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

Prasanna Balaprakash[1,2], Yuri Alexeev[2], Sheri Mickelson[1], Sven Leyffer[1], Robert Jacob[1], and Anthony Craig[3]

[1] Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL
[2] Leadership Computing Facility, Argonne National Laboratory, Argonne, IL
[3] UCAR, Seattle, WA

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 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 contributions of the paper are 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 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` $\in$ {blkrobin, roundrobin, spacecurve, blkcart, cartesian} and `decomp.set` $\in$ {null, slenderX1, slenderX1}. A binary parameter `mask.h` $\in$ {0,1} that specifies to run the code with or without halo.

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

The constraints that define a feasible set $\mathcal{D}$ of configurations are as follows. The parameter `max.block` $\in$ {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 $\times$ CICE_Y_Grid_Size) / (`block.x` $\times$ `block.y` $\times$ 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.

The problem of static load-balancing can be formulated as an $\mathcal{NP}$-hard graph-partitioning problem; the algorithms developed to tackle this problem can be grouped into geometry-based algorithms, graph-based algorithms, and partitioning algorithms [5]. In [6], the authors showed that a relatively simple min-min heuristic performs well compared with other techniques such as simulated annealing, genetic algorithms, and tabu search. However, the state-of-the-art high-performing algorithms comprise hybrid algorithms, multilevel approaches, and parallel implementations of these algorithms [5]. We refer the reader to [5][7] for a survey of static load-balancing approaches. Nonetheless, the domain-specific constraints of the CICE component make the search problem hard and prevents the straightforward adoption of heuristic search algorithms [8]. 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 [9], decision tree [10], and reinforcement learning approaches [11]. 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. Moreover, this is the first work on the use of machine learning approaches for analyzing the sensitivity of the load-balancing parameters.

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

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

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

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

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 $\cup$ 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 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) [12], support vector machines (SVM) [13], Gaussian process regression (GP) [14], and neural networks (NN) [15].

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 [16] 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 Figure 1. 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`,

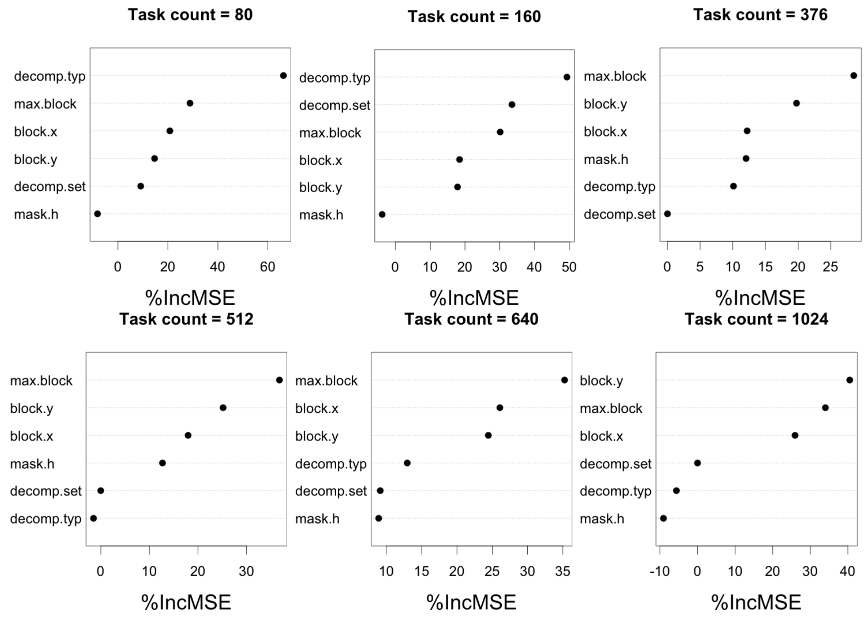

Fig. 1. Sensitivity analysis of the load-balancing parameters on the runtime of the CICE component for different task counts. For each parameter, the plot shows the percentage increase in mean squared error (%IncMSE) when the values of the corresponding parameter gets imputed.

