Greening AI: A Framework for Energy-Aware Resource Allocation of ML Training Jobs with Performance Guarantees

Roberto Sala, Federica Filippini, Danilo Ardagna,
Daniele Lezzi, Francesc Lordan, Patrick Thiem

Abstract The rapid expansion of Machine Learning (ML) and Artificial Intelligence (AI) has profoundly influenced the technological landscape, reshaping various industries and applications. This surge in computational demands has led to the widespread adoption of Cloud data centers, crucial for supporting the storage and processing requirements of these advanced technologies. However, this expansion poses significant challenges, particularly in terms of energy consumption and associated carbon emissions. As the reliance on cloud data centers intensifies, there is a growing concern about the environmental impact, necessitating innovative solutions to enhance energy efficiency and reduce the ecological footprint of these computational infrastructures. This paper focuses on addressing the challenges linked to training ML and AI applications, emphasizing the importance of energy-efficient solutions. The proposed framework integrates components from the AI-SPRINT project toolchain, such as Krake, Space4AI-R, and PyCOMPSs. Our reference application involves training a Random Forest model for electrocardiogram classification, profiling available resources to obtain a performance model able to predict the training time, and dynamically migrating the workload to sites with cleaner energy sources providing guarantees on the training process due date. Results demonstrate the framework's capacity to estimate execution time and resource requirements with low error, highlighting its potential for establishing an environmentally sustainable AI ecosystem.

Roberto Sala, Federica Filippini, Danilo Ardagna
Politecnico di Milano, e-mail: name.lastname@polimi.it

Daniele Lezzi, Francesc Lordan
Barcelona Supercomputing Center, e-mail: name.lastname@bsc.es

Patrick Thiem
Cloud and Heat, e-mail: patrick.thiem@cloudandheat.com

1 Introduction

Training ML and, in particular, AI applications is challenging. It demands large computational resources and powerful accelerators, e.g., GPUs and TPUs, resulting in large investment for organizations, especially those with large-scale ML infrastructure [1]. On the other hand, the AI training process is energy-intensive and associated with carbon emissions that significantly contribute to environmental degradation [2].

As ML and AI become more pervasive, there is a pressing need to adopt energy-efficient solutions both for algorithm implementation and hardware configuration [3]. The environmental impact of Cloud data centers has indeed become a significant concern, with global electricity usage surging to 200 Tera-Watt hours (TWh) in 2019 and 240-340 TWh in 2022 [4]. Cloud datacenters significantly contribute to carbon emissions, prompting global policymakers to establish ambitious targets, such as Net-Zero emissions by 2050 in countries like the UK, France, Denmark, and New Zealand. The Top500 list [5], a benchmark for supercomputing performance, is indeed complemented by the Green500 [6], emphasizing energy efficiency.

A strategic allocation of computational tasks to clusters exploiting sustainable energy, such as solar or wind power, is crucial, although challenges arise from the intermittent nature of renewable energy. Migrating AI workloads during periods of increased clean energy availability optimizes costs and fosters a more ecologically sustainable AI ecosystem, emphasizing the integration of environmental considerations into the ML infrastructure deployment.

The framework presented in this paper showcases results from integrating various components within the AI-SPRINT project toolchain [7]: Krake, SPACE4AI-R, and PyCOMPSs. Krake is an orchestrator engine that streamlines and manages containerized and virtualized workloads across diverse cloud platforms. SPACE4AI-R [8] is a runtime tool designed to tackle the Resource Selection and Component Placement problem efficiently. It aids in executing AI applications across a spectrum of Computing Continuum resources, spanning Edge devices, Cloud Virtual Machines, and Function as a Service configurations. PyCOMPSs [9] is a parallel programming framework for Python applications. It streamlines the development of computational workflows for distributed infrastructures with a programming model based on sequential development.

In our study, we focus on a PyCOMPSs application that trains a Random Forest model tailored for classifying electrocardiograms. This application is profiled on the existing resource infrastructure, collecting data to construct ML-based performance models used by SPACE4AI-R to identify the optimal configuration in terms of cost and maximum execution time. We consider a reference scenario where the application is initially deployed at site *A* and later migrated to a different site *B* when new resources powered with a cleaner energy source become available; the checkpointing features of PyCOMPSs allow transparently re-starting the computation from the last saved point on the new resources. The interaction between Krake and SPACE4AI-R allows determining the computational resources required on the new site to guarantee a user-specified maximum execution time, considering the

setup time at site B and the consequent migration. Additionally, the application runs periodically, allowing us to access logs from previous executions. We aim to ensure an upper bound on the execution time, which may vary for each run. This assumption is reasonable in the case of AI applications, given that models are regularly retrained to prevent model drift. The results show that our model can estimate both the execution time on the new cluster and the number of cores needed to satisfy the time constraint, with an error lower than 20%.

2 Related Work

In recent years, there has been widespread attention on energy consumption and efficiency in cloud data centers due to the exponential growth in the number and power of machines. For instance, [10] conducts a comprehensive analysis of power consumption characteristics in HPC workloads, examining them from the perspectives of systems, jobs, and users. Conversely, [11] surveys power conservation techniques at both hardware and software levels, emphasizing the importance of developing energy-efficient solutions for green IT.

