Bootstrapping In-situ Workflow Auto-Tuning via Combining Performance Models of Component Applications

Tong Shu
Southern Illinois University
Carbondale, IL, USA
tong.shu@siu.edu

Yanfei Guo
Argonne National Laboratory
Lemont, IL, USA
yguo@anl.gov

Justin Wozniak
Argonne National Laboratory
Lemont, IL, USA
woz@anl.gov

Xiaoning Ding
New Jersey Institute of Technology
Newark, NJ, USA
xiaoning.ding@njit.edu

Ian Foster
Argonne Natl. Lab and Univ. Chicago
Lemont and Chicago, IL, USA
foster@anl.gov

Tahsin Kurc
Stony Brook University
Stony Brook, NY, USA
tahsin.kurc@stonybrook.edu

ABSTRACT

In an in-situ workflow, multiple components such as simulation and analysis applications are coupled with streaming data transfers. The multiplicity of possible configurations necessitates an auto-tuner for workflow optimization. Existing auto-tuning approaches are computationally expensive because many configurations must be sampled by running the whole workflow repeatedly in order to train the auto-tuner surrogate model or otherwise explore the configuration space. To reduce these costs, we instead combine the performance models of component applications by exploiting the analytical workflow structure, selectively generating test configurations to measure and guide the training of a machine learning workflow surrogate. Because the training can focus on well-performing configurations, the resulting surrogate model can achieve high prediction accuracy for good configurations despite training with fewer total configurations. Experiments with real applications demonstrate that our approach can identify significantly better configurations than other approaches for a fixed computer time budget. For example, with a budget of 50 training samples, it reduces execution time and computer time for a realistic workflow by 18.5% and 47.5% relative to random sampling, and by 11.2% and 39.8% relative to a state-of-the-art algorithm, GEIST.

1 INTRODUCTION

Scientific workflows couple multiple component applications, each of which can be run independently: e.g., a simulation application plus tools to analyze, visualize, and learn [54] from the simulation results. Conventionally, component applications are often executed only post-hoc; the simulation application saves the results to persistent storage from which downstream applications read the results. This approach is increasingly infeasible due to the high I/O costs incurred when saving results and corresponding delays in downstream processing. In-situ workflow solutions address these issues by allowing simulation applications to pass results to downstream applications on the fly, via network or shared memory, without writing to storage [3, 17, 19].

Despite their advantages, in-situ workflows raise performance tuning challenges. The performance of a workflow is largely determined by configuration parameters, such as the number of nodes, processes, and threads used to run each component application, among other settings. The component applications in an in-situ workflow run concurrently and often exchange data while executing. Thus, they contend for hardware resources, such as processor cores and network bandwidth, making it impossible to tune the application configurations separately. When tuning the applications together, the multiplicative increase in the number of potential configurations form a huge configuration space that human experts can rarely handle.

Thus, auto-tuners, particularly model-based auto-tuners, become a promising method. However, it is still a challenging problem to build such an auto-tuner for an in-situ workflow. The core of an auto-tuner is a surrogate model of the coupled workflow performance that can be used to predict the performance that would be observed for a specific workflow configuration. To build an accurate surrogate model of this sort, the complete workflow must be run for a large range of selected configurations (a.k.a. samples) to collect the performance data required for model training. Given the huge configuration space and high cost of each workflow run, users usually cannot afford the cost of building such an auto-tuner.

Performance modeling conventionally involves the use of either black-box or white-box models. However, neither approach is effective for the above problem. Black-box modeling based on machine learning (ML) can build an accurate model if sufficient sample configurations are tested. However, the large size of the potential configuration space means that this approach is unlikely to be affordable for production applications running representative workloads [30, 57].

White-box modeling has been used to tackle large configuration spaces for conventional workflows [40–44]. It focuses on analyzing the interactions (particularly data dependencies) between component applications, with an analytical coupling model (ACM) used to combine the performance models of individual applications into a workflow model. This divide-and-conquer method reduces complexity, because each application has a relatively small configuration space and its *component model* can be obtained with existing methods (mainly black-box modeling methods) [8, 11, 34, 49]. However,

SC '21, November 14–19, 2021, St. Louis, MO, USA

ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3476197

while this approach works well for workflows in which component interactions occur before or after component executions (i.e., are separable), allowing an analytic model to reflect such interactions accurately, it does not work well for in-situ workflows. Due to the frequent run-time interaction (particularly synchronization) and more complex resource contention between component applications, component models built based on observations of applications running separately cannot accurately predict the performance that components display when they are run together in an in-situ workflow. Furthermore, ACMs cannot easily incorporate representations of such interactions. Thus, a workflow model built with an ACM and component models only often lacks the accuracy required for auto-tuning of in-situ workflows [21]. To build accurate models, performance data from actual workflow runs must be integrated.

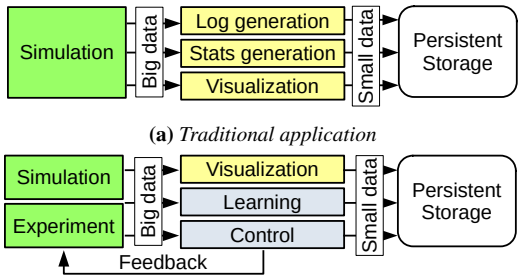
To address this challenge in auto-tuning in-situ workflows, we propose to combine the ACM-based white-box modeling and ML-based black-box approaches so as to leverage both the cost-effectiveness of the white-box approach and the greater accuracy of the ML-based black-box approach. We will show that this combined approach can allow accurate auto-tuning models to be built at low cost.

To implement this idea, the first and foremost issue is a method that can make the two approaches work synergistically. After comparing a few possible methods (e.g., boosting an ACM with ML and auto-selection of ACMs) in §8.2, we developed a bootstrapping method to combine these approaches. Specifically, we first use the white-box approach to build a low-fidelity model of the workflow; second, we use the low-fidelity model to test configurations selectively and guide the training of a high-fidelity ML-based model for use in the auto-tuner.

The bootstrapping method is based on the observation that the auto-tuner does not require its performance model to have high prediction accuracy for all possible configurations. Notably, for configurations that lead to poor performance, the accuracy requirements are relatively low, and as long as the model does not predict that poor configurations will lead to high performance, the auto-tuner will not mistakenly recommend these configurations. For good configurations leading to high performance, the accuracy requirements are relatively high so that optimal configurations can be identified. By exploiting this observation we reduce the number of poor configurations selected for experimental measurement and model training. This reduces the required workflow runs, particularly long-running, high-cost workflow runs. Our bootstrapping method achieves this using the low-fidelity model formed by the white-box approach.

Two technical issues must be addressed in the bootstrapping method. The first issue is how to determine the number of poor configurations and good configurations selected as training samples, so as to minimize the total number of samples and to make the model have a reasonable accuracy on poor configurations at the same time. To address this issue, we continually monitor the prediction accuracy of poor and good configurations, and dynamically adjust the number of configurations of each type.

The second issue is how to determine when bootstrapping should be stopped. The ML-based model is gradually refined in the training process. After its accuracy is higher than the ACM-based model, it becomes a better choice for selecting training samples. Thus, for the black-box techniques that can select training samples actively using their own models, such as active learning (AL) [29], reinforcement learning (RL) [39], and Bayesian optimization (BO) [32], the bootstrapping should be stopped. To stop the bootstrapping at an appropriate moment, we continually monitor the accuracy of both the ACM-based model and the ML-based model.



(a) *Traditional application*

(b) *Integration-learning application*

Figure 1: Typical in-situ workflow application patterns.

This paper makes the following contributions: 1) As far as we know, our approach is the first work to auto-tune in-situ workflows under a tight budget on measurements. 2) To build accurate in-situ workflow auto-tuning models, we synergistically combine ML-based black-box modeling and ACM-based white-box modeling using the bootstrapping method. 3) We implement the idea in a new in-situ workflow auto-tuning algorithm, CEAL (**C**omponent-based **E**nsemble **A**ctive **L**earning). 4) We use three in-situ HPC workflows to experimentally verify the superiority of CEAL over other auto-tuning algorithms. With just 25 training samples, CEAL can reduce the computer time of a realistic workflow by 12–48% in comparison with other existing auto-tuning algorithms.

2 MOTIVATION AND BACKGROUND

In this section, we first illustrate the importance of in-situ processing in scientific applications. Then, we describe the basic mechanisms used in auto-tuners and identify the challenges in designing such auto-tuners for in-situ workflows.

2.1 Importance of In-situ Workflow Applications

Recent workshop reports [24, 37] have highlighted the importance of in-situ workflows to scientific applications as simulation-generated and experimental/observational data sizes increase. These reports call for advanced systems that can manage streaming data and efficiently propagate data through multiple stages of processing. The benefits of this approach include reducing the load on persistent storage technologies, and enabling AI-centric scientific studies.

An example of traditional uses of in-situ technologies is shown in Figure 1a. For example, a computational fluid dynamics simulation running at half system scale on a near exascale machine could output data at a rate that exceeds the capability of the parallel file system, blocking simulation progress and impacting other users. The raw simulation data may not be needed, but rather human-readable logs, statistics, and visualizations [20]. In-situ technologies are even more critical in the emerging AI for Science paradigm [47] as shown in Figure 1b. Consider the case of a material for analysis via neutron scattering, in which a sample is scanned at the beamline while a

digital twin of the sample is represented in a supercomputer. The operations in visualization and machine learning could be applied on the live data before it is stored persistently, enabling feedback to the experiment in real-time [15]. While these uses are transformative in obtaining high system utilization and in the application of AI, they are also sensitive to configuration, as the wrong settings could cause catastrophic slowdowns. Workflow-aware tools must be developed to optimize performance by exploiting the known workflow structure of these composite applications.

