# Artemis: Automatic Runtime Tuning of Parallel Execution Parameters Using Machine Learning

Chad Wood[1(✉)], Giorgis Georgakoudis[2], David Beckingsale[2], David Poliakoff[3], Alfredo Gimenez[2], Kevin Huck[1], Allen Malony[1], and Todd Gamblin[2]

[1] University of Oregon, Eugene, OR, USA
{cdw,khuck,malony}@cs.uoregon.edu

[2] Lawrence Livermore National Laboratory, Livermore, CA, USA
{georgakoudis1,beckingsale1,giminez1,gamblin2}@llnl.gov

[3] Sandia National Laboratory, Albequerque, NM, USA
dzpolia@sandia.gov

**Abstract.** Portable parallel programming models provide the potential for high performance and productivity, however they come with a multitude of runtime parameters that can have significant impact on execution performance. Selecting the optimal set of those parameters is non-trivial, so that HPC applications perform well in different system environments and on different input data sets, without the need of time consuming parameter exploration or major algorithmic adjustments.

We present Artemis, a method for online, feedback-driven, automatic parameter tuning using machine learning that is generalizable and suitable for integration into high-performance codes. Artemis monitors execution at runtime and creates adaptive models for tuning execution parameters, while being minimally invasive in application development and runtime overhead. We demonstrate the effectiveness of Artemis by optimizing the execution times of three HPC proxy applications: Cleverleaf, LULESH, and Kokkos Kernels SpMV. Evaluation shows that Artemis selects the optimal execution policy with over 85% accuracy, has modest monitoring overhead of less than 9%, and increases execution speed by up to 47% despite its runtime overhead.

**Keywords:** Artemis · HPC · Performance · In situ · Machine learning

## 1 Introduction

HPC software can contain tens to thousands of parallel code regions, each of which may have independent performance tuning parameters. Optimal choices for these tuning parameters can be specific to a target system architecture, the set of input data to be processed, or the overall shared state of the machine during a job's execution. There are costs associated with discovering and maintaining optimal choices, in a developer's time to manually adjust settings and rebuild

B. L. Chamberlain et al. (Eds.): ISC High Performance 2021, LNCS 12728, pp. 453–472, 2021.
https://doi.org/10.1007/978-3-030-78713-4_24

projects, or the compute time to explore the space of possible configurations to find optimal settings automatically.

The goal of *performance portability* in HPC is for applications to operate optimally across a range of current and future systems without the need for costly code interventions in each new deployment. Given large job scales, increasing software complexity, platform diversity, and hardware performance variability, a performance portability is a challenging problem – with the same inputs, code performance is observed to change between invocations on the same machine and, worse, can be variable even during execution.

Recent work has turned to machine learning techniques to train classification models on code and execution feature vectors that then can be used to make dynamic tuning selection for each kernel of interest [3]. For instance, the Apollo [9] work demonstrated the use of offline machine learning methods to optimize the selection of RAJA [8] kernels at runtime. The RAJA programming methodology provides abstractions that allow code regions to be implemented once but compiled for a variety of architectures, with several execution policies capable of being selected at runtime. Apollo's offline training approach built statistical classifiers that directly selected values for tuning parameters. The classification model could then be embedded in RAJA programs to provide a dynamic, low-overhead, data-driven auto-tuning framework. The decision to do offline training was a trade-off Apollo made to avoid costly online search for autotuning.

Offline machine learning methods are not sufficient for guiding *online optimizations* that deliver general performance portability. There are several reasons for this to be the case: 1. Without knowing what the user is actually doing, combinatorial exploration of all possible settings is difficult to exhaust, even with a decent sampling strategy. A great many different models need to be represented by whatever ends up being deployed, hopefully providing optimal recommendations for every unique combination of architectures, configurations, input decks, and so on. 2. In order to cover all scenarios, the expense of training and re-training will grow. The entire campaign of parameter testing would need to be done with any new code deployment, significant modification, change in configuration, use of new input deck, or increase in job scale. Certainly, moving to a new platform or modification of an existing platform could trigger a new training study. Ideally, the testing should happen at the full scale and duration that the job was intended to be run at once its model was in use, but this is a costly proposition. Ultimately, this suggests that offline training is unable to fully capture enough for model fitness to be reliable over time. 3. Once trained offline, static models are unable to adapt to changes between application invocations or simulation steps in a workflow. Such changes can make even very good models go stale over time. Furthermore, the potential dynamic variations in the execution environment can expose gaps in the model due to the fact that they never occurred during training.

To further motivate the need for online methods, we note the paradigmatic shift in HPC underway in the move to extreme scales and cloud-based computing. Applications are increasingly being developed and deployed where it is accepted as a given that there will be dynamism in their runtime environment. Even within tightly-controlled on-site dedicated clusters, novel *in situ* resources and services are being deployed in support of classic block-synchronous applications, decreasing the emphasis on their synchronous behavior to maximally saturate available computation and I/O resources.

Our current research is motivated by the need to address tuning challenges presented by these performance complexities and realities of new *in situ* development models: the scale of jobs, asynchronous data movement, and dynamic performance characteristics of modern hardware. Instead of working against the general nature of the problem, we propose to embrace it and investigate the productive outcomes of adopting modern (online) training techniques. In the spirit of prior work, we created the *Artemis* continuous tuning framework to analyze code kernels online during application execution. Artemis trains new kernel performance models *in situ*, deploying and evaluating them at runtime, observing each model's recommendations during execution to rate its ongoing fitness.

Our primary research contributions are:

– We present Artemis, an online framework that dynamically tunes the execution of parallel regions by training optimizing models.
– We provide an implementation of a RAJA parallel execution policy that uses Artemis to optimize the execution of `forall` and `collapse` loop pattern.
– We extend Kokkos to use Artemis for tuning CUDA execution on GPUs.
– We evaluate Artemis using three HPC proxy applications: LULESH, Cleverleaf, and Kokkos Kernels SpMV. Results show that Artemis has overhead of less than 9%, and model training and evaluation overhead is in the order of hundreds of microseconds. Artemis selects the optimal policy $\tilde{8}5\%$ of the time, and can provide up to 47% speedup.

