Predicting Runtime in HPC Environments for an Efficient Use of Computational Resources

Mariza Ferro[1], Vinicius P. Klôh[1], Matheus Gritz[1], Vitor de Sá[1], Bruno Schulze[1]

[1]Grupo de Computação Científica Distribuída (ComCiDis),
Laboratório Nacional de Computação Científica (LNCC)
Av. Getúlio Vargas, 333 – 25651-075
Petrópolis – RJ – Brazil

Abstract. *Understanding the computational impact of scientific applications on computational architectures through runtime should guide the use of computational resources in high-performance computing systems. In this work, we propose an analysis of Machine Learning (ML) algorithms to gather knowledge about the performance of these applications through hardware events and derived performance metrics. Nine NAS benchmarks were executed and the hardware events were collected. These experimental results were used to train a Neural Network, a Decision Tree Regressor and a Linear Regression focusing on predicting the runtime of scientific applications according to the performance metrics.*

1. Introduction

High-Performance Computing (HPC) has become fundamental to discover new knowledge, and therefore its use has become crucial to scientific research across many research domains. Despite the impressive advances, several areas of science are still too complex for the available resources and should benefit from the HPC growth, as expected for the next generation of exascale supercomputing. On the other hand, the increase of computational capability increases the need for efficient use of computational resources, which will be even more representative of the upcoming exascale. Thus, studies search for approaches to increase scientific applications' performance and the better use of computational resources. One proposal is the use of autonomic techniques that allow the best resource allocation for applications [Klôh et al. 2020]. The scientific problem's requirements should guide the orchestration of different techniques and mechanisms, improving the performance, and using the computational resources efficiently.

To reach this, predicting the applications' runtime is one of the first steps, enabling, for example, the best job scheduling. Moreover, the runtime prediction is very useful for HPC users when submitting a job. Despite the complexity of scientific applications and the HPC systems, these users usually are requested to estimate their jobs' runtime for system scheduling. When jobs are under or overestimated could figure in a costly situation for users, systems, and the environment since they waste time and consumed energy [Guo et al. 2018].

Therefore, this work aims to predict the runtime of scientific applications to develop an application-aware autonomous framework capable of allocating resources and jobs in different HPC architectures to use computational resources efficiently. Also, specific contributions are identifying parameters that impact performance, shared in different architectures, and understanding how they impact runtime. For this, we propose an approach using performance counters and derived performance metrics as features in Machine Learning (ML) models. Three supervised ML algorithms are evaluated to these objectives using a Neural Network (NN), a Decision Tree Regressor (DTR), and a traditional Linear Regression (LR) for comparison with the others. The results show that the use of performance counters and derived metrics proved to be effective in predicting the performance of applications and allowed to identify relationships of features and how they influence the runtime in four different architectures.

This work is organized as follows: in Section 2 are presented the background and the related works. In Section 3 are presented the methodology for the experiments, details about the applications and collected parameters. The experiments and results are in Section 4 and, the final considerations in Section 5.

2. Related Work

As mentioned in Section 1, the use of autonomic techniques will be significant to allow a better understanding of how an application's requirements affect the runtime and how to effectively use this data to model a scheduling framework in future works. Several works make use of predictive models to improve specific applications such as [Souza et al. 2019, Siegmund et al. 2015, Balladini et al. 2014, Wu et al. 2016] and are classified according to their approaches (analytical or empirical) [Kaltenecker 2016].

Many researchers explore the possibility of using Machine Learning techniques to obtain this kind of knowledge. Considering the different applications performed in HPC environments and the complexity of these media, the amount of resources that need to be analyzed grows exponentially, making analytical approaches unfeasible. Thus, with the advancement of ML techniques, the use of empirical approach has increased considerably and has been used in several different works to predict the performance of different HPC systems.

