INVITED PAPER

Autotuning in High-Performance Computing Applications

This paper discusses how to make automatic performance tuning a standard technique for high-performance computing applications.

By Prasanna Balaprakash, Jack Dongarra, *Fellow IEEE*, Todd Gamblin, *Member IEEE*, Mary Hall, *Senior Member IEEE*, Jeffrey K. Hollingsworth, *Senior Member IEEE*, Boyana Norris, and Richard Vuduc, *Member IEEE*

ABSTRACT | Autotuning refers to the automatic generation of a search space of possible implementations of a computation that are evaluated through models and/or empirical measurement to identify the most desirable implementation. Autotuning has the potential to dramatically improve the performance portability of petascale and exascale applications. To date, autotuning has been used primarily in high-performance applications through tunable libraries or previously tuned application code that is integrated directly into the application. This paper draws on the authors' extensive experience applying autotuning to high-performance applications, describing both successes and future challenges. If autotuning is to be widely used in the HPC community, researchers must address the software engineering challenges, manage configuration overheads, and continue to demonstrate significant performance gains and portability across architectures. In particular, tools that configure the application must be integrated into the application build process so that tuning can be reapplied as the application and target architectures evolve.

KEYWORDS | High-performance computing; performance tuning programming systems.

I. INTRODUCTION

Since the first petascale supercomputer nearly a decade ago—the RoadRunner comprised of standard AMD64 multicores and custom IBM Cell processors—we have witnessed a diversity of supercomputing architectures that pose significant challenges for scientific application developers. Indeed, the four most powerful supercomputers in the world at the end of 2017—TaihuLight, Tianhe-2, PizDaint, and Gyoukou—rely on fundamentally different processor architectures from distinct hardware vendors. Because of profound differences in architecture and programming models, high-performance applications must be optimized and frequently rewritten in an architecture-specific way to attain acceptable performance. A manual code rewrite necessitated each time a new supercomputer architecture or architecture generation enters the scene is prohibitively expensive and limits the porting of applications to new platforms.

Clearly, a desirable feature of high-performance applications is *performance portability*, whereby the same application code can achieve high performance across a diversity of architectures. Performance portability is currently difficult to achieve. Even with programming model changes that enable this diversity of target architectures to be expressed, such as support for CPUs and GPUs in OpenMP 4, a code that targets one type of platform still may not perform well on another.

Manuscript received July 7, 2017; revised December 7, 2017; accepted January 15, 2018. Date of publication July 31, 2018; date of current version October 25, 2018. The work of P. Balaprakash and M. Hall was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. The work of J. Dongarra was supported by the National Science Foundation under Award ACI-1642441. A portion of the work of T. Gamblin was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. The work of M. Hall, J. K. Hollingsworth, and B. Norris was supported by the U.S. Department of Energy, Office of Advanced Scientific Computing Research (ASCR), Scientific Discovery through Advanced Computing (SciDAC) program under Award ER26054. The work of M. Hall was additionally supported by National Science Award SHF-1564074. The work of J. K. Hollingsworth was additionally supported by the ASCR X-Stack Project under Award ER26143 and the Department of Defense through a contract with the University of Maryland. *(Corresponding author: Mary Hall.)*

P. Balaprakash is with the Argonne National Laboratory, Argonne, IL 60439 USA.

J. Dongarra is with the University of Tennessee, Knoxville, TN 37996 USA, with the Oak Ridge National Laboratory, Oak Ridge, TN 37831 USA, and also with the University of Manchester, Manchester M13 9PL, U.K.

T. Gamblin is with the Lawrence Livermore National Laboratory, Livermore, CA 94550 USA.

M. Hall is with the University of Utah, Salt Lake City, UT 84112 USA (e-mail: mhall@cs.utah.edu).

J. K. Hollingsworth is with the University of Maryland, College Park, MD 20742 USA.

B. Norris is with the University of Oregon, Eugene, OR 97403 USA.

R. Vuduc is with the Georgia Institute of Technology, Atlanta, GA 30332 USA.

Digital Object Identifier 10.1109/JPROC.2018.2841200

Suppose that a single implementation is written by application developers and that porting of that application to different architectures is somehow automated. If successful, this strategy eliminates the high-performance computing (HPC) programmers' burden in managing architectural diversity. This paper describes such an approach, called *autotuning*, which involves automatic generation of a search space of possible implementations of a computation that are evaluated through models and/or empirical measurement to identify the most desirable implementation. Although autotuning is usually employed to reduce execution time, multiobjective tuning may optimize across a variety of criteria including performance, energy efficiency, peak power, or reliability. The impact of using autotuning as opposed to manual tuning includes increased programmer productivity, ease of porting to new platforms and, in some cases, better-performing applications.

This paper describes our experiences working with HPC application developers to develop autotuning technology that meets these goals and is compatible with the development of HPC production codes. In this paper we examine the state of the practice in incorporating autotuned code into HPC applications, insights from prior work, and the challenges in advancing this technology into wider and long-term use.

To examine the various ways in which autotuning can be employed, we first categorize a number of aspects of autotuning. The subsequent sections describe existing autotuning tools, followed by case studies from prior work that illustrate the strengths and weaknesses of approaches with respect to these aspects. The last two sections discuss the software engineering challenges and future directions that will increase the benefits of this valuable technology.

II. OVERVIEW

Fig. 1 presents an overview of the components of autotuning systems. Autotuning starts with an application or kernel of interest, along with a set of known tuning parameters. Tuning parameters are used in conjunction with the kernel to generate a set of versions or *variants* of the code. The goal of the autotuner is to select the *best-performing* variant from the search space described by the tuning parameters. Rectangles capture functionality that might be separate tools or tuning data. The figure also shows the differences in autotuning frameworks on the continuum from systems that perform autotuning at compile time (or tuning time) to runtime. The vertical bars delineate this range. For example, the first or leftmost vertical bar indicates that all analysis, modeling, empirical tests, and so on are deferred to actual program execution time, whereas the rightmost vertical bar indicates that all the autotuning is completed in an offline tuning phase. In the middle are online or incremental tuning, which occurs over one or multiple runs of an application, and runtime variant selection, where the selection of code to be executed is deferred to runtime based on a model derived from offline training.

Table 1 characterizes several of these aspects of autotuning in current use. The aspects considered cover the space of the autotuning literature and will be described in more detail, including tradeoffs in approaches, in the remainder of this paper.

A. How Packaged

At the heart of autotuning is a *search space* of code variants that are functionally equivalent to an original implementation. These are often packaged as libraries of commonly used numerical functions such as linear algebra and fast Fourier transform (FFT). However, one also can employ compiler and code generation tools that compose a collection of code transformations to generate optimized code or express the code variants at the application level. Recently, autotuning has been built into embedded domain-specific tools and execution frameworks, where the search space arises from the common framework.

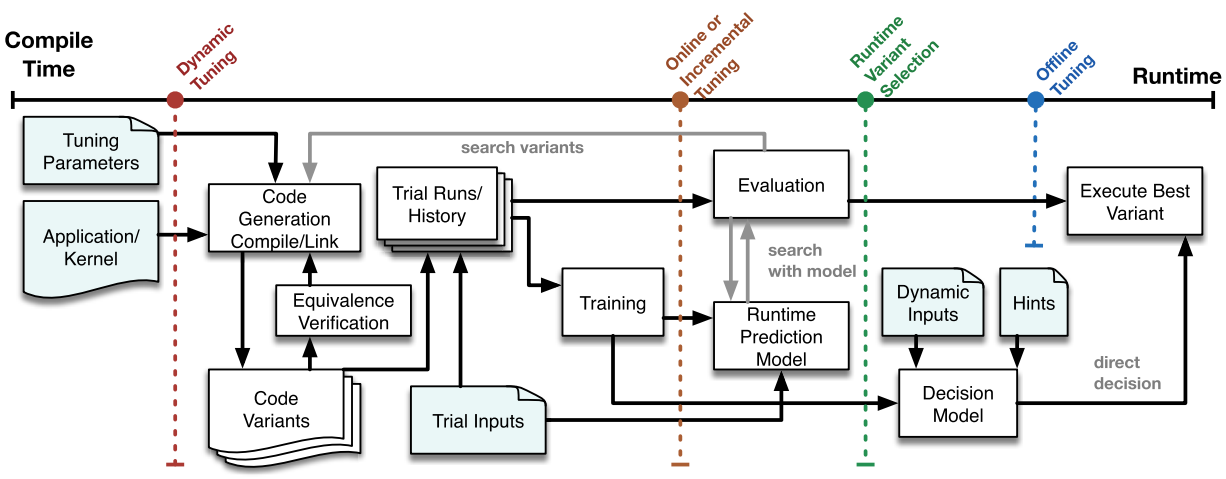

Fig. 1. *Components of autotuners on a continuum from compile time to runtime support, with examples.*