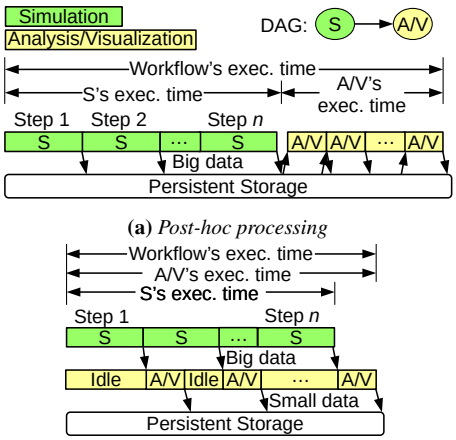
2.2 Empirical Model-Based Auto-tuners

These applications are often run repeatedly on similar computers and problems, and the similarities can make it beneficial to tune configuration parameters to improve performance in a reusable way. Given the growing complexity of applications and HPC infrastructures, empirical model-based auto-tuners must be leveraged, which run experiments in the configuration space and train a performance model and select a good set of configuration parameters. Although many auto-tuning approaches exist [5], this paper focuses on empirical model-based auto-tuning because of its effectiveness and prevalence. For brevity, hereon we will refer to "empirical model-based auto-tuning/auto-tuners" as "auto-tuning/auto-tuners."

An auto-tuner typically has three components: collector, modeler, and searcher [6, 9, 22, 30, 31, 45, 50, 51]. The *collector* runs the target application with different configurations selected by the modeler, and collects performance measurements. The *modeler* selects configurations from the parameter configuration space of the target application, drives the collector to obtain the corresponding performance measurements, and uses the measurements as training data to construct a surrogate performance model: a high-dimensional function of configuration parameters, usually obtained by ML. The *searcher* uses the model to search for a good configuration, i.e., one that produces good performance. During the search, the searcher uses the model to predict the performance for the configurations being examined, and selects the configuration with the best predicted performance.

The core of an auto-tuner design is a modeling algorithm. Factors to consider when designing this algorithm include model type and the methods used to select configurations used as training samples, to train the model, and to search configuration space. Thus, for example, neural networks (NN) [33], which require many training samples, are not used in our solution, because the cost of collecting so many samples is prohibitive for HPC workflows.

A well-designed modeling algorithm can substantially both reduce the cost and improve the performance of an auto-tuner. For resource-intensive applications, such as HPC programs, *auto-tuner cost* is dominated by the time required to run the target application repeatedly to collect training data. Model training and configuration space search, in contrast, are inexpensive: Some traditional ML models, such as boosted trees (BT) and random forests (RF), may take only a few minutes. *Auto-tuner performance*, the capacity to find good configurations and improve application performance, is determined primarily by how well the model predicts application performance for given configurations, particularly good configurations, because a bad configuration is rarely choosen as the top configuration as long as its prediction error is not very high to make it seem like a good one. (It is also affected by how the parameter space is searched, but as search mechanisms are mature, most efforts focus on improving the model.) Accurately modeling a complex target depends on the power of the ML model and the number of training samples. If there are only a limited number of training samples, traditional ML methods, such as BT and RF, achieve better prediction accuracy than the more powerful NN, because they typically have many fewer weights to be trained.



(a) *Post-hoc processing*

(b) *In-situ workflow*

Figure 2: Post-hoc processing vs. in-situ workflows

2.3 Auto-Tuning for In-situ Workflows

In our context, a workflow is a directed acyclic graph (DAG), with application components as nodes and data dependency as edges. Components are typically coupled by using a high-level programming language or library, thus exposing a *structure* that can be exploited for performance modeling and optimization.

In a file-based post-hoc processing workflow, component applications are executed in an order determined by their data dependencies, and thus have no interactions during their respective execution. For example, in Fig. 2a, when the simulation finishes, it saves data to persistent storage; only then can the analysis/visualization, which processes the data, be started. Modeling for performance optimization can then proceed in two steps. First, we model the performance of each component independently. Second, we build an analytical workflow model based on the DAG and the component models. Such white-box modeling is accurate enough to optimize a file-based post-hoc processing workflow.

In contrast, as shown in Fig. 2b, component applications in an in-situ workflow run concurrently, frequently exchanging data via network or shared memory during their respective execution. Workflow performance is determined by the complex interplay of the applications, which may involve factors such as load imbalances, contending network bandwidth, synchronizations, and locks [21]. High performance requires that the component applications execute in a balanced and coordinated way.

For in-situ workflow optimization, performance modeling cannot simply be done through an analytical model combining separate performance models of component applications due to the complicated

run-time interactions among component applications. Instead, a performance model for the whole workflow must be built by leveraging the accuracy of ML-based black-box modeling, and all parameters from all components should be optimized together. For empirical model-based auto-tuners, the fact that an in-situ workflow includes multiple coupled components raises considerable challenges. Because all parameters from all components must be considered together, the potential parameter combinations increases multiplicatively. For example, in the two-component workflows of §7.1, the configuration space sizes are more than $10^5\times$ larger than those of their component applications. This dramatically raises the number of configurations to be measured as training samples for building a usable surrogate model. However, for in-situ workflows, it is not realistic to measure many parameter combinations, given the high resource consumption of running HPC applications.

Without fundamentally renovating auto-tuning algorithms, particularly the techniques used to build the surrogate model, auto-tuners for in-situ workflows face a difficult dilemma—whether to suffer a prohibitive cost in creating the accurate surrogate model needed for optimal performance, or to tolerate the poor performance associated with an inaccurate surrogate model generated at an affordable cost.

The CEAL algorithm proposed in this paper fundamentally improves the techniques to build surrogate models, such that auto-tuner cost can be reduced substantially while retaining high performance. The large resources needed to run a complete workflow repeatedly when building an auto-tuner are usually limited in practical settings by a resource budget. Thus, in this scenario, the advantage of the algorithm is reflected by improving the performance of the auto-tuner within a cost budget.

3 BOOTSTRAPPING METHOD OVERVIEW

This section gives an overview of the bootstrapping method, focusing on the main idea and the major steps in this method. We will introduce its major techniques in detail in §4 and §5. We implement the bootstrapping idea and the techniques with the CEAL algorithm, which is introduced in §6.

As auto-tuning cost is dominated by the collection of training samples, we must select training samples carefully and use them effectively, instead of selecting training samples indiscriminately and extensively (e.g., by random sampling). The general idea of intelligent sampling has been explored in different ways in ML and in auto-tuner designs [6, 29, 30, 50]. Our work is distinguished by how we exploit the workflow structure (§2.3) to develop techniques that are particularly effective for auto-tuning in-situ workflows.

Our bootstrapping method leverages the following two characteristics of in-situ workflows: 1) An in-situ workflow consists of multiple components, which can run independently and may be reused across workflows. 2) The synchronization among components means that if any component performs poorly, the workflow is unlikely to achieve high performance.

Based on the principle that more training samples collected in a region of good configurations lead to higher accuracy of the trained model in the region, we should avoid collecting samples in which a workflow performs poorly, as such samples are unlikely to help with finding well-performing configurations. But how are we to avoid collecting poor-performing samples in the absence of a performance

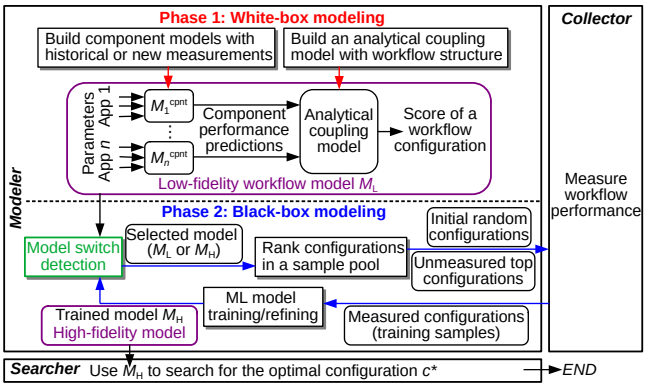


Figure 3: In-situ workflow auto-tuning with bootstrapping.

model, which is why we want those samples in the first place? We employ two ideas. 1) Leveraging the first characteristic, we build performance models for individual component applications. Because the parameter spaces of component applications are much smaller than that of the in-situ workflow, these models can be built at low cost, i.e., with only a few component application runs. Also, the models of component applications can be reused or built on historic measurements during their previous standalone uses or reuse in other workflows, further lowering costs. 2) Leveraging the second characteristic, plus the component models that we have just developed, we build a simple low-fidelity model that we use to guide our search for well-performing configurations for the whole in-situ workflow, and focus sample collection on these configurations.

Note that the low-fidelity model itself has intrinsic limitations. First, the component models are built based on the solo runs of each application. Thus, they cannot accurately predict the performance of the applications when they run together in a workflow, where they interact frequently and contend hardware resource with each other. Second, the component models are combined in a simple way and cannot reflect the interactions and resource contentions between component applications. Therefore, it is not realistic to refine this model with more component training samples to achieve high accuracy and then use the refined model in auto-tuning.

We refer to the surrogate model used in the auto-tuner as a high-fidelity model. For the training of the high-fidelity model, any black-box modeling techniques can be used and "bootstrapped" with the low-fidelity model. The techniques that can select training samples actively using their own models are preferred and can be integrated with the bootstrapping method more seamlessly, as we will show later. For example, active learning iteratively uses the model that is being refined to identify configurations that may lead to good performance, and focuses sample collection on those configurations [6, 29]. Other techniques include RL and BO. In the paper, we select to use the active learning technique as an example to describe and evaluate our bootstrapping method.

Fig. 3 gives an overview of auto-tuning an in-situ workflow with the bootstrapping method, including collecting samples and building the models to searching optimal configurations. As highlighted in the figure, with the bootstrapping method, building the surrogate model consists of two main phases:

• **Phase 1: Low-fidelity Model Generation via Component Model Combination.** As shown in the upper part of the modeler in Fig. 3, we generate performance models for a workflow's component applications and build an ACM which is used to combine component models to form a simple yet integral workflow model. This simplicity means that the integral model can be obtained at low cost but yields only approximate predictions. We use this *low-fidelity* workflow model (M_0 in Fig. 3) in the second phase, to evaluate configurations.
• **Phase 2: High-fidelity Model Generation via Dynamic Ensemble Active Learning.** As shown in the lower part of the modeler in Fig. 3, we use a series of samples selected based on low-fidelity model scoring to establish and improve a second, high-fidelity model of the workflow (M_1 in Fig. 3). The *high-fidelity model* is the surrogate model that the searcher will use to predict workflow performance so as to find an ideal configuration. To refine it, we use a model selected from M_0 and M_1 to rank all configurations in a sample pool and measure the performance of top ranked configurations as incremental training data for M_1. The high-fidelity model is primitive when first established, but keeps evolving as more samples are collected and used in training, and may become a better choice for evaluating configurations than the low-fidelity model. Thus, we use a *model switch detection module* to monitor the two models, and switch to using the high-fidelity model to evaluate configurations when it becomes a better choice. We stop evolving the high-fidelity model when the cost budget is reached (i.e., we have tested a preset number of configurations).