In [Martínez et al. 2017], the authors propose a specific ML approach, adapted to stencil computing using *Support Vector Machines* models. They conclude that its performance can be predicted with high accuracy due to appropriate hardware counters. In [Malakar et al. 2018] the authors present a benchmark study evaluating eleven ML techniques for modeling the performance of four representative scientific applications. They assess the impact of feature engineering, the size of the training set, and the extrapolation on the prediction accuracy on four different architectures. In [Guo et al. 2018] and [Tanash et al. 2019], authors use system reports that include details such as the percentage of resource usage and number of CPUs in use as the basis for a relatively accurate runtime prediction model with a small number of parameters for model training. The study in [Amaris et al. 2016] compares three different ML techniques with an analytical model to predict the performance of nine applications executed over nine distinct GPUs. All these studies trained the models individually for each architecture but did not have a generic model for different systems like in this work. In [Wu et al. 2020] authors used MuMMI and 10 ML methods to model, predict and compare the performance and power of two algorithm-based fault-tolerant (FTLA and HDC) on three different architectures. With MuMMI they instrument the codes to collect performance data, power data, and performance counters. Based on what-if prediction system, they identify the most significant performance counters for potential optimization efforts associated with the application characteristics and the target architectures, and they predict theoretical outcomes of the potential optimizations. Comparing the prediction accuracy using MuMMI with that using 10 ML methods, they observe that MuMMI resulted in more accurate prediction in both performance and power. In [Masouros et al. 2019] is proposed Rusty, a monitoring framework that leverages Long Short-Term Memory (LSTM) networks that provide fast and accurate interference aware predictability. They evaluate Rusty's efficiency on workloads derived from the scikit-learn library and the Cloudsuite benchmarking suite. Experiments focused on predicting the IPC and the LLC misses of the core executing the target workload and the energy consumption of the respective socket. They analyzed and explored several schemes of LSTMs, concluding to a generic efficient LSTM architecture in terms of accuracy, responsiveness to runtime constraints, and computational cost. In addition, they demonstrated the advantage of LSTM networks over two simpler approaches, LR and MLP. [Lewis et al. 2020] demonstrates a method for predicting application runtimes using logs collected from jobs run on Argonne's Mira supercomputer. The authors evaluated the effectiveness of runtime prediction using information available only before a queued job is run to predict how long the application will run. To accomplish this task, they used 12 hardware performance counters as features and trained an XGBoost regression model. Also, this work investigated how effective hardware performance counters are in classifying application runs based on different resource consumption patterns. Applications are defined as either file I/O-, computation-, or MPI communication-intensive by looking at the ratio of time spent by an application on each category relative to that application's total runtime. This investigation used the same hardware performance counters as the training features for an XGBoost classification model.

The work of [Johnston 2019] proposes that it is necessary to make accurate performance predictions for workloads on different computing devices to support efficient scheduling on HPC. They collect a set of 28 architecture-independent features in 4 categories (parallelism, diversity of instructions, memory, and control), measured by counting targets collect while the application was executing. These features were used to create the prediction model using Random Forest and identify the applications' requirements.

In previous work [Klôh et al. 2019], the authors of this work presented the proposal of the autonomic framework where this work is inserted, explored the use of hardware counters as the initial steps on using DTR models to predict runtime, and gather knowledge on hardware counters relevant to it. This work expands on these concepts by introducing a MLP, different architectures, and evaluating the derived performance metrics as part of a new dataset using feature construction.

3. Methodology of Experiments

The methodology for the execution of the experiments consisted of the following steps: a set of nine OpenMP applications were prepared; each application was executed 30 times across predefined architectures, workloads, and thread interval; for each execution, a set of performance counters were monitored and collected. Section 3.1 details these architectures, applications and counters.