## 2 Background

Parallel programming frameworks have emerged to address the performance portability challenge by providing a "write once, run anywhere" methodology where alternate versions of a code section (called kernels) can be generated to target architectural tuning parameters. In this manner, the programming methodology decouples the specification of a kernel's parallelism from the parameters that govern policies for how to execute the work in different forms. The tuning of the policy choices and execution variants can be done without changing the high-level program.

Parallel frameworks such as RAJA [19] and Kokkos [13,14] use lightweight syntax and standard C++ features for portability and ease of integration into production applications. Related prior work on Apollo [9] focused on developing an autotuning extension for RAJA for input-dependent parameters where the best kernel execution policy depends on information known only at application runtime. However, Apollo's methodology required executions under all runtime scenarios to create an offline static training database, leading to many of the limitations discussed in the introduction. Thus, it is interesting to pursue a new question: is it possible to train a classification model online and apply it during application execution? Of course, this question immediately raises several concerns, mainly having to do with how training data is generated, the overhead of measurement, and the complexity costs of machine learning algorithms.

## 3 Artemis: Design and Implementation

Artemis is at once a methodology for in situ, ML-based performance auto-tuning and an architecture and operational framework for its implementation. The following captures these aspects as we describe how Artemis actually works. In a nutshell, it is the observation of an application's execution of its tunable parallel code regions, extracting features and performance data with different execution policies, coupled with the training of ML models online to select optimized execution policies per-region and feature set.

### 3.1 Design

Without loss of generality, Artemis thinks of applications being iterative where a sequence of *steps* are conducted during which parallel regions are being executed. At the end of those steps, the application ends.

If the a parallel region is to be tuned, it must be provide the different execution policy variants it can choose between, and then Artemis must be invoked for that region. In the case of the reference implementations presented here, this can be largely automated.

The *user* of Artemis need not be thought of as the ultimate end-user of an application, but more likely the developer implementing a performance portability framework such as RAJA or Kokkos within some application. By design, our embedding of an Artemis interface into the portability framework layer enables all parallel regions of an application to be automatically decorated with the necessary Artemis API calls, and furnished with a set of common execution policies that come pre-packaged, and may be integrated into any application making use of that performance portability framework. Artemis is designed to be extensible and programmable, so expert users are always going to be able to provide their own execution policy variants, or make use of the Artemis API directly without the benefits of a performance portability layer managing it.

In the common case where an application is making use of performance portability framework as described above, all an end-user will need to do to is to select to enable Artemis functionality at build time, and then at run time they could opt to enable the Artemis tuning capabilities for any given session, which would then exploit the built-in policies that are bundled with the framework. Essentially, this is the end of involvement for the Artemis user.

Within a step, each parallel region executed is done so for a particular policy as determined by the policy model. Artemis controls how the policy model behaves. It could either be controlled to test out different policies during training, thereby allowing performance measurements to be obtained for analysis, or it could select a particular policy determined by the auto-tuned model evaluation. Each application step represents an opportunity for parallel region training or re-training. Within a step, each encounter with an Artemis-guided parallel region allows that region's model to make an optimized policy selection based on immediately-observed local features.

Artemis instruments parallel regions to collect data on their execution and tune them. Marking the beginning of region execution, the user additionally provides a set of *features* that characterize the execution and a set of execution *policies* that are selectable for the execution of this region. After the call marking the beginning of a region, the user calls the Artemis API function that returns the policy to use when executing the region. The region proceeds to execute a refactored variant of itself that corresponds to that policy selection. Finally, the instrumented region calls the Artemis API to mark the end of its execution, and Artemis makes note of the features and performance measurements. Region execution time is the primary measurement of interest, but it is possible to capture other performance data for analysis.

Artemis is implemented as a runtime library that merges with the application to provide region performance/metadata measurement/analysis, ML model training, and auto-tuning optimization. It presently targets parallel MPI programs that use RAJA or Kokkos for on-node parallelization.

## 3.2 Training and Optimization

The set of user-provided features and policies for each region are the input data to Artemis for ML training and optimization. During training, Artemis explores among the available policies and in particular measures their execution times, which is the optimization target we selected for our experimental evaluation. Artemis keeps per-region records of the feature set, policy, and measured execution time as tuples of (feature set, policy, execution time) to compile the training data and create an optimizing policy selection model. Whenever a region is executed multiple times per step, if different features are captured or policies are explored, each unique combination will have executions times recorded for use in model development.

By design, Artemis exposes an API call to the user to invoke optimization on-demand. Artemis expects the user to invoke the optimization API function after a sensible amount of computation has executed, permitting Artemis to have collected a representative set of measurement records. This can be different for different applications, and depends somewhat on the number of optimization points to be explored when searching the space of available policies. If models are initially trained from an inadequate set of measurements inputs, such that their fitness is insufficient to make reasonably accurate predictions of the measures for an iteration, Artemis will place the deviating regions into a training mode again to gather data on additional policies, so that future models for that region, within the run, will be more robustly informed. Programs with iterative algorithms should typically invoke optimization every time step of execution. When the user invokes the API, Artemis performs the following steps:

1. For every instrumented region it goes through the measurement records and finds the policy with the fastest measured execution for each feature set

to enunciate the optimal pairs of each unique (feature set, policy) combination for this region; 2. In case of multi-process execution, Artemis communicates per-process best policy data between all executing processes to build a unified pool of these pairs and implement *collective training*, 3. From those feature set and policy pairs, it creates the training data to feed to the classification ML model, where the feature set is the feature input to the model and policy is the response; 4. Artemis feeds those data to train the ML model and derive an optimizing policy classifier for each region, that takes as input a feature set and produces as output the optimized selection policy.

When later executions of the instrumented regions query Artemis for the policy to execute, the trained model provides the optimizing policy index. Note that even after training an optimized policy selection model, Artemis continues to collect execution time data for optimized regions to monitor execution and trigger re-training, which we discuss next.

