T/NA086/20
Code structure and coordination

2047358-TN-01 Task 1

Approaches to Performance Portable Applications for Fusion

Steven Wright, Ben Dudson, Peter Hill, and David Dickinson

University of York

Gihan Mudalige

University of Warwick

May 26, 2021

Contents

Glossary

AVX Advanced Vector eXtensions

CFD Computational Fluid Dynamics

DIMM Dual In-line Memory Module

DRAM Dynamic Random Access Memory

DSL Domain Specific Language

eDSL Embedded Domain Specific Language

FLOP/s Floating point operations per second

FPGA Field Programmable Gate Array

HBM High Bandwidth Memory

ILP Instruction Level Parallelism

ISA Instruction Set Architecture

JIT Just-in-time Compilation

MCDRAM Multi-Channel DRAM

N-1 N processes writing data to a single file

N-N N processes writing data to their own files

N-M N processes writing to M files

PCIe Peripheral Component Interconnect Express

SIMD Single-instruction, multiple-data

SMT Simultaneous multi-threading

SPMD Single-program, multiple-data

SSE Streaming SIMD Extensions

SVE Scalable Vector Extensions

1 Executive Summary

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 [1]. 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. 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 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 20 cores for high-end processors, and increasing vector lengths (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.

New memory technologies such as High Bandwidth Memory (HBM) has produced "stacked memory" designs where embedded DRAM is integrated on to

CPU chips. New non-volatile memory technologies such as Intel's 3D X-Point (Optane) memory, which can be put in traditional DIMM memory slots, provides higher storage capacity but with lower bandwidth. 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 allows 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. However, this has led to the need of a 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 clear notion of how such devices would in general be programmed" [2].

Nearly 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 run-up to deploying 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.

For most large scientific simulation applications, maintaining multiple versions of the code-base is simply not a reasonable option given the significant time and effort, not to mention the expertise required. Even with multiple versions, it does not guarantee a future-proof application where the next innovation in hardware may well require yet another parallel programming "model" to obtain best performance for the new device. These challenges are now general and applicable equally to any scientific domain that rely on numerical simulation software using HPC systems. As a recent review for applications in the computational fluid dynamics (CFD) domain [3] 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 application, features and maintain existing ones.

Over the years, attempts at developing a general programming model that delivers all three has not had much success. Auto-parallelising compilers for general purpose languages have consistently failed [4]. Compilers for imperative languages as 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 change 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 will now both be GPU based, with one system containing AMD GPUs and the other 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 description of what to compute is separated from how the computation is implemented. This is in direct contrast to languages such as C or Fortran, which explicitly describes the computation.

In this report, we aim to review the key approaches and tools currently used to develop new numerical simulation applications targeting modern HPC architectures and systems, including methods of re-engineering existing codes to modernise them. We focus on applications 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.

The remainder of this report is organised as follows:

Section 2 reviews the current hardware landscape, and outlines the hardware expected in the coming five years. It concludes with a summary of some of the pre- and post-Exascale machines expected in Europe and the United States.

Section 3 discusses current approaches to performance portable scientific application development.

Section 4 reviews a number of libraries and approaches to data and file management at Exascale.

Section 5 concludes this report, highlighting risks and initial recommendations from the report.

2 Pre- and Post-Exascale Hardware

In this section, we briefly introduce the architectures that are available, or likely to become available in the coming years and provide an overview of the state-of-the-art in Supercomputing in the UK, Europe and the rest of the world.

2.1 Computational Hardware

Recent trends in supercomputing suggest that reaching exascale will likely require a heterogeneous approach, or at the very least the use of a manycore architecture (i.e., processors with a high number of parallel cores) [5, 6]. There are already a number of systems in use or in active development that embody this principle – composed of computational nodes coupling a multi-CPU architecture with GPU accelerators.

2.1.1 CPU Architectures

The **Intel Xeon** product line has dominated large HPC installations over the past decade. Currently (as of November 2020) 90% of the Top500 use Intel Xeon processors to provide some or all of their performance. However, coupled with delays to Intel's 10nm and 7nm production processes, there are signs that this dominance may be beginning to wane with a number of recent (and planned) systems opting for **AMD** products, or non-x86 platforms.

The most widely used, and recent, Intel Xeon CPUs are **Skylake** and **Cascade Lake**. Both are manufactured using a 14nm process, with the Skylake being available in configurations with between 2 and 28 cores, and Cascade Lake being available with between 4 and 56 cores. Besides an increase in the number of cores per package, Cascade Lake additionally includes support for **Intel Optane Persistent Memory** DIMMs, and some additional instructions targeted at Neural Networks.

Alongside the large number of processing cores, parallelism is provided by hyper-threading and instruction-level parallelism (ILP) in the form of the Streaming SIMD Extensions (SSE) and Advanced Vector eXtensions (AVX) instruction sets – with Skylake onwards including 512-bit wide AVX instructions, introduced by the **Intel Xeon Phi** product line.

The next server-level architectures available from Intel will be **Ice Lake** and **Sapphire Rapids**, available in mid-2020 and 2021, respectively. Ice Lake will be the first Intel server architecture using a 10nm production process and will provide an additional two memory channels and support for PCIe Gen 4; Sapphire Rapids will additionally add support for DDR5 and PCIe Gen 5. Performance of the Sapphire Rapids architecture will be further boosted by the addition of on-package support for High Bandwidth Memory (HBM) – another feature taken from the Intel Xeon Phi product line.

Intel's main competitor in the x86_64 market comes from AMD in the form of their **AMD EPYC** product line. Following a significant decline in popularity between 2010 and 2015, the AMD EPYC family now power 12 of the Top 100, and this looks set to increase in the coming years. The two current generation EPYC CPUs are **Rome**, based on the **Zen2** microarchitecture, and **Milan**, based on the **Zen3** microarchitecture. Both are manufactured using a 7nm process, with Rome using a 14nm process for I/O components. The are available in configurations up to 64 cores, and support all SIMD extensions up to AVX2 (256-bit), and have support for PCIe Gen 4. In contrast to the current generation Intel CPUs, the AMD Rome and Milan CPUs already provide 8 memory channels – providing a performance boost for memory-bound applications.

The successor to Milan will be **Genoa** in 2022. Little technical information is currently known about the Genoa, but it will be manufactured using a 5nm production process and may be available in configurations up to 128 cores per package. It is also likely to add support for DDR5 and PCIe Gen 5.

Outside of x86 architectures, **IBM** have a long history of successful supercomputers, from the BlueGene/L, through to Sierra and Summit. While Sierra and Summit both get most of their performance from GPU accelerators, they are driven by **IBM Power9** processors. The Power9 CPU is built using a 14nm production process, and is available in configurations up to 24 cores, with additionally parallelism provided by 4-way simultaneous multithreading (SMT4). Like the Rome architecture, memory-bound applications may also benefit from the availability of 8 memory channels. The Power9 architecture is also notable for its inclusion of **NVIDIA's NVLink** protocol. NVLink allows for faster communication between connected GPUs – hence the use of Power9 CPUs on the Sierra and Summit systems.

IBM Power10 will be the next generation of the Power ISA, and will be released in the second half of 2021. It will be manufactured using a 7nm production process and will increase the number of cores up to 48. Alongside support for NVLink v3.0, there will also be additional instructions specialised for AI inference operations.

Another non-x86 architecture gaining in popularity comes from **Arm**. Chipsets based on the ARMv8.1-A and ARMv8.2-A ISAs are available from **Marvell** and **Fujitsu**, among others. The **Marvell ThunderX2** currently powers the UK's Isambard platform [7], and is also installed in Sandia's Astra – the first Petascale supercomputer built using Arm processors. The ThunderX2 is a 14nm platform available with up to 32 cores, with 4-way multithreading. Its successor, the ThunderX3 will not be produced for general-purpose use which may limit its use in HPC systems – beyond availability on cloud platforms.

The Fujitsu manufactured **A64FX** platform currently powers Fugaku, the current #1 supercomputer in the world. The A64FX uses a 7nm production process, and consists of 48 computational cores per CPU (with 2 or 4 assistant cores available on some models). It provides 512-bit Scalable Vector Extensions (SVE) and 32 GB of on-package High Bandwidth Memory (HBM2). Besides Fugaku, the A64FX is also available in other systems and is soon to be available in Isambard-2. The EU's Mont Blanc project is also in the process of pursuing an Arm ISA based supercomputer in the coming years.

Alongside the architectures discussed above, alternative manycore CPUs are available from manufacturers such as **Sunway**, installed in TaihuLight. However, these architectures are unlikely to be widely used in Europe and the US and so are not discussed in this report.

2.1.2 Accelerator Architectures

Computational accelerators provide a significant performance boost to many of the biggest HPC systems, and are the key components in many planned Exascale systems. The most prominent accelerators in the Top500 are **NVIDIA** GPUs.

