Adaptive Load Balancing based on Machine Learning for Iterative Parallel Applications

Anna Victoria C R Oikawa, Vinicius Freitas, Márcio C Castro, Laércio Lima Pilla

Adaptive Load Balancing based on Machine Learning for Iterative Parallel Applications

Anna Victoria C. R. Oikawa, Vinicius Freitas, Márcio Castro
Federal University of Santa Catarina (UFSC)
Florianópolis, Brazil
anna4victoria@gmail.com, vinicius.mctf@posgrad.ufsc.br, marcio.castro@ufsc.br

Laércio L. Pilla
LRI, Univ. Paris-Saclay – CNRS
Orsay, France
pilla@lri.fr

***Abstract*—The performance of irregular scientific applications can be easily affected by an uneven distribution of work among the computing resources. In this context, Load Balancing (LB) stands as one of the most important solutions to improve resource utilization. However, choosing the best-performing load balancing algorithm for a given application is not a trivial task. For instance, manually and statically choosing an LB algorithm does not work in situations where applications have a dynamic or unknown behavior. In this context, we propose a Machine Learning-based Adaptive Load Balancer (ADAPTIVELB) to automate the load balancing algorithm decision at run time. This approach monitors and collects information about the application dynamically, and according to the analyzed data, it makes a decision of invoking the most suitable LB algorithm. Our experiments show that ADAPTIVELB can select a good load balancing algorithm in most of the cases, leading to performance improvements over statically chosen LB algorithms and over the absence of a load balancer.**

***Keywords*-dynamic load balancing; machine learning; performance.**

I. INTRODUCTION

Load imbalance is a recurring problem that haunts High Performance Computing (HPC) applications since their inception. Dynamic simulations, such as Molecular Dynamics (MD) [1], Adaptive Mesh Refinement (AMR) [2], and other N-Body simulations [3], tend to exhibit unpredictable workload behavior. As machines grow larger, power consumption becomes one of the main concerns of large-scale computing [4], and wasting resources due to load imbalance in parallel applications becomes unacceptable.

Dynamic Load Balancing (LB) emerges as a solution to unpredictable application behavior, enabling the scalability of these simulations [5]. However, solutions differ among applications, even though they are (in an abstract sense) solving the same problem, scheduling jobs in parallel machines. For instance, iterative and bulk synchronous parallel applications may use periodical load balancers [6]; data-flow and direct asynchronous graph-decomposed applications may profit from work stealing [7]; and AMR implementations may perform dynamic migrations during refinement stages [2].

Nonetheless, there is no LB approach that is fit to every parallel application. Additionally, since aspects of application behavior may vary during execution (e.g., communication and computation costs of each task), a single load balancer may not be able to always serve an application efficiently. Selecting an LB algorithm is no trivial task, especially for those who are not intimate with the imbalanced application.

With this in mind, we propose a Machine Learning-based Adaptive Load Balancer (ADAPTIVELB) that chooses a suitable LB algorithm for a given state of the application. ADAPTIVELB considers system workload information and communication among jobs to determine what is the best possible LB algorithm among those that it was trained with. We implemented ADAPTIVELB in Charm++ using a K-Nearest Neighbors (KNN) classifier in Python, and our experimental results using it on a parallel platform showed improvements over statically-chosen LB algorithms.

The remainder of this paper is divided as follows: Section II discusses basic concepts related to load balancing and Machine Learning (ML). Section III presents ADAPTIVELB, its multiple phases, and the ML algorithm decision process. Section IV describes the experimental evaluation, while Section V discusses related work on LB and ML. Section VI presents concluding remarks.

II. BACKGROUND

A. Load Balancing

Load Balancing (or Global Scheduling [8]) is regarded as defining *where* to run tasks (or work units) in a parallel system, while the *when* is left to a local scheduler. We may abstract this as the problem of minimizing the maximum usage of resources in a parallel machine ($P||C_{max}$) in a system of identical machines, which has been regarded as NP-Hard [9]. The dynamic and iterative nature of parallel scientific applications, along with the increasing number of

DOI 10.1109/PDP50117.2020.00021

Processing Elements (PEs) in HPC platforms, make the load imbalance an unavoidable issue.

We may divide LB algorithms in two main categories: *Global* and *Diffusive* [10]. While global algorithms centralize system information to make scheduling decisions (most list scheduling approaches follow this methodology [11], [12]); diffusive algorithms will use local machine information to dissipate load among its neighbors or other machines in the system [13], [14], [15].

Recently, some Parallel Runtime Systems (RTSs) that feature different LB algorithms have been proposed. Charm++ [16] is one of the well-known RTSs that offer some interesting features such as LB framework, overdecomposition, migrability and introspection. *Overdecomposition* allows the division of computation into independent units, resulting in data and work units that will be mapped to each PE by the Charm++ Runtime System (Charm++ RTS) [16]. This initial mapping can be changed during execution by migrating objects (Charm++ *chares*) to other PEs if the application is in an imbalanced state in terms of workload, exploiting the objects *migrability*. *Introspection* allows the system to collect information about the application during run time, so it is possible to know the existing workload and to make load balancing decisions.

