Machine-Learning-Based Load Balancing for Community Ice Code Component in CESM

Prasanna Balaprakash[1,2(✉)], Yuri Alexeev[2], Sheri A. Mickelson[1], Sven Leyffer[1], Robert Jacob[1], and Anthony Craig[3]

[1] Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA
pbalapra@mcs.anl.gov
[2] Leadership Computing Facility, Argonne National Laboratory, Argonne, IL, USA
[3] UCAR, Seattle, WA, USA

Abstract. Load balancing scientific codes on massively parallel architectures is becoming an increasingly challenging task. In this paper, we focus on the Community Earth System Model, a widely used climate modeling code. It comprises six components each of which exhibits different scalability patterns. Previously, an analytical performance model has been used to find optimal load-balancing parameter configurations for each component. Nevertheless, for the Community Ice Code component, the analytical performance model is too restrictive to capture its scalability patterns. We therefore developed machine-learning-based load-balancing algorithm. It involves fitting a surrogate model to a small number of load-balancing configurations and their corresponding runtimes. This model is then used to find high-quality parameter configurations. Compared with the current practice of expert-knowledge-based enumeration over feasible configurations, the machine-learning-based load-balancing algorithm requires six times fewer evaluations to find the optimal configuration.

1 Introduction

The Community Earth System Model (CESM) is one of the most widely used climate models in the world. Results from this model are a major part of the Intergovernmental Panel on Climate Change assessment reports [1]. CESM1.1.1 consists of six model components—atmosphere, ocean, sea-ice (CICE), land, river, and land-ice models—that communicate through a coupler. Each of the CESM model components has different scalability patterns and performance

The submitted manuscript has been created by the UChicago Argonne, LLC, Operator of Argonne National Laboratory (Argonne) under Contracts No. DE-AC02-06CH11357 and DE-FG02-05ER25694 with the U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government. The NCAR is sponsored by the National Science Foundation.

M. Daydé et al. (Eds.): VECPAR 2014, LNCS 8969, pp. 79–91, 2015.
DOI: 10.1007/978-3-319-17353-5_7

characteristics. In this paper, we focus on static load-balancing of computation, which is usually simple to implement with negligible overhead, making it suitable for "fine-grained" parallelism consisting of many small tasks. Previously, the load-balancing problem has been formulated as a mixed-integer nonlinear optimization problem and solved by using the optimization solver MINOTAUR [2]. This is a heuristic method that consists of gathering benchmarking data, calibrating a performance model using the data, and making decisions about optimal allocation by using the model. The performance model predicts the execution time of the program running in parallel as a function of problem size and the number of processors employed. Nonetheless, several challenges in intramodel load balancing for the CICE computations occur only where sea ice is located and the sun is shining. This restriction presents a load-balance problem because processors are allocated across the entire Earth grid and several locations on the grid that do not have any sea ice [3]. The poor fit of the CICE results in inefficient processor allocations to all components—incorrect allocation of the CICE affects all other allocations because the total number of processors available to components is a fixed number. This is the primary motivation for us to develop sophisticated approaches for load balancing the CICE component of the CESM.

Recently, machine-learning methods [4] have received considerable attention for tuning performance of large scientific codes and kernels on high-performance computing systems. In particular, supervised machine-learning tries to learn the relationship between the input and the output of an unknown response function by fitting a model from few representative examples. When the model is accurate enough, it can predict the output at new unseen inputs, which provides numerous benefits, in particular when the evaluation becomes expensive.

In this paper, we present a machine-learning-based approach for static load-balancing problems, and we apply it to find high-quality parameter configurations for load balancing the CICE component of the CESM on IBM Blue Gene/P (BG/P). The novelty of the proposed algorithm consists of iteratively using the model to choose configurations with shorter predicted runtime for evaluation on the target architecture. We emphasize, however, that the algorithm is general and not specific to the CESM and/or BG/P. The paper is structured as follows: (1) a machine-learning-based algorithm for static load-balancing problem, (2) deployment of a machine-learning method as a diagnostic tool for analyzing the sensitivity of the load-balancing parameters on the execution time, (3) empirical analysis of several state-of-the-art machine-learning methods for modeling the relationship between the load-balancing parameters and their corresponding execution time, and (4) 6x savings in core-hour usage for load balancing the CICE component of the CESM on BG/P.