Manufacturer	Name	Architecture	Key Features
Intel	Cascade Lake	x86_64	10nm, SSE and AVX (up to AVX-512), up to 48 cores, PCIe Gen 5, DDR5
AMD	Genoa	x86_64	7nm, SSE and AVX (up to AVX2), up to 96 cores, PCIe Gen 5, 12-channel DDR5
IBM	Power10	Power ISA	7nm, SMT4, up to 48 cores, NVLink
Fujitsu	A64FX	ARMv8.2-A	7nm, 512-bit SVE, SMT4, up to 48 cores, 32GB HBM2

Table 1: Summary of CPU architectures likely to be present in Exascale Systems

The two most recent generations of NVIDIA GPUs are **Volta** and **Ampere**. Volta (V100) uses a 12nm production process and has a peak double-precision performance of 7.8 TFLOP/s. It provides 16 or 32 GB of HBM2, and can be connected via PCIe Gen 3.0 or NVLink 2.0 – allowing GPU-GPU communication (where available). It was also NVIDIA's first chip to feature Tensor cores, specifically designed to accelerate deep learning.

Similarly, Ampere (A100) is available as a PCIe card (Gen 3.0), or can be connected via NVLink 3.0. It is manufactured using a 7nm process, and can provide 9.7 TFLOP/s of double-precision performance. The A100 is available with 40 or 80 GB HBM2, and features a number of architectural improvements in particular to the Tensor cores.

Details of the next NVIDIA GPU are scant, but it will likely appear in 2022 or 2023, and will be called "**Hopper**". Hopper will be manufactured at 5nm and is likely to provide double the performance of the Ampere architecture.

NVIDIA's main competitor in the discrete GPU space is AMD. The **Radeon Instinct** is AMDs primary offering in the HPC space. The current generation MI50 and MI60 GPUs are manufactured with a 7nm process, and can provide 6.6 and 7.3 TFLOP/s double-precision performance, respectively. Both offer up to 32 GB HBM2, and are connected via PCIe Gen 4.0. Like NVIDIA GPUs, AMD GPUs can also communicate GPU-GPU via Infinity Fabric.

AMD Radeon Instinct GPUs will power the Frontier and El Capitan supercomputers, to be installed at Oak Ridge National Laboratory and Lawrence Livermore National Laboratory, respectively. The MI100 GPU likely to be used in Frontier is produced at 7nm, and has a peak double-precision performance of 11.5 TFLOP/s. Like the MI60, it includes 32 GB of on-package HBM2 and is connected via PCIe Gen 4.0. The MI100 is built around the new AMD Compute DNA (CDNA) architecture, specifically designed with HPC and AI in mind [8].

The Intel Xeon Phi began as a computational accelerator with the Knight Corner (KNC) range. KNC was manufactured on a 22nm process and introduced 4-way SMT and 512-bit wide vector instructions to Intel hardware. Its successor, Knights Landing (KNL), was manufactured on a 14nm process and introduced high bandwidth memory (in the form of MCDRAM), and was available as a host platform, rather than a computational accelerator. While KNL's successor was expected to power Argonne National Laboratory's Aurora supercomputer, the Xeon Phi programme was cancelled in 2017, with many of the features from the Phi range now being available on the Xeon line of CPUs.

In 2018, Intel announced the launch of a new discrete GPU, named **Xe**. Following the cancellation of the Xeon Phi product line, six '**Ponte Vecchio**' Xe-HPC GPUs will be paired with two Sapphire Rapids CPUs in each node of Aurora. It will be manufactured using a 7nm production process, and will feature a new instruction set architecture.

Besides NVIDIA, AMD and Intel, computational accelerators are available from manufacturers such as the National University of Defense Technology (NUDT) and PEZY Computing, but these are unlikely to find widespread usage in systems installed in Europe and the US, so are omitted from this report.

Manufacturer	Name	Architecture	Key Features
NVIDIA	Hopper	Hopper	5nm, ∼20 TFLOP/s, HBM2
AMD	Radeon Instinct MI100	CDNA	7nm, 11.5 TFLOP/s, PCIe gen 4.0, HMB2
Intel	Xe	Unknown	7nm, Unknown

Table 2: Summary of accelerator architectures likely to be present in Exascale Systems

2.1.3 Reconfigurable Architectures

For the past decade, accelerator architectures have demonstrated the benefit of hardware specialisation to achieving high performance. Field-Programmable Gate Arrays (FPGAs) may represent the next step towards application-specific hardware. At compile-time, entire algorithms can be synthesised as sequential logic circuits in hardware [9, 10].

The use of reconfigurable hardware in large HPC installations is currently rare, but there are signs that this may change as new programming models emerge. In particular, both OpenCL and Intel's Data Parallel C++ can target FPGAs directly. Further, since FPGAs can synthesise circuitry specific to a computational kernel, they are able to eliminate computational units that would otherwise be powered but unused on CPU- and GPU-like architectures – potentially reducing energy wastage.

It should be noted that, while a number of recent studies [9, 10] have shown that FPGAs can achieve comparable performance to GPUs on some kernels, specialised non-trivial optimisations are required, coupled with long compilation times. The relative immaturity of the compiler toolchains, means that currently targeting FPGAs may significantly harm developer productivity.

The FPGA market leaders are **Xilinx** (now part of AMD), and **Intel** (following their acquisition of Altera). The **Xilinx Alveo U280** is a 16nm architecture that is connected via PCIe Gen 4.0, and provides 8 GB of HBM2. Intel's data centre offering is the **Stratix 10**, manufactured using a 14nm process. Like the Alveo, it can be connected via PCIe Gen 4 and offers HBM2 – 8 or 16 GB.

On an FPGA, different kernels contain different control flow structures, arithmetic functions, and data types, which all lead to synthesised hardware of hugely varying efficiency. Making direct comparisons between FPGAs and traditional architectures therefore needs to be done based on a higher-level goal, such as time-to-solution of a problem or indeed using a performance portability metric such as discussed later in this report. While many kernels can be optimised on modern FPGAs to outperform similar sized GPUs, implementing full production applications such as from plasma-fusion would be significantly difficult and near infeasible in terms of cost/benefit on current hardware. This is especially the case on latest generation FPGA fabrics, considering the high resource requirement for implementing double precision mathematics.

Manufacturer	Name	Architecture	Key Features
Xilinx	Alveo	FPGA	16nm, HBM2, PCIe gen 4.0
Intel	Stratix	FPGA	14nm, HBM2, PCIe gen 4.0

Table 3: Summary of reconfigurable architectures

2.2 Notable Systems

In 2020, the **Fugaku** system became the fastest supercomputer in the world with a theoretical peak double-precision performance in excess of half an ExaFLOP. The system consists of 160,000 Fujitsu A64FX CPUs and is connected with a 6-dimensional torus interconnect (Torus Fusion). In addition to topping the Top500, Fugaku also tops the Graph500, HPC-AI and HPCG lists – being the first supercomputer to achieve this feat.

Prior to Fugaku, **Summit** and **Sierra** were the #1 and #2 systems, with peak performances of 150 PFLOP/s and 95 PFLOP/s, respectively. Both systems consist of two IBM Power9 CPUs, supplemented with NVIDIA V100 GPUs (6 per node in Summit, 4 per node in Sierra) and a Fat-tree infiniband interconnect.

Also of note, **Sunway TiahuLight** is a 93 PFLOP/s supercomputer powered by 41,000 Sunway SW26010 manycore processors. Each node is connected to 255 other nodes via PCIe Gen 3.0 to form a *supernode*; each supernode is connected via an infiniband interconnect [11].

2.2.1 Planned Systems

Perhaps more interesting than the largest systems currently in use are the systems that are planned for installation in the next three years. These systems offer the best clue as to which hardware any potential programming model or DSL must target.

The Department of Energy have a number of pre- and post-Exascale systems planned. The first to be installed is likely to be **Perlmutter**, being installed at

NERSC. Perlmutter will be a **HPE/Cray Shasta** system using AMD EPYC Milan CPUs and NVIDIA A100 GPUs, with performance that should exceed 100 PFLOP/s [12].

The Cray Shasta architecture allows for a wide range of processors, coprocessors, node configurations, and system interconnects within a new cabinet design. The majority of these systems will be supported by Cray's own new **Slingshot interconnect**, based on "HPC Ethernet". Perlmutter will embrace this flexibility with the system being comprising of two phrases, the first with NVIDIA GPUs, and the second with 64-core AMD CPUs.

The three Exascale systems currently in development are **Aurora**, **Frontier** and **El Capitan**, to be installed at Argonne National Laboratory, Oak Ridge National Laboratory and Lawrence Livermore National Laboratory, respectively. All three systems are also Cray Shasta systems, using the Slingshot interconnect.

The first US supercomputer expected to exceed an ExaFLOP will be Frontier. Frontier will be installed in 2021 and will consist of AMD EPYC Milan CPUs with AMD Radeon Instinct GPUs. Aurora will follow and will be constructed with Intel CPUs and GPUs – with each node being two Sapphire Rapids CPUs, with six Ponte Vecchio GPUs. These systems will be followed in 2023 by El Capitan, expected to exceed two ExaFLOP/s. Like Frontier, El Capital will consist of AMD hardware, with EPYC Genoa CPUs and a next generation Radeon Instinct architecture.