Table 1 Overview of Issues in Autotuning for High-Performance Computing

Approach	Definition	Examples
How Packaged		
Library	A library is tuned for different architectures and input data sets.	ATLAS, OSKI, MKL, FFTW, SPIRAL
Compiler-directed	The compiler generates multiple implementations and selects among them.	CHiLL, Orio, PetaBricks
Application-level	The programmer expresses parameters and code variants to select among.	ActiveHarmony, Orio, OpenTuner
Approach to Selection		
Model-based	A performance or ML model guides selection.	Performance model, Classifier model
Search-based	Empirical search to identify selection.	Heuristic search
Hybrid	A model prunes search space, followed by search	Prune undesirable areas of search space
Types of Decisions		
Algorithm selection	Select among alternative algorithms that are functionally equivalent.	PETSc, Trilinos, Lighthouse
Code variant selection	Select among fundamentally different implementations of an algorithm.	Data layout/representation
Parallelization	Select among different parallelization strategies.	SIMD(y/n), OpenMP vs. CUDA, multi-level
Code transformation	Select among different code transformation sequences.	Tile (y/n), Fusion (y/n), Wavefront (y/n)
Parameter tuning	Adjust context-specific parameters.	# of ranks/threads, Tile size, Unroll factor
When to Apply		
During porting	Library/application migrates to new architecture.	Adjust data layouts and parameters
Offline as needed	Search or training phase explores mapping selection.	Code targets new context
Online, incrementally	Autotuning decisions improved over application runs.	Expand search space for context
Dynamic, runtime	Consult model to select appropriate implementation.	Based on input features
Integration into Application		
Tuned code	Tuned code is inserted into the application.	Architecture-specific code
Selectable code	Multiple implementations and a selector.	Input-dependent selection
Dynamic	Code is dynamically generated.	Too many variants
Full tuning integration	Tuning occurs as part of build.	Compact code and forward scalability
Runtime Measurement		
Instrumentation	Gather standard measurements transparently to the user.	Dyninst, TAU, HPCToolkit
Semantic annotation	Tools or user direct instrumentation.	Caliper
Performance database	Collect measurements in a database.	TAUdb

B. Approach to Selection

Autotuners can select among code variants in many different ways. The simplest approach is to execute each code variant, measure its runtime (or other objective function), evaluate the performance of all variants, select the best one, and include that variant in the final code to be run. These are called *empirical* autotuners. Because the search space may be large, intelligent search methods and models may be used to iteratively prune the variant space as evaluation takes place. Instead of executing trials directly, some autotuners may train models from trial executions or from historical data. A runtime prediction model can be used as a proxy for real kernel executions, which allows a tool to more rapidly search the tuning parameter space, especially for long-running kernels. Models arising from training are particularly useful when selection depends on input data or other aspects of execution context; such decision models are consulted at runtime to select variants based on contextual features. Application developers may also embed hints in their code to influence the choice of variant at runtime.

C. Types of Decisions

Code variants may affect code organization, data structures, high-level algorithms, and low-level implementation details. In order to achieve performance portability, decisions on parallelization (how much and how many levels of parallelism) and memory hierarchy optimizations (e.g., data placement, blocking/tiling and tile size) will necessarily depend on the architecture. If the transformations used in code generation can alter the behavior or accuracy of the kernel, a functional equivalence verification step may be added to ensure that each variant yields correct results. Incorrect variants may be regenerated or excluded from consideration.

D. When to Apply

Depending on the architecture, the application domain, and the type of code being tuned, the components can be implemented in many ways as shown in Fig. 1 by the dashed lines. An extensive autotuning search can be expensive. Consequently, autotuning is typically performed offline, prior to application execution (rightmost dashed line). The end user can be completely uninvolved in autotuning if it is applied only once each time the software is ported to a new architecture by the library or application developer (not shown). If the application is being modified, then offline autotuning is needed each time the tuned computation or its context changes (e.g., a new data layout changes the performance). At the other end of the spectrum, the entire search is executed at runtime, with help from dynamic compilation servers to generate different code variants (leftmost). Alternatively, when an end user is unable or unwilling to participate in an offline or dynamic tuning phase, a changing application must be incrementally tuned. This requires significant additional infrastructure that includes a performance database that records the history of prior measurements as well as dynamic compilation and linking (second from left). Other tools take a hybrid approach, deferring variant selection

decisions until runtime, and leverage training data from prior or training runs (third from left).

E. Integration Into Applications

The most common use of autotuning in production HPC codes is to directly incorporate the tuned code, particularly libraries, into the application through either compilation or linking. In this way, the code does not change; once the code is verified, the application developer can be confident that the code is correct. This approach has two disadvantages, however. First, the code resulting from autotuning will be architecture specific (and possibly unreadable if automatically generated), and so the goal of performance portability is lost. Second, because the tuned code was generated for a specific execution context, over time it will become less optimized or unsuitable for new architectures or in conjunction with application changes. Where multiple code variants may be appropriate depending on execution context, these may all be compiled into the application, along with a selection function that decides which code variant to execute. When the desired code variants become too large to compile into the application, then online code generation, compilation, and linking are also required for dynamic tuning.

F. Runtime Measurement

For tuning tools that rely on dynamic feedback at runtime, measurement tools need to be integrated with the application as well as autotuned code. This integration can be accomplished with binary instrumentation/profiling tools such as Dyninst [1], or with hints from the application using semantic annotation tools such as Caliper [2]. In the former case, measurement is transparent, whereas tools like Caliper are designed to collect useful tuning hints from the developer. For example, the Apollo system described in the next section can build tuning models that make different decisions based on the particular physics phase or solver iteration of a simulation. Caliper simplifies the correlation of annotations at multiple levels of the software stack. Once collected, performance data can be stored in a database for retrieval in future experiments, with a naming scheme that clarifies the point in the search space for each measurement and adequate provenance information to understand its relevance, for example, TAUdb [3].

III. EXISTING AUTOTUNING TOOLS

We describe in more depth a collection of autotuning tools that have been used for autotuning HPC applications.

A. Libraries

The first autotuning systems were packaged as libraries. This choice was enabled in part by layered library designs in which a relatively small number of performance-critical subroutines could be isolated behind a standard application program interface (API), which hardware vendors could then tune for their platforms. However, wherever such platform-specific implementations were unavailable, too costly, or not fast enough, researchers naturally studied techniques to generate and tune a library implementation automatically.

One of the earliest motivating examples of such an API was the Basic Linear Algebra Subprograms (BLAS) standard, which defines a core set of primitives to support dense linear algebra [4]. On top of the BLAS, it then becomes possible to build fully featured linear algebra, such as the widely used defacto standard, LAPACK and its associated libraries ScaLAPACK, CLAPACK, and LAPACK95. Within the BLAS, one of the most important primitives is the general matrix multiply or *GEMM* operation [5]. The GEMM subroutine became an immediate target for autotuning in the early 1990s, including the PHiPAC and the Automatically Tuned Linear Algebra Software (ATLAS) systems [6], [7].

To see how a typical library autotuner system might work, consider ATLAS as an example. It generates efficient code by running a series of timing experiments using standard performance engineering techniques (e.g., loop unrolling and blocking) to determine optimal parameters and code structures. The process begins with detecting specific hardware properties, such as cache sizes and the floating-point pipeline length. Then, ATLAS systematically explores the different possible implementations (of, say, GEMM) of which there can be hundreds of thousands of variations. After eliminating unlikely candidates by using heuristics based on gathered information, ATLAS generates code to implement the remaining choices and then compiles, executes, and measures their execution time to choose the fastest. One also can extend the methodology to work on modern platforms, such as GPUs [8], which provide end-user developers a measure of performance portability. The result is that the best implementations often achieve large fractions of the highest possible performance for a given platform.

Library autotuners are, by design, domain specific. They exploit domain knowledge aggressively to improve performance. As the ATLAS GEMM example demonstrates, constraining the tuning problem to a specific kernel can make it tractable to enumerate, prune, and then explore a space of candidates. Beyond code, domain knowledge can also enable the exploration of variants at a higher level. For instance, the SPIRAL system encodes linear signal transforms symbolically and then uses a symbolic algebra engine to derive candidate algorithms [9]. Symbolic algebraic techniques have also been applied to automatic derivation of linear algebra methods [10], [11]. These methods use structural information that is usually unavailable to more general-purpose code transformation systems.

Additionally, domain-specific knowledge can extend to inputs. Examples include the Sparsity and OSKI systems for sparse matrix computations [12], [13], as well as more recent extensions for sparse direct solvers and sparse

tensor computations [14], [15]. These systems use combined knowledge about the computational kernels, the likely runtime use-cases and costs, and the kinds of input patterns likely to occur. This information is used to design specific methods for analyzing the input and selecting a data structure and tuned implementation at runtime. Being specific to the library, these methods can be both effective but also hard to generalize.

A library is also a natural setting in which to try to exploit execution history with little or no user intervention. For instance, consider the FFTW library system for autotuning fast Fourier transforms [16], [17], developed independently at around the same time as PHiPAC and ATLAS. It had a "wisdom" feature that tracked the performance of execution candidates and used that information to improve timing over subsequent executions. However, this process had the downside of introducing variable or unpredictable execution time cost.

Many ways exist for generalizing the techniques described above outside of a domain-specific library, including within the compiler, runtime system, or application. Over time, the simple code generators of early systems have adapted ideas from compilers, producing domain-specific compilers [18]–[20]. However, library methods may still be preferable, at least initially, in certain contexts, such as when selecting among candidate sparse iterative solvers or settings when explicit reasoning about numerical accuracy tradeoffs are necessary.

More generally, packaging autotuning within a library has several practical benefits related to use, distribution, and maintenance. First, applications naturally rely on libraries, and a library API forms a natural abstraction for isolating parts of the program that might need platform-specific tuning. Second, if that API is already widely used, as is the case with the BLAS, using an autotuned library can be as simple as linking to a different implementation. Third, an end-user developer can also rely on a library when it is backed by a community standard, even if the autotuned version becomes outdated or otherwise unavailable, reference implementations or vendor-provided alternatives exist.