In essence, our approach is not to reduce the configuration space to be explored in model building. It still selects configurations from all parts of configuration space. (We show in Section 6 that the configurations selected from all the areas of the configuration space are used in training.) But, it aims to control the number of samples selected from different areas based on the corresponding performance of the workflow. In the areas where the workflow performs poorly, a relatively small number of samples are selected to keep the total number of samples low; in the areas where the workflow performs well, a relatively large number of samples are selected, in order to achieve high accuracy and thus increase the capability of the auto-tuner to find optimal configurations in these areas. Building and using a low-fidelity model ensures that such control can be imposed as early as possible and these benefits are maximized. This approach requires that well-performing configurations are not evenly distributed in configuration space. (Actually, most applications show this trait.) It is most effective when well-performing configurations are concentrated in a small region.

4 LOW-FIDELITY MODEL GENERATION

We use the low-fidelity model to evaluate configurations by predicting how well the workflow may perform. The main challenge is to build a useful model with a minimal cost.

We build the low-fidelity model by first constructing and then combining predictive models of the component applications (see the red dashed rectangle in Fig. 3). Each individual *component model*, M_j ($j = 1, 2, \cdots$), outputs a performance prediction for its component for a given configuration. Its predictions should be aligned with the optimization goal of the auto-tuner; for example, execution times if the goal is shortening workflow execution time. Component models can be built by using conventional methods, e.g., by randomly selecting configurations to collect samples and then training a boosted tree ML model. Since component applications are independent, they may be used separately or in other workflows. Thus, costs can be reduced by incorporating measurements collected in earlier runs into training data, or by reusing component models developed for other workflows. Due to space limitations, we do not elaborate here on how to build or reuse these models, but focus on how to combine component models to form the integral low-fidelity model.

There are two issues in forming the low-fidelity model. The first is what the model should produce. As we will use this model only to choose among configurations, we do not need it to predict workflow performance directly. Instead, we make it output for each configuration just a score indicating how well the workflow performs relative to other configurations.

The second issue is how to combine per-component model results to build a low-fidelity model with minimal cost. A black-box modeling approach, which we implemented in an algorithm called ALpH for later quantitative comparison, is to train a component-combining model M_0' from both component model predictions and actual workflow runs. That is, for each candidate configuration c, we use the component models $\{M_j'\}$ ($j = \{1, 2, \cdots\}$) to predict the performance values $\{v_j\}$ of the components for c; run the workflow with c, measuring its performance v; and use $\{c, \{v_j\}, v\}$ as the training data for M_0'. ALpH uses AL [29] to select the configurations for which it generates such workflow training samples. A deficiency of this approach is that it does not exploit any knowledge of the workflow structure, thus incurring high training data collection cost.

A white-box modeling method, which we use in CEAL, is to use a simple function (e.g., max, min, or sum), chosen according to the performance metric being optimized by the auto-tuner, to combine the component model predictions. We select this function as follows. If the performance metric is determined largely by the bottleneck components, such as execution time and throughput, we use *max* (for execution time) or *min* (for throughput). If the performance metric is largely an aggregation of the shares from all components, such as computing resource and energy consumption, we use *sum*. Notice that the component model combination approach in CEAL, unlike that in ALpH, does not need to run the workflow. We examine the relative accuracy and costs of CEAL and ALpH in §7.5.

We postpone detailed evaluation to §7.5, but report here on a motivating study in which we use two optimization objectives with different metrics—shortening execution time and minimizing computer time—to illustrate and characterize the function approach. We define execution time as wall-clock time and computer time as the number of core-hours consumed by workflow execution. We define the functions used to determine a configuration's scores as

$$Score^{\mathrm{e}}(c) = \max_j t^{\mathrm{e}}(c_j), \tag{1}$$

$$Score^{\mathrm{c}}(c) = \sum_j t^{\mathrm{c}}(c_j), \tag{2}$$

where for a configuration c, $Score^{\mathrm{e}}(c)$ and $Score^{\mathrm{c}}(c)$ are the ***e***xecution and ***c***omputer times (the lower, the better) of configuration c; c_j is the parameter values related to component j extracted from c; and $t^{\mathrm{e}}(c_j)$ and $t^{\mathrm{c}}(c_j)$ are the model-predicted execution and computer times of the j^{th} component.

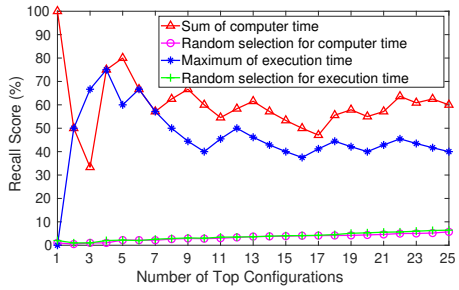


Figure 4: Recall scores based on combination functions.

To illustrate the effectiveness of this approach, Fig. 4 shows the recall scores (RS) of the low-fidelity models (Eqns. 1 and 2) when used to score 500 randomly selected configurations for workflow *LV*. (For details on experimental settings, see §7.1.) Recall scores reflect the possibility that the highest-scoring configurations lead to high workflow performance (§7.2.2). To calculate recall scores, we rank configurations based both on their model-predicted scores and on the performance observed when the workflow is executed with them. The recall score of the top n configurations is then the ratio between 1) the number of common configurations found in the top n configurations on these ranked lists and 2) n; and it varies with n increasing. If the goal is to minimize computer time, we rank configurations with two methods: 1) sorting their scores based on the model in Eqn. 2 and 2) placing them in a random order, and refer the two methods to as "sum of computer time" and "random selection for computer time," respectively. It is similar to the goal of minimizing execution time. We see that the models achieve recall scores above 30% for top 2 to 25 configurations, much higher than those of random selection. This verifies that even using simple functions in the low-fidelity model can effectively locate good configurations.

The solution described in this section mainly targets loosely-coupled in-situ workflows [21, 48], in which components are coupled via high-level libraries (e.g., ADIOS [27], Flexpath [12], DataSpaces [14], FlexIO [59], GLEAN [52], DIMES [58], Decaf [16], or Zipper [21]). The relative ease of development and deployment of loosely-coupled in-situ workflows relative to tightly-coupled in-situ workflows makes the former much more prevalent. However, our solution can easily be adapted to optimize tightly-coupled in-situ workflows.

5 HIGH-FIDELITY MODEL GENERATION

To build the high-fidelity model, CEAL first creates a sample pool C_{pool} by selecting configurations randomly from the workflow's configuration space C. All configurations used subsequently to train the high-fidelity model are selected from C_{pool}. As we evolve the model, we repeatedly evaluate and rank all configurations remaining in C_{pool}; thus, we want $|C_{\text{pool}}| \ll |C|$ to keep evaluation costs manageable. However, we also want C_{pool} to be reasonably representative of C and, in particular, to contain enough well-performing configurations to train a good high-fidelity model. Say that we want the best configuration in C_{pool} to be in the top $1/n$ of all configurations, with probability P. With a pool size p, the chance of selecting p configurations each not in the top $1/n$ is less than $(1-1/n)^p$. Thus, we want $p \approx -n \cdot \ln(1-P)$, because $P > 1-(1-1/n)^p = 1-[(1-1/n)^n]^{p/n} > 1-(1/e)^{p/n}$. For example, if $1/n = 1/500 = 0.2\%$ and $P = 98.2\%$, then $p \approx 2000$.

To establish the high-fidelity model, CEAL selects $m_0/2$ configurations at random from C_{pool} (as at most m_0 random samples are chosen totally); selects the m_B best of the configurations remaining in C_{pool}, based on scores from the low-fidelity model; runs the workflow with the $m_0/2 + m_B$ selected configurations; and uses the results as samples to train an initial high-fidelity model. We discuss the factors influencing the choice of m_B and m_0 in §6.

To evolve the model, CEAL evaluates the configurations remaining in C_{pool} and selects the m_B highest-scoring. Then, it runs the workflow with those configurations and uses those results to train the high-fidelity model further. It repeats these operations until the cost budget is reached.

The high-fidelity model is initially primitive and thus the low-fidelity model is a superior choice for evaluating configurations. As more training data are acquired, the high-fidelity model eventually outperforms the low-fidelity model. Thus, we monitor the capability of the high-fidelity model in evaluating configurations, and substitute it for the low-fidelity model when it becomes a better choice. Specifically, each time that we perform more runs, we use the new data to compute the top-1, top-2, and top-3 recalls (see §7.2.2) of the low- and high-fidelity models: that is, the extent to which their best 1, 2, and 3 configurations, respectively, match the best 1, 2, and 3 as determined by experiment. When these scores for the high-fidelity model (summed to increase stability) exceed the low-fidelity sum, we switch to the high-fidelity model.

As we show quantitatively when we present experimental results in §7.4.2, the power of the CEAL approach derives from the fact that it selects mostly top configurations when collecting data to train the high-fidelity model. The rationale here is this: our ultimate goal is a surrogate model that the searcher can use to find the best configurations, and for that purpose it should be highly accurate for good configurations, but can be less accurate for bad configurations. Thus, we prefer to use our limited sample budget for samples collected for high-performing configurations. (Focusing sample collection on high-performing configurations also has the expedient side effect that these samples, by definition, take less time.)