## 3.3 Validation and Retraining

Artemis includes a *regression* model to trigger re-training, anticipating that time-dependent or data-dependent behavior may change the execution profiles of regions, thus rendering previous optimizing models sub-optimal. Specifically, Artemis creates a regression model to predict execution time given the measurement records. The input features to train this regression model are the features set by instrumentation, including the policy selection, and the response outputs are the measured execution times.

At every invocation of the optimization API call by the user, Artemis compares the measured execution time per region, feature set, and policy to the predicted execution time provided by the regression model. When the measured time exceeds the predicted time over a threshold, Artemis discards the optimizing model and reverts the region to a training regime, trying out different execution policies on region execution to collect new data for training an optimized model. On a later invocation of the optimization API call, Artemis creates the new optimizing classification model and the new regression model for a new cycle of optimization and monitoring.

## 3.4 Extending RAJA OpenMP Execution

The RAJA [8] programming model was extended to enable Artemis optimization by defining an auto-tuned execution policy for parallel loop programming patterns implemented with OpenMP. Interestingly, much of region instrumentation is hidden by the end-user of RAJA since instrumentation happens inside the RAJA header library. The only refactoring required for a RAJA program is to make on-demand calls to the optimization API of Artemis and use the Artemis-recommended execution policy when defining parallel kernels through the RAJA templated API.

Specifically, we create an Artemis tuning policy for the `forall` programming pattern, which defines a parallel loop region, and for the `Collapse` kernel pattern, which collapses 2-level and 3-level nested to a single parallel loop, fusing the nested iteration spaces. For this implementation, we choose the `forall` and `Collapse` patterns since they are frequently used in applications. Artemis can integrate with other parallel patterns of RAJA, such as scans, OpenMP offloading, and CUDA, which is work-in-progress. The Artemis policy used in our evaluation framework tunes execution by choosing between two policies: either OpenMP or sequential. The choice for those two policies is motivated by prior work [9] concluding that varying additional OpenMP parameters (number of threads, loop scheduling policy) results in sub-optimal tuning. Nevertheless, Artemis is general to tune for additional OpenMP parameters, which can be abstracted as different execution policies to input to the Artemis API. Artemis instrumentation is within the implementation of those patterns, in the RAJA header library.

```
template <typename Iterable, typename Func>
RAJA_INLINE void forall_impl (artemis_exec &,
                              Iterable    &&iter,
                              Func        &&loop_body) {
static Artemis::Region *region = nullptr;
if (region == nullptr)
  region = Artemis::create_region(num_policies=2);
region->begin({ distance(begin(iter), end(iter)) });
int policy = region->getPolicyIndex();
switch(policy) {
case 0: {
  #pragma omp parallel
  { RAJA_EXTRACT_BED_IT(iter);
    #pragma omp for
    for (decltype(distance_it) i = 0; i < distance_it; ++i)
      loop_body(begin_it[i]);
  } } break;
case 1: {
    RAJA_EXTRACT_BED_IT(iter);
    for (decltype(distance_it) i = 0; i < distance_it; ++i)
      loop_body(begin_it[i]);
  } break; };
region->end();
}
```

**Fig. 1.** Using Artemis in the RAJA forall execution pattern.

Listing 1 shows a code excerpt for the instrumentation of the `forall` implementation with Artemis, redacting implementation details for RAJA closure privatization, for brevity of presentation. Note, the code for the `Collapse` kernel is similar. The `forall` implementation instruments the region execution with a call to `region->begin()` providing the number of iterations as the single feature in the feature set. For the `Collapse` implementation, the feature set consists of the iterations of all loop levels, creating a vector of features. Next, the implementation calls `region->getPolicyIndex()` which returns an index selecting the execution policy variant; 0 indicates executing with OpenMP and 1 indicates executing the region sequentially. This policy index is the input to the following `switch-case` statement that selects the execution variant. Lasty, there is a call to `region->end()` to marks the end of region execution.

This pattern of API use is general, and serves as a model for other interfaces and ports of Artemis, such as it's integration with the tuning API of the Kokkos portability framework.

## 3.5 Enhancing Kokkos CUDA Execution

Besides RAJA OpenMP execution, we integrate Artemis to tune CUDA kernel execution within Kokkos [14]. Specifically, our experiment tuned parameters for the execution of an SpMV kernel computation in CUDA, including the *team size*, which is the outer level of parallelism of thread blocks, the *vector size*, which is the inner level of parallelism of numbers of threads and the *number of rows* of computation assigned to each thread.

## 3.6 Training Measurement

Initially, when Artemis first encounters an instrumented region, it deploys a *round-robin* strategy to collect training data. This strategy cycles through the set of provided policies, which contains the OpenMP execution policy and the sequential policy in our RAJA implementation, or policies representing combinations of the various kernel launch parameters in the Kokkos integration. When searching, Artemis returns a policy index to explore a particular execution variant. In our implementation, round-robin advances the policy selection index for each region and each set of unique features independently. While searching the space of available policies, the Artemis runtime library records the unique feature set and the measured execution time for each instrumented region.

When Artemis is being used in an MPI application, it is capable of *collective training*, whereby training datasets across the processes are analyzed together.

At the end of an application step, every process issues a collective *allgather* operation to share their training datasets and gather the training datasets of every other process. Each process combines them to create a unified training dataset per region, informed by the rank-offset parallel round-robin searches, to find the best explored policy that minimizes execution time across both the local and peer training data.

## 3.7 Training Model Analysis and Optimization

Artemis processes the metrics gathered during training to construct the matrix of features to use in model construction. This includes the feature set, the performance responses, and the optimal policies. A Random Forest Classifier (RFC) model is trained per region, implemented using the OpenCV machine learning library. Artemis evaluates this RFC model in later invocations of `region->getPolicyIndex()` of a trained region, to return the optimized execution policy using as input the feature set provided in the arguments of the `region->begin(` *`features`* `)` call. We choose RFC modeling because it has fast evaluation times of $\mathcal{O}(m \log n)$ complexity for $m$ decision trees of $n$ depth in the forest. Fast evaluation is important for reducing the overhead during execution since `region->getPolicyIndex()` is called with every region's execution. For experimentation, we set the depth to 2 levels and the forest size to 10 trees, which has shown to be effective for optimization.