However, packaging as a library can play out in different ways. The widespread adoption of ATLAS was encouraged, at least in part, by the fact of and interest in hardware-specific implementations of the BLAS. In the case of FFTW, the first library-based autotuner for (FFT) computations, it actually became a standard way to call the fast Fourier transform (FFT), in part because there was no standard.[1]

By contrast, in the domain of sparse linear algebra, the community has arguably not converged on an API for its core computational kernels, despite community efforts to develop one.[2] One explanation is that the basic data structures and calling sequences of sparse matrix computations have much more variety, which has led to many candidate libraries and APIs, but no convergence on a single one. Furthermore, proper tuning may involve reconsidering the choice of data structure in an input-dependent way, which may involve a potential runtime cost to deploy a sparse matrix autotuner that an application developer must now consider. Consequently, despite the development of autotuners for certain sparse matrix primitives [12], [13], they typically have not been integrated into applications or widely used sparse solver libraries.

Notably, the history of library-based autotuning is long enough that we can look back on some of the early systems and assess their impact. The PHiPAC, ATLAS, and FFTW systems have received impact awards for the original papers in their respective publication venues.[3] And beyond the ideas, the resulting software has been just as important. ATLAS enjoys wide use and has been included as a part of several Linux distributions. Before ATLAS, vendors charged significant prices for their tuned libraries, which discouraged some independent software vendors from using the work in their products. ATLAS removed this obstacle, a move that had significant implications for commercial software. Similarly, FFTW also has many users and received a major prize for numerical software.[4] Indeed, vendors have even adopted autotuning methodologies as part of their library-building processes. These include the Cray Scientific Library (LibSci), Intel MKL and its more recent library tuned for small matrices [25], and NVIDIA's cuFFT, which like Intel MKL adopts an FFTW-like interface.

B. Compilers and Code Generators

While libraries can encapsulate common computations and eliminate the need for programmer involvement in autotuning, libraries are limited in the scope of their applicability and the contextual information that allows composition of optimizations beyond individual library calls. For computations for which a library is unavailable or too limited, autotuning compilers and code generators such as CHiLL [26], [27], Orio [28] and POET [29] can potentially generate a collection of architecture-specific codes from the same high-level input. Parallel code generation can include parallelization (via SIMD pragmas, OpenMP, CUDA, etc.). Other transformations for HPC codes, available in compilers but commonly applied during manual tuning, include loop tiling (often called blocking by application developers), loop unrolling, loop permutation,

[1]For instance, Intel's Math Kernel Library (Intel MKL) and NVIDIA's cuFFT adopted FFTW's interface. Regarding an official standard, DARPA supported an effort to develop the Vector Signal and Image Processing Library (VSIPL) API, but that was still being actively discussed at the time of FFTW's release.

[2]For example, the BLAS standards committee has defined an interface for sparse computational kernels, but it has not been widely adopted.

[3]FFTW was recognized as a "Most influential PLDI paper" in 2009 [21]. PHiPAC received the "most influential paper in 25 years" award in 2014 [22]; ATLAS received a Best Paper award from ACM/IEEE Supercomputing (SC) in its publication year, as well as a Test-of-Time Award in 2016 [23].

[4]In particular, FFTW received the Wilkinson Prize for numerical software in 1999 [24].

fusion, distribution, prefetching, and software pipelining. Data transformations may also be applied to reorganize the data layout or copy it to other memory structures. The decisions that must be resolved during autotuning include which transformations to apply and in which order, as well as adjusting values for parameters of the optimizations, such as number of parallel threads/ranks, tile size, unroll factor, or prefetch distance.

Historically, compiler optimization decisions have been based on analytical models and heuristics. These decisions are governed by a do-no-harm philosophy such that they are not performed in cases where they may slow down common workloads. Using autotuning, a compiler may be far more aggressive and tailor optimization to the needs of a specific application running on a specific target architecture, and thus it is more likely to achieve the performance of manual tuning.

To integrate autotuning into a compiler framework requires a search space of possible implementations of each computation. Such a search space potentially can be generated automatically by a compiler decision algorithm, but this is a difficult challenge for general applications and architectures. Some success has been achieved when specialized for a specific application or application domain and a specific architecture or class of architectures, such as the Nek5000 example in Section V. As another example, Orio has been used to optimize data layouts and generate optimized sparse linear algebra computations on GPUs for finite-difference stencil-based solution of partial differential equations [30], [31]. From this knowledge, the search space of desirable optimizations can lead to a fixed decision algorithm that fully automates the tuning process. For other applications, however, the search space is unknown, or automating the decision algorithm is premature until an expert programmer, compiler developer or machine learning algorithm has figured out what the search space should be. Therefore, it is desirable to design autotuning compiler and code generation frameworks that are configurable and permit description of the search space by expert users.

The current state of the art in expressing a search space of transformations encodes these in scripts or *transformation recipes* [28], [32], [33]. Tools such as POET and Xevolver even support programmers' expression of the transformations to be applied [29], [34]. Users of such systems annotate loop nest computations with possible transformations and the set of associated parameters. From a set of such recipes or an encoding of multiple recipes, a large collection of code variants can be described and searched by using the techniques described below with regard to selection approach.

As shown in the case studies and other work, compiler-directed autotuning can produce code that achieves performance comparable to and sometimes exceeding that of manual tuning. The strength of such an approach lies in the ease of exploring completely different implementations that would be time-consuming for a programmer to produce. In particular, it can optimize more of an application than the time-consuming portion that is a programmer's focus, and can try more combinations of optimizations. The future of compiler-directed autotuning requires automating or encapsulating derivation of the search space so that nonexperts can benefit from the technology without having to interact with it directly.

C. Application-Level Autotuning

Autotuning may also be specified at the application level, and many programming systems have been developed that permit expression of tunable parameters and code variants representing alternative implementations [27], [35]–[37]. The advantage of specifying what to tune at this highest level of semantics is that significant algorithmic changes can be expressed. For example, fundamentally different approaches to solving the problem can be encoded for the autotuner. A solver with better performance or convergence properties may be selected, or a sort algorithm can be tailored to its input data set (see Section III-D). Libraries and compilers or code generators are unable to provide such a dramatic change to the program.

A distinguishing characteristic of application-level autotuning systems is the criteria for selecting the appropriate code variant. Much of the prior work selects among different implementations based on problem size. For example, PetaBricks [35] and Sequoia [36] are designed to recursively decompose algorithms to target different levels of the memory hierarchy or parallelism, with autotuning used to find the inflection points based on problem size for selecting among implementations. For this purpose, offline autotuning can be used to build a table of implementations, and runtime code variant selection then involves a simple table lookup based on problem size. However, suppose code variant selection is dependent on the input data set, known only at runtime. Recent systems have developed selection criteria for input-dependent code variant selection, where programmers express code variants along with metainformation that aids the system in variant section at runtime; a training phase constructs a selection model using machine learning, and this model is consulted at runtime when a new input is presented to make the selection [37]–[39]. Alternatively, runtime selection can be achieved through dynamic tuning; Active Harmony [40] and ADAPT [41] are capable of creating, compiling, linking, and testing new code variants in parallel with execution during iterative computations and replacing default implementations when better variants are found.

Another advantage of application-level autotuning is that it allows the use of autotuning options that can change the output of the program. In many cases, the accuracy of the computer simulation is limited based on uncertainties in the underlying physical system. A domain expert is aware of these limits and can specify tunable parameters that can change the answer but ensure that

any such changes are within the limitations of the inherent uncertainty of the simulation. For example, autotuning the GS2 plasma physics code [42] involved tuning the number of grid points and the energy grid. Both parameters can change the answer. However, domain experts assisted in this process to constrain the permissible range of values for these parameters so that any changes in the answer were acceptable. This additional freedom resulted in an extra 30% reduction in the program's runtime compared to the best autotuned version that only considered parameters that left the answer the same.

However, a disadvantage of making changes at the application level is that each application developer must specify the autotuning. When autotuning is done within common libraries or by the compiler, autotunable transformations can be specified once and shared by multiple applications.

D. Frameworks and Domain-Specific Systems

An increasingly common strategy for achieving high performance and performance portability on HPC applications is using specialization of high-level code for particular architectures and application domains. For performance portability, application developers need tight integration *and* the ability to adapt over time. Performance portability frameworks such as RAJA [43] and Kokkos [44] are becoming more popular in HPC, and they provide C++ template abstractions around application loops and data structures. These frameworks typically abstract a loop as a template function taking a *lambda* function as an argument along with several *policy* template parameters to control how the lambda is to be executed. This approach clearly separates tuning concerns from application semantics: application developers can write loop bodies in the code context where they are relevant, and performance experts can write hardware-specific code in policy implementations. Without a tool like Apollo [45], frameworks like these are limited to static tuning decisions. Apollo allows template instantiations to be treated as code variants and to be compiled with the application code. In addition, it allows code variant selection to be implemented as an external library, so that decision models can be updated over time. This is a useful compromise between libraries and direct compilation, and it can be combined with online code generation if the number of variants grows too large.

Another emerging approach that capitalizes on specialization performs optimizations for particular application domains, where the optimizations that are effective are known and autotuning is used to fine-tune optimization decisions or retarget to different architectures. For example, Halide, an embedded domain-specific language (DSL) for image-processing workflows, emerged from the research community [46] and is now in production use by Adobe and Google. Many DSLs have been developed, particularly for stencil computations [19], [20]. Tools for building DSLs have also emerged [47]. In spite of this significant progress, however, DSLs are not widely used in HPC, for many of the adoption reasons that will be discussed in Sections VI and VII.

IV. SEARCH: MODEL-FREE, MODEL-BASED, AND HYBRID SELECTION

For all the different ways that autotuning search spaces arise, as described in the preceding section, a mechanism is needed to evaluate some points within that search space in order to arrive at an optimized solution. In this section, we describe various approaches to explore the autotuning search space.