Thus, as described above, we use the low-fidelity model to bootstrap the sample selection process, and transition to the high-fidelity trained model as the number of collected samples grows. But what if our low-fidelity model is biased in such a way that it never gives good scores to high-performing configurations? Then the high-fidelity model may never improve. This concern motivates us to select, in the initial phase, $m_0/2$ random configurations and the m_B configurations selected with the low-fidelity model, and dynamically adjust them in the subsequent phase. We discuss the sensitivity of CEAL to these hyper-parameters in §7.6.

6 CEAL ALGORITHM

The CEAL algorithm, Alg. 1, takes as input a data collection budget (m), expressed in terms of the number of workflow runs for simplicity. (If a budget on real resource consumption is preferred, the algorithm can be adapted to monitor the resource consumption of the workflow and its component applications.)

Algorithm 1 **C**omponent-based **E**nsemble **A**ctive **L**earning

Inputs: Workflow runs budget $\boldsymbol{m}$; budget used to run component applications $\boldsymbol{m_R}$; for each component application $1 \le j \le J$, configuration space $\boldsymbol{C_j}$ and historical measured configuration-performance samples $\boldsymbol{D_j^{hist}}$; sample pool $\boldsymbol{C_{pool}}$; upper bound on the number of random samples $\boldsymbol{m_0}$; number of iterations $\boldsymbol{I}$.

Output: High-fidelity workflow model $\boldsymbol{M}_{\mathrm{H}}$.

1: **for** $j \in \{1, 2, \cdots, J\}$ **do**
2: Randomly select m_{R} configurations from C_j as C_j^{meas};
3: Run j^{th} component with configurations in C_j^{meas}, giving D_j^{meas};
4: $D_j^{\mathrm{meas}} \leftarrow D_j^{\mathrm{meas}} \cup D_j^{\mathrm{hist}}$;
5: Train component model M_j^{cpnt} with D_j^{meas} for the j^{th} application;
6: Generate low-fidelity model M_{L} from component models and combination function (see §4);
7: Move $m_0' = m_0/2$ randomly selected configurations from C_{pool} to form C_{meas};
8: $m_B \leftarrow (m - m_0 - m_{\mathrm{R}})/I$;
9: Score all configurations in C_{pool} with M_{L};
10: Move top m_B configurations from C_{pool} into C_{meas};
11: $M \leftarrow M_{\mathrm{L}}$; // Set the model used for evaluating configurations
12: $M_{\mathrm{H}} \leftarrow$ ML model (e.g., boosted tree [10]); // Initialize high-fidelity model
13: **for** $i \in \{1, \cdots, I\}$ **do**
14: Run workflow for all configurations in C_{meas}, giving D_{meas};
15: $C_{\mathrm{meas}} \leftarrow \emptyset$;
16: **if** $M = M_{\mathrm{L}}$ **then** // Begin model switch detection
17: // Score models: recalls for top 1, 2, 3 configurations
18: $S_{\mathrm{H}} = \sum_{i=1,2,3} S_r(i, C_{\mathrm{meas}}, M_H, D_{\mathrm{meas}})$; // See §7.2.2
19: $S_{\mathrm{L}} = \sum_{i=1,2,3} S_r(i, C_{\mathrm{meas}}, M_L, D_{\mathrm{meas}})$; // See §7.2.2
20: **if** $|\mathrm{top}(3, M_H) \cap \mathrm{top}((m_0' + i \times m_B)/2, D_{\mathrm{meas}})| < 3$ **then**
21: Move $(m_0 - m_0')/2$ random configurations from C_{pool} to C_{meas};
22: $m_0' = m_0' + (m_0 - m_0')/2$;
23: **if** $S_{\mathrm{H}} \ge S_{\mathrm{L}}$ **then**
24: $M \leftarrow M_{\mathrm{H}}$; $m_B \leftarrow m_B + (m_0 - m_0')/(I - i)$;
// End model switch detection
25: Use D_{meas} to train/refine M_{H}, and update M if it switched to M_{H};
26: Use M to evaluate the configurations in C_{pool};
27: Move the top m_B configurations from C_{pool} to C_{meas};
28: **return** M_{H}.

Lines 1–6, the phase of low-fidelity model generation via component model combination, run each component application m_{R} times to test randomly selected configurations and build component models. The cost is equivalent to running the complete workflow m_{R} times, incurring a charge of m_{R} from budget m (Line 8). If a component application has been tested earlier, some configuration-performance data D_j^{hist} can be reused to further improve component model quality (Line 4), in which case m_{R} should be close to 0. Otherwise, m_{R} is generally set to be from $m \cdot 25\%$ to $m \cdot 75\%$.

The second phase, high-fidelity model generation via dynamic ensemble active learning, is Lines 7–28. First, $m_0' = m_0/2$ random training samples are selected (Line 7) to characterize the overall performance distribution of the workflow over all configurations. m_0' will be increased up to m_0, if needed. In general, m_0 is set to be from $m \cdot 5\%$ to $m \cdot 30\%$, depending on workflow structure and optimization objective. Sensitivity studies reported in §7.6 show that CEAL is insensitive to m_{R} and m_0 values, over a large range. We recommend that $m_0 \approx 35\% \cdot m$ when $|D_j^{\mathrm{hist}}| \gg m$ (j=1,...,J), and $m_0 \approx 15\% \cdot m$ if no historical measurements are available (i.e., $|D_j^{\mathrm{hist}}|$=0). Then, top configurations are selected (Lines 10 and 27) based on the evaluation with model M. Lines 16–24 handle the switching from low-fidelity to high-fidelity model. The high-fidelity model is retrained repeatedly as more training data are acquired (Lines 13–27), and returned as the output of the algorithm.

7 EXPERIMENTAL EVALUATION

We describe our benchmarks (§7.1), evaluation metrics (§7.2), and comparison targets (§7.3). Then, we evaluate the performance of CEAL and other algorithms in a general auto-tuning scenario without historical measurements, and investigate reasons for CEAL's superiority (§7.4). We also consider optimization with historical component measurements, and compare CEAL with an algorithm that incorporates component performance by training a ML model (§7.5). Finally, we study CEAL's sensitivity to hyper-parameter values (§7.6).

7.1 Experimental Setup

We implemented our auto-tuner system following the EMEWS Framework [36] and using the high-performance dataflow computing language Swift/T [2]. To enable job-level fault-tolerance of the collector in our auto-tuner, we enhanced Swift/T by developing a noval MPI function MPI_Comm_launch [55]. Then, we conducted experiments on a 600-node cluster with Intel Omni-Path Fabric Interconnect. Each node has two 18-core 2.10GHz Intel Broadwell Xeon E5-2695 v4 processors with hyperthreading disabled and 128 GB DDR4 SDRAM. We ran each workflow with exclusive access to node resources, on allocation sizes up to 32 compute nodes. We used three in-situ workflows coupled via the I/O library ADIOS [1] in our experiments:

LV couples two components: the LAMMPS [25] molecular dynamics simulator and Voro++ [53], a Voronoi tesselator. LV involves full-featured, realistic applications. The sample run used here simulates 16 000 atoms and streams position and velocity data into the tesselator for analysis and visualization. This application is a model for many cases in particle simulation and visualization.

HS also couples two components: a Heat Transfer [23] simulation with an analysis application, Stage Write. Heat Transfer is a mini-application that runs the heat equation over the grid of a given size and forwards simulation state to Stage Write, which produces output in the file system. This application executes the heat equation over the grid of a given size and forwards the state of the simulation to the Stage Write application, which writes it. This application is a model for many cases in numerical PDE calculations and I/O buffering and forwarding.

GP couples four components: Gray-Scott reaction-diffusion simulation; an analysis application, PDF calculator, applied to the Gray-Scott output; a visualization application, G-Plot, also applied to Gray-Scott output; and a second visualization application, P-Plot, applied to PDF output. GP involves applications of intermediate complexity. Two component applications, Gray-Scott and PDF calculator are configurable, but G-Plot and P-Plot are not. This application is a model for many cases in chemical reaction dynamics and more complex multi-purpose analysis workflows.

Application configuration options, shown in Tbl. 1, form a total of 2.9×10^9 possible configurations for LV (LAMMPS: 7.6×10^4; Voro++: 7.6×10^4), 5.1×10^{10} ones for HS (Heat Transfer: 5.4×10^6; Stage Write: 1.9×10^4), and 8.5×10^7 ones for GP (Gray-Scott: 1.9×10^4; PDF calculator: 9.0×10^3). We obtained expert-recommended configurations for each workflow.

In order to compare expert-recommended vs. good configurations, we generated for each workflow, as C_{pool}, 2000 configurations of

Table 1: Parameter spaces for our three target workflows

Workflow	Application	Parameter	Options
LV	LAMMPS	# processes	2, 3, ···, 1085
		# processes per node	1, 2, ···, 35
		# threads per process	1, 2, 3, 4
	Voro++	# processes	2, 3, ···, 1085
		# processes per node	1, 2, ···, 35
		# threads per process	1, 2, 3, 4
HS	Heat transfer	# processes in X	2, 3, ···, 32
		# processes in Y	2, 3, ···, 32
		# processes per node	1, 2, ···, 35
		# outputs	4, 8, ···, 32
		buffer size (MB)	1, 2, ···, 40
	Stage write	# processes	2, 3, ···, 1085
		# processes per node	1, 2, ···, 35
GP	Gray-Scott	# processes	2, 3, ···, 1085
		# processes per node	1, 2, ···, 35
	PDF calculate	# processes	1, 2, ···, 512
		# processes per node	1, 2, ···, 35
	Gray plot	# processes	1
	PDF plot	# processes	1

Table 2: Configurations and performance of benchmarks

Wf.	Objective	Option	Performance	Configuration
LV	Exec. time	Best	24.6 secs	(561, 25, 1, 75, 14, 1)
		Expert	36.8 secs	(288, 18, 2, 288, 18, 2)
	Comp. time	Best	3.13 core-hrs	(112, 28, 1, 36, 18, 4)
		Expert	4.07 core-hrs	(18, 18, 2, 18, 18, 2)
HS	Exec. time	Best	6.02 secs	(13, 17, 14, 4, 29, 19, 3)
		Expert	28.0 secs	(32, 17, 34, 4, 20, 560, 35)
	Comp. time	Best	0.517 core-hrs	(5, 25, 35, 4, 3, 5, 3)
		Expert	0.894 core-hrs	(8, 4, 32, 4, 20, 35, 35)
GP	Exec. time	Best	98.7 secs	(175, 13, 24, 23, 1, 1, 1)
		Expert	102 secs	(525, 35, 525, 35, 1, 1)
	Comp. time	Best	6.95 core-hrs	(66, 34, 41, 22, 1, 1)
		Expert	5.85 core-hrs	(35, 35, 35, 35, 1, 1)

randomly selected parameter values. Then, for each such configuration, we launched all workflow components at once and recorded the end-to-end wall-clock time of each component in the in-situ mode. We then used the longest component execution time as the configuration's *execution time*, and the product of execution time, number of computing nodes used, and number of cores per node as the configuration's *computer time*. We list in Tbl. 2 the expert-recommended and best configurations for each workflow and optimization objective, and their achieved performance. The expert recommendations only do well for GP. (Since the unconfigurable G-Plot is the bottleneck in GP, many GP configurations have similar execution times, close to that of G-Plot alone, 97.0 seconds.)