Artemis uses the same measurement data to train a per-region Random Forest Regression (RFR) model that predicts expected execution time. Artemis uses this regression model to detect time-dependent or data-dependent divergence in the execution of a region that invalidates a previously trained RFC optimizing model, indicating that re-training is needed. In the implementation, RFR models train with regression accuracy of $1e-6$, hence micro-second resolution for predicting time, and implement a forest size of 50 trees. RFR evaluation is off the critical path, hence affords the largest forest size, since it is called only on invocations of `Artemis::processMeasurements()`. For time regression analysis, Artemis compares the profiled execution time with the predicted one for all the region's feature sets. If the measured time for a feature set is greater than the predicted one given a threshold, then the model is considered *diverging*. This threshold limits re-trains due to transient perturbations when measuring execution time. We have experimentally found that this threshold value of $2\times$ filters out needless re-trains for the applications under test. Nonetheless, the threshold value is configurable and also re-training can be turned completely off, through environment variables. If the execution of an application region is pathological, such that execution time continuously diverges with the same features, then this region is ineligible for tuning and should be omitted or re-training should be turned off. This is a challenging scenario to naively automate, and future work involves exploring strategies to effectively manage regions that do not have stable performance profiles even when features or loop inputs are held constant.

Artemis counts all diverging feature sets in a region. If they are found to be more than a threshold, more than half feature sets in a region for our implementation, Artemis deems the RFC model invalid and sets up the round-robin search strategy to re-train an optimized model for that region.

Artemis is generalized to support *heterogeneous execution*, where an application deploys to a cluster of heterogeneous machines, or for cases where a heterogeneous workload is specified on the same regions. Differences in machine architectures can be captured as a feature that describes the machine type, e.g., CPU or GPU micro-architecture. Differences in a heterogeneous workload, for the same code region, can be captured as a feature describing the condition causing it, e.g., the MPI rank or an application-designated parameter.

## 4 Experimentation Setup

The Artemis framework is intended to target environments where performance portability is important. When evaluating Artemis we want to compare its benefits to standard configurations of application and systems that they run on. On the one hand, Artemis is optimizing an application's execution on a machine from some point of reference. If that starts with an already optimized version, there is little likely to be gained. Thus, choosing a "default" version of the application with standard settings is more appropriate to gauge improvement. On the other hand, Artemis is optimizing an application across machines, where different architecture component (e.g., CPU, memory) could lead to different code variants being selected. The application code needs to be developed in such a way that making selection of those code variants is possible without completely rewriting the application. This is the reason for working with RAJA and Kokkos for the experiments discussed below.

## 4.1 Comparators

The applications used in our study are developed with either RAJA or Kokkos, and we focus our attention on the parallel regions impacted by those portability frameworks. We define the *baseline* in performance comparison to be, for OpenMP, execution with the RAJA OpenMP execution policy using the same thread count for all regions, or in the CUDA case, the expert-tuned and hard-coded settings within the Kokkos Kernels suite. This is the *default mode* of executing these parallel applications. To quantify the instrumentation overhead of Artemis, we create a version of Artemis with this baseline that always selects the fixed default policy when guiding execution of a region, but does not perform any of the collection of performance measurements or online training. We call this the *Artemis-OpenMP* or *Artemis-Expert Heuristic* version. Lastly, we denote as *Artemis* the configuration where Artemis dynamically optimizes execution, using online profiling and machine learning for optimized policy selection and regression monitoring.

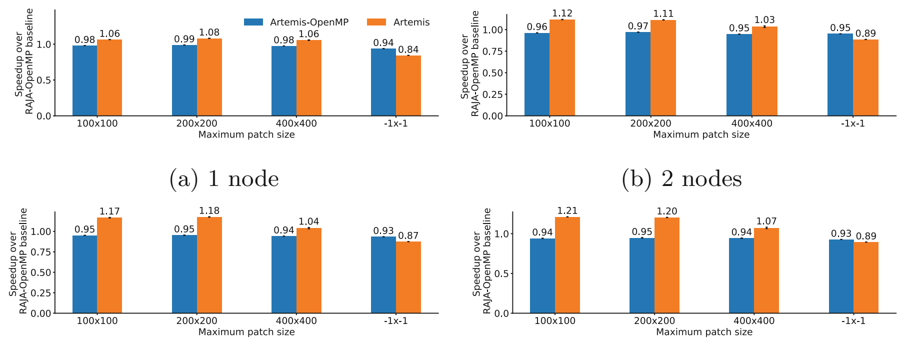
**Table 1.** Applications and their configurations

| Application | Inputs | Nodes |
|---|---|---|
| LULESH | –r 100 –c 1 *or* 2 *or* 4 *or* 8 –i 100 | 1 |
| Cleverleaf | Domain: (500, 500), triple point calculation,<br>4 refinement levels, 25 timesteps,<br>max patch size: $100 \times 100$ *or* $200 \times 200$,<br>$400 \times 400$ *or* $-1 \times -1$(no limit) | 1, 2, 4, 8 |
| Kokkos Kernels SpMV | Domain: 100 M to 600 M non-zero values<br>team size: 1–1024, vector size: 1–32<br>rows per thread: 1–4096 | 1 |

## 4.2 Applications

We chose three HPC proxy-applications to perform our experiments: LULESH [1, 20] and Cleverleaf [6, 10] for OpenMP, and Kokkos Kernels SpMV [24] for CUDA.

Table 1 shows details of the application inputs used and execution configurations. LULESH is configurable to create regions of different computational



(a) 1 node

(b) 2 nodes

(c) 4 nodes

(d) 8 nodes

**Fig. 2.** Cleverleaf, speedup of Artemis-OpenMP and Artemis over the baseline.

cost, to mimic multi-material calculation. Cleverleaf uses adaptive mesh refinement to create a range of problem subdomains, called patches, with varying computational cost. Thus, both data-dependent and input-dependent settings can create regions of different computation. Kokkos Kernels SpMV computes a sparse matrix vector product for very large matrices, allowing for a configurable count of non-zero values.

