Area Exam: General-Purpose Performance Portable Programming Models for Productive Exascale Computing

Alister Johnson
Department of Computer and Information Sciences
University of Oregon

Spring 2020

Abstract

Modern supercomputer architectures have grown increasingly complex and diverse since the end of Moore's law in the mid-2000s, and are far more difficult to program than their predecessors. While HPC programming models have improved such that applications are now generally portable between architectures, their performance can still vary wildly, and developers now need to spend a great deal of time tuning or even rewriting their applications for each new machine to get the performance they need. New *performance portable* programming models aim to solve this problem and give high performance on all architectures with minimal effort from developers.

This area exam will survey many of these proposed general-purpose programming models, including libraries, parallel languages, directive-based language extensions, and source-to-source translators, and compare them in terms of use cases, performance, portability, and developer productivity. It will also discuss compiler and general-purpose language standard (e.g., C++) support for performance portability features.

1 Introduction

Since the end of Moore's law and Dennard scaling in the mid-2000s, chip designers are hitting the physical limits of what can be done with single- and multi-core processors. The high performance computing (HPC) community has realized simply scaling up existing hardware will no longer work. As a result, there has been an explosion of new supercomputer architectures in the last two decades that attempt to innovate around these physical limits to reach higher levels of performance. Many of these new architectures use accelerators, such as graphics processing units (GPUs) or Intel's Xeon Phi, as well as traditional CPUs, to achieve higher performance than CPUs alone.

Unfortunately, these types of architectures are very difficult to program, and there is still no consensus on the best way to program them. In addition, the wide variety of architectures means code written for one architecture may not run on another. Over the last couple decades, portable programming models, such as OpenMP and MPI, have matured to the point that most code can now be ported to multiple architectures with minimal effort from developers. However, these models do not guarantee that ported code will run *well* on a new architecture.

Performance tuning and optimization are now the most expensive parts of porting an application. The explosion of new architectures also included an explosion of micro-architectures that can have wildly different performance characteristics, and to make the best use of micro-architectural features, developers often have to rewrite large portions of their applications. For very large applications (100,000+ lines of code), this is prohibitively expensive, and developers need a better option.

Even when developers could rewrite their application, they often want to run on multiple architectures, and must therefore keep multiple versions of their code. Maintaining multiple versions of the same code can be prohibitively expensive as well, since fixes and updates must be added to each version and these versions can diverge as time goes on. Keeping multiple code paths within the same version runs into the same problem, so developers in this situation also need more options.

In short, porting applications to new architectures currently requires excessive developer effort and portable programming solutions are not enough: we need *performance portable* programming models that will increase developer *productivity*. This paper will discuss several potential general-purpose, performance portable programming models that have been introduced in recent years, including:

– Libraries, such as RAJA and SkelCL (Section 3)

– Parallel languages, like Chapel and Cilk (Section 4)

– Directive-based language extensions, including

OpenACC and OpenMP (Section 5)

- Source-to-source translators, such as Omni and Clacc (Section 6)

This paper will prioritize discussion of models that are popular in the HPC community, are currently under development, were recently proposed, and/or provide some novelty, and future directions such models could take, where applicable. This paper will *not* discuss: domain specific languages (DSLs), since these by definition are not general-purpose (although there have been promising results in using DSLs for performance portability); autotuning or other machine learning-based methods for automatic optimization, since those are meant to be used with the programming models, but are not programming models themselves; or generic programming frameworks such as MapReduce or StarPU, since these are not truly programming models, but efficient runtime systems.

In addition, Section 7 will discuss current compiler support for parallelism and potential improvements that these models could make use of. Section 8 will summarize and compare these models in terms of use cases, performance, portability, and developer productivity. Section 2 will begin by providing background information, and Section 9 will conclude.

The main sections of this paper (3, 4, 5, and 6) will be structured as follows. A brief introduction will be given, then each programming model will be described. Each model will be discussed in terms of the "3 P's" (portability, performance, and productivity) and given a (qualitative, subjective) "P3 Ranking" based on how well it meets the criteria for being performance portable and productive, as defined later in Sec. 2.4. All of this will be summarized in two tables at the end.

2 Background

The first part of this section will describe the goals and challenges of exascale computing, as well as how it relates to performance portability. The second part will discuss background information on performance portability, including its most common mathematical definition and several criticisms of that definition. The section will conclude with some definitions and brief descriptions of popular non-(performance) portable programming models, which will be compared to various performance portable models throughout this paper.

2.1 Exascale Computing

The aim of exascale computing is to build a machine that can do 10^{18} floating point operations in a second — 1 exaFLOP. Exascale computing is essential for doing new research in many domains. It will allow simulations to have higher resolution, scientific computations to get results more quickly, and machine learning applications to train on more data.

Current supercomputers can do on the order of 100 petaFLOPs (1 exaFLOP = 1,000 petaFLOPs). Summit (Oak Ridge National Lab) can do ∼150–200 petaFLOPs; Sierra (Lawrence Livermore National Lab) and Sunway TaihuLight (National Supercomputing Center in Wuxi, China) can both do ∼100–125 petaFLOPs [171]. Aurora, arriving at Argonne National Lab in 2021, will theoretically be an exascale machine. China and Japan are aiming to have their own exascale machines by 2020 and 2023, respectively [65].

2.1.1 Goals for Exascale

The U.S. Department of Energy (DoE) set a goal to build an exascale machine that has a hardware cost of less than $200M and uses less than 20MW of power [149]. Aurora will not meet the cost goal, and it's still unknown if it will meet the power goal, but there is hope for future machines.

Other (implicit) goals for exascale computing include (1) making exascale machines "easy" to program, (2) verifying that these machines can do a "useful" exaFLOP, and (3) verifying they can perform sustained exaFLOPs. The first of these is also a goal of performance portability, and will be discussed further in Sec. 2.3 (on productivity).

As for (2) and (3), the origin of these questions goes back to how supercomputer performance is measured. The measurement method used by Top500 is the LINPACK Benchmark, a dense linear algebra solver [170]. LINPACK has been criticized for being overly specific and thus not representative of real applications that will be run on these machines. For example, LINPACK does not account for data transfers, which is one of the bottlenecks on current machines. Very few applications can achieve even close to the peak performance of LINPACK because they cannot make use of all the floating point units on a chip and/or they have to wait on data movement [73].

The HPC community wants an exascale machine that can perform a "useful" exaFLOP with a real application that has these kinds of problems. If the machine can only do exaFLOPs with highly tuned, compute-bound programs like LINPACK, that isn't helpful for domain scientists, whose applications are much more varied. If the machine cannot perform sustained exaFLOPs, but only burst to an exaFLOP under some circumstances (e.g., the kind of dense math performed by LINPACK), that also isn't helpful. Building an exascale machine *that can meet goals (2) and (3)* will be a challenge beyond merely building an exascale machine.

2.1.2 Challenges of Exascale

A DoE report on exascale computing [94] identified the following as the top ten challenges to building an exascale supercomputer. Many other works have also identified a subset of these as major difficulties for exascale systems [149, 74, 107].

1. Energy efficiency — the goal is to use only 20 MW of power, but simply scaling up current technology would use far more than this.

2. Interconnect technology — we need communication to be fast and energy efficient, otherwise an exascale machine "would be more like the millions of individual computers in a data center, rather than a supercomputer" [94].

3. Memory technology — we need to minimize data movement in our programs, make movement energy efficient, and have affordable high-capacity and high-bandwidth memory.

4. Scalable system software — current system software was not designed to handle as many cores and nodes as exascale systems will have. Systems also need better power management and resilience to faults.

5. **Programming systems — we need better programming environments that allow developers to express parallelism, data locality, and resilience, if they so choose.**

6. Data management — our software needs to be able to handle the volume, velocity, and diversity of data that will be produced by applications.

7. Exascale algorithms — current algorithms weren't designed with billion-way[1] parallelism in mind, and we need to rework them or design completely new algorithms.

8. Algorithms for discovery, design, and decision — we need software to be able to reason about uncertainty and optimizations (e.g., error propagation in physics simulations or the optimal instruction set to use for a machine learning algorithm).

9. Resilience and correctness — exascale computers will have many more nodes than current petascale computers, and hardware faults will therefore be more frequent. We need both machines and applications to be able to recover from these faults and guarantee correctness.

10. **Scientific productivity — we want to increase productivity of domain scientists with new tools and environments that let them work on exascale machines easily.**

The two bold challenges, (5) and (10), are of particular interest to performance portability research. Performance portability is concerned with creating programming models that run equally well on multiple architectures/machines; the more difficult question is, how can we do so while allowing developers to express parallelism, data locality, and fault tolerance in ways that won't tie them to a particular machine or get them bogged down in details?

2.1.3 A Brief History of Supercomputing

Before discussing how exascale computing and performance portability impact and inform each other, we need a brief digression to the history of supercomputer architectures.

In the early years, processor performance improvements were mainly due to technological advancements, and processor performance doubled roughly every 3.5 years. In the mid-1980s, reduced instruction set (RISC) architectures became prevalent, and this led to great increases in processor performance – Moore's law was born, saying performance (based on transistor count) would double every 2 years. Dennard scaling, which relates the power density of transistors to transistor size, allowed chip manufacturers to drastically increase the number of transistors per chip while increasing clock frequency, giving large performance gains with essentially the same base architecture. Figure 1 shows this steady growth beginning around 1986. The following decades of steady progress got developers used to performance improvements for "free" – if their application was too slow, they could just wait for the next generation of processor to come out, and (without modifying their code!) it would run faster.

However, in the early to mid-2000s, Dennard scaling began to break down. Chip designers began hitting physical limits, like the power wall: the power density of processors grew so high that scaling any further would make it physically impossible to dissipate the excess heat. Processor and compiler developers began seeing diminishing returns from instruction-level parallelism, and, as Fig. 1 shows, progress began to slow. Processor performance was only doubling every 3.5 years, and it was clear another solution would be needed, which began the transition to multicore chips. After this, performance growth came from increasing the number of cores per chip while clock rates stabilized [74, 149], and we continue to see decreases in performance gains. If the trend of the mid-2010s continues, processor performance will only double every 10-20 years.

[1] To get 10^{18} FLOPs with cores running at 1 GHz (a reasonable approximation of core frequencies in current petascale supercomputers), we would need at least 1 billion cores.

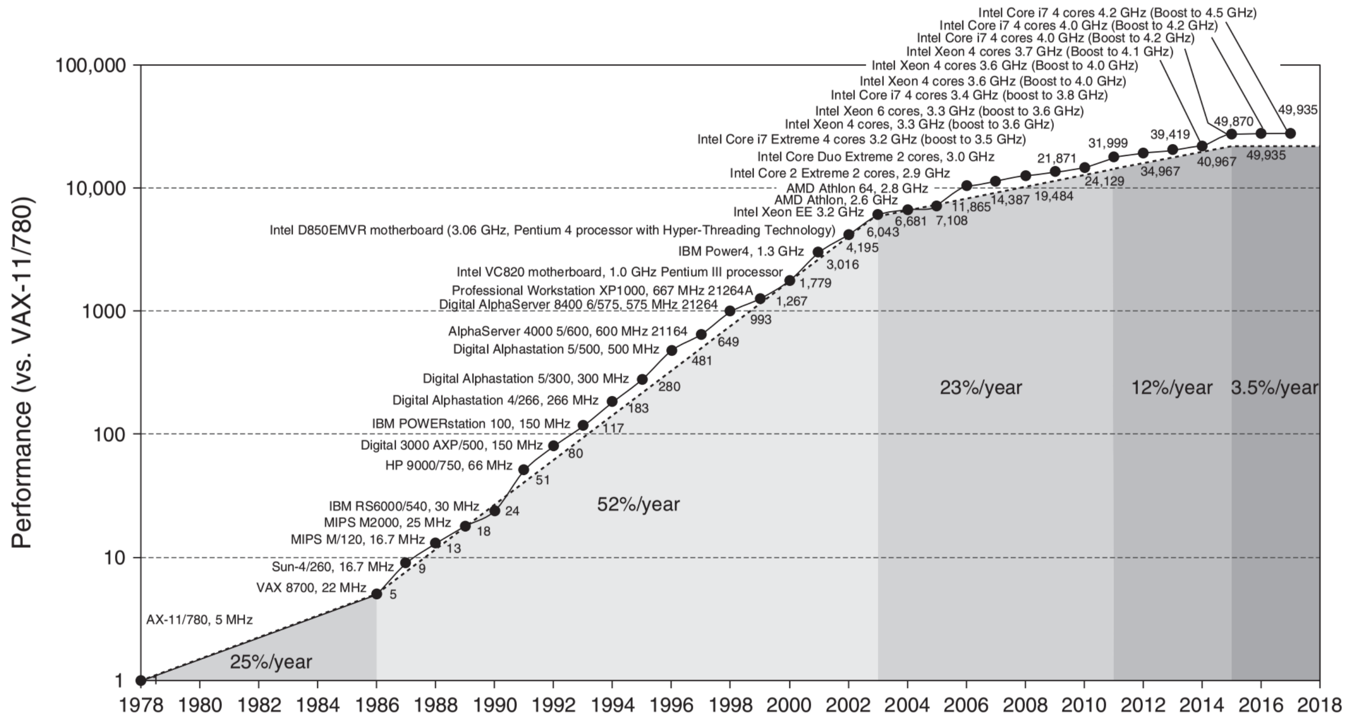

Figure 1: Forty years of processor performance improvements, as measured by SPECint. Image source: [56].

Before the mid-2000s, supercomputers were made up of simple nodes (if they had nodes at all) and programmed with regular programming languages like C and Fortran or, as massively parallel and cluster architectures became more popular, with C and Fortran plus the MPI message passing library, for communication between nodes. Afterwards, as intra-node parallelism increased and MPI's scalability was called into question [166], MPI+X (where "X" is an intra-node parallel programming model) became the default.

2.1.4 Modern Architectures

To cope with the end of Moore's law and Dennard scaling, three families of supercomputer architectures have emerged: heavyweight, based on pre-2004 models with a few powerful cores with high clock speeds; lightweight, with many less powerful cores and slower clock speeds (e.g., IBM's BlueGene architecture); and heterogeneous, a mix of heavy- and lightweight processors, like a CPU+GPU system. Performance projections from 2013 [74] implied that only heterogeneous systems had a hope of making the exascale compute goal within the power limit. Recent developments have borne this out – 7 out of the current top 10 systems have some kind of accelerator, and 5 of these use GPUs as their accelerator [171]. 8 of the top 10 most power efficient machines use GPU accelerators [169].

This proliferation of architectures is shown in Table 1, which lists the architectures of current and future supercomputers from around the world. The HPC community hasn't agreed yet on the best way to program these pre-exascale machines, but we can agree that we don't know what future exascale (and larger) machines will look like. The US DoE machines most recently announced, Aurora [65], Perlmutter [108], and Frontier [15], have wildly different architectures that use different programming models, and users will likely want to run their applications on all of them at some point.[2] Future machines may even have multiple types of accelerators in each node, each specialized for a different type of computation, or programmable coprocessors, like FPGAs [184], that users want to use at the same time. Therefore, the HPC community wants to "future-proof" its applications by developing new *performance portable* programming models.

2.1.5 Why Performance Portability Matters

Currently, developers need to expend effort to port their applications to a new machine, and speedup is no longer guaranteed. Sometimes application development teams spend months porting and optimizing for a new architec-

[2]Aurora will have Intel CPUs and Intel GPUs, Perlmutter will have AMD CPUs and Nvidia GPUs, and Frontier will have AMD CPUs and AMD GPUs. Each type of GPU uses a different (incompatible) programming model.

	Lifespan	Architecture	CPU vendor	CPU version	Accel. vendor	Accel. version
Aurora	2021–	Cray Shasta	Intel	Xeon	Intel	X^e
Frontier	2021–	Cray	AMD	Epyc	AMD	Radeon Instinct
Fugaku	2021–	Fujitsu	Fujitsu	A64FX	N/A	N/A
Perlmutter	2020–	Cray Shasta	AMD	Epyc	Nvidia	Unknown Tesla
Frontera	2019–	Dell C6420	Intel	Xeon (Cascade Lake)	N/A	N/A
Summit	2018–	IBM AC922	IBM	Power9	Nvidia	V100
Sierra	2018–	IBM AC922	IBM	Power9	Nvidia	V100
ABCI	2018–	Fujitsu CX2560 M4	Intel	Xeon (Skylake)	Nvidia	V100
Theta	2017–	Cray XC40	Intel	Xeon Phi (KNL)	N/A	N/A
Gyoukou	2017–2018	ZettaScaler-2.2	Intel	Xeon (Broadwell)	PEZY	PEZY-SC2
TaihuLight	2016–	Sunway MPP	Sunway	SW26010	N/A	N/A
Cori	2016–	Cray XC40	Intel	Xeon Phi (KNL)	N/A	N/A
Oakforest-PACS	2016–	Fujitsu CX1640 M1	Intel	Xeon Phi (KNL)	N/A	N/A
Trinity	2015–	Cray XC40	Intel	Xeon (Haswell)	Intel	Xeon Phi (KNL)
Tianhe-2A	2013–	TH-IVB-FEP	Intel	Xeon (Ivy Bridge)	NUDT	Matrix-2000
Titan	2012–2019	Cray XK7	AMD	Opteron	Nvidia	K20X
Piz Daint	2012–	Cray XC50	Intel	Xeon (Haswell)	Nvidia	P100
Sequoia	2012–	BlueGene/Q	IBM	Power BQC	N/A	N/A
Mira	2012–	BlueGene/Q	IBM	Power BQC	N/A	N/A
K computer	2011–2019	Fujitsu	Fujitsu	SPARC64 VIIIfx	N/A	N/A

Table 1: Recent supercomputers, from the 2015-2019 Top500 lists and various announcements.

ture, only to have to repeat all that work again a year or two later when the next machine comes out. Many HPC applications have a lifespan measured in decades, while supercomputers usually last far less than that. Being forced to refactor for new machines every few years leaves these applications fragile and error-prone, since the time developers have to test, debug, and otherwise improve their code decreases significantly – the current situation is actively harming work done by domain scientists [148, 96].

The end goal of performance portability is to solve this problem and minimize the work users need to put into porting and optimizing their programs for future architectures. Instead of rewriting the same code again and again, developers will have time to improve the functionality of their application. To quote the OpenACC website: "more science, less programming" [120].

2.2 Performance Portability

Performance portability is not new, but increased interest in it is. Computer architectures have gone through (and are continuing to go through) so many shifts that programmers have had to port or rewrite their code multiple times, which isn't sustainable in the long run (see Sec. 2.1.5).

When developers port an application to a new architecture, they do not want to be locked into that architecture – they want to move between architectures with minimal porting effort. In addition, they want their application to perform well on new architectures with minimal optimization and performance tuning. Developers want their applications to be *performance portable*.

2.2.1 Definitions

Several different definitions of performance portability have been proposed in recent years. Some definitions, as listed by Pennycook et al. [137]:

1. An approach to application development, in which developers focus on providing portability between platforms without sacrificing performance [137].
2. The ability of the same source code to run productively on a variety of different architectures [78].
3. $P_n^{\mathscr{P}}(b \to t) = \frac{S_n^t}{S_n^b} \times 100\%$ for program $\mathscr{P}$, base system b, target system t, and speed-up on n nodes S_n [186].
4. The ability of an application to achieve a similar high fraction of peak performance across target devices [102].
5. The ability of an application to obtain the same (or nearly the same) performance as a variant of the code that is written specifically for that device [37].

While each of these definitions has its strengths, each also has weaknesses. Definition (1) is intuitive and provides a good baseline for determining if an application

can claim to be performance portable, but it is subjective and provides no way to measure how performance portable an application is. Definition (2) restricts the application code to a single version, which is desirable for many reasons, but suffers from the same problems as (1) (how should we define "productively?"). Definition (3) does provide a metric, but this metric is difficult to compare between applications, systems, and program inputs. Definition (4)'s metric is problematic because it is somewhat subjective, peak efficiency is difficult to measure (see Sec. 2.2.2 on architectural efficiency), and two architectures might be sufficiently different that achieving peak on one is simple, but on another is impossible (e.g., one machine has vector units that the application can't utilize). Definition (5) is similarly somewhat subjective, and might be difficult to measure if code version for a certain device doesn't exist.

An ideal definition of performance portability would be objective and provide an easy way to both measure *and* compare values for different applications. The most commonly utilized performance portability definition is from Pennycook et al. [137], because it comes with such a metric. However, there are still several criticisms of Pennycook et al.'s metric, which will be discussed in the next section.

2.2.2 Primary Metric

Pennycook et al.'s definition of performance portability is "a measurement of an application's performance efficiency for a given problem that can be executed correctly on all platforms in a given set" [137]. Pennycook et al. designed their definition to reflect both the performance and portability aspects. In addition, they specifically mention executing *correctly* on a *given problem* to ensure that applications ported incorrectly are not considered portable and to note that different inputs can yield different performance characteristics.

The corresponding metric is defined as the harmonic mean of the efficiency of the application on all supported platforms in a set (see Smith [152] for the logic behind choosing the harmonic mean). When one or more platforms are not supported, the metric goes to zero:

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

where a is the application, p is the problem/input, H is the set of platforms, and $e_i(a,p)$ is the efficiency of application a solving problem p on platform $i \in H$. Higher is better — Φ will be high when an application ports well to all architectures in H, and low when it only ports well to a few (or none) of them.

The Φ metric can be used by comparing application performance to either the theoretical peak performance of the architecture — the *architectural efficiency* — or the best known performance of the application on any architecture — the *application efficiency*. Pennycook et al. note that both these efficiencies are important since just looking at one can bias results. Only considering architectural efficiency can give artificially low values for Φ if an application physically cannot take advantage of architectural features (e.g., fused multiply-add instructions or vectorization), and only looking at application efficiency can give artificially high results when no truly efficient implementation exists.

The rest of this section will discuss some criticisms and improvements suggested since Pennycook et al. first published their metric.

Architectural Efficiency. Dreuning et al. [34] note that, when using architectural efficiency, we must choose between comparing the application's achieved FLOPs *or* achieved memory bandwidth to the architecture's theoretical peaks. Choosing the wrong peak can lead to incorrectly high or low values of Φ depending on whether an application is compute or memory bound. Dreuning et al. suggest fixing this by using a performance portability model similar to the Roofline model [176]: compute bound applications should be compared to the platform's theoretical peak FLOPs, and memory bound applications to theoretical peak bandwidth. The operational intensity (operations per memory access) of the application compared to the hardware can be used to determine whether applications are compute or memory bound.

Yang et al. [183] add that, when using the Roofline model, we need accurate theoretical ceilings. In particular, they note that vendors can be optimistic when reporting numbers, so real measurements should be used. In addition, application FLOPs should not be counted by hand, because compiler optimizations and special hardware units can make hand-counted FLOPs inaccurate. Yang et al. demonstrate exactly how far off vendor estimates of their hardware's performance can be, as well as how different hand-counted FLOPs can be from measured FLOPs. Ofenbeck et al. [119] similarly describe why using hardware counters is more accurate than (vendor) estimates, although still problematic for reasons that are outside the scope of this survey.

Yang et al. also note that we need to choose *relevant* ceilings when measuring performance portability. Using an unrealistic theoretical peak will give artificially bad results. For example, if the application is not using fused multiply-add (FMA) instructions, the ceiling measurement should also not use FMA. Knowing how close an application truly is to the platform's most relevant roofline will help developers decide how and where to optimize

their code.

Application Efficiency. Architectural efficiency can give a theoretical upper bound for application performance, but does not tell us whether an application is actually efficient. Application efficiency can give us a practical upper bound and does not require estimation, but can be prone to bias. Sometimes there is no good, efficient implementation, or the developers are unaware of an implementation that performs better than their chosen reference. Dreuning et al. note that the best implementation of an application may be from another programming model that developers are unfamiliar with [34]. There is also a long history in high performance computing of using statistics to obfuscate application performance results or make numbers appear more favorable [7, 47, 131], which only makes matters more murky.

Platform Set Choice. Another criticism from Dreuning et al. is the need for an additional metric for platform set diversity [34]. If an application does well on one type of architecture (e.g., a Xeon Phi), then a platform set containing only that architecture will give artificially high Φ. Indeed, in their original paper, Pennycook et al. note that their metric is only useful when the platform set is known [137]. Researchers using Pennycook et al.'s metric need to carefully consider their choice of platform set.

Ideally, applications would support every architecture, but there are often trade-offs between optimizing for performance and optimizing for performance *portability* – what's good for one platform is not necessarily good for another. Optimizing for one platform will either improve performance on all platforms, or improve performance on a subset of the platforms to the detriment of the others. Optimization is likely to improve performance portability overall, but may actually decrease it, and there's no way of knowing what will happen beforehand.

Based on these observations, Dreuning et al. give three ways to improve an application's performance portability:

- Add platforms to the set, especially ones similar to platforms it already performs well on, since the application is more likely to perform well on those, and/or require less work to perform well.
- Remove poorly performing platforms from the set, if there are sufficiently few of them and/or the optimization effort required would outweigh the performance benefits.
- Do the work of improving performance on some or all platforms.

2.2.3 Other Metrics

Other metrics for performance portability have been proposed as alternatives or extensions to Pennycook et al.'s metric. This section will describe two recent proposals.

$\mathbf{PP_{MD}}$. The PP_{MD} metric is motivated by the experiences of Sedova et al. [148] investigating the performance portability of various molecular dynamics applications. The applications they looked at are all best-of-class (or nearly so) and get high performance on many HPC platforms (so they all score very highly in Φ under application efficiency). However, these applications got there with high-effort ports and are thus not truly performance portable, since migrating to a new platform would likely result in a great deal more work specific to that platform. Sedova et al. desired a metric that would take different source versions and program components into account, so they developed their PP_{MD} metric.

Sedova et al.'s metric is based on the sources of speedup in a particular code. If speedup comes mostly from portable program components, like standard libraries, the application's PP_{MD} value will be high. If non-portable components, such as CUDA kernels, are responsible for most of the speedup, PP_{MD} will be low. The mathematical definition of their metric is similar to Pennycook et al.'s, using the harmonic mean of the fraction of speedup from non-portable components:

$$PP_{MD}(a,p,Q) = \begin{cases} \dfrac{|Q|}{\sum_{i \in Q} S_i(a,p)}, & \text{if } G \neq Q \text{ and } Q \neq \emptyset \\ 1, & \text{if } Q = \emptyset \\ 0, & \text{if } G = Q \end{cases}$$

where G is the set of all program components that contribute to speedup for application a on input p, Q is the subset of G that is non-portable, and S_i is the speedup over baseline application performance that component $i \in Q$ is responsible for. Ideally, all program components responsible for speedup will be portable, so Q will be empty and PP_{MD} will be 1. PP_{MD} doesn't depend on any concept of "peak performance," which is helpful since measuring peaks is difficult, but assumes that program components don't interact with or influence each other and can be measured independently, which may not be the case. Sedova et al. calculated PP_{MD} for each of their applications, and none scored particularly well; the best only got around 40%, which is because many of them use CUDA or other vendor-specific libraries (vendor implementations of standard libraries can vary in performance). Sedova et al.'s PP_{MD} metric might be a good companion to Pennycook et al.'s Φ, since it does penalize applications for using non-portable programming models, whereas Φ does not.

$\mathbf{P_D}$. Pennycook et al.'s metric gives different values for the same application on different inputs, which can make it difficult to see whether an application is truly performance portable, since different platforms might do better on different problem sizes. Daniel et al. [25] propose an alternative metric to better cope with problem size variations, which they call P_D, performance portability divergence. They compute P_D as the root-mean-square of the relative error in performance compared to the "best" performance (using either architectural or application efficiency):

$$P_D = \frac{\sum_{i \in H} \Delta_{RMS}}{|H|}$$

$$\Delta_{RMS} = \sqrt{\frac{\sum_{s \in S} \delta(a, \alpha)^2}{|S|}}$$

where a and α are the Φ values for two applications on the same input, $\delta(a, \alpha)$ is the relative error, S is the set of all input sizes, and H is the set of all platforms. Ideally, an application will perform identically on all inputs across all architectures, and performance portability divergence will be zero.

While this does give users more information on how an application behaves across problem sizes, Daniel et al.'s definition of application does not exclude programs with multiple source versions, so applications can still get good P_D scores by writing code specific to one architecture. A combination of this metric and PP_{MD}, perhaps, might be a better extension to Φ than this metric alone, since it's already been decided keeping multiple code versions is not a sustainable solution.

2.2.4 More History

Past research placed the burden of performance portability solely on the application, but more current work has shifted it to languages, compilers, and other programming tools [179]. For example, when the developers of the Weather Research and Forecast (WRF) model were trying to make the application more performance portable across vector-based and RISC-based computers in the late 1990s/early 2000s [106], they primarily looked into reordering the loops to improve memory access patterns, not introducing a new library or compiler that would do it for them.[3] However, most application teams today are looking for solutions that won't require modifying their code (or at least, won't require modifying it more than once), which (interestingly) may involve writing an application-specific performance portability layer (see Sec. 3.3), going back to placing more of the burden on applications. This survey will discuss many other potential solutions.

[3]Interestingly, they did discuss looking into source-to-source translators that would automatically reorder loops based on the type of machine, but at the time decided not to since they were able to get satisfactory performance without translation. This is an avenue new work is exploring though; see Secs. 5.6 and 6 for more.