2 The CICE Component on BG/P

For the CICE component, we need to find the optimal load-balancing parameter configuration x^* with the shortest the runtime (f^*) for task counts $\in \{80, 128, 160, 256, 320, 376, 512, 640, 800, 1024\}$. The task count corresponds to number of

Table 1. Decomposition strategies and their corresponding `block.x` and `block.y` sizes

`decomp.set`	`decomp.typ`	`block.x`	`block.y`
null	blkrobin, blkcart roundrobin, spacecurve	1, 2, 4, 8	24, 48, 96, 192, 3840
slenderX1 slenderX2	cartesian	4, 5, 8, 10	4 6 8 12

MPI tasks; the number of OpenMP threads per MPI task is set to four because of memory restrictions on BG/P. The CICE component comprises six parameters. Three integer parameters, namely, maximum number of CICE blocks, `max.block`; the size of a CICE block in the first and second horizontal dimensions `block.x` and `block.y` respectively. Two categorical parameters that determine the decomposition strategy, `decomp.typ` ∈ {blkrobin, roundrobin, spacecurve, blkcart, cartesian} and `decomp.set` ∈ {null, slenderX1, slenderX1}. A binary parameter `mask.h` ∈ {0,1} that specifies to run the code with or without halo.

The constraints that define a feasible set $\mathcal{D}$ of configurations are as follows. The parameter `max.block` ∈ {1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 20, 24, 26, 30, 32, 40, 48, 64} is determined by computing (CICE_X_Grid_Size × CICE_Y_Grid_Size) / (`block.x` × `block.y` × task count). The feasible values for `decomp.set`, `decomp.typ`, `block.x`, and `block.y` are constrained as shown in Table 1. The decomposition strategies have different rules, and not all combinations of block sizes are possible. The blkcart method must have a multiple of four blocks per compute core. The spacecurve method must have 2, 3, and 5 only in `max.block`. The slenderX1 method requires that the `block.x` multiplied by the task count divide evenly into the CICE X grid size. The value of `block.y` must also be divisible by the CICE Y grid size. The slenderX2 method requires that the `block.x` multiplied by the task count be divisible by the CICE X grid size multiplied by 2. The decomposition also requires that the `block.y` multiplied by 2 divide evenly into the CICE Y grid size.

3 Machine-Learning Based Load-Balancing Algorithm

Given a set of training data $\{(x_1, y_1)), \ldots, (x_l, y_l))\}$, where $x_i \in \mathcal{D}$ and $y_i = f(x_i) \in \mathbb{R}$ are the load-balancing parameter configuration and its corresponding runtime, respectively, the supervised machine-learning approach includes finding a surrogate function h for the expensive f such that the difference between $f(x_i)$ and $h(x_i)$ is minimal for $\forall i \in \{1, \ldots, l\}$. The function h, which is an empirical performance model, can be used to predict the runtimes of unevaluated $x' \in \mathcal{D}$. The key idea behind the machine-learning-based load-balancing algorithm is iteratively using the model to choose configurations with shorter predicted runtime for evaluation and retrain the model with the evaluated configurations.

The pseudo-code is shown in Algorithm 1. The symbols ∪ and − denote set union and difference operators, respectively. Given a task count c, a pool $\mathcal{X}_p$ of unevaluated configurations of task count c, the maximum number $n_{\max}$ of

Algorithm 1. Pseudo-code for the machine-learning-based load-balancing algorithm

Input: task count c, configuration pool $\mathcal{X}_p$ of task count c, max evaluations $n_{\max}$, initial sample size n_s

$\mathcal{X}_{\text{out}} \leftarrow$ sample $\min\{n_s, n_{\max}\}$ distinct configurations from $\mathcal{X}_p$
$\mathcal{Y}_{\text{out}} \leftarrow$ `Evaluate_Parallel`$(c, \mathcal{X}_{\text{out}})$
$\mathcal{M} \leftarrow$ `fit`$(\mathcal{X}_{\text{out}}, \mathcal{Y}_{\text{out}})$
$\mathcal{X}_p \leftarrow \mathcal{X}_p - \mathcal{X}_{\text{out}}$
for $i \leftarrow n_s + 1$ to $n_{\max}$ **do**
    $\mathcal{Y}_p \leftarrow$ `predict`$(\mathcal{M}, \mathcal{X}_p)$
    $x_i \leftarrow x \in \mathcal{X}_p$ with the shortest runtime in $\mathcal{Y}_p$
    $y_i \leftarrow$ `Evaluate`(c, x_i)
    `retrain` $\mathcal{M}$ with (x_i, y_i)
    $\mathcal{X}_{\text{out}} \leftarrow \mathcal{X}_{\text{out}} \cup x_i$; $\mathcal{Y}_{\text{out}} \leftarrow \mathcal{Y}_{\text{out}} \cup y_i$
    $\mathcal{X}_p \leftarrow \mathcal{X}_p - x_i$
end for