Although Charm++ provides a set of global and diffusive LB algorithms, we focus on global LB algorithms, which achieve better results than diffusive ones on shared memory architectures. We exploit the introspection feature to feed ADAPTIVELB with useful information about the workload. Based on this information, ADAPTIVELB chooses the most suitable LB algorithm for an application at run time in a transparent manner.

B. Machine Learning

Machine Learning has become a common tool to model the behavior of complex interactions between applications, systems and platforms. It provides a portable solution to predict the behavior of new instances based on a set of a priori trained ones. Within the field of ML, there are mainly three types of techniques: supervised, unsupervised, and reinforcement learning.

In supervised learning, we train the machine using data which is well labeled (i.e., data that is already tagged with the correct answer) [17]. Thus, the main objective of supervised learning is to learn a function that, given a set of samples and their desired outputs, best approximates the relationship between inputs and outputs observable in the data [18]. Common algorithms in supervised learning include logistic regression, decision trees, support vector machines, artificial neural networks and random forests [17].

Unsupervised learning, on the other hand, tries to infer the natural structure present within a set of samples without using any explicitly-provided labels [18]. Common algorithms in unsupervised learning are usually used to solve clustering (e.g., k-means) and association (e.g., a priori) problems [18]. Finally, reinforcement learning algorithms employ agents that take actions in sequence while trying to maximize a cumulative reward [18]. The agents learn during run time by receiving rewards based on the results of the actions that they take at each step and state.

In this work, we focus on supervised learning algorithms. As we will explain in the next section, ADAPTIVELB relies on ML techniques to predict the most suitable LB algorithm for an application at run time.

III. MACHINE LEARNING-BASED ADAPTIVE LOAD BALANCER

Supervised learning-based approaches share a common framework that is usually composed of static and dynamic phases. In the following sections, we show an overview of ADAPTIVELB and explain in detail each of these phases.

A. Overview

ADAPTIVELB is an entity that takes into account the different characteristics that make up the context where the application is being executed, whether they are the instant when the balancing occurs, aspects of the parallel platform of execution, or characteristics of the application itself. This gives us the ability to choose different heuristics for the same application execution at run time, according to the set of characteristics collected. Additionally, this approach avoids taking load balancing decisions manually by the user for each HPC application. This manual process demands time and knowledge of the application context, since the user must estimate each task load, predict their behavior to decide which algorithm should be used, and define the frequency that the load balancing should occur.

In this work, we considered the most used global LB algorithms in Charm++ applications. A brief description of each LB algorithm is given below:

- GREEDYLB is a greedy strategy that only uses task loads for its decisions (Longest Processing Time (LPT) policy [11]). It sorts tasks in decreasing load order and iteratively maps the unassigned task with the highest load to the least loaded PE. This process leads to a large number of task migrations, since GREEDYLB does not take into account the initial task mapping at each LB decision.
- GREEDYCOMMLB is similar to GREEDYLB, but it also considers the communication costs involved in task migrations. It sorts tasks in decreasing load order, and iteratively chooses a PE to assign the unassigned task with the highest load. However, the PE is chosen among the least loaded PE and all PEs that have tasks that communicate with the task being mapped. The decision is based on the *total load* of the PE that includes its current load, its communication load, and the communication load of the task. The communication load

of a task is related to how many messages and bytes it sends to tasks mapped to other PEs whereas the communication load of a PE is equal to the sum of the communication load of tasks currently mapped to it.

- REFINELB tries to improve load balance by incrementally adjusting the current task map, instead of creating an empty task map in each LB decision as done by GREEDYLB and GREEDYCOMMLB. To do so, it splits the PEs into two classes (*heavy* or *light*) according to their loads. A PE is considered heavy (or overloaded) if its load is greater than a specified threshold over the average PE load. Then, it checks all possible task migrations from the most loaded PEs to all light PEs, and migrates the task that leaves its new PE the closest to the threshold. This process is repeated until no migrations seem to improve load balance, or no PE is considered heavy.
- GREEDYREFINELB specifies a migration tolerance, and then uses a greedy algorithm to migrate the heaviest tasks in overloaded PEs to the least loaded ones. This is done until the migration tolerance limit is achieved. Like REFINELB, it attempts to minimize migrations, but it is adaptive when accounting the cost of migrations.

Fig. 1 shows the overview of the proposed approach, which is composed of static and dynamic phases. The main goal of the static phase is to generate ADAPTIVELB, which will be responsible for selecting the most appropriate LB algorithm during the dynamic phase. In the dynamic phase, the application is analyzed at run time and, based on its characteristics, a suitable LB algorithm is selected and applied by ADAPTIVELB to improve load balancing.