2.3 Productivity

Developer productivity is an important (though often overlooked) aspect and motivating factor of performance portability. If it takes years to port an application to a high-performance portable model, that model is not as useful as another that can be adopted more quickly (even at the cost of lower performance), especially if the port cannot be done incrementally so the application can still be used in the meantime. Performance portable models should *increase* developer productivity overall and allow developers to spend less time writing (and rewriting) code during the porting process. This section will discuss issues relating to defining and measuring productivity, as it relates to HPC and performance portability.

2.3.1 Defining Productivity

Productivity is a very subjective, qualitative concept and it means different things in different contexts, making it difficult to define and measure. An intuitive definition might be "getting higher performance with less time/effort," but how do we define (and measure) effort?

2.3.2 Measuring Productivity

There have been numerous attempts to measure software developer effort and productivity across both industry and academia, but measuring *HPC* developer productivity is a very different problem. Most industry productivity measurement tools look at how much code a developer writes, but in HPC, developer productivity isn't just time spent writing code – it's time spent optimizing, tuning, parallelizing, and porting existing code.

There are two main ways to measure developer effort and productivity: direct logging (e.g., Wienke et al. [175] and Harrell et al. [53]) and indirect approximations. Lines of code (LOC) and code divergence (how "different" code versions are) are common metrics for approximating code complexity, and hence, developer effort, but they don't take into account how difficult writing a single line of code can be (e.g., a complex OpenMP pragma versus a simple variable declaration) or development time not spent coding, such as time spent making performance measurements or designing new features with team members [175]. It is not always clear whether these approximations are accurate, and they don't include all of the productivity data we want, which is why direct logging approaches are also important.

Development time logs would be the best metric for developer effort, since they contain data on what devel-

opers were working on and what their results were, but this data is very difficult to collect. Developer diaries sound like a good idea in theory, but in practice there are numerous problems. They require a high level of commitment from developers to collect the type, regularity, and granularity of data that would be useful, but this often doesn't happen because developers don't want to take the time, are inconsistent in their entries, or simply forget. At the other end of the spectrum, automated activity monitoring software that tracks data such as keystrokes and applications used requires no commitment from developers, but misses out on development activities that happen away from the computer, such as planning and training, as well as performance data.

Wienke et al.'s [175] direct logging method tries to combat these problems by creating a journal that pops up at predefined intervals (so it can't be forgotten) and provides a short form with multiple choice questions (so a quick entry still has useful data) and an open-ended comment section. However, their tool doesn't collect performance data very often, and doesn't seem associate that data with the particular code version it came from.

Harrell et al.'s direct logging method [53] was motivated by the desire to keep productivity and performance data *with* their code and better track changes over time, improving on Wienke et al. They wanted to minimize disruption for developers, so they integrated the logging process with their projects' version control system (`git commit`). Harrell et al. used these logs to confirm their intuition about using approximations such as LOC and code divergence as proxies for developer effort; in particular, they demonstrate that low divergence ports (e.g., adding OpenMP or OpenACC directives) are often less expensive in terms of developer effort than high divergence ports (e.g., rewriting all compute loops as CUDA or OpenCL kernels).[4]

2.3.3 Productivity and Performance Portability

Harrell et al.'s primary criticism [53] of Φ is similar to Sedova et al.'s [148]: it does not penalize applications that keep separate code versions for each platform, which are difficult to maintain and require much more developer effort to create. Indeed, one of the primary goals when developers were first porting applications to the Titan supercomputer (one of the first major accelerator-based machines) was to avoid "version bifurcation," which Joubert et al. equated with a loss of maintainability [68]. Maintaining separate code paths or separate optimizations in a single version runs into similar problems. This is a good representation of the trade-off between programming for performance and programming for productivity and maintainability [68] – one goal of performance portability is to do away with this trade-off.

Harrell et al. suggest there is a need for a metric to measure developer effort to achieve performance. Such a metric could be used to penalize applications and programming models that require high amounts of developer effort to obtain and/or maintain performance portability. They note again that there are often trade-offs between programming model abstraction levels (a proxy for portability) and performance. Models with higher abstraction levels are usually easier to port to new architectures, but have lower performance. Models with low levels of abstraction have better performance, but are much more difficult to port. A metric for productivity would help quantify this trade-off.

[4] As an example of this intuition, a survey of the CAAR teams, who were porting applications to Titan for the first time, revealed that 85% of the developers (most of whom were using CUDA, which is known for being difficult to adopt) felt the amount of effort to needed to get good performance on accelerator architectures was "moderate to high." Many expressed interest in moving to directive-based models like OpenACC [68], which is generally considered easier to adopt, even though there was little evidence to back this interest up at the time.

2.4 Some Definitions

For the purposes of this paper, qualitative definitions of both performance portability and productivity will be used, since this is not a quantitative comparison of programming models, but rather a discussion of how each model is working towards performance portability and productivity and how these models can inform and improve each other.

This paper will consider a model *performance portable* and *productive* if:

- it supports most or all major architectures currently in use and can be easily extended to potential future architectures (if an application team suddenly discovers they need to use another architecture, they can easily do so, regardless of what the architecture is);
- it can achieve reasonable performance on each architecture it supports with minimal source code changes (configuration changes are acceptable, as long as they occur outside the code);
- it reduces the burden on developers compared to other models (e.g., CUDA) in significant ways (e.g., shorter code, fewer code changes for porting, easier debugging, etc.).

2.5 Non-(Performance) Portable Programming Models

This paper will often mention various non-portable programming models (or non-performance portable models)

as comparison points for the performance portable models discussed. These non-portable models have generally been highly tuned for their one architecture, and can provide a good estimate of peak application efficiency and performance. This section will describe the main non-portable models mentioned.

2.5.1 CUDA

CUDA [118] is Nvidia's proprietary C++-based GPU programming language, which naturally only works on Nvidia devices. CUDA allows developers to write compute kernels in a subset of C++ that will be run on a machine's GPU. The downsides of CUDA are that it requires users to manually manage memory movement between the host and GPU, and the parallelism model for writing kernels can be non-intuitive for newcomers. This makes CUDA code difficult to read and debug, and porting an existing application to CUDA requires major changes and restructuring. Nvidia has released several libraries for CUDA, including cuBLAS (dense linear algebra), cuSOLVER (linear system solvers), cuDNN (neural networks), and Thrust [10], a high-level, productivity-oriented library containing general purpose algorithms, in an attempt to resolve these problems, however, CUDA is still not portable.

2.5.2 OpenCL

OpenCL [72] is a programming language similar to CUDA and developed by the Khronos Group, but as an open standard, it is implemented for far more devices. OpenCL supports GPUs, CPUs, and FPGAs from multiple vendors, including AMD, Intel, and Nvidia. OpenCL requires a great deal of boilerplate code to set up data structures and kernels, and kernels are represented as strings in the application, which makes debugging difficult. Like CUDA, OpenCL also has many libraries dedicated to reducing the severity of these problems. Unfortunately, since OpenCL is so low-level (arguably more so than CUDA), different code is required to get good performance on different architectures. While OpenCL is portable, it is not *performance* portable.

2.5.3 OpenMP 3

OpenMP [124] is an open standard for directive-based extensions to C, C++, and Fortran that enables developers to add parallelism to their applications by annotating their code with various pragmas. Early versions of OpenMP (≤ 3) were solely for CPUs, and since it is much simpler than almost every other parallel programming model available, OpenMP became very popular. Vendors heavily optimized their implementations, and as a result OpenMP is generally very high performance, but only on CPUs. OpenMP 4.0 began to add support for other architectures, including GPUs (mostly Nvidia GPUs) and Intel's Xeon Phi accelerator, making OpenMP 4+ a good candidate for being performance portable, and this is discussed in more detail in Sec. 5.1.

2.5.4 Intel TBB and ArBB

Intel's Threading Building Blocks (TBB) [63] and Array Building Blocks (ArBB) [115] are two libraries designed for high performance, but not necessarily performance portability. ArBB was a C++ data parallelism library that showed promise for being performance portable, but Intel discontinued it in favor of TBB (and Cilk Plus, see Sec. 4.1) shortly after it was introduced, so it was never extensively studied [1]. TBB is another C++ library for task parallelism, but as it only supports accelerators via writing kernels in other languages [174], and does not appear to be making any moves to change this,[5] it is only a task-based runtime system instead of a true performance portable model, so it will not be covered here.

2.5.5 MPI and SHMEM

MPI [109] is the most commonly used model for communication between processors and nodes on supercomputers. It is based on a two-sided message passing model, where both processes need to be involved in sending and receiving messages. MPI is highly portable, and will generally get high performance on any machine without source code changes, making it very performance portable by some standards. However, MPI alone doesn't allow users to take advantage of all a machine's hardware (like accelerators) and has some scalability problems [166], meaning MPI alone is not enough to write performance portable exascale programs.

SHMEM, standardized by OpenSHMEM [128], is a newer communication model, based on the partitioned global address space (PGAS) model, that is gaining popularity. SHMEM uses a one-sided message passing model, where only one process puts (or gets) data into (from) another process's memory space, without interfering with the other process's execution. Like MPI, SHMEM is highly portable, and in some ways performance portable, but it also cannot access all a machine's hardware on its own.

Both MPI and SHMEM need to be used with another programming model to become truly performance portable, but beyond that, both are too low-level to be very productive programming models. Many of the performance portable models below are built on top of MPI

[5]Intel may in fact be choosing to deprecate this in favor of DPC++ (see Sec. 4.6).

or SHMEM, but work at an even higher abstraction level to be productive as well.

3 Libraries

This section will describe library-based programming models. For the purposes of this paper, libraries are considered to be add-ons to existing languages, and not themselves languages or language extensions, like CUDA or OpenACC. Categorizing some of these programming models as one thing or another can be difficult, so in some cases we default to what the authors/developers consider their model to be.

3.1 Overview

There is a plethora of high performance domain specific libraries, many of which are highly scalable and portable to many architectures. This paper will not discuss them, however it will briefly discuss some methods for making these libraries more performance portable. It will also discuss multiple kinds of libraries, and will give a brief overview of classifications for them.

3.1.1 Making Domain Libraries Performance Portable

Domain-specific libraries are very helpful for application developers in that they allow for high levels of code reuse and enable code portability. However, to remain portable and general, a single library can't be optimized for every architecture it might be used on; while domain libraries may be portable, they are often not performance portable.

A potential solution to this is proposed by the Broadway compiler [11, 48]. The Broadway compiler can optimize an application's use of libraries by using information about the library stored in an annotation language. The annotations are created by experts and kept with the library (no source modifications to the library are necessary). While creating the annotations might be time-intensive, the cost of creating and tuning them is amortized over many uses of the library, and the annotations are transparent to users, who only need to switch out their compiler. Broadway reads in those annotations along with the application and library, then performs source-to-source translations on the application's usage of the library, for example removing multiplications by zero or replacing an expensive library call with a more specialized, inexpensive call when conditions described in the annotations are met.

The goal of Broadway is separation of concerns between compiler, library, and application developers; the compiler doesn't need to worry about what's in the library, since it is explicitly told, and the application doesn't need to worry about optimizing the library because it's done by Broadway. Broadway has three general classes of optimizations it can apply: (1) optimizations that always improve performance, (2) optimizations that depend on the machine, and (3) optimizations that depend on the machine *and* the application. Many of the optimizations that fall under classes (2) and (3) affect performance portability. Class (2) optimizations can be turned on more or less mechanically, based on the target architecture, but optimizations from class (3), such as optimizations for parallelism granularity and degree, require more work. To better support these kinds of optimizations, the annotation language was being modified to support dynamic feedback at run time – when there's no clear "best" option for a function at compile time, Broadway will make multiple versions of the function a try each at run time, then pick the best. The annotation language can even support varying how the best version is chosen at run time, further enabling performance portability.

As a proof of concept, the Broadway authors tested their work various applications using the PLAPACK library [2] and saw significant performance improvement on many functions, with modest or small gains on the rest, which demonstrates that even a library that has already been heavily tuned, like PLAPACK, can benefit from some machine- and application-specific optimization.

3.1.2 Library Patterns

The libraries discussed in the rest of this section share many points in common, such as the parallelism patterns and data structures they support. The merged OPL and PLPP projects [69], while never fully finished it appears, contain the vast majority of these patterns and provide a useful framework for discussing them in the rest of this section.

The OPL and PLPP projects (from here on, only OPL) were both pattern languages, which provided high-level interfaces and functions to describe parallelism in applications. The main goal of OPL was to allow users to develop correct and efficient programs that could be written quickly, which the authors of OPL equated to giving users better ways to expose parallelism. The OPL language provided a hierarchy of patterns, where the top levels were structural patterns (how the application was organized) and computational patterns (classes of algorithms) which had no explicit parallelism. Ideally, a user wouldn't need to go beyond these levels, but just in case OPL also provided lower level interfaces which did deal directly with parallel constructs. The lower levels of OPL included parallel algorithm strategies (how to exploit parallelism), implementation strategies (program and data structures that deal with parallelism), and parallel execu-

tion patterns (how parallelism gets mapped to hardware, e.g., SIMD execution, thread pools, etc.). The lowest level is the foundations of parallelism, or how concurrency is actually implemented, e.g., threading models, message passing, and memory coherency models.

OPL included the following patterns (ones particularly relevant to HPC in bold – many of these also show up throughout the rest of this paper, not just in this section):

- Structural patterns:
  - **Pipe and filter – data is passed through a series of stateless operations.**
  - Agent and repository – a central collection of data (e.g., a database) is modified by several independent units of execution.
  - Process control – a running process needs to be monitored, and actions might need to be taken if the process enters certain states (e.g., turning on the heat if a thermostat says a room gets too cold).
  - Event-based – processes watch an environment for events and take some action when events occur.
  - Model-view-controller – an internal data model has interfaces for viewing and modifying it or controlling its behavior (e.g., many web-based applications).
  - **Iterative refinement – run an operation until some termination conditions are met.**
  - **Map-reduce – apply a function to data and aggregate the results.**
  - **Layered systems – different software layers provide interfaces to each other for interaction but must remain independent.**
  - **Puppeteer – a set of processes must interact to work on a problem and need to be managed by a puppet-master.**
  - **Arbitrary task graph – A catch-all for applications which don't quite fit the other structures.**
- Computational patterns (reminiscent of Berkeley's 13 dwarves [5]):
  - **Backtrack/branch and bound – search algorithms for a large search space.**
  - Circuits – boolean operations or logic circuits.
  - Dynamic programming – a more efficient alternative for recursive algorithms.
  - **Dense linear algebra.**
  - **Sparse linear algebra.**
  - Finite state machines.
  - **Graph traversal algorithms.**
  - Graphical models – similar to state machines, but modeling a collection of random variables (e.g., hidden Markov models).
  - **Monte Carlo – a statistical method for estimating unknown values.**
  - **N-body – calculating pair-wise interactions between N entities.**
  - **Spectral methods – changing how data is represented to make sense of it (e.g., Fourier transforms).**
  - **Structured mesh.**
  - **Unstructured mesh.**
- Parallel algorithm strategies:
  - **Task parallelism.**
  - **Pipeline or stream.**
  - Discrete events – tasks interact rarely and interactions are handled as events by an event manager.
  - Speculation – tasks interact rarely and are run speculatively; if errors/conflicts are detected, the tasks at fault are rerun.
  - **Data parallelism.**
  - **Divide and conquer – split a large problem in to progressively smaller subproblems that can be more easily solved and the solutions aggregated.**
  - **Geometric decomposition – each process is responsible for a piece of the problem domain, and data is communicated to neighboring processes as needed.**
- Program implementation strategies:
  - **Single program, multiple data (SPMD).**
  - **Data parallelism over an index space.**
  - **Fork/join.**
  - Actors – units of execution are associated with data objects, and method calls amount to message passing between them.
  - **Loop-level parallelism – loops are structured such that each iteration is independent and can run in parallel.**
  - **Task queue.**
- Data structure implementation strategies:

– Shared queue.
– Shared map.
– **Partitioned graph.**
– **Distributed array.**
– Generic shared data – catch-all for data sharing that doesn't fall into the previous categories.

As can be seen, this is a great deal for any single library to attempt to implement, and in practice, some of these patterns are much more commonly used than others, especially in HPC (for example, meshes and linear algebra are much more common in HPC than dynamic programming problems or finite state machines). Therefore, most performance portability libraries only implement a subset of the most popular patterns, which tend to be those well suited to solving domain science problems in fields that utilize HPC techniques, like physics, chemistry, and biology. These problems mostly fall under the linear algebra, mesh, n-body, and spectral patterns, and the strategies that work well with these patterns. The next section will go into detail on how libraries have implemented these patterns and strategies.

3.2 Libraries for Performance Portability

This section describes several libraries designed specifically to provide performance portability across multiple domains.

3.2.1 Skeletons

Skeleton programming is a concept borrowed from functional programming, where higher-order functions can take other functions as arguments to specialize their operation. Since imperative languages are much more common in HPC, developers have begun implementing them in these languages as well [39]. Skeleton libraries tend to be modeled as arbitrary task graphs that provide various communication and computation patterns (skeletons) as higher-order functions. These skeletons generally implement data parallel patterns, but not loop parallelism, and make use of various distributed data structures.

SkelCL. SkelCL [155] (code sample in Listing 1) is a high level library for programming heterogeneous systems, built on top of OpenCL. The primary design philosophy of SkelCL is to abstract away all the tricky parts of writing (multi-)GPU code to make programming easier. SkelCL provides a modest set of data parallel skeletons for some of the computation patterns above, which users can specialize with their own C-like functions, including `Map` (apply a function to all elements of a collection), `Zip` (apply a function to a pair of collections), `Reduce` (condense the elements of a collection to a single value), and

```
/* Dot product */
/* create skeletons */
SkelCL::Reduce<float> sum (
    "float sum (float x,float y){return
    x+y;}");
SkelCL::Zip<float> mult(
    "float mult(float x,float y){return
    x*y;}");

/* allocate, initialize host arrays */
float *a_ptr = new float[ARRAY_SIZE];
float *b_ptr = new float[ARRAY_SIZE];
fillArray(a_ptr, ARRAY_SIZE);
fillArray(b_ptr, ARRAY_SIZE);

/* create input vectors */
SkelCL::Vector<float> A(a_ptr, ARRAY_SIZE);
SkelCL::Vector<float> B(b_ptr, ARRAY_SIZE);

/* execute skeletons */
SkelCL::Scalar<float> C = sum( mult( A, B
    ) );

/* fetch result */
float c = C.getValue();
```

Listing 1: SkelCL code sample.

`Scan` (prefix-sum). Later works [159, 14] add `MapOverlap` (simple stencil), `Stencil` (more complex stencils), and `AllPairs` (apply a function to each pair of elements in two sets; n-body) skeletons to better support some linear algebra applications. The two different stencil-like skeletons allow users more flexibility in defining their stencil computations – e.g., `MapOverlap` uses padding around the edges of the vector/matrix to minimize branching, while `Stencil` allows users to specify the number of iterations to run the stencil and various synchronization properties. All skeletons can take additional arguments aside from the defaults to give users even more flexibility. For example, if a user wants to define a generic "add x" function for vectors, they can use the map skeleton plus an additional argument for x.

SkelCL also provides a generic vector class that works with both CPU and GPU code and hides data transfers from users – the vector class will do data copies lazily (only when data is actually used) to eliminate unnecessary data transfers. The vector class can also automatically distribute data to multiple GPUs based on a set of predefined data distribution patterns, including `single` (only one GPU gets a copy), `block` (split evenly among all GPUs), and `copy` (each GPU has a full copy) [156]. A later paper [159] adds an `overlap` distribution that automates halo exchanges for stencils, as well as a generic

2-dimensional matrix class that supports the same distributions along the rows of the matrix.

All skeletons support all data distributions, and the GPUs automatically cooperate on `block` and `overlap` distributed data structures. If the user doesn't specify a data distribution with multiple GPUs, each skeleton has preferred distributions for their pre-defined arguments, but the user or runtime can override this (for additional, user-defined arguments, the user must always specify a distribution since the runtime can't reason about the semantics of user-defined arguments).

- *Portability:* Since SkelCL is built on top of OpenCL, it supports the many architectures that OpenCL does, but is also prone to the same performance variations as OpenCL across architectures. The LIFT project (see Sec. 7.3.3) is the spiritual successor to SkelCL (written by some of the same authors), which attempts to remedy these problems.

- *Performance:* Across several performance tests on various mini-apps (utilizing all their skeletons), SkelCL consistently had only 3-5% overhead compared to optimized OpenCL (and was sometimes faster than naive OpenCL), and trailed behind CUDA by the same amount as OpenCL (usually around 20-40% on Nvidia GPUs) [155, 156, 159]. SkelCL also appears to scale well across multiple GPUs, getting only slightly less than linear speedup on a Gaussian blur stencil operation (1.9x for 2 GPUs, 2.66x for 3 GPUs, and 3.34x for 4 GPUs in one experiment) [14].

- *Productivity:* While SkelCL code is generally *much* shorter and easier to read than OpenCL or CUDA code (due to removing the extensive boilerplate that OpenCL is known for) [156, 159, 14], SkelCL is still subject to some of the productivity problems of OpenCL. OpenCL takes in kernels as strings and compiles them as needed at run time, and SkelCL does the same, which makes debugging kernels very difficult, since errors are never caught at compile time.

- *P3 Ranking:* fair – has all the same performance portability problems as OpenCL, and a few of the same productivity problems, but is much higher-level.

SkePU. Not to be confused with SkelCL, SkePU [38, 41] (code sample in Listing 2) is another skeleton library based on C++, with back ends for CUDA, OpenCL, and OpenMP. Similarly to SkelCL, SkePU provides data parallel computation skeletons for Map, Reduce, MapReduce, MapArray (similar to zip), and MapOverlap (stencil), which users could customize with their own functions. Scan and Generate (initialization) skeletons were added later [96]. All of these skeletons can support multi-GPU execution, and even multi-node execution, via an MPI layer to manage SkePU instances on multiple nodes [96]. SkePU also supports passing additional arguments to each skeleton via a user-defined struct, like SkelCL.

```
// Vector sum
// define kernel
template <typename T>
T add(T a, T b) {
    return a + b;
}

// create vectors
skepu2::Vector <float> v1(N), v2(N),
    res(N);

// init skeleton
auto vec_sum = skepu2::Map<2>(add <float>);

vec_sum(res, v1, v2);
```

Listing 2: SkePU code sample.

Early implementations of SkePU [38, 96] had users define their functions using macros, which would be expanded into structs containing versions of the function for each of the three back ends, but SkePU 2 [41] updated this to use C++ templates and a source-to-source translator (based on Clang) to create the struct of function variants. This new version removes all the type-safety problems of the original SkePU, as well as improving SkePU's flexibility and the compiler's ability to optimize SkePU code, and giving the users the option to define functions as lambdas or template functions, instead of based on pre-defined macros. The authors intend to extend their source-to-source compiler to let users define different specialized versions of their functions, and automatically choose between them at compile time based on the target architecture. SkePU 2 also added the Call skeleton (which simply calls a user function) for even more flexibility and changed the semantics of the Map skeleton to handle cases which had previously been handled by MapArray.

Like SkelCL, SkePU provides generic vector and matrix objects (which they call "smart containers" [26]) that hide data transfers from users, manage memory consistency with an MSI-like model, and do data movement between GPUs and nodes lazily to minimize transfers [38]. Even for non-GPU back ends, this is useful for decreasing communication costs. These objects also handle data distribution to multiple GPUs and nodes and can do auto-

mated halo exchanges, although unfortunately users have no control over how data is distributed, unlike SkelCL [156] – data is always divided equally among GPUs/nodes. Their most recent implementation [26] also has support for using Nvidia's GPUDirect (an RDMA library) to improve performance, support for partial copies of arrays and matrices, and an improved consistency model that can handle overlapping ranges on multiple devices/nodes correctly. When integrated with the StarPU task runtime [6], these containers can even help the runtime determine data dependencies and ensure asynchronous tasks are launched and completed in a correct order. The SkePU developers believe smart containers and other memory abstractions are going to become more important for performance portability, since they allow the same code to run on systems with and without separate (device) memories; see Sec. 7.2 for further discussion on containers.

- *Portability:* SkePU supports back ends for CUDA, OpenCL, and OpenMP, which allows it to target a wide variety of CPUs and GPUs [38]. In addition, SkePU's multi-node capabilities allow the same code to run on a laptop or on a cluster without modification, which is extremely helpful for debugging and boosting productivity [96].
- *Performance:* Initial performance tests [38] on GPUs showed that on some kernels, SkePU matched the performance of a cuBLAS implementation, although hand written CUDA was slightly faster on other kernels because of implementation differences. Scalability tests across multiple nodes and multiple GPUs show that SkePU scales well for various combinations of OpenMP processes and numbers of GPUs, and also demonstrate the performance benefits of using their lazy memory transfer mechanism [96]. After implementing their new containers and memory management model, the benefits of this increased a further 20x, on average [26]. Preliminary performance tests of SkePU 2 show it surpassing SkePU 1.2 on several tests already, even though it hasn't been fully optimized. The results for SkePU 2 do seem to vary by base compiler though, which demonstrates a need for more mature optimizing C++11 compilers [41] (see Sec. 7 for more about this).
- *Productivity:* One of the goals of SkePU is to make parallel code appear sequential by hiding parallelism, data transfers, and synchronization inside the library, which can greatly increase usability and productivity, since developers don't need to learn or worry about parallelism or heterogeneous programming [96]. In addition, with SkePU 2, using a normal C++ compiler instead of their source-to-source compiler will give valid sequential code, which is useful for debugging and testing [41].

  The SkePU 2 developers did a further usability survey [41] on their new template interface compared to the original macro interface with a group of graduate students. They introduced half the students to SkePU 1 first, then to SkePU 2, and reversed this for the other half, then asked the students which version's code was clearer on a simple example and on a complex example. For the simple example, students sometimes preferred SkePU 1, especially when they weren't familiar with C++11 features (this led the SkePU developers to reconsider some design choices). For the complex example, students generally preferred SkePU 2, which the developers believe is because the reduced macros and user function declarations make the code easier to read.
- *P3 Ranking*: good – can avoid OpenCL's performance portability problems by using OpenMP or CUDA, and multi-node capabilities are good. Productivity is also good, as long as the developer is good with C++.

```
//Cannon's matrix multiplication algorithm
template <typename T>
DMatrix<T>& matmult(DMatrix<T>& A,
    DMatrix<T>& B, DMatrix<T>* C) {
     //initial shift
     A.rotateRows(&negate);
     B.rotateCols(&negate);

     for (int i = 0; i <
    A.getBlocksInRow(); i++) {
         DotProduct<T> dp(A, B);

         //submatrix multiplication
         C->mapIndexInPlace(dp);

         //stepwise shift
         A.rotateRows(-1);
         B.rotateCols(-1);
     }
     return *C;
}
```

Listing 3: Muesli code sample.

Muesli. Muesli [22, 39, 40] (code sample in Listing 3) is another C++ template library for skeleton programming. Unlike the other skeleton libraries listed here, Muesli was originally designed for multi-node systems, and has focused on multi-node scalability, only adding multi-core and GPU support recently. Muesli has back ends for

MPI, OpenMP, CUDA, and combinations thereof. Muesli also has a Java implementation utilizing MPJ (MPI for Java), Aparapi (a GPU programming interface), and Java OpenCL, although support for GPUs and Xeon Phis in Java is not as mature.

Muesli provides both task parallel and data parallel skeletons, with the data parallel skeletons provided as member functions of their distributed data structures for vectors, dense matrices, and sparse matrices. Developers can use the task parallel skeletons, including `Pipe`, `Farm`, `DivideAndConquer`, and `BranchAndBound`, to define their desired process organization as a task graph. The data parallel skeletons can be used for computation and communication between processes, and include `fold`, `map`, `scan`, `zip`, and some variants of these. All these skeletons can be arbitrarily nested to build a scalable program. The developers have been careful to make each skeleton highly scalable for each back end – see their description of implementing the `Farm` skeleton [39] for an example.

Users can pass functions to these skeletons as either normal functions or as functors (currently, the GPU back end requires functors [40]). Another unique feature of Muesli is support for *currying* user (and skeleton) functions, which allows users to create function objects that have been partially applied to their arguments, and apply it to the remaining arguments later. This can help encourage code reuse and modular programming. Unfortunately, currying isn't supported with the CUDA back end, but users can achieve similar functionality by modifying member variables in functors.

- *Portability:* Muesli can target most CPUs, Nvidia GPUs, and Intel Xeon Phis with C++ and many CPUs and GPUs with Java. Xeon Phis are only supported in "native" exectuion mode, not offload mode. Additionally, GPU performance with Java often suffers because the underlying GPU offload library Muesli uses doesn't use the correct data layout to enable coalesced memory accesses.

- *Performance:* Performance tests [21, 39] show that all of Muesli's skeletons scale well on multi-node, multi-core system, and scalability is very similar to handwritten MPI+OpenMP. The combination of task and data skeletons also allows users to experiment with task vs. data parallel implementations to handle load balancing and other concerns. Comparing the Java and C++ implementations shows that Java performance is generally similar to C++, except on some GPU benchmarks, but this is likely due to the problems with Java's GPU library mentioned earlier.

- *Productivity:* Muesli's skeletons abstract away all threading and communication, so users don't need to worry about these low-level details. Furthermore, the developers noted that when turning sequential code into part of a parallel `Pipe`, only minor changes have to be made to the sequential code (mostly, placing it into a functor), so porting sequential code to Muesli is fairly low-effort [39]. However, lambda support could make this better, since functors are known for being verbose and difficult to understand for those not used to C++.