Within Europe, the EuroHPC Joint Undertaking governing body selected 8 sites for supercomputing centres in June 2019. Of these 8 sites, 3 will host pre-Exascale machines capable of at least 150 PFLOP/s. **LUMI** will be installed in Kajaani, in Finland, and will be a Cray Shasta system comprising of AMD EPYC CPUs and AMD Radeon Instinct GPUs. It is expected to be capable of approximately 550 PFLOP/s.

LEONARDO will be installed at Cineca, Italy, and will be a Atos BullSequana system. It will be constructed of Intel Sapphire Rapids CPUs, coupled with 14,000 NVIDIA A100 GPUs, connected with Infiniband.

MareNostrum 5 will be installed within the Barcelona Supercomputing Centre. Its predecessor, MareNostrum 4, comprises of 4 distinct systems – an Intel Xeon system, an IBM Power9 + NVIDIA V100 system, an AMD EPYC +

Radeon Instinct system, and a Fujitsu A64FX system. Likewise, MareNostrum 5 will be two distinct (and currently unknown) systems, but may feature some use of the ARM and RISC-V architectures currently being explored by the EU's Mont Blanc project [13].

2.3 HPC Provision in the UK

The UK's Tier-1 national supercomputer is **ARCHER2**, installed at the Edinburgh Parallel Computing Centre (EPCC). ARCHER2 is a Cray Shasta system, with an estimated peak performance of 28 PFLOP/s. Unlike the US Shasta systems, ARCHER2 is a homogeneous cluster, where the compute is provided by AMD EPYC Rome CPUs, connected with Cray Slingshot fabric.

In contrast to ARCHER2, there is a significant degree of diversity in the UK's regional HPC centres (Tier-2). The N8's **Bede** system, installed at the University of Durham, offers an architecture similar to that found on Sierra and Summit, of IBM Power9 CPUs coupled with NVIDIA V100 GPUs. The **Isambard-2** system, at the University of Bristol, will replace a ThunderX2-based system with an A64FX-based system. Isambard-2 will also contain partitions with many of the other competing architectures likely to be available at Exascale.

Besides these systems, many of the other Tier-2 sites offer predominantly CPU-based systems with small GPU partitions. For example, the **Viking** cluster at the University of York contains both Intel-based CPU nodes, alongside two GPU nodes, each containing four V100 GPUs.

Although the currently available UK systems are relatively small when compared to the European and US systems mentioned here, they are representative of the hardware likely to be available at pre- and post-Exascale. The UK itself is planning to deploy an Exascale supercomputer by 2025[1].

Additionally, there are a number of partnerships in place between UK institutions and US/EU counterparts that mean these systems will be available for UK researchers. Developing applications that are portable between UK systems is therefore vitally important and will ensure that these applications can benefit

[1] https://www.theyworkforyou.com/wrans/?id=2021-02-22.156386.h&s=exascale#g156388.q2

from the performance available on upcoming Exascale systems.

2.4 Summary

The end of the “free lunch” [14] and the breakdown of Dennard scaling [15] has meant that today’s performance improvements come from increasing parallelism rather than clock speed. Server-grade CPUs typically contain 10-50 cores, and offer increasingly wide vector operations. GPUs and other accelerators, that offer hundreds of simple cores, now represent a significant proportion of the compute available on many of the worlds biggest supercomputers.

The diversity of architectures that are, or will be, available at Exascale represents a significant challenge for users of these systems – the majority of pre- and post-Exascale systems currently being installed will use both CPUs and Accelerators to achieve their stated performance. With this in mind, being able to develop applications and algorithms that can exploit the hierarchical parallelism likely to be available on Exascale systems will be vitally important. Even considering the likely prevalence of GPUs, the extensive use of GPU-GPU communication, and MPI-Aware programming models the architectures provided differ sufficiently such that a platform-agnostic approach will be vital to the success of any future-proofed Fusion simulation code.

3 Software Approaches to Exascale Application Development

Considering the systems that are likely to be available in the next 5-10 years, it is clear that heterogeneity is likely to be a key feature, particularly with the efforts to build Exascale systems. With the exception of Fugaku, all announced pre- and post-Exascale systems make use of a CPU architecture coupled with GPU accelerators. As such, achieving high performance on such systems requires exploitation of hierarchical parallelism.

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.

In this section, we outline the programming languages, models and libraries that provide abstractions for developers at various levels to develop applications targeting these systems. Our survey follows much of the findings from [3] together with specific considerations for algorithms of interest for the fusion domain.

3.1 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. On ARCHER, the UK's recently retired Tier-1 resource, Fortran applications accounted for 69.3% of the machine's core hours, while C and C++ applications made up 6.3% and 7.4%, respectively [16]. This skew towards Fortran is in part due to a number of mature applications with large user bases, such as CASTEP and VASP, and its longevity in HPC, meaning that it benefits from mature compiler support more than most other languages.

Although usage of Fortran-based applications currently dwarfs C/C++ applications in HPC, there are signs that this is changing, likely as a result of the levels of support for C/C++ in new programming models and libraries. Of particular note are those that make extensive use of templates. These programming models encourage portability across different hardware – a key motivation as HPC becomes more heterogeneous.

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 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 3.5). 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 (Numba), and whole programs (PyPy). However, the parallel performance of Python remains poor, limited by the Global Interpreter Lock (GIL) present in the reference CPython implementation, PyPy and Stackless Python. Removing this lock has proven difficult, limiting Python's use in HPC to 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, Fortress and X10 (DARPA 2002) 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[2]. 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 (e.g. DifferentialEquations.jl, [17]). It aims to combine the flexibility and high productivity of Python, with

[2] `https://julialang.org/`

high performance.

Developing applications in these general purpose programming languages present 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.

3.2 Parallel Programming Models

In this class we consider the programming models that extend from traditional general purpose programming languages to provide parallelisation both on- and off-node. 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 (in the form of **SSE** and **AVX**) provide parallelism within a core, while threading (or Symmetric Multithreading, SMT) provides parallelism within a node. Parallelism across a system is usually provided in the form of 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.

A developer can aid the compiler with the use of **compiler directives** or **vector intrinsics**. SIMD compiler directives, such as `#pragma omp simd`, were added to the OpenMP 4.0 standard, and should be supported in any compliant compiler. The pragmas allow a developer to indicate that an assumed dependency can be ignored, potentially resulting in the compiler generating vectorisable code that is portable across architectures. However, the compiler may still believe there is a dependency present; in this case, the developer must use intrinsics to directly manipulate the vector registers. This is likely to result in higher performance at the expense of both portability and productivity [18].

Distributing execution across all cores in a node typically requires threading and shared memory, and in HPC this is often done with **OpenMP** [19] – compiler directives are used to parallelise iterations using a fork-join model.

Parallelisation beyond a single node requires inter-node communications. The de facto standard in HPC is the **Message Passing Interface (MPI)** [20]. 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** and **Unified Parallel C**, among others. In this model, communications are hidden to the application developer, but are typically implemented using MPI in the backend library.

3.2.1 Accelerator Extensions

For heterogeneous systems, host code is typically written using the programming languages mentioned previously to coordinate between compute nodes, however, the accelerators themselves usually require a different approach. This is a consequence of the significant differences in the accelerator architectures compared to traditional CPUs.

Each vendor offers their own platform-specific programming model, such as **CUDA** from NVIDIA and **HIP/ROCm** from AMD. However, these approaches are typically not portable between vendors and algorithms often require significant re-engineering. 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. It follows a Single Instruction Multiple Data (SIMD) programming model where large number of threads are executed in lock-step on different data. **OpenCL** largely mirrors the SIMD model of CUDA, having a one-to-one equivalent API, but is developed as an open standard. With CUDA and 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) 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** directives can be used to indicate/instruct a compiler what code can be executed 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 including when data is moved on 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 device leading to significant slowdowns. Currently OpenACC is provided in commercial compilers from 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. 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.

Support for the OpenMP 4.0 and above 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 [21] and GCC open-source compilers, with support for accelerators from NVIDIA, AMD and Intel.

While the explicit device control provided by the CUDA and OpenCL programming model may be more powerful than directive-based approaches, it may also significantly increase developer effort. More recently, the Khronos Group released **SYCL**, a new high-level cross-platform abstraction layer, which can be viewed as a data-parallel version of C++ based on OpenCL. Much 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.

Building on SYCL, Intel announced a 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 standard, providing some extensions to SYCL. Support for SYCL and DPC++ is provided in a number of compilers from vendors such as AMD, Intel, Codeplay and Xilinx, and can target a number of device types directly, or via existing OpenCL targets. In the case of the Intel and Xilinx 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 [22, 23].

Parallelisation based on OpenMP and MPI have a long history in HPC application development. CUDA also now has about 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, OpenACC all support C/C++ and Fortran, OpenCL and SYCL only support C/C++. If indeed C/C++ based 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.
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 [23], for example to handle the different memory layouts required to optimise for CPUs vs GPUs.