We also measured the execution and computer times of 500 configurations randomly selected from the parameter space of each configurable component application, and used these samples as component measurements, from which CEAL may select training samples for component models.

7.2 Evaluation Metrics

We use three metrics to evaluate auto-tuning algorithms.

7.2.1 Performance of best predicted configuration. Execution time is the better metric to minimize when seeking to optimize a single workflow, while computer time is better when multiple workflows are active at the same time. As our goal is to optimize the execution and computer times of in-situ workflows, the time achieved by the best configuration predicted for a workflow is an important evaluation metric.

7.2.2 Robustness in finding top configurations. We use **recall score** [30] to measure the error tolerance of an autotuning algorithm in predicting top configurations. Recall score is, for a value n, the percentage of configurations as predicted by the algorithm that are within the top n configurations. Given a set $\mathbf{c}$ of configurations for which we have workflow performance data $\mathbf{D_c}$, a model $\mathbf{M}$ for scoring configurations, and a function top(n, S) for selecting the top n entries from a set S of scored configurations, the recall score for n is:

$$S_r(n, \mathbf{c}, \mathbf{M}, \mathbf{D_c}) = |\text{top}(n, \mathbf{M}(\mathbf{c})) \cap \text{top}(n, \mathbf{D_c})|/n \times 100\%. \quad (3)$$

A higher recall score indicates a more robust auto-tuning algorithm; in general, $S_r(i)$ is more important than $S_r(j)$ ($i < j$). For $n = 1$, the recall score also represents the probability of finding the best-performing configuration.

7.2.3 Practicality in performance optimization. Since the data collection cost is considerable and unignorable for empirical auto-tuning algorithms, we monitor the least number of workflow runs required to pay off the auto-tuning cost, and use that metric to evaluate the practicality of auto-tuning algorithms. The **least number of uses** is $N = c/\Delta p$. Here, Δp is the actual improvement per workflow run in the optimization objective (execution time or computer time reduction) achieved by the auto-tuning algorithm relative to an expert recommendation, and c is the training data collection cost incurred in achieving the performance optimization objective, i.e., the sum of the workflow's execution times or computer times over all training samples.

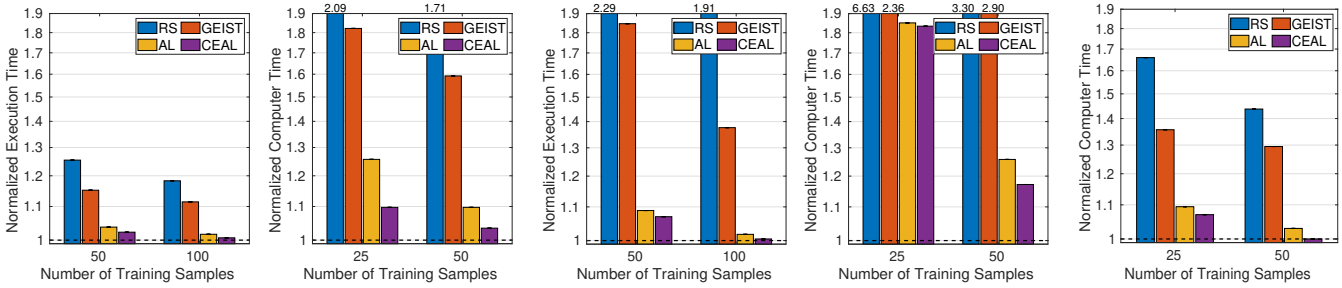
7.3 Comparison Targets

Since there exist few auto-tuning algorithms customized for in-situ workflows, we compare CEAL with three auto-tuning algorithms for general HPC applications. **RS** selects training data by random sampling. **AL** is a typical AL algorithm that iteratively selects as training samples a batch of the best configurations predicted by gradually refined models [6, 29]. **GEIST**, a state-of-the-art AL-based auto-tuning algorithm for finding performance-optimizing configurations [50] is guided by a parameter graph to choose training samples with a high possibility of being optimal (defined as in top 5% configurations) in each iteration. We also report in §7.5.2–7.5.4 on comparisons with a variant of CEAL, **ALpH** (introduced in §4) that uses black-box modeling to combine component models.

In all algorithms, we use the xgboost.XGBRegressor implementation of extreme gradient boosting regression as the original ML model. We adjust the hyperparameters of GEIST, AL, ALpH, and CEAL with and without historical measurements, such as I, m_0, and m_R in CEAL, and select the best settings for each algorithm. Hyperparameter optimization methods [56] are beyond the scope of this paper. In all experiments, we report the average of 100 runs of each algorithm.

7.4 Autotuning without Historical Measurements

We first examine the overall performance of our auto-tuner in the absence of historical measurements. We compare the actual performance of workflows auto-tuned by CEAL and others (§7.4.1), and explain CEAL's superiority by experimentally validating its design



(a) *LV* **(b)** *HS* **(c)** *GP*

Figure 5: The best configuration auto-tuned w/o historical measurements (dashed lines: the best configuration in the test set)

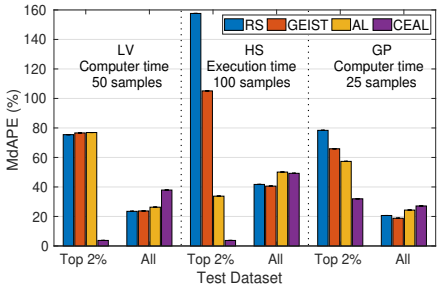
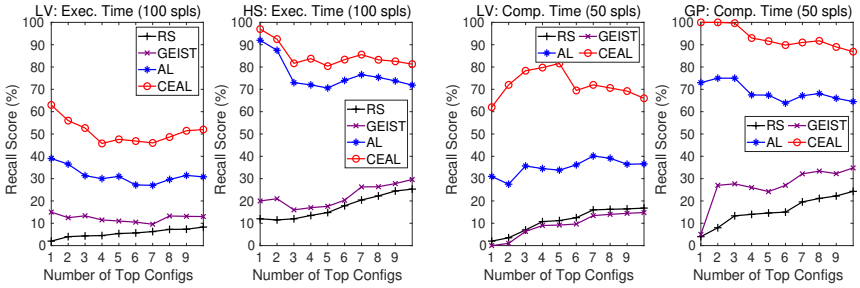


Figure 6: Prediction accuracy of models in autotuning w/o historical measurements



(a) *Optimizing execution time of LV and HS* **(b)** *Optimizing computer time of LV and GP*

Figure 7: Robustness of autotuning w/o historical measurements

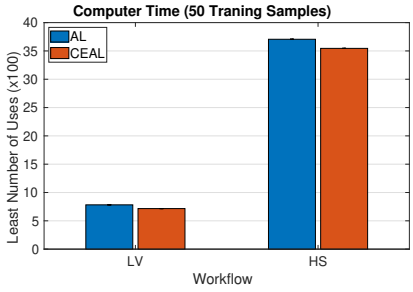
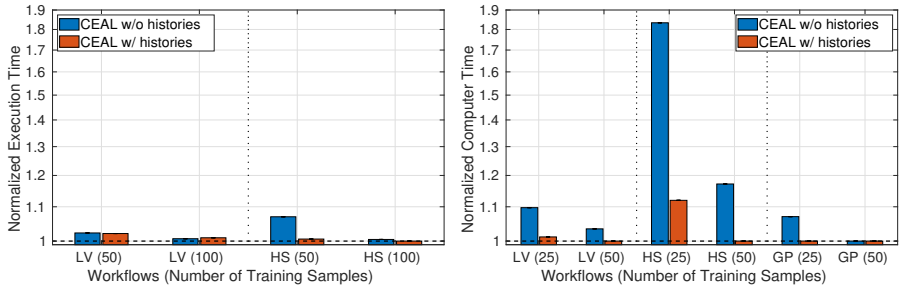


Figure 8: Practicality of autotuning w/o historical measurements

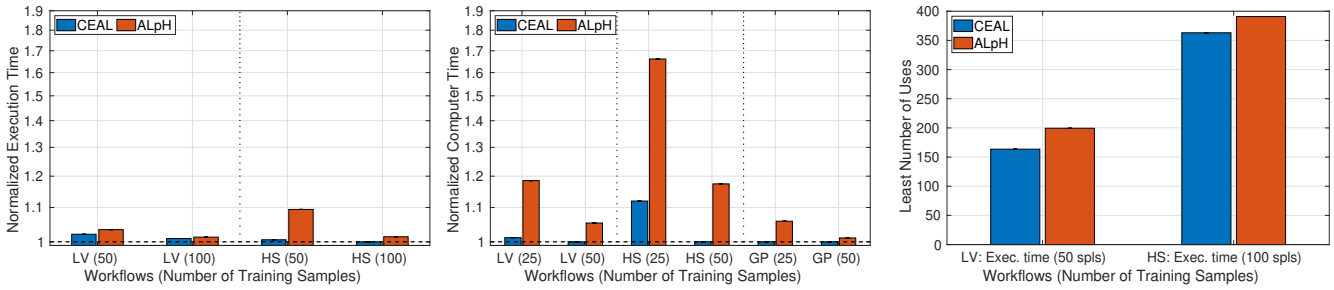


(a) *Execution time of predicted best conf.* **(b)** *Computer time of predicted best conf.*

Figure 9: Effect of historical measurements on CEAL (dashed lines: best test config.)

