R. Oropeza, E. A. Chowdhury, A particle-in-cell code comparison for ion acceleration: EPOCH, LSP, and WarpX, Physics of Plasmas 28 (2021) 074505.

[161] R. F. Bird, P. Gillies, M. R. Bareford, A. Herdman, S. Jarvis, Performance Optimisation of Inertial Confinement Fusion Codes using Mini-applications, The International Journal of High Performance Computing Applications 32 (2018) 570–581.

[162] M. Bareford, minEPOCH3D Performance and Load Balancing on Cray XC30, Technical Report eCSE03-1, Edinburgh Parallel Computer Centre, 2016.

[163] K. Germaschewski, B. Allen, T. Dannert, M. Hrywniak, J. Donaghy, G. Merlo, S. Ethier, E. D'Azevedo, F. Jenko, A. Bhattacharjee, Toward exascale whole-device modeling of fusion devices: Porting the GENE gyrokinetic microturbulence code to GPU, Physics of Plasmas 28 (2021) 062501.

[164] T. Görler, X. Lapillonne, S. Brunner, T. Dannert, F. Jenko, F. Merz, D. Told, The global version of the gyrokinetic turbulence code GENE, Journal of Computational Physics 230 (2011) 7053–7071.

[165] B. Dudson, M. Kryjak, H. Muhammed, P. Hill, J. Omotani, Hermes-3: Multi-component plasma simulations with BOUT++, 2023. arXiv:2303.12131.

[166] N. Chalmers, A. Mishra, D. McDougall, T. Warburton, HipBone: A performance-portable graphics processing unit-accelerated C++ version of the NekBone benchmark, The International Journal of High Performance Computing Applications 0 (2023) 560–577.

[167] T. Kolev, P. Fischer, M. Min, J. Dongarra, J. Brown, V. Dobrev, T. Warburton, S. Tomov, M. S. Shephard, A. Abdelfattah, V. Barra, N. Beams, J.-S. Camier, N. Chalmers, Y. Dudouit, A. Karakus, I. Karlin, S. Kerkemeier, Y.-H. Lan, D. Medina, E. Merzari, A. Obabko, W. Pazner, T. Rathnayake, C. W. Smith, L. Spies, K. Swirydowicz, J. Thompson, A. Tomboulides, V. Tomov, Efficient exascale discretizations: High-order finite element methods, The International Journal of High Performance Computing Applications 35 (2021) 527–552.

[168] J. C. Camier, Laghos summary for CTS2 benchmark, Technical Report LLNL-TR-770220, Lawrence Livermore National Laboratory, 2019.

[169] A. M. B. Owenson, S. A. Wright, R. A. Bunt, Y. K. Ho, M. J. Street, S. A. Jarvis, An unstructured CFD mini-application for the performance prediction of a production CFD code, Concurrency and Computation: Practice and Experience 32 (2020) 1–14.

[170] P. T. Lin, M. A. Heroux, R. F. Barrett, A. B. Williams, Assessing a Mini-application as a Performance Proxy for a Finite Element Method Engineering Application, Concurrency and Computation: Practice and Experience 27 (2015) 5374–5389.

[171] R. F. Barrett, L. Tang, S. X. Hu, Performance and Energy Implications for Heterogeneous Computing Systems: A MiniFE Case Study., Technical Report SAND2014-20215, Sandia National Laboratories, 2014. doi:10.2172/1494614.

[172] E. Merzari, P. Fischer, M. Min, S. Kerkemeier, A. Obabko, D. Shaver, H. Yuan, Y. Yu, J. Martinez, L. Brockmeyer, L. Fick, G. Busco, A. Yildiz, Y. Hassan, Toward Exascale: Overview of Large Eddy Simulations and Direct Numerical Simulations of Nuclear Reactor Flows with the Spectral Element Method in Nek5000, Nuclear Technology 206 (2020) 1308–1324.

[173] J. Shin, M. W. Hall, J. Chame, C. Chen, P. F. Fischer, P. D. Hovland, Speeding up Nek5000 with Autotuning and Specialization, in: Proceedings of the 24th ACM International Conference on Supercomputing, ICS '10, Association for Computing Machinery, New York, NY, USA, 2010, p. 253–262. doi:10.1145/1810085.1810120.

[174] J. Gong, S. Markidis, E. Laure, M. Otten, P. Fischer, M. Min, Nekbone Performance on GPUs with OpenACC and CUDA Fortran Implementations, The Journal of Supercomputing 72 (2016) 4160–4180.