Output: $x \in \mathcal{X}_{\text{out}}$ with the shortest runtime in $\mathcal{Y}_{\text{out}}$

allowed evaluations, and initial sample size n_s, the algorithm proceeds in two phases: parallel initialization phase and sequential iterative phase. In the initialization phase, the algorithm first samples n_s configurations at random and evaluates them in parallel to obtain their corresponding runtimes. A supervised learning method uses these points as a training set to build a predictive model. The sequential iterative phase consists of predicting the runtimes of all remaining unevaluated configurations using the model, evaluating the configuration with shortest predicted runtime, and retraining the model with the evaluation results. Without loss of generality, Algorithm 1 can be run in parallel for each task count $c \in C$. Because the best supervised learning algorithms depends on the relationship between the input and output, we test four state-of-the-art machine-learning algorithms as candidates for Algorithm 1: random forest (RF) [5], support vector machines (SVM) [6], Gaussian process regression (GP) [7], and neural networks (NN) [8].

RF belongs to the class of recursive partitioning methods [9]. They are widely used tools for predictive modeling in many scientific fields. These methods recursively partition the multi-dimensional input space $\mathcal{D}'$ of training points into a number of hyper rectangles. The partition is done in such a way that input configurations with similar outputs fall within the same rectangle. The partition gives rise to a set of if-else rules that can be represented as a decision tree. For each hyper rectangle, a constant value is assigned—typically this is an average over the output values that fall within the given hyper rectangle. An example tree which is obtained on the CICE component data is shown in Fig. 1. Given an unseen input $x^* \in \mathcal{D}^* \subset \mathcal{D}$, the algorithm uses the if-else rule to find the leaf and returns the corresponding constant value as the predicted value. RF uses a collection of regression trees, where each tree is obtained by the principle of recursive

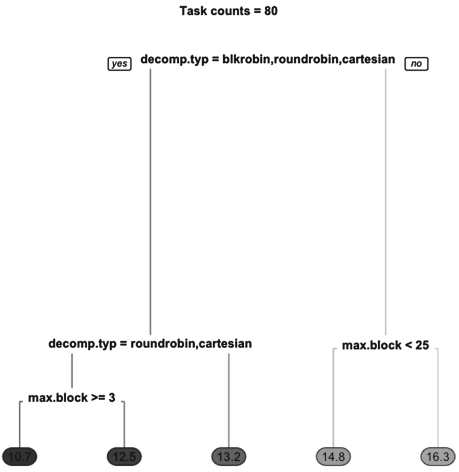

Fig. 1. Illustration of a decision tree obtained via recursive partitioning on CICE component data for the task count 80.

partitioning. For each tree generation, the algorithm takes a subsample of random points from the given training set. The subsample is either a bootstrap sample of the same size drawn with replacement or a subset of smaller size, drawn without replacement. Due to the randomness in the sampling, each subsample differs from each other. Given that each individual tree is build on the subsample, it can differ significantly from other trees. For a given x^*, each tree can make a prediction with respect to its own subsample. The power of RF comes from the aggregation of predicted output values from different trees and the natural way of handling the categorical parameters. Consequently, it can deal with large dimensional inputs even in the presence of complex interactions and non-linearity.

SVM for nonlinear regression consists of mapping the given $\mathcal{D}'$ of the training points into a high dimensional feature space and performing linear regression in the feature space:

$$g(\mathcal{D}') = \langle w \cdot \psi(\mathcal{D}') \rangle + b, \tag{1}$$

where $\psi : \mathbb{R}^n \rightarrow \mathcal{F}$ being the nonlinear transformation, b being the bias term, and $w \in \mathcal{F}$. Finding $g(\mathcal{D}')$ consists in specifying a loss function that need to be optimized and a kernel function $k(\cdot)$ for nonlinearity transformation ψ. For the former, we use ϵ intensive-loss function in which zero penalty is added to the loss function when predicted value of a training point is within ϵ from its observed value. For the latter, we use the widely used Gaussian radial basis function kernel. Now, Eq. 1 can be written as follows:

$$g(\mathcal{D}') = \sum_{i=1}^{l} \alpha_i \times [k(x_i, x_1), \ldots, k(x_i, x_l)] + b, \tag{2}$$

$$k(x_i, x_j) = \exp\left(-\frac{||x_i - x_j||_2^2}{2\sigma^2}\right), \tag{3}$$

where coefficients α_i can be found by solving ϵ intensive-loss function, $||\mathbf{x} - \mathbf{x}'||_2^2$ is squared Euclidean distance that decreases with an increase in dissimilarity between x_i and x_j, and σ is a parameter of the kernel.