In the OpenMP codes, Artemis dynamically optimizes each parallel region by selecting OpenMP execution policies only when there is enough work to justify the overhead of parallel execution, otherwise it will elect for sequential execution. LULESH inputs create heterogeneous computation by using a large count of regions (100) that emulate different materials, changing the computational cost of various region subsets by 1, 2, 4, or 8 times the base cost – LULESH adjusts the cost of 45% of the regions to be this multiple and 5% of regions to be 10× this multiple. For Cleverleaf, heterogeneous computation is created by changing the maximum patch size permitted during refinement, ranging from from $100 \times 100$, $200 \times 200$, $400 \times 400$, up to an unlimited maximum by selecting $-1 \times -1$. The RAJA LULESH implementation does not support distributed execution with MPI, thus our experiments are single node. Cleverleaf provides support for MPI execution, so we performed experiments on multiple nodes to show Artemis's response to Cleverleaf's strong scaling properties. Kokkos Kernels SpMV experiments used Artemis to explore and select policies representing combinations of Kokkos settings and CUDA kernel launch parameters, across a variety of problem sizes.

## 4.3 Hardware and Software Platforms

Experiments were run on nodes featuring dual-socket Intel Xeon E5-2695v4 processors for 36 cores and 128 GB of RAM per node and the TOSS3 software stack. We compiled applications and Artemis using GCC version 8.1.0 and MVAPICH2 version 2.3 for MPI support. Artemis used the OpenCV machine learning library version 4.3.0. For Kokkos CUDA we targeted the NVIDIA V100 (Volta) on an IBM Power9 architecture, using CUDA version 10.

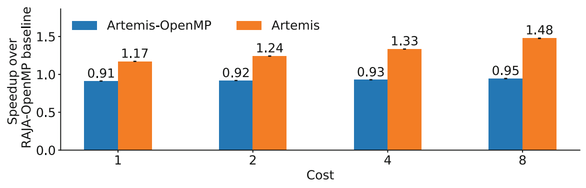


**Fig. 3.** LULESH, speedup over the baseline of RAJA-OpenMP execution.

### 4.4 Statistical Evaluation

For each OpenMP proxy application and configuration we performed 10 independent measurements. Unless otherwise noted, measurement counts the total application execution time end-to-end. Confidence intervals shown correspond to a 95% confidence level, calculated using Bootstrapping to avoid assumptions on the sampled population's distribution.

## 5 Evaluation

Here we provide results and detailed analysis of tuning for OpenMP with RAJA, as well as summary results from applying Artemis to tune Kokkos settings and CUDA kernel launch parameters.

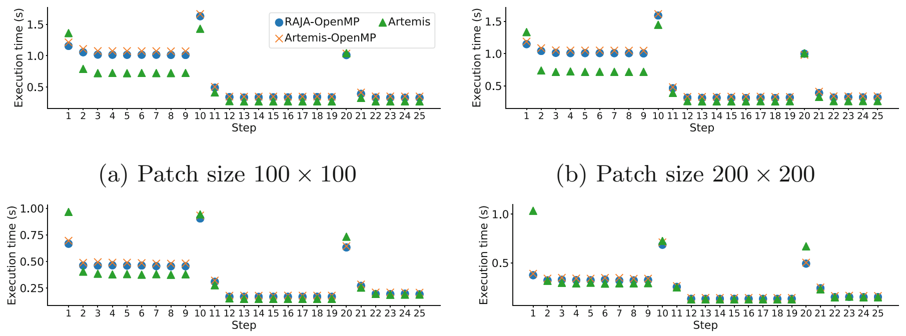
For evaluating the performance of Artemis with OpenMP, we compute the speedup over the baseline of RAJA-OpenMP execution for both Artemis-OpenMP, which always selects OpenMP execution, and the optimizing Artemis, which dynamically chooses between OpenMP or sequential execution for a region, using the machine learning methods we described. Artemis-OpenMP exposes the instrumentation overhead of Artemis, hence the expected slowdown compared to non-instrumented RAJA-OpenMP execution. Figure 2 shows results for Cleverleaf, and Fig. 3 shows results for LULESH. Values on bars show the mean speedup (or slowdown) compared to RAJA-OpenMP execution.

### 5.1 Instrumentation Overhead

Observing the slowdown of Artemis-OpenMP, the overhead of instrumentation is modest, cumulatively less than 9% across both applications and tested configurations of input and node numbers. This shows that Artemis does not overburden execution and given tuning opportunities, it should recuperate the overhead and provide speedup over non-instrumented RAJA-OpenMP execution.

### 5.2 Model Training and Evaluation Overhead

The average training time for LULESH is 310 ms, while for Cleverleaf is 150 ms, which is minimal contrasted with the timescale of execution of regions, as we show in later measurements, so Artemis recovers this overhead, effectively



(a) Patch size 100 × 100

(b) Patch size 200 × 200

(c) Patch size 400 × 400

(d) Patch size −1 × −1

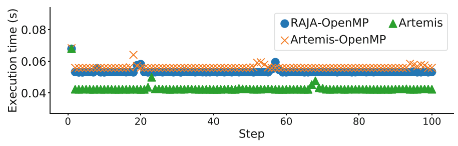
**Fig. 4.** Execution time per timestep for Cleverleaf on 8 nodes, varying the maximum patch size. Regridding operation performed after every 10 steps.

tuning and speeding up execution. Moreover, model training (or re-training) is infrequently done as trained models persist during execution. By contrast, model evaluation happens at every execution of a tunable region. Its overhead depends on the forest size and tree depth of the trees in the evaluated forest. Given the limits in forest size (10) and tree depth (2) set in our implementation, see Sect. 3, we measure the time overhead for evaluating the maximum possible forest configuration to be less than 10 microseconds.