To find good parameter configurations, some autotuners perform complete enumeration either of all possible parameter configurations or of a pruned set of parameter configurations obtained by exploiting expert knowledge and architecture-specific and/or application-specific information. Examples include application-specific autotuners such as lattice Boltzmann computations [48], stencil computations [49], and matrix multiplication kernels [50], [51].

The main drawback of these autotuners is scalability; as codes and architectures become more complex, the number of tunable parameters and parameter configurations grows rapidly. Consequently, the computation time needed to enumerate all parameter configurations in a large decision space is prohibitively expensive. Hence, effective autotuners that examine a small subset of possible configurations are required. Two classes of algorithm exist: *model-free* and *model-based* algorithms.

Model-free algorithms do not use models to navigate the search space to find high-performing configurations. These algorithms can be grouped into global and local search algorithms. Global algorithms are characterized by their dynamic balance between exploration of the search space and exploitation of the accumulated search history. Examples include simulated annealing, genetic algorithms, and particle swarm optimization. They are theoretically guaranteed to find the globally best configuration at the expense of a long search time. In practice, however, they are run until a user-defined stopping criterion is met. In contrast, local search algorithms do not emphasize exploration and instead repeatedly try to move from a current configuration to a nearby improving configuration. Typically, the neighborhood of a given configuration is problem-specific and defined by the user or algorithm. These algorithms terminate when a current configuration does not have any improving neighbor and hence is locally optimal. The disadvantage of local search algorithms is that, depending on the search space and initial configuration, they can terminate with a locally optimal configuration that performs much worse than the globally optimal configuration. Examples include the Nelder–Mead simplex, orthogonal search, and variable neighborhood search.

Several global and local search algorithms have been deployed for autotuning. Seymour *et al*. [52] performed an experimental comparison of several global (random search, a genetic algorithm, simulated annealing, particle swarm) and local (Nelder–Mead and orthogonal search)

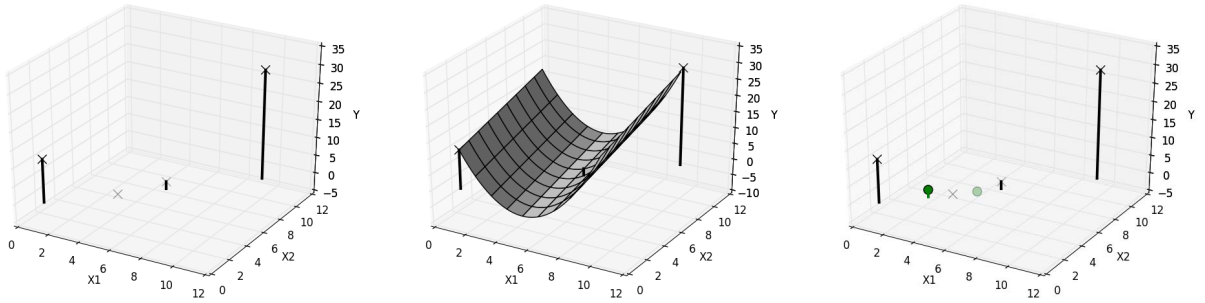

Fig. 2. ***Illustration of model-based search. (left) At each iteration, the algorithm considers a set of evaluated configurations, (middle) fits a surrogate model, and (right) evaluates configurations that are predicted to be high-performing by the model.***

optimization algorithms. Similarly, Kisuki *et al*. [53] compared random search, a genetic algorithm, and simulated annealing with pyramid search and window search. In both these studies, the experimental results showed that the random search was more effective than the other algorithms tested. The reason is that in the tuning tasks considered, the number of high-performing parameter configurations is large and hence finding one of them is easy. Moreover, in all these works the local search algorithms are less effective since they were not customized. Norris *et al*. [54] implemented the Nelder–Mead simplex method, simulated annealing, and a genetic algorithm in the empirical performance-tuning framework Orio. A number of previous works deploy local search algorithms for empirical performance tuning. Examples include orthogonal search in ATLAS [55], pattern search in loop optimization [56], and a modified Nelder–Mead simplex algorithm in Active Harmony [27], [57]. Balaprakash *et al*. [58] investigated the issue of global versus local search in autotuning using illustrative global and local algorithms under short computation times. The results showed that the exploration capabilities of global algorithms are less useful; given good initial configurations, local search algorithms can find high-performing code variants in short computation time. Moreover, poor initial configurations can significantly reduce the effectiveness of both global and local search algorithms that are sensitive to the starting point. When the available tuning time is severely limited, carefully customized local search algorithms are promising candidates for empirical performance tuning problems that have integer parameters and bound constraints.

The primary goal of model-based selection algorithms in autotuning is to avoid the cost of running code on the target machine by predicting performance metrics of a given parameter configuration. Analytical performance models, which use closed-form expressions for predicting performance metrics, have enjoyed significant success in the compiler optimization community for accelerating serial codes. However, this approach is limited by the quality and extrapolatory power of the analytical model, which often fails to capture complex interactions between the code, runtime systems, and architecture. Moreover, developing a complex mathematical model requires a wide range of expertise in the target system architecture, programming models, and scientific applications. Consequently, analytical models are less well suited for highly specialized kernels and libraries for scientific applications that require portability, scalability, and performance. Another analytical model-based autotuning approach involves analysis of the source or binary code of the implementation to estimate analytical model parameters. For example, a static analysis tool can extract control flow information and instruction counts from the compiled PTX code of a CUDA GPU implementation, from which one can estimate (analytically) metrics such as occupancy, which can be used to determine parameters such as thread counts and block sizes. While the autotuner still must generate different code versions that are compiled before the static analysis can be applied, this approach greatly reduces or in many cases completely eliminates the need for executing and timing code variants.

When analytical performance models become too restrictive for a given scientific workload and HPC architecture, empirical performance modeling is an effective alternative. In this approach, a small subset of parameter configurations (code variants) is evaluated on the target machine to measure the required performance metrics, and a predictive model is built by using machine learning approaches. Here, the choice of the supervised machine learning algorithm for building the surrogate performance model is crucial. Often this choice is driven by an exploratory analysis of the relationship between the parameter configurations and their corresponding runtimes. A typical model-based approach is a two-step process in which an analytical or empirical model is built first and a search algorithm is used to find high-performing configurations using the model.

In recent years, a new class of empirical model-based search has received considerable attention and has been shown to be effective for autotuning. This approach consists of sampling a small number of input parameter configurations and progressively fitting a surrogate model over the input–output space until exhausting the user-defined maximum number of evaluations. The surrogate

model is iteratively refined in the promising input parameter region by obtaining new output metrics at input configurations that are predicted to be high performing by the model [59]–[61].

V. CASE STUDIES

We describe a number of case studies from our prior work that illustrate the current role of autotuning in HPC applications.

A. Library Autotuning

Some libraries, such as PETSc [62]–[64], provide high-level data structure-independent interfaces that present an opportunity for seamless integration of new, optimized data structures and low-level operations without having to modify the application source code. Taking advantage of this design, researchers developed new matrix and vector data structures for PETSc stencil-based computations specifically targeting GPUs. Orio [30] was used to tune all matrix-vector operations involving the new data structure [31]. Because the matrix structure is known a priori and does not change, it can be represented by using a packed dense format (instead of the typical compressed sparse row format), which is more storage-efficient and eliminates the indirect memory accesses that make sparse matrix algebra difficult to optimize. Any application involving a finite-difference, discretization-based PDE solution on a regular grid can take advantage of the tuned implementations without any code modification. Moreover, because Orio generates size-specific optimizations, the resulting library generally performs better than manually optimized libraries, which typically do not provide size specialization. In an evaluation of a PDE application discretized by using a 3-D seven-point stencil, this approach achieved speedups of the matrix-vector computations ranging between 1.8 and 4.8 over vendor-optimized libraries (NVIDIA Cusp).

B. Solver Selection

Numerical toolkits such as PETSc and Trilinos [65] provide a large number of parallel solution methods for large sparse linear systems. In the Lighthouse project [66]–[68], machine learning was used to classify solution methods (solver-preconditioner pairs) based on a small number (fewer than ten) of easy-to-compute linear system features. The classifiers are built through sampling the solver space on a large and varied training set of linear systems. At runtime, the linear system features are used as input to the model (classifier) to obtain a list of solver configurations that are likely to perform well. This algorithmic autotuning does not require application code change and is integrated into the libraries' existing solver interfaces.

C. End-to-End Autotuning

Another exercise demonstrated an automated, end-to-end optimization of the SMG2000 benchmark, a semi-coarsening multigrid on structured grids [69]. This demonstration combined *outlining* using the ROSE compiler, transformation and code generation using CHiLL, and search space navigation with Active Harmony. With outlining, ROSE extracts computationally intensive loop nests into separate executable functions with representative input data that are to be the focus of autotuning. The outlined loop nests are then tuned by the framework and subsequently integrated back into the application. Each loop nest is optimized through a fixed series of composable code transformations (permute, tile and unroll), with the transformations parameterized by unbound optimization parameters that are bound by Active Harmony during the tuning process. When the full application is run using the code variant found by the system, overall performance improves by 27%.

D. Tuned Code Integrated Into the Application

Compiler-directed autotuning was used to optimize Nek5000, and the compiler-generated code was integrated into the production application [70]. Nek5000, a spectral element code, spends the bulk of its computation in matrix–matrix multiplication of small, rectangular matrices. Because BLAS libraries are typically optimized for large square matrices that exceed memory hierarchy capacity, there was significant opportunity to improve performance by specializing the generated code to the specific matrix sizes arising in the application. The optimizations applied focus on SIMD code generation (for Intel SSE), register reuse, and instruction-level parallelism. Therefore, CHiLL was used to apply loop permutation and loop unrolling, to achieve a loop order that was best suited for SSE, and expose register reuse and instruction-level parallelism to the Intel ICC native compiler. The small search space that arose was explored in a brute-force manner. As presented in [70], we observed overall speedups of up to 1.26X on the entire application running on 256 nodes of Jaguar at Oak Ridge, and the optimized code was integrated into the production application.