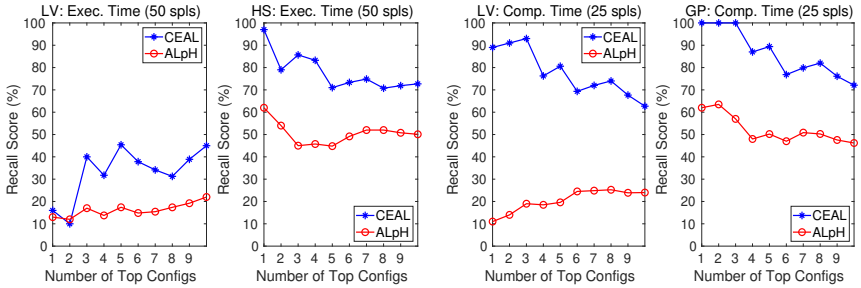
principle (§7.4.2). Also, we investigate CEAL's robustness and practicality (§7.4.3 and §7.4.4). (When comparing costs, we consider the cost of running an in-situ workflow as being comparable to the total cost of running all of its component applications separately.)

7.4.1 Actual performance of auto-tuned workflows. We measured the actual execution and computer times of the best configurations of LV, HS, and GP predicted by RS, GEIST, AL, and CEAL, and plot normalized values in Fig. 5, with the performance of the best configuration in the test set shown as "1" (the same for Figs. 9 and 10). We test CEAL with different numbers m of training samples by doubling m from 25 until the auto-tuned performance of LV is at most 5% worse than the best. We show here results for the largest two values of m tested: 100 and 50 for execution time and 50 and 25 for computer time. For consistency, we also select the same m for all workflows in all experiments. Fig. 5 shows that the execution and computer times achieved by CEAL are always better than by RS, GEIST, and AL. For example, CEAL improves both execution and computer times by 15–72% relative to RS and 10–60% relative to GEIST, and reduces LV computer time relative to AL by 12.7% and 5.7% with 25 and 50 training samples, respectively. CEAL outperforms AL, because performance models trained with the same number of training samples are much more accurate for component applications than in-situ workflows, and our method of determining workflow performance provides relatively accurate configuration ranking over top configurations.



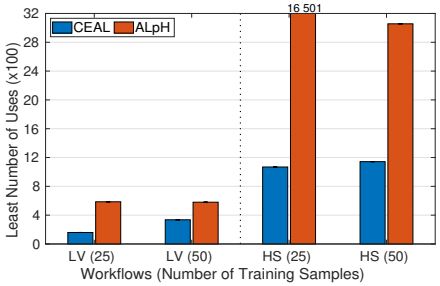
(a) *Optimizing execution time* **(b)** *Optimizing computer time*

Figure 10: Best configuration auto-tuned with histories (dashed lines: best test config.)



(a) *Optimizing execution time of LV and HS* **(b)** *Optimizing computer time of LV and GP*

Figure 11: Robustness of autotuning with historical measurements



(b) *Optimizing computer time*

Figure 12: Practicality of autotuning with historical measurements

7.4.2 Why CEAL outperforms RS, GEIST, and AL. The absolute percentage error (APE) of a sample i is $|(y_i - y_i')/y_i|$, where y_i is actual performance and y_i' is predicted performance. The median APE (MdAPE) for a set of samples is a commonly used measure of model prediction quality. To further understand why CEAL beats AL, GEIST and RS, we plot in Fig. 6 the MdAPEs of models generated by RS, GEIST, AL, and CEAL when used to predict performance over all, and the top 2%, of test dataset configurations. The MdAPEs of CEAL are much less than those of RS, GEIST, and AL for the top 2% of configurations; thus, CEAL outperforms RS, GEIST, and AL, even though its MdAPEs are comparable to, or a little higher than, those of RS, GEIST and AL over *all* configurations. This result verifies our intuition that by picking higher-performing configurations, CEAL improve prediction accuracy for top configurations and thus make best use of the few training samples allotted.

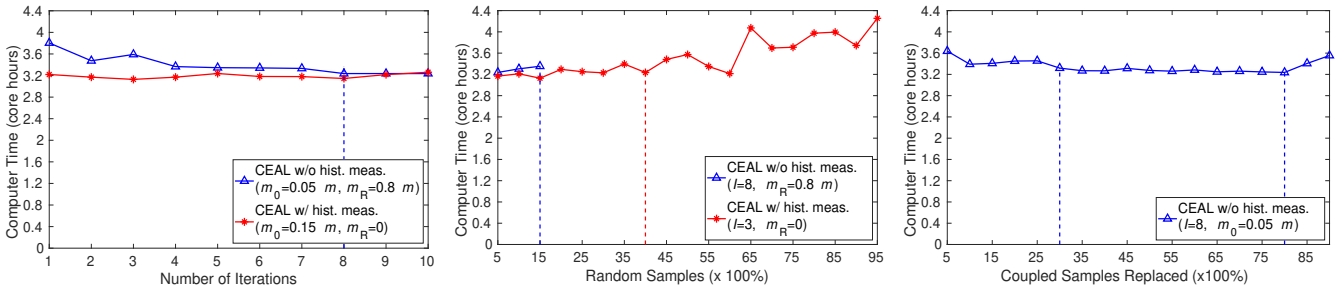
7.4.3 Robustness of auto-tuning algorithms. We use the recall scores (§7.2.2) of the top n ($n = 1, \cdots, 10$) configurations to evaluate the robustness of RS, GEIST, AL, and CEAL in auto-tuning our workflows for execution time and computer time. We see in Fig. 7 that CEAL is more robust than RS, GEIST, and AL in all cases. For the top-one configuration recall score (the most important performance measure), CEAL achieves 63% (or 62%) when optimizing the execution (or computer) time of LV with 100 (or 50) training samples, as against 2% (or 2%), 15% (or 0%), and 39% (or 31%) for RS, GEIST, and AL.

7.4.4 Practicality of auto-tuning algorithms. We examine the practicality of the four auto-tuning algorithms in optimizing the computer time of LV and HS. Since the computer time of LV/HS achieved by RS and GEIST with only 25 and 50 training samples is worse than that at the expert-recommended configuration, the practicality of RS and GEIST is limited. Then, we focus on auto-tuning the computer time of LV and HS by AL and CEAL with 50 training samples, and plot the least number of runs (§7.2.3) in Fig. 8, which reveals that CEAL is superior to AL in terms of practicality. For example, LV is worth auto-tuning by CEAL because it is expected to run 716 times, 8.4% less than the 782 times required for AL. We attribute CEAL's superiority to its more accurate selection of training samples that take less computer time, as boosted by the combined low-fidelity model.

7.5 With Historical Component Measurements

We next examine auto-tuner performance when component historical measurements can be used. We show that CEAL can make good use of these component measurements to enhance workflow performance (§7.5.1). We also demonstrate the superiority of CEAL over ALpH's component integration method (§7.5.2–7.5.4).

7.5.1 Effect of previous component measurements. If historical performance measurements are available for component applications, then CEAL can use those measurements to train component application models without charge against its training sample budget, and then perform more full workflow runs that would otherwise be possible. In order to explore the benefits that results, we compare workflow performance when optimized by CEAL with and without historical measurements. In the first case, we assume no historical measurements, and thus subtract the m_R component samples used to train component models from CEAL's training sample budget; in the second, we treat those measurements as historical and do not



(a) I **(b)** m_0/m **(c)** m_R/m

Figure 13: Impact of parameter settings.

count them toward the cost. We see from Fig. 9 that historical measurements improve CEAL performance in most cases. In addition, Fig. 9b shows that historical measurements help CEAL, in the case of 25 training samples, reduce computer time for LV by 7.8%, HS by 38.9%, and GP by 6.6%. We conclude that CEAL can make effective use of historical component measurements.

7.5.2 Performance of auto-tuned workflows. Recall from §4 that ALpH differs from CEAL in using black-box rather than white-box modeling to combine component model performance predictions. We compare these approaches to combining component models by measuring LV, HS, and GP performance when auto-tuned by ALpH and CEAL with different numbers of training samples. The normalized execution and computer times in Fig. 10 show that CEAL is superior to ALpH in all cases. For example, Fig. 10b shows that the computer times of LV, HS, and GP when optimized by CEAL with 25 training samples are 14.7%, 32.6%, and 5.6% less than when optimized by ALpH, respectively. We attribute CEAL's superiority here to the effectiveness of its component combination in improving sampling distribution for black-box modeling.

7.5.3 Robustness of auto-tuning algorithms. We also evaluate the robustness of ALpH and CEAL with historical component measurements in optimizing workflow execution and computer times. The recall scores (§7.2.2) at top configurations are plotted in Fig. 11; we see that CEAL is always more robust than ALpH. It is worth noting that CEAL's best-1, best-2, and best-3 configuration recall scores are all 100% in optimizing the computer time of GP with only 25 training samples.

7.5.4 Practicality of auto-tuning algorithms. We examine the practicality of ALpH and CEAL with historical component measurements for auto-tuning LV and HS, and plot the least number of runs (§7.2.3) in Fig. 12. It can be observed that when CEAL is used to optimize LV execution time with 50 training samples and LV computer time with 25 training samples, the number of LV runs (§7.2.3) required to recoup the auto-tuning cost is only 164 and 160, implying great practicality of CEAL. As to the execution time cost, we consider workflow instances at training configurations to run sequentially, even though the training data collection can sometimes be completed in parallel.