B. Input Data Extraction

The first step to be taken in the static phase is to generate a set of synthetic applications (a.k.a input instances) — in our case, using Charm++'s LBTest benchmark. This set should contain applications that vary in communication and computation load, leading to samples with characteristics and behaviors that differ from one another. This first stage is essential so that ADAPTIVELB is trained well with any possible scenarios that may come up in the future. In a real situation, the selection of the best algorithm will only be successful if, during the training phase, there was an input instance with characteristics that are similar to the ones of the real application being currently analyzed. Hence, we generate input instances that contain a variety of parameter values to illustrate different scenarios of real applications that may arise at a later time.

The main goal of the *Data Extraction* stage is to obtain, for every input instance in the training set, the characteristics that compose it and the best LB algorithm for those characteristics. In other words, characteristics will become the

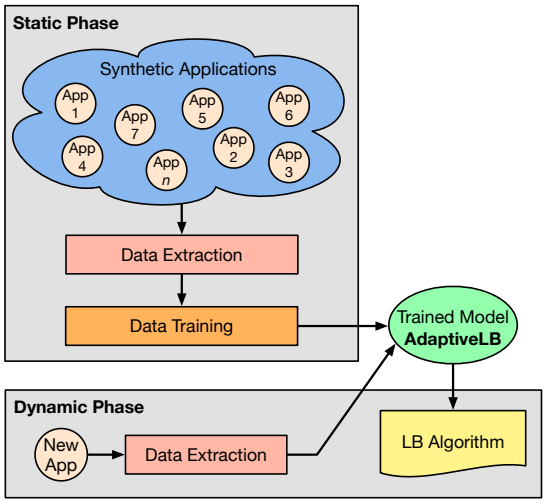


Figure 1. An overview of the proposed approach.

features for the future ML model, and the LB algorithm will become their respective class/label, modeling a classification problem. Therefore, this phase gathers the data for training and it validates the ML model.

For a given input instance, we want to know its characteristics at a given instant and which LB algorithm is the most appropriate one, given the data extracted from that instance. To do so, we take the average execution time of several runs of each input instance with each one of the LB algorithms described in Section III. Then, we compare the average execution times of each instance/LB pair to find the best LB algorithm for each input instance. That LB algorithm is said to be the most appropriate one for that input instance being analyzed.

Once we have the best LB algorithm for each one of the input instances, we need information about *how to represent* the input instance based on its characteristics (a.k.a. features). We implemented a *profiler* to monitor the application at run time with the purpose to collect relevant data about its load and behavior. This means that not only do we need to define which type of information represents an input instance well, but the overhead to calculate this data must be low enough so that ADAPTIVELB can still be beneficial.

The Charm++ RTS provides system load information by having an LB manager that resides on each processor monitoring the PE load, tasks load, background load, and other statistics such as idle time. This information is stored in an LB database that is accessed during this phase, allowing ADAPTIVELB to use the stored data on the decision making process for the new mapping of tasks to be performed. In this work, we focus on centralized strategies, where there is a dedicated processor that collects global information about the state of the entire machine.

It is important to choose the right amount of information

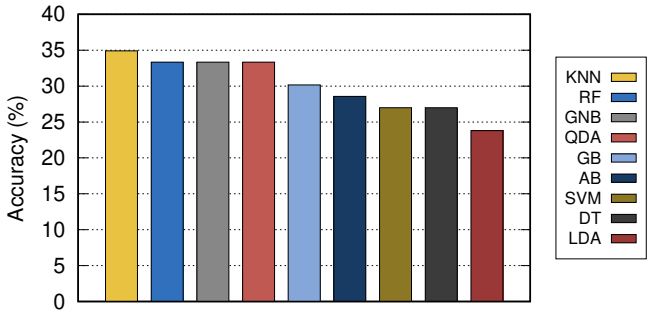


Figure 2. Accuracy measured for the different ML algorithms tested.

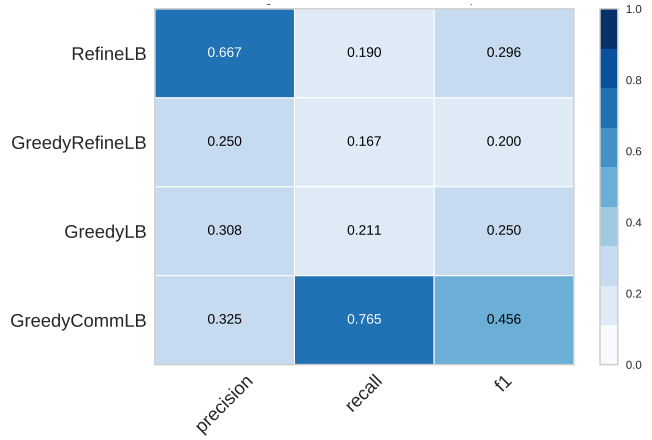


Figure 3. Precision, recall and F_1 results for each class for the KNN Classifier.

to represent an application, not only to avoid unnecessary overhead, but also to prevent an overfitting situation in our ML model. Hence, an empirical study was made to discard ambiguous features and to keep the ones that can be collected with low overhead. Overall, we selected 7 features: 1) average load of PEs; 2) percentage of overloaded PEs; 3) load imbalance (maximum PE load divided by the average load of PEs); 4) average PE idle time; 5) average load of tasks; 6) communication/computation ratio; and 7) average percentage of messages sent to other PEs.