Recent work expanded the scope of this optimization of Nek5000 to the entire `local_grad3` calculation that subsumes the matrix-matrix multiplication, and targeting an NVIDIA GPU [71]. Here, CUDA-CHiLL (a thin CUDA layer added to CHiLL) and the SURF search algorithm in Orio were combined, along with a tensor contraction frontend DSL called Octopi, to fully automate the GPU code generation. This GPU code has not been adopted by the application developers at the time of this writing.

E. Performance and Programmer Productivity

Compiler-directed autotuning in CHiLL was used to optimize the Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) solver [72], and was shown to outperform by 3% a manually tuned code for the same algorithm [73]. Specifically, an important kernel within LOBPCG is the sparse matrix multivector multiplication (SpMM), which is a generalization of the SpMV kernel in

which a sparse m-by-n matrix A is multiplied by a tall and narrow dense n-by-k matrix B ($k \ll n$). While the manually optimized SpMM implementation was over 2000 lines of code, the input to CHiLL was only 7 lines of code, a 300× difference, thus improving programmer productivity. The optimizations required in CHiLL to replicate the manually tuned code were extensive. Using an inspector-executor approach, a data transformation converted the large, symmetric sparse matrix from a compressed sparse row format to a compressed sparse block (CSB) format. This representation was well suited for parallelization of the matrix and its transpose, since it permitted storing only the upper triangular portion. A number of additional transformations were added to CHiLL, inspired by the manually tuned code. In order to reduce the data movement associated with indices of the matrices, a short integer was used as the type for the matrices that pointed to the beginning of each CSB block. Targets for AVX SIMD code generation were marked with pragmas for the native Intel ICC compiler. To summarize, this experiment demonstrated that an autotuning compiler could generate high-performance sparse matrix code, but also that integration of transformations used in manual tuning could greatly enhance the capability of such compilers.

F. Vertically Integrated Autotuning

Fast Fourier transforms are a critical part of many parallel programs. FFTs require extensive communication and have floating point requirements that necessitate careful instruction sequences to achieve good performance. In addition, an effective implementation requires overlapping computation and communication. Because of the need to optimize these parameters for different processors and the tedious nature of properly tuning them, FFT has been a popular library for autotuning. Performance of FFT at scale depends on a combination of the node-level computation, communication requirements, and the underlying communication layer implementation, typically MPI. We refer to the use of autotuning at multiple software layers as *vertical integration*.

The Active Harmony system has been used to obtain a highly optimized FFT implementation using vertically integrated autotuning [74]. This approach includes a communication optimization technique called dynamic polling intervals. In many MPI implementations, to actually overlap computation and communication, programs must periodically call a polling routine to query whether a nonblocking communication operation has completed. This polling routine also must be called periodically to ensure that the MPI implementation actually transfers data. The frequency of polling can have a significant impact on performance. If the polling is done too often, it results in wasted effort. However, if it is not done frequently enough, communication can stall. The optimized FFT library allows this polling frequency to be tuned.

Overall, the autotuning approach encompasses 24 tunable parameters: two communication tile sizes, two communication window sizes, eight places where polling intervals are specified, and eight subtile sizes. Despite the large number of parameters (each of which have dozens of possible values), the Active Harmony system is able to converge to an optimal configuration after trying about 35 parameter combinations.

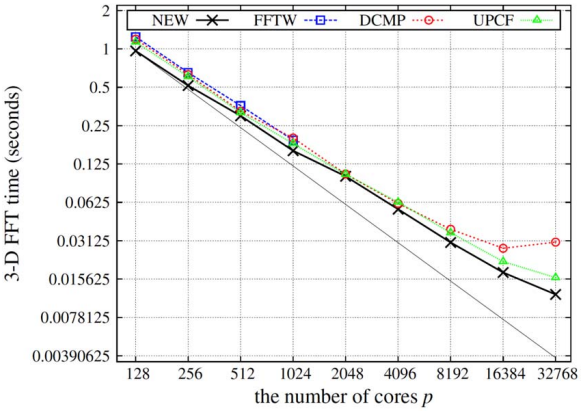

Fig. 3. *Strong scaling results for OFFT and other 3-D FFT libraries.*

Fig. 3 shows the results of using the optimized library `OFFT` compared with other 3-D FFT algorithms. The results shown are for strong scaling from 128 to 32K cores on NERSC's Edison system. The light gray line shows the ideal speedup. The solid black line shows the results for OFFT. The blue line shows the time for FFTW. Since the original FFTW supports only a 1-D data decomposition, the results for FFTW stop at 1024 cores. The red and green lines show the results for running using DCMP and UPCF, respectively. The OFFT results are always faster than those of the other implementations, and the performance advantage grows as the number of cores is increased.

G. Fast, Data-Dependent Autotuning

In multiphysics, multimaterial models and in adaptive mesh refinement (AMR) codes, the same kernel may be executed on very different data structures. A time step loop in an AMR code may iterate over thousands of patches, each of different sizes and aspect ratios; and multimaterial kernels may require more computation on certain parts of a mesh and not on others. Fig. 4(a) shows the range of runtimes for the top eight kernels in two hydrocodes: CleverLeaf, an Eulerian AMR hydrodynamics proxy application, as well as ARES, a multiphysics Arbitrary Lagrangian-Eulerian hydrodynamics code. The runtimes can vary by orders of magnitude depending on the input array sizes and aspect ratios and the particular execution model (OpenMP, sequential, GPU) used for each kernel.

These codes use RAJA [43], a performance portability framework developed at LLNL, which was extended with the Apollo [45] autotuning framework. Apollo allows a user to train a lightweight decision model using machine learning and to use it to select an execution model based

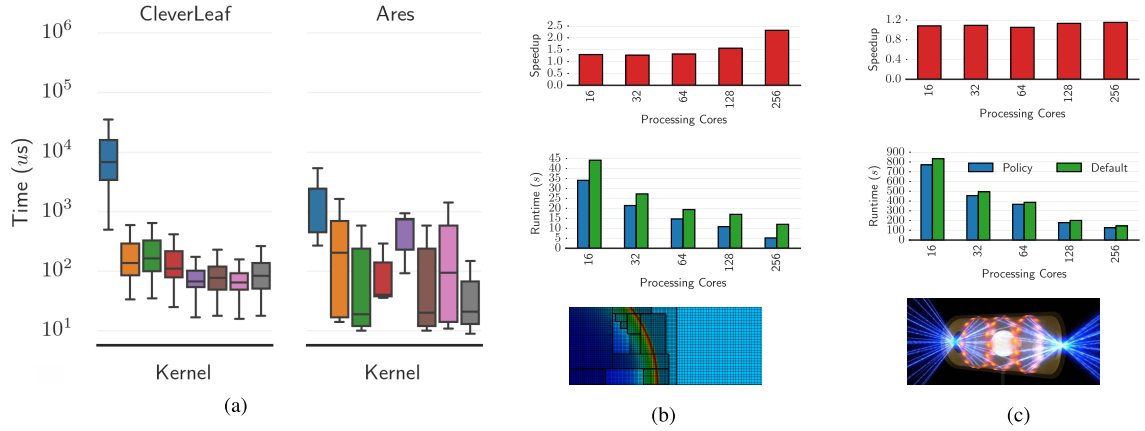

Fig. 4. ***Runtime of kernels in hydrocodes can vary by orders of magnitude depending on the execution policy (CPU, GPU, OpenMP, etc.) and input arrays. We are able to achieve considerable speedup by dynamically tuning the execution policy based on input data. (a) Runtimes of top kernels for different execution models. (b) CleverLeaf AMR, Sedov problem. (c) ARES, hotspot problem.***

on the input arrays each time control passes over a RAJA kernel. These experiments chose between running a kernel sequentially or with OpenMP. CleverLeaf running the Sedov problem [Fig. 4(b)] achieved up to 2.5x speedup for a run on 256 cores, with the entire code using Apollo. ARES 4c achieved up to 15% speedup on 256 cores, with only the Lagrange hydro phase using the autotuner. In these examples, speedup *increases* for larger-scale runs; the reason is that the domain becomes increasingly finely decomposed with strong scaling, and with small patches it is not worthwhile to pay the overhead of launching on-node OpenMP parallel regions. For these highly unstructured domains, the code may iterate over thousands of irregularly sized mesh patches in a single timestep, and one cannot possibly know the size and correct code variant without dynamic information.

VI. SOFTWARE ENGINEERING CHALLENGES FOR AUTOTUNING

For autotuning to succeed in production, it must integrate seamlessly into the application development process, and one must be able to maintain autotuned parts of large applications as the codes evolve. Indeed, the changes demanded of the software development process are the biggest obstacle to mainstream use of autotuning.

Because offline autotuning can involve many empirical compilations, it typically requires a significantly longer compilation process, and it may require execution of code variants on the target platform. Thus it can severely impact the speed with which a developer can iterate on the code. Having an option to disable lengthy compiles is essential. Perhaps more importantly, empirical autotuning burdens developers with the task of managing the tuning process. This can be tedious for developers, as they must now decide how often to retune and how to manage profile data.