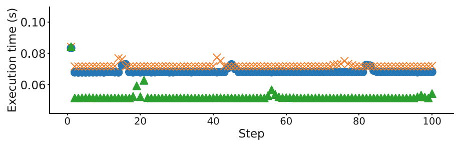
## 5.3 Speedup on Cleverleaf

For Cleverleaf, varying the maximum patch size changes the number and size of computational regions. A smaller size means more regions, hence more parallelism, but also finer-grain decomposition of the computation domain. So, there is greater disparity between regions that lack enough work, hence sequential policy is fastest, and regions with enough parallel work, for which OpenMP execution is fastest. Note, the special value −1 ×−1 means there is no maximum set and Cleverleaf by default prioritizes decomposing in larger regions. Figure 2 shows results for all node configurations, demonstrating that Artemis consistently speeds up execution for the smaller patch sizes of 100×100 and 200×200, no less than 8%, executing with one node, and up to 21%, executing on 8 nodes. For the larger patch size of 400 × 400, execution with Artemis is on par with RAJA-OpenMP, successfully recuperating the overhead with marginal gains, within measurement error. For the unlimited patch size of −1 × −1, Artemis results in a net slowdown, also compared with Artemis-OpenMP, since there is lack of optimization opportunity, and the training and monitoring overhead inflated execution time.

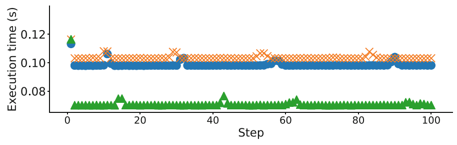
For further analysis, we show results comparing execution times per timestep for different execution modes. Figure 4 shows results when executing with 8 nodes. Results for other node counts are similar, thus we omit them for brevity.



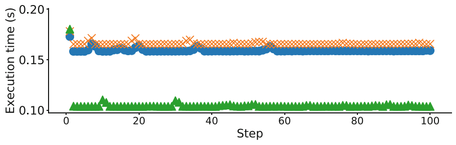
(a) Cost 1



(b) Cost 2



(c) Cost 4



(d) Cost 8

**Fig. 5.** Execution time per timestep for LULESH, showing different execution modes on one node, varying the cost of computational regions.

Note that Cleverleaf performs a *re-gridding* operation [7] every 10 timesteps that re-shuffles domain decomposition to reduce computation error, thus the spikes in execution time in the 10th and 20th timesteps.

Observing results, Artemis inflates execution time for the first timestep across all patch sizes, since this step includes training for bootstrapping tunable regions. For most of the rest of timesteps, Artemis reduces execution time, by as much as 40% for the least patch size of $100 \times 100$, compared to the default execution with RAJA-OpenMP. Artemis tuning potential lessens the larger the patch size, since larger regions favor OpenMP execution. Nevertheless, observing Fig. 4d for the largest patch size selection, Artemis correctly selects OpenMP execution and any performance lost is due to the initial training overhead. Notably, Cleverleaf execution with 8 nodes has second to sub-second timesteps, and Artemis is fast enough to optimize execution even at this short time scale. Expectedly, Artemis-OpenMP has slightly higher execution time per timestep compared to RAJA-OpenMP, reflecting instrumentation overhead as seen by the speedup results.

## 5.4 Effectiveness of Cleverleaf Policy Selection

Cleverleaf instantiates a multitude of regions and each region executes with multiple different feature sets, corresponding to different patch sizes from decomposing the domain and load balancing. So, to highlight Artemis effectiveness we fix the patch size to $100 \times 100$, which presents the most optimization potential, and pick one region to plot the average execution time of each feature set for the top-20 most frequently executed ones, contrasting OpenMP only execution vs. sequential execution vs. Artemis execution with dynamic policy selection. The region comprises of feature sets corresponding to 2d collapsed loops, so there are two values describing (outer,inner) loop iterations. Depending on the feature set size, OpenMP or sequential is the best. For example, feature set (3,201) executes faster with OpenMP and feature set (55, 2) executes faster sequentially. Observing execution times measured for Artemis, policy recommendations converge to the optimal policy for the majority of feature sets for which the performance difference between the sequential and OpenMP policy selection is more than 20%. Artemis selects the optimal policy in 10 of the 15 such regions.

Further, we find positive results for the accuracy of Artemis in selecting optimal policies. For the initial timestep, Artemis has low accuracy, ranging from 10% to 20%, due to training, without any discernible trend among different patch sizes. However, accuracy significantly improves after this initial, training step to a range of 85% to 95%, showing Artemis is effective in selecting the optimal policy most of the time.

## 5.5 Strong Scaling with Different Node Counts

Figs. 2a–d show results for increasing node counts. Following the discussion on smaller patch sizes that present optimization opportunities for Artemis, increasing the number of nodes also boosts the speedup achieved by Artemis. Cleverleaf distributes computational regions among different MPI ranks and executes bulk-synchronous, advancing the simulation time step after all MPI ranks have finished processing. Artemis dynamically optimizes execution per rank, thus it reduces execution time on the critical path, with multiplicative effect on the overall execution.

## 5.6 Speedup on LULESH

Figure 3 shows results for LULESH on a single node due to the limitation of the RAJA version of LULESH supporting only single node execution. For this experiment, the number of regions is kept constant (100) and the cost of computation varies between 1× (default) and 8×, as explained in Sect. 4. Similarly to Cleverleaf, the instrumentation overhead of Artemis, shown by observing the slowdown of Artemis-OpenMP, is within 9% of non-instrumented execution of RAJA-OpenMP.

Regarding speedup of Artemis, it is consistently faster than RAJA-OpenMP. Artemis improves execution time even for the default setting of cost 1× by 16%. Expectedly, increasing the cost creates more computational disparity between LULESH computational regions, thus Artemis achieves higher speedup. For the highest cost value we experiment with, a cost of 8×, Artemis achieves significant speedup of 47% over the RAJA-OpenMP baseline.

For more detailed results, Fig. 5 shows execution time per timestep for all execution modes varying the cost of computational regions. Observations are similar to Cleverleaf, the first timestep under Artemis is slower due to training while the rest of the timesteps execute faster than RAJA-OpenMP. Artemis speeds up the execution of timestep up to 50% compared to RAJA-OpenMP, increasingly so as the cost input increases. Different than Cleverleaf, the resolution of the execution time of LULESH is much more fine-grain, in the range of hundreds of milliseconds. Nonetheless, Artemis effectively optimizes execution even at this time scale, showing that training effectively optimizes policy selection and overcomes any instrumentation overhead.