  When adding OpenMP support, the Muesli developers also put effort into ensuring Muesli scales *down* as well as up – they wanted Muesli code written for multi-core to be able to run on a laptop or cluster without OpenMP as well as the other way around. To do this, they added an OpenMP abstraction layer inside the library so that it can be compiled with or without OpenMP support, and users can run code that would normally be multi-threaded in a single-threaded mode, for easy testing and debugging.

- *P3 Ranking:* good (C++), fair (Java) – C++ is quite performance portable, Java is less so because of poor library support. Both are quite productive, it appears.

```
// Producer-consumer pattern (Collatz
   conjecture)
pipeline(parallel_execution_thr{}
    // Consumer function
    [&]() -> optional<int> {
        auto value = read_value(instream);
        return (value > 0) ? value : {};
    },
    stream_iteration(
        // Kernel functions
        pipeline(
            [](int e){ return 3*e; },
            [](int e){ return e-1; }
        ),
        // Termination function
        [](int e){ return e < 100; },
        // Output guard function
        [](int e){ return e%2 == 0; },
    ),
    // Producer function
    [&](int e){ outstream << e << endl; }
);
```

Listing 4: GrPPI code sample.

GrPPI. GrPPI [31, 30] (code sample in Listing 4) is somewhat different from the other skeleton libraries described previously. While those libraries describe data

parallel skeletons, GrPPI describes skeletons for stream processing, which falls under the pipeline pattern from above. Stream processing works under the assumption that the full sequence of data is *not* available, and is in fact constantly arriving and needs to be processed as it arrives. This model of data as a (possibly infinite) sequence or stream has become more relevant as IoT and big data have grown more prominent, but is also very applicable to HPC programming problems, such as condensing large amounts of data from a simulation so it will fit on disk or performing in situ analysis. GrPPI is built on C++ templates, and has multiple back ends, including C++ threads, OpenMP, and Intel's TBB.

GrPPI provides skeletons for working on streams of data, including Pipeline (run a series of operations on each element in the stream), Farm (similar to map, run a single function in parallel on all elements of the stream), Filter (run a test on all items in parallel, only items that pass are put in the output stream), and Stream-Reduce (like normal reduce; run a binary function on all items in the stream to create a single aggregate result, e.g., a running average) [31]. A later paper [30] adds further skeletons for manipulating the flow of streams to give users more flexibility and control. The new stream skeletons include Split-Join (divides a stream into sub-streams, either duplicating the input stream or putting elements into sub-streams round-robin, and later recombines the sub-streams), Window (groups elements, based on how much some value has changed, how much time has passed, how many elements have been seen, or whether some specific element, e.g., a period in a text stream, has been seen), Stream-Pool (for modeling evolution of a population), Windowed-Farm (combination of Window and Farm, passes groups of elements to a farm operation), and Stream-Iterator (apply a function to an item until some condition is met, a rare example of iterative refinement from the list above). All of these skeletons, old and new, are fully composable, allowing users to build a full application out of only skeleton calls.

– *Portability:* While the current back ends (C++ threads, OpenMP, and Intel TBB) allow users to target most CPUs, and test different models to see which works best for their case, GrPPI currently does not support GPUs. The authors note that they wish to add support for GPUs in the future [31].

– *Performance:* The GrPPI developers ran preliminary performance tests on edge detection and synthetic benchmarks to test each of their skeletons against code written natively in each back end [31]. GrPPI performance generally matched C++ threads, OpenMP, and TBB well, although TBB (with our without GrPPI) performed worse than C++ threads or OpenMP, likely because TBB uses an internal locking mechanism the other two back ends lack. A later performance study [30] confirmed these results, including TBB's low performance with or without GrPPI, demonstrating that GrPPI adds negligible overhead.

– *Productivity:* When trying out various parallelization schemes for a video processing application, the authors counted how many lines of code each programming model (C++ threads, OpenMP, Intel TBB, and GrPPI) added to the sequential version. They found that, even for the most complex parallelization scheme, GrPPI added far fewer lines of code – <5%, compared to 26% for TBB, which was second [31]. GrPPI has the added benefit of supporting multiple back ends without tying users to any single one. A later study [30] confirmed these findings on other applications, and added that the cyclomatic complexity[6] of GrPPI code is much lower than its back ends, with the occasional exception of TBB. The authors of GrPPI, however, say they find GrPPI code more structured and readable than TBB regardless.

– *P3 Ranking:* poor, but promising – GrPPI is very productive, but doesn't yet support enough platforms or get high enough performance to be called performance portable. However, with more work it could be very performance portable.

3.2.2 Parallel Loop Libraries

This section will discuss libraries based on parallel loop abstractions. These mostly fall under the data parallelism and index space abstraction strategies, and many use C++ templating to provide a more generic interface. In general, these libraries do not provide computational patterns, and users must write all computation themselves.

Kokkos. Kokkos [36, 37] is a loop-based data parallelism library for performance portability that uses C++ templating capabilities. Kokkos has back ends for various other programming models, including CUDA, OpenMP, and pthreads, which allow users to target a wide variety of architectures, such as Intel's Xeon Phi and Nvidia GPUs, without modifying their code. These back ends can be swapped out with minimal changes to application code so users can easily run on multiple architectures.

Kokkos abstracts away parallelism so architecture-specific optimizations exist outside the main application. Kokkos' abstractions can be loosely grouped into memory and execution abstractions. Memory layout and execution environment are the most common program ele-

[6] This is a metric for code complexity based on the number of independent paths through a program [101].

```
// Heat conduction stencil (TeaLeaf)
#define DEVICE Kokkos::OpenMP

// initialize a view
Kokkos::View<double*, DEVICE> p("p", x*y);

// lambda-style loop
Kokkos::parallel_for(x*y, KOKKOS_LAMBDA
    (int index) {
     const int kk = index;
     const int jj = index / x;

     // if in view interior...
     if (kk >= pad & kk < x - pad &&
         jj >= pad && jj < y - pad) {

         // recalculate value
         p(index) = beta*p(index) +
    r(index);
     }
});
```

Listing 5: Kokkos code sample.

ments that need to change for each new architecture, so the Kokkos developers chose to abstract them away to decrease porting effort.

Kokkos' abstractions are multidimensional array views (array layouts) [35], memory spaces, and execution spaces. The array views allow developers to change the memory layout and access patterns to best suit the underlying architecture at compile time, so the compiler can make inlining and vectorizing optimizations. Memory spaces keep track of where data resides in heterogeneous architectures, and similarly, execution spaces contain information on the type of hardware code should run on. Execution spaces only have access to memory spaces that make sense; e.g., a CPU execution space cannot access a GPU-only memory space, but can access a host burst buffer memory space. All of these can be modified to add optimizations for a particular application on a specific architecture. For example, users could write their own tiled matrix layout and add it to an application by merely swapping out array views. Kokkos keeps these configurations consolidated and out of application code, which is excellent for portability and productivity.

– *Portability:* Kokkos supports back ends for CUDA, HPX, OpenMP 3, pthreads, and serial execution, with back ends for OpenMP+offloading (4+) and AMD's HIP and ROCm in progress [75]. The current back ends allow Kokkos to target a wide array of devices, including Nvidia GPUs, Intel's Xeon Phi, and many CPUs. The new back ends will allow Kokkos to target AMD GPUs as well.

– *Performance:* The Kokkos developers tested their CUDA and OpenMP back ends against native/hand-written CUDA and OpenMP mini-applications on an Intel CPU, Intel Xeon Phi, and Nvidia GPU [37]. While they did need to tweak some settings in Kokkos (particularly for CUDA), Kokkos' performance was comparable to native OpenMP and CUDA in most cases, even outperforming native CUDA in one case. Kokkos also scaled similarly to native OpenMP across multiple CPUs and Xeon Phis. These experiments demonstrate that Kokkos has low overhead and is at least as performance portable as the back ends it is built on.

  A later performance portability study [100] confirms these results, showing Kokkos having low overhead versus native OpenMP on Intel CPUs, although some tuning was needed to get the same performance on IBM's Power8 CPUs and the Xeon Phi, which was very new when the first performance tests were done, and likely had better compilers by the time of the second test. The more recent study showed slightly higher overheads on an Nvidia GPU, but still comparable to native CUDA.

  A third study [133] shows Kokkos can have performance comparable to OpenCL on Intel and AMD CPUs for some benchmarks, although Kokkos struggled on most of the benchmarks. On Nvidia GPUs though, Kokkos' performance is comparable to CUDA and OpenCL on all benchmarks.

– *Productivity:* Kokkos was designed to be added incrementally to large applications (100,000+ lines of code) to aid the porting process. Developers can add Kokkos abstractions one loop at a time and check that their results are still correct before moving on to the next loop. This makes it less of a monumental effort to port such large applications, and since Kokkos allows users to swap out back ends at will, porting becomes worth the effort since future optimizations and ports to new architectures will happen inside Kokkos, not the application code.

  However, a productivity study including Kokkos [100] have note that the functor programming pattern Kokkos uses for kernels is difficult to understand and very verbose. The increasing number of compilers that implement C++ lambdas greatly alleviates this problem, since it allows kernels to be written inline instead of in a separate functor. The authors of this study still note, though, that Kokkos' array views aren't terribly intuitive, and the Kokkos API could be improved by simplifying them.

```
// Heat conduction stencil (TeaLeaf again)
// set up execution policty
typedef RAJA::IndexSet::ExecPolicy<
    RAJA::seq_segit, // sequential on
    segments
    RAJA::omp_parallel_for_exec // OpenMP
    parallel for
> policy;

// Custom box segment for index set
RAJA::BoxSegment box (/*...*/);
RAJA::IndexSet inner_domain_list;
inner_domain_list.push_back(box);

// lambda-style loop
RAJA::forall<policy>(inner_domain_list,
    [=] RAJA_DEVICE (int index) {
    p[index] = beta*p[index] + r[index];
});
```

Listing 6: RAJA code sample.

– *P3 Rating:* very good – the selection of back ends and Kokkos' performance with many of them make it very performance portable. However, its productivity could perhaps be better.

RAJA. RAJA [61, 9] (code sample in Listing 6) is very similar to Kokkos in that it is a loop-based, data parallel C++ template library with multiple, interchangeable back ends, including CUDA and OpenMP. RAJA also abstracts away parallelism (via memory and execution patterns) with the goal of hiding architecture-specific code inside the library where users won't have to interact with it.

RAJA's chosen abstractions are execution policies, iteration spaces, and execution templates. The execution policy defines where a kernel will run and which back end it will use for parallelism. Iteration spaces describe which indices of an object the loop will access, and are made up of index sets and segments. Segments contain indices that can be treated as a single unit, and index sets are sets of arbitrary segments. These index sets can contain non-contiguous indices, as can segments, which makes them good for representing sparse data structures. RAJA was originally designed to work well with both structured (regular) and unstructured (irregular) mesh-based codes, which is why this particular design was used. Execution templates define the type of operation being done in the kernel, such as a standard parallel loop (`forall`) and reductions. Similarly to Kokkos, all of these can be modified to suit an application's needs. For example, when Martineau et al. were porting TeaLeaf to RAJA [100], they implemented a new index set type that corresponded to a 2-d section of an array so they could use tiled array accesses, which improved TeaLeaf's performance on several architectures.

– *Portability:* RAJA has full support for serial, OpenMP 3, CUDA, and "loop" back ends, with "SIMD," OpenMP+offloading (4+), and Intel TBB back ends in the works [9]. The "loop" back end allows the compiler to do its own optimizations without interference, and the "SIMD" back end forces vectorization. These back ends allow RAJA to target most CPUs, Xeon Phis, and Nvidia GPUs.

– *Performance:* Early performance studies of RAJA versus pure C+OpenMP on a proxy application showed that RAJA+OpenMP scales better than pure C, but has higher overheads with low thread counts, likely due to higher OpenMP overheads inside RAJA rather than any problem with RAJA itself.

  A later study [100] demonstrated that RAJA has very low overheads and performs almost as well as native OpenMP 3 for Intel CPUs and Power8 CPUs, although more work is needed to reduce overhead on an Nvidia GPU. RAJA also needs some index space/memory access pattern tweaks to get better performance on most platforms.

– *Productivity:* Again, like Kokkos, RAJA was designed for incremental porting, and has the same benefits as Kokkos. Similarly, better lambda support has helped RAJA move away from using functors for kernels and improved productivity, although even early on the developers had feedback from domain scientists that the RAJA version of their code was more readable and easier to maintain (in addition to running faster because it was now parallel) [61]. In addition, one study [100] prefers RAJA's loop index spaces to Kokkos' array views, finding them more intuitive. This study also found that porting a code to RAJA adds fewer lines of code than porting to Kokkos.

– *P3 Ranking:* good, and promising – RAJA could support more platforms, but it gets good performance on the platforms it does support, making it a promising candidate for being very performance portable.

```
// triad, functor version
phast::vector<double> a(n);
phast::vector<double> b(n, 1.0);
phast::vector<double> c(n, 2.0);
// computation functor
```

```
template <typename T, unsigned int policy
    = phast::get_default_policy()>
struct triad_functor :
    phast::functor::func_scal_scal_scal<T,
    policy>
{
    T scal;
    _PHAST_METHOD void operator()(
        phast::functor::scalar<T>& a,
        const phast::functor::scalar<T>& b,
        const phast::functor::scalar<T>& c)
    {
        a = b + scal * c;
    }
};
// instantiate functor
triad_functor<double> triad;
triad.scal = 3.0;
// run triad calculation
phast::transform(a.begin(), a.end(),
    b.begin(), c.begin(), triad);
```

Listing 7: PHAST code sample.

PHAST. The PHAST library [132, 133] (code sample in Listing 7) is based heavily on C++11 features, which let users write parallel code in terms of containers, iterators, and STL-like algorithms – these algorithms are data parallel and similar to skeletons, but PHAST's focus is parallel loops, which is why it's in this section. PHAST code looks like normal C++ with library calls. PHAST has back ends for CUDA, Boost.Threads, and C++11 threads, so it can target Nvidia GPUs and most CPUs. The main design goal of PHAST was to hide architecture and language details from users and manage the degree and type of parallelism, so the same code can run well on multiple architectures, while still allowing users to tune whatever they wish. PHAST allows users to set all the parameters for every platform at once and will only use the parameters relevant to the current platform. These parameters are kept separate from user code, so parallelization and data mappings can be changed without modifying application code. When no parameter values are specified, PHAST tries to infer good values.

To do this, PHAST provides STL-like parallel algorithms, including for_each (the primary algorithm), copy, and find, some of which take user-defined functors to specialize the algorithm. While PHAST tries to encourage writing platform- and architecture-independent code, it is still possible for users to further optimize their functors by adding different code versions with `#ifdef`s. PHAST provides four containers that all of these algorithms can work with: vector, matrix, cube (a generic 3-d data structure, not strictly a cube), and grid, which can be defined over another container to create a mesh. PHAST also automatically maps standard types to vector types (when appropriate) and overloads their operations with vector operations; e.g., a series of `uint4_t`s will be mapped to a `__m128i` on SSE architectures. Users don't have to work with vector types, PHAST does this transparently with no user interaction necessary.

- *Portability:* PHAST supports back ends for CUDA and C++ threads, which allows it to target Nvidia GPUs and most CPUs. The developers intend to add more back ends to allow targeting a wider variety of devices.

- *Performance:* An early performance test of PHAST [132] compared it to CUDA and OpenCL on multicore CPUs and Nvidia GPUs. For the CUDA back end, they noticed that each GPU they tested with needed different parameters to get the best performance, which reinforces the necessity of keeping parameter choices out of application code. PHAST actually outperformed CUDA on one benchmark, but that was due to major implementation differences. When testing against two different OpenCL vendor implementations (Intel and AMD), they noticed that each implementation performed very differently, despite running the same code on the same hardware. They note that this is likely due to Intel providing better auto-vectorization.

  Another performance test [133] versus SYCL, Kokkos, CUDA, and OpenCL revealed that on multicore, all models except SYCL get similar performance on simple benchmarks, while on a full application SYCL and OpenCL perform the best, although this is due to SYCL and OpenCL's ability to reorder data structures. PHAST intends to add this capability in the future. On GPU tests, PHAST is on par with CUDA and OpenCL; SYCL does much worse but this is likely because its implementation for Nvidia architectures is very new.

- *Productivity:* When PHAST was first introduced, the developers noted that it makes for much cleaner code than CUDA while still giving users a similar level of control [132]. A later productivity study [133] calculated several complexity metrics over PHAST, SYCL, Kokkos, CUDA, and OpenCL on several benchmarks and confirmed that PHAST is much simpler than OpenCL and CUDA. PHAST scores similarly to Kokkos and SYCL on these metrics; it is more concise, but arguably a bit more complex. The PHAST developers intend to add support for lambdas and the parallel STL extensions from C++17 to further improve productivity and code simplicity.

```
__targetConst__ double a;
__targetEntry__ void scale(double* field) {
    int baseIndex;
    __targetTLP__(baseIndex, N) {
        int iDim, vecIndex;
        for (iDim = 0; iDim < 3; iDim++) {
            __targetILP__(vecIndex)

    field[INDEX(iDim,baseIndex+vecIndex)] =
    a*field[INDEX(iDim,baseIndex+vecIndex)];

        }
    }
    return;
}

targetMalloc((void **) &t_field, datasize);

copyToTarget(t_field, field, datasize);
copyConstToTarget(&t_a, &a,
    sizeof(double));

scale __targetLaunch__(N) (t_field);
targetSynchronize();

copyFromTarget(field, t_field, datasize);
targetFree(t_field);
```

Listing 8: targetDP code sample.

In addition, PHAST can be used as a source-to-source translator if desired. PHAST can produce code that can be given to `nvcc` or other standard compilers; a skilled programmer could use PHAST to generate a baseline implementation and then optimize from there.

– *P3 Ranking:* fair, but promising – for being so new, PHAST is doing well with performance, although it could support more architectures. It is also very productive, for developers who know C++.

targetDP. Unlike the other libraries listed here, targetDP [44, 45] (code sample in Listing 8) uses pure C (not C++) with preprocessor macros and library functions to provide data parallel loops. targetDP's macros and library are designed to be maintainable and extensible; they were originally designed for structured mesh codes, and work best with regular codes, but are applicable to other codes as well. The motivation behind targetDP was to provide a single-source implementation for mesh-based applications that could target both CPUs and GPUs. targetDP was developed in conjunction with the Ludwig lattice application, which has several loop nests with small inner loops that are difficult to vectorize – they wanted a solution that could (correctly) force vectorization.

targetDP targets OpenMP for CPUs and Xeon Phis (only supported in "native" execution mode), and CUDA for Nvidia GPUs. To provide parallelism, targetDP defines the macros `TARGET_ILP` and `TARGET_TLP` for instruction-level and thread-level parallelism, respectively, that expand to OpenMP `simd` and `parallel loop` constructs or CUDA kernel invocations. These macros allow users to control vector length and parallelism degree, and can help force vectorization for loops that may not otherwise be vectorized. They also provide a macro for abstracting away memory layout, similar to index macros from other scientific codes that treat a single-dimensional vector as a multi-dimensional array. This allows users to change memory layouts without modifying their code.

Even when targeting CPUs, targetDP maintains a host vs. device distinction, since this makes implementing the macros somewhat simpler, removes some memory consistency and data race concerns, and doesn't add much overhead in their experience (though decreasing memory usage is a possible future improvement). For example, their `copyToTarget` macro will expand to `cudaMemcpy` when targeting CUDA or a plain `memcpy` when targeting OpenMP.

– *Portability:* Targeting OpenMP and CUDA allows targetDP to target many CPUs, Intel Xeon Phis (although they current only use "native" execution, which doesn't require using OpenMP's offload model), and Nvidia GPUs.

– *Performance:* Performance tests [44] showed that targetDP improved application performance on both an Intel CPU and an Nvidia GPU, likely due to better vectorization, which does make applications more performance portable. Vectorization also improved performance on a Xeon Phi [45]. However, the authors never compare their performance to hand-written CUDA or OpenMP on any architecture, so it's difficult to say whether the performance is actually good.

– *Productivity:* While targetDP's macros do allow users to only write kernel code once, they must still manually manage memory transfers between the host and "device," which is challenging and time consuming, and one of the major criticisms of CUDA and even other performance portable models like OpenMP and OpenACC. C preprocessor macros are also not type-safe, which may lead to errors; SkePU 2 [41] deliberately moved away from macros to alleviate this problem. However, if application developers are unwilling or unable to migrate from pure C to

C++, they also need a solution, and targetDP is one option.

– *P3 Ranking:* poor – doesn't support many models and performance hasn't been compared to any other models, but if an application needs C, it's an option.

3.3 Application-Specific Libraries

In a recent paper, Holmen et al. [59] suggest a way for large legacy codes to incrementally adopt performance portability models by adding a dedicated, application-specific *performance portability layer* (PPL), which acts as an application-specific performance portability library. The primary goal of a PPL is to improve long-term portability for legacy code, or even a newer application that is expected to have a long lifetime. A PPL insulates users from changes in the underlying programming models by consolidating all performance portability-related code into a single place – the application only needs to port to the PPL once, and all future ports take place inside the PPL. This allows applications to experiment with multiple programming models without disrupting users and domain developers. If the user base for the application's chosen model(s) dies out (as the user bases for some early programming models did [52, 181]), the application can migrate to new models in the PPL, not in the application code. The PPL also allows an application to specialize its use of performance portable models.

Holmen et al. describe the adoption of a PPL into the Uintah fluid-structure interaction simulator. The Uintah PPL is based heavily on Kokkos and contains parallel loop abstractions similar to Kokkos' and RAJA's (including an iteration range, execution policy, and lambda kernel function), as well as application-level tags that denote which loops support which back ends (e.g., CUDA vs. OpenMP), and build-level support for selective compilation of loops that enables incremental refactoring and even simultaneous use of multiple underlying models. Currently, their PPL only supports Kokkos, but the authors see no reason why it couldn't also support RAJA.

While their PPL is very similar to Kokkos, the developers didn't want to directly adopt Kokkos because they wanted to preserve legacy code, simplify the abstractions for their domain developers, and make re-working their implementation or adopting another performance portable model later easier. They've succeeded at some of these goals already, since their domain developers have liked the new parallel loop interface and haven't had much trouble with it, even though they don't know much about parallel programming. The new portable implementation also demonstrates better speedup and scalability on both CPU and GPU architectures.

Since the Uintah code base is so large, the authors did several small-scale refactors to validate their PPL and standardize an adoption process, and came up with general advice for porting to performance portable models:

– Put a build configuration system in place before fully migrating to a PPL to avoid additional refactors.

– Include a tagging system for loops to let the build system know which loops have PPL implementations for which interfaces; this will make porting go much smoother.

– Have a thorough testing apparatus in place to verify correctness pre-, during, and post-port.

The Uintah developers also acknowledge that, while adding a PPL solves many problems, it also raises new questions, including how to use domain libraries (e.g., PETSc) in both the application and PPL simultaneously, how to manage increasingly complex build configurations, how to make intelligent, optimal use of underlying programming models, and how to efficiently manage memory and execution while potentially using multiple underlying programming models. In some ways, adding a PPL is moving responsibility for performance portability back onto applications, where it was originally (see Sec. 2.2.4), which raises many more questions about the roles of applications, compilers, and programming models in performance portability.

4 Parallel Languages

In an ideal world, sequential languages could automatically be parallelized to get high performance on modern machines, but this is an exceptionally difficult problem, and debatably not one worth solving,[7] so numerous explicitly parallel languages have been created instead. These languages vary widely in their feature sets and parallel constructs, from high-level languages like Chapel to low-level languages like OpenCL. The barrier to entry for new languages is higher than for all the other models discussed in this paper, since developers of new applications are reluctant to use a language that might not exist in a few years, and incremental porting to a new language is difficult if not impossible [177]. This section describes the main languages that have managed to break into HPC, as well as some new contenders that show promise.

```
// quicksort
template <typename T>
void qsort(T begin, T end) {
    if (begin != end) {
        T middle = partition(begin, end,
   bind2nd(
```

[7] As some have noted [17], sequential and parallel programming are fundamentally different, so why should we try to use the same tools for both?

```
            less<typename
    iterator_traits<T>::
            value_type>(),*begin));
        cilk_spawn qsort(begin, middle);
        qsort(max(begin + 1, middle), end);
        cilk_sync;
    }
}
// Simple test code:
int main() {
    int n = 100;
    double a[n];
    cilk_for (int i=0; i<n; ++i) {
        a[i] = sin((double) i);
    }
    qsort(a, a + n);
    copy(a, a + n,
    ostream_iterator<double>(cout, "\n"));
    return 0;
}
```

Listing 9: Cilk code sample.

4.1 Cilk

The Cilk language [13, 43] (code sample in Listing 9) started as an MIT project extending C for parallelism, spun off into a startup that created Cilk++ to extend C++ [87], was acquired by Intel and upgraded to Cilk Plus (which worked with C and C++) [144], and has now been deprecated by Intel, but picked up by an open source project called Open Cilk [145]. Cilk/Cilk++ is a task-based language that is unique in that its work-stealing task scheduler provides provable performance guarantees. In general, the philosophy of Cilk has been to allow users to define what *can* be run in parallel, not what *should* be run in parallel; the latter can lead to resource contention. The scheduler decides the parallelism granularity and when/where tasks should be run without user intervention.

Cilk/Cilk++ is built on a source-to-source translator to C/C++ and Cilk runtime calls, which support intra-node parallelism, but leave inter-node parallelism to other models like MPI. Cilk/Cilk++ is a faithful extension of C/C++; the serial elision (the program with all Cilk keywords removed) of any Cilk/Cilk++ program is valid serial C/C++.

The original language [13] used explicit continuation passing to perform synchronization and pass values between threads, but this was quickly abandoned in favor of more traditional spawn and join semantics, since it was too convoluted for most developers [43]. The updated version of Cilk still used the same provably efficient, work-stealing scheduler, however, which is based on the "work-first" philosophy of minimizing overhead by moving it onto the critical path instead of places it would increase the theoretical serial execution time.

Cilk [43] used the `spawn` keyword to create new asynchronous child functions, and the `sync` keyword as a local barrier to force the parent function to wait until all its children had completed (all functions implicitly synced before returning). When a child function returned, its return value was usually stored into a variable in the parent. If a user wanted to do something more complex with return values, like a reduction, they could pass the value to an "inlet" function, which is a function internal to the parent that could perform some operations on the value.[8] If a program did speculative work (e.g., a parallel binary tree search), and discovered it needed to cancel child functions, the parent could call `abort` in an inlet and all its children would return.

Cilk++ [87] used the same scheduler and very similar semantics, but changed the keyword to `cilk_spawn` and `cilk_sync`, added easier parallel loops with `cilk_for`, and removed inlets and abort, replacing them with reducer objects. Reducers provide a non-locking mechanism to perform reductions and other operations by allowing each child thread to have its own local view of an object, and on return combine these local views in a sensible (potentially user-defined) way. Cilk++ also added support for exceptions.

Cilk Plus [144] uses the same semantics as Cilk++ (`cilk_spawn`, `cilk_sync`, `cilk_for`, reducer objects), but adds support for vector parallelism (ILP) with `simd` directives and array notation similar to Fortran's. While array notation may be more familiar to Fortran developers, Robison [144] notes that the `simd` directive is more flexible and portable between compilers. Cilk Plus also adds a tool to check for race conditions; if a program is free of race conditions, it's equivalent to its serial elision.

Open Cilk [145] is fully compatible with Cilk Plus and built on LLVM and Tapir (see Sec. 7.3). Currently, Open Cilk uses the Cilk Plus runtime built by Intel, but there are plans to write a fully open source runtime. The Open Cilk group intends to continue innovating and adding new features to Cilk.

- *Portability:* Cilk is based on C and C++, and as such it is highly portable to most CPUs. While Cilk does not support GPU programming itself, it seems that it could be combined with another programming model, such as OpenCL or CUDA, to launch tasks that run on the GPU.

- *Performance:* Cilk's work-stealing scheduler is highly scalable, both to high numbers of cores (pro-

[8] The serial elision of inlets is not valid ANSI C, but it is valid GNU C.

viding near-linear speedup, as long as there is sufficient work) and to small numbers of cores (negligible overhead) [87]. In a performance study of Cilk [114], Cilk was consistently one of the fastest models, on par with TBB, which has been highly optimized. Cilk had an average of 15-20x speedup on 32 cores over serial execution; its lightweight tasks and automatic load balancing give it excellent scalability even for dynamic problems that give other models trouble.

– *Productivity:* While Cilk is somewhat more verbose than other languages described below, it is still simpler and shorter than many other models, including TBB; its code size is only about 1.5x larger than Chapel, which is possibly the most succinct model discussed in this paper (see Sec. 4.4) [114]. In one study, Cilk's development time was compared to models like TBB and Chapel, and Cilk scored similarly to Chapel.

– *P3 Rating:* good – performance portable and productive, but could add direct support for more accelerators.

4.2 Legion and Regent

Legion [8] is a task-based programming system embedded in C++ and based on logical regions, which describe the data usage of a function/task and can be used to discover dependencies and task parallelism. Regions and tasks can both be partitioned to discover even more parallelism. Regions don't describe the physical data layout or location on the machine, but which data a task uses and its access privileges (e.g., read-only, read-write, etc.); physical data locations are managed by the Legion runtime. At any given time, a logical region might be mapped to zero or more physical instances of its data. The Legion runtime manages where these instances are located on multi-node machines to minimize communication and keeps instances containing the same data coherent.

The main choices Legion developers need to make are how to group data into regions and how to map the program onto the hardware. Legion provides a mapping API so users can specify rules for how Legion should map tasks and regions to hardware. This mapping is where most machine-specific Legion code ends up, which makes Legion very (performance) portable, since all users have to do to port to a new machine is write a new mapping, not modify source code (Legion also provides a default mapping to get applications up and running quickly). The mapping only affects performance, not correctness, and makes it very easy to compose different Legion applications, which can all use their own custom mappings.