In sequence, the data obtained from this first monitoring and data collection phase were parsed, cleaned, and stored for use in the ML experiments. The objectives are: i) identify which parameters are most relevant to represent the performance of applications and find a set of relevant parameters common to different HPC architectures; ii) contribute models for runtime prediction for the different architectures used (individuals and multi-architecture), which are relevant tasks in research for an autonomous framework; iii) still, it is objective to understand which parameters impact the runtime during the execution of scientific applications. Section 3.2 provides an outline of the ML models and how they were trained and tested.

3.1. Experimental Setup: Dataset Collection

The first experimental setup consists of a series of OpenMP applications from the NAS Parallel Benchmark suite [Bailey et al. 1991], which derived from Computational Fluid Dynamics (CFD). They were designed to help evaluate the performance of parallel supercomputers, including a range of problem sizes [1]. Table 1 presents the applications, problem sizes and the number of elements for each size.

Table 1. Applications, problem sizes and number of elements for each application.

Applications	**Problem Sizes**			
	W	**A**	**B**	**C**
	Number of elements			
BT - Block Tri-diagonal solver SP - Scalar Penta-diagonal solver LU - Lower-Upper Gauss-Seidel solver	24^3	64^3	102^3	162^3
CG - Conjugate Gradient	7000	14000	75000	150000
EP - Embarrassingly Parallel	2^{25}	2^{28}	2^{30}	2^{32}
FT - discrete 3D fast Fourier Transform	-	$128^2 \times 32$	$256^2 \times 128$	512×256^2
IS - Integer Sort	2^{20}	2^{23}	2^{25}	-
MG - Multi-Grid	128^3	256^3	256^3	512^3
UA - Unstructured Adaptive mesh	700	2400	8800	33500

The experiments were performed using the four architectures presented in Table 2. Each application was executed 30 times, to ensure the consistency of the results[2], across a predefined thread interval defined by the number of processing cores (1, 2, 4, 6, 8, 10, 12 for Turing, SimCluster and Beholder and this same interval plus 16, 32 and 64 for AMD). The different number of threads were used to understand how this affect the performance of applications and to observe the difference in the runtime and how it impacts on the performance counters. These counters were collected using *perf* tool, available in the *Linux Kernel* since version 2.6.31. It performs counts of events accessing the model-specific registers directly, with zero (0.00) overhead on running applications.

From the measured performance counters, some derived performance metrics were built, and used to feature construction (presented in Table 3 with $*$). These derived metrics were defined based on the work of [Klôh et al. 2020], because they allow observing the performance behavior of the applications on the use of computational resources, which was not possible with the use of performance counters purely. The instructions_per_cycle (IPC) is the ratio of Instructions to total Cycles. The metrics *L1 load*

[1]The sizes of NAS applications vary from S, W, A to F, as problem sizes increase. A complete description of all sizes is available at `https://www.nas.nasa.gov/publications/npb.html`

[2]We found that the variation of the application runtime are very small (less than 1%).

Table 2. Architectures used in this work.

	AMD	Beholder	SimCluster	Turing
Processor	Opteron 6376	Xeon CPU X5650	Xeon CPU X5650	Core i7-8700 CPU
cpu_freq (GHz)	1.4	2.66	2.66	3.2
Sockets	4	2	2	1
cores_per_socket	8	6	6	6
threads_per_core	2	1	1	2
available_threads	64	12	12	12
cache_size_L1 (MB)	16	32	32	32
cache_size_L2 (MB)	2048	256	256	256
cache_size_L3 (MB)	6144	12288	12288	12288
RAM_size (GB)	128	24	24	64
RAM_freq (GHz)	1.6	1.3	1.3	2.66
Total width	72	72	72	64
Data width	64	64	64	64

ratio and *branch miss predict ratio* provide the ratio between hits and misses. The *PTI* (Per Thousand Instructions rate metric) provides the performance relative to the computational resource per thousand of completed instructions. These ratios and rates make it possible to measure the performance of applications in different computational resources, in addition to being easy to measure metrics, since they use performance counters.

Table 3. Performance counters and derived performance metrics used as features of ML tasks.