Autotuners have historically complicated the build process, even for simple codes. Designing autotuners to leverage existing build infrastructure is ideal. For example, Orio can be used as a compiler wrapper (together with other wrappers such as those provided by MPI implementations) to enable a single build configuration to be used for regular development and for autotuning. For large codes, injecting compiler wrappers into all parts of the build may not be straightforward but we may be able to leverage the work already done in build-from-source package management systems such as Spack [75]. Spack can build over a thousand packages, and it provides a harness around each package's build system that injects compiler wrappers into the build. Depending on the host build system, Spack may set the `CC` variable, patch the build, or explicitly set the compilers in the build. This could be a useful integration point for autotuners and would enable codes with potentially hundreds of dependency libraries to be tuned easily.

Because the final autotuned binaries contain automatically generated code, debugging can be an issue, although the debugging of any application that uses libraries developed elsewhere is complicated by the presence of code that may be less familiar or for which source code is not available at all. The code generated by autotuners, on the other hand, is likely to have fewer bugs than do human-developed portions, and the availability of multiple versions also enables functional equivalence verification during autotuning.

VII. CHALLENGES AND FUTURE DIRECTIONS

To summarize the challenges for widespread adoption, autotuning must become a standard or at least a common part of the build process for HPC programmers. Making it transparent to end users seems to be the most desirable way to do this, but on the other hand autotuning is more likely to achieve high performance with some support from application or domain experts. Therefore, we envision a spectrum of possible interactions with autotuning by HPC

Table 2 Challenges to Autotuning Adoption

Challenge	Proposed Solution
Overhead Concerns	
1. Tuning search can be expensive	Improve search algorithms and use cutoffs
2. Specifying autotuning seach space is cumbersome	Automate or simplify specification
3. Selection and configuration of algorithms difficult	Simplify and automate modeling
4. Off-line tuning increases runtime, programmer burden	Incremental or dynamic tuning for full transparency
Scope of Applicability	
5. Tuning must be repeated for new contexts	Integrate into build process and anticipate execution contexts
6. Exascale resources will vary during execution	Models must adapt to changing resources
7. Achieving economies of data scale	Use common interfaces, cooperate with HPC facilities on modeling
Other Programmer Concerns	
8. Correctness of dynamically-changing code	Incorporate equivalence verification into execution
9. Long-term tool availability	Use simple tools or integrate into widely-used open source software

programmers, from complete transparency to complete control. Looking across this spectrum of options, Table 2 summarizes the challenges to adoption we foresee and some possible solutions, organized into three main categories: 1) overhead concerns, which refers to compile time, tuning time, and runtime overheads as well as increased programming complexity; 2) scope of applicability, which acknowledges that widespread use of autotuning requires that its scope of applicability must be expanded to new and dynamically changing execution contexts, learning from prior application runs; and 3) other programmer concerns, which we have collected from applications communities.

Perhaps the first concern that is often raised about autotuning is the inherent cost of searching across different implementations and its scalability for large application codes (line 1). Section IV addresses how improvements in search algorithms lead to better solutions and less search time, but the tolerance for search time may vary by user and tuning scenario. While we improve search algorithms and incorporate incremental tuning support as described in the next paragraph, it is also important to offer a hard cutoff, in terms of number of points to search or time to spend tuning, to limit the cost of autotuning.

For users who want to maintain some control of the autotuning process, ease of use will be advanced by improving the mechanisms by which autotuning variants and parameters are expressed (lines 2 and 3). Autotuning systems must first and foremost allow programmers to compactly describe a search space or derive it automatically. When selecting among code variants using a classifier, manually specifying features may also be too much of a programming burden, potentially costly and suboptimal. It is desirable to automate the feature collection, for example within a common framework like Apollo's collection of RAJA features from Section V-G or via a domain-specific framework. If the programmer is providing any of this information, then tools must support the programmer in logically mapping the vast performance or other data arising from autotuning back to programmer abstractions of the computation, so that the relationship between areas of the search space and optimization is understood. This latter capability can be used to train programmers to understand how to establish effective autotuning search spaces and participate in optimization.

Consider the requirements of users who may not want to be involved in the autotuning process at all. Such users will benefit from all the previous solutions whenever they are fully automated. Incremental or dynamic tuning (line 4) can hide the time spent in performing autotuning, as it is amortized over prior runs or within a single application run. Dynamic tuning must be sparing in how much work is done in a run; therefore, it could greatly benefit from prerun learning or integration with incremental tuning that accesses the measurements from prior runs. Another strategy for hiding the programmer burden of autotuning is to rely on domain-specific frameworks (e.g., Halide) that can be tuned by expert users. The effect of tuning can benefit other users, but they can use the harnesses or results of autotuning without directing it.

The next two challenges in the table (lines 5 and 6) involve changes in applications, their input data and resources, which we refer to as the *context* or *execution context* of an application. As discussed in the preceding section, autotuning must adapt to these changes and must therefore be integrated into an application's build process. Further, an exascale platform, because of energy management and component failure, may have varying resources available to an application for execution. In addition, offline tuning or training may need to be performed on a proxy system rather than the target architecture. Both challenges imply that autotuning decisions may need to predict expected resources at runtime and tune accordingly. Early work in predicting performance for unseen execution contexts has relied on information from prior tuning on source architectures that are different from the final target architecture [76], [77]. Such models work best when performance differences can be captured with proxies; for example, if synchronization costs are the dominant predictor for autotuning, then proxies for synchronization can be used to predict execution-time behavior. Findings to date, however, show that prediction is most effective when training or offline tuning occurs on a similar architecture or with related resources.

Building good tuning models requires exploring a large performance search space, and a single HPC user is unlikely to be able to train high-quality, general tuning models with only a limited set of applications and inputs. Users will be able to train robust performance models only if autotuning achieves broad exposure and if the performance of many different algorithms, inputs, and architectures are included in the training data (line 7). To gather such a corpus of data, we must make it easy to monitor code performance in a wide variety of contexts and to record this data in a way that can be shared among HPC users to train versatile models. However, typical HPC users do not want to be in the business of managing voluminous performance data or of controlling which runs are monitored and measured and which are not. The only way we can achieve this kind of scale is with help from the HPC facilities–who have visibility across the entire workload of their HPC centers and can deploy tools to measure a wide range of applications. This is a difficult task that requires not only technical work but also work to secure the performance data from potentially sensitive codes. Ideally, users could use simple, transparent interfaces to measure their production codes and to provide performance data securely to HPC facilities at runtime. Performance data management for autotuners should not require developers to manually manage any historical tuning data or model outputs.

Autotuning can alleviate the burden of finding a high-performance code variant, but application developers must trust that the optimized version of the code produces equivalent results, particularly if tuning is performed dynamically (line 8). The outputs need not match identically in order to be functionally equivalent, particularly in the presence of reordering operations like reductions. A number of ways exist for verifying functional equivalence of the tuned code. The most common simple approach is to compare output from some trusted version, with some specified tolerance for differences in results. A preferred solution may be to exploit domain knowledge of the algorithms to verify output. For example, LAPACK provides comprehensive error bounds for most computed quantities. It includes a table for various routines that describes the bound so a user can determine how accurate the computed solution is for a given problem. If autotuning was involved, the resulting software could be checked according to the same criteria. Other features of the output or execution could be used as a proxy for verification, such as number of iterations to convergence, or sensitivity to input perturbations. A more comprehensive approach could use code synthesis and formal correctness proofs to automatically generate complicated members of the design space, but this may be substantially more costly.

The final line reflects a common concern among HPC developers that impedes adoption of new technology. HPC applications have long lifetimes, sometimes spanning decades. In contrast, new technology goes through a lengthy process of exploring appropriate approaches before it gains traction. Many tools simply do not reach a level of maturity to support production applications. Other tools may no longer be supported once a funded project ends. Therefore, to adopt new technology, HPC developers must be convinced that technology will be available for the lifetime of the application. The only obvious solution to this problem is to integrate the technology into trusted and widely used open source software.

VIII. SUMMARY

Autotuning is a proven technology to achieve high performance and performance portability. In this paper, we have presented examples where autotuning tools have facilitated high-performance implementations. Programmer productivity is also enhanced when tools can simplify the code the programmer writes and eliminate the need for manually writing low-level architecture-specific implementations. As new and diverse architecture features continue to appear in exascale architectures and beyond, the need for approaches that improve performance portability and programmer productivity will grow stronger. We believe that autotuning is a powerful and appropriate technology for addressing this need.

The goal of this paper was twofold: to describe the state of the art in autotuning and to present future requirements to move autotuning toward mainstream use in HPC. We believe that many of the challenges in gaining widespread deployment of autotuning relate to ease of use (including overhead), predictability, and a reimagining of its integration into the application build process. We have seen other fundamental changes to application development in HPC succeed or fail based on these issues.

Looking to future architectures and applications, we believe that the size of the space of desirable code variants for autotuning likely will grow substantially as architectures change and software adapts in response. Today's applications are currently undergoing rewrites and other adaptations to target emerging many-core, GPU, and heterogeneous architectures. New ways of organizing algorithms, new data structures, and even fundamentally new algorithms with different numerical convergence and stability properties are appearing. Indeed, a new theory of communication-avoiding algorithms shows how to construct algorithms that do asymptotically less data movement, which are more efficient on current architectures where communication costs are increasingly dominant.

Beyond HPC, autotuning technology is highly relevant for some emerging computation-intensive workloads, such as deep learning and data analytics. In order to improve the single-node performance, deep learning requires highly optimized kernels for key computational operations. These include sparse matrix–vector, matrix–transpose–vector, matrix–vector–transpose, and matrix–matrix products. As the set of possible algorithms and implementations continues to grow, tools for more easily generating members of this set become important.