We run each input instance several times with the profiler to collect information about all selected features. Then, we compute the average value of each profiled feature. Finally, ML algorithms can be trained using the set of features and classes found in the previous stages.

C. Training

The final stage in the static phase is training our ML model using a set of input instances that have been described by their features (characteristics) and by their class/label (the chosen LB). This fits perfectly in a classification problem for a supervised learning approach. Since several ML algorithms could be used, we performed an empirical study to determine the best ML algorithm for our problem.

We used the `scikit-learn` library [19] available for Python for this purpose. One advantage of using this library is the support for a variety of ML algorithms. Additionally, Python has become popular in the field of scientific computing due to its interactive nature, support to data analysis and visualization, and scientific libraries, such as NumPy, SciPy and Cython [20].

Given the data generated in the previous stage, we made an analysis based on the accuracy, classification report, and confusion matrix for each ML algorithm. The analyzed data was normalized, since the values from the features have a significant range variation. Finally, the input dataset was split into a training set (70%) and a validation set (30%), allowing to train the model with a significant amount of data and to later validate the trained model to verify its accuracy. The ML algorithms considered in our empirical study are: Ada Boost (AB), Decision Trees (DT), K-Nearest Neighbors (KNN), Gradient Boosting (GB), Gaussian Naive Bayes (GNB), Linear Discriminant Analysis (LDA), Quadratic Discriminant Analysis (QDA), Support Vector Machines (SVM), and Random Forests (RF). The values of the parameters used for each algorithm are listed in Appendix A.

We generated 314 input instances to train the ML algorithms using LBTest. Fig. 2 shows the classification accuracy results of all the aforementioned ML algorithms, which corresponds to the ratio of the number of correct predictions to the total number of input samples. As it can be noticed, the best ML algorithm for the context of this work was the KNN Classifier with a classification accuracy of 35%. After a deep analysis, we concluded that the KNN Classifier chose the second best LB algorithm in most of its wrong decisions and the second best LB algorithm usually had a performance similar to the best one. As we show in Section IV, this resulted in almost no performance degradation when compared to the best LB algorithm.

KNN performs the classification by computing the similarity based on the distance from the data to its closest neighbors. It verifies the distance from the data being tested to its neighbors, which have been classified previously in the training phase, and the data group (class) that presents the smallest distance is selected. The precision, recall and F_1 results for the KNN Classifier are shown in Fig. 3.

Before moving to the dynamic phase, the model is trained with the input dataset. Then, the trained model is serialized first before writing it to the disk for future use. The serialization procedure is done by the `pickle`[1] module available in Python, which implements serialization and deserialization protocols for Python objects. It converts an in-memory Python object into a serialized byte stream that contains all the information necessary to reconstruct the

[1]https://docs.python.org/3/library/pickle.html

Algorithm 1: ADAPTIVELB decision process.

```
1  features ← profiler.extract_features();
2  selected_balancer ← predict_balancer(features);
3  if selected_balancer == 'GreedyLB' then
4  |  greedyLB.work()
5  else if selected_balancer == 'GreedyCommLB' then
6  |  greedyCommLB.work()
7  else if selected_balancer == 'GreedyRefineLB' then
8  |  greedyRefineLB.work()
9  else
10 |  refineLB.work()
```

object afterwards. The de-serialization procedure is done in the dynamic phase and the trained model is kept in memory to be used by ADAPTIVELB during the execution of the application.

D. Online Predictor

The last stage is the dynamic phase, having as the final goal to predict the best LB algorithm for new, unseen, applications. Overall, the dynamic phase works as follows. The application is profiled at run time and its features are extracted when LB occurs. Then, this information is sent to ADAPTIVELB, which uses the trained model to make decisions. Algorithm 1 illustrates the decision making process executed by ADAPTIVELB.

Based on the features collected at run time from the new application (line 1), ADAPTIVELB is able to predict the most suitable LB algorithm. This is done by having a system call in the code of ADAPTIVELB in Charm++. Then ADAPTIVELB uses its KNN Classifier to choose an LB algorithm (line 2). In our experiments, these two steps took approximately 0.7 seconds to complete, on average.

Once an LB strategy has been selected for a given application at a given instant of time, ADAPTIVELB invokes the selected LB strategy by calling its `work()` method (lines 3 – 10). This method is responsible for performing the task migrations according to its LB heuristics. As it can be noticed, our solution can be easily extended to other LBs, since every centralized load balancer in Charm++ must implement a `work()` method.

It is important to point out that, during this phase, we can see one of the main advantages of using ADAPTIVELB. Depending on the workload characteristics, some LB heuristics make task migration decisions better than others. Applications that feature unpredictable and irregular behaviors at run time may achieve suboptimal performance due to wrong decisions made by a statically chosen LB heuristic. Since ADAPTIVELB chooses an LB algorithm dynamically, it is able to switch to a suitable LB algorithm based on the current state of the workload.