Feature	**Description**
instructions	# of instructions sent to the CPU
cycles	# of CPU cycles completed
cpu_migrations	# of times that the processes moved to another CPU
branches	# of conditional instructions that alter the flow of a process
branch_misses	# of times the CPU failed to predict the outcome of a branch
context_switches	# of times a process was halted so another could be run
cache_misses	# of times something was not found in the cache
L1_dcache_stores	# of times data was stored in the Level 1 dcache
L1_dcache_loads	# of times data was loaded from the Level 1 dcache
L1_dcache_load_misses	# of times data was not found in the Level 1 dcache
LLC_stores	# of times that data was stored in the Last Level cache
LLC_loads	# of times data was loaded from the Last Level cache
LLC_load_misses	# of times data was not found in the Last Level cache
instructions_per_cycle$*$	instructions / cycles
loads_PTI$*$	LLC-loads / (instructions$\times 10^{-3}$)
stores_PTI$*$	LLC-stores / (instructions$\times 10^{-3}$)
L1_load_ratio$*$	L1-dcache-loads / L1-dcache-load-misses
L1_load_rate_PTI$*$	L1-dcache-load-misses / (instructions$\times 10^{-3}$)
branch_miss_predicted_ratio$*$	branch-misses / branch
branch_miss_predicted_rate_PTI$*$	branch-misses / (instructions$\times 10^{-3}$)
cache_misses_PTI$*$	cache-misses / (instructions$\times 10^{-3}$)
Seconds	the application's runtime and the target of this work

3.2. Experimental Setup: Machine Learning

All this information is then used to compose five datasets, one for each architecture and another one with data from all the architectures named Multi-Archit (Table 4). Thus, it was possible to create prediction models for each architecture individually, besides investigating which parameters have more impact on the runtime for each architecture. Table 4 presents the summary of each dataset, with the total number of examples (# Examples), the number of examples for train and test (train/test) and the number and types of features (# Features: numeric or continuous - num. / nom.). Each subset of data was divided into 80% for training and 20% for testing.

Table 4. Summary of each dataset for runtime prediction. The 23 features are each row of Feature column in the Table 3.

Architecture	Examples (train / test)	Features (num. / nom.)
Turing	7305 (5844 / 1461)	23 (23 / 0)
AMD	10800 (8640 / 2160)	23 (23 / 0)
Beholder	7440 (5952 / 1488)	23 (23 / 0)
Simcluster	7517 (6013 / 1504)	23 (23 / 0)
Multi-Archit	33062 (26449 / 6613)	23 (23 / 0)

The experiments were performed according to the learning tasks, that is, the objective to be reached with ML. For this, three supervised ML algorithms were used: Decision Tree for regression learning task in order to reach the objective (i) and (iii); LR and NN model with MLP architecture to objectives (ii) and (iii).

Before the training and test, we did the data preprocessing. Considering that some features have values that can vary in scale among single-digit numbers and trillions, normalizing the data was necessary to keep the training performance viable and avoid issues like overfitting and underfitting. Using *MinMaxScaler*[3] method, the data was then normalized between 0 and 1 while still keeping a proportional distance between each value. Tree-based models, like DTR, are usually not dependent on scaling, but non-tree models models such as MLP and LR, are often hugely dependent on it. This can be useful and necessary for some ML models like the MLP, where the back-propagation can be more stable and even faster when input features are min-max scaled. However, one important thing to keep in mind when using the *MinMaxScaler* is that it is highly influenced by the maximum and minimum values in our data so if our data contains outliers it is going to be biased. It was found that the examples for the largest workload size (D) were adding this problem with outliers. Therefore, the regression models presented so far are limited only to the other workloads (W-C) - Table 1.