3.3 Software Libraries

In this class we consider classical software libraries that target scientific application development, implementing a diverse set of numerical algorithms.

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

The basis of many of these libraries is **BLAS** (Basic Linear Algebra Subprograms), first developed in 1979. BLAS provides vector operations, matrix-vector operations and matrix-matrix operations. **LAPACK** (Linear Algebra Package) builds on BLAS and provides routines for solving systems of linear equations. The **FFTW** 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** provides dense linear algebra kernels for multicore and accelerator architectures [24].

The **Portable, Extensible Toolkit for Scientific Computation (PETSc)** 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 [25].

Similarly, **HYPRE** 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 offload, or using RAJA or Kokkos.

Trilinos is an extensive collection of open-source libraries that can be used to build scientific software, developed by Sandia National Laboratories. It provides a large number of packages for solving linear systems, preconditioning, using sparse graphs and matrices, among many others. It supports distributed memory computation through MPI and also provides shared memory computation through its own Kokkos package. Trilinos is included on Cray supercomputers as part of the **Cray Scientific and Math Libraries**.

The **CoPA Cabana** library provides a number of data structures, algorithms and utilities specifically for particle-based simulations [26]. Parallel execution of particle kernels is achieved through Kokkos for on-node parallelism (see Section 3.4) 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.

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.

3.4 C++ Template Libraries

For this class we consider libraries that facilitate scheduling and execution of data parallel or task-parallel algorithms in general, but themselves do not implement numerical algorithms.

An approach, exclusive to C++ is the use of template libraries, which enables developers to write a generic “template” to express the 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 [27] – indeed, their specification and implementation is often considered as a precursor towards inclusion in the C++ STL. The largest such projects are **Boost** [28], **Eigen** [29], and parallel runtime system, **HPX** [30]. While there are countless such libraries, here we focus on ones that also target performance portability in HPC.

Kokkos [31] is a C++ performance portability layer that provides data containers, data accessors, and a number of parallel execution patterns. It supports execution on shared-memory parallel platforms, namely CPUs using OpenMP and pthreads, and NVIDIA GPUs using CUDA. It does not consider distributed memory parallelism, rather it is designed to be used in conjunction with MPI. Kokkos ships with Trilinos, and is used to parallelise various libraries in Trilinos, but it can also be used as a stand-alone tool. 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 more narrower application domain.

RAJA is a similar abstraction developed by Lawrence Livermore National Laboratory [32]. It is in many respects very similar to Kokkos but offers more flexibility for manipulating loop scheduling, particularly for complex nested loops.

It also supports CPUs (with OpenMP and TBB), as well as NVIDIA GPUs with CUDA.

Both Kokkos and RAJA were designed by US DoE labs to help move existing software to new heterogeneous hardware, and this very much is 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 the cost of moving data to/from the device. Kokkos and RAJA are used heavily within the Exascale Computing Project (ECP) [33], and due to their reliance on template meta programming, can be used alongside almost any modern C++ compiler.

Using C++ template libraries comes with the following considerations:

1. Development time may be high due to high compilation times that come with using heavily templated code.
2. Applications are restricted to being developed in modern C++.
3. Debugging heavily templated code can be difficult, with errors obfuscated by numerous templates. This can be particularly problematic for novice physicist programmers.
4. 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.

3.5 Domain Specific Languages

In this category 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 key challenges to building a successful DSL or framework are:

1. An abstraction that is wide enough to cover a range of interesting applications, but narrow enough so that powerful optimisations can be applied.
2. An approach to long-term support. A feasible model would be to follow the maintenance pattern of classical libraries.

DSLs can be categorised based on their level of abstraction. At a low level a DSL might provide abstractions for sequences of basic algorithmic primitives, such as parallel for-each loops, reduction, scan operations etc. Kokkos and RAJA can be thought of as such loop-level abstractions supporting a small set of computation-communication “patterns”.

3.5.1 DSLs for Stencil Computations

At a higher-level we could consider DSLs for stencil computations, providing abstractions for structured or unstructured stencil-based algorithms. 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. Similar DSLs can be constructed for domains such as molecular dynamics that help express N-body interactions.

There are a number of notable and currently active DSLs at this level of abstractions. **Halide** [34] is a DSL intended for image processing pipelines, but generic enough to target structured-mesh computations [35], it has its own language, but is also embedded into C++ – it targets both CPUs and GPUs, as well as distributed memory systems. **YASK** [36] 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** [37] 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** [38] is part of a larger European project, ExaStencils, 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** [39] 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 [40] and its Python extension, **PyOP2** [41] 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 needs to be carefully handled when parallelising given the existence of data dependencies and as such needs different code-paths to obtain the best performance on different architectures [23]. OP2 generates parallel code targeting CPU and GPU clusters making use of a range of parallel programming models (SIMD, OpenMP, CUDA, SYCL etc. and their combinations with MPI). For mixed mesh-particle, and particle methods, **OpenFPM** [42], embedded in C++, provides a comprehensive library that targets CPUs, GPUs, and supercomputers.

A number of DSLs have emerged from the weather prediction domain such as **STELLA** [43] and **PSyclone** [44]. STELLA is a C++ template library for stencil computations, that is used in the COSMO dynamical core [45], and supports structured mesh stencil computations on CPUs and GPUs. PSyclone is part of the effort in modernising the UK MetOffice's Unified Model weather code and uses automatic code generation. It currently uses only OpenACC for executing on GPUs. A very different approach is taken by the **CLAW-DSL** [46], used for the ICON model [47], which is targeting Fortran applications, and generates CPU and GPU parallelisations – mainly for structured mesh codes, but it is a more generic tool based on source-to-source translation using preprocessor directives. It is worth noting that these DSLs are closely tied to a larger software project (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.

3.5.2 Higher-Level DSLs

Domain specificity can be at even 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), and (semi-) automatically discretise and generate solutions for them. Most are focused on a particular set of equations and discretisation methods, and 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** [48], which targets the Finite Element Method. However it only supports CPUs and distributed memory cluster execution with MPI. **Firedrake** [49] is a similar project with a different feature set, which also only supports CPUs – it 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 [50] 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** [51] 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** [52] 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 paralleisation, where GPU acceleration is obtains by generating OpenACC directives.

In fusion research, the **BOUT++** framework has been developed as a flexible toolbox for solving a wide range of PDEs [53, 54]. 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. For example the MHD equations (Eq. 1-4) can be expressed in C++ as in Figure 1.

$$\frac{\partial \rho}{\partial t} = -\mathbf{v} \cdot \nabla \rho - \rho \nabla \cdot \mathbf{v} \tag{1}$$

$$\frac{\partial p}{\partial t} = -\mathbf{v} \cdot \nabla p - \gamma p \nabla \cdot \mathbf{v} \tag{2}$$

$$\frac{\partial \mathbf{v}}{\partial t} = -\mathbf{v} \cdot \nabla \mathbf{v} + \frac{1}{\rho}(-\nabla p + (\nabla \times \mathbf{B}) \times \mathbf{B}) \tag{3}$$

$$\frac{\partial \mathbf{B}}{\partial t} = \nabla \times (\mathbf{v} \times \mathbf{B}) \tag{4}$$

```
ddt(rho) = -V_dot_Grad(v, rho) - rho*Div(v);
ddt(p)   = -V_dot_Grad(v, p) - g*p*Div(v);
ddt(v)   = -V_dot_Grad(v, v) + (cross(Curl(B),B) - Grad(p))/rho;
ddt(B)   = Curl(cross(v,B));
```

Figure 1: BOUT++ MHD equations implementation

The BOUT++ framework then solves these equations, and allows the user run-time 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 a 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 necessitates 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 or Blitz++), which have their own downsides.

In addition to the above eDSL for implementing physics models, BOUT++ has a second DSL to specify the inputs and initial conditions for the simulations. This started from a simple INI input format, but has developed over time into a Turing-complete language of its own, with a custom interpreter included in

BOUT++. This gradual increase in complexity has been driven by the needs of physics studies, improving ease of use (reducing or eliminating pre-processing steps), and to facilitate testing with complex analytical expressions using the Method of Manufactured Solutions (MMS).

```
[n]  # Density
height = 0.5
width = 0.05

blob1 = height * exp(-((x-0.35)/width)^2
                     - ((z/(2*pi) - 0.5)/width)^2)
blob2 = height * exp(-((x-0.15)/width)^2
                     - ((z/(2*pi) - 0.4)/width)^2)

function = 1 + blob1 + blob2
```

Figure 2: Part of a BOUT++ input file, specifying the density initial condition as a function of position in x and z.