Legion's execution model is based on a software out-of-order processor (SOOP) that schedules and runs a series of tasks on a (possibly distributed) system. The SOOP runs ahead of the actual computation to schedule tasks and distribute data to amortize the overhead; since each task describes which regions it uses and the privileges it requires, the SOOP can dynamically discover dependencies and parallelism at run time to create a correct schedule. The SOOP itself is pipelined so different stages discover dependencies and map tasks to increase parallelism further. After a task is mapped, it will execute once all its dependencies have completed, and if it spawns subtasks, those are sent to the SOOP for scheduling.

```
// Regent advection task
task simulate(zones_all : region(zone),
     zones_all_p : partition(disjoint,
    zones_all),
     points_all : region(point),
     points_all_private : region(point),
     points_all_private_p :
    partition(disjoint, points_all_private),
     conf : config)
where
     reads(zones_all, points_all_private),
     writes(zones_all, points_all_private)
do
     var dt = conf.dtmax
     var time = 0.0
     var tstop = conf.tstop
     while time < tstop do
          dt = calc_global_dt(dt, dtmax,
    dthydro, time, tstop)
          for i = 0, conf.npieces do
               adv_pos_full(
                    points_all_private_p[i],
    dt)
          end
          time += dt
     end
end
```

Listing 10: Regent code sample.

Regent [151] (code sample in Listing 10) is a task-based language built on top of Legion – Regent code appears to be sequential task calls, but the tasks are run in parallel by the Legion runtime. Regent tasks will be issued in program order, but the Legion runtime is free to reorder and run tasks in parallel, as long as all data dependencies are met. Regent is implemented as a source-to-source optimizing translator to Legion. As in Legion, users define tasks with the logical regions they use and what privileges they need on those regions; however, Regent automates much that Legion does not. Regent only exposes the logical level of Legion to users, hiding low-level details, including managing physical instances and

partitioning regions, which greatly reduces the amount of code users need to write. The Regent translator performs several optimizations on the Legion code it generates, including passing data as futures to avoid blocking, branch elision, and vectorization, which help Regent get closer to hand-written Legion performance.

- *Portability:* Since Regent/Legion is based only on C++, it can run on a wide variety of CPUs. However, the only GPU back end supported is CUDA, so accelerators are limited to Nvidia GPUs.

- *Performance:* Legion alone can get within 5% of hand-written application code when configured using pthreads for CPU code, CUDA for GPU code, and GASNet for communication, and when run on a single node. It also has good strong scaling [8]. With optimizations turned on, Regent gets similar performance [151], however, there are optimizations that have not yet been implemented that could further improve performance. Users must be careful when configuring Legion/Regent's CPU usage, though, since assigning too many threads/cores to the application leaves too few for Legion to use, and the application and Legion will interfere with each other execution, degrading performance.

- *Productivity:* Legion alone requires a great deal of code, but Regent can reduce this by half while also decreasing code complexity. Both Legion and Regent are highly scalable (Regent perhaps a little less than Legion), so codes can be tested on small machines and scaled up to larger machines with little work. Keeping the task and region mapping out of source code helps this a great deal.

- *P3 Rating:* good – Legion is high performance, and Regent is productive, but they could directly support more accelerators to be truly performance portable.

4.3 X10 and Habanero-Java

X10 [19] is an object-oriented (OO), task-based PGAS language based on Java and developed by IBM specifically to be high-productivity. The X10 developers were dissatisfied with the productivity of other HPC languages and with OO languages' support for cluster computing, so they developed X10 to solve those problems. The main design goals of X10 were safety (eliminating certain classes of errors), analyzability (for compilers, tools, and users), scalability, flexibility, and performance. They decided to use Java as a starting point because it already had most of those features, but its parallelism support was clunky and slow, so they replaced it with their own parallelism constructs.

X10 adds support for distributed systems with places, which are logical processes that contain a collection of (non-migrating) data objects and asynchronous activities (tasks) that operate on that data – somewhat similar to Regent's regions, but more like Chapel's locales. While the data and computations inside places don't migrate, the X10 runtime is free to move places as needed for load balancing. Tasks are similar to Cilk, but are created with the `async` keyword, and unlike Cilk, tasks may terminate before their children. X10 handles this with `finish` blocks, which force a task to wait until all children spawned inside the block have completed before returning itself; the main function is implicitly wrapped in a `finish` block. The `finish` block also handles any exceptions thrown by its child tasks.

X10 has `foreach` and `ateach` loops that launch tasks for parallel iteration over arrays within a place and across places, respectively. Arrays can be mapped to multiple places with a distribution attribute, but the distribution cannot change during the array's lifetime. X10's memory model is "globally asynchronous, locally synchronous;" within a place, data accesses are synchronous and ordered, but to modify data in another place, a task must launch another asynchronous task in that place, which has weak ordering semantics.

To give users more control over this, X10 provides atomic regions and clocks for synchronization. Atomic regions can contain multiple statements, and will behave as if those statements were executed atomically, with no interruptions. Clocks are a generalized barrier; at any time, a clock is in a certain phase and can have multiple tasks registered on it. Tasks can indicate to the clock they would like to move to the next phase, and will block until all tasks registered on that clock are ready to go on. Interestingly, if a program only uses `async`, `finish`, `foreach`, `ateach`, clocks, and atomic regions, X10 guarantees that the program will be deadlock free, which is a feature none of the other languages described here provide.

Habanero-Java (HJ) [16] (code sample in Listing 11) is a teaching language for parallelism built on X10. Like X10, HJ is focused on being safe, productive, and efficient. HJ uses the same abstractions as X10, although it upgrades clocks to phasers, which are even more general barriers. Tasks can register on phasers in different modes: signal-only and wait-only modes can be used to achieve producer-consumer behavior, or signal-wait mode can be used to get the same behavior as X10 clocks. HJ also adds array views similar to Kokkos array views, which allow users to view a one dimensional array as multidimensional using a custom index space.

- *Portability:* The JVM and runtime X10 and HJ are built on are very portable, enabling both to run on most CPUs. In addition, X10 can call GPU code

```
// Population simulation
void sim_village_par(Village village) {
// Traverse village hierarchy
    finish {
        Iterator
    it=village.forward.iterator();
        while (it.hasNext()) {
            Village v = (Village)it.next();
            async seq
                ((sim_level -
    village.level)
                    >= bots_cutoff_value)
                sim_village_par(v);
        } // while
[...]
    } // finish
}
```

Listing 11: Habanero-Java code sample.

(and C or Fortran) via its `extern` keyword, although the GPU code must be written separately.

– *Performance:* While the performance of X10 and HJ has not been as extensively studied as some other languages, X10 has been shown to get over 80% of the performance of the optimized versions of some benchmarks [165].

– *Productivity:* A productivity study was done on X10 [19], measuring the effort to port a serial Java benchmark suite to shared memory parallelism and distributed memory parallelism. They used LOC and source statement count (SSC) as proxies for productivity, since LOC can vary by programming style but SSC doesn't, and compute the change ratio for both. Using existing Java features, the total code change ratio to achieve distributed memory parallelism is 45-73%, which is extremely high. With X10, the change ratio is only 15-23%, which is much less effort. X10's deadlock free properties and other OO features also help productivity.

– *P3 Rating:* fair – X10/HJ is a teaching language now, rather than a production language, so performance and portability aren't as critical.

```
// triad benchmark
proc main() {
  // make domain (index set)
  const ProblemSpace: domain(1) dmapped
    Block(boundingBox={1..m}) = {1..m};

  // create and init distributed vectors
  var A, B, C: [ProblemSpace] elemType;
  initVectors(B, C);

  // an array of timings
  var execTime: [1..numTrials] real;

  // loop over the trials
  for trial in 1..numTrials {
    const startTime = getCurrentTime();

    forall (a, b, c) in zip(A, B, C) do
      a = b + alpha * c;

    execTime(trial) = getCurrentTime() -
   startTime;
  }
  const validAnswer = verifyResults(A, B,
   C);
}
```

Listing 12: Chapel code sample.

4.4 Chapel

Chapel [17, 18] (code sample in Listing 12) is a parallel language from Cray, built on top of a source-to-source translator to C. Like X10, Chapel is designed to be highly productive and high performance (in that order), and to remedy many problems with existing parallel languages and programming models. In brief, Chapel is intended to be as programmable as Python, as fast as Fortran, as scalable as MPI, as portable as C, and as flexible as C++. Chapel's specific goals are:

– provide a global-view PGAS programming model where users express their algorithms and data structures cohesively, at a high level, and map them to hardware separately (as opposed to a "fragmented" model, where users write individual kernels specific to the hardware and must manage many low-level details, like CUDA[9]);

– support generalized, multi-level, composable parallelism – both task and data parallelism;

– separate algorithms from implementations, so user code is independent from the underlying hardware configuration and data layout in memory (e.g., dense and sparse matrix multiplication can be done with the same code);

– provide more modern language features, such as object-oriented programming, function overloading,

[9] The Chapel developers believe the prevalence of fragmented models is one of the primary reasons parallel computing remains so difficult. They do admit that writing compilers and runtimes for fragmented languages is much easier than for global-view languages, though.

garbage collection, generic/template programming, and type inference;

– provide more data abstractions than simple dense arrays and structs, like sets, graphs, and hash tables;

– provide execution model transparency, since developers can write better code when they have an idea how what they write will map to the hardware (for parallel languages, this includes data distribution and communication);

– interoperate with legacy code, since there's no way all legacy code will be rewritten in a new language and users don't want to throw away the effort that went into writing and testing it;

– be portable;

– be high performance.

To achieve these goals, Chapel provides a global-view, block-imperative structure, similar to C or Fortran, but with rather different syntax. The Chapel developers deliberately moved away from C- and Fortran-like syntax to make it more difficult for their users fall into old C and Fortran habits and to force them to think about the code they're writing. While many languages target a "lowest common denominator" execution model, Chapel decided to target a more sophisticated, multithreaded model whose more complex components can be emulated in software on systems that don't fully support them (albeit with a potential performance hit, which was later significantly reduced [18]), which improves portability by allowing for higher level abstractions. Even though Chapel is focused on providing high-level abstractions for the sake of productivity, it provides multiple lower-levels of abstractions for performance tuning, which they call "multi-resolution programming." Chapel operates on the "90/10" rule: if 90% of a program's time is spent in 10% of the code, the user should be able to optimize that 10% while keeping the other 90% simple – the common case should be fast (to write).[10]

Chapel abstracts away hardware configurations as locales, where each locale is roughly equivalent to a node in a cluster, where one or more cores all share the same memory. Chapel provides users with an array of locales that they can partition and rearrange to mimic the logical arrangement of their computation. To cope with changing node architectures, Chapel added support for user-defined locales, which describe how to manage memory and computations within a locale, so novel architectures can easily be supported with high performance. Chapel plans to provide various locale types itself (an experimental Xeon Phi locale was in the works as of 2018), but being able to define their own locales can help users get up and running more quickly than waiting for official language support.

Data parallelism is done with domains, which are index sets that can contain traditional ranges of integers or arbitrary types to provide key-value, hash map-like behavior. Domains can be partitioned arbitrarily into subdomains, or a generic array slice. Domains can be divided among locales with distributions; Chapel provides some default distributions like block or cyclic, or users can define their own. Chapel's `forall` loop can iterate over (sub)domains, arrays, or any expression that produces a sequence or collection. If a distributed domain is part of the loop, Chapel will automatically distribute the computation to all the relevant locales. Since domains and distributions abstract array accesses, this enables separation of algorithms and implementations, as desired, and improves performance portability. To change to a new architecture or data model, all users need to do is swap distributions; no code changes are necessary.

Task parallelism is done with `begin` and `cobegin` statements that launch tasks to run in parallel. These tasks can perform one-sided, SHMEM-style communication with synchronization variables that use full/empty semantics – when a task writes to the variable, it becomes full and a task waiting on it can read it so it becomes empty again (users can modify these semantics as well). Chapel also provides support for atomics.

All these parallelism constructs can be composed arbitrarily to provide highly nested parallelism. One of the Chapel developers' complaints about other parallel programming models is that they only provide one or maybe two levels of parallelism before another model needs to be brought in (e.g., using MPI+OpenMP+CUDA to program a multicore+GPU cluster in the days before OpenACC and OpenMP 4); Chapel does not have this problem. The developers note that specifying what *can* be run in parallel is distinct from specifying *where* it can run; this philosophy allows Chapel to be highly portable, as well, to multicore, multi-node, and heterogeneous systems.

In addition, Chapel provides all the modern features listed above, and then some, including: objects and classes, garbage collection, generic programming via latent types/type inference and type variables (similar to C++ template types or Haskell type variables), iterators, currying, and modules/namespaces. Chapel is also interoperable with C and Fortran, two of the most popular languages in HPC.

Chapel has been extended to target Nvidia GPUs [150] (while remaining performance portable to multicores) by providing new domain distributions, a new back end for CUDA, and a set of transformations so that code

[10]This is similar to OPL's (Sec. 3.1.2) philosophy of providing many abstraction levels.

meant for GPUs still has high performance on multi-cores.[11] Chapel's `GPUDist` domain distribution indicates data should be moved/stored in GPU memory and that computations should happen on the GPU; `forall` loops over GPU data will map each loop iteration to a single GPU thread. GPU variables are created the same as normal variables, so no code changes are necessary to port to GPUs beside changing the distribution. Chapel can automatically handle data movement to the GPU; if the user declares a variable with a GPU distribution, it will be available on both the device and host, and the Chapel compiler will do data dependence analysis to determine when to copy it. If the user wants more control over data movement, they can change the distribution to `GPUExplicitDist` and perform data movement themselves by creating a host and device version of the array and assigning one to the other. Users can also specify that data should live in a special GPU memory (e.g., a fast read-only texture cache) by changing the distribution.

More recently, a new multi-level GPU interface was proposed [54, 55] that supports both Nvidia and AMD GPUs and gives users more control over their device code. The new interface gives users several options, from writing CUDA or OpenCL kernels and managing memory transfers themselves to writing a kernel in a mid-level language that Chapel translates to GPU code. This places the GPU interface more in line with Chapel's "90/10" rule. If users wish to further optimize their most important GPU kernels, they can now do so.

- *Portability:* While Chapel was originally designed by Cray for their multi-node multicore systems, it has always been intended to be very portable, even to non-Cray machines [18]. Support for Nvidia and AMD GPUs has been added via CUDA and OpenCL back ends and GPU data distributions [150]. User-defined locales (described earlier) have also been added, making Chapel extremely portable [18].

- *Performance:* Early on in the Chapel development process, performance and scalability were not high priority, and thus Chapel often performed much worse than other models [114]. However, as Chapel began to mature, performance and scalability became concerns, and since the end of the HPCS project in 2013, have been a major development focus. Chapel has improved a great deal, to be comparable with MPI+OpenMP [18]. Much optimization was done to their array implementation (now 100+x faster), communication implementation (moved from single-item to bulk messages), and the runtime in general. The Chapel developers are investigating more optimizations to improve scalability, which lags slightly behind MPI at very large (1000+) node counts. It was never Chapel's design limiting its performance, only the implementation.

  Chapel also offers users some compile-time support for optimizing their programs via the `void` type, which has semantics quite different from `void` in other languages [18]. Chapel supports setting variable types dynamically based on compile-time parameters, and if a type is `void`, all code that uses it is ignored and not compiled. This gives users a way to specialize code while maintaining a single source, which also improves productivity.

  Chapel's CUDA support [150] also gives performance equivalent to native CUDA, when explicit copies are used. Implicit copies add some overhead because their analysis is conservative. Their compiler is also capable of compiling code meant for GPUs into multicore code and getting excellent performance (much better than the other CUDA-to-multicore compilers of the day), which makes Chapel very performance portable.

- *Productivity:* Chapel is much shorter than other parallel code [114] (particularly CUDA [150]), and is also cleaner and easier to understand. While Chapel was lacking in traditional language features early in its development since the team was more interested in its research features, that has since been remedied and these features have been added and/or much improved. The semantics of the language also underwent some changes to make things work as users expected. For example, previously, variables declared outside parallel constructs were treated as global variables and liable to cause data races, but now these variables are treated as being "passed in" and each thread gets a private copy (users can modify this with task/forall data intents if they desire different behavior). Automatic communication is also a great productivity boost, since users don't have to worry about managing those details. Porting between architectures is also incredibly simple and requires very few code changes. Overall, Chapel has many high productivity features.

- *P3 Rating:* excellent – Chapel provides users with a way to increase its portability, and its performance is generally very high, making it exceptionally performance portable, and very productive, if developers are willing to take on a new language.

[11] Chapel was later extended with an OpenCL back end to support AMD GPUs [20].

```
// Vector addition
// declare vectors
std::vector<int> a(N), b(N), c(n);

// create kernel as lambda
auto vadd = kernel([](const auto& a, const
    auto& b auto& c){
     auto i = Thread::get().global;
     if (i >= a.size()) return;
     c[i.x] = a[i.x] + b[i.x];
// specify launch configuration
}, {{N/1024 + 1}, {1024}};);

// call kernel, wait for result
std::future<void> F = std::async(vadd, a,
    b);
F.wait();
```

Listing 13: PACXX code sample.

4.5 PACXX

PACXX (Programming Accelerators with C++) [50, 51] (code sample in Listing 13) is a language built on C++ that simplifies accelerator programming by allowing users to write pure C++ code (with STL-like library calls) that is compiled and run on GPUs. The main goal of PACXX is to simplify GPU programming and address various other problems of OpenCL and CUDA. PACXX is based on C++14 and the C++14 STL additions with no syntax extensions, so PACXX code is valid C++ (unlike CUDA, which extends C++ with triple angle-bracket syntax).

PACXX kernel functions are specified as lambdas or normal C++ functions in the style of CUDA kernels (with a C++ interface to get thread indices and such), not as strings like in OpenCL, which is both difficult to debug and can pose a security risk.[12] Kernel invocations are done with a library call to a templated function that takes the kernel function and launch parameters. PACXX kernels also work with standard STL containers, atomics, and other functions to simplify kernel writing, as well as allowing users to pass by reference, return values, and do synchronization with futures. Like many of the libraries described earlier, PACXX's implementation of STL containers can automatically manage data movement via a lazy copying mechanism without user involvement. C++ attributes can also be used to specify kernel launch configurations and which memory data should reside in to allow users to further optimize their code. Users can manage accelerators and memory through PACXX-provided classes if they want even more control.

PACXX was implemented as a custom compiler[13] [50] that uses the Clang front end and LLVM and modifies host code to launch kernels automatically – the user only needs to write C++ as they normally would. PACXX compiles C++ down to SPIR (to target AMD devices) or PTX (to target Nvidia devices). The PACXX compiler has been updated [51] to support just-in-time (JIT) compilation for kernels; the compiler first makes the host executable, then partially compiles the device kernels and adds them to the binary to be JIT compiled at run time when more values are known and can be used to optimize the kernels (see Sec. 7.2.1 for more about JIT).

- *Portability:* PACXX has back ends for both Nvidia and AMD GPUs (a somewhat rare, but important, combination to support), and, via the C++17 additions to the STL, multicore and sequential CPU configurations. Since the C++17 STL standard was released, PACXX has modified its STL usage to be fully compatible (by adding a PACXX execution policy; see Sec. 7.2.1 for more on the C++ STL) so PACXX code can fall back to multicore or sequential execution on platforms without a GPU [51]. However, since PACXX still requires users to specify kernel launch configurations and allows users to specify architecture-specific attributes, it may require changes to source code or `#ifdefs` to port to another machine.

- *Performance:* PACXX performance has been compared against pure CUDA and OpenCL [50, 51] as well as Thrust, a productivity-oriented template library for Nvidia GPUs [51]. PACXX matches CUDA performance with a small overhead of around 3% on Nvidia GPUs, and matches OpenCL performance on Xeon Phis. OpenCL slightly outperforms PACXX on Nvidia GPUs and Intel CPUs, but this is likely because the OpenCL implementation they tested against used lower precision, non-IEEE compliant arithmetic on the GPU and the Intel compiler performs extra optimizations to the generated SPIR kernels. PACXX outperforms Thrust on several benchmarks because it has better heuristics for launch configurations and JIT-ing kernels gives them better implementations of transforms and reductions.

- *Productivity:* As mentioned earlier, requiring users to specify kernel launch configurations can hurt productivity by making users change their code for each architecture. However, since it uses plain C++ and the STL, PACXX code is much shorter and easier to write and understand than CUDA and other GPU programming models, especially for users used to

[12] The source code is available at run time, and is more easily read or modified by a malicious party.

[13] Which is why it is in the languages section, not the libraries section.

```
// Buffer example
int b[N], c[N];

// put data in buffers
buffer<int, 1> B(b, range<1>{N});
buffer<int, 1> C(c, range<1>{N});

// submit this to a device queue
myQueue.submit([&](handler& h) {
    // request buffer access
    auto accB =
   B.get_access<access::mode::read>(h);
    auto accC =
   C.get_access<access::mode::write>(h);

    // create kernel
    h.parallel_for<class
   computeC>(range<1>{N}, [=](id<1> ID) {
        accC[ID] = accB[ID] + 2;
    });
});
```

Listing 14: SYCL code sample with buffers.

```
// Shared memory example
// create host and shared arrays
int* hostArray = (int*)
    malloc_host(N*sizeof(int));
int* sharedArray = (int*)
    malloc_shared(N*sizeof(int));

// submit this to a device queue
myQueue.submit([&](handler& h) {

    // create kernel
    h.parallel_for<class
    myKernel>(range<1>{N}, [=](id<1> ID) {
        int i = ID[0];

        // access shared and host array on
    device
        // shared array will be copied
    over; host array will not
        sharedArray[i] = hostArray[i] + 1;
    });
});
```

Listing 15: DPC++ code sample with shared memory.

C++. Furthermore, since PACXX supports type inference in kernels via the new `auto` type inside lambdas [51], it allows users to write kernel templates that work with many types, which can significantly reduce code duplication and improve productivity.

– *P3 Rating:* good, promising – if the need for launch configurations was removed, PACXX's performance portability and productivity would improve greatly, but it's already quite portable and productive, especially for C++ developers.

4.6 SYCL and DPC++

SYCL [182] (code sample in Listing 14) is a high-level language built on OpenCL and designed to improve OpenCL's usability so it can be used to write single-source C++ for accelerators. This means host and device code live in the same source file, which can be analyzed and optimized by the same compiler for better performance. SYCL is built on C++14 and intended to follow the C++ specification as much as it can, so it does not change or extend C++ syntax in any way (i.e., a CPU-only implementation could work with any C++ compiler). It allows C++ libraries to work with OpenCL, and allows OpenCL kernels to work with C++ features, such as templates and lambdas. SYCL is intended to be easy to use while still giving the performance and control of OpenCL, and focuses on intra-node parallelism, leaving inter-node parallelism to other models like MPI.

SYCL takes an explicit data parallelism approach, where the user declares what can be run in parallel with Kokkos-style `parallel_for` templated function calls. Kernels can be written as lambdas, functors, OpenCL code, or loaded as SPIR binaries. SYCL can output kernels at compile time as device-specific executables, or as SPIR, so kernels can be compiled at run time for any device. To manage data movement, SYCL uses buffers, which are an abstraction over C++ objects, and can represent one or more objects at any given time (as opposed to a specific memory location). Inside a kernel, users can request access to a buffer with a set of permissions (similar to Legion/Regent privileges), and SYCL will automatically handle data movement to and from the device based on those permissions. While this can be cumbersome, it does allow SYCL to optimize data transfers to the device.

SYCL kernels are run by submitting them to device queues, which allow for some task parallelism, since by default these queues impose no ordering on the execution of kernels submitted to them. Each device can also have multiple queues running kernels on it (although each queue can only be bound to one device). The data dependencies and access permissions of each kernel are used to enforce an ordering on each queue, or users can explicitly specify dependencies between kernels. SYCL queues only specify which kernels *can* be run in parallel, but provide no guarantees about what kernels will be run in parallel.

Data Parallel C++ (DPC++) [142] (code sample in

Listing 15) is a language from Intel built on top of SYCL (it's "SYCL with extensions"). Intel's new oneAPI [62] is also built primarily on DPC++, with many additional libraries and interoperability with languages other than C++. DPC++ is intended to be a place to experiment with new parallelism features that could be added to the C++ standard. Some of the features already added include support for hierarchical parallelism inside kernels, ordered device queues, per-device versions of kernels, and unified shared memory.

Unified shared memory is perhaps the most interesting of these. It removes the need for specifying data use permissions or manually managing data movement by allowing users to create host, device, and shared pointers via `malloc_host`, `malloc_device`, and `malloc_shared` functions. Device pointers are allocated only on the device and can't be accessed by the host, while host pointers are allocated only the host, but can be accessed by the device *without being transferred*, so accesses are likely to be very slow. Data can be explicitly moved by copying a host pointer into a device pointer inside a kernel, or vice versa. Shared pointers can migrate data automatically between the host and device whenever the data is accessed, so the first few accesses are slow, but afterwards accesses can happen from fast device memory. This greatly lessens the burden on the developer, so they can quickly prototype an application, then optimize memory transfers as necessary.

- *Portability:* Since SYCL and DPC++ are built on top of OpenCL and C++, they are extremely portable, and can be used to target any device OpenCL can target (which is a great deal).

- *Performance:* SYCL and DPC++'s performance is theoretically bounded above by OpenCL's, but it isn't subject to the same performance variations as OpenCL. While in the past SYCL's performance has lagged behind OpenCL's due to implementation difficulties [24, 133], it now performs similarly to OpenCL on many architectures [28]. However, as it is very new, some vendors do not yet support it, or the performance of their implementation could use improvement.

- *Productivity:* SYCL and DPC++ are certainly more productive than OpenCL, since they remove most of the boilerplate OpenCL is known for and update many features to be compatible with C++. However, some SYCL syntax (buffers in particular) is still verbose, and compared to other languages (e.g., Chapel), a great deal of code is still required because users must write and launch individual kernels. DPC++ has been remedying some of this already (e.g., with unified memory), and will hopefully continue to do so.

- *P3 Rating:* very good, promising – SYCL and DPC++ solve many problems with OpenCL's performance and productivity which were preventing it from being a truly performance portable model. All that's currently lacking is robust vendor support.

4.7 OpenCL

While OpenCL is not performance portable in and of itself (as noted in Sec. 2.5 and above), due to its popularity, there have been efforts besides SYCL/DPC++ to make performance portable implementations, which have been somewhat successful. pocl [64] is one such implementation, which implements new LLVM passes inside Clang to transform OpenCL kernel code meant for one architecture into kernels that will have high performance on another architecture. PPOpenCL [93] is another implementation built on Clang that performs whole-program transformations across host and device code for the same purpose.

pocl enables kernels written for one device to achieve performance close to that of a device-specific kernel on another device. PPOpenCL can achieve a geometric mean speedup of 1.55x with a single source code over the baseline OpenCL implementation on several architectures, even outperforming OpenACC on some benchmarks. Combining pocl and PPOpenCL (PPOpenCL does no kernel-only transformations, but pocl does) outperforms each individually, so while writing new OpenCL code with the intention of running it on multiple devices might not be advisable, existing OpenCL can at least achieve some degree of performance portability.

5 Directive-based Models

Directives are language extensions, although sometimes they're considered languages in and of themselves. Directives are annotations for base languages, usually C, C++, and Fortran, that allow users to give the compiler extra information, such as which loops can be parallelized or what arrays need to be moved to the GPU. While some [159, 132] consider directives to be low-level because users must still specify data movement and describe parallelism themselves, directives are still more concise, portable, and high-level than models like OpenCL or pthreads. This section will describe the most popular directive-based models.

```
// Heat conduction (TeaLeaf)
// set up data environment, copy r and p
    to device
#pragma omp target data map(to: r[:r_len])
    map(tofrom: p[:p_len])
{
    // describe loop parallelism
    #pragma omp target teams distribute
    collapse(2)
    for (int jj = pad; jj < y-pad; ++jj) {
        for (int kk = pad; kk < x-pad;
    ++kk) {
            const inst index = jj*x + kk;
            p[index] = beta*p[index] +
    r[index];
        }
    }
} // p gets copied back
```

Listing 16: OpenMP code sample.

5.1 OpenMP

As described in Sec. 2.5, OpenMP [124, 125, 126] (code sample in Listing 16) began as a way to simplify programming shared memory CPUs, specifically targeting parallel loops with directives to tell the compiler how loops should be parallelized. (Version 3 also added directives for task-based parallelism.) Version 4.0 began adding support for heterogeneous, accelerator-based computing and performance portability features, a trend which has continued for Version 5.0 – both of these are discussed further below. OpenMP follows a more "prescriptive" programming model, where developers explicitly define how their code should be mapped onto the hardware for parallel execution, but is beginning to add some "descriptive" directives that give the compiler more freedom. (This idea will be revisited in Sec. 5.2 on OpenACC.)

OpenMP's original goals when it was created in the mid-1990s were to provide portable, consistent, parallel, shared-memory computing for Fortran, C, and C++, maintain independence from the base language, make a minimal specification, and enable serial equivalence [27]. OpenMP has mostly kept to these goals, with the exception of adding support for models other than data parallelism, which has made it difficult to keep the specification small, and abandoning serial equivalence as unrealistic for modern architectures.[14] There is an ongoing debate about which parallelism models OpenMP should support, and exactly how many basic programming constructs should be added to OpenMP to support these other forms of parallelism; some expect that OpenMP will move closer to becoming a general-purpose language in its own right in the future [27].