REFERENCES

[1] J. K. Hollingsworth, B. P. Miller, and J. Cargille, "Dynamic program instrumentation for scalable performance tools," in *Proc. Scalable High-Perform. Comput. Conf.*, May 1994, pp. 841–850.

[2] D. Boehme, "Caliper: Performance introspection for HPC software stacks," in *Proc. Supercomputing (SC)*, Salt Lake City, UT, USA, Nov. 2016, pp. 550–560.

[3] K. A. Huck, A. D. Malony, R. Bell, and A. Morris, "Design and implementation of a parallel performance data management framework," in *Proc. Int. Conf. Parallel Process. (ICPP)*, Jun. 2005, pp. 473–482.

[4] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for Fortran usage," *ACM Trans. Math. Softw.*, vol. 5, no. 3, pp. 308–323, Sep. 1979.

[5] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, Mar. 1990.

[6] J. Bilmes, K. Asanović, C. W. Chin, and J. Demmel, "Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology," in *Proc. Int. Conf. Supercomput.*, Vienna, Austria, Jul. 1997, pp. 253–260.

[7] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proc. ACM/IEEE Conf. Supercomput. (SC)*, 1998, pp. 1–27.

[8] J. Kurzak, S. Tomov, and J. Dongarra, "Autotuning GEMM kernels for the Fermi GPU," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 11, pp. 2045–2057, Nov. 2012.

[9] M. Püschel, "SPIRAL: Code generation for DSP transforms," *Proc. IEEE*, vol. 93, no. 2, pp. 232–275, Feb. 2005.

[10] V. Eijkhout, P. Bientinesi, and R. van de Geijn, "Proof-driven derivation of Krylov solver libraries," Texas Adv. Comput. Center, Univ. Texas Austin, Austin, TX, USA, Tech. Rep. TR-10-02, 2010.

[11] D. Fabregat-Traver and P. Bientinesi, "Application-tailored linear algebra algorithms: A search-based approach," *Int. J. High Perform. Comput. Appl.*, vol. 27, no. 4, pp. 425–438, Nov. 2013.

[12] E.-J. Im and K. A. Yelick, "Optimizing sparse matrix vector multiplication on SMPs," in *Proc. SIAM Conf. Parallel Process. Sci. Comput.*, San Antonio, TX, USA, Mar. 1999, pp. 1–9.

[13] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," *J. Phys., Conf. Ser.*, vol. 16, no. 16, pp. 521–530, 2005.

[14] P. Sao, X. Liu, R. Vuduc, and X. Li, "A sparse direct solver for distributed memory Xeon phi-accelerated systems," in *Proc. Int. Parallel Distrib. Process. Symp. (IPDPS)*, Hyderabad, India, May 2015, pp. 71–81.

[15] J. Li, J. Choi, I. Perros, J. Sun, and R. Vuduc, "Model-driven sparse CP decomposition for higher-order tensors," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May/Jun. 2017, pp. 1048–1057.

[16] M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *Proc. Int. Conf. Acoust., Speech, Signal Process.*, vol. 3, May 1998, pp. 1381–1384.

[17] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proc. IEEE*, vol. 93, no. 2, pp. 216–231, Feb. 2005.

[18] G. Baumgartner, "Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models," *Proc. IEEE*, vol. 93, no. 2, pp. 276–292, Feb. 2005.

[19] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *Proc. Symp. Parallel Algorithms Architect. (SPAA)*, Jun. 2011, pp. 117–128.

[20] M. Christen, O. Schenk, and H. Burkhart, "PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *Proc. IEEE Parallel Distrib. Process. Symp. (IPDPS)*, Anchorage, AK, USA, May 2011, pp. 676–687.

[21] M. Frigo, "A fast Fourier transform compiler," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, May 1999, vol. 34, no. 5, pp. 169–180.

[22] J. Bilmes, K. Asanović, C.-W. Chin, and J. Demmel, "Author retrospective for optimizing matrix multiply using PHiPAC: A portable high-performance ANSI C coding methodology," in *Proc. 25th Anniversary Volume Int. Conf. Supercomput. (ICS)*, 2014, pp. 42–44.

[23] R. C. Whaley and J. Dongarra (2016). *SC16 Test of Time Award Winner: Official Citation*. [Online]. Available: http://sc16.supercomputing.org/conference-components/awards/test-time-award-page/

[24] M. Frigo and S. G. Johnson (1999). *J. H. Wilkinson Prize for Numerical Software: Official List of Winners*. [Online]. Available: http://www.anl.gov/mcs/about-us/j-h-wilkinson-prize-numerical-software

[25] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "LIBXSMM: Accelerating small matrix multiplications by runtime code generation," in *Proc. ACM/IEEE Int. Conf. High-Perform. Comput. Netw., Storage Anal.*, Nov. 2016, pp. 981–991.

[26] C. Chen, "Model-guided empirical optimization for memory hierarchy," Ph.D. dissertation, Univ. Southern California, Los Angeles, CA, USA, 2007.

[27] A. Tiwari, C. Chen, C. Jacqueline, M. Hall, and J. K. Hollingsworth, "A scalable auto-tuning framework for compiler optimization," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, Washington, DC, USA, May 2009, pp. 1–12.

[28] A. Hartono, B. Norris, and P. Sadayappan, "Annotation-based empirical performance tuning using Orio," in *Proc. IPDPS*, May 2009, pp. 1–11.

[29] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan, "POET: Parameterized optimizations for empirical tuning," in *Proc. IPDPS*, Long Beach, CA, USA, Mar. 2007, pp. 1–8.

[30] A. Mametjanov, D. Lowell, C.-C. Ma, and B. Norris, "Autotuning stencil-based computations on GPUs," in *Proc. IEEE Cluster*, Sep. 2012, pp. 266–274.

[31] C. Choudary, "Stencil-aware GPU optimization of iterative solvers," *SIAM J. Sci. Comput.*, vol. 35, no. 5, pp. S209–S228, Oct. 2013.

[32] S. Donadio, "A language for the compact representation of multiple program versions," in *Proc. Workshop Lang. Compilers Parallel Comput. (LCPC)*, Oct. 2005, pp. 136–151.

[33] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan, "Loop transformation recipes for code generation and auto-tuning," in *Proc. 22nd Int. Workshop Lang. Compilers Parallel Comput.*, Oct. 2009, pp. 50–64.

[34] H. Takizawa, S. Hirasawa, Y. Hayashi, R. Egawa, and H. Kobayashi, "Xevolver: An XML-based code translation framework for supporting HPC application migration," in *Proc. 21st Int. Conf. High Perform. Comput. (HiPC)*, Dec. 2014, pp. 1–11.

[35] J. Ansel, "PetaBricks: A language and compiler for algorithmic choice," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implement. (PLDI)*. New York, NY, USA: ACM, 2009, pp. 38–49.

[36] M. Ren, J. Y. Park, M. Houston, A. Aiken, and W. J. Dally, "A tuning framework for software-managed memory hierarchies," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, Oct. 2008, pp. 280–291.

[37] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro, "Nitro: A framework for adaptive code variant tuning," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2014, pp. 501–512.

[38] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," in *Proc. 36th ACM SIGPLAN Conf. Program. Lang. Des. Implement. (PLDI)*, 2015, pp. 379–390.

[39] R. Nair, S.-L. Bernstein, E. Jessup, and B. Norris, "Generating customized sparse eigenvalue solutions with Lighthouse," in *Proc. 9th Int. Multi-Conf. Comput. Global Inf. Technol.*, Seville, Spain, Jun. 2014, pp. 1–4.

[40] A. Tiwari and J. K. Hollingsworth, "Online adaptive code generation and tuning," in *Proc. Int. Conf. Parallel Distrib. Process. Syst.*, May 2011, pp. 879–892.

[41] M. J. Voss and R. Eigemann, "High-level adaptive program optimization with ADAPT," in *Proc. ACM Principles Pract. Parallel Program.*, Jun. 2001, pp. 93–102.

[42] I.-H. Chung and J. K. Hollingsworth, "A case study using automatic performance tuning for large-scale scientific programs," in *Proc. 15th IEEE Int. Conf. High Perform. Distrib. Comput.*, Jun. 2006, pp. 45–56.

[43] R. D. Hornung and J. A. Keasler, "The RAJA portability layer: Overview and status," Lawrence Livermore Nat. Lab., Livermore, CA, USA, Tech. Rep. LLNL-TR-661403, Sep. 2014.

[44] H. C. Edwards, C. Trott, and D. Sunderland, "Kokkos: A manycore device performance portability library for C++ HPC applications," in *Proc. Workshop Program. Abstractions Data Locality*. Livermore, CA, USA: Sandia National Laboratories, Mar. 2014, pp. 1–37.

[45] D. Beckingsale, O. Pearce, I. Laguna, and T. Gamblin, "Apollo: Fast, dynamic tuning for data-dependent code," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, Orlando, FL, USA, May/Jun. 2017.

[46] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Des. Implement. (PLDI)*, 2013, pp. 519–530.

[47] K. J. Brown, "A heterogeneous parallel framework for domain-specific languages," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, Oct. 2011, pp. 89–100.

[48] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick, "Optimization of a lattice Boltzmann computation on state-of-the-art multicore platforms," *J. Parallel Distrib. Comput.*, vol. 69, no. 9, pp. 762–777, Sep. 2009.

[49] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM Rev.*, vol. 51, no. 1, pp. 129–159, 2009.

[50] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix–vector multiplication on emerging multicore platforms," *Parallel Comput.*, vol. 35, no. 3, pp. 178–194, Mar. 2009.

[51] J. Shin, M. W. Hall, J. Chame, C. Chen, and P. D. Hovland, "Autotuning and specialization: Speeding up matrix multiply for small matrices with compiler technology," in *Proc. 4th Int. Workshop Autom. Perform. Tuning*, Japan, 2009, pp. 353–370.