This flexibility in the input has proven to be extremely useful to users, and as a DSL the format is well suited to its specialised task of providing input expressions to BOUT++ simulations. Because of how it has gradually evolved in BOUT++, it is however a DSL with a very limited number of users, with all the disadvantages which come with this discussed previously. BOUT++ currently only supports execution on CPUs with OpenMP for multi-threading and MPI for distributed memory execution. Experimental branches exists with ongoing development to support GPU execution. These include (1) Using Hypre [55] 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 [56] 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. For example [3], the modified Helmholtz equation:

$$-\nabla^2 u + u = f \tag{5}$$

$$\nabla \cdot \hat{n} = 0 \qquad \text{on boundary } \Gamma \tag{6}$$

[3] From ://www.firedrakeproject.org/demos/helmholtz.py.html

```python
from firedrake import *
mesh = UnitSquareMesh(10, 10)   # Define the mesh
V = FunctionSpace(mesh, "CG", 1)  # Function space of the solution
u = TrialFunction(V)
v = TestFunction(V)
f = Function(V)  # Define a function and give it a value
x, y = SpatialCoordinate(mesh)
f.interpolate((1+8*pi*pi)*cos(x*pi*2)*cos(y*pi*2))
# The bilinear and linear forms
a = (inner(grad(u), grad(v)) + inner(u, v)) * dx
L = inner(f, v) * dx
u = Function(V)  # Re-define u to be the solution
# Solve the equation
solve(a == L, u, solver_parameters={'ksp_type': 'cg'})
```

Figure 3: UFL implementation of the Helmholtz equation

can be transformed into variational form by multiplying by a test function v and integrating over the domain Ω:

$$\int_\Omega \nabla u \cdot \nabla v + uvdx = \int vfdx + \underbrace{\int_\Gamma v\nabla u \cdot \hat{n}ds}_{\rightarrow 0 \text{ due to boundary condition}} \tag{7}$$

This can be implemented in UFL as in Figure 3.

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 of 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 it is the case with many complex performance issues there is no silver-bullet to solve all cases.

3.6 Summary

The increasingly diverse range of hardware being used in modern day HPC systems is making programming for these systems much more difficult. While most vendors provide hardware-specific programming models for dealing with heterogeneous parallelism, these are typically not portable between competing architectures and therefore may require significant redevelopment for any new hardware platforms.

Instead, a number of *performance portable* approaches have been proposed and developed. These approaches range from lightweight directive-based approaches, instructing a compiler to parallelise code effectively, to kernelising code specifically for execution on an accelerator.

Achieving high performance on the today's largest HPC systems requires application developers to deal with hierarchical parallelism. For many new applications, this will likely require a mix of programming languages and programming models (e.g. so called "MPI+X"). Additionally, this may require multiple levels of DSL, e.g., a DSL that allows users (domain scientists) to express the equations required, while a lower-level DSL generates efficient application code for execution on a wide-range of hardware. Certainly combining the expertise of DSL developers at these different levels, optimising for a multi-layered solution seems to be the most feasible and performant. Additionally, such an approach appears to provide the best future-ready option with transparent layers aiding in maintenance and extensibility.

4 Data Structures, I/O and Parallel File Systems

While compute has generally accelerated in line with Moore's law, data movement has fast become the bottleneck to achieving high performance. Different hardware typically requires different memory layouts in order to obtain maximum performance.

Alongside data movement between main memory and the CPU, data movement to persistent storage has also historically lagged developments in compute performance. This further degrades application performance, when data is frequently written to disk for visualisation, analysis and error recovery (e.g. checkpointing).

In response to these challenges, the "standard" memory hierarchy has been significantly extended to include new intermediary data storage such as High Bandwidth Memory (HBM), Non-volatile RAM (NVRAM) and Burst Buffers (see Figure 4). Achieving the highest possible performance on these machines requires careful orchestration of data between these layers.

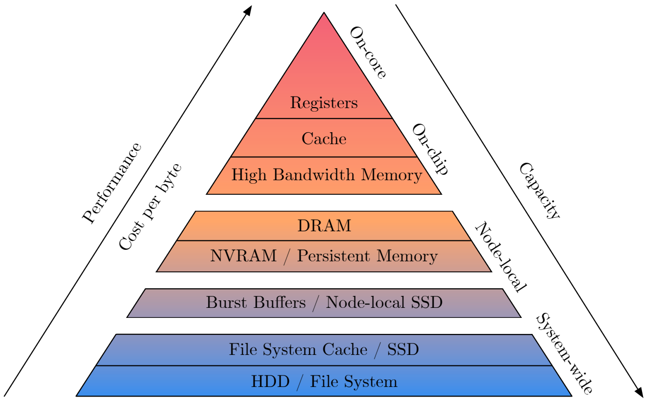

Figure 4: The memory hierarchy

In this section, we outline a number of libraries designed for in-memory data management, a number of libraries designed for the persistent storage of simulation data, and some of the parallel file systems likely to be available on Exascale systems.

4.1 Data Layouts and Memory Management

Modern HPC hardware exposes parallelism in a variety of different ways, ranging from SIMD through to SPMD approaches. Key to achieving high performance from these approaches, is to ensure extraneous data movement between main memory and the CPU is minimised through the use of caches – a cache miss can incur a significant performance penalty.

Minimising this penalty requires application developers to carefully consider which platforms they are targeting and how their simulation data is stored in memory. For example, when deciding between an Array-of-Structures (AoS) or Struct-of-Arrays (SoA) approach, they may wish to consider whether they are targeting CPU-like architectures or accelerator architectures. Picking one in favour of another may impact the performance portability of their application on current- and future-generation architectures [57, 58]. To solve this, a number of libraries are available that abstract away the in-memory data layout, providing a convenient and consistent API to developers.

SIMD Data Layout Templates (SDLT) is a C++11 template library developed by Intel that allows developers to represent arrays of “plain old data” using layouts that enable the C++ compiler to generate efficient SIMD vectorised code.

Similarly, Kokkos provides a View class for creating N-dimensional arrays. The data layout of these “Views” can be specified at compile time and can be adjusted based on the target architecture, whether column or row-major. Additionally, Kokkos views can be manipulated to provide an AoS or SoA-like layout, such as that achieved by VPIC 2.0 [59].

Although RAJA also provides a similar, lightweight view-like container format, they instead encourage use of **CHAI** [60] and **UMPIRE** for data management [56] – with UMPIRE providing the underlying memory management and CHAI providing the user API. Like Kokkos views, data can be allocated on either the host platform or an accelerator device (or both), and data is transferred between the two when required.

There are a number of similar libraries such as **Sidre** (part of Axom) [61] and the **Warwick Data Structure** (WDS) library [62]. Again, they each provide an API for storing scientific data and associated metadata, and have operations for simple data transformations to better enable portable software development.

There are also a number of libraries that specialise in storing data for a particular class of algorithms. Notable examples include **phdMesh** [63] from Trilinos for storing unstructured meshes, **ATLAS** from ECMWF [64] for storing grid and mesh data for numerical weather prediction systems, and **CoPA Cabana** from Los Alamos National Laboratories for storing particle-based data [26].

Much like in the previous section, native memory management (i.e. using malloc or cudaMalloc, etc) and choosing the optimal data layout may provide the highest performance possible for a given platform. However, this can significantly reduce the portability of the application. Abstracting away how data is stored therefore leads to a cleaner, more *performance portable* application, and enables developers to make efficient data transformations when necessary, without having to redevelop large portions of the code base.

4.2 High Performance I/O

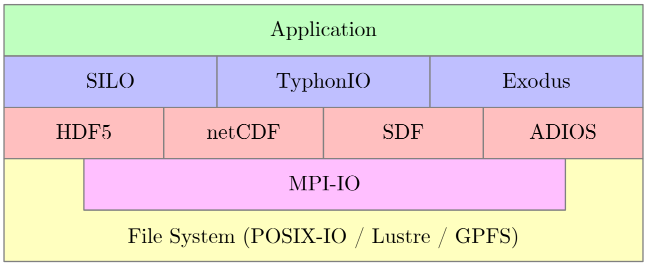

Figure 5: I/O Stack

Just as MPI has become the de facto standard for building massively parallel applications, the **MPI-IO** library has become the standard for managing parallel file accesses. The library contains functionality designed to orchestrate N-1 file writing across a cluster (i.e. N processes writing to a single shared file). The most widely adopted MPI-IO implementation is ROMIO [65], which is used by the vast majority of MPI implementations (OpenMPI, MPICH, etc).

The MPI-IO library gives low-level control to the programmer, akin to POSIX-IO. Because of this, there are a number of file format libraries that exist to abstract these low-level I/O operations from the application. These formats typically have good support in applications such as VisIt.

Perhaps the two most commonly found file formats for scientific software development are **HDF-5** [66] and **NetCDF** [67]. Both implement an tree-like structure for storing large volumes of scientific data and associated metadata. HDF-5 can be compiled with MPI-IO support to enable efficient parallel I/O, and there is a parallel extension of netCDF (PnetCDF) that also uses MPI-IO for low-level file operations. The longevity and widespread adoption of these libraries is testament to their versatility.