In the next section, we validate and analyze the results obtained from different executions of new applications with ADAPTIVELB on a real parallel platform.

IV. EXPERIMENTAL EVALUATION

This section presents a performance evaluation of ADAPTIVELB. We first detail the characteristics of the platform and applications used in the experimental evaluation. Then, we discuss the performance results obtained with ADAPTIVELB.

A. Platform and Applications

All experiments were run on an Intel(R) Xeon(R) ES-2640 v4 @ 2.40GHz (10 physical cores + HT) with 128 GB of RAM. The operating system used was Ubuntu Linux 16.04 with Charm++ version 6.9.0 installed (build multicore-linux-x86-64). All applications and Charm++ were compiled with GCC version 5.4.0 with `-O3` optimization.

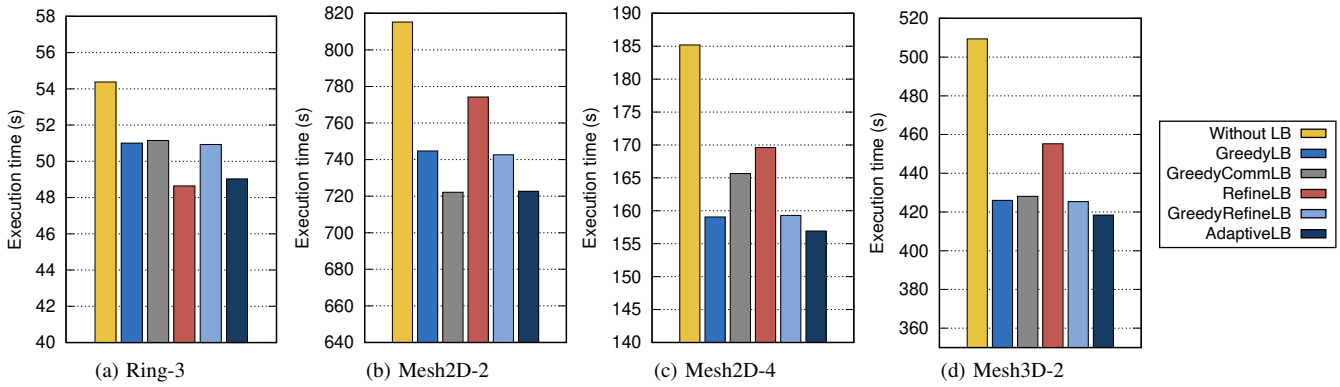
All applications considered in the experimental evaluation were generated with LBTest, which is a synthetic benchmark implemented in Charm++ with the purpose of testing LB algorithms. The advantage of its use is the flexibility in its parameters tuning, allowing to simulate a wide variety of computation and communication load distributions. The parameters set at launch time include: number of tasks, number of iterations, task communication topology (Ring, Mesh2D, Mesh3D), minimum duration time for a task, maximum duration time for a task and the LB algorithm.

As we mentioned before, we generated 314 input instances to train the KNN Classifier. In addition, we created 12 synthetic applications (4 applications for each task communication topology, named *Ring-1*, ..., *Ring-4*; *Mesh2D-1*, ..., *Mesh2D-4*; and *Mesh3D-1*, ..., *Mesh3D-4*), which have not been used in the training phase, to test the efficiency of ADAPTIVELB. The specific values of parameters used in these 12 synthetic applications are listed in Appendix A. All results in the next section represent the average over five repetitions for each experiment.

B. Experimental Results

Fig. 4 presents the average execution times achieved by 4 synthetic applications (*Ring-3*, *Mesh2D-2*, *Mesh2D-4* and *Mesh3D-2*) with different LB algorithms (GREEDYLB, GREEDYCOMMLB, REFINELB, GREEDYREFINELB, and ADAPTIVELB) and without load balancing. Note that each figure has its own y-axis scale because the execution times of these synthetic applications are significantly different. Moreover, the y-axis does not start at 0 to better show the differences between LB algorithms.

Fig. 4c shows the results obtained with *Mesh2D-4*. As it can be noticed, ADAPTIVELB was able to obtain a gain of approximately 28.3 seconds compared to the case where no load balancing occurs. When compared to the best LB algorithm for this case (GREEDYLB), ADAPTIVELB was 2.14 seconds faster. Among the other load balancers, we



(a) Ring-3 (b) Mesh2D-2 (c) Mesh2D-4 (d) Mesh3D-2

Figure 4. Execution times (in seconds) of 4 synthetic applications (*Ring-3*, *Mesh2D-2*, *Mesh2D-4* and *Mesh3D-2*). Vertical axes in different scales and with different starting points to emphasize differences.

can see that GREEDYCOMMLB and REFINELB showed the worst performance. A similar behavior was observed with *Mesh3D-2* (Fig. 4d), where ADAPTIVELB achieved the best performance among all other load balancers. Among the other load balancers, GREEDYREFINELB achieved the best performance for *Mesh3D-2*, although this meant taking approximately 6.9 seconds more than ADAPTIVELB.