[14] A subset of OpenMP does maintain serial equivalence, but keeping to this subset severely restricts what programmers can do.

5.1.1 OpenMP 4.x

As mentioned earlier, OpenMP's primary abstraction is parallel loops, and OpenMP provides directives for users to describe how their loops should be mapped onto parallel hardware. OpenMP 4.0 added `target` and `map` directives to denote that code regions and data should be offloaded to an accelerator, as well as various clauses to modify how code is mapped to the device. Version 4.0 also added the `simd` directive to force vectorization (auto-vectorization can vary by compiler and greatly influence performance) and improvements for task-based parallelism and error handling [27].

OpenMP 4.5 further improved accelerator support by adding unstructured data regions, so `map` directives can be moved into functions, which improves readability and usability. However, OpenMP 4.5 also modified how certain types of data (e.g., scalars) are copied onto the device, which can make porting between OpenMP 4.0 and 4.5 error-prone, as noted by Martineau et al. [98]. The 4.x specification also does not define support for copying pointers in data structures to the device ("deep copy" support), but some compilers still support it, which can make porting between compilers difficult.

OpenMP 4+ has been struggling with implementation support. Even though the 4.0 specification was released in 2013, it has taken many years for some compilers to offer even basic support for offloading directives, and performance can still vary wildly between compilers. As Martineau et al. note, developers who were used to consistent high performance from OpenMP 3 will be in for a surprise, and may be better served by waiting until compiler vendors have had more time to improve their implementations.

5.1.2 OpenMP 5.0

OpenMP 5.0 adds several new features specifically for performance portability and to make supporting multiple architectures easier. Version 5.0 adds `metadirective`, `declare variant`, and `requires` constructs that allow users to denote that some directives or functions should only be used on certain hardware. While semantically similar to preprocessor directives like `#ifdef`, these constructs give the compiler more information to reason about which version should be used, instead of naively copy-pasting code [136]. Version 5 also adds support for deep copying (to handle data structures with pointers, as mentioned above), iterator-based ranges, and interacting with the memory hierarchy, all of which are becoming more widely used in modern code.

Version 5.0 is also adding some more "descriptive" constructs, such as `loop` (similar to OpenACC's `parallel loop` construct) and `order(concurrent)`, so users can opt to let the compiler make choices for them. One goal of OpenMP 5.0 is not to interfere with threading and memory models that are currently being added to the base languages, so allowing the compiler (which should know more about these models) to make more choices could be helpful.

5.1.3 Future OpenMP

OpenMP is still evolving, and there are many features that may be added in the future [27]. Some of these features include: data transfer pipelining, memory affinity, user-defined memory spaces, event-driven programming, and lambda support. The standards committee is also considering adding "free-agent" threads that can help with load balancing by joining parallel regions.

- *Portability:* Since OpenMP was very popular for shared memory programming before it ever began to be used for heterogeneous performance portable programming, a great deal of effort has been put into porting it to a wide variety of architectures. OpenMP 3 had implementations for a wide variety of CPUs, and OpenMP 4 has added support for GPUs and Xeon Phis. There are numerous implementations of OpenMP [127] from both academia and industry, including ones from Intel [135, 136], AMD, Cray [79], PGI, IBM, GNU, and LLVM [12, 4, 129].

- *Performance:* OpenMP 3 has many mature implementations that get extremely high performance, but OpenMP 4+ is still fairly new and its performance varies. In one study [98], Martineau et al. compared six OpenMP 4.5 implementations (from Intel, Cray, LLVM, PGI, IBM, and GNU) on multiple applications and architectures, and found that each compiler implemented the standard slightly differently – directives that gave good performance with one compiler and architecture didn't with another compiler, even on the same architecture.

  OpenMP 4+'s performance can match the performance of OpenACC, another directive-based model, but usually falls short of CUDA's performance [97, 98]. This isn't surprising, since CUDA is much more low level, but with further optimization to GPU code generation, it may be possible to match CUDA. Giving OpenMP a fair comparison can also be difficult, since support for the specification has been lagging, which may interfere with results since developers have to hack around deficiencies; for example, this impacted Pennycook et al.'s [135] study of OpenMP's performance portability compared to Kokkos.

- *Productivity:* OpenMP is generally considered to be very expressive for parallelism, but without requiring details like CUDA or explicit threading, and the porting process is usually straightforward [97]. The new features also allow C and Fortran users to express concepts that would require C++ templates or excessive code with directives, which is a great productivity boost [135]. Unfortunately, OpenMP currently requires different directives to get optimal performance on CPUs and GPUs, so it doesn't quite provide a single-source solution for applications (which is not good for performance portability), but new additions such as the `metadirective` construct ameliorate that to some degree.

  The OpenMP 5.0 specification was released quite recently, near the end of 2018, and as of now there are no compilers that fully support it, but some work has been done evaluating the new features' usability. A study of some additions to OpenMP 5.0 [135] found that `metadirective` is good for describing simple patters, like `target` versus `parallel for` (for GPU vs. CPU), but not for more complex directives where users might want to turn specific clauses on or off depending on the target. The `declare variant` directive was also useful, but still creates the same version bifurcation problems as `#ifdef`, and users need a way to specify which version to call, if necessary. The authors don't believe there will be much use for the `requires` directive in production codes, but during the development and debugging process it could have been very useful.

- *P3 Rating:* good – OpenMP is almost performance portable, but its performance consistency needs more work, and it would be preferable if accelerator versions didn't need different directives. It is, however, highly productive.

5.1.4 OpenMP 3 to GPGPU

As an interesting aside, even before OpenMP moved to officially support heterogeneous computing, there were efforts to enable OpenMP-based GPU computing. Lee et al. [86, 83] created OpenMPC, a source-to-source translator from OpenMP 3 to CUDA. Their translator took in OpenMP code, transformed it so it was better organized for the CUDA programming model, then translated parallel loops into optimized CUDA kernels with the appropriate data transfers. They provided extra directives so users could control and further optimize the translation into CUDA, if desired. Lee et al. compared their optimized, generated CUDA against hand-tuned CUDA, and found that the average performance gap was less than 12%. They noted that the generated CUDA took signif-

```
// Head conduction (TeaLeaf)
// set up data environment, copy r and p
   to device
#pragma acc data copyin(r[:r_len])
    copy(p[:p_len])
{
    // describe loop parallelism
    #pragma acc kernels loop independent
    collapse(2)
    for (int jj = pad; jj < y-pad; ++jj) {
        for (int kk = pad; kk < x-pad;
    ++kk) {
            const inst index = jj*x + kk;
            p[index] = beta*p[index] +
    r[index];
        }
    }
} // copy p back
```

Listing 17: OpenACC code sample.

icantly less effort to make, demonstrating that directives can greatly increase productivity, if developers are willing to take a small performance hit (though hopefully in the future, performance will be more similar).

5.2 OpenACC

OpenACC [121, 122, 123] (code sample in Listing 17) is a directive-based model originally designed for GPU computing, although there are now implementations that target multicore, Xeon Phis, and FPGAs [84, 85, 180, 77]. OpenACC's main goals were to enable easy, directive-based, portable accelerated computing (which OpenMP didn't have at the time) and to merge several individual efforts into a single standard. OpenACC came out of a combination of CAPS' OpenHMPP [33], PGI Accelerator [181], and Cray's extensions to OpenMP. Like OpenMP, OpenACC supports C, C++, and Fortran as base languages.

Unlike OpenMP, however, OpenACC follows a more "descriptive" approach, whereas OpenMP is "prescriptive." OpenMP requires developers to explicitly define how parallelism is mapped onto hardware, while OpenACC leaves most choices up to the compiler. There has been much discussion about which approach is best for performance portability, but in truth (as noted by de Supinski et al. [27]), this is a false binary, and these models exist on a spectrum. While it would be nice if users could add a few descriptive directives and have things "just work," often that isn't possible and prescriptive models are still necessary. OpenACC has recently begun adding more prescriptive constructs so users can have more control over tuning their code.

5.2.1 OpenACC 2.x

The original OpenACC standard had fairly basic directives for annotating parallel loops and describing data movement. The 2.x versions added support for asynchronous compute regions, function calls within compute regions, atomics, an interface for profiling tools, and other usability improvements, such as modifying the semantics of `copy/copyin/copyout` to include checking whether data was already present to minimize unnecessary data transfers. Basic support for user-defined deep copy operations on data structures containing pointers was also added, which is very important for scientific applications that use deeply nested data structures.

5.2.2 OpenACC 3.0 and Future Versions

Version 3.0 (very recently released) adds more support for multi-GPU configurations, which are becoming more common, and lambdas in compute regions. It also introduces somewhat stricter rules for which parallelism clauses can appear together. OpenACC is generally adding more prescriptive options, such as loop scheduling policies and optimization directives (e.g., `unroll`), to allow users to make decisions where the compiler can't.

- *Portability:* OpenACC has many implementations from industry and academia, including PGI, Cray, GNU, OpenARC [84, 85], OpenUH [89], accULL [143], and IPMACC [80]. Together, these implementations support most hardware currently being used, including most CPUs, Nvidia GPUs, AMD GPUs, and various FPGAs. However, very few individual implementations support all of these, so to move between hardware, users might need to swap compilers, which can be problematic for performance.

- *Performance:* While performance varies by compiler vendor [57] and parallelism construct (`kernels` vs. `parallel` directives), optimized OpenACC generally achieves performance close (within 80-90%) to native CUDA or OpenCL [57, 85, 134, 99]. Choosing the correct optimizations can be difficult, however, and choosing an incorrect optimization can hurt performance on some architectures. In particular, loop ordering to coalesce memory accesses, minimizing data transfers, and using a sufficient number of threads can have a large impact on performance.

- *Productivity:* Like OpenMP, OpenACC code tends to be significantly shorter and simpler than code from other models. Unlike OpenMP, however, OpenACC doesn't require different directives to target different architectures, so a single implementation of an OpenACC program can run on any architecture (albeit with a potential performance hit). OpenACC's

```
// XACC distributed array example
// set up distribution
#pragma xmp nodes n(1, NDY, NDX)
#pragma xmp template t(0:MKMAX-1,
    0:MJMAX-1, 0:MIMAX-1)
#pragma xmp distribute t(block, block,
    block) onto n

// apply distribution to array
float p[MIMAX][MJMAX][MKMAX];
#pragma xmp align p[k][j][i] with t(i, j,
    k)

// set up halo region
#pragma xmp shadow p[1:2][1:2][0:1]

// computation with OpenACC directives
    omitted for brevity

// halo exchange (ik-plane and jk-plane)
#pragma xmp reflect(p) width(1,1,0) acc
```

Listing 18: XACC code sample.

descriptive philosophy also arguably makes it easier to learn than OpenMP, since it requires fewer extra clauses per loop to get good performance, although some consider the two to be roughly equivalent [99].

– *P3 Rating:* very good – OpenACC is very portable, and (while its performance might not be quite as high as other models), it is performance portable. It is also very productive.

5.3 XcalableMP and XcalableACC

XcalableMP (XMP) [112] and XcalableACC (XACC) [113, 162] (code sample in Listing 18) are directive-based, PGAS models built on the Omni compiler for C and Fortran. XcalableMP is based on OpenMP and High Performance Fortran, and is meant to be a replacement for MPI, providing high-level directives for data movement instead of low-level functions. XcalableACC is an extension to XMP that modifies some XMP directives to work with accelerators and defines how XMP interacts with OpenACC to support heterogeneous computing. Both XMP and XACC support C and Fortran and can use CUDA and OpenCL to target Nvidia and AMD GPUs. XMP and XACC also provide coarray notation (another form of distributed arrays) for C and are interoperable with coarrays in Fortran 2008 and later (for further discussion of coarrays, see Sec. 7.2.2).

XMP and XACC use `template` directives to define a virtual index range, `node` directives to define a set of units of execution (not necessarily compute nodes), and a `distribute` directive to divide the index range across the nodes. The `align` directive declares a global array that uses this distribution. Both XMP and XACC support uniform block, cyclic, and block-cyclic distributions, as well as general, arbitrary-sized block distributions. The `shadow` and `reflect` directives declare halo regions on distributions and synchronize halos, respectively. With XACC, all data directives can be used to control data allocated on accelerators as well. XMP has its own directives for parallelizing loops, and XACC uses normal OpenACC directives.

– *Portability:* XMP and XACC both have back ends for CUDA and OpenCL, and can therefore target most CPUs and accelerators, including Nvidia GPUs, AMD GPUs, and Intel Xeon Phis. By supporting both MPI-style two-sided communication and PGAS-style one-sided communication (via coarrays), XMP and XACC allow users to have a great deal of control over communication patterns that can improve portability and performance when moving between architectures.

– *Performance:* Performance tests have shown that XMP has better performance than similar PGAS languages like UPC [112]. Early performance tests comparing XACC without coarrays to MPI+OpenACC across various node configurations show that XACC outperforms MPI+OpenACC on small problem sizes (even when MPI+OpenACC uses Nvidia's GPUDirect RDMA utility) and performs similarly to MPI+OpenACC on larger problems sizes [113]. Later tests with coarrays (and more mature OpenACC implementations) show that XACC usually gets over 90% of MPI+OpenACC performance, and is often within 97-99% [162].

– *Productivity:* MPI can be difficult to program since it is low-level and users have to do all communication manually. XMP replaces these with high-level directives that greatly simplify the process. XMP adds less code than other PGAS languages like UPC, and XACC adds less code than MPI+OpenACC.

  Since XACC is very similar to XMP+OpenACC, developers can use existing OpenACC resources so the learning curve isn't too steep, and porting from XMP to XACC is as simple as adding OpenACC directives. Going from OpenACC to XACC is also simple, since users just need to add XMP directives.

– *P3 Rating:* excellent – XcalableMP and XcalableACC are very performance portable and productive, although they may not give the level of control some users want, being so high-level.

```
// OpenMC example
// split MIC 0 into two workers
#pragma omc worker name(MIC:0:0-29:1,
    "M00")
#pragma omc worker
    name(MIC:0:30-59:1,"M01")
#pragma omc worker name(CPU:0:0-15:1,"P0")
#pragma omc worker name(CPU:1:0-15:1,"P1")

// launch task
#pragma omc agent // Agent 1
{
    PF(0); BR(0); SW(0);
}
#pragma omc wait all
for(i = nb; i < N; i += nb) // main loop
{
    // launch two more tasks on part of MIC
    #pragma omc agent flag(i) on(M00,M01)
    priority(1) // Agent 2
        Upd(i, nb);

    #pragma omc agent deps(i) // Agent 3
    {
        PF(i/nb); BR(i/nb); SW(i/nb);
    }
    #pragma omc wait all
}
```

Listing 19: OpenMC code sample.

5.4 OpenMC

OpenMC [91] (code sample in Listing 19) is a directive-based model meant for the TianHe series of supercomputers, but applicable to other architectures as well, and with several unique features that could improve the productivity of other directive-based models. Liao et al.'s motivation for creating OpenMC was some unusual architectural features of the TianHe supercomputers. The TianHe machines use multiple types of accelerators, including Xeon Phis and GPUs, and the primary way of programming them was MPI+OpenMP+X, where X is an accelerator-specific language; if developers wanted their application to use the different kinds of accelerators, they had to keep multiple versions of their code, which (as has been mentioned several times) isn't sustainable. Liao et al. wanted to provide a way to make use of all the hardware, including the multicore CPUs, in each node, so they developed OpenMC as a new intra-node parallel programming model.[15]

OpenMC treats hardware as a logical group of workers, onto which a master thread launches tasks, which can be serial or parallel. OpenMC calls tasks "agents," and parallel regions "accs." Workers are groups of hardware cores that can be partitioned if the hardware allows it (e.g., only using some cores of a Xeon Phi), which is a unique feature that allows OpenMC to take advantage of task parallelism by running more tasks on the subdivided hardware. OpenMC considers all tasks capable of being run in parallel, unless the user specifies dependencies. OpenMC's memory model is very similar to OpenACC and OpenMP's; all agents' serial code runs on the host in a shared memory environment (but can create private copies of variables as needed), and each agent can launch parallel kernels via accs and is responsible for managing communication and synchronization with its accs. Agent and acc directives can have data clauses, but OpenMC has no stand-alone data directives, like OpenMP and OpenACC do.

OpenMC has several directives that OpenMP and OpenACC have no equivalent for, and which could greatly improve their performance portability and readiness for exascale. OpenMC provides an `implemented-with` clause for acc regions that allows users to tell the compiler which programming model has been used, which is similar to the annotations used by the Uintah PPL (see Sec. 3.3) and could be modified to let users provide multiple implementations of a acc region, similar to OpenMP 5.0's `declare variant`. Agents and accs can also be annotated with a `priority` clause that lets the user give hints to the scheduler to determine when to run a region, for greater control and flexibility. Most interestingly, OpenMC provides a `time` clause for agents and accs that can be used for both performance monitoring and resiliency. If an agent or acc takes longer (or shorter, if the user chooses) than the specified time, OpenMC will issue a warning, so the user can monitor the machine for abnormal behavior that might indicate a soft fault or other hardware problem, or check how consistently a new kernel implementation outperforms an old implementation. Since exascale machines will be larger and more error prone than their predecessors, this kind of embedded support for performance monitoring will be increasingly important going forward.

- *Portability:* OpenMC only supports C as a base language, and OpenMP and CUDA for acc regions, which allows it to target many CPUs, Intel Xeon Phis, and Nvidia GPUs. This is the hardware used by TianHe machines, so it may be sufficient for its original purpose, but support for more architectures is desirable.

- *Performance:* OpenMC's performance was compared to native CUDA and OpenMP performance on Nvidia GPUs and Intel Xeon Phis. The GPU version was slower than the hand-tuned CUDA, and

[15] OpenMC was developed around the same time as OpenMP 4.0, and before an implementation of OpenMP offloading existed.

```
// triad in HSTREAM
#pragma hstream in(b, c, a, scalar) out(a)
    device(*) scheduling(4096)
{
     a = b + scalar*c
}
```

Listing 20: HSTREAM code sample.

this is likely due to OpenMC's lack of support for overlapping computation and communication, which is left for future work. On the Xeon Phi, OpenMC was able to get a higher percentage of hand-written performance, and splitting the Xeon Phi into multiple workers for more parallelism further improved performance.

- *Productivity:* Unfortunately, the parallel regions of OpenMC agents must be implemented in a target-specific language (e.g., CUDA when targeting GPUs), which does limit OpenMC's portability, but if OpenMC were to be integrated with more modern programming models, this would not be necessary. As it stood at the time, OpenMC required users to provide different worker directives and agent/acc implementations per architecture, and possibly modify data movement clauses. To port an OpenMC code designed for GPUs to OpenMC for Xeon Phis, users would have to modify about 10% of their code, which the authors of OpenMC believe to be acceptable, but for large code bases may still be intractable – 10% of 1 million lines of code is 100,000, after all. However, OpenMC does require far less code than hand-written CUDA, and only slightly more than OpenMP. The additional performance monitoring capabilities are also very intriguing and could greatly help productivity.
- *P3 Rating:* fair – OpenMC is a good prototype, but needs work with respect to requiring different worker directives and kernels. It is somewhat portable and gets good performance on TianHe machines, but is not as productive as it could be.

5.5 HSTREAM

HSTREAM [105] (code sample in Listing 20) is a new directive-based extension for stream computing, as opposed to traditional data parallel computing (similar to the GrPPI library, Sec. 3.2.1). Its programming model is based on a (possibly infinite) partially known stream of data, and contains a data producer, a data processor, and a data store (data writer), all of which can be overlapped to increase parallelism. HSTREAM is meant to be similar to OpenMP, and uses a source-to-source translator to target CPUs with OpenMP, Nvidia GPUs with CUDA, and Xeon Phis with Intel's Language Extensions for Offloading (LEO). Unlike other directive models, HSTREAM allows developers to automatically use all the available hardware in a node simultaneously by handling data and computation distribution (and, it seems, load balancing). To determine how computations should be divided, HSTREAM requires an additional platform description file that contains relevant data about all the hardware in a system, such as the number of cores, memory size, and so on.

Since HSTREAM is so new, the following descriptions of its portability, performance, and productivity will be more sparse than for other models.

- *Portability:* HSTREAM uses OpenMP as a CPU back end, so it can target most CPUs, but it uses CUDA and Intel's LEO to target GPUs and Xeon Phis, so its portability to other devices is limited.
- *Performance:* HSTREAM's performance hasn't yet been compared directly to OpenMP or CUDA, but its performance on the STREAM benchmarks using CPUs and GPUs combined is better than using either separately. This is a strong proof-of-concept for high performance automatic load balancing across devices, and is in many ways a continuation of the work begun by the Qilin compiler (see Sec. 6.1.1).
- *Productivity:* HSTREAM is approximately as complex as other directive based models like OpenMP or OpenACC. It comes with the added bonus of automatically using multiple devices, at the cost of providing a per-machine platform description file, which is a trade-off that could be worth it to some developers.
- *P3 Rating:* promising – HSTREAM is an interesting model, but hasn't been around long enough to judge.

5.6 Customizable Directives

In addition to the standardized and otherwise prepackaged directives described above, there have been efforts to allow users to define their own, specialized directives. Allowing users to define their own directives can give them more control over their application. This section will describe a framework that helps users define their own directives and an example of how user-defined directives can improve performance portability.

5.6.1 Xevolver

Xevolver [163] is a source-to-source translation tool built on top of ROSE [139] (see Sec. 6.3) that allows users

to define their own (parameterized) code transformations and directives to specify where those transformations should be applied. The main goal of Xevolver is to separate optimizations/transformations from application code – these transformations can be kept outside application code bases, which removes the need to keep platform-specific versions of code and consolidates knowledge about how to optimize for a given platform. Transformations can be shared across applications as well, which reduces the need to re-implement optimizations for each program.

The authors of Xevolver note that existing solutions for writing specialized code for each architecture aren't sufficient, since they generally involve keeping separate source versions or directly modifying application code. As an example, using C preprocessor macros to conditionally compile different code versions (even ones kept in the same file) can quickly devolve into "the so-called `#ifdef` hell" [163]. Users want specialized code, but they don't want to *write* specialized code, so having a set of transformations to pull from could be extremely helpful.

Performance tests of Xevolver-enhanced applications on various architectures confirm that adding these transformations helps performance. One motivation for creating Xevolver was to help users with applications optimized for vector machines port to other architectures (Xevolver also enables incremental porting). The authors demonstrated that it takes a set of non-trivial but consistent transformations to port these codes, and that transformations that help one architecture can be detrimental to another. Since these transformations don't actually modify the source code, they help make applications more performance portable.

Xevolver has been used to port part of a weather simulation to OpenACC [76] and migrate a numerical turbine code [160], among other things.

5.6.2 The CLAW DSL

The CLAW DSL [23] is an example of application specific, user-defined directives meant to enable performance portability for weather and climate models. CLAW is not based on Xevolver, but on the Omni source-to-source Fortran translator [110] (see Sec. 6.2) and uses a single-column abstraction to take advantage of certain domain properties of most climate modeling programs. The CLAW compiler outputs OpenMP or OpenACC annotated Fortran and is interoperable with normal OpenMP or OpenACC code to enable incremental porting.

Climate models have been using OpenMP and OpenACC for performance portability, but different architectures require different directives for optimal performance. The differences are consistent, however, so the DSL provides directives to abstract away these differences.

The CLAW port of a portion of one climate model outperformed the original serial implementation and a naive OpenMP implementation, and matched the corresponding hand-tuned OpenACC implementation. This proof of concept demonstrates how user-defined transformations can improve the performance portability of an application.

6 Source-to-source Translators

This section will discuss source-to-source translators specifically designed to improve the performance portability of applications. Source-to-source translators are considered here to be programs that transform one input language into another, but *do not themselves* compile it down to an executable; i.e., translators need another base compiler to work.

Several translators have been mentioned already, particularly when discussing the implementations of some directive based models in Sec. 5. This section will discuss how those translators work, as well as some other frameworks for code translation for performance portability. Note that, while previous sections have always contained a brief description of performance, portability, and productivity for each model, this section will not always do so, since for several translators this has already been discussed or is not applicable.

6.1 Early Translators

Source-to-source translators are not a new concept, and several that did not directly address performance portability were introduced that could nevertheless improve it. This section will discuss a couple of those.

6.1.1 Qilin

The Qilin compiler [95] was intended to solve the problem of load balancing computation on heterogeneous systems – in other words, to decide what fraction of computation should happen on the CPU vs. on the GPU. Even for a single application, the ideal fraction can change for different inputs, and, of course, different hardware. Qilin automates this mapping process by doing it adaptively at run time.

Qilin provides two APIs for writing parallel applications; the compiler doesn't have to extract parallelism, only map it to the hardware. The first API is the stream API, which provides data parallel algorithms (similar to skeleton libraries), and the second is the threading API, which allows users to provide parallel implementations in the underlying programming models, Intel TBB and CUDA. The compiler dynamically translates these API

calls into native code and decides a mapping using an adaptive algorithm.

This algorithm is based on a database of execution time projections that Qilin maintains for all programs it has seen. The first time Qilin sees a program, it builds a model of how that code performs when different percentages of work (per kernel) are run on the CPU vs. the GPU. For future runs (even on different problem sizes), Qilin refers to that model to find the mapping that will minimize run time. This adaptive mapping is always faster than GPU-only or CPU-only execution, and within 94% of the best manual mapping (at granularities of 10%). This kind of adaptive mapping can improve performance portability by providing good performance regardless of architecture changes, and it improves productivity by automating the process.

6.1.2 R-Stream

The R-Stream compiler and translator for C [147, 103] is somewhat unique among all the other models discussed here, in that it requires no source code modifications at all, not even annotations like OpenMP or Bones (see 6.4). R-Stream has been used to target several processors and accelerators, including IBM's Cell processor, ClearSpeed's processors, and Nvidia GPUs [88]. The compiler takes in unmodified C and outputs C with parallelizable portions in the chosen parallel back end. R-Stream is based partially on the polyhedral optimization model.[16] Its general compilation flow is as follows:

- Parse in C, translate it to an SSA IR.
- Run optimizations.
- Run analyses to determine which portions could be mapped onto an accelerator.
- "Raise" map-able portions into a polyhedral IR and optimize them under the polyhedral model to find parallelism.
- Lower map-able portions back to SSA IR.
- Emit non-mapped code as part of the master thread, and emit mapped code in target language (e.g., CUDA).

Interestingly, R-Stream generates the best parallel code when given what the authors call "textbook" C code – code without any optimizations or clever implementation strategies, just the basic algorithm as you might find in a textbook. This implies that, to create performance portable code, less is more, and simple, high-level expressions of algorithms can be more useful (in some ways, at least) than optimized versions.

[16] It is not important to understand the polyhedral model for this survey, but more information can be found in Griebl et al. [46].

6.2 Omni

The Omni compiler [110] is a source-to-source translator for Fortran and C (C++ is in development) that is used by XcalableMP, XcalableACC, and the CLAW DSL, among others. Omni is based on the idea of metaprogramming: it allows users to write code to transform their code. Not all compilers support every optimization, or can determine whether an optimization is safe, so metaprogramming allows users to transform their code so it is easier for compilers to analyze or to directly apply optimizations themselves.

Omni is composed of three main pieces: a front end, which parses in C and Fortran and turns them into XcodeML, Omni's IR (based on XML); a translator, which turns the XcodeML IR into Xobject (Java-based XML objects) and applies transformations to it; and a back end, which translates XcodeML back into C or Fortran which can be compiled by a regular compiler. Omni could theoretically support multiple "meta-languages" for users to define what transformations should be done, and where, including an Xobject-based meta-language, an XML-based meta-language, or a new DSL; however, since the Xobject-based meta-language was simplest to implement, Murai et al. chose to only implement that one. To define a transformation in their Xobject-based language, users must write a Java class that implements an Omni-specific interface, which describes the transformation to perform on the Xobject(s). To choose where that transformation is applied to their code, users add an Omni directive to their application. This is how XcalableMP and XcalableACC were implemented (see Sec. 5.3).

Unfortunately, this interface isn't terribly user-friendly, especially for non-compiler-expert users and domain scientists, although the vast majority of high-level compiler optimizations, including loop unrolling and array-of-struct to struct-of-array transformations, can be defined using it. Murai et al. realize this, though, and note that future work includes building a nicer interface. Perhaps another solution would be for compiler experts to write an open source collection of transformations that the HPC community could use and modify for their own purposes.

6.3 ROSE

The ROSE translator [139] is different from other translators listed here in that it is designed to specifically support analysis and optimization for source code *and* binaries. The ROSE front end supports many languages, including C, C++, Fortran, Python, OpenMP, UPC, and Java, as well as both Linux and Windows binaries. ROSE is meant to support rewriting large DoE applications for future architectures and programming models, as well as

research into new compiler optimizations, automatic parallelization, software-hardware codesign, and proof-based compilation techniques for software verification. ROSE has multiple levels of interfaces for working with source code ASTs, and many optimizations and analyses have been implemented to help users modernize their code.

One of the more interesting features of ROSE, from a compiler design perspective, is its IR, which is based on the Sage family of IRs. Even though ROSE supports a wide variety of languages with very different feature sets, around 80% of the IR is shared between all these languages, and only 10% is for special cases. That such a variety of languages can all be represented by one IR is very promising for compilers that wish to support multiple programming models. The topic of IRs is further discussed in Sec. 7.3.

Some projects that have been built on ROSE include Xevolver (see Sec. 5.6.1), a fault-tolerance research tool [92], a benchmark suite for data race detection [90], and more.