The **ADIOS** (Adaptable I/O System) library, developed at Oak Ridge National Laboratory, is a more recent addition to the file format space [68]. ADIOS can support a range of file formats, though typically uses the ADIOS Binary-pack (BP) format, and implements an API similar to POSIX-IO. BP files are containers that allow applications to have a view over a single file, while the backend can parallelise the file operations by writing up to a file per rank (i.e. N-N, N processes each writing to a separate file). This has led to ADIOS demonstrating the highest level of synchronous I/O for a number of key DoE applications.

There are a number of libraries that further abstract I/O operations from an application. Notable examples include **Exodus** [69], from Sandia National Laboratories, **SILO** from Lawrence Livermore National Laboratory [70] and **TyphonIO** from UK AWE [71]. Each of these libraries provide applications with a simple API that can be targeted to alternative formats such as HDF-5, netCDF and ADIOS.

4.2.1 Parallel File Systems

The performance dominant factor when performing I/O operations on an HPC platform is usually the parallel file system that is available. While users typically do not have control over which file system to use, there are often still considerations to be made when choosing which file format library to use and whether there are any configuration options that might provide higher performance.

In a parallel file system, files are typically split into chunks and striped across targets in parallel. The number of targets available typically controls maximum throughput.

The most popular parallel file system in use on the Top500 is **Lustre** [72]. A Lustre system is constructed from a number of Object Storage Servers, each with numerous Object Storage Targets, and a Metadata server for file system information. File striping can be controlled for individual files and directories, but otherwise assumes the systems default. Optimal configuration of these settings can significantly affect application I/O performance – a setting that is too low will reduce the parallelism that is available, a setting that is too high may be cause contention with other jobs on the system.

IBM's Spectrum Scale (GPFS) file system is also popular among large HPC sites [73]. In GPFS, metadata is also distributed, meaning that some operations, such as directory and file creation, can also be parallelised. This may be especially important when applications output data in an N-M or N-N configuration.

Besides Lustre and GPFS, there are a number of other systems in use such as **BeeGFS** [74] and **PVFS2** [75]. In each case, these file systems all provide a POSIX-like interface to users alongside an optimised API. When using MPI-IO, the file system driver can have a significant effect on whether the file system is used efficiently, and whether configuration options such as the stripe count and width can be set per job [76].

As the gap has further widened between compute and I/O performance, the I/O hierarchy has been extended to include node-local (or cabinet-local) burst buffers, or specialised in-situ analysis nodes. Intel's **DAOS** (Distributed Asynchronous Object Storage) system operates above a Lustre file system, using Optane persistent memory and Optane SSDs to provide an object store for parallel applications [77]. DAOS will be deployed on the Aurora supercomputer.

LLNL's El Capitan system will have a similar solution called **Rabbit**. Small groups of El Capitan's compute nodes will be connected to Rabbit I/O nodes that will have capability to run containerised code. These I/O nodes can therefore be configured to act like burst buffers, or could perform analysis in-situ [78].

4.3 Summary

In a similar vein to the previous section, the best achievable performance for an application will require optimisations that are specific to the system in use. However, this will come at the expense of portability and (in most cases) productivity. To address this, there are solutions that abstract these complexities away from applications, making data layout and file I/O operations transparent to the developer, while providing *portable* high performance.

Among the current NEPTUNE proxy applications, some of these libraries are already in use. There is currently ongoing work implementing portions of Bout++ in RAJA using Chai, and it uses netCDF for reading and writing data sets. Similarly, Nektar++ uses HDF5 for its parallel IO.

These libraries remove the complexity of managing scientific data file formats from the application developers, meaning optimisations to file systems and these associated libraries will benefit numerous applications. Furthermore, integration between these libraries and the parallel file systems in use will further boost the potential performance of applications.

5 Risks and Recommendations

The breakdown of Dennard scaling around 2004 has meant that today's performance improvements have come about from increased levels of parallelism, rather than increases in processor clock speeds. In turn, this has led to the widespread adoption of accelerator devices, such as GPUs that can provide hundreds of simple arithmetic cores operating in a SIMD-like fashion. Where the Supercomputing landscape was once dominated by large homogeneous Beowulf-type clusters, it now consists of a wide range of host and accelerator architectures, connected with increasingly complex interconnection strategies.

The recent diversification of hardware has further exacerbated the task of achieving *performance*, *portability* and *productivity* for any given scientific application. Achieving high performance on a heterogeneous cluster relies on careful exploitation of hierarchical parallel and fine grained control over data movement between compute nodes and accelerator devices. Achieving portability between platforms replies on developing applications that can adapt their execution strategies and data layouts/placements to ensure efficient execution, regardless of the platform. Achieving productivity relies on providing developers with a programming model that is powerful enough to express their problems, while high-level enough such that a compiler can generate efficient code, with as much information as possible.

In this report, we have summarised some of the hardware and systems that are currently in development and that will likely provide an ExaFLOP of performance in the coming years. The diversity among these systems will necessitate a vendor-agnostic approach to application development; while programming in CUDA or HIP/ROCm may provide the best performance on NVIDIA and AMD devices, applications developed with these vendor-led solutions are unlikely to be portable between competing architectures.

To combat vendor lock-in, a number of programming models, libraries and domain specific languages have been (or are being) developed. This report has outlined a number of these libraries, and enumerated some of the challenges that exist when using some of these solutions. Ultimately, it is likely that developing applications for execution on an Exascale platform will require a mixture of these programming languages and models (i.e. "MPI+X"). Furthermore, achieving the trinity of performance, portability and productivity for Fusion applications most likely will require the development or adoption of multiple domain specific languages/frameworks operating at different levels – with a high-level DSL for scientists to express their equations, while a low-level DSL is used to produce performance portable application code for execution on HPC platforms. Such a multi-layered approach, we believe, will pose the least risk in maintaining and extending the resultant production applications for Fusion research.

While the compute performance of modern day systems has continued to improve approximately in line with Moore's law, the memory and I/O subsystems have historically lagged. Achieving the highest level of attainable performance is heavily dependant on the optimisation (and sometimes minimisation) of data movement. This report has outlined a number of these solutions that abstract away data movement and I/O operations from application, while providing an easy to use programming interface. These libraries allow compilers to automatically control data data layout and movement, as well as enabling efficient parallel I/O operations – some of these libraries are already in use among the NEPTUNE proxy applications.

5.1 Assessing Performance Portability

The remainder of this project will be primarily focused on assessing the *performance portability* of some of the approaches outlined in this report specifically in relation to fusion applications. In the first instance this will mean gathering performance data from a number of proxy- and mini-applications that are similar to the proposed NEPTUNE proxy applications, but implemented in some of the performance portable approaches discussed. This performance data will be gathered from other studies, or from execution on some of the systems that are available to us during this project.

There are a number of ways to analyse the performance of these miniapps in order to form some insights into how a particular approach to application development might benefit the NEPTUNE project. In the first instance, performance models such as Roofline [79] will be used to identify potential bottlenecks to achieving high performance.

In the second instance, we will assess the portability of an application using the metric introduced by Pennycook et al. [80]. Equation 8 outlines the Pennycook approach to calculating the performance portability (Φ) of an application a, solving problem p, on a given set of platforms (H). In Equation 8, $e_i(a,p)$ is the performance efficiency of application a, calculated as the fraction of performance achieved compared to the best recorded (possibly non-portable) performance of the application solving the same problem, on a given platform, i. Essentially, the metric returns the *harmonic mean* of efficiencies for a given set of platforms, or zero if there is a platform on which the application will not run.

$$\Phi(a,p,H) = \begin{cases} \dfrac{|H|}{\sum\limits_{i \in H} \dfrac{1}{e_i(a,p)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

Since publication of this metric, it has been used extensively to assess the performance portability of a number of applications and programming models [81, 82, 83, 84, 85, 86]. In this project, we seek to replicate this effort with a focus on applications and algorithms of interest to the Fusion community.

5.2 Considerations

This work package seeks to assess a number of approaches to developing a performance portable Fusion applications for the NEPTUNE project. Throughout the project there are a number of risks that must be considered.

- Achieving the highest levels of performance on a given platform will likely require using vendor-led approaches, however applications developed in this manner are unlikely to be portable and may lead to vendor lock-in. For this reason, this work package will not look at vendor-specific libraries, unless there is widespread support, such as with libraries like BLAS.
- A number of the approaches outlined have been developed by large multidisciplinary teams. However, there is always a risk that these projects are wound down or abandoned. Focusing on open standards with widespread adoption and support should minimise this risk.
- Libraries and DSLs are often developed with specific use cases in mind.

Applications must be developed within the bounds of the Library/DSL, or an appropriate escape mechanism must be provided that can retain high performance.

References

[1] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[2] David Patterson. The trouble with multi-core. *IEEE Spectrum*, 47(7):28–32, 53, 2010.

[3] István Z. Reguly and Gihan R. Mudalige. Productivity, performance, and portability for computational fluid dynamics applications. *Computers & Fluids*, 199:104425, 2020.

[4] Jaswinder Pal Singh and John L Hennessy. An empirical investigation of the effectiveness and limitations of automatic parallelization. *Shared memory multiprocessing*, pages 203–207, 1992.