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 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 lose 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 2 shows the comparison between the approaches. The results show that

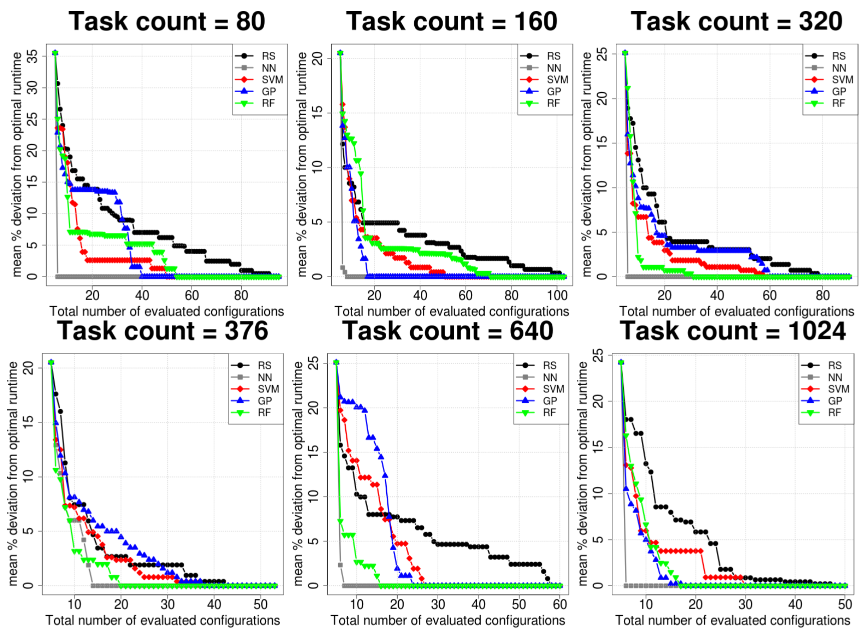

Fig. 2. Comparison between approaches for different task counts of the CICE component. The lines represent the mean percentage deviation from the optimal runtime as a function of the number of evaluated configurations.

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; 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.

5 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. We plan to investigate parallel machine-learning algorithms for static load-balancing. We also will deploy the proposed algorithm for load-balancing various climate simulations in CESM on other architectures.

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. B. Metz, O. Davidson, P. Bosch, R. Dave, and L. Meyer, “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. C. M. Bishop *et al.*, *Pattern Recognition and Machine Learning.* Springer, New York, 2006, vol. 1.
5. Y. Hu and R. Blake, “Load balancing for unstructured mesh applications,” *Parallel and Distributed Computing Practices*, vol. 2, no. 3, pp. 117–148, 1999.
6. T. D. Braun *et al.*, “A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems,” *Journal of Parallel and Distributed Computing*, vol. 61, no. 6, pp. 810–837, 2001.
7. S. Ichikawa and S. Yamashita, “Static load balancing of parallel PDE solver for distributed computing environment,” in *Proc. 13th Int’l Conf. Parallel and Distributed Computing Systems*, 2000, pp. 399–405.
8. P. Balaprakash, S. M. Wild, and P. D. Hovland, “Can search algorithms save large-scale automatic performance tuning?” in *Int. Conf. on Computational Science*, 2011.
9. Y. Jia and J.-Z. Sun, “A load balance service based on probabilistic neural network,” in *International Conference on Machine Learning and Cybernetics*, vol. 3. IEEE, 2003, pp. 1333–1336.
10. M. A. Dantas and A. R. Pinto, “A load balancing approach based on a genetic machine learning algorithm,” in *19th International Symposium on High Performance Computing Systems and Applications (HPCS 2005).* IEEE, 2005, pp. 124–130.
11. T. Helmy and S. Shahab, “Machine learning-based adaptive load balancing framework for distributed object computing,” in *Advances in Grid and Pervasive Computing.* Springer, 2006, pp. 488–497.
12. L. Breiman, “Random forests,” *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.
13. M. A. Hearst, S. Dumais, E. Osman, J. Platt, and B. Scholkopf, “Support vector machines,” *Intelligent Systems and Their Applications, IEEE*, vol. 13, no. 4, pp. 18–28, 1998.
14. C. E. Rasmussen and C. K. Williams, “Gaussian processes for machine learning (adaptive computation and machine learning),” 2005.
15. S. Haykin, *Neural Networks: A Comprehensive Foundation*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1994.
16. R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2013. [Online]. Available: http://www.r-project.org