Although REFINELB showed the worst performance in both previous cases, Fig. 4a (*Ring-3*) displays a scenario where it performed the best among all load balancers, even surpassing ADAPTIVELB by approximately 0.4 seconds. After analyzing this scenario in more detail, we concluded that this difference came mostly from the overhead of the invocation of the ML model.

Fig. 4b illustrates the results for experiment *Mesh2D-2*, which uses the same communication pattern of the experiment in Fig. 4c. In this experiment, GREEDYCOMMLB provided the best performance with ADAPTIVELB taking approximately 0.6 seconds more in its execution. It is interesting to note that, even though the same communication pattern is being used, there is an inversion in performance behavior in these two experiments between GREEDYCOMMLB and both GREEDYLB and GREEDYREFINELB. Furthermore, each of these four experiments shows a different load balancing algorithm as the best one (besides ADAPTIVELB), reinforcing the difficulty of finding the best LB for an application.

Table I summarizes ADAPTIVELB's performance throughout the different experiments. Positive values represent the cases where ADAPTIVELB was faster than the approach it is being compared to, and the negative values represent the cases where our strategy was slower. It is important to note that there is an overhead of about 0.7 seconds involved in each invocation of the trained ML model at run time, as mentioned in Section III-D. Thus, if that overhead would happen to be reduced in future works, then ADAPTIVELB would have achieved the best[2] performance in all experiments.

Table I
ADAPTIVELB OVERALL PERFORMANCE

Experiment	Performance Gain with LB (s)	Performance Gain without LB (s)
Ring-1	2.0278	21.3634
Ring-2	0.0776	7.4856
Ring-3	-0.3937	5.3408
Ring-4	0.1118	0.2416
Mesh2d-1	0.0007	0.8634
Mesh2d-2	-0.5747	92.4698
Mesh2d-3	0.7320	5.2299
Mesh2d-4	2.1443	28.2617
Mesh3d-1	-1.0290	19.3386
Mesh3d-2	6.9394	90.9603
Mesh3d-3	0.5797	10.8362
Mesh3d-4	7.3626	6.1183

The application *Mesh3D-4* represents an interesting scenario. In this case, ADAPTIVELB was about 7.4 seconds faster than the second best balancer and about 6.1 seconds faster than the case where no load balancing occurs. In other words, if we were to compare only the four LB algorithms described in Section III with the scenario without load balancing, then we would see that all other load balancers would *degrade* the performance of the application. For the application being executed in this experiment, its behavior shows that using the *same* balancer throughout its *entire* lifetime does not present any benefits, since the cost of migrations made the overall performance worse. Since ADAPTIVELB has an adaptive nature and reacts to the current state of the application, it was able to switch between LB algorithms throughout the application execution, resulting in a better overall performance.

[2]Although ADAPTIVELB is around 1 second slower than the best LB for *Mesh3d-1*, this experiment includes two LB calls for a total of 1.4 seconds of overhead.

V. Related Work

Since load balancing in parallel machines is an NP-hard problem, which can lead to major impacts on application performance [5], it is a very well studied subject. Procedural solutions are the most prevalent in the literature, since dynamic load balancers must solve this issue in a timely manner.

Existing solutions tend to account for different machine-specific or application-specific characteristics, such as memory contention [21], network congestion [22], and number of hops [22]. Additionally, they may employ related strategies such as graph partitioning [23], hierarchical decision making [24], or topology mapping [25].

ML may be employed in several different ways to optimize scheduling in parallel machines. It has been used to optimize thread mapping in Transactional Memory (TM) applications [26], [27], using TM benchmarks to train and evaluate an ID3 decision tree model, which outputs a mapping strategy to be employed.

In a different approach, statistical learning has been used to infer the best thread and data mapping [28], with focus on Non-Uniform Memory Access (NUMA) machines. This strategy employs metrics acquired from hardware counters in order to profile applications and determine the best mapping strategy in a given situation.

Reinforcement learning (RL) has been employed in iterative applications to find, during execution, the loop scheduling algorithms that are the best performing or the most robust to perturbations [29], [30]. In this approach, each internal loop of the application has its own learning agent and, at each time step, the agent must choose a scheduling algorithm for its loop. Although RL algorithms are shown to be suitable for this scenario, they do not map well to ours where an LB is chosen only a few times during the execution of the application.

Automated LB decisions have also been employed to determine the best time to perform load balancing in Charm++ [31]. The idea is to continually collect application behavior data in order to start the LB, instead of using a static iteration interval. The same author also proposed the use of ML to determine the best possible LB algorithm in Charm++ [32]. However, the author did not analyze the performance impact of the proposed approach on the applications. Moreover, the proposed solution only employed a decision tree and a random forest algorithm for its decision-making process. In contrast, we evaluated 9 different ML algorithms and selected the most appropriate one. In addition, we carried out a performance evaluation with different applications, illustrating the benefits of AdaptiveLB.