GP follows a probabilistic approach for regression. Given a training data of l points, GP assumes that $\mathbf{Y} = [y_1, \ldots, y_l]$ as a sample from a l-variate Gaussian distribution. For an unseen input x^*, the probability $p(y^*|\mathbf{Y})$ follows the Gaussian distribution $\mathcal{N}$ with a user defined kernel function $k(\cdot)$:

$$y^*|\mathbf{Y} \sim \mathcal{N}(\mathbf{K}_*\mathbf{K}^{-1}\mathbf{Y}, \mathbf{K}_{**} - \mathbf{K}_*\mathbf{K}^{-1}\mathbf{K}_*^{\mathbf{T}}), \tag{4}$$

where

$$\mathbf{K} = \begin{bmatrix} k(x_1, x_1) & \cdots & k(x_1, x_n) \\ \vdots & \ddots & \vdots \\ k(x_n, x_1) & \cdots & k(x_n, x_n), \end{bmatrix} \tag{5}$$

$$\mathbf{K}_* = [k(x_*, x_1), \ldots, k(x_*, x_n)],$$
$$\mathbf{K}_{**} = k(x_*, x_*).$$

Note that $\mathbf{T}$ represents matrix transpose operation. For $k(\cdot)$, we use the Gaussian radial basis function as in Eq. 3. The predicted value $\hat{y}_*$ and variance $var(y^*)$ of y^* are given by the parameters of $\mathcal{N}$:

$$\begin{aligned} \hat{y}^* &= \mathbf{K}_*\mathbf{K}^{-1}\mathbf{Y}, \\ var(y^*) &= \mathbf{K}_{**} - \mathbf{K}_*\mathbf{K}^{-1}\mathbf{K}_*^{\mathbf{T}}. \end{aligned} \tag{6}$$

NN is a classical and one of most widely used supervised learning approaches. We focus on a single-hidden-layer neural network, an effective variant that comprises one input layer, one hidden layers, and one output layer. The nonlinear regression performed by NN can be written as follows:

$$\mathbf{Y} = h(\mathcal{D}') = \mathbf{B}\boldsymbol{\varphi}(\mathbf{A}\mathcal{D}' + \mathbf{a}) + \mathbf{b},$$

where $\mathbf{A}$ and is the matrix of weights and bias vector for the first layer (between input and hidden layer) and $\mathbf{B}$ and $\mathbf{b}$ are the weight matrix and the bias vector of the second layer (between hidden and output layer). The function $\boldsymbol{\varphi}$ denotes an element wise nonlinearity. The training in neural network consists in adapting all the weights and biases $\mathbf{A}$, $\mathbf{B}$, $\mathbf{a}$, and $\mathbf{b}$ to their optimal values for the training set $\{(x_1, y_1)), \ldots, (x_l, y_l))\}$. The optimization problem consists in minimizing the squared reconstruction error $\sum_{i=1}^{l} ||h(x_i) - y_i||^2$ and it can be solved effectively with back-propagation algorithm.

4 Experimental Results

We evaluated the effectiveness of the proposed load-balancing algorithm with the four machine-learning methods. In addition, we include two approaches in the

comparison: Expert-knowledge-based enumeration (EE) and random search (RS). EE is the current practice for finding the optimal load-balancing configuration for the CICE component of the CESM. In addition to the application-specific constraints, expert knowledge of the code and the architecture were used to prune the feasible set of configurations $\mathcal{D}$ for the CICE component. As a result, for each task count c, there are 50 to 60 ($|\mathcal{D}_c|$) feasible configurations; in total, for all the 10 task counts, there are $|\mathcal{D}| = 653$ parameter configurations. This method evaluates all 653 parameter configurations. Moreover, we followed the current practice for defining the runtime $f(x)$ for x: the code was run twice with the same x and the shortest runtime was taken as $f(x)$. In RS, for each task count c, parameter configurations were sampled at random without replacement from $\mathcal{D}_c$ and were evaluated. To minimize the impact of randomness involved in the initialization procedure of Algorithm 1 and in the five approaches, we repeated all of them 10 times, each with a different random seed. Moreover, we stored the runtime of each configuration from EE in a lookup table and reused the results for running all other algorithms. For Algorithm 1, for each task count c, $\mathcal{D}_c$ obtained in the EE approach was given as the configuration pool $\mathcal{X}_p$, and the initial sample size n_s was set to 5. The approaches were implemented and run in the **R** programming language and environment [10] version 2.15.2 using the `nnet` (NN), `kernlab` (SVM, GP), and `randomForest` (RF) packages. The default parameter values were used for each method. Experiments were carried out on Intrepid, a BG/P supercomputer at Argonne.