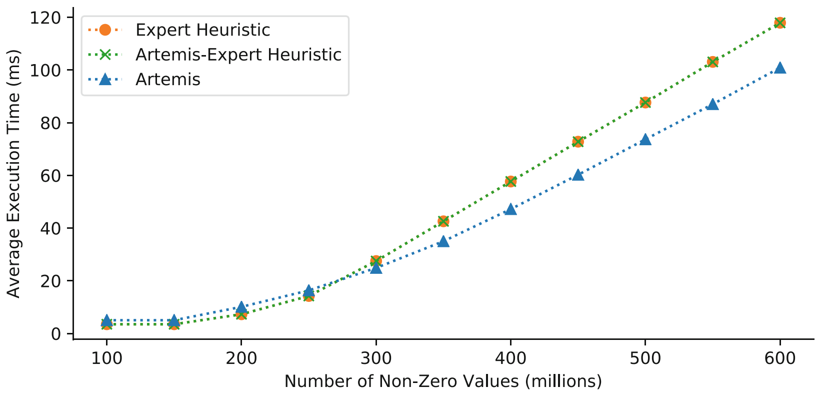


**Fig. 6.** Artemis improves performance of the Kokkos SpMV kernel up to 16.8% compared to the hardcoded expert heuristic.

### 5.7 Speedup on Kokkos Kernels SpMV

Figure 6 shows the results of our integration with Kokkos, tuning the parallel team size, vector size, and number of rows assigned to each thread. The x-axis shows scaling the number of non-zero elements y-axis plots the average execution time for 1500 SpMV kernel invocations. *Expert Heuristic* is the existing, hardcoded tuning strategy set by the expert kernel developer, setting those parameters based on the input data and expert knowledge. This heuristic function settles on 1 row per thread, a vector length of 2, and a team size of 256 for inputs shown. *Artemis-Expert Heuristic* exposes the instrumentation overhead of Artemis, by foregoing tuning, instead executing with the same settings of the expert heuristic. The performance of Artemis-Expert Heuristic is on par with execution of Expert Heuristic without Artemis intervening, thus instrumentation overhead is minimal. *Artemis* shows the performance improvement when tuning is enabled. Kokkos provides a range of 664 selectable policies to Artemis for tuning, with parameters team size ranging from 1–1024, vector size from 1–32, and number of rows per thread from 1–4096. Results show that Artemis succesfully navigates the tuning space, and provides increasingly faster performance as the problem size increases, for a maximum of 16.8% performance improvement on the largest input of 600 M non-zero elements.

## 6 Related Work

Existing tuning frameworks are either application-specific [5,28], programming-model-specific [2,23], hardware-specific [4,15], or feature the need for offline training [9,27], and thus have limited scope. By design, Artemis is a general framework that gives an API to tune at any of those levels, and we show its generality by integrating Artemis with the RAJA programming model, tuning a variety of HPC proxy applications and kernels. The closest to our work is the

Apollo paper by Beckingsale et al. [9], with the important distinction that, rather than exhaustive offline tuning, the Artemis framework performs the search space exploration at runtime.

Empirical techniques directly measure all the possible variants and select the fastest. Established projects like the ATLAS [4,29] and FFTW [15] libraries apply this technique with great success, but it requires the up front cost of finding the best code variant choices for each system. ATF [25,26]presents a generic extensible framework for automated tuning, independent of programming language or domain. Oski [28] performs runtime tuning, optimizing over sparse linear algebra kernels. Orio [17] and OpenTuner [2] are able to facilitate general purpose kernel tuning using empirical techniques to select the best performing configurations for production. ActiveHarmony [18] uses parallel search strategies to perform online tuning, though sweeping large parameter spaces can take significant amounts of time.

Using some form of a model to predict the performance of the code, analytical examples make tuning decisions based on model output. Similarly to Artemis, AutoTuneTMP [23] makes use of C++ template metaprogramming to abstract-away the tuning mechanisms of kernels and facilitate performance portability. It constrains the search space for online training using parameterized kernel definitions. Unlike Artemis's use of RAJA policies that are compiled in alongside the application, AutoTuneTMP uses JIT compilation and dynamic linking at runtime to produce kernel variants, a mechanism which could impose non-trivial overhead in a large large class of HPC codes in production settings. Mira [21] uses static performance analysis to generate and explore performance models offline. Mira's abstract performance models allow it to avoid some of the limitations to offline learning.

A statistical model is built by applying machine learning techniques, and this model is used to make tuning decisions. Sreenivasan et al. [27] demonstrated performance gains using an OpenMP autotuner framework that performs offline tuning using a random forest statistical model of the reduced search space to eliminate exhaustive tuning. HiPerBOt [22] presents an active learning framework that uses Bayesian techniques to maintain optimal outcomes while collapsing the required number of samples for learning.

Other work [11,12,16] has looked into auto-tuning the number of OpenMP threads in multi-program execution. Those approaches look at architectural metrics, such as Instructions-Per-Cycle and memory stalls, to dynamically throttle thread allocation when contention occurs.

## 7 Conclusion and Future Work

We have presented Artemis, a novel framework that optimizes performance by tuning an application's parallel computational regions online. Artemis provides a powerful API to integrate online tuning in existing applications, by defining tunable regions and execution variants. Artemis automatically adapts to data-dependent or time-dependent changes in execution using decision tree and regression models. We integrated Artemis with RAJA and Kokkos and evaluated online tuning performance on HPC proxy applications: Cleverleaf and LULESH, and a CUDA SpMV kernel. Results show that Artemis is up to 47% faster and its operating overhead is minimal.

Future work includes: 1. using Artemis for tuning of additional GPU-offloaded compute kernels with heterogeneous memory hierarchies. 2. tuning additional parallel execution parameters such as loop tiling and nesting. 3. expanding experimentation to large applications by extending the Artemis codebase and integration with RAJA, Kokkos, and lower level parallel programming models, such as OpenMP, CUDA, and HIP.