6.4 Bones

Bones [117, 116] is a source-to-source compiler for C, and targeting OpenMP, OpenCL, and CUDA, based on the concepts of algorithmic species and skeletons, similar to the skeleton libraries described earlier in Sec. 3.2.1. Nugteren et al. [117, 116] criticize existing compilers for parallelism as lacking in one of these three aspects: they aren't fully automatic and require users to change their code, they produce binaries or otherwise non-human-readable code, or they don't generate highly efficient code. Bones is intended to fix these problems by using algorithm species[17] and knowledge of traditional compiler optimizations to drive a skeleton-based translation process.

The Bones translation process goes like this:

1. Extract algorithmic species information from source code, either by hand or (preferably) using an automated tool like A-Darwin or Aset, and add species annotations to source code.

2. Pass annotated source code to Bones compiler, which uses species annotations to pick appropriate algorithmic skeletons.

3. Bones uses skeleton information to perform source code optimizations and outputs transformed C code.

Bones skeletons are not quite like the skeleton functions found in the libraries in Sec. 3.2.1, but are more like pieces of template or boilerplate code into which the compiler can insert pieces of user code; e.g., types are left as parameters, loop bodies are empty, and so on. Multiple species can map to a single skeleton, since species provide much higher granularity than skeletons, in general, and the number of species is near infinite.

Bones can do many different optimizations based on skeleton information, such as multi-dimensional array flattening, loop collapsing, and thread coarsening. The optimizations Bones does based on the skeleton information can also be target-dependent. For example, when targeting a GPU (either with OpenCL or CUDA), Bones can do data analysis on the species (kernels) identified to determine which data needs to be moved to the GPU, as well as optimize data transfers and synchronization events. One uncommon benefit of Bones' optimizations is that they need not be just permutations of the original code – Bones can add extra code to, e.g., ensure data accesses on GPUs are properly coalesced. However, Bones does still rely on the optimizations the underlying C compiler can do, since not all optimizations (like vectorization) can be written as skeletons, and not all performance relevant data (like register pressure) can be contained in species or skeletons.

- *Portability:* Bones can target OpenMP, OpenCL, and CUDA, which gives users access to the vast majority of CPUs and GPUs, however, since Bones only supports C, the number of codes that can use it is restricted.

- *Performance:* Bones is based partially on the polyhedral optimization and compilation model, but unlike other polyhedral compilers, Bones can be used on non-polyhedral code (this requires manual algorithm classification, which is not optimal). To test and validate Bones, Nugteren et al. [117, 116] compare it against Par4All [3] and ppcg [173], two other popular C-to-CUDA polyhedral compilers. On average, Bones outperforms both Par4All and ppcg, although Nugteren et al. do note that this is with Par4All and ppcg in fully automatic mode and with all optimizations turned on for Bones, for a more fair comparison, since Bones plus an algorithm classifier is fully automatic. If users wanted to spend more time tuning parameters for Par4All or ppcg, they could likely get much higher performance and possibly outperform Bones, but with all compilers in full automatic mode, Bones performs best. Bones even performs comparably to handwritten CUDA on the Rodinia benchmark suite.

- *Productivity:* One of the more common criticisms of skeleton libraries is that it is difficult and time-consuming for humans to select the correct skeleton,

[17] Algorithm species are similar to skeletons, but at higher granularity – species describe memory access patterns on all data structures in a particular loop nest. See Nugteren et al. [117, 116] for more.

and libraries can be error-prone when users choose the wrong skeleton. Bones significantly reduces the chance for human error by automating this process using either Aset or A-Darwin. However, non-polyhedral codes may still need a human to add algorithmic species annotations; Nugteren et al. believe their annotation structure is "descriptive and intuitive," so this process should not be difficult, but users unfamiliar with their species concepts and classifications may disagree.

Since Bones produces human-readable OpenMP, OpenCL, or CUDA, it can improve productivity by providing a baseline implementation that expert users can then tune. Bones is also easy to extend with more skeletons, and users may be able to use this to optimize their code as well.

– *P3 Rating:* fair – Bones is only good at enhancing performance portability of affine C programs, but it is very good at doing that.

6.5 OpenACC to OpenMP

There have been several proposals for translating between OpenACC and OpenMP, including those from Pino et al. [138], Sultana et al. [161], and Denny et al. [32]. Many current machines have only one of OpenMP or OpenACC, or have a poor implementation of one but a good implementation of the other, so being able to translate between them would alleviate the problems this causes. However, because of semantic differences between OpenMP and OpenACC, mechanical translation from OpenMP to OpenACC is generally not possible or advisable [178].

OpenMP was designed when most machines used multiprocessor architectures, and processor vendors wanted to provide a unified interface for programming multiprocessors. Therefore, the meaning of each OpenMP directive was very important and the OpenMP specification has a detailed, prescriptive definition of what each one means. OpenACC, on the other hand, was designed when there were many diverse, heterogeneous architectures, so the OpenACC specification decided to leave many more (architecture-specific) choices to the compiler and only provide a descriptive definition of what each directive does/means. OpenMP also has many synchronization primitives and atomics, which OpenACC does not; some OpenMP concepts simply cannot be expressed in OpenACC. Going from OpenACC to OpenMP, though, is possible, as this direction doesn't have these problems.

While there is a simple, one-to-one mapping for OpenACC and OpenMP data directives, the same is not so for compute directives. Even though many OpenMP directives seem similar to OpenACC, they have different definitions and meanings. For example, OpenMP can't parallelize within a thread with only `parallel for` unless the user specifies it (this is why OpenMP needed to add the `simd` directive), while OpenACC can – the `parallel loop` directive guarantees there are no data dependencies across iterations. This means that translating loops from OpenMP to OpenACC is non-trivial and requires data dependence analysis, while going the other way is simple and always valid.

The difficult part of writing an OpenACC to OpenMP translator is adding the right prescriptive OpenMP keywords to loop nests (to ensure the loops are properly mapped to hardware). To make matters more interesting, these keywords may be different for each particular device, as the next sections describe.

6.5.1 Sultana et al.'s Translator

Sultana et al. [161] made a prototype tool for automatically translating OpenACC to OpenMP, focusing on translating for the *same* target device, e.g., Nvidia GPUs. They demonstrated that some parts of the translation are indeed mechanical, but others require more work. One of their goals was to provide a deterministic translation, i.e., the same set of OpenACC directives will always be translated to the same set of OpenMP directives. To this end, they created a deterministic set of translation rules for each OpenACC data and compute directive, as well as deterministic rules for adding `gang`, `worker`, and `vector` clauses to OpenACC loops that did not already possess them. This was necessary because, while OpenACC allows the compiler to decide how to parallelize nested loops, OpenMP requires more direction. In addition, OpenMP has no equivalent for OpenACC's `seq` directive, so the translator needed to remove directives from these sequential loops, and redistribute any reductions or private clauses on these loops.

However, Sultana et al. ran into problems with changing compilers when going from OpenACC to OpenMP (specifically, OpenMP offloading), including large performance differences. Some of this is expected, as both OpenMP and OpenACC leave quite a bit up to the compiler. They used PGI as their baseline OpenACC compiler and Clang as their OpenMP compiler, and noticed that the kernels PGI generated were almost always faster than the kernels Clang generated. This could be because, at the time, PGI's OpenACC implementation was much more mature than Clang's OpenMP implementation, but it does show that compiler implementations of these models can have significant impact on overall performance.

Furthermore, as future work, Sultana et al. believe that they may need to device specific translation rules, echoing Wolfe [178]. For example, the rules for an Nvidia GPU will probably not be optimal for a Xeon Phi, and vice versa.

6.5.2 Clacc

A more recent project, Clacc [32], desires to build a production-quality, open-source OpenACC compiler, built on Clang, and to generally improve OpenACC and GPU support in Clang. Clacc translates OpenACC into OpenMP to take advantage of existing OpenMP support in Clang. The authors of Clacc note that this both improves code portability between machines without good OpenACC or OpenMP implementations (as Sultana et al. [161] also describe), but also opens up possibilities for using existing OpenMP tools to analyze OpenACC. As OpenACC is much newer than OpenMP, tool support for OpenACC is less mature, and this could significantly benefit OpenACC developers. Many developers are also worried that OpenACC will be soon be subsumed into OpenMP, which is much more popular, and that porting to OpenACC will therefore be a waste of effort, so being able to translate from OpenACC to OpenMP easily and automatically would ease their concerns.

To implement OpenACC support in Clang, the Clacc developers first translate OpenACC code to an OpenACC AST inside Clang, then create shadow OpenMP sub-trees, which can be compiled to an executable or used to output an OpenMP AST (or source code) equivalent to the OpenACC input.

While their implementation is very new and doesn't yet fully support GPUs or languages other than C, the Clacc team has been able to get performance comparable to PGI's on multi-core, with the exception of one benchmark, as long as `gang`, `worker`, and `vector` clauses are specified.

7 Compiler Support

This section will discuss support for parallelism and performance portable features in compilers, as well as the language standards these compilers implement.

7.1 Current State of Support

Most general-purpose languages used in HPC have little to no support for parallelism, although, as described in Sec. 7.2, recent additions to both C++ and Fortran have been changing this. However, implementation of these standards has been slow in coming, which means different compilers often offer different levels of support and performance. In particular, support for lambda functions, generic/templated functions, and optimizations that work across these and through calls to performance portability abstraction layers like RAJA and Uintah's PPL [59] would be a great help to performance portability efforts. Unfortunately, these are some of the standard additions that have been most inconsistently implemented. The implementations are much better than they were when Hornung and Keasler called for better support in C++ [60], but there is still room for improvement.

Sadly, these compilers were first designed when most code was sequential and they have no easy way of internally representing parallelism. Most compilers turn parallel constructs into function calls in their internal representation (a process called outlining, see de Supinski et al. [27] for an example of this with OpenMP), and this causes several issues for users. The most important problem is that the compiler has no way to reason about parallelism and must make conservative choices, which often means not applying an optimization even when it is valid. It also contributes to the inconsistency described previously, and can mean that code changes are necessary to get the same performance out of different compilers. There have been several efforts lately to add parallelism to existing IRs, which are described in Sec. 7.3.

7.2 Language Standards

Language standards define exactly what a programming language is and does, and are very important for traditional portability. If compilers implement a standardized language, users know that, no matter which compiler they choose or what machine they run on, they can be reasonably confident their code will work as intended. Compilers often implement their own extensions to a language, or might have a more efficient implementation of standard features, but as long as users restrict themselves to the standard, they can know their code will compile and run correctly.

Adding features into a standardized programming language is a long and difficult process that begins when a developer proposes a new feature to a language standards committee. If the committee likes the idea, they iterate with the developer on design choices and wording (sometimes for years) before finally presenting it to the committee's full membership for approval. If the feature proposal is approved, it goes into the standard, and compilers that want to support the standard must implement it. Implementation and testing is also a long process, so the time lag between a feature first being proposed and a working implementation becoming available can be many years.

As mentioned previously, most languages have little or no explicit support for parallelism, since parallel programming is still comparatively new and the standardization process takes so long. Performance portability is an even more recent concept with even less language support. This section will discuss efforts by the C++ and Fortran standards to add explicitly parallel features and even some performance portability features.

```
squared = map (\x -> x * x) nums
```

Listing 21: A lambda in Haskell.

```
std::for_each(nums.start(), nums.end(),
    [](int& x){ x = x * x; });
```

Listing 22: The lambda from Listing 21 in C++.

```
template <class MyType>
MyType add(MyType a, MyType b) {
 return a+b;
}

float x = add<float>(4.2, 3.14);
int y = add<int>(1, 2);
```

Listing 23: A simple C++ template function.

7.2.1 C++

Lambdas. Lambda functions are a concept borrowed from functional programming, where higher-order functions can take other functions as arguments. Lambdas are (usually short) functions that are unnamed and written in-line, which is why they're also called "anonymous" functions. Listing 21 shows an example of a lambda function in Haskell, a functional programming language. This example applies the `map` higher-order function to a lambda that takes a number (and squares it) and a list `nums` to produce a list of squared numbers. Listing 22 shows the same operation in C++, assuming `nums` contains integers.

Lambdas are one of the most commonly cited C++11 features that enhance performance portability, but their true purpose is increasing developer productivity within other performance portable models. Many models, particularly libraries like RAJA, SkePU, and Kokkos, benefit a great deal from the capability to specify functions inline. Their other option is to use C++ functors[18] (function objects), which are much more verbose, difficult to understand, and must be placed elsewhere in the code, possibly far away from where they're used. Numerous papers have noted their usefulness in writing clean, performance portable code, especially as implementations have improved [60, 61, 100, 51, 41, 59, 133, 9].

Templates. Templated functions and classes allow developers to write generic code that works with many types, as long as those types support any operations used. Primitive types in C++ function and class definitions are not interchangeable, despite the fact that the code to operate on each primitive is often exactly the same. Templates fix this. Listing 23 shows a short template function to add two numbers and a couple invocations of that function.

The C++ Standard Template Library (STL) was first added in C++98, and contains many templated classes and functions, including containers, like `std::vector` and `std::map`, and algorithms, like `std::sort` and `std::binary_search`. As of C++17, the STL also has several parallel algorithms, such as `std::reduce` and a parallel version of `std::for_each`, that are differentiated from their sequential counterparts by an execution policy; however, there are few implementations of the parallel STL.

Many C++ libraries (see Sec. 3) have taken advantage of the new templating abilities to provide generic parallelism to their users. In fact, the SkePU developers redesigned their entire interface to take advantage of templates and improve the type safety and usability of SkePU [41]. In some places templates enabled them to reduce their internal runtime's code size by 70%. Other libraries like PHAST [132] and Muesli [40] use templates to provide users with generic data containers and parallel algorithms.

Containers. Containers and data structures deserve separate discussion outside of templates in general due to the sheer number of libraries that provide their own implementations of the exact same generic vector/matrix classes [38, 155, 159, 96, 14, 39, 26, 40, 41, 22]. Libraries that provide so-called "smart containers" that automatically handle memory transfers between the host and device in particular each seem to have reimplemented the wheel with identical lazy copying mechanisms. If the standard library had an implementation of this, it would consolidate a great deal of knowledge and allow for more interoperability between various programming models.

The Kokkos developers have taken note of the this and have been working to add their View data container (see Sec. 3.2.2) to the C++ standard library as `std::mdspan` [58]. The proposed container would provide a multidimensional view on a contiguous span of memory, while not truly owning that memory. The `mdspan` object would contain a mapping from a multi-dimensional index space onto a scalar offset from a base memory address which would be used to access memory, but allow other memory management activities to happen elsewhere. This would let applications interact with memory without knowing or understanding the underlying layout of memory. For example, the developers could experiment with row-major versus column-major or tiled data layouts in memory by changing the `mdspan` object's mapping function without

[18] Not to be confused with the concept of functors in category theory and functional languages.

changing anything in the rest of their code. This would be incredibly helpful for performance portability, since it would move programs closer to single-source implementations that can be configured for multiple architectures. There are many other benefits to a standardized container view like the proposed `mdspan`, such as better integration with other libraries, combining static and dynamic extents, and array slicing, which are fully described by Hollman et al. [58]. Hollman et al.'s `mdspan` implementation has very low or negligible overheads, although they noted that compiler version (including the same version of the same compiler with different options enabled) did impact overheads, since different optimizations were applied, which is another point in favor of standardizing implementation of these features as described earlier in Sec. 7.1.

Just-in-Time Compilation (JIT). Just-in-time compilation is a technique that, as the name implies, waits until the last minute to compile code into an executable. There are multiple ways to do this, but for the compiled languages most commonly used in HPC, the general process is to link the main executable with a compiler library, and when a function to be JIT-ed is reached, invoke the compiler on that function, then put in the arguments and run the result. A common optimization to this process is to cache JIT-ed functions to avoid repeated compilation overheads.

JIT compilation is one way to help compilers generate faster code. It is often difficult for compilers to decide if an optimization is safe or helpful at compile time, when some information (like loop trip counts or pointer aliases) is unknown, but at run time, this information can be made available and the compiler can make better choices. To get any real improvement, though, users need to decide whether the overhead of compiling parts of their application at run time is offset by faster execution times.

Some languages and translators, like PACXX [51] and Qilin [95], have used JIT techniques to provide run time optimization and specialization for their users' code. However, a recent project, ClangJIT [42], has been moving to add JIT capabilities to the C++ standard so users can benefit from JIT specialization and optimization of their code without relying on an external library. ClangJIT proposes adding some syntax to the language so users can specify which functions they want JIT-ed, and goes on to demonstrate that the specialization provided by JIT-ed functions can result in shorter benchmark execution times. They also show JIT can reduce compile times for heavily templated full applications (without impacting the execution time!), since only the versions that are actually used for a particular run are compiled.

7.2.2 Fortran

Do concurrent. Fortran's `do concurrent` construct [140] was first added to the standard in Fortran 2008 to provide users with language support for parallel loops. The `concurrent` version of Fortran's traditional do-loop indicates to the compiler that there are no data dependencies between iterations, so each iteration can be done in parallel (similar to OpenACC's `independent` loop modifier). This enables the compiler to do many optimizations, such as vectorization, loop unrolling, and automated threading, that otherwise might be applied if data dependencies were unclear. The `contiguous` attribute (also added in Fortran 2008) is complementary; it denotes that an array occupies a contiguous block of memory and allows the compiler to perform optimizations with that knowledge. Features such as this allow compilers to be more consistent and aggressive in which optimizations they apply, which is good for performance portability.

While C++ has added support for performing parallel operations on arrays with the parallel STL algorithms, it's unclear how well compilers can optimize those operations since they happen inside library calls, and adding a generic parallel for-loop (similar to Kokkos/RAJA parallel loop abstractions) to both C and C++ would help both languages be more consistent in their optimizations as well.

Coarrays. Coarray Fortran was first proposed in 1998 as an extension to Fortran 95, but was not adopted into the specification until Fortran 2008 [140, 104]. Coarrays are a PGAS concept for sharing data, where a single array is allocated across multiple processes, and all processes can access each other's coarray data. In Fortran terms, a collection of process "images" (all executing the same code in an SPMD manner) declare a coarray with the syntax `<type> ::  a(n,m)[*]`, which means that array `a` is allocated across all images with local array dimensions `n x m`, and codimensions (the logical dimensions of the set of images the coarray is allocated across) specified in brackets. Codimensions can be used to specify logically multi-dimensional groups of processes, e.g., for a 3-codimensional case, each process can be thought of as being a point in a cube, potentially with neighbor images above and below, to the left and right, and to the front and back.

Additional coarray features added in Fortran 2018 [141] (in response to criticisms from Mellor-Crummey et al. [104], among others) include the introduction of teams of images and the ability to allocate local coarrays among only the images in the team, instead of across all images as global variables.[19] While there don't appear to be any

[19] Teams also have the ability to handle failed images, which is extremely important for resiliency going forward.

implementations in major compilers yet, the OpenCoarray library and wrapper [153] appears to be a mostly complete implementation that can be added to any compiler. OpenCoarray can be set to use either MPI or OpenSHMEM as a communication back end, giving users flexibility.

Adding coarrays to the language specification enables performance portability by standardizing a parallel programming feature. As OpenCoarray demonstrates, the back end can be switched out to use a different parallel programming model without making changes to user code, bringing Fortran closer to a single-source solution for shared data.[20]

7.3 Parallel Internal Representations (PIRs)

Internal representations (IRs) are the data structures compilers use internally to represent and reason about users' code before translating it into assembly or machine language. All transformations and optimizations the compiler performs are done on the IR. Some compilers, such as Habanero-Java (see 7.3.4), use multiple IRs that represent different levels of abstraction between the high-level language being compiled and the machine's instruction set, since many optimizations are easier to reason about at higher or lower levels of abstraction.

Ideally, an IR would be able to express all operations of the compiler's target architecture(s) and all concepts in the high-level language(s) the compiler accepts, and would additionally be independent of architecture and language, but in reality this is often not the case. Features are added to language standards, and compiler writers either can't or won't extend their IR, so they make do with what they have. This has been especially true for parallelism features, and this section will discuss several attempts to add support for parallelism into IRs.

7.3.1 LLVM

The LLVM compiler infrastructure is a group of projects, including the Clang compiler, LLDB debugger, LLD linker, and an OpenMP runtime library used by Clang. LLVM's primary feature is its IR, which has become widely used since its initial introduction. LLVM IR [81] was designed to be language independent and useful at all stages of program compilation, including linking and debugging. The IR itself is very low level and based on a RISC-like instruction set with type information, a control flow graph, and data flow graph via static single assignment (SSA) value representations. Due to its popularity, LLVM has been a common target for efforts to add parallelism to IRs.

[20] There have also been several implementations of coarrays for C++ [164, 111, 66], but there has yet to be any attempt to add them to the standard. Unified Parallel C (UPC) [172] adds similar PGAS features to C, but not with coarrays.

LLVM for Parallelization, Vectorization, and Offload. Tian et al. [167, 168] proposed additions to LLVM's IR to enable more efficient parallelization, vectorization, and offloading, primarily for OpenMP and other directive-based models. Their first proposal [167] did this by adding four new intrinsics to LLVM that represented basic directives, directives with qualifiers, and directives with various numbers of qualifiers with operands. Their second proposal [168] trimmed this down to two intrinsics, `llvm.directive.region.entry()` and `llvm.directive.region.exit()`, to represent when a region is controlled by directives. These intrinsics can be attached to LLVM OperandBundles that contain data on which directive(s) they represent and any operands, and entry/exit pairs are matched with an LLVM Token that the entry creates, which is passed as an argument to the exit. Singleton directives (e.g., `omp barrier`) are also represented by an entry/exit pair with no code in between to eliminate the need for a third intrinsic. After the front end adds these intrinsics, parallelization and offloading are handled by transforming groups of basic blocks into "work regions," which are single-entry single-exit sub-graphs of the program's control flow graph that can be put through their new optimizations on their own and then mapped onto OpenMP runtime calls (and possibly offloaded). Vectorization is handled similarly.

Tian et al.'s second proposal better reflects their design goals of enabling more and better optimizations, minimizing the impact on existing LLVM infrastructure, and providing a minimal, highly reusable threaded code generation framework. They implemented their additions and modified existing LLVM passes to work with them in Intel's OpenCL compiler, to see how the new IR structures would affect existing FPGA simulator workloads, and found that very few changes were needed to existing LLVM code. Their additions gave the FPGA simulator workloads good speedup – up to 35x for one workload – demonstrating the benefits of extending parallelism support. This clearly improves performance, but can also improve productivity by removing the need for developers to do these kinds of optimizations to their code; they can just let the compiler do it.

Tapir. The Tapir project [146] added fork-join parallelism to LLVM IR. The authors did this by adding three instructions: `detach`, `reattach`, and `sync`. The `detach` instruction is the final instruction for a block, and takes references to two blocks, a detached block and a continuation block, as its arguments. These two blocks can be run in parallel; the detached block must end with a `reattach` instruction that points to the continuation block. The

sync instruction enables synchronization by dynamically waiting for all detached blocks with its context to complete. To spawn an arbitrary number of tasks, the compiler can nest detaches and reattaches, as long as certain properties are obeyed. These properties (described fully in their paper) ensure that the parallel representation maintains serial semantics; this allows serial optimizations to "just work" on parallel IR with minimal or even no modifications. In fact, most of the code for Tapir was additions to the LLVM back end, lowering Tapir IR into Cilk calls, and not modifications to existing code. The authors also added some new optimization passes, such as parallel loop scheduling, synchronization elimination, and "puny task" elimination. The authors evaluated the correctness and performance of their implementation on 20 Cilk benchmarks.

Tapir has also been used to implement OpenMP tasks [154]; the task portion of Clang's OpenMP implementation was replaced with code to generate Tapir IR, and then lower it into Cilk API calls, using the back end from [146]. The authors compared their OpenMP task implementation to the implementations in GCC, Intel, and unmodified Clang using the Barcelona OpenMP Task Suite. While their incomplete implementation caused problems for one benchmark (they had yet to implement critical sections or atomics), the Tapir implementation matched or exceeded, sometimes by 10x, the other three implementations in all cases. Tapir does especially well on benchmarks where the overhead to work ratio is high, since Tapir spends far much less time in overhead than the other three (30% compared to 80+%). They also did experiments to see whether their new optimizations improved performance at all, which they did. This implementation of OpenMP is interesting for performance portability, since it implies the possibility of having multiple high-level parallel languages targeting multiple back ends – a single IR can represent multiple programming models (Cilk and OpenMP) and still achieve good performance.

7.3.2 SPIRE

The Sequential to Parallel Intermediate Representation Extension project (SPIRE) [70] provides a methodology for turning sequential IRs into parallel IRs that have representations for both control parallelism (e.g., threading) and data parallelism (e.g., vectorization). To do this, SPIRE adds execution and synchronization attributes to the IR representations of functions and blocks. The execution attribute describes whether the function/block should be run in parallel, and if so, how, and the synchronization attribute describes how that function/block-/instruction behaves in relation to others, and includes support for atomic operations, spawning new units of execution, and barriers. SPIRE also adds an event primitive type which acts like a semaphore, for finer-grained synchronization, and two new intrinsic functions, send and recv, to represent point-to-point communication. The authors of SPIRE also provide operational semantics for these additions, both as a proof of correctness and to systematically define how they work.

In their original work [70], Khaldi et al. applied their methodology to the IR of PIPS, a source-to-source translator, and showed that it could represent parallelism concepts from a wide variety of models, including Cilk, Chapel, OpenMP, OpenCL, and MPI. Later, they applied it to LLVM IR and used OpenSHMEM to evaluate their implementation, focusing on optimizing the performance of shmem_get and shmem_put [71]. To transform LLVM, they added an execution attribute to functions and blocks and a synchronization attribute to blocks and instructions, similarly to their original paper. They also added an event type to act as a semaphore, and send and recv intrinsics. In an addition to their original work, which did not directly address programming models based on separate address spaces, they added a location attribute to values and identifiers and an expression attribute to load/store operations. The location attribute describes whether data is resident locally, must be fetched, or is shared. They decided to represent shmem_get and shmem_put as load and store operations, so the expression attribute can be used to mark loads and stores as volatile when they might be waiting on a remote SHMEM operation (this tells the compiler some optimizations might not be safe).

As a proof-of-concept for how their additions improve the compiler, Khaldi et al. modified some existing LLVM compiler passes to optimize OpenSHMEM get and put operations, primarily turning single-element gets and puts into bulk operations. They did in fact get significant speedup, especially when they were able to turn the entire series of data transfers into a single transfer. This is another point in favor of adding representations of parallelism to IRs – current compilers can't reason about data transfers like this because they're treated as function calls, which can't be optimized. If compilers could reason about these kinds of optimizations, users wouldn't have to and could spend their time on development instead of performance tuning.

7.3.3 Lift

The Lift IR [157] is somewhat different from the IRs discussed previously in that it is a functional IR. The authors of Lift were deliberately targeting performance portability for their compiler, which takes in their functional language and outputs OpenCL which has been optimized for a particular architecture [158]. The functional language

is reminiscent of skeleton libraries – in fact, it is in may ways the continuation of one of SkelCL's authors' work there. Lift attempts to remedy the problems SkelCL had with OpenCL's performance portability issues.

Lift has several high-level operations (including map, zip, reduce, split, join, and stencil, which was added later [49]) that take in a user-defined specialization function. This high level language is translated into their functional IR, lowered using rewrite rules to a second, completely interoperable IR with semantics closer to OpenCL, and finally compiled to OpenCL.

The Lift project is interesting because of the capabilities enabled by their rewrite rules. The rewrite rules allow them to explore the "implementation space" of high-level code by incrementally applying them, compiling to OpenCL, running performance tests, and repeating until they've found a semi-optimal implementation. Indeed, their work is in some ways proof that OpenCL is *not* performance portable, because when compiling for an Nvidia GPU, an AMD GPU, and an Intel CPU, their compiler generated different code for each, and got good performance on each (matching and sometimes exceeding the state-of-the-art at the time) [158]. As future work, they want to replace their current rewrite rule search algorithm with a machine learning model to decrease very high compile times.

One of the main goals of Lift (which other languages and IRs could learn from) is to decouple exploiting parallelism in code from mapping parallelism to hardware, which is also a goal of performance portability – make the main code base architecture-independent, so it can be modified to run well on any architecture.

7.3.4 Other PIRs[21]

PLASMA. The PLASMA IR [130] was designed to extend an exiting IR with IR-level support for SIMD parallelism (vectorization) by abstracting away details for any specific SIMD implementation, like SSE or CUDA. Their goals were to abstract away data parallelism granularity (vector length), vector-specific instructions (i.e., it must also be possible to mark vector-only instructions like a vector shuffle as parallelizable), and parallel idioms (like map and reduce). To do this, they include an operator abstraction in their IR that distinguishes between operators that only accept a single element and true vector-only operators that must take a vector. To vectorize a single-element operation, their IR contains distributors that take a single-element operation and vector(s), and map the elements of the vector(s) onto the operation according to the distributor type. For example, to do a parallel/vectorized add of vectors `a` and `b`, the compiler passes `a`, `b`, and the scalar `add` operation to the `par` (parallel) distributor: `c = par(add, a, b)`. PLASMA is also able to optimize these vector operations by composing them, so fewer/no intermediate vectors need to be created.

To test their IR, the authors implemented a source-to-source translator for C+PLASMA extensions to pure C, with either SSE3 or CUDA for vectorized code. They compare performance to state-of-the-art vectorized libraries for both CPU and GPU, and while their performance does vary by platform, it is generally comparable to libraries for each. With respect to performance portability, it's interesting to note that they were able to use the same vector abstraction for two very different architectures (SSE and CUDA) and get reasonable performance, which implies the possibility of a single-source code being able to be translated for multiple vector architectures.