Sensitivity Analysis: First, we present an empirical analysis to explain why the previously proposed analytical performance model fails to predict the runtime of the CICE component and why distinct models may be constructed for each task count. For this purpose, we used the RF method to analyze the impact of each load-balancing parameter on the resulting runtimes. For the training data, we randomly sampled 50 % of the data (parameter configuration and runtimes) obtained with EE approach. An RF model was fitted on this training set. The mean squared error (MSE) on the original training set is given by $\frac{\sum_{i=1}^{l}(f(x_i)-\hat{f}(x_i))^2}{l}$, where l is the number of training points, and $f(x_i)$ and $\hat{f}(x_i)$ are the original and predicted runtime value of parameter configuration x_i, respectively. In order to assess the impact of a parameter m, the values of m in the training set were randomly permuted. Again, an RF model was fitted on this imputed training set, and the mean squared error was computed. If a parameter m is important, then permuting the values of m should affect the prediction accuracy significantly and eventually increase the mean squared error. The results are shown in Fig. 2. We observe that the trend in the parameter importance is not the same over all the task counts. For task counts up to 320, `decomp.set` and/or `decomp.type` have a strong impact on the runtimes; for large task counts, they become relatively less important—`max.block`, `block.x`, and `block.y` have a strong impact on the runtime. For 1024, only `max.block`, `block.x`, and `block.y` have an impact on the runtime; the other three parameters have negative %IncMSE, suggesting that they do not affect the runtime. In summary, the impact of parameter values on the runtimes and the type of

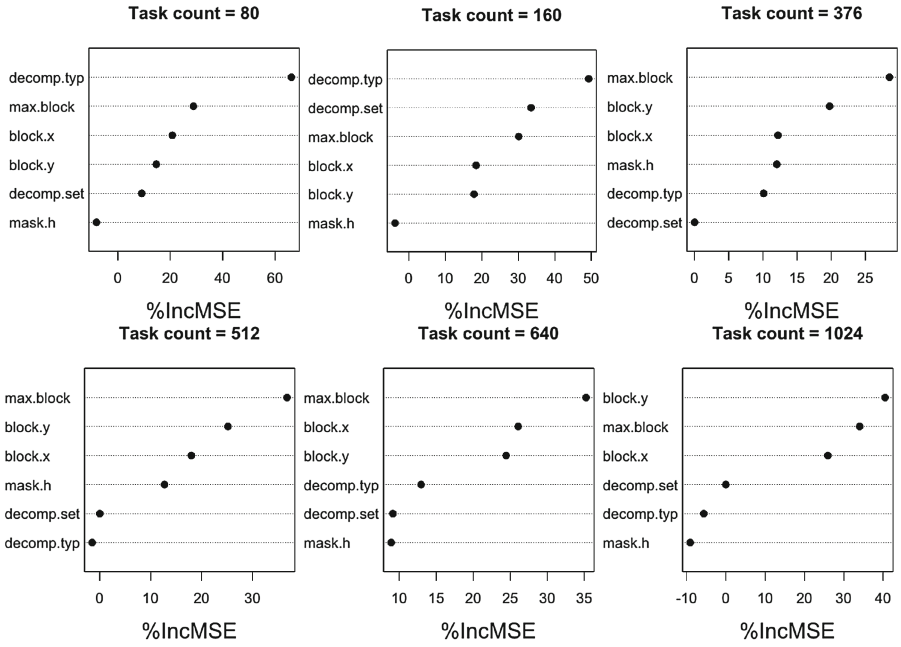

Fig. 2. Sensitivity analysis of the load-balancing parameters on the runtime of the CICE component for different task counts. For each parameter, the plot shows the percentage increase in mean squared error (%IncMSE) when the values of the corresponding parameter gets imputed.

nonlinear interactions between them change with an increase in the task counts. The previously developed analytical model does not take this effect into account for the CICE component, and consequently it falls short in runtime prediction. Moreover, if we build a single model for all task counts with task count being an input to the model, we might loose these task-count-specific interactions, thus affecting the runtime quality of the obtained configurations.