In sequence, a Decision Tree Regressor (DTR) model using an optimized version of the CART algorithm available in the *Scikit-learn* [Pedregosa et al. 2011] library. The DTR models were trained with hyperparameter max depth = 5, based on results obtained in previous works [Gritz et al. 2019, Klôh et al. 2020]. As noted in these works, the error metrics of the DTR models decreased according to depth, but the generalization capacity worsens. The main interest in this DTR model is to gather further knowledge of the data from the perspective of a non-black-box ML technique.

Linear Regression model checks for the existence of a relationship between, at least, two variables. That is, given x and y, how much does x explain y. The model performs a forecast by calculating a weighted sum of the input characteristics plus a con-

[3]https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

stant known as the linear coefficient. Although it is considered a simple algorithm, LR is trained to compare against other ML results.

The Neural Network (NN) used in this work is a Multilayer Perceptron (MLP), trained using Tensorflow[4] and Keras[5]. The final topology contains an Input Signal that refers to the number of features from Table 3 minus the target, an Output Layer with a single neuron as there is only one target feature (*runtime*) and two Hidden Layers with 26 and 18 neurons, respectively. We also performed experiments including more than two hidden layers. However, increasing the number of layers did not improve the results, and it raised the time to train the model while also risking overfitting it. Also, Sigmoid activation function was used for both the Input Signals and Hidden layer. However, due to the number of neurons and features in the dataset, the vanishing gradient problem became a concern [Nwankpa et al. 2018]. Predictions of longer runtime were severely impacted by it, resulting in very high values for all error metrics. So the Rectified Linear Unit (ReLU) activation function was evaluated as it is known to prevent this issue and is slightly more accurate than Sigmoid [Hara et al. 2015]. The Linear activation function was used for the Output Layer. Also, the MLP models were compiled using the *tf.keras.optimizers.Adam* optimization function. The number of epochs was defined as 1000 with a batch size of 50, meaning a total of 117 batches required to complete a single epoch in the Turing architecture, for example.

Two error metrics were used in this work in order to evaluate how effective the regression models are and how close the real values are to the predictions: the average of the difference between the original values and the predicted values (Mean Absolute Error - MAE); the Root Mean Squared Error (RMSE) that measures the average magnitude of the error. It's the square root of the average of squared differences between prediction and actual observation.

4. Experiments and Results

This section presents the experiments and discusses the results using ML techniques developed with the objectives of: predicting the applications' runtime and to understand which parameters have more impact on it during the execution of this set of scientific applications. Also, to define a set of features (performance counter and derived metrics), common in different architectures, which are significant to predict the runtime of the applications. Finally, the development of predictive models that could contribute to the future autonomic system.

Firstly, DTR and MLP models were trained and tested for each architecture to learn about each architecture individually and provide runtime prediction models. After, multi-architecture models were built. We present the results for all architectures, individually and together (multi-architecture). Then, a traditional regression model was constructed with the Linear Regression technique for comparison with the other two.

Decision Tree Regressor:

We used the DTR model as an explainable ML technique to verify the importance of each feature in each of the architectures and if there is any common relation of these

[4]https://www.tensorflow.org/
[5]https://keras.io/

features in the different architectures. For decision tree models, each node in the tree represents a test on an attribute, and the branches of that node represent a test with the features' values. The algorithm uses a divide-and-conquer approach to build the model, and the feature that "best" discriminates the examples according to the label is used at the tree's root. The DTR model for each architecture selected different roots for the tree as:

- Turing: *L1_dcache_loads*;
- AMD: *LLC_stores*;
- Beholder: *L1_dcache_stores*;
- SimCluster: *cycles*;
- Multi-architecture: *instructions*;

As can be seen, the most important feature (performance counter) changes according to the architecture, showing that different performance parameters influence the runtime when the applications are running on different architectures. It is worth mentioning that this does not imply re-training the model for each architecture when the model will be in use. However, this set of parameters must be collected for all architectures. Although we are searching for a small set of features, this is not a problem since they are present in all architectures (limited for these four evaluated in this work). All the architectures' models selected some features coming from performance counters: *cycles*, *L1_dcache_loads*, *LLC_stores*. From derived metrics was selected by all: *instructions_per_cycle*. However, some features were not selected in any model: *branch_misses* (used only on the derived metric *branch_miss_rate_PTI*), *LLC_load_misses, L1_load_rate_PTI, branch_miss_predicted_ratio*.