VI. Conclusion

Load imbalance is a critical issue that affects the performance and scalability of diverse parallel applications. Selecting the best LB algorithm for applications that are unpredictable, irregular and dynamic is a non-trivial task. Additionally, choosing an LB algorithm statically may present a good performance at an initial moment, but due to the application dynamic behavior, that LB can become inappropriate throughout the execution.

In this context, we presented an approach for automating the LB algorithm decision at run time by using ML named AdaptiveLB. We demonstrated that AdaptiveLB is able to switch between LB algorithms according to the needs that a given application presents at a given instant of time, achieving the best performance in most cases. We showed that AdaptiveLB presented advantages over the scenario where no load balancing occurs, even in the case that a statically balancer selection had a slower performance compared to the context without load balancing. In the cases where AdaptiveLB was the second best strategy, it was approximately one second slower than the fastest approach. We observed that most of this performance degradation was due to the overhead of about 0.7 seconds for each invocation of the trained model at run time.

As future work, we intend to reduce the overhead caused by the ML model invocation, leading to a better overall performance. Additionally, we intend to extend the current solution to diffusive (distributed) LB algorithms. In that case, other features related to communication overhead and network congestion shall be added to the model. Finally, we intend to apply the same ML-based approach to other application programming libraries.

Acknowledgment

This work was partially supported by the *Conselho Nacional de Desenvolvimento Científico e Tecnológico* – Brasil (CNPq) under the Universal Program (grant number 401266/2016-8) and by the *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior* – Brasil (CAPES) under the Capes-PrInt Program (grant number 88881.310783/2018-01).

References

[1] A. Bhatele *et al.*, "NAMD: A portable and highly scalable program for biomolecular simulations," University of Illinois, Urbana-Champaign, US, Tech. Rep., 2009, technical Report.

[2] R. F. Van der Wijngaart, E. Georganas, T. G. Mattson, and A. Wissink, "A new parallel research kernel to expand research on dynamic load-balancing capabilities," in *Proceedings of the International Supercomputing Conference (ISC)*. Frankfurt, Germany: Springer, 2017, pp. 256–274.

[3] P. Jetley *et al.*, "Scaling hierarchical N-body simulations on GPU clusters," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. New Orleans, USA: IEEE Computer Society, 2010, pp. 1–11.

[4] J. Dongarra *et al.*, "The international exascale software project roadmap," *The International Journal of High Performance Computing Applications*, vol. 25, no. 1, pp. 3–60, 2011.

[5] C. Mei *et al.*, "Enabling and scaling biomolecular simulations of 100 million atoms on petascale machines with a multicore-optimized message-driven runtime," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Seattle, USA: IEEE/ACM, 2011, pp. 61:1–61:11.

[6] H. Menon and L. Kalé, "A distributed dynamic load balancer for iterative applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Denver, USA: ACM, 2013, pp. 15:1–15:11.

[7] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM*, vol. 46, no. 5, pp. 720–748, 1999.

[8] T. L. Casavant and J. G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Transactions on Software Engineering*, vol. 14, no. 2, pp. 141–154, 1988.

[9] M. R. Garey and D. S. Johnson, "Strong NP-completeness results: Motivation, examples, and implications," *Journal of the ACM*, vol. 25, no. 3, pp. 499–508, Jul. 1978.

[10] O. Pearce *et al.*, "Quantifying the effectiveness of load balance algorithms," in *Proceedings of the International Conference on Supercomputing (ICS)*. Venice, Italy: ACM, 2012, pp. 185–194.

[11] R. L. Graham, "Bounds for certain multiprocessing anomalies," *Bell System Technical Journal*, vol. 45, no. 9, pp. 1563–1581, 1966.

[12] J. Lifflander, S. Krishnamoorthy, and L. V. Kalé, "Work Stealing and Persistence-based Load Balancers for Iterative Overdecomposed Applications," in *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*. Delft, NL: ACM, 2012, pp. 137–148.

[13] M. H. Willebeek-LeMair and A. P. Reeves, "Strategies for dynamic load balancing on highly parallel computers," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 4, no. 9, pp. 979–993, 1993.

[14] M. Lieber, K. Gössner, and W. E. Nagel, "The potential of diffusive load balancing at large scale," in *Proceedings of the European MPI Users' Group Meeting (EuroMPI)*. Edinburgh, United Kingdom: ACM, 2016, pp. 154–157.

[15] V. Freitas, A. Santana, M. Castro, and L. L. Pilla, "A batch task migration approach for decentralized global rescheduling," in *Proceedings of the International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. Lyon, France: IEEE, 2018, pp. 49–56.

[16] B. Acun *et al.*, "Parallel programming with migratable objects: Charm++ in practice," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. New Orleans, USA: IEEE/ACM, 2014, pp. 647–658.

[17] E. Alpaydin, *Introduction to Machine Learning*, 2nd ed. The MIT Press, 2010.

[18] Y. Hamid, M. Sugumaran, and L. Journaux, "Machine learning techniques for intrusion detection: A comparative analysis," in *Proceedings of the International Conference on Informatics and Analytics (ICIA)*, ser. ICIA-16. Pondicherry, India: ACM, 2016, pp. 53:1–53:6.