Comparison Between Variants: With EE as a baseline, we next examined the effectiveness of the five approaches in finding the optimal load-balancing configuration for the CICE component. As a measure of the effectiveness of each variant, we use the percentage deviation from the optimal runtime ($\%dev$). Given a variant v and task count c, this is given by $\frac{f_v^c - f_{opt}^c}{f_{opt}^c} \times 100$, where f_v^c is the shortest runtime obtained by variant v and f_{opt}^c is the optimal runtime obtained from EE. Because we repeated each method 10 times to reduce the impact of randomness, we consider the mean percentage deviation from the optimal runtime of a variant as $\%dev$ averaged over 10 repetitions. We also used a statistical t-test to check whether the observed differences in the $\%dev$ of the variants are significant. Figure 3 shows the comparison between the approaches. The results show that RS requires almost the same number of evaluations as does EE for all task counts. These results indicate that the problem of finding high-quality configurations is not an easy task;

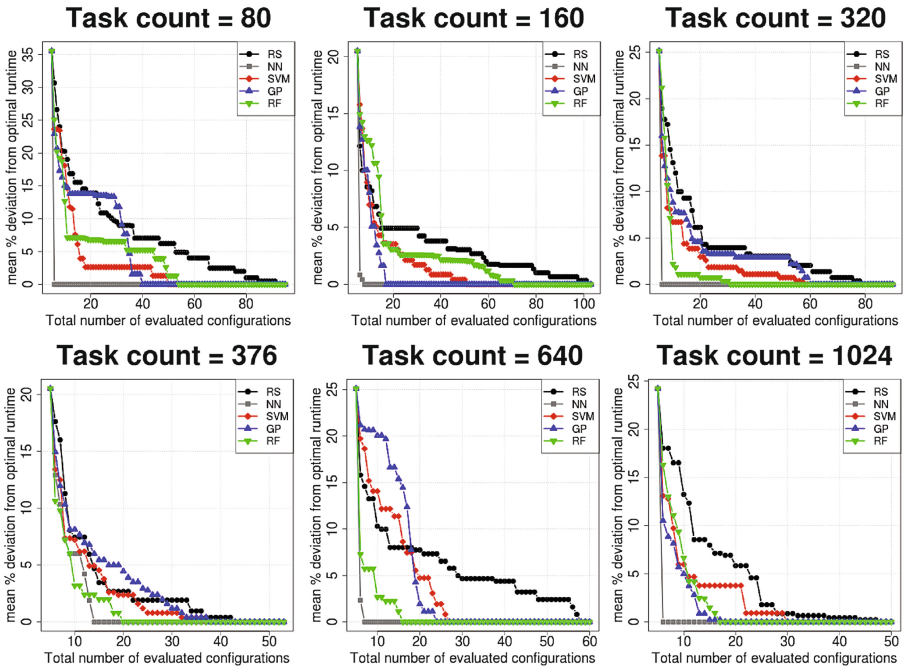

Fig. 3. Comparison between approaches for different task counts of the CICE component. The lines represent the mean percentage deviation from the optimal runtime as a function of the number of evaluated configurations.

clearly, we need more sophisticated approaches to find high-quality configurations within fewer evaluations. The variants of Algorithm 1 obtain optimal configurations with fewer evaluations, and they outperform RS. NN completely dominates all other variants and RS. The key advantage of NN comes from its requiring less than 10 evaluations to find the optimal parameter configuration on 9 out of 10 task counts—only on $c = 376$, does it require 15 evaluations.

In Table 2, we analyze $\%dev$ of each variant, when it is allowed to perform only 10 evaluations (for machine-learning variants this corresponds to five evaluations after the initialization). The results show that mean $\%dev$ of NN is zero and it lower than all other variants. For all but one task counts, the observed differences are significant in statistical sense. NN fails to find optimal runtime for $c = 376$, where it is 6 % away from the optimal runtime and it is comparable to other approaches.

As soon as a new evaluation becomes available, each machine-learning variants is retrained on all the available input-output pairs. This is the most computationally expensive part in the iterative phase of Algorithm 1. In Fig. 4, we analyze the retraining time required by the machine-learning variants after each evaluation. The reported time is an average time over all repetitions and task count. The results show that NN outperforms all other variants in retraining

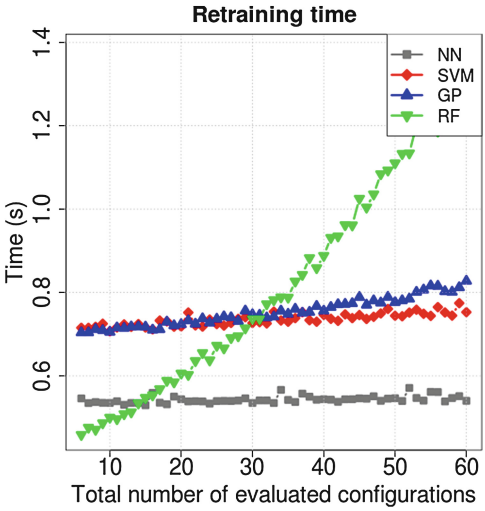

Fig. 4. Time taken by various machine-learning methods in Algorithm 1 for retraining after each evaluation.