For the following experiments, the dataset in Table 3 was maintained, and no features were removed for the construction of the models.

Multilayer Perceptron Neural Network:

Figure 1 details the error progression of the training (red line) and validation process (blue line) on Turing architecture. For each epoch, the test dataset is validated using the current state of the model in order to evaluate if the NN is improving its prediction capabilities. It is possible to see that along the epochs, the MAE is closer to zero (the same for RMSE errors). Through the process, MAE improves the most before epoch 150, and then it halts before epoch 300 due to the early stopping rule implemented in the model to prevent overfitting.

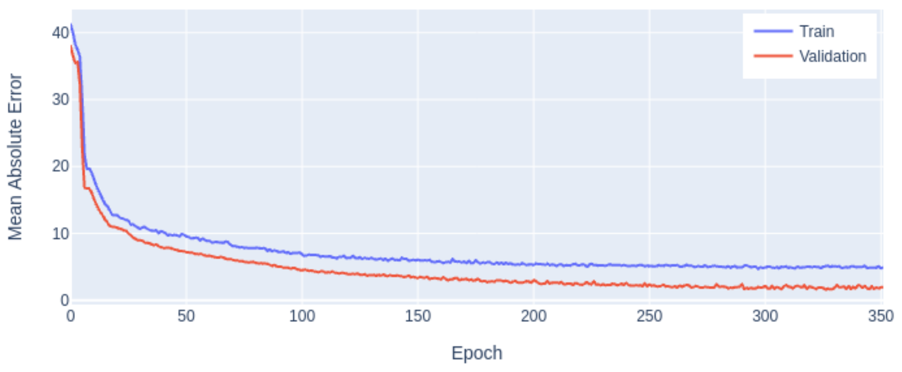

Figure 1. MAE progression on Turing.

These results are also observed for the other architectures, and for space limitations' reason, we are showing only the graph for Turing. However, all MAE and RMSE results for DTR and MLP are showed in Table 6 and could be compared with LR.

Table 5 presents a comparison among the real values and the values predicted by the NN model , DTR model and LR in the Turing architecture. Table shows only the first and last three examples of the test set and the values predicted by the ML models for Turing. Despite having a slightly elevated RMSE (Table 6), the NN models are capable of predicting the runtime of the scientific applications with reasonable accuracy as most of the predicted values are very close to the real values in the original datasets (the same is observed for other architectures). That shows that NN could be a well-suited algorithm to predict the runtime of scientific applications based on performance counters. The DTR models are also capable of accurate predictions but score significantly higher in the error metrics, and the LR model was the one that presented the worst runtime prediction, as detailed in Table 6. This is easily explained by the complexity of the dataset and the simplest model that LR can create.

Table 5. Real vs predicted values on Turing.

Test Examples	Real	NN	DTR	LR
1	2.73	3.49	3.50	-6.19
2	209.2	199.81	211.60	302.92
3	31.3	33.11	32.46	33.03
...	...	...	...	...
1459	1.96	3.11	3.50	-8.50
1460	7.33	9.03	8.21	-10.35
1461	3.18	4.46	3.50	11.67

In Table 6 are presented the metrics for each constructed models for all regression models and architectures. The closer they are to zero, the more accurate the models are. The NN predicted values are considerably closer to the real runtime values, while the metrics for the DTR model other than the MAE are generally higher. Thus, the NN models presented the best results, followed by the DTR models.

Table 6. Error metrics for runtime prediction.