[5] Thiruvengadam Vijayaraghavan, Yasuko Eckert, Gabriel H. Loh, Michael J. Schulte, Mike Ignatowski, Bradford M. Beckmann, William C. Brantley, Joseph L. Greathouse, Wei Huang, Arun Karunanithi, Onur Kayiran, Mitesh Meswani, Indrani Paul, Matthew Poremba, Steven Raasch, Steven K. Reinhardt, Greg Sadowski, and Vilas Sridharan. Design and analysis of an apu for exascale computing. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 85–96, 2017.

[6] Jack Dongarra, Steven Gottlieb, and William T. C. Kramer. Race to exascale. *Computing in Science and Engg.*, 21(1):4–5, January 2019.

[7] Simon McIntosh-Smith, James Price, Tom Deakin, and Andrei Poenaru. A performance analysis of the first generation of hpc-optimized arm processors. *Concurrency and Computation: Practice and Experience*, 31(16):e5110, 2019. e5110 cpe.5110.

[8] AMD. Introducing AMD CDNA Architecture. `https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf` (accessed April 27, 2021), 2020.

[9] K. Kamalakkannan, Gihan R. Mudalige, Istvan Z. Reguly, and Suhaib A. Fahmy. High-level fpga accelerator design for structured-mesh-based ex-

plicit numerical solvers. In *35th IEEE International Parallel & Distributed Processing Symposium*. IEEE, May 2020.

[10] Tan Nguyen, Samuel Williams, Marco Siracusa, Colin MacLean, Douglas Doerfler, and Nicholas J. Wright. The performance and energy efficiency potential of fpgas in scientific computing. In *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 8–19, 2020.

[11] Jack Dongarra. Report on the Sunway TaihuLight System. Technical report, University of Tennessee, June 2016.

[12] NERSC. NUG Monthly Meeting. `https://www.nersc.gov/assets/Uploads/NUG-Meeting-21-Jan.pdf` (accessed April 27, 2021), 2021.

[13] Michael Feldman. Europe Will Enter Pre-Exascxale Realm With MareNostrum 5, 2019.

[14] H. Sutter. The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3):202–210, March 2005.

[15] W. Haensch, E. J. Nowak, R. H. Dennard, P. M. Solomon, A. Bryant, O. H. Dokumaci, A. Kumar, X. Wang, J. B. Johnson, and M. V. Fischetti. Silicon CMOS Devices Beyond Scaling. *IBM Journal of Research and Development*, 50(4.5):339–361, 2006.

[16] Andrew 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, April 2015.

[17] Christopher Rackauckas and Qing Nie. Differentialequations.jl–a performant and feature-rich ecosystem for solving differential equations in julia. *Journal of Open Research Software*, 5(1), 2017.

[18] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, and S. Jarvis. Exploring simd for molecular dynamics, using Intel Xeon processors and Intel Xeon Phi coprocessors. In *Parallel and Distributed Processing Symposium, International*, pages 1085–1097, Los Alamitos, CA, USA, may 2013. IEEE Computer Society.

[19] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1):46–55, January–March 1998.

[20] Message Passing Interface Forum. MPI: A Message Passing Interface Standard Version 2.2. *High Performance Computing Applications*, 12(1–2):1–647, 2009.

[21] David Truby, Carlo Bertolli, Steven A. Wright, Gheorghe-Teodor Bercea, Kevin O'Brien, and Stephen 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)*, pages 10–17, 2018.

[22] Tom Deakin and Simon McIntosh-Smith. Evaluating the Performance of HPC-Style SYCL Applications. In *Proceedings of the International Workshop on OpenCL*, IWOCL'20, New York, NY, USA, 2020. Association for Computing Machinery.

[23] I. Z. Reguly, A. M. B. Owenson, A. Powell, S. A. Jarvis, and G. R. Mudalige. Under the Hood of SYCL – An Initial Performance Analysis With an Unstructured-mesh CFD Application. In *Proceedings of the International Supercomputing Conference (ISC 2021)*, June 2021.

[24] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, June 2010.

[25] Junchao Zhang et al. The petscsf scalable communication layer. *arXiv*, page 2102.13018, 2021.

[26] Los Alamos National Laboratory. CoPA Cabana - The Exascale Co-Design Center for Particle Applications Toolkit. `https://github.com/ECP-copa/Cabana` (accessed April 20, 2021), 2021.

[27] P.J. Plauger, Meng Lee, David Musser, and Alexander A. Stepanov. *C++ Standard Template Library*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[28] Boris Schling. *The Boost C++ Libraries*. XML Press, 2011.

[29] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[30] Hartmut Kaiser, Bryce Adelstein Lelbach, Thomas Heller, Agustín Bergé, Mikael Simberg, John Biddiscombe, Anton Bikineev, Grant Mercer, Andreas Schäfer, Adrian Serio, Taeguk Kwon, Kevin Huck, Jeroen Habraken, Matthew Anderson, Marcin Copik, Steven R. Brandt, Martin Stumpf, Daniel Bourgeois, Denis Blank, Shoshana Jakobovits, Vinay Amatya, Lars Viklund, Zahra Khatami, Devang Bacharwar, Shuangyang Yang, Erik Schnetter, Patrick Diehl, Nikunj Gupta, Bibek Wagle, and Christopher. STEllAR-GROUP/hpx: HPX V1.2.1: The C++ Standards Library for Parallelism and Concurrency, February 2019.

[31] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[32] Rich Hornung, Holger Jones, Jeff Keasler, Rob Neely, Olga Pearce, Si Hammond, Christian Trott, Paul Lin, Courtenay Vaughan, Jeanine Cook, et al. ASC Tri-lab Co-design Level 2 Milestone Report 2015. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2015.

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

[34] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman 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, pages 519–530, New York, NY, USA, 2013. ACM.

[35] B. Mostafazadeh, F. Marti, F. Liu, and 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)*, pages 753–762, May 2018.

[36] C. Yount, J. Tobin, A. Breuer, and 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)*, pages 30–39, Nov 2016.

[37] István Z. Reguly, Gihan R. Mudalige, Michael B. Giles, Dan Curran, and Simon 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, pages 58–67, Washington, DC, USA, 2014. IEEE Computer Society.

[38] Sebastian Kuckuk, Gundolf Haase, Diego A. Vasco, and Harald Köstler. Towards generating efficient flow solvers with the ExaStencils approach. *Concurrency and Computation: Practice and Experience*, 29(17):e4062, 2017.

[39] T. Zhao, S. Williams, M. Hall, and 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)*, pages 59–70, Nov 2018.

[40] G. R. Mudalige, M. B. Giles, I. Reguly, C. Bertolli, and 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)*, pages 1–12, May 2012.

[41] Florian Rathgeber, Graham R Markall, Lawrence Mitchell, Nicolas Loriant, David A Ham, Carlo Bertolli, and Paul HJ Kelly. PyOP2: A high-level framework for performance-portable simulations on unstructured meshes. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 1116–1123. IEEE, 2012.

[42] Pietro Incardona, Antonio Leo, Yaroslav Zaluzhnyi, Rajesh Ramaswamy, and Ivo F. Sbalzarini. OpenFPM: A scalable open framework for particle and particle-mesh codes on parallel computers. *Computer Physics Communications*, 241:155 – 177, 2019.

[43] Oliver Fuhrer, Carlos Osuna, Xavier Lapillonne, Tobias Gysi, Ben Cumming, Mauro Bianco, Andrea Arteaga, and Thomas Schulthess. Towards

a performance portable, architecture agnostic implementation strategy for weather and climate models. *Supercomputing Frontiers and Innovations*, 1(1), 2014.

[44] PSyclone Project, 2018. `http://psyclone.readthedocs.io/`.

[45] Michael Baldauf, Axel Seifert, Jochen Förstner, Detlev Majewski, Matthias Raschendorfer, and Thorsten Reinhardt. Operational convective-scale numerical weather prediction with the COSMO model: description and sensitivities. *Monthly Weather Review*, 139(12):3887–3905, 2011.

[46] Valentin Clement, Sylvaine Ferrachat, Oliver Fuhrer, Xavier Lapillonne, Carlos E. Osuna, Robert Pincus, Jon Rood, and William Sawyer. The CLAW DSL: Abstractions for Performance Portable Weather and Climate Models. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '18, pages 2:1–2:10, New York, NY, USA, 2018. ACM.

[47] V. Clément, P. Marti, O. Fuhrer, and 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*, page 13435, April 2018.

[48] Martin S. Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E. Rognes, and Garth N. Wells. The FEniCS Project Version 1.5. *Archive of Numerical Software*, 3(100), 2015.

[49] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. Mcrae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. Firedrake: Automating the Finite Element Method by Composing Abstractions. *ACM Trans. Math. Softw.*, 43(3):24:1–24:27, December 2016.