[52] K. Seymour, H. You, and J. Dongarra, "A comparison of search heuristics for empirical code optimization," in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep./Oct. 2008, pp. 421–429.

[53] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle, "Combined selection of tile sizes and unroll factors using iterative compilation," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, Washington, DC, USA, Oct. 2000, pp. 237–246.

[54] B. Norris, A. Hartono, and W. Gropp, *Annotations for Productivity and Performance Portability* (Computational Science). Boca Raton, FL, USA: CRC, 2007, pp. 443–461.

[55] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proc. ACM/IEEE Conf. Supercomput. (SC)*, Washington, DC, USA, Nov. 1998, pp. 1–27.

[56] A. Qasem, K. Kennedy, and J. Mellor-Crummey, "Automatic tuning of whole applications using direct search and a performance-based transformation system," *J. Supercomput.*, vol. 36, no. 2, pp. 183–196, May 2006.

[57] V. Tabatabaee, A. Tiwari, and J. K. Hollingsworth, "Parallel parameter tuning for applications with performance variability," in *Proc. ACM/IEEE Conf. Supercomput. (SC)*, Washington, DC, USA, 2005, p. 57.
[58] P. Balaprakash, S. M. Wild, and P. D. Hovland, "An experimental study of global and local search algorithms in empirical performance tuning," in *Proc. 10th Int. Conf. Revised Sel. Papers High Perform. Comput. Comput. Sci. (VECPAR)*. Springer, 2013, pp. 261–269.
[59] T. Nelson, "Generating efficient tensor contractions for GPUs," in *Proc. 44th Int. Conf. Parallel Process. (ICPP)*, Sep. 2015, pp. 969–978.
[60] P. Balaprakash, S. M. Wild, and P. D. Hovland, "Can search algorithms save large-scale automatic performance tuning?" in *Proc. Int. Conf. Comput. Sci. (ICCS)*, vol. 4, 2011, pp. 2136–2145.
[61] J. Bergstra, N. Pinto, and D. Cox, "Machine learning for predictive auto-tuning with boosted regression trees," in *Proc. Innov. Parallel Comput. (InPar)*, May 2012, pp. 1–9.
[62] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object-oriented numerical software libraries," in *Modern Software Tools for Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Cambridge, MA, USA: Birkhäuser, 1997, pp. 163–202.
[63] S. Balay (2015). *PETSc Web Page*. [Online]. Available: http://www.mcs.anl.gov/petsc
[64] S. Balay, "PETSc users manual," Argonne Nat. Lab., Lemont, IL, USA, Tech. Rep. ANL-95/11, 2015. [Online]. Available: http://www.mcs.anl.gov/petsc
[65] M. A. Heroux, "An overview of the Trilinos project," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 397–423, Sep. 2005.
[66] K. Sood, B. Norris, and E. Jessup, "Lighthouse: A taxonomy-based solver selection tool," in *Proc. 2nd Workshop Softw. Eng. Parallel Syst. (SEPS)*, Pittsburgh, PA, USA, Oct. 2015, pp. 66–70.
[67] P. Motter, K. Sood, E. Jessup, and B. Norris, "Lighthouse: An automated solver selection tool," in *Proc. 3rd Int. Workshop Softw. Eng. High Perform. Comput. Comput. Sci. Eng. (SEHPCCSE)*, Austin, TX, USA, Nov. 2015, pp. 16–24.
[68] E. Jessup, P. Motter, B. Norris, and K. Sood, "Performance-based numerical solver selection in the Lighthouse framework," *SIAM J. Sci. Comput.*, vol. 38, no. 5, pp. S750–S771, 2016.
[69] A. Tiwari, "Auto-tuning full applications: A case study," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 3, pp. 286–294, Aug. 2011.
[70] J. Shin, M. W. Hall, J. Chame, C. Chen, P. F. Fischer, and P. D. Hovland, "Speeding up Nek5000 with autotuning and specialization," in *Proc. 24th ACM Int. Conf. Supercomput. (ICS)*, 2010, pp. 253–262.
[71] T. Nelson, "Generating efficient tensor contractions for GPUs," in *Proc. 44th Int. Conf. Parallel Process.*, Sep. 2015, pp. 969–978.
[72] K. Ahmad, A. Venkat, and M. Hall, "Optimizing LOBPCG: Sparse matrix loop and data transformations in action," in *Proc. 29th Int. Workshop Lang. Compilers Parallel Comput.*, C. Ding, J. Criswell, and P. Wu, Eds. Springer-Verlag, 2016, pp. 218–231.
[73] H. M. Aktulga, A. Buluc, S. Williams, and C. Yang, "Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, May 2014, pp. 1213–1222.
[74] S. Song and J. K. Hollingsworth, "Computation–communication overlap and parameter auto-tuning for scalable parallel 3-D FFT," *J. Comput. Sci.*, vol. 14, pp. 38–50, May 2016.
[75] T. Gamblin, "The Spack package manager: Bringing order to HPC software chaos," in *Proc. Supercomput. (SC)*, Austin, TX, USA, Nov. 2015, pp. 1–12. [Online]. Available: http://tgamblin.github.io/pubs/spack-sc15.pdf
[76] A. Roy, P. Balaprakash, P. D. Hovland, and S. M. Wild, "Exploiting performance portability in search algorithms for autotuning," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, May 2016, pp. 1535–1544.
[77] S. Muralidharan, A. Roy, M. Hall, M. Garland, and P. Rai, "Architecture-adaptive code variant tuning," in *Proc. 21st Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2016, pp. 325–338.

ABOUT THE AUTHORS

Prasanna Balaprakash received the B.S. degree in computer science engineering from the Periyar University, Salem, India, the M.S. degree in computer science from the Otto-von-Guericke University, Magdeburg, Germany, and the Ph.D. degree in engineering sciences from CoDE-IRIDIA (AI Lab), Université libre de Bruxelles, Brussels, Belgium.

He was a Marie Curie Fellow and later an FNRS Aspirant at AI Lab. Currently, he is a Computer Scientist with a joint appointment in the Mathematics and Computer Science Division and the Leadership Computing Facility, Argonne National Laboratory. His research interests span the areas of artificial intelligence, machine learning, optimization, and high-performance computing. Currently, his research focus is on the automated design and development of scalable algorithms for solving large-scale problems that arise in scientific data analysis and in automating application performance modeling and tuning.

Jack Dongarra (Fellow, IEEE) holds an appointment at the University of Tennessee, Oak Ridge National Laboratory, and the University of Manchester. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced-computer architectures, programming methodology, and tools for parallel computers.

Dr. Dongarra was awarded the IEEE Sid Fernbach Award in 2004. In 2008, he was the recipient of the first IEEE Medal of Excellence in Scalable Computing; in 2010, he was the first recipient of the SIAM Special Interest Group on Supercomputing's award for Career Achievement. In 2011, he was the recipient of the IEEE Charles Babbage Award, and in 2013 he received the ACM/IEEE Ken Kennedy Award. He is a Fellow of the AAAS, ACM, IEEE, and SIAM and a foreign member of the Russian Academy of Science and a member of the U.S. National Academy of Engineering.

Todd Gamblin (Member, IEEE) received the B.A. degrees in computer science and Japanese from Williams College, in 2002, and the M.S. and Ph.D. degrees in computer science from the University of North Carolina, Chapel Hill, in 2005 and 2009, respectively.

He is a Computer Scientist in the Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, where he has been since 2008. His research focuses on scalable tools for measuring, analyzing, and visualizing parallel performance data. He is also the creator of Spack, a popular HPC package management tool.

Mary Hall (Senior Member, IEEE) received the B.A., M.S., and Ph.D. degrees in computer science, all from Rice University.

Currently, she is a Professor in the School of Computing at the University of Utah. Her research focuses on compiler technology for exploiting performance-enhancing features of a variety of computer architectures: automatic parallelization for multicores and GPUs, superword-level parallelism, processing-in-memory architectures, and FPGAs.

Jeffrey K. Hollingsworth (Senior Member, IEEE) received the B.S. degree in electrical engineering from the University of California, Berkeley, CA, USA, and the M.S. and Ph.D. degrees in computer sciences from the University of Wisconsin.

He is currently serving as Interm Chief Information Officer of the University of Maryland. He is a Professor in the Computer Science Department, University of Maryland, College Park. In his research, he seeks to develop a unified framework to understand the performance of large systems and focuses on performance measurement and autotuning. He was Editor-in-Chief of the journal *Parallel Computing*, was general chair of the SC12 Conference, and is Chair of ACM SIGHPC.

Boyana Norris received the B.S. degree from Wake Forest University and the Ph.D. degree from the University of Illinois at Urbana-Champaign, both in computer science.

She is an Associate Professor in the Computer and Information Science Department at the University of Oregon. Her research in high-performance computing focuses on methodologies and tools for performance reasoning and automated optimization of scientific applications, while ensuring continued or better usability of HPC tools and libraries and improving developer productivity.

Richard Vuduc (Member, IEEE) received the B.S. degree in computer science from Cornell University, Ithaca, NY, USA, and the Ph.D. degree in computer science from the University of California, Berkeley, CA, USA.

He is an Associate Professor in the School of Computational Science and Engineering, Georgia Institute of Technology, Atlanta, GA, USA. His research lab, The HPC Garage (@hpcgarage), is interested in high-performance computing, with an emphasis on algorithms, performance analysis, and performance engineering. From 2014 to 2016, he served as an Associate Editor for the IEEE Transactions on Parallel and Distributed Systems.