		ML Techniques		
Architecture	Metrics	NN	DTR	LR
Turing	MAE	1.89	3.2	15.54
	RMSE	3.98	9.54	26.79
AMD	MAE	4.55	8.13	35.95
	RMSE	7.64	15.44	71.14
Beholder	MAE	3.26	9.30	30.02
	RMSE	5.74	16.52	52.08
SimCluster	MAE	3.29	5.76	30.80
	RMSE	4.86	9.59	54.74
Multi-Archit.	MAE	5.36	9.68	33.52
	RMSE	9.69	18.24	63.69

The NN models were more accurate than DTR in all the experiments, with lower error for all metrics evaluated. However, the DTR models provided necessary knowledge regarding the ranges of values of features, how these values impact the runtime and the feature importance. These ranges of values add knowledge about the computational requirements of scientific applications and their performance in using hardware resources.

Therefore, the results of these experiments contribute to selecting the relevant parameters to be collected when the objective is to predict the runtime in different computational architectures. In addition, regression models were constructed and compared using different ML techniques, which proved effective for runtime prediction and could be used in developing the autonomic system.

It is important to note that the proposed autonomic system is still under development [Klôh et al. 2020]. However, in practice, all these contributions build a knowledge base that allows its future development. The definition of the set of parameters will allow the monitoring of the performance behavior of applications in different architectures and how different factors can change the performance. After, the developed regression model will be used to orchestrate the use of computational resources efficiently and be aware of scientific applications' requirements. We aim to implement this integrated with the OpenMP runtime library because this one could be dynamically linked to applications and so, the framework could be entirely transparent to user applications. Afterward, the regression models could be used to predict the runtime of these applications. Thus, finding the best experiment configuration that allows the balance between runtime and energy consumption, meeting the requirements of the applications.

5. Closing considerations

In this work, nine NAS applications of the CFD area were executed in four computational architectures. This work's main objective was to predict the runtime of these applications using a NN and a DTR while also extracting knowledge of what features influence the runtime of an application. Using the supervised ML technique with the DTR allowed identifying relationships of features and how they influence the runtime. All developed models were tested with data never seen during the training phase. In general, suitable results were obtained. Overall, a high number of instructions performed and read and write to Level 1 cache proved to impact the runtime significantly, closely followed by Last Level Cache features. These can be observed by the features on the root of the trees (Turing - *L1_dcache_loads*, AMD - *LLC_stores* and Beholder - *L1_dcache_stores*) and used by all the models (*L1_dcache_loads*, *LLC_stores*)

The use of performance counters and derived metrics proved to be effective in predicting the performance of applications. It should contribute to the development of autonomous systems, aware of the computational requirements of the applications, which can predict the runtime of applications and guide the orchestration of different techniques and mechanisms.

Although the models developed in this work are restricted to the four architectures used and nine different applications in OpenMP, the most important is the methodology presented here that can be extended to other HPC architectures and applications. This is because performance counters available across different HPC architectures were evaluated and shown to be effective for this prediction. This also allows, in future works, to characterize the requirements of scientific applications when executed on different HPC architectures, evaluate energy metrics on architectures that support them, and explore the possibility of developing a job scheduler using the data and knowledge attained.

For reproducibility, the datasets and codes used are available at `https://github.com/ViniciusPrataKloh/Dissertation_Results_Models`.

In future works, it will be investigated if the derived performance metrics are enough to characterize the performance of the applications regarding the use of computational resources. This could be useful to compose a small and more generic set of attributes, which are sufficient for the modeling and prediction tasks of the runtime for different architectures. Still, another group of experiments is also being executed with data obtained in ARM-based and other x86 architectures to understand better the influence that the collected parameters have on the runtime in architectures different from those used in this work. The possibility of using these results as part of a job and application scheduler as part of an autonomous framework is also being currently explored and evaluated, alongside a version of this model tuned for energy consumption.