[50] Christian Lengauer, Sven Apel, Matthias Bolten, Armin Größlinger, Frank Hannig, Harald Köstler, Ulrich Rüde, Jürgen Teich, Alexander Grebhahn, Stefan Kronawitter, Sebastian Kuckuk, Hannah Rittich, and Christian Schmitt. ExaStencils: Advanced Stencil-Code Engineering. In Luís Lopes, Julius Žilinskas, Alexandru Costan, Roberto G. Cascella, Gabor Kecskemeti, Emmanuel Jeannot, Mario Cannataro, Laura Ricci, Siegfried Benkner, Salvador Petit, Vittorio Scarano, José Gracia, Sascha Hunold,

Stephen L. Scott, Stefan Lankes, Christian Lengauer, Jesús Carretero, Jens Breitbart, and Michael Alexander, editors, *Euro-Par 2014: Parallel Processing Workshops*, pages 553–564, Cham, 2014. Springer International Publishing.

[51] David J. Lusher, Satya P. Jammy, and Neil D. Sandham. Shockwave/boundary-layer interactions in the automatic source-code generation framework OpenSBLI. *Computers & Fluids*, 173:17 – 21, 2018.

[52] M. Lange, N. Kukreja, M. Louboutin, F. Luporini, F. Vieira, V. Pandolfo, P. Velesko, P. Kazakas, and G. Gorman. Devito: Towards a generic finite difference dsl using symbolic python. In *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, pages 67–75, Nov 2016.

[53] Benjamin Daniel Dudson, Peter Alec Hill, David Dickinson, Joseph Parker, Adam Dempsey, Andrew Allen, Arka Bokshi, Brendan Shanahan, Brett Friedman, Chenhao Ma, David Schwörer, Dmitry Meyerson, Eric Grinaker, George Breyiannia, Hasan Muhammed, Haruki Seto, Hong Zhang, Ilon Joseph, Jarrod Leddy, Jed Brown, Jens Madsen, John Omotani, Joshua Sauppe, Kevin Savage, Licheng Wang, Luke Easy, Marta Estarellas, Matt Thomas, Maxim Umansky, Michael Løiten, Minwoo Kim, M Leconte, Nicholas Walkden, Olivier Izacard, Pengwei Xi, Peter Naylor, Fabio Riva, Sanat Tiwari, Sean Farley, Simon Myers, Tianyang Xia, Tongnyeol Rhee, Xiang Liu, Xueqiao Xu, and Zhanhui Wang. BOUT++, 10 2020.

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

[55] Robert D Falgout, Jim E Jones, and Ulrike Meier Yang. The design and implementation of hypre, a library of parallel high performance preconditioners. In *Numerical solution of partial differential equations on parallel computers*, pages 267–294. Springer, 2006.

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

[57] Simon John Pennycook, Simon David Hammond, Steven Alexander Wright, John Andrew Herdman, Iain Miller, and Stephen Andrew Jarvis. An Investigation of the Performance Portability of OpenCL. *Journal of Parallel and Distributed Computing (JPDC)*, 73(11):1439–1450, November 2013.

[58] M. B. Giles, G. R. Mudalige, B. Spencer, C. Bertolli, and I. Reguly. Designing OP2 for GPU Architectures. *Journal of Parallel and Distributed Computing*, 73(11):1451–1460, 2013.

[59] Robert F. Bird, Nigel Tan, Scott V. Luedtke, Stephen Lien Harrell, Michela Taufer, and Brian J. Albright. VPIC 2.0: Next generation particle-in-cell simulations. *CoRR*, abs/2102.13133, 2021.

[60] Lawrence Livermore Alamos National Laboratory. CHAI. `https://github.com/LLNL/CHAI` (accessed April 20, 2021), 2021.

[61] Richard D. Hornung, Aaron Black, Arlie Capps, Ben Corbett, Noah Elliott, Cyrus Harrison, Randy Settgast, Lee Taylor, Kenny Weiss, Chris White, George Zagaris, and USDOE National Nuclear Security Administration. Axom, 10 2017.

[62] R. O. Kirk, M. Nolten, R. Kevis, T. R. Law, S. Maheswaran, S. A. Wright, S. Powell, G. R. Mudalige, and S. A. Jarvis. Warwick data store: A data structure abstraction library. In *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 71–85, 2020.

[63] H. Carter Edwards. phdmesh, version 00, 1 2008.

[64] Willem Deconinck, Peter Bauer, Michail Diamantakis, Mats Hamrud, Christian Kühnlein, Pedro Maciel, Gianmarco Mengaldo, Tiago Quintino, Baudouin Raoult, Piotr K. Smolarkiewicz, and Nils P. Wedi. Atlas : A library for numerical weather prediction and climate modelling. *Computer Physics Communications*, 220:188 – 204, 2017.

[65] Rajeev Thakur, Ewing Lusk, and William Gropp. Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation. Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, October 1997.

[66] Quincey Koziol and R. Matzke. *HDF5 – A New Generation of HDF: Reference Manual and User Guide*. Champaign, IL, 1998.

[67] Russ K. Rew and Glenn P. Davis. NetCDF: An Interface for Scientific Data Access. *IEEE Computer Graphics and Applications*, 10(4):76–82, 1990.

[68] William F. Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, Axel Huebl, Mark Kim, James Kress, Tahsin Kurc, Qing Liu, Jeremy Logan, Kshitij Mehta, George Ostrouchov, Manish Parashar, Franz Poeschel, David Pugmire, Eric Suchyta, Keichi Takahashi, Nick Thompson, Seiji Tsutsumi, Lipeng Wan, Matthew Wolf, Kesheng Wu, and Scott Klasky. Adios 2: The adaptable input output system. a framework for high-performance data management. *SoftwareX*, 12:100561, 2020.

[69] Gregory Sjaardema, Harry Ward, Ron Oldfield, Craig Ulmer, and Shyamali Mukherjee. Snl atdm: I/o and data management. 1 2019.

[70] Lawrence Livermore National Laboratory. SILO – A Mesh and Field I/O Library and Scientific Database. `https://wci.llnl.gov/simulation/computer-codes/silo/` (accessed April 20, 2021), 2021.

[71] James Dickson, S. A. Wright, Satheesh Maheswaran, J. A. Herdman, Mark C. Miller, and Stephen A. Jarvis. Replicating HPC I/O workloads with Proxy Applications. In *1st Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS'16)*. IEEE Computer Society, Los Alamitos, CA, November 2016.

[72] Philip Schwan. Lustre: Building a File System for 1,000-node Clusters. In *Proceedings of the Linux Symposium*, pages 380–386, Ottawa, Ontario, Canada, July 2003. The Linux Symposium.

[73] Frank Schmuck and Roger Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*, pages 231–244, Monterey, CA, January 2002. USENIX Association Berkeley, CA.

[74] Jan Heichler. An introduction to beegfs, 2014.

[75] Philip H Carns, Walter B Ligon III, Robert B Ross, and Rajeev Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th Annaul Linux Showcase and Conference (ALS'00)*, pages 317–327, Atlanta, GA, October 2000. USENIX Association.

[76] Rob Latham, Rob Ross, and Rajeev Thakur. The impact of file systems on mpi-io scalability. In Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 87–96. Springer Berlin Heidelberg, 2004.

[77] Intel. DAOS: Revolutionizing High-Performance Storage with Intel Optane. `https://www.intel.co.uk/content/www/uk/en/high-performance-computing/daos-high-performance-storage-brief.html` (accessed April 20, 2021), 2021.

[78] Tiffany Trader. Livermore's El Capitan Supercomputer to Debut HPE 'Rabbit' Near Node Local Storage. `https://www.hpcwire.com/2021/02/18/livermores-el-capitan-supercomputer-hpe-rabbit-storage-nodes/` (accessed April 20, 2021), 2021.

[79] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2009.

[80] S. J. Pennycook, J. Sewall, and V. Lee. Implications of a Metric for Performance Portability. *Future Generation Computer Systems (In Press)*, 2017.

[81] R. O. Kirk, G. R. Mudalige, I. Z. Reguly, S. A. Wright, M. J. Martineau, and 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)*, pages 834–841, Sep. 2017.

[82] Daniela F. Daniel and Jairo Panetta. On applying performance portability metrics. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 50–59, 2019.

[83] S. L. Harrell, J. Kitson, R. Bird, S. J. Pennycook, J. Sewall, D. Jacobsen, D. N. Asanza, A. Hsu, H. C. Carrillo, H. Kim, and R. Robey. Effective performance portability. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 24–36, Nov 2018.

[84] T. R. Law, R. Kevis, S. Powell, J. Dickson, S. Maheswaran, J. A. Herdman, and 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)*, pages 0–12, Nov 2018.

[85] Simon McIntosh-Smith. Performance Portability Across Diverse Computer Architectures. In *P3MA: 4th International Workshop on Performance Portable Programming models for Manycore or Accelerators*, 2019.

[86] Tom Deakin, Simon McIntosh-Smith, James Price, Andrei Poenaru, Patrick Atkinson, Codrin Popa, and Justin Salmon. Performance portability across diverse computer architectures. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 1–13, 2019.