**Acknowledgment.** This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-809192). Additional support was provided by a LLNL subcontract to the University of Oregon, No. B631536. This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

## References

1. Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Tech. Rep. LLNL-TR-490254, Lawrence Livermore National Laboratory
2. Ansel, J., et al.: Opentuner: an extensible framework for program autotuning. In: Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, pp. 303–316 (2014)
3. Balaprakash, P., Dongarra, J., Gamblin, T., Hall, M., Hollingsworth, J.K., Norris, B., Vuduc, R.: Autotuning in high-performance computing applications. Proc. IEEE **106**(11), 2068–2083 (2018)
4. Baldeschwieler, J.E., Blumofe, R.D., Brewer, E.A.: Atlas: an infrastructure for global computing. In: Proceedings of the 7th Workshop on ACM SIGOPS European Workshop: Systems Support for Worldwide Applications, pp. 165–172 (1996)
5. Bari, M.A.S., Chaimov, N., Malik, A.M., Huck, K.A., Chapman, B., Malony, A.D., Sarood, O.: Arcs: adaptive runtime configuration selection for power-constrained openmp applications. In: 2016 IEEE International Conference on Cluster Computing, pp. 461–470. IEEE (2016)
6. Beckingsale, D.A., Gaudin, W.P., Herdman, J.A., Jarvis, S.A.: Resident block-structured adaptive mesh refinement on thousands of graphics processing units. In: 44th International Conference on Parallel Processing, pp. 61–70 (2015)

7. Beckingsale, D., Gaudin, W., Herdman, A., Jarvis, S.: Resident block-structured adaptive mesh refinement on thousands of graphics processing units. In: 2015 44th International Conference on Parallel Processing, pp. 61–70. IEEE (2015)
8. Beckingsale, D.A., Hornung, R.D., Scogland, T.R.W., Vargas, A.: Performance portable C++ programming with RAJA. In: Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, pp. 455–456 (2019)
9. Beckingsale, D.A., Pearce, O., Laguna, I., Gamblin, T.: Apollo: reusable models for fast, dynamic tuning of input-dependent code. In: 31st IEEE International Parallel & Distributed Processing Symposium, pp. 307–316 (2017)
10. Beckingsale, D.A.: Towards scalable adaptive mesh refinement on future parallel architectures. Ph.D. thesis, University of Warwick (2015)
11. Creech, T., Kotha, A., Barua, R.: Efficient multiprogramming for multicores with scaf. In: 2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 334–345 (2013)
12. Creech, T., Barua, R.: Transparently space sharing a multicore among multiple processes. ACM Trans. Parallel Comput. **3**(3) (Nov 2016). https://doi.org/10.1145/3001910
13. Edwards, H.C., Trott, C.R.: Kokkos: Enabling performance portability across manycore architectures. In: 2013 Extreme Scaling Workshop (xsw 2013), pp. 18–24. IEEE (2013)
14. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: enabling manycore performance portability through polymorphic memory access patterns. J. Parallel Distrib. Comput. **74**(12), 3202–3216 (2014)
15. Frigo, M., Johnson, S.G.: FFTW an adaptive software architecture for the FFT. In: Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP 1998) (Cat. No. 98CH36181). vol. 3, pp. 1381–1384. IEEE (1998)
16. Georgakoudis, G., Vandierendonck, H., Thoman, P., Supinski, B.R.D., Fahringer, T., Nikolopoulos, D.S.: Scalo: scalability-aware parallelism orchestration for multi-threaded workloads. ACM Trans. Archit. Code Optim. **14**(4) (Dec 2017). https://doi.org/10.1145/3158643
17. Hartono, A., Norris, B., Sadayappan, P.: Annotation-based empirical performance tuning using orio. In: 2009 IEEE International Symposium on Parallel & Distributed Processing, pp. 1–11. IEEE (2009)
18. Hollingsworth, J., Tiwari, A.: End-to-end auto-tuning with active harmony. In: Performance Tuning of Scientific Applications, pp. 217–238, CRC Press, Boca Raton (2010)
19. Hornung, R.D., Keasler, J.A.: The RAJA Portability Layer: Overview and Status. Tech. Rep, Lawrence Livermore National Lab (2014)
20. Karlin, I., Keasler, J.A., Neely, R.: Lulesh 2.0 updates and changes. Tech. Rep. LLNL-TR-641973, Lawrence Livermore National Laboratory (August 2013)
21. Meng, K., Norris, B.: Mira: a framework for static performance analysis. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER), pp. 103–113. IEEE (2017)
22. Menon, H., Bhatele, A., Gamblin, T.: Auto-tuning parameter choices in HPC applications using Bayesian optimization. In: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS) (2020)
23. Pfander, D., Brunn, M., Pflüger, D.: AutoTuneTmp: auto-tuning in C++ with runtime template metaprogramming. In: 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 1123–1132. IEEE (2018)

24. Rajamanickam, S.: Kokkos kernels: Performance portable kernels for sparse/dense linear algebra graph and machine learning kernels. Tech. Rep., Sandia National Lab. (SNL-NM), Albuquerque, NM (United States) (2020)
25. Rasch, A., Gorlatch, S.: ATW a generic directive-based auto-tuning framework. Concurr. Comput. Prac. Exp. **31**, e4423 (2019)
26. Rasch, A., Haidl, M., Gorlatch, S.: AFT: a generic auto-tuning framework. In: 2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pp. 64–71. IEEE (2017)
27. Sreenivasan, V., Javali, R., Hall, M., Balaprakash, P., Scogland, T.R.W., de Supinski, B.R.: A framework for enabling openMP autotuning. In: Fan, X., de Supinski, B.R., Sinnen, O., Giacaman, N. (eds.) IWOMP 2019. LNCS, vol. 11718, pp. 50–60. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-28596-8_4
28. Vuduc, R., Demmel, J.W., Yelick, K.A.: OSKI: a library of automatically tuned sparse matrix kernels. J. Phys. Conf. Ser. **16**, 521 (2005)
29. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the atlas project. Parallel Comput. **27**(1–2), 3–35 (2001)