Acknowledgments

Authors would like to thank the financial support from LNCC/MCTI, CNPq and Faperj. This work is also part of the collaborative projects CLIMAT-AmSud GreenAI (21-CLIMAT-07), and *Towards a Sustainable Artificial Intelligence - SusAIn* Associate Team INRIA-LNCC.

References

Amaris, M., de Camargo, R. Y., Dyab, M., Goldman, A., and Trystram, D. (2016). A comparison of gpu execution time prediction using machine learning and analytical modeling. In *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, pages 326–333. IEEE.

Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., et al. (1991). The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73.

Balladini, J., Morán, M., Rexachs del Rosario, D., et al. (2014). Metodología para predecir el consumo energético de checkpoints en sistemas de hpc. In *XX Congreso Argentino de Ciencias de la Computación (Buenos Aires, 2014).*

Gritz, M., Silva, G., Klôh, V., Schulze, B., and Ferro, M. (2019). Towards an autonomous framework for hpc optimization: A study of performance prediction using hardware counters and machine learning. *XIX Simpósio de Pesquisa Operacional e Logística da Marinha*.

Guo, J., Nomura, A., Barton, R., Haoyu, Z., and Matsuoka, S. (2018). *Machine Learning Predictions for Underestimation of Job Runtime on HPC System*, pages 179–198.

Hara, K., Saito, D., and Shouno, H. (2015). Analysis of function of rectified linear unit used in deep learning. In *2015 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.

Johnston, B. (2019). *Characterizing and Predicting Scientific Workloads for Heterogeneous Computing Systems*. PhD thesis.

Kaltenecker, C. (2016). Comparison of analytical and empirical performance models: A case study on multigrid systems. *Masterthesis, University of Passau, Germany*, page 10.

Klôh, V., Gritz, M., Schulze, B., and Ferro, M. (2019). Towards an autonomous framework for hpc optimization: Using machine learning for energy and performance modeling. In *Anais Principais do XX Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 438–445. SBC.

Klôh, V., Schulze, B., and Ferro, M. (2020). Use of machine learning for improvements in performance and energy consumption in hpc systems. Master's thesis, National Laboratory for Scientific Computing.

Lewis, R. D., Liu, Z., Kettimuthu, R., and Papka, M. E. (2020). Log-based identification, classification, and behavor prediction of hpc applications. In *In HPCSYSPROS20: HPC System Professionals Workshop*, Atlanta, GA.

Malakar, P., Balaprakash, P., Vishwanath, V., Morozov, V., and Kumaran, K. (2018). Benchmarking machine learning methods for performance modeling of scientific applications. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 33–44. IEEE.

Martínez, V., Dupros, F., Castro, M., and Navaux, P. (2017). Performance improvement of stencil computations for multi-core architectures based on machine learning. *Procedia Computer Science*, 108:305–314.

Masouros, D., Xydis, S., and Soudris, D. (2019). Rusty: Runtime system predictability leveraging lstm neural networks. *IEEE Computer Architecture Letters*, PP:1–1.

Nwankpa, C., Ijomah, W., Gachagan, A., and Marshall, S. (2018). Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378*.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Siegmund, N., Grebhahn, A., Apel, S., and Kästner, C. (2015). Performance-influence models for highly configurable systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 284–294. ACM.

Souza, A., Rezaei, M., Laure, E., and Tordsson, J. (2019). Hybrid resource management for hpc and data intensive workloads. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 399–409.

Tanash, M., Dunn, B., Andresen, D., Hsu, W., Yang, H., and Okanlawon, A. (2019). Improving hpc system performance by predicting job resources via supervised machine learning. pages 1–8.

Wu, X., Taylor, V., Cook, J., and Mucci, P. J. (2016). Using performance-power modeling to improve energy efficiency of hpc applications. *Computer*, 49(10):20–29.

Wu, X., Taylor, V. E., and Lan, Z. (2020). Performance and power modeling and prediction using mummi and ten machine learning methods. *CoRR*, abs/2011.06655.