7.6 Hyperparameter Sensitivity Analysis

To study CEAL's sensitivity to hyper-parameter values, we use it to predict the best computer time for LV with 50 training samples. Fig. 13 shows the actual computer times of the best configurations predicted in various settings. (1) We run CEAL with from 1 to 10 iterations (I). We see in Fig. 13a that LV computer time converges to the best after eight iterations for CEAL without historical measurements and it converges faster for CEAL with historical measurements. (2) We run CEAL as we increase the number of random samples (m_0) from $5\% \cdot m$ to $(m - m_R)$ at an interval of $5\% \cdot m$. Fig. 13b shows that LV computer time is stable over a large range of m_0: from $5\% \cdot m$ to $15\% \cdot m$ for CEAL without historical measurements and from $5\% \cdot m$ to $40\% \cdot m$ for CEAL with historical measurements. (3) We test CEAL as the number of coupled random samples replaced by component samples (m_R) is varied from $5\% \cdot m$ to $(m - m_0)$ at an interval of $5\% \cdot m$. Fig. 13c shows that LV computer time is stable over a large range of m_R: from 30% to 80% training samples.

Repeating these studies on our other workflows and optimization metrics gave similar results, except that in two cases we found that an m_0 value of around $30\% \cdot m$ was best, indicating that the low-fidelity model was not performing well (as discussed in §5).

8 RELATED WORK

Our approach is related to the research in the following three areas.

8.1 Analytical Modeling of In-situ Workflows

Few analytical models [26, 28] have been proposed for estimating in-situ workflow performance. Fu et al. [21] analyzed and modeled the performance of their Zipper system, which integrates simulation and analysis to eliminate inefficiency issues in I/O libraries used by in-situ workflows. They experimentally verified an execution time model for end-to-end Zipper-coupled in-situ workflows, expressed as the maximum of simulation time, analysis time, and data transfer time, similar to our analytical coupling model. Although Zipper, as an well-optimized in-situ integration tool, almost makes the execution time of coupled applications approach the lower bound of the workflow time, this expression still cannot exactly describe the execution time of Zipper-coupled workflows. At present, there is a lack of accurate analytical model of in-situ workflows.

8.2 Combining White and Black Box Modeling

Didona et al. [13] provided three ensemble algorithms of analytical modeling and ML, namely K nearest neighbors (KNN), hybrid boosting (HyBoost), and probing (PR), to enhance performance modeling and prediction.

KNN evaluates the accuracy of several analytical/ML models and chooses the best one for performance prediction. The measured samples are evenly divided into two training and test sets with the same distribution. In total, there are $M + 1$ performance models including one analytical model (AM) and M ML models from M different ML regressors trained with the same data. For any given configuration, its K nearest configurations in the test set are used to verify the accuracy of each performance model. The model with the smallest prediction error over the K neighbors is selected as the performance model at the given configuration. However, the method does not only assume that a parameter graph representing distance among configurations is known, but also fails to improve any candidate analytical/ML model itself.

In HyBoost, a chain of ML models are used to learn the residual errors of an AM. The actual performance prediction is based on the output of AM, adjusted by the error corrector function. This algorithm assumes that the function that characterizes the error of the AM may be learned more easily than the original target function that describes the relation between input and output variables. However, this assumption requires a relatively higher accuracy of the AM. Therefore, the aforementioned two ensemble algorithms are not suitable for auto-tuning an in-situ workflow that has an extremely huge configuration space but a rough AM.

Along with bootstrapping, PR uses ML to perform predictions exclusively on the regions of the configuration space in which the AM does not achieve sufficient accuracy. Although we take inspiration from this idea, PR is to form an accurate model across the whole space, while we use lower cost to generate a model only for accurately identify the top configuration.

8.3 Auto-tuning for HPC Applications

This review complements an existing HPC auto-tuning survey [5].

Ritter et al. [39] improved an empirical modeling tool, Extra-P [7], by reducing the number of its experiments from an exponential increase to a polynomial increase with the number of parameters. However, the generated performance model represents the metric by following a normal form of functions in parameters. The limitation from the normal form degrades the model accuracy, significantly affecting subsequent performance optimazation. Our paper targets building cost-effective models used for performance optimazation.

Sourouri et al. [46] integrated fine-grained auto-tuning with user-controllable hardware switches and threads to improve the energy efficiency of a memory-bound HPC application. However, their application-specific AM is not directly applicable to other applications. Popov et al. [38] reduced data collection costs by extracting and running short representative codelets to jointly optimize page and thread mappings in NUMA systems. However, short representative codelets extracted from individual applications do not reflect the complex interaction between component applicaitons. Hence, these methods cannot be applied to in-situ workflow autotuning.

Others have applied active learning to HPC auto-tuning [4, 18, 35]. Thiagarajan et al. use semi-supervised learning based on a parameter graph for fast parameter space exploration [50]; it can auto-tune HPC applications with configuration parameter spaces in the range of 18000. Our work targets optimizing coupled HPC applications with configuration space size of 10^{10} or more at an affordable cost.

Marathe et al. [30] proposed an HPC application auto-tuning algorithm that uses three fully connected NNs to capture the relationship between configuration parameters and performance metrics from many cheap and a few expensive training samples. However, the small, cheap samples often have low similarity to the large, expensive samples, failing to provide transferable knowledge for accurate model generation.

9 CONCLUSION AND FUTURE WORK

The use of auto-tuners based on pure black- or white-box models has been considered infeasible for in-situ workflows due to their large configuration spaces and complex interactions among components. Our new approach achieves high-quality auto-tuners for in-situ workflows even under a tight cost budget, by leveraging the fact that in-situ workflows often couple multiple component applications in simple structures and correlate with the components in terms of performance. Specifically, our CEAL algorithm: 1) combines component models into a low-fidelity workflow model, and 2) uses the low-fidelity model to guide the collection of samples for training a high-fidelity model. Experiments with three scientific in-situ workflows confirm the viability of CEAL, showing it can build auto-tuners that are better at finding best-performing configurations than those built with other methods. In one example, CEAL with just 50 training samples optimizes workflow execution time so well that only 164 subsequent runs are required to recoup cost.

In future, we will use other black-box techiques such as RL and BO to improve auto-tuner quality over AL in the bootstraping method. Generally, auto-tuning cost is considerable and workflow surrogate models are rarely reused for other workflows. Compared with other software, HPC applications running on supercomputers have obvious common characteristics. The agent in RL can achieve transfer learning from historial HPC application autotuning and be built on powerful NNs. Specially, our approach can incorporate RL to dynamically update the sample pool containing higher-performing configurations according to measured configurations. Also, to avoid errors and interference (e.g., from network congestion), existing methods select the average/median of three to five measurements at each chosen configuration. BO may naturally consider noise in selecting top configurations, thus reducing actual measurements.

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC) from U.S. Department of Energy and start-up funds from Southern Illinois University Carbondale. We also thank the WORKFLOW and PACC projects' computer time allocation from Laboratory Computing Resource Center at Argonne National Laboratory, and Dr. Zhengchun Liu's help on the usage of the Python package xgboost.XGBRegressor.

REFERENCES

[1] ADIOS, 2021. https://csmd.ornl.gov/adios.

[2] T. G. Armstrong, J. M. Wozniak, M. Wilde, and I. T. Foster. Compiler techniques for massively scalable implicit task parallelism. In *IEEE/ACM Intl. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 299–310, Nov 2014.

[3] U. Ayachit, et al. Performance analysis, design considerations, and applications of extreme-scale in situ infrastructures. In *IEEE/ACM Intl. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2016.

[4] P. Balaprakash, R. B. Gramacy, and S. M. Wild. Active-learning-based surrogate models for empirical performance tuning. In *IEEE Cluster*, Sep 2013.

[5] P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, and R. Vuduc. Autotuning in high-performance computing applications. *Proceedings of the IEEE*, 106(11):2068–2083, 2018.

[6] B. Behzad, S. Byna, Prabhat, and M. Snir. Optimizing I/O performance of HPC applications with autotuning. *ACM Trans. on Parallel Computing (TOPC)*, 5(4): 15:1–15:27, 2019.

[7] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf. Using automated performance modeling to find scalability bugs in complex codes. In *IEEE/ACM Intl. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, Nov 2013.

[8] A. Calotoiu, M. Copik, T. Hoefler, M. Ritter, S. Shudler, and F. Wolf. ExtraPeak: Advanced automatic performance modeling for HPC applications. In *Spring Software for Exascale Computing*, pages 453–482, Jul 2020.

[9] Z. Cao, V. Tarasov, S. Tiwari, and E. Zadok. Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems. In *USENIX Annual Technical Conference (ATC)*, pages 893–907, Jul 2018.

[10] T. Chen and C. Guestrin. XGBoost: A scalable tree boosting system. In *ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 785–794, Aug 2016.

[11] J. Choi, D. F. Richards, L. V. Kale, and A. Bhatele. End-to-end performance modeling of distributed gpu applications. In *ACM International Conference on Supercomputing (ICS)*, pages 30:1–12, Jun 2020.

[12] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, and N. Podhorszki. Flexpath: Type-based publish/subscribe system for large-scale science analytics. In *IEEE/ACM intl. Symp. on Cluster, Cloud, and Internet Computing (CCGrid)*, pages 246–255, May 2014.

[13] D. Didona, F. Quaglia, P. Romano, and E. Torre. Enhancing performance prediction robustness by combining analytical modeling and machine learning. In *ACM International Conference on Performance Engineering (ICPE)*, pages 145–156, Jan 2015.

[14] C. Docan, M. Parashar, and S. Klasky. DataSpaces: An interaction and coordination framework for coupled simulation workflows. *Cluster Computing*, 15(2): 163–181, 2012.

[15] M. Doucet et al. Machine learning for neutron scattering at ORNL. *Machine Learning: Science and Technology*, 2(2):023001, jan 2021. doi: 10.1088/2632-2153/abcf88. URL https://doi.org/10.1088/2632-2153/abcf88.

[16] M. Dreher and T. Peterka. Decaf: Decoupled dataflows for in situ high-performance workflows. Technical Report ANL/MCS-TM-371, ANL, Jul 2017.