time. The time remains fairly constant throughout with an average of 0.5 s. This can be attributed the effective back propagation algorithm adopted in the underlying optimization routine. For GP and SVM, there is a slight increase in retraining time. Nonetheless, the retraining time of RF increases linearly with an increase in the number of training points suggesting that it might not be suitable for sequential learning with large number of points. Note that there exists some advanced algorithm-specific techniques to avoid retraining from scratch, however, none of the machine-learning methods adopts such technique in our study. Furthermore, in all these algorithms, the time to predict an unseen input x^* is negligible (in the order of milli to micro seconds) because they belong to a class of eager learning algorithms as opposed to lazy learning algorithms where a model is built only when x^* needs to be predicted.

5 Related Work

Compared with dynamic strategies [11–16], static load-balancing approaches have received relatively less attention in the literature. The problem of static load-balancing can be formulated as a graph-partitioning problem that belongs to a class of $\mathcal{NP}$-hard problem for which finding optimal solution is computationally hard. Many efficient algorithms are developed to tackle this problem in operations research community and are used for static load-balancing. These algorithms can be grouped into geometry-based algorithms, graph-based algorithms, and partitioning algorithms [17]. In [18], the authors carried out an experimental comparison of eleven static load-balancing algorithms for heterogeneous distributed computing systems. They showed the relatively simple

Table 2. Mean percentage deviation from the optimal runtime averaged over 10 replications with the maximum budget of 10 evaluations

Task count	NN	RF	GP	SVM	RS
80	0.000	*12.668*	*15.032*	*18.089*	*20.246*
128	0.000	3.269	*7.620*	*5.177*	*12.846*
160	0.000	*12.649*	*8.050*	*6.989*	*8.563*
256	0.000	4.575	*8.468*	*7.340*	*10.024*
320	0.000	*2.208*	*8.818*	*6.709*	*13.105*
376	6.005	**3.186**	8.132	7.206	7.456
512	0.000	*10.269*	*11.794*	*6.472*	*9.090*
640	0.000	2.674	*20.058*	*14.072*	*10.292*
800	0.000	*7.435*	*5.996*	*6.182*	*8.770*
1024	0.000	*6.645*	*4.985*	5.966	*13.241*

Note: The value is typeset in *italics* (bold) when a variant is significantly worse (better) than NN according to a t-test with significance (alpha) level 0.05.

Min-Min heuristic performs well in comparison to the other techniques such as simulated annealing and genetic algorithms, and tabu search. However, the state-of-the-art high-performing algorithms comprises hybrid algorithms, multilevel approaches, and parallel implementations of the above algorithms [17]. We refer the reader to [17,19] for a survey for static load balancing approaches. Recently, in [20], a genetic algorithm was adopted for tasks scheduling and load balancing in heterogeneous parallel multiprocessor system. Nonetheless, the domain-specific constraints of the CICE component make the search problem hard and prevents the straightforward adoption of heuristic search algorithms [21]. In order to handle these constraints effectively, the search algorithms need a sophisticated constraint-handling mechanism; consequently they loose generality and become problem-specific.

The idea of using machine learning in load-balancing has received considerable attention for dynamic strategies. Examples include neural network [22], decision tree [23], and reinforcement learning approaches [24]. However, to the best of our knowledge, the adoption of machine-learning approaches for application and architecture specific static load-balancing has not been investigated before. Finally, this is the first work on the use of machine learning approaches for analyzing the sensitivity of the load-balancing parameters.

6 Summary and Outlook

We developed a machine-learning-based approach for static load-balancing problem and applied it for load balancing the CICE component of the CESM running on BG/P. We deployed a machine-learning method as a diagnostic tool for

analyzing the sensitivity of the load-balancing parameters on the runtime and provided an explanation for inadequacy of the analytical performance model. The main contribution of the paper is the development and empirical analysis of the machine-learning-based algorithm that allowed us to load balance the CICE component of the CESM on BG/P with significant savings in core-hour usage. Compared to the current practice of expert-knowledge-based enumeration over feasible parameter configurations, we showed that the proposed algorithm requires 6x fewer evaluations to find the optimal load-balancing configurations.

A inherent limitation of our algorithm consists in the sequential evaluation of parameter configurations that will affect the wall clock time. To address this issue, we will develop unsupervised learning methods to partition the feasible set into a number of similar groups and learning those regions in parallel. To that end, we will investigate parallel machine-learning algorithms. Since the inefficient processor allocations of CICE component can affect overall scaling of the CESM, we will use the proposed approach and assess the overall performance of the CESM. Furthermore, two projects, Climate-Science Computational End Station Development and Attributing Changes in the Risk of Extreme Weather and Climate, granted computational time on ALCF's BG/P and Q supercomputers under the DOE INCITE program will directly benefit from this work. We are planning to investigate the effectiveness of the proposed algorithm for load-balancing various climate simulations in these projects.