[175] I. Ivanov, J. Gong, D. Akhmetova, I. B. Peng, S. Markidis, E. Laure, R. Machado, M. Rahn, V. Bartsch, A. Hart, P. Fischer, Evaluation of Parallel Communication Models in Nekbone, a Nek5000 Mini-Application, in: 2015 IEEE International Conference on Cluster Computing, 2015, pp. 760–767. doi:10.1109/CLUSTER.2015.131.

[176] S. Markidis, J. Gong, M. Schliephake, E. Laure, A. Hart, D. Henty, K. Heisey, P. Fischer, OpenACC acceleration of the Nek5000 spectral element code, The International Journal of High Performance Computing Applications 29 (2015) 311–319.

[177] M. Bareford, N. Johnson, M. Weiland, On the trade-offs between energy to solution and runtime for real-world cfd test-cases, in: Proceedings of the Exascale Applications and Software Conference 2016, EASC '16, Association for Computing Machinery, New York, NY, USA, 2016, pp. 1–8. doi:10.1145/2938615.2938619.

[178] M. Bussmann, H. Burau, T. E. Cowan, A. Debus, A. Huebl, G. Juckeland, T. Kluge, W. E. Nagel, R. Pausch, F. Schmitt, U. Schramm, J. Schuchart, R. Widera, Radiative Signatures of the Relativistic Kelvin-Helmholtz Instability, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, ACM, New York, NY, USA, 2013, pp. 5:1–5:12. doi:10.1145/2503210.2504564.

[179] G. Diamond, C. W. Smith, C. Zhang, E. Yoon, M. S. Shephard, PUMIPic: A mesh-based approach to unstructured mesh Particle-In-Cell on GPUs, Journal of Parallel and Distributed Computing 157 (2021) 1–12.

[180] S. McIntosh-Smith, M. Martineau, T. Deakin, G. Pawelczak, W. Gaudin, P. Garrett, W. Liu, R. Smedley-Stevenson, D. Beckingsale, TeaLeaf: A Mini-Application to Enable Design-Space Explorations for Iterative Sparse Linear Solvers, in: 2017 IEEE International Conference on Cluster Computing (CLUSTER), 2017, pp. 842–849. doi:10.1109/CLUSTER.2017.105.

[181] M. Martineau, S. McIntosh-Smith, W. Gaudin, Assessing the performance portability of modern parallel programming models using TeaLeaf, Concurrency and Computation: Practice and Experience 29 (2017) e4117.

[182] R. O. Kirk, G. R. Mudalige, I. Z. Reguly, S. A. Wright, M. J. Martineau, S. A. Jarvis, Achieving Performance Portability for a Heat Conduction Solver Mini-Application on Modern Multi-core Systems, in: 2017 IEEE International Conference on Cluster Computing (CLUSTER), 2017, pp. 834–841. doi:10.1109/CLUSTER.2017.122.

[183] T. Deakin, S. McIntosh-Smith, J. Price, A. Poenaru, P. Atkinson, C. Popa, J. Salmon, Performance Portability across Diverse Computer Architectures, in: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2019, pp. 1–13.

[184] Y. Asahi, G. Latu, V. Grandgirard, J. Bigot, Performance Portable Implementation of a Kinetic Plasma Simulation Mini-App, in: S. Wienke, S. Bhalachandra (Eds.), Accelerator Programming Using Directives, series, Springer International Publishing, Cham, 2020, pp. 117–139.

[185] Y. Asahi, G. Latu, J. Bigot, V. Grandgirard, Optimization strategy for a performance portable Vlasov code, in: 2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2021, pp. 79–91. doi:10.1109/P3HPC54578.2021.00011.

[186] R. Bird, N. Tan, S. V. Luedtke, S. Harrell, M. Taufer, B. Albright, VPIC 2.0: Next Generation Particle-in-Cell Simulations, IEEE Transactions on Parallel and Distributed Systems (2021) 1–1.

[187] N. Tan, R. F. Bird, G. Chen, S. V. Luedtke, B. J. Albright, M. Taufer, Analysis of Vector Particle-In-Cell (VPIC) memory usage optimizations on cutting-edge computer architectures, Journal of Computational Science 60 (2022) 101566.

[188] K. J. Bowers, B. J. Albright, B. Bergen, L. Yin, K. J. Barker, D. J. Kerbyson, 0.374 PFLOP/s Trillion-Particle Kinetic Modeling of Laser Plasma Interaction on Roadrunner, in: SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, 2008, pp. 1–11. doi:10.1109/SC.2008.5222734.