[17] S. Duan, P. Subedi, P. E. Davis, and M. Parashar. Addressing data resiliency for staging based scientific workflows. In *IEEE/ACM Intl. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 87:1–22, Nov 2019.

[18] D. Duplyakin, J. Brown, and R. Ricci. Active learning in performance analysis. In *IEEE Cluster*, pages 182–191, Taipei, Taiwan, Sep 2016.

[19] I. Foster, M. Ainsworth, J. Bessac, F. Cappello, J. Choi, S. Di, Z. Di, A. M. Gok, H. Guo, K. A. Huck, C. Kelly, S. Klasky, K. K. van Dam, X. Liang, K. Mehta, M. Parashar, T. Peterka, L. Pouchard, T. Shu, O. Tugluk, H. van Dam, L. Wan, M. Wolf, J. M. Wozniak, W. Xu, I. Yakushin, S. Yoo, and T. Munson. Online data analysis and reduction: An important co-design motif for extreme-scale computers. *International Journal of High Performance Computing Applications (IJHPCA)*, 2021.

[20] G. Fox, S. Jha, and L. Ramakrishnan. Streaming and Steering Applications: Requirements and Infrastructure final report, 2015.

[21] Y. Fu, F. Li, F. Song, and Z. Chen. Performance analysis and optimization of in-situ integration of simulation with data analysis: Zipping applications up. In *ACM Intl. Symp. on High-Performance Parallel and Distributed Computing (HPDC)*, pages 192–205, Jun 2018.

[22] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley. Google vizier: A service for black-box optimization. In *ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data (KDD)*, pages 1487–1496, Aug 2017.

[23] Heat Transfer, 2019. https://github.com/CODARcode/Example-Heat_Transfer/blob/master/README.adoc.

[24] K. Keahey and J. Ahrens. Future Online Analysis Platform workshop report, 2017.

[25] LAMMPS, 2021. https://lammps.sandia.gov.

[26] M. Larsen, C. Harrison, J. Kress, D. Pugmire, J. S. Meredith, and H. Childs. Performance modeling of in situ rendering. In *IEEE/ACM Intl. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2016.

[27] Q. Liu, et al. Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks. *Concurrency and Computation: Practice and Experience*, 26(7):1453–1473, 2014.

[28] P. Malakar, V. Vishwanath, T. Munson, C. Knight, M. Hereld, S. Leyffer, and M. E. Papka. Optimal scheduling of in-situ analysis for large-scale scientific simulations. In *IEEE/ACM Intl. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, Austin, TX, USA, Nov 2015.

[29] A. Mametjanov, P. Balaprakash, C. Choudary, P. D. Hovland, S. M. Wild, and G. Sabin. Autotuning FPGA design parameters for performance and power. In *IEEE Intl. Symp. on Field-Programmable Custom Computing Machines*, pages 84–91, May 2015.

[30] A. Marathe, R. Anirudh, N. Jain, A. Bhatele, J. Thiagarajan, B. Kailkhura, J.-S. Yeom, B. Rountree, and T. Gamblin. Performance modeling under resource constraints using deep transfer learning. In *IEEE/ACM Intl. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, Nov 2017.

[31] K. Meng, J. Li, G. Tan, and N. Sun. A pattern based algorithmic autotuner for graph processing on GPUs. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 201–213, Feb 2019.

[32] H. Menon, A. Bhatele, and T. Gamblin. Auto-tuning parameter choices in HPC applications using bayesian optimization. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 831–840, May 2020.

[33] A. Morcos, H. Yu, M. Paganini, and Y. Tian. One ticket to win them all: generalizing lottery ticket initializations across datasets and optimizers. In *ACM Intl. Conf. on Neural Information Processing Systems (NeurIPS)*, pages 1–11, Dec 2019.

[34] J. Mu, M. Wang, L. Li, J. Yang, W. Lin, and W. Zhang. A history-based auto-tuning framework for fast and high-performance DNN design on GPU. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, Jul 2020.

[35] W. F. Ogilvie, P. Petoumenos, Z. Wang, and H. Leather. Minimizing the cost of iterative compilation with active learning. In *IEEE/ACM Intl. Symp. on Code Generation and Optimization (CGO)*, pages 245–256, Feb 2017.

[36] J. Ozik, N. T. Collier, J. M. Wozniak, C. M. Macal, and G. An. Extreme-scale dynamic exploration of a distributed agent-based model with the emews framework. *IEEE Transactions on Computational Social Systems*, 5(3):884–895, 2018.

[37] T. Peterka. ASCR Workshop on In Situ Data Management report, 2019.

[38] M. Popov, A. Jimborean, and D. Black-Schaffer. Efficient thread/page/parallelism autotuning for NUMA systems. In *ACM International Conference on Supercomputing (ICS)*, pages 342–353, Jun 2019.

[39] M. Ritter, A. Calotoiu, S. Rinke, T. Reimann, T. Hoefler, and F. Wolf. Learning cost-effective sampling strategies for empirical performance modeling. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 884–895, May 2020.

[40] T. Shu. *Performance Optimization and Energy Efficiency of Big-data Computing Workflows*. Dissertation, New Jersey Institute of Technology, Newark, NJ, USA, Aug 2017. http://archives.njit.edu/vol01/etd/2010s/2017/njit-etd2017-096/njit-etd2017-096.pdf.

[41] T. Shu and C. Q. Wu. Energy-efficient mapping of big data workflows under deadline constraints. In *Proc. of Workshop on Workflows in Support of Large-Scale Science in conjunction with ACM/IEEE Supercomputing Conference*, pages 34–43, Salt Lake City, UT, USA, Nov 2016. http://ceur-ws.org/Vol-1800/paper5.pdf.

[42] T. Shu and C. Q. Wu. Energy-efficient dynamic scheduling of deadline-constrained MapReduce workflows. In *Proc. of IEEE eScience*, pages 393–402, Auckland, New Zealand, Oct 2017.

[43] T. Shu and C. Q. Wu. Performance optimization of Hadoop workflows in public clouds through adaptive task partitioning. In *Proc. of IEEE INFOCOM*, pages 2349–2357, Atlanta, GA, USA, May 2017.

[44] T. Shu and C. Q. Wu. Energy-efficient mapping of large-scale workflows under deadline constraints in big data computing systems. *Future Generation Computer Systems (FGCS)*, 110:515–530, 2020. https://www.sciencedirect.com/science/article/pii/S0167739X17300468.

[45] T. Shu, Y. Guo, J. Wozniak, X. Ding, I. Foster, and T. Kurc. Poster: In-situ workflow auto-tuning through combining component models. In *Proc. of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 467–468, Virtual Event, Feb 2021.

[46] M. Sourouri, E. B. Raknes, N. Reissmann, J. Langguth, D. Hackenberg, R. Schöne, and P. G. Kjeldsberg. Towards fine-grained dynamic tuning of HPC applications on modern multi-core architectures. In *IEEE/ACM Intl. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2017.

[47] R. Stevens, J. Nichols, and K. Yelick. AI for Science Report on the Department of Energy (DOE) Town Halls on Artificial Intelligence (AI) for Science, 2020.

[48] P. Subedi, P. Davis, S. Duan, S. Klasky, H. Kolla, and M. Parashar. Stacker: An autonomic data movement engine for extreme-scale data staging-based in-situ workflows. In *IEEE/ACM Intl. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2018.

[49] J. Sun, G. Sun, S. Zhan, J. Zhang, and Y. Chen. Automated performance modeling of hpc applications using machine learning. *IEEE Trans. on Computers (TC)*, 69 (5):749–763, 2020.

[50] J. J. Thiagarajan, N. Jain, R. Anirudh, A. Gimenez, R. Sridhar, A. Marathe, T. Wang, M. Emani, A. Bhatele, and T. Gamblin. Bootstrapping parameter space exploration for fast tuning. In *ACM International Conference on Supercomputing (ICS)*, pages 385–395, Jun 2018.

[51] P. Tillet and D. Cox. Input-aware auto-tuning of compute-bound hpc kernels. In *IEEE/ACM Intl. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2017.

[52] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka. Topology-aware data movement and staging for i/o acceleration on blue gene/p supercomputing systems. In *IEEE/ACM Intl. Conf. on High Performance Computing, Networking, Storage and Analysis (SC)*, Nov 2011.

[53] Voro++, 2021. http://math.lbl.gov/voro++.

[54] J. M. Wozniak, P. Davis, T. Shu, J. Ozik, N. Collier, I. Foster, T. Brettin, and R. Stevens. Scaling deep learning for cancer with advanced workflow storage integration. In *Proc. of the 4th Workshop on Machine Learning in HPC Environments in conjunction with ACM/IEEE Supercomputing Conference*, pages 114–123, Dallas, TX, USA, Nov 2018.

[55] J. M. Wozniak, M. Dorier, R. Ross, T. Shu, T. Kurc, L. Tang, N. Podhorszki, and M. Wolf. Mpi jobs within mpi jobs: a practical way of enabling task-level fault-tolerance in hpc workflows. *Future Generation Computer Systems (FGCS)*, 101:576–589, 2019.

[56] Y. Xia, C. Liu, Yuying, and N. Liu. A boosted decision tree approach using Bayesian hyper-parameter optimization for credit scoring. *Expert Systems with Applications*, 75:225–241, 2017.

[57] Z. Yu, Z. Bei, and X. Qian. Datasize-aware high dimensional configurations auto-tuning of in-memory cluster computing. In *ACM Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 564–577, Mar 2018.

[58] F. Zhang, T. Jin, Q. Sun, M. Romanus, H. Bui, S. Klasky, and M. Parashar. In-memory staging and data-centric task placement for coupled scientific simulation workflows. *Concurrency and Computation: Practice and Experience*, 29(12): 1–19, 2017.

[59] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T.-A. Nguyen, J. Cao, H. Abbasi, S. Klasky, N. Podhorszki, and H. Yu. FlexIO: I/O middleware for location-flexible scientific data analytics. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 320–331, May 2013.