Acknowledgments. This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357. An award of computer time was provided by the Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program. This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

1. Metz, B., Davidson, O., Bosch, P., Dave, R., Meyer, L.: Contribution of working group III to the fourth assessment report of the Intergovernmental Panel on Climate Change (2007)
2. MINOTAUR: a toolkit for MINLP. http://wiki.mcs.anl.gov/minotaur/index.php/Main_Page
3. 2013. http://www.cesm.ucar.edu/events/ws.2012/Presentations/SEWG2/craig.pdf
4. Bishop, C.M., et al.: Pattern Recognition And Machine Learning. Springer, New York (2006)
5. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001)
6. Hearst, M.A., Dumais, S., Osman, E., Platt, J., Scholkopf, B.: Support vector machines. Intell. Syst. Appl. **13**(4), 18–28 (1998). IEEE
7. Rasmussen, C.E., Williams, C.K.: Gaussian Processes For Machine Learning. adaptive computation and machine learning. MIT Press, Cambridge (2005)

8. Haykin, S.: Neural Networks: A Comprehensive Foundation, 1st edn. Prentice Hall PTR, Upper Saddle River (1994)
9. Atkinson, E.J., Therneau, T.M.: An Introduction To Recursive Partitioning Using The Rpart Routines. Mayo Foundation, Rochester (2000)
10. R Core Team, R: A Language and Environment for Statistical Computing, R Foundation for Statistical Computing, Vienna, Austria (2013). http://www.r-project.org
11. Kale, L.V., Krishnan, S.: CHARM++: a portable concurrent object oriented system based on C++. ACM SIGPLAN Not. **28**(10), 91–108 (1993)
12. Barker, K., Chernikov, A., Chrisochoides, N., Pingali, K.: A load balancingframework for adaptive and asynchronous applications. IEEE Trans. Parallel Distrib. Syst. **15**(2), 183–192 (2004)
13. Barker, K.J., Chrisochoides, N.P.: An evaluation of a framework for the dynamic load balancing of highly adaptive and irregular parallel applications. In: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, p. 45. ACM (2003)
14. Huang, C., Zheng, G., Kalé, L., Kumar, S.: Performance evaluation of adaptive MPI. In: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 12–21. ACM (2006)
15. Boneti, C., Gioiosa, R., Cazorla, F.J., Valero, M.: A dynamic scheduler for balancing HPC applications. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, p. 41. IEEE Press (2008)
16. Sharma, R., Kanungo, P.: Dynamic load balancing algorithm for heterogeneous multi-core processors cluster. In: 2014 Fourth International Conference on Communication Systems and Network Technologies (CSNT), pp. 288–292. IEEE (2014)
17. Hu, Y., Blake, R.: Load balancing for unstructured mesh applications. Parallel Distrib. Comput. Pract. **2**(3), 117–148 (1999)
18. Braun, T.D., et al.: A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. J. Parallel Distrib. Comput. **61**(6), 810–837 (2001)
19. Ichikawa, S., Yamashita, S.: Static load balancing of parallel PDE solver for distributed computing environment. In: Proceedings of the 13th International Conference on Parallel and Distributed Computing Systems, pp. 399–405 (2000)
20. Effatparvar, M., Garshasbi, M.: A genetic algorithm for static load balancing in parallel heterogeneous systems. Procedia Soc. Behav. Sci. **129**, 358–364 (2014)
21. Balaprakash, P., Wild, S.M., Hovland, P.D.: Can search algorithms save large-scale automatic performance tuning? In: International Conference on Computational Science (2011)
22. Jia, Y., Sun, J.-Z.: A load balance service based on probabilistic neural network. In: International Conference on Machine Learning and Cybernetics, vol. 3, pp. 1333–1336. IEEE (2003)
23. Dantas, M.A., Pinto, A.R.: A load balancing approach based on a geneticmachine learning algorithm. In: 19th International Symposium on HighPerformance Computing Systems and Applications (HPCS 2005), pp. 124–130. IEEE (2005)
24. Helmy, T., Shahab, S.A.: Machine learning-based adaptive load balancing framework for distributed object computing. In: Chung, Y.-C., Moreira, J.E. (eds.) GPC 2006. LNCS, vol. 3947, pp. 488–497. Springer, Heidelberg (2006)