[189] L. Fedeli, A. Huebl, F. Boillod-Cerneux, T. Clark, K. Gott, C. Hillairet, S. Jaure, A. Leblanc, R. Lehe, A. Myers, C. Piechurski, M. Sato, N. Zaim, W. Zhang, J.-L. Vay, H. Vincenti, Pushing the Frontier in the Design of Laser-Based Electron Accelerators with Groundbreaking Mesh-Refined Particle-In-Cell Simulations on Exascale-Class Supercomputers, in: International Conference for High Performance Computing, Networking, Storage and Analysis (SC'22), 2022, pp. 1–12. doi:10.1109/SC41404.2022.00008.

[190] J.-L. Vay, A. Huebl, A. Almgren, L. D. Amorim, J. Bell, L. Fedeli, L. Ge, K. Gott, D. P. Grote, M. Hogan, R. Jambunathan, R. Lehe, A. Myers, C. Ng, M. Rowan, O. Shapoval, M. Thévenet, H. Vincenti, E. Yang, N. Zaïm, W. Zhang, Y. Zhao, E. Zoni, Modeling of a chain of three plasma accelerator stages with the WarpX electromagnetic PIC code on GPUs, Physics of Plasmas 28 (2021) 023105.

[191] A. Myers, A. Almgren, L. Amorim, J. Bell, L. Fedeli, L. Ge, K. Gott, D. Grote, M. Hogan, A. Huebl, R. Jambunathan, R. Lehe, C. Ng, M. Rowan, O. Shapoval, M. Thévenet, J.-L. Vay, H. Vincenti, E. Yang, N. Zaïm, W. Zhang, Y. Zhao, E. Zoni, Porting WarpX to GPU-accelerated platforms, Parallel Computing 108 (2021) 102833.

[192] C. Zhang, G. Diamond, C. W. Smith, M. S. Shephard, Development of an unstructured mesh gyrokinetic particle-in-cell code for exascale fusion plasma simulations on GPUs, Computer Physics Communications 291 (2023) 108824.

[193] S. M. Mniszewski, J. Belak, J.-L. Fattebert, C. F. A. Negre, S. R. Slattery, A. A. Adedoyin, R. F. Bird, C. Chang, G. Chen, S. Ethier, S. Fogerty, S. Habib, C. Junghans, D. Lebrun-Grandié, J. Mohd-Yusof, S. G. Moore, D. Osei-Kuffuor, S. J. Plimpton, A. Pope, S. T. Reeve, L. Ricketson, A. Scheinberg, A. Y. Sharma, M. E. Wall, Enabling particle applications for exascale computing platforms, The International Journal of High Performance Computing Applications 35 (2021) 572–597.

[194] T. D. Arber, K. Bennett, T. Goffrey, S. A. Wright, EPOC++: A Future-Proofed Kinetic Simulation Code for Plasma Physics at Exascale, EPSRC Grant References: EP/W03008X/1, EP/W029111/1, 2022.

[195] P. Fischer, S. Kerkemeier, M. Min, Y.-H. Lan, M. Phillips, T. Rathnayake, E. Merzari, A. Tomboulides, A. Karakus, N. Chalmers, T. Warburton, Nekrs, a gpu-accelerated spectral element navier–stokes solver, Parallel Computing 114 (2022) 102982.

[196] G. Ozen, M. Wolfe, Performant Portable OpenMP, in: Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction,

CC'22, ACM, New York, NY, USA, 2022, p. 156–168. doi:10.1145/3497776.3517780.

[197] S. J. Pennycook, S. A. Jarvis, Developing Performance-Portable Molecular Dynamics Kernels in OpenCL, in: 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, 2012, pp. 386–395. doi:10.1109/SC.Companion.2012.58.

[198] F. Rathgeber, L. Mitchell, D. Ham, M. Lange, A. McRae, F. Luporini, G. teodor Bercea, P. Kelly, Firedrake: Re-imagining FEniCS by Composing Domain-specific Abstractions, 2014. URL: http://kynan.github.io/fenics14, the FEniCS Conference.

[199] G. R. Mudalige, I. Z. Reguly, A. Prabhakar, D. Amirante, L. Lapworth, S. A. Jarvis, Towards Virtual Certification of Gas Turbine Engines With Performance-Portable Simulations, in: 2022 IEEE International Conference on Cluster Computing (CLUSTER), 2022, pp. 206–217. doi:10.1109/CLUSTER51413.2022.00034.