Habanero-Java. Habanero-Java (described in Sec. 4.3) uses three separate IRs at various levels of abstraction [185]. The different IRs each lend themselves to different types of optimizations, which makes the process of lowering through each IR and to machine code much easier on compiler writers. Their high-level PIR (HPIR) has a tree structure (similar to a traditional AST) which enables simpler parallelism optimizations, such as elimination of redundant `asyncs`, strength reduction on synchronizations, and may-happen-in-parallel analyses. The middle-level PIR (MPIR) linearizes HPIR by transforming some control flow structures and eases data flow analyses. The low-level PIR (LPIR) lowers MPIR to add runtime calls for concurrency, making runtime optimizations easier, and transforms all IR constructs into a standard sequential IR that can be used for traditional compiler optimizations. Using multiple IR abstraction levels like this allows the compiler to optimize for parallelism while also reusing sequential optimization passes, which is a process other compilers could utilize to improve the consistency of their optimizations, and hence the performance portability of the code they produce.

INSPIRE. The INSPIRE IR [67] (no relation to SPIRE [70]) is a functional IR like Lift but for traditional HPC languages, like C and C++. INSPIRE has two parts: the core IR, including primitives for types, expressions, and statements, and extensions built on these. INSPIRE's parallel control flow is based on a thread group model, where thread groups collectively process a "job," which is a collection of local variables and a function for the threads to execute. The IR has primitives for threads to spawn and merge with other threads, and communicate with each other. It also has primitives for distributing data and work among threads. INSPIRE is different

[21]Note: not an exhaustive list, but a selection based on how recent, novel, and relevant each IR is.

from the other IRs listed here in that it doesn't process single translation units, it works over the entire program at the same time, enabling a number of whole-program optimizations. The authors' implementation uses Clang's front end for OpenMP, Cilk, OpenCL, and MPI, and they run performance experiments to verify that their base compiler (without any new optimizations) produces code that performs similarly to a normal C compiler, as this is how developers researching new optimizations would use it and they want the initial state of any code to be the same across compilers. For all the sequential and OpenCL inputs, this does hold true, but for a few OpenMP inputs (mostly using OpenMP's tasks, which their IR is very well suited for), the performance improves significantly, just by being translated through their IR. This demonstrates once again the importance of using appropriate IR constructs for compiling high performance (and performance portable) code. It can make all the difference for optimizing parallel constructs.

7.3.5 MLIR

The variety of home-grown IRs described above all have many features in common, and a great deal of effort could have been saved if their authors all had some sort of shared IR-building library to work with, especially if that library came with built in debugging support and parser logic. The MLIR project [82] aims to solve exactly these problems, along with others in the compiler community, such as the difficulty of writing high-quality DSL compilers and many mainstream compilers defining their own IRs on top of LLVM. Instead of implementing many better compilers, it seems easier to instead implement a better compiler building infrastructure.[22]

MLIR is designed to standardize SSA IRs and includes built in support for documentation generation, debugging infrastructure, and parsing logic, among other things. MLIR users can define their own operations, types, rewrite patterns (similar to Lift's rewrite rules), and optimization passes on top of MLIR's infrastructure, while keeping their IR extensible for the future. MLIR's design principles include having minimal built-ins in their IRs, allowing users to customize anything and everything, progressive lowering, maintaining more high-level semantics, IR validation, and source location tracking and traceability. All of these features would greatly improve modern compilers, but one of their most interesting features, and the one that will probably do the most to support performance portability, is *IR dialects*.

IR dialects are a way of logically grouping operations, attributes, and types into sets within an IR, similar to namespaces in C++. Dialects make IRs more flexible and modular, reduce name conflicts, and allow for interesting interactions and reuse possibilities by mixing IRs. For example, with an accelerator dialect, host and device code could be generated and optimized together instead of separately, or an OpenMP dialect could be reused across many source languages instead of reimplemented for each. Both of these would improve performance portability, since compilers could work with a consistent (standardized?) IR, instead of their own home-grown IR. MLIR optimization passes are designed to work with any dialect, even ones they weren't intended for. Dialects and IR operations can register properties (like "legal to inline" or "no side effects") with each other to help this process, or the pass can treat operations from unfamiliar dialects conservatively. MLIR has already been adopted by several groups, including TensorFlow, a polyhedral compiler, the Flang compiler, and various DSLs, demonstrating that this is something both industry and academia are very interested in and find useful.

[22] The authors of Tapir [154] agree that it would be helpful to move the HPC community towards a single, standardized IR. By having standard, common IRs, we avoid duplicating effort and can have multiple programming models work together easily.

8 Discussion

This section will give a high level review of how all the models described above support the various aspects of performance portability, beginning with a brief summary of each section, and continuing with a discussion of how each model can be useful and how well each type of model supports portability, performance, and productivity.

Tables 2 and 3 give a summary of how well each model meets some criteria for performance, portability, and productivity, and assign each model a score based on how well it meets these criteria. Scores are calculated by simply summing up "points" according to these rules:

- ×= 0 points
- ○ = 0.5 points
- ✓= 1 point
- High (when "High" is good, as with performance) = 2 points
- Moderate (good) = 1 point
- Low (good) = 0 points
- High (when "High" is bad, as with SLOC added) = 0 points
- Moderate (bad) = 1 point
- Low (bad) = 2 points
- N/A = 0 points

– Combined values (e.g., Low/Moderate) are worth 0.5 points in the appropriate direction

– Varied or Unknown values will be treated as "Moderate"

These scores are intended to provide a (very) rough comparison of the models discussed here, *not* to provide a new metric for performance portability or anything similar. Many readers may disagree with this scoring system, or care about different qualities. (As can be seen, even the qualitative assessments from earlier in this same paper do not always agree with the numeric scores, as criteria were weighted differently, and human judgement was involved.) Some may emphasize Fortran support over GPU support, for example, or may not care how many lines of code are added to their program. Those readers are welcomed to devise their own scoring system based on their own wants and needs. Again, the intent here is to provide a high level overview and comparison, not definitively say one model is better than another.

8.1 Summary

This section gives a brief summary of the content of the previous five sections.

8.1.1 Libraries

There are two main classes of libraries designed for performance portability: skeleton libraries and loop-based libraries. Skeleton libraries provide higher-order functions based on common parallel patterns users can customize to run computations in parallel, while loop-based libraries abstract away iteration spaces and data structures to run loops in parallel. Application-specific performance portability layers can further insulate users and application code from the details of parallelism and changes in the underlying performance portability models.

8.1.2 Languages

While the barriers to entry for parallel languages are higher than those for other models, some languages have been successful. All of these languages are primarily task-based, but most include data parallelism as well, since modern hardware relies heavily on vectorization for its performance. Some of these languages are very high-level (e.g., Chapel and HJ) while others are less so (e.g., DPC++).

8.1.3 Directives

The two most popular directive-based models are OpenMP and OpenACC, which fall on opposite ends of the prescriptive vs. descriptive spectrum, although both have been moving closer to the middle. Other directive-based models, like OpenMC and XcalableACC, haven't been nearly as successful, but have unique features OpenMP and OpenACC could learn from. There are also tools that let users define their own directives, to give them more control over what transformations and optimizations happen to their code.

8.1.4 Translators

Many of the other models in this paper were built on source-to-source translators like Omni and ROSE, which were both designed to enhance user productivity by helping users transform their code into more performance portable versions. Other translators, like Bones and Clacc, were designed more as compilers than translators, but also provide translation capabilities so users can further tune their code before compiling it.

8.1.5 Compiler Support

Most compilers currently have limited internal support for parallelism, but this is changing. Language standards, particularly for languages like C and Fortran that are popular in HPC, have been adding native support for more parallelism constructs, and compilers will need to follow suit to be able to effectively analyze these codes. Many compilers are attempting to move to PIRs, and MLIR in particular is an exciting project, since it opens up a new world of IR building and composability.

8.2 Use Cases

Each model described here was meant to support a different use case; it's very difficult to make one programming model that does everything, and the effort usually isn't worth it when there are so many other models that already exist. This section will discuss the options users have for the three main application-developing use cases.

8.2.1 Porting Legacy Applications

There are still many legacy HPC applications in use today (e.g., weather simulations) that were first written in C or Fortran decades ago, and these applications need to be ported to take advantage of hardware advances as well. However, wholesale rewrites of these programs are generally not possible, since they may contain millions of lines of code, and years of work have gone into verifying their correctness and adding a wide variety of features. Developers don't want to throw all this work away to start again with a new programming model, especially one they might have to abandon a few years later when the next great thing comes out.

Fortran developers in particular have a hard time; C developers can port to C++ (a non-trivial task, but also not a complete rewrite) and instantly gain access to a wide variety of libraries and languages for performance portability. Even if porting to C++ isn't an option, there are still some libraries and translators (like targetDP and Bones) that support pure C. Fortran doesn't have these options. Fortunately, some directive-based models, like OpenMP and OpenACC, have deliberately included support for Fortran, and directives are minimally invasive and allow for incremental porting, so they are a good option for porting legacy code to modern architectures.

8.2.2 Porting Newer Applications

Newer HPC applications are much more likely to use C++, so they have more options than older C and Fortran codes. There are many C++ libraries for performance portability, and even some languages (like PACXX and DPC++) that are built on C++ so a porting effort would be possible. If the porting process needs to be incremental though, developers should check that a language supports incremental ports, e.g., by allowing access to raw pointers for data stored in containers. Some applications may be better served by a library, since libraries are generally designed with incremental porting in mind. Most of the directive-based models support C++ as well.

8.2.3 Writing New Applications

New applications are free to choose whatever programming model they want, but developers still have a great deal to consider when making their choice, including:

- Does this model let us express what we want?
- Does it provide tuning capabilities we want?
- Does it target the architectures we want, and will it add support for new architectures we might need in the future?
- Does it have a user community we can turn to for help?
- Will this model still exist in ten years (or the lifetime of the application)?[23]

Developers may also want to consider using a PPL (see Sec. 3.3) to insulate the application code (and non-expert software developers, such as domain scientists) from changes in underlying programming models. Using a PPL goes a long way towards removing the need to ask the last question above; if community support for a model disappears, the PPL can migrate to another model without disturbing application code. Using a PPL is also something existing applications can and should consider – PPLs were originally proposed to help legacy applications port to new architectures.

8.3 Portability

Most of the models described here are very portable, and many of the ones that aren't are actively working on adding support for more architectures. Portability is in many ways a prerequisite for performance portability (as defined for this paper, especially), since a model can't be performance portable if it only runs on one or two architectures. Regardless, all claims of portability should be taken with a grain of salt. A recent, large-scale study on performance portability by Deakin et al. [29] notes that one of the most difficult parts of comparing performance portability values for different programming models is simply getting applications to run on a variety of architectures in the first place. Immature implementations or lack of testing mean that, while small benchmarks might run, larger applications may still fail; Deakin et al.'s analysis had to cope with multiple failures on some of their applications. In addition, for many models, portability varies by implementation (see Table 2), and to migrate to a new architecture, users may have to switch implementations. This can be problematic since the implementations' support for the standard may vary, and performance can also vary, but this is an implementation problem and not a problem with the model itself.

Perhaps the most important aspect of the models themselves to consider, with respect to portability, is how easily they can be extended to support new architectures – particularly architectures that aren't just a variation on traditional CPUs, GPUs, or clusters. We don't know what HPC machines will look like in the future, so it's worth considering how extendable a model is and whether it has the right abstractions for parallelism, computation, and communication to be "future-proof," regardless of what happens to the underlying hardware, including CPUs, GPUs, the memory hierarchy, configurable computing such as FPGAs, and a host of other special purpose processors.

Higher-level models that internally utilize other, lower-level models, such as high-level libraries (e.g., SkePU, Kokkos) or high-level languages (e.g., Chapel), are a good example of extensibility – if a new architecture with a new programming model comes out, they can implement a new back end and their users can automatically take advantage of it. Other models, like OpenMP and the C++

[23] There were many, many libraries and languages proposed in the last 10-20 years that are not included in this paper because they never got past the "proposal" stage of community adoption. Many more made it past that stage only to be deprecated and mostly abandoned by their user communities.

STL, have had to work much harder to adapt, but have ultimately succeeded. The users of these models have also had to work to update their code, however, so this is not a sustainable path, as it can greatly impact the model's productivity during the changeover. If a programming model has to go through a major API change every time there is a shift in supercomputer architectures, it isn't truly portable.

8.4 Performance

Most of the models discussed here also give high performance, but this is much more inconsistent, since performance portability is still a struggle for many. (See Sec. 2.2 for a discussion of how optimizing for performance vs. for portability can be at odds.) Furthermore, optimizations that are good for one architecture might hurt performance on another, and this can be very difficult to reason about.

Because of this, it's important for programming models to give users lots of performance knobs to tune (if they so desire), but to keep those knobs *out of the application code* and preferably in one centralized location. Almost all the models described here do a good job of this, with the exceptions of OpenMP, OpenCL, and Legion, to some degree. The primary two ways models abstract out performance tuning are by providing high-level interfaces with specialized back ends (e.g., OpenACC and RAJA) or by allowing users to define their own specialized mappings to hardware (e.g., PPLs, Chapel, and custom directives). Both of these methods are functional, although the latter is more flexible at the cost of putting more responsibility on the user. Better support for parallelism in compilers could reduce this burden by making it easier for compilers to reason about parallelism, so users don't have to provide extra information about hardware mappings.

8.5 Productivity

Productivity is perhaps the most difficult of the 3 P's. There isn't yet a good definition or way to measure it (see Sec. 2.3), which means most discussion of portability (even in this paper) is qualitative, based on individuals' opinions and experiences. However, there is still a general consensus that even the performance portable models with the steepest learning curves are still better than device-specific models like CUDA and OpenCL in terms of productivity, if only because porting from one machine to another takes less work. Keeping these more portable models productive is good for performance portability – it forces them to stay higher-level and avoid falling into the "OpenCL trap" of becoming overly-specific, so they have a better chance of achieving consistent performance on multiple architectures.

Models that support incremental porting (directives and libraries) are in many ways the most productive for porting applications, since a small part of the application can be ported to test out a model without committing to rewriting thousands of lines of code, even though high-productivity languages like Chapel and HJ might be more succinct. Tools that can automate the porting process (like Bones and the other translators) are also a boon for productivity.

Perhaps the best solution for productivity, though, is an application-specific performance portability layer (see Sec. 3.3), since it can allow parallelism experts to play with different models in the background while domain scientists get on with their work, and it removes all dependence on a specific performance portable model from the application code itself. While the concept is still relatively new, separation of concerns between application programming and application tuning seems to be a very productive path to take.

The tools that go with these models, such as debuggers, testing apparatuses, and performance measurement tools, are also essential for productivity, but they are outside the scope of this paper – each of them is likely worthy of its own paper.

To conclude, a list of programming model features that are good for performance portability and productivity:

- High-level abstractions.
- Simple front end that targets multiple (non-performance portable) back ends.
- Keeping configuration separate from code.
- Multiple abstraction levels.
- Allow users to extend anything and everything.
- Automate, but let users override.
- Scale down as well as up.
- Standardization.
- Separate expressing parallelism from mapping it to hardware.

9 Conclusion

In short, each category of model has its uses, and progress is being made along the path to performance portability. The HPC community is better off now than it was even five years ago, and developers have more (and better) options than they used to. Some of these options are starting to converge; for example, OpenMP and OpenACC are both taking on some of each other's characteristics, some libraries (e.g., Kokkos and RAJA) are doing the

same, significantly better compiler tools (MLIR) are being developed, and large user communities are starting to form around some of these models. The HPC community seems to be moving out of the “innovation” and “early adoption” phases of the technology adoption lifecycle and into the “early maturity” phase [68]. The next two to five years will be very exciting to watch unfold; the new supercomputers will hopefully achieve the first exaFLOP, and the programming models for these machines will hopefully continue to coalesce into truly performance portable (and productive!) models.

Table 2: Comparison table for performance and portability.

		Performance			Portability									
	Model	CPU	GPU	Other	Intel CPU	AMD CPU	IBM CPU	Other CPU	Nvidia GPU	AMD GPU	Xeon Phi	FPGA	Other	Single vs. Multi-node
Non-Portable	CUDA	N/A	High	N/A	×	×	×	×	✓	×	×	×	×	Single
	OpenCL	High	High	Varied[a]	✓	✓	✓	◦[b]	✓	✓	✓	✓	◦[b]	Single
	OpenMP 3	High	N/A	N/A	✓	✓	✓	◦[b]	×	×	×	×	×	Single
	TBB	High	N/A	High[d]	✓	✓	✓	✓	×	×	✓	×	×	Single
Libraries	SkelCL	High	High	Varied[a]	✓	✓	✓	◦[b]	✓	✓	✓	✓	◦[b]	Single
	SkePU 2	High	High	Varied[a]	✓	✓	✓	◦[b]	✓	✓	✓	✓	◦[b]	Multi
	Muesli	High	Varied[c]	High[d]	✓	✓	✓	◦[b]	✓	◦[b]	✓	◦[b]	◦[b]	Multi
	GrPPI	High	N/A	N/A	✓	✓	✓	✓	×	×	×	×	×	Single
	Kokkos	High	High	High	✓	✓	✓	✓	✓	◦[i]	✓	×	×	Single
	RAJA	High	High	High	✓	✓	✓	✓[e]	✓	×	✓	×	×	Single
	PHAST	High	High	N/A	✓	✓	✓	✓	✓	×	×	×	×	Single
	targetDP	Moderate	Moderate	Moderate	✓	✓	✓	◦[b]	✓	×	✓	×	×	Single
Languages	Cilk	High	N/A	High[d,f]	✓	✓	✓	✓	×	×	✓[f]	×	×	Single
	Legion/Regent	High	High	N/A	✓	✓	✓	✓	✓	×	×	×	×	Multi
	X10/Habanero-Java	Moderate	N/A	N/A	✓	✓	✓	✓	×	×	×	×	×	Multi
	Chapel	High	High	Varied[i,d]	✓	✓	✓	✓	✓	✓	◦[i]	×	×	Multi
	PACXX	High	High	High[d]	✓	✓	✓	✓	✓	✓	✓	◦[g]	×	Single
	SYCL/DPC++	High	High	Varied[a]	✓	✓	✓	◦[b]	✓	✓	✓	✓	◦[b]	Single
Directives	OpenMP 4+	High	Varied[h]	Moderate[i]	✓	✓	✓	◦[b]	✓	◦[b]	✓	◦[i,b]	◦[i,b]	Single
	OpenACC	High	High	Moderate[i]	✓	✓	✓	◦[b]	✓	◦[b]	✓	◦[i,b]	◦[b]	Single
	XcalableMP/ACC	High	High	Moderate	✓	✓	✓	◦[b]	✓	✓	✓	✓	◦[b]	Multi
	OpenMC	Moderate	Moderate	Moderate[d]	✓	✓	✓	◦[b]	✓	×	✓	×	×	Single
	HSTREAM	Unknown[j]	Unknown[j]	Unknown[j]	✓	✓	✓	◦[b]	✓	×	✓	×	×	Single
Translators	Bones	High	High	High	✓	✓	✓	◦[b]	✓	✓	✓	✓	◦[b]	Single
	Clacc	High	High	Moderate[b]	✓	✓	✓	◦[b]	✓	◦[b]	✓	◦[i,b]	◦[i,b]	Single

[a] Support is experimental and performance varies by implementation.
[b] Varies by implementation (or chosen back end), but many/most do not support.
[c] C++ performance is good, Java performance is inconsistent.
[d] Xeon Phi only.
[e] Via the serial back end, at least.
[f] Only with Cilk Plus.
[g] If the FPGA supports SPIR.
[h] GPU performance varies wildly by implementation, but is improving.
[i] Support is experimental.
[j] Performance has not been compared to other models yet.

Table 3: Comparison table for productivity, and overall P3 scores.

	Model	Productivity: C	C++	Fortran	Other langs.	SLOC added[a,b]	Same code per platform[c]	Difficulty[d]	Incremental porting	Overall P3 Scores: Numeric	Qualitative
Non-Portable	CUDA	✓	✓	○[e]	✓[f]	High	✓	High	✓	8.5	Poor
	OpenCL	✓	✓	×	✓[g]	High	×	High	✓	17	Fair
	OpenMP 3	✓	✓	✓	○[h]	Low	✓	Moderate	✓	14	Fair
	TBB	○[j]	✓	×	×	Moderate/High	✓	Moderate/High	✓	13.5	Fair
Libraries	SkelCL	○[j]	✓	×	×	Moderate	✓	Low/Moderate	✓	19	Fair
	SkePU 2	○[j]	✓	×	×	Moderate[k]	✓	Low/Moderate	✓	19	Good
	Muesli	○[j]	✓	×	✓[i]	Moderate	✓	Moderate	✓	18.5	Good/Fair
	GrPPI	○[j]	✓	×	×	Moderate	✓	Moderate	✓	11.5	Poor, Promising
	Kokkos	○[j]	✓	×	×	Moderate[k]	✓	Low/Moderate[k]	✓	18.5	Very Good
	RAJA	○[j]	✓	×	×	Low/Moderate[k]	✓	Moderate[k]	✓	18	Good, Promising
	PHAST	○[j]	✓	×	×	Low/Moderate	✓	Moderate	✓	15	Fair, Promising
	targetDP	✓	×	×	×	Moderate	✓	Moderate/High	✓	13	Poor
Languages	Cilk	○[l]	○[m]	×	×	Low	✓	Moderate	○[n]	14.5	Good
	Legion/Regent	○[j]	✓	×	×	Moderate/High	✓	Moderate/High	×	12.5	Good
	X10/Habanero-Java	×	×	×	✓[i]	Low/Moderate	✓	Moderate	×	9.5	Fair
	Chapel	N/A	N/A	N/A	✓[o]	Very Low	✓	Low/Moderate	×	17	Excellent
	PACXX	○[j]	✓	×	×	Low/Moderate	✓	Low/Moderate	✓	20	Good, Promising
	SYCL/DPC++	○[j]	✓	×	×	Moderate	✓	Moderate	✓	18.5	Very Good, Promising
Directives	OpenMP 4+	✓	✓	✓	○[h]	Low	×	Moderate	✓	18.5	Good
	OpenACC	✓	✓	✓	×	Low	✓	Low/Moderate	✓	20.5	Very Good
	XcalableMP/ACC	✓	○[p]	✓	×	Low	✓	Low/Moderate	✓	21	Excellent
	OpenMC	✓	×	×	×	Low	×	Moderate	✓	13.5	Fair
	HSTREAM	○[j]	✓	×	×	Low	✓	Low/Moderate	✓	15.5	Promising
Translators	Bones	✓	×	×	×	None	✓	Low	✓	21	Fair
	Clacc	✓	○[q]	×	×	Low	✓	Low/Moderate	×	18	Very Good

[a] Compared to serial/base version.
[b] For languages, this is code size increase compared to serial C/C++/Fortran/etc.
[c] No code changes required to run or get reasonable (not necessarily high) performance on different platforms. E.g., OpenMP 4+ requires different pragmas for GPU and non-GPU platforms and thus does not have this property.
[d] For a programmer to learn, understand, and/or maintain.
[e] Only through PGI's CUDA Fortran.
[f] Python and Haskell bindings, possibly others.
[g] Fortran, Pascal, Go, Haskell, Java, Julia, Rust, Scala, Python, Ruby, Erlang, and .NET bindings, possibly others.
[h] A partial Java implementation is available from the University of Auckland.
[i] Java.
[j] Only via porting to C++.
[k] Depends on lambda support.
[l] The serial elision of a Cilk or Cilk Plus program is valid (GNU) C.
[m] The serial elision of a Cilk++ or Cilk Plus program is valid C++.
[n] Must port entire translations units at the same time.
[o] Chapel has unique syntax different from other languages.
[p] C++ is in development for the underlying translator, Omni.
[q] Support is in the works.

References

[1] "Intel Array Building Blocks," accessed via the Internet Archive, 6/1/2020. [Online]. Available: https://web.archive.org/web/20130129203900/http://software.intel.com/en-us/articles/intel-array-building-blocks/

[2] P. Alpatov, G. Baker, H. C. Edwards, J. Gunnels, G. Morrow, J. Overfelt, and R. van de Geijn, "PLAPACK parallel linear algebra package design overview," in *SC '97: Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, 1997.

[3] M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, J. O. Mcmahon, F.-X. Pasquier, G. Péan, and P. Villalon, "Par4All: From convex array regions to heterogeneous computing," in *IMPACT 2012 : Second International Workshop on Polyhedral Compilation Techniques HiPEAC 2012*, Paris, France, Jan. 2012.

[4] S. F. Antao, A. Bataev, A. C. Jacob, G.-T. Bercea, A. E. Eichenberger, G. Rokos, M. Martineau, T. Jin, G. Ozen, Z. Sura, T. Chen, H. Sung, C. Bertolli, and K. O'Brien, "Offloading support for OpenMP in Clang and LLVM," in *Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC*, ser. LLVM-HPC '16. Piscataway, NJ, USA: IEEE Press, Nov. 2016, pp. 1–11.

[5] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from Berkeley," Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Tech. Rep., 2006.

[6] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[7] D. Bailey, "Twelve ways to fool the masses when giving performance results on parallel computers," Jun. 1991.

[8] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov. 2012, pp. 1–11.

[9] D. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. Kunen, O. Pearce, P. Robinson, B. Ryujin, and T. Scogland, "RAJA: Portable performance for large-scale scientific applications," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, Nov. 2019, pp. 71–81.

[10] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for CUDA," in *GPU computing gems Jade edition*. Elsevier, 2012, pp. 359–371.

[11] E. D. Berger, S. Z. Guyer, and C. Lin, "Customizing software libraries for performance portability," 2000.

[12] C. Bertolli, S. F. Antao, G.-T. Bercea, A. C. Jacob, A. E. Eichenberger, T. Chen, Z. Sura, H. Sung, G. Rokos, D. Appelhans, and K. O'Brien, "Integrating GPU support for OpenMP offloading directives into Clang," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM '15. New York, NY, USA: ACM, 2015, pp. 5:1–5:11.

[13] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55 – 69, 1996.

[14] S. Breuer, M. Steuwer, and S. Gorlatch, "Extending the SkelCL skeleton library for stencil computations on multi-GPU systems," in *Proceedings of the 1st International Workshop on High-Performance Stencil Computations*, 2014, pp. 15–21.

[15] P. Bright, "Cray, AMD to build 1.5 exaflops supercomputer for US government," May 2019. [Online]. Available: https://arstechnica.com/gadgets/2019/05/cray-amd-to-build-1-5-exaflops-supercomputer-for-us-governme

[16] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-Java: The new adventures of old X10," in *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, ser. PPPJ '11. New York, NY, USA: ACM, Aug. 2011, pp. 51–61.

[17] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the Chapel language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[18] B. L. Chamberlain, E. Ronaghan, B. Albrecht, L. Duncan, M. Ferguson, B. Harshbarger, D. Iten, D. Keaton, V. Litvinov, P. Sahabu, and G. Titus, "Chapel comes of age: Making scalable programming productive," *Cray User Group Meeting 2018*, 2018.

[19] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: ACM, Oct. 2005, pp. 519–538.

[20] M. Chu, A. Aji, D. Lowell, and K. Hamidouche, "GPGPU support in Chapel with the Radeon Open Compute platform," Jun. 2017.

[21] P. Ciechanowicz and H. Kuchen, "Enhancing Muesli's data parallel skeletons for multi-core computer architectures," in *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, 2010, pp. 108–113.

[22] P. Ciechanowicz, M. Poldner, and H. Kuchen, "The Münster skeleton library Muesli: A comprehensive overview," Working Papers, ERCIS-European Research Center for Information Systems, Tech. Rep., 2009.

[23] V. Clement, S. Ferrachat, O. Fuhrer, X. Lapillonne, C. E. Osuna, R. Pincus, J. Rood, and W. Sawyer, "The CLAW DSL: Abstractions for performance portable weather and climate models," in *Proceedings of the Platform for Advanced Scientific Computing Conference*, ser. PASC '18. New York, NY, USA: ACM, Jul. 2018, pp. 2:1–2:10. [Online]. Available: http://doi.acm.org/10.1145/3218176.3218226

[24] H. C. da Silva, F. Pisani, and E. Borin, "A comparative study of SYCL, OpenCL, and OpenMP," in *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, 2016, pp. 61–66.

[25] D. Daniel and J. Panetta, "On applying performance portability metrics," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, Nov. 2019, pp. 50–59.

[26] U. Dastgeer and C. Kessler, "Smart containers and skeleton programming for GPU-based systems," *International journal of parallel programming*, vol. 44, no. 3, pp. 506–530, 2016.

[27] B. R. de Supinski, T. R. W. Scogland, A. Duran, M. Klemm, S. M. Bellido, S. L. Olivier, C. Terboven, and T. G. Mattson, "The ongoing evolution of OpenMP," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 2004–2019, Nov. 2018.

[28] T. Deakin and S. McIntosh-Smith, "Evaluating the performance of HPC-style SYCL applications," in *Proceedings of the International Workshop on OpenCL*, ser. IWOCL '20. New York, NY, USA: Association for Computing Machinery, Apr. 2020.

[29] T. Deakin, S. McIntosh-Smith, J. Price, A. Poenaru, P. Atkinson, C. Popa, and J. Salmon, "Performance portability across diverse computer architectures," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, Nov. 2019, pp. 1–13.