[19] F. Pedregosa *et al.*, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[20] S. van der Walt, S. Colbert Chris, and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," *Computing in Science and Engineering*, vol. 13, pp. 22–30, 2011.

[21] Q. Chen and M. Guo, "Contention and locality-aware work-stealing for iterative applications in multi-socket computers," *IEEE Transactions on Computers*, vol. 67, no. 6, pp. 784–798, 2018.

[22] M. Deveci, K. Kaya, B. Uçar, and U. V. Catalyurek, "Fast and high quality topology-aware task mapping," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. Hyderabad, India: IEEE, 2015, pp. 197–206.

[23] U. V. Catalyurek *et al.*, "Hypergraph-based dynamic load balancing for adaptive scientific computations," in *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*. Long Beach, USA: IEEE, 2007, pp. 1–11.

[24] L. L. Pilla *et al.*, "A topology-aware load balancing algorithm for clustered hierarchical multi-core machines," *Future Generation Computer Systems (FGCS)*, vol. 30, pp. 191–201, 2014.

[25] E. Jeannot, G. Mercier, and F. Tessier, "Process placement in multicore clusters:algorithmic issues and practical techniques," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 25, no. 4, pp. 993–1002, 2014.

[26] M. Castro, L. F. W. Góes, C. P. Ribeiro, M. Cole, M. Cintra, and J.-F. Méhaut, "A machine learning-based approach for thread mapping on transactional memory applications," in *Proceedings of the High Performance Computing Conference (HiPC)*. Bangalore, India: IEEE, 2011, pp. 1–10.

[27] M. Castro, L. F. Góes, L. G. Fernandes, and J.-F. Méhaut, "Dynamic thread mapping based on machine learning for transactional memory applications," in *Proceedings of the International European Conference on Parallel and Distributed Computing (Euro-Par)*, ser. Lecture Notes in Computer Science (LNCS), vol. 7484. Rhodes Island, Greece: Springer-Verlag, 2012, pp. 465–476.

[28] N. Denoyelle, B. Goglin, E. Jeannot, and T. Ropars, "Data and thread placement in NUMA architectures: A statistical learning approach," in *Proceedings of the International Conference on Parallel Processing (ICPP)*. Kyoto, Japan: ACM Press, 2019, pp. 1–10.

[29] I. Banicescu, F. M. Ciorba, and S. Srivastava, *Scalable Computing: Theory and Practice*. John Wiley&Sons, Inc., 2013, ch. Performance Optimization of Scientific Applications using an Autonomic Computing Approach, pp. 437–466.

[30] A. Boulmier, I. Banicescu, F. M. Ciorba, and N. Abdennadher, "An autonomic approach for the selection of robust dynamic loop scheduling techniques," in *Proceedings of the International Symposium on Parallel and Distributed Computing (ISPDC)*. Innsbruck, Austria: IEEE, 2017, pp. 9–17.

[31] H. Menon, N. Jain, G. Zheng, and L. Kalé, "Automated load balancing invocation based on application characteristics," in *Proceedings of the International Conference on Cluster Computing (CLUSTER)*. Beijing, China: IEEE, 2012, pp. 373–381.

[32] H. Menon, "Adaptive load balancing for HPC applications," Ph.D. dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 2016.

APPENDIX A.
ML AND LBTEST PARAMETERS

ML training was done using Python 3.6.8 with modules `scikit-learn` (v0.21.3) and `yellowbrick` (v1.0.post1). Use used the default parameters for GB, GNB, LDA and QDA. The parameters used for the other ML algorithms were the following: 1) KNN: number of neighbors = 3; 2) SVM: kernel = rbf, penalty parameter C = 0.025, probability estimated enabled; 3) DT: maximum depth = 20; 4) RF: number of estimators = 50, maximum depth = 20; and AB: number of estimators = 80.

Table II summarizes the parameters used in LBTest. All experiments were run with a *print frequency* of 1.

Table II
LBTEST PARAMETERS AND TIMES WITHOUT LB PER EXPERIMENT

Experiment Name	# Elem.	# Iter.	LB Freq.	Min. Task Time (μs)	Max. Task Time (μs)	Exec. Time Without LB (s)
ring-1	5400	2000	300	1	4000	205.82
ring-2	4400	501	100	1	4000	94.46
ring-3	1200	1001	250	1	4000	54.37
ring-4	500	101	100	1000	3100	3.73
mesh2d-1	6400	101	100	100	5000	33.69
mesh2d-2	5000	3001	200	1	5000	815.20
mesh2d-3	3600	601	200	1	4000	93.30
mesh2d-4	5400	601	100	1	5000	185.19
mesh3d-1	9200	601	300	1	4000	238.18
mesh3d-2	5400	2000	300	1	4000	509.40
mesh3d-3	4500	601	150	1000	3000	194.62
mesh3d-4	6600	601	100	1000	5000	355.98