[30] D. del Rio Astorga, M. F. Dolz, J. Fernández, and J. D. García, "Paving the way towards high-level parallel pattern interfaces for data stream processing," *Future Generation Computer Systems*, vol. 87, pp. 228 – 241, 2018.

[31] D. del Rio Astorga, M. F. Dolz, L. M. Sanchez, J. G. Blas, and J. D. García, "A C++ generic parallel pattern interface for stream processing," in *Algorithms and Architectures for Parallel Processing*, J. Carretero, J. Garcia-Blas, R. K. Ko, P. Mueller, and K. Nakano, Eds. Cham: Springer International Publishing, 2016, pp. 74–87.

[32] J. E. Denny, S. Lee, and J. S. Vetter, "Clacc: Translating OpenACC to OpenMP in Clang," in *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, Nov. 2018, pp. 18–29.

[33] R. Dolbeau, S. Bihan, and F. Bodin, "HMPP: A hybrid multi-core parallel programming environment," in *Workshop on general purpose processing on graphics processing units (GPGPU 2007)*, vol. 28, 2007.

[34] H. Dreuning, R. Heirman, and A. L. Varbanescu, "A beginner's guide to estimating and improving performance portability," in *High Performance Computing*, R. Yokota, M. Weiland, J. Shalf, and S. Alam, Eds. Cham: Springer International Publishing, 2018, pp. 724–742.

[35] H. C. Edwards, D. Sunderland, C. Amsler, and S. Mish, "Multicore/GPGPU portable computational kernels via multidimensional arrays," in *2011 IEEE International Conference on Cluster Computing*, Sep. 2011, pp. 363–370.

[36] H. C. Edwards and D. Sunderland, "Kokkos array performance-portable manycore programming model," in *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM '12. New York, NY, USA: ACM, Feb. 2012, pp. 1–10.

[37] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[38] J. Enmyren and C. W. Kessler, "SkePU: A multi-backend skeleton programming library for multi-GPU systems," in *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, ser. HLPP '10. New York, NY, USA: ACM, Sep. 2010, pp. 5–14.

[39] S. Ernsting and H. Kuchen, "A scalable farm skeleton for hybrid parallel and distributed programming," *International Journal of Parallel Programming*, vol. 42, no. 6, pp. 968–987, Dec. 2014.

[40] ——, "Data parallel algorithmic skeletons with accelerator support," *International Journal of Parallel Programming*, vol. 45, no. 2, pp. 283–299, Apr. 2017.

[41] A. Ernstsson, L. Li, and C. Kessler, "SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems," *International Journal of Parallel Programming*, vol. 46, no. 1, pp. 62–80, 2018.

[42] H. Finkel, D. Poliakoff, J.-S. Camier, and D. F. Richards, "ClangJIT: Enhancing C++ with just-in-time compilation," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, Nov. 2019, pp. 82–96.

[43] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI '98. New York, NY, USA: ACM, 1998, pp. 212–223.

[44] A. Gray and K. Stratford, "targetDP: an abstraction of lattice based parallelism with portable performance," in *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)*, 2014, pp. 312–315.

[45] A. Gray and K. Stratford, "A lightweight approach to performance portability with targetDP," *The International Journal of High Performance Computing Applications*, vol. 32, no. 2, pp. 288–301, 2018.

[46] M. Griebl, C. Lengauer, and S. Wetzel, "Code generation in the polytope model," in *In IEEE PACT*. IEEE Computer Society Press, 1998, pp. 106–111.

[47] J. Gustafson, "Twelve ways to fool the masses when giving performance results on traditional vector computers," Jun. 1991.

[48] S. Z. Guyer and C. Lin, "Broadway: A compiler for exploiting the domain-specific semantics of software libraries," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 342–357, 2005.

[49] B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach, "High performance stencil code generation with Lift," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO 2018. New York, NY, USA: ACM, Feb. 2018, pp. 100–112.

[50] M. Haidl and S. Gorlatch, "PACXX: Towards a unified programming model for programming accelerators using C++14," in *2014 LLVM Compiler Infrastructure in HPC*, 2014, pp. 1–11.

[51] M. Haidl and S. Gorlatch, "High-level programming for many-cores using C++14 and the STL," *International Journal of Parallel Programming*, vol. 46, no. 1, pp. 23–41, 2018.

[52] T. D. Han and T. S. Abdelrahman, "hiCUDA: A high-level directive-based language for GPU programming," in *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. New York, NY, USA: ACM, 2009, pp. 52–61.

[53] 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)*, Nov. 2018, pp. 24–36.

[54] A. Hayashi, S. R. Paul, and V. Sarkar, "GPUIterator: Bridging the gap between Chapel and GPU platforms," in *Proceedings of the ACM SIGPLAN*

6th Chapel Implementers and Users Workshop, ser. CHIUW 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 2–11.

[55] ——, "Exploring a multi-resolution GPU programming model for Chapel," in *7th Chapel Implementers and Users Workshop*, ser. CHIUW 2020, New York, NY, USA, May 2020.

[56] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Elsevier Science, 2019.

[57] J. A. Herdman, W. P. Gaudin, O. Perks, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis, "Achieving portability and performance through OpenACC," in *2014 First Workshop on Accelerator Programming using Directives*, Nov. 2014, pp. 19–26.

[58] D. Hollman, B. Lelbach, H. C. Edwards, M. Hoemmen, D. Sunderland, and C. Trott, "mdspan in C++: A case study in the integration of performance portable features into international language standards," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, Nov. 2019, pp. 60–70.

[59] J. Holmen, B. Peterson, and M. Berzins, "An approach for indirectly adopting a performance portability layer in large legacy codes," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, Nov. 2019, pp. 36–49.

[60] R. Hornung and J. Keasler, "A case for improved C++ compiler support to enable performance portability in large physics simulation codes," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2013.

[61] R. D. Hornung and J. A. Keasler, "The RAJA portability layer: Overview and status," Sep. 2014.

[62] Intel Corp., "Intel oneAPI programming guide (beta)," May 2020. [Online]. Available: https://software.intel.com/content/www/us/en/develop/documentation/oneapi-programming-guide/top.html

[63] ——, "Intel Threading Building Blocks documentation," Mar. 2020. [Online]. Available: https://software.intel.com/content/www/us/en/develop/documentation/tbb-documentation/top/intel-threading-building-blocks-developer-reference.html

[64] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "pocl: A performance-portable OpenCL implementation," *International Journal of Parallel Programming*, vol. 43, no. 5, pp. 752–785, Oct. 2015.

[65] J. Jeffrey-Wilensky, "New 'Aurora' supercomputer poised to be fastest in U.S. history," Mar. 2019. [Online]. Available: https://www.nbcnews.com/mach/science/new-aurora-supercomputer-poised-be-fastest-u-s-history-ncna9

[66] T. A. Johnson, "Coarray C++," in *7th International Conference on PGAS Programming Models*, 2013, p. 54.

[67] H. Jordan, S. Pellegrini, P. Thoman, K. Kofler, and T. Fahringer, "INSPIRE: The insieme parallel intermediate representation," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2013, pp. 7–17.

[68] W. Joubert, R. Archibald, M. Berrill, W. Michael Brown, M. Eisenbach, R. Grout, J. Larkin, J. Levesque, B. Messer, M. Norman, B. Philip, R. Sankaran, A. Tharrington, and J. Turner, "Accelerated application development: The ORNL Titan experience," *Comput. Electr. Eng.*, vol. 46, no. C, pp. 123–138, Aug. 2015.

[69] K. Keutzer, B. L. Massingill, T. G. Mattson, and B. A. Sanders, "A design pattern language for engineering (parallel) software: Merging the PLPP and OPL projects," in *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ser. ParaPLoP '10. New York, NY, USA: Association for Computing Machinery, 2010.

[70] D. Khaldi, P. Jouvelot, C. Ancourt, and F. Irigoin, "SPIRE: A sequential to parallel intermediate representation extension," MINES ParisTech, Tech. Rep., 2012.

[71] D. Khaldi, P. Jouvelot, F. Irigoin, C. Ancourt, and B. Chapman, "LLVM parallel intermediate representation: Design and evaluation using OpenSHMEM communications," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM '15. New York, NY, USA: ACM, Nov. 2015, pp. 2:1–2:8.

[72] Khronos OpenCL Working Group, "The OpenCL specification," Jul. 2019. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/2.2/html/OpenCL_API.html

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

[74] P. Kogge and J. Shalf, "Exascale computing trends: Adjusting to the "new normal" for computer architecture," *Computing in Science Engineering*, vol. 15, no. 6, pp. 16–26, Nov. 2013.

[75] Kokkos Team, May 2020. [Online]. Available: https://github.com/kokkos/kokkos

[76] K. Komatsu, R. Egawa, S. Hirasawa, H. Takizawa, K. Itakura, and H. Kobayashi, "Translation of large-scale simulation codes for an OpenACC platform using the Xevolver framework," *International Journal of Networking and Computing*, vol. 6, no. 2, pp. 167–180, 2016.

[77] J. Lambert, S. Lee, J. Kim, J. S. Vetter, and A. D. Malony, "Directive-based, high-level programming and optimizations for high-performance computing with FPGAs," in *Proceedings of the 2018 International Conference on Supercomputing*, ser. ICS '18. New York, NY, USA: ACM, Jun. 2018, pp. 160–171.

[78] J. Larkin, "Performance portability through descriptive parallelism," Apr. 2016.

[79] V. V. Larrea, W. Joubert, M. G. Lopez, and O. Hernandez, "Early experiences writing performance portable OpenMP 4 codes," in *Proc. Cray User Group Meeting, London, England*, 2016.

[80] A. Lashgar, A. Majidi, and A. Baniasadi, "IPMACC: Translating OpenACC API to OpenCL," in *Poster session of The 3rd International Workshop on OpenCL (IWOCL)*, 2015.

[81] C. Lattner and V. Adve, "LLVM: a compilation framework for lifelong program analysis and transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86.

[82] C. Lattner, J. Pienaar, M. Amini, U. Bondhugula, R. Riddle, A. Cohen, T. Shpeisman, A. Davis, N. Vasilache, and O. Zinenko, "MLIR: A compiler infrastructure for the end of Moore's law," *ArXiv*, vol. abs/2002.11054, 2020.

[83] S. Lee and R. Eigenmann, "OpenMPC: Extended OpenMP programming and tuning for GPUs," in *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2010, pp. 1–11.

[84] S. Lee and J. S. Vetter, "OpenARC: Extensible OpenACC compiler framework for directive-based accelerator programming study," in *2014 First Workshop on Accelerator Programming using Directives*, Nov. 2014, pp. 1–11.

[85] ——, "OpenARC: Open accelerator research compiler for directive-based, efficient heterogeneous computing," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '14. New York, NY, USA: ACM, Jun. 2014, pp. 115–120.

[86] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: A compiler framework for automatic translation and optimization," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '09. New York, NY, USA: ACM, Feb. 2009, pp. 101–110.

[87] C. E. Leiserson, "The Cilk++ concurrency platform," *The Journal of Supercomputing*, vol. 51, no. 3, pp. 244–257, Mar. 2010.

[88] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin, "A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3. New York, NY, USA: ACM, Mar. 2010, pp. 51–61.

[89] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng, "OpenUH: an optimizing, portable OpenMP compiler," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 18, pp. 2317–2332, Dec. 2007.

[90] C. Liao, P.-H. Lin, M. Schordan, and I. Karlin, "A semantics-driven approach to improving DataRaceBench's OpenMP standard coverage," in *Evolving OpenMP for Evolving Architectures*, B. R. de Supinski, P. Valero-Lara, X. Martorell, S. Mateo Bellido, and J. Labarta, Eds. Cham: Springer International Publishing, 2018, pp. 189–202.

[91] X.-K. Liao, C.-Q. Yang, T. Tang, H.-Z. Yi, F. Wang, Q. Wu, and J. Xue, "OpenMC: Towards simplifying programming for TianHe supercomputers," *Journal of Computer Science and Technology*, vol. 29, no. 3, pp. 532–546, May 2014.

[92] J. Lidman, D. J. Quinlan, C. Liao, and S. A. McKee, "ROSE::FTTransform – a source-to-source translation framework for exascale fault-tolerance research," in *IEEE/IFIP International Conference*

on Dependable Systems and Networks Workshops (DSN 2012), 2012, pp. 1–6.

[93] Y. Liu, L. Huang, M. Wu, H. Cui, F. Lv, X. Feng, and J. Xue, "PPOpenCL: A performance-portable OpenCL compiler with host and kernel thread code fusion," in *Proceedings of the 28th International Conference on Compiler Construction*, ser. CC 2019. New York, NY, USA: ACM, Feb. 2019, pp. 2–16.

[94] R. Lucas, J. Ang, K. Bergman, S. Borkar, W. Carlson, L. Carrington, G. Chiu, R. Colwell, W. Dally, J. Dongarra, A. Geist, R. Haring, J. Hittinger, A. Hoisie, D. M. Klein, P. Kogge, R. Lethin, V. Sarkar, R. Schreiber, J. Shalf, T. Sterling, R. Stevens, J. Bashor, R. Brightwell, P. Coteus, E. Debenedictus, J. Hiller, K. H. Kim, H. Langston, R. M. Murphy, C. Webster, S. Wild, G. Grider, R. Ross, S. Leyffer, and J. Laros III, "DOE Advanced Scientific Computing Advisory Subcommittee (ASCAC) report: Top ten exascale research challenges," US Department of Energy, Tech. Rep., Feb. 2014.

[95] C. Luk, S. Hong, and H. Kim, "Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec. 2009, pp. 45–55.

[96] M. Majeed, U. Dastgeer, and C. W. Kessler, "Cluster-SkePU: A multi-backend skeleton programming library for GPU clusters," in *PDPTA 2013*, 2013.

[97] M. Martineau, S. McIntosh-Smith, and W. Gaudin, "Evaluating OpenMP 4.0's effectiveness as a heterogeneous parallel programming model," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 338–347.

[98] M. Martineau and S. McIntosh-Smith, "The productivity, portability and performance of OpenMP 4.5 for scientific applications targeting Intel CPUs, IBM CPUs, and NVIDIA GPUs," in *Scaling OpenMP for Exascale Performance and Portability (IWOMP '17)*, B. R. de Supinski, S. L. Olivier, C. Terboven, B. M. Chapman, and M. S. Müller, Eds. Cham: Springer International Publishing, 2017, pp. 185–200.

[99] M. Martineau, S. McIntosh-Smith, M. Boulton, and W. Gaudin, "An evaluation of emerging many-core parallel programming models," in *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM'16. New York, NY, USA: ACM, Mar. 2016, pp. 1–10.

[100] M. Martineau, S. McIntosh-Smith, and W. Gaudin, "Assessing the performance portability of modern parallel programming models using TeaLeaf," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 15, p. e4117, 2017, e4117 cpe.4117.

[101] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[102] S. McIntosh-Smith, M. Boulton, D. Curran, and J. Price, "On the performance portability of structured grid codes on many-core computer architectures," in *Supercomputing*, J. M. Kunkel, T. Ludwig, and H. W. Meuer, Eds. Cham: Springer International Publishing, 2014, pp. 53–75.

[103] B. Meister, A. Leung, N. Vasilache, D. Wohlford, C. Bastoul, and R. Lethin, "Productivity via automatic code generation for PGAS platforms with the R-Stream compiler," in *Workshop on Asynchrony in the PGAS Programming Model*, 2009.

[104] J. Mellor-Crummey, L. Adhianto, W. N. Scherer, and G. Jin, "A new vision for Coarray Fortran," in *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, ser. PGAS '09. New York, NY, USA: Association for Computing Machinery, Oct. 2009.

[105] S. Memeti and S. Pllana, "HSTREAM: A directive-based language extension for heterogeneous stream computing," in *2018 IEEE International Conference on Computational Science and Engineering (CSE)*, Oct. 2018, pp. 138–145.

[106] J. Michalakes, R. Loft, and A. Bourgeois, "Performance-portability and the weather research and forecast model," 2001.

[107] Z.-y. Mo, "Extreme-scale parallel computing: bottlenecks and strategies," *Frontiers of Information Technology & Electronic Engineering*, vol. 19, no. 10, pp. 1251–1260, Oct. 2018.

[108] S. Moss, "Lawrence Berkeley to install Perlmutter supercomputer featuring Cray's Shasta system," Oct. 2018. [Online]. Available: https://www.datacenterdynamics.com/news/lawrence-berkeley-install-perlmutter-supercomputer-featuring-c

[109] MPI Forum, "MPI: A message-passing interface standard," Jun. 2015.

[110] H. Murai, M. Sato, M. Nakao, and J. Lee, "Metaprogramming framework for existing HPC languages based on the Omni compiler infrastructure," in *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)*, Nov. 2018, pp. 250–256.

[111] F. Mößbauer, R. Kowalewski, T. Fuchs, and K. Fürlinger, "A portable multidimensional coarray for C++," in *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, 2018, pp. 18–25.

[112] M. Nakao, J. Lee, T. Boku, and M. Sato, "Productivity and performance of global-view programming with XcalableMP PGAS language," in *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, May 2012, pp. 402–409.

[113] M. Nakao, H. Murai, T. Shimosaka, A. Tabuchi, T. Hanawa, Y. Kodama, T. Boku, and M. Sato, "XcalableACC: Extension of XcalableMP PGAS language using OpenACC for accelerator clusters," in *2014 First Workshop on Accelerator Programming using Directives*, Nov. 2014, pp. 27–36.

[114] S. Nanz, S. West, K. S. d. Silveira, and B. Meyer, "Benchmarking usability and performance of multicore languages," in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, 2013, pp. 183–192.

[115] C. J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. D. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang, "Intel's Array Building Blocks: A retargetable, dynamic compiler and embedded language," in *International Symposium on Code Generation and Optimization (CGO 2011)*, 2011, pp. 224–235.

[116] C. Nugteren and H. Corporaal, "Bones: An automatic skeleton-based C-to-CUDA compiler for GPUs," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 35:1–35:25, Dec. 2014.

[117] C. Nugteren, P. Custers, and H. Corporaal, "Automatic skeleton-based compilation through integration with an algorithm classification," in *Advanced Parallel Processing Technologies*, C. Wu and A. Cohen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 184–198.

[118] Nvidia, "CUDA Toolkit documentation," Nov. 2019. [Online]. Available: https://docs.nvidia.com/cuda/

[119] G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, and M. Püschel, "Applying the roofline model," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2014, pp. 76–85.

[120] OpenACC Group, "OpenACC." [Online]. Available: https://www.openacc.org/

[121] OpenACC Standards Group, "The OpenACC application program interface, version 1.0," Nov. 2011.

[122] ——, "The OpenACC application program interface, version 2.7," Nov. 2018.

[123] ——, "The OpenACC application program interface, version 3.0," Nov. 2019.

[124] OpenMP Architecture Review Board, "OpenMP application program interface, version 3.1," Jul. 2011.

[125] ——, "OpenMP application program interface, version 4.5," Nov. 2015.

[126] ——, "OpenMP application program interface, version 5.0," Nov. 2018.

[127] ——, "OpenMP compilers and tools," Nov. 2019. [Online]. Available: https://www.openmp.org/resources/openmp-compilers-tools/

[128] OpenSHMEM Contributors Committee, "OpenSHMEM application programming interface," Dec. 2017.

[129] G. Özen, S. Atzeni, M. Wolfe, A. Southwell, and G. Klimowicz, "OpenMP GPU offload in Flang and LLVM," in *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, Nov. 2018, pp. 1–9.

[130] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil, "PLASMA: Portable programming for SIMD heterogeneous accelerators," in *Workshop on Language, Compiler, and Architecture Support for GPGPU, held in conjunction with HPCA/PPoPP 2010*, Jan. 2010.

[131] S. Pakin, "Ten ways to fool the masses when giving performance results on GPUs," Dec. 2011.

[132] B. Peccerillo and S. Bartolini, "PHAST library — enabling single-source and high performance code for GPUs and multi-cores," in *2017 International Conference on High Performance Computing Simulation (HPCS)*, 2017, pp. 715–718.

[133] ——, "PHAST - a portable high-level modern C++ programming library for GPUs and multi-cores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 1, pp. 174–189, Jan. 2019.

[134] H. Peng and J. J. Shann, "Translating OpenACC to LLVM IR with SPIR kernels," in *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*, Jun. 2016, pp. 1–6.

[135] S. J. Pennycook, J. D. Sewall, and J. R. Hammond, "Evaluating the impact of proposed OpenMP 5.0 features on performance, portability and productivity," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, Nov. 2018, pp. 37–46.

[136] S. J. Pennycook, J. D. Sewall, and A. Duran, "Supporting function variants in OpenMP," in *Evolving OpenMP for Evolving Architectures*, B. R. de Supinski, P. Valero-Lara, X. Martorell, S. Mateo Bellido, and J. Labarta, Eds. Cham: Springer International Publishing, 2018, pp. 128–142.

[137] S. J. Pennycook, J. D. Sewall, and V. W. Lee, "A metric for performance portability," in *Proceedings of the International Workshop on Performance Modeling, Benchmarking, and Simulation*, 2016.

[138] S. Pino, L. Pollock, and S. Chandrasekaran, "Exploring translation of OpenMP to OpenACC 2.5: lessons learned," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2017, pp. 673–682.

[139] D. Quinlan and C. Liao, "The ROSE source-to-source compiler infrastructure," in *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, 2011.

[140] J. Reid, "The new features of Fortran 2008," *SIGPLAN Fortran Forum*, vol. 27, no. 2, p. 8–21, Aug. 2008.

[141] ——, "The new features of Fortran 2018," *SIGPLAN Fortran Forum*, vol. 37, no. 1, p. 5–43, Apr. 2018.

[142] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, and X. Tian, *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. Apress, Nov. 2019, unedited advance preview of chapters 1-4.

[143] R. Reyes, I. López-Rodríguez, J. J. Fumero, and F. de Sande, "accULL: An OpenACC implementation with CUDA and OpenCL support," in *Euro-Par 2012 Parallel Processing*, C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 871–882.

[144] A. D. Robison, "Composable parallel patterns with Intel Cilk Plus," *Computing in Science Engineering*, vol. 15, no. 2, pp. 66–71, 2013.

[145] T. B. Schardl, I.-T. A. Lee, and C. E. Leiserson, "Brief announcement: Open Cilk," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 351–353.

[146] T. B. Schardl, W. S. Moses, and C. E. Leiserson, "Tapir: Embedding fork-join parallelism into LLVM's intermediate representation," in *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '17. New York, NY, USA: ACM, 2017, pp. 249–265.

[147] E. Schweitz, R. Lethin, A. Leung, and B. Meister, "R-stream: A parametric high level compiler," *Proceedings of HPEC*, 2006.

[148] A. Sedova, J. D. Eblen, R. Budiardja, A. Tharrington, and J. C. Smith, "High-performance molecular dynamics simulation for biological and materials sciences: Challenges of performance portability," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, Nov. 2018, pp. 1–13.

[149] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *High Performance Computing for Computational Science – VECPAR 2010*, J. M. L. M. Palma, M. Daydé, O. Marques, and J. C. Lopes, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–25.

[150] A. Sidelnik, S. Maleki, B. L. Chamberlain, M. J. Garzar'n, and D. Padua, "Performance portability with the Chapel language," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, May 2012, pp. 582–594.

[151] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: a high-productivity programming language for HPC with logical regions," in *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2015, pp. 1–12.

[152] J. E. Smith, "Characterizing computer performance with a single number," *Communications of the ACM*, vol. 31, no. 10, pp. 1202–1206, 1988.

[153] Sourcery Institute, "OpenCoarrays," Oct. 2019. [Online]. Available: https://github.com/sourceryinstitute/OpenCoarrays

[154] G. Stelle, W. S. Moses, S. L. Olivier, and P. McCormick, "OpenMPIR: Implementing OpenMP tasks with Tapir," in *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM-HPC'17. New York, NY, USA: ACM, Nov. 2017, pp. 3:1–3:12.

[155] M. Steuwer, P. Kegel, and S. Gorlatch, "SkelCL - a portable skeleton library for high-level GPU programming," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, May 2011, pp. 1176–1182.

[156] ——, "Towards high-level programming of multi-GPU systems using the SkelCL library," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, May 2012, pp. 1858–1865.

[157] M. Steuwer, T. Remmelg, and C. Dubach, "Lift: A functional data-parallel IR for high-performance GPU code generation," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb. 2017, pp. 74–85.

[158] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach, "Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance OpenCL code," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP 2015. New York, NY, USA: ACM, Sep. 2015, pp. 205–217.

[159] M. Steuwer and S. Gorlatch, "SkelCL: Enhancing OpenCL for high-level programming of multi-GPU systems," in *Parallel Computing Technologies*, V. Malyshkin, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 258–272.

[160] R. Suda, H. Takizawa, and S. Hirasawa, "Xevtgen: Fortran code transformer generator for high performance scientific codes," *International Journal of Networking and Computing*, vol. 6, no. 2, pp. 263–289, 2016.

[161] N. Sultana, A. Calvert, J. L. Overbey, and G. Arnold, "From OpenACC to OpenMP 4: Toward automatic translation," in *Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale*, ser. XSEDE16. New York, NY, USA: ACM, Jul. 2016, pp. 44:1–44:8.

[162] A. Tabuchi, M. Nakao, H. Murai, T. Boku, and M. Sato, "Implementation and evaluation of one-sided PGAS communication in XcalableACC for accelerated clusters," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2017, pp. 625–634.

[163] H. Takizawa, S. Hirasawa, Y. Hayashi, R. Egawa, and H. Kobayashi, "Xevolver: An XML-based code translation framework for supporting HPC application migration," in *2014 21st International Conference on High Performance Computing (HiPC)*, Dec. 2014, pp. 1–11.

[164] A. T. Tan and H. Kaiser, "Extending C++ with co-array semantics," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ser. ARRAY 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 63–68.

[165] O. Tardieu, B. Herta, D. Cunningham, D. Grove, P. Kambadur, V. Saraswat, A. Shinnar, M. Takeuchi, and M. Vaziri, "X10 and APGAS at petascale," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 53–66.

[166] R. Thakur, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. Lusk, and J. Larsson Träff, "MPI at exascale," *Procceedings of SciDAC 2010*, vol. 2, Jan. 2010.

[167] X. Tian, H. Saito, E. Su, A. Gaba, M. Masten, E. Garcia, and A. Zaks, "LLVM framework and IR extensions for parallelization, SIMD vectorization and offloading," in *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, Nov. 2016, pp. 21–31.

[168] X. Tian, H. Saito, E. Su, J. Lin, S. Guggilla, D. Caballero, M. Masten, A. Savonichev, M. Rice, E. Demikhovsky, A. Zaks, G. Rapaport, A. Gaba, V. Porpodas, and E. Garcia, "LLVM compiler implementation for explicit parallelization and SIMD vectorization," in *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM-HPC'17. New York, NY, USA: ACM, Nov. 2017, pp. 4:1–4:11.

[169] "Green500 list," Top500, Nov. 2019. [Online]. Available: https://www.top500.org/green500/lists/2019/11/

[170] "The LINPACK benchmark," Top500, 2019. [Online]. Available: https://www.top500.org/project/linpack/

[171] "Top500 list," Top500, Nov. 2019. [Online]. Available: https://www.top500.org/lists/2019/11/

[172] UPC Consortium, D. Bonachea, and G. Funck, "UPC language and library specifications, version 1.3," Nov. 2013.

[173] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, Jan. 2013.

[174] M. Voss, "CPUs, GPUs, FPGAs: Managing the alphabet soup with Intel Threading Building Blocks," Intel Webinar, Jun. 2017.

[175] S. Wienke, J. Miller, M. Schulz, and M. S. Müller, "Development effort estimation in HPC," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2016, pp. 107–118.

[176] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009.

[177] M. Wolfe, "Compilers and more: Exascale programming requirements," Apr. 2011.

[178] ——, "Compilers and more: OpenACC to OpenMP (and back again)," Jun. 2016.

[179] ——, "Compilers and more: What makes performance portable?" Apr. 2016.

[180] M. Wolfe, S. Lee, J. Kim, X. Tian, R. Xu, S. Chandrasekaran, and B. Chapman, "Implementing the OpenACC data model," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2017, pp. 662–672.

[181] M. Wolfe, "Implementing the PGI Accelerator model," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3. New York, NY, USA: ACM, Mar. 2010, pp. 43–50.

[182] M. Wong, A. Richards, M. Rovatsou, and R. Reyes, "Khronos's opencl sycl to support heterogeneous devices for c++," 2016.

[183] C. Yang, R. Gayatri, T. Kurth, P. Basu, Z. Ronaghi, A. Adetokunbo, B. Friesen, B. Cook, D. Doerfler, L. Oliker, J. Deslippe, and S. Williams, "An empirical roofline methodology for quantitatively assessing performance portability," in *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, Nov. 2018, pp. 14–23.

[184] G.-W. Yang and H.-H. Fu, "Application software beyond exascale: challenges and possible trends," *Frontiers of Information Technology & Electronic Engineering*, vol. 19, no. 10, pp. 1267–1272, Oct. 2018.

[185] J. Zhao and V. Sarkar, "Intermediate language extensions for parallelism," in *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11*, ser. SPLASH '11 Workshops. New York, NY, USA: ACM, Oct. 2011, pp. 329–340.

[186] W. Zhu, Y. Niu, and G. R. Gao, "Performance portability on EARTH: a case study across several parallel architectures," *Cluster Computing*, vol. 10, no. 2, pp. 115–126, Jun. 2007.