The scheduling of jobs and orchestration of resources play a crucial role in reducing energy consumption and implementing green policies in HPC. In [12], the authors introduce the Energy-Aware Multi-Cluster scheduling policy (EAMC-policy), automating job placement and optimal clock frequency selection. This prioritizes a delicate balance among performance, energy consumption, and response time within heterogeneous environments. Some works propose Deep Reinforcement Learning (DRL) models for resource planning. [13] presents a partition-based task scheduling approach for efficiently managing heterogeneous resources. It preassigns tasks to partitions based on current conditions and allocates them to suitable servers, utilizing nonlinear regression for a precise energy consumption model without network fitting. On the other hand, [14] develops a scheduling policy incorporating workload shifting and cloud bursting within a geographically distributed hybrid multi-cloud environment, aiming to maximize renewable energy utilization and prevent deadline constraint violations. Supervised learning models are also employed to optimize CPU frequency selection during job execution, aiming to minimize energy consumption in HPC systems [15].

Finally, dynamic resource allocation methods are developed, as in [16, 17, 18]. [16] focuses on autonomously managing converged edge platforms to enable resource-efficient workload orchestration and promote environmental sustainability. The proposed solution emphasizes intelligent dynamic resource configuration in multi-tenant edge computing platforms, aiming to ensure Service Level Objectives (SLO) for each service while encouraging eco-friendly communication practices. Moreover, in [17], it is shown that the effectiveness of prevalent static power allocation strategies in value-based algorithms depends on the applied power constraint. The study reveals shortcomings in these static approaches, resulting in underutilized resources despite system oversubscription. To address this, the paper suggests a dy-

namic power management strategy for value-based algorithms, utilizing application power-performance models to reallocate power and optimize system productivity, resource utilization, and job completion rates. Finally, [18] focuses on the management of Deep Learning training applications executed in GPU-accelerated clusters. It proposes a stochastic scheduler that automatically selects the optimal type and amount of resources to be assigned to each training job over time, minimizing the energy execution costs and the penalties for due dates violations. The developed method considers the stochasticity in the applications training times and enables GPU space sharing to maximize the resources utilization for less demanding applications.

3 AI Training Job Energy Management Framework

In this section, we provide an overview of the AI-SPRINT tools we used for the creation and the training of our application. The PyCOMPSs application consists of a Random Forest model designed for the classification of Electrocardiograms (ECGs) into categories such as normal ECG, Atrial Fibrillation (AF), inconclusive, or noise. Specifically, the training process utilises a set of 15 distinct functions, identified by their IDs in PyCOMPSs log files. Each function is invoked a specific number of times during the training process. The coordination of the distributed execution of these interdependent functions is managed by the COMPSs framework. In the following we present PyCOMPSs (Section 3.1), the performance models (Section 3.2), Krake (Section 3.3) and SPACE4AI-R (Section 3.4).

3.1 PyCOMPSs

PyCOMPSs is a Python based programming framework aiming at easing the development of general-purpose applications targeting the Cloud-Edge-IoT Continuum. It is composed of a task-based programming model and of a runtime that orchestrates the execution of such tasks in a serverless manner on top of any distributed platform. PyCOMPSs applications are described as workflows composed of many tasks with data dependencies and, at runtime, the engine determines the best host where to run a task and handles all the necessary data movements to offload the task execution.

The PyCOMPSs runtime provide an automatic checkpointing mechanism. As the application runs, the runtime engine decides to copy some of the data values of the application. In the case of an application error or an abrupt end of the execution (the user decides to kill all the VMs), a future execution could resume the application execution from that point. Being able to save the state of the application at any point allows the migration of stateful executions to improve the execution time or the energy consumption of the execution. Besides the default mechanisms, the application

developer can request checkpointing all the values on demand adding a specific API call in the application or develop a customized mechanism implementing a simple interface with three methods.

3.2 Performance models

During the training of the application, PyCOMPSs produces a log which provides a snapshot of the training status, i.e. the number of executed, running and completed functions, to be extracted at each instant. Analysing the aforementioned logs, we developed the following model for estimating the residual execution time, which is also valid for estimating the total training time:

$$\tilde{T} = \sum_{f \in F} \left\lceil \frac{n_f}{cores} \right\rceil t_f \tag{1}$$

where n_f and t_f are, respectively, the number of times the single function (executed in a task) f have to be executed until the end of the training of the application and its average execution time, and F is the set of functions called during the entire execution. Here by function we mean the code executed by a COMPSs task. This formula is based on the fact that functions are executed in parallel, when multiple workers are available. The ratio $\left\lceil \frac{n_f}{cores} \right\rceil$ indicates the number of waves with which the single function is executed. This approximate model is accurate when the number of function calls is greater than the number of workers, which happens in general in HPC systems and for PyCOMPSs [19]. Since our objective is to estimate the residual execution time in unobserved configurations, the treatment of t_f largely influences this estimate. Therefore, we developed four different scenarios, which differ in the way the ”history”, i.e. the previous runs, and the current run-times are taken into account:

- *Naive* approach: if the individual function was used at least once in the current simulation, t_f is calculated on the current log. Otherwise, the one calculated on the history is used.
- *Mixed* approach: t_f is a weighted average between the history and the current execution, where the weights are given by the number of executions of the single functions versus their total number of executions.
- *Balanced* approach: t_f is a weighted average between the history and the current execution, where the weights are given by the step at which any migration occurs versus the total number of steps.
- *Full post-processing* approach: t_f is only computed on the history.

The results for the estimation of residual training times are shown in Section 4.2.

3.3 Krake

XXX Patrick please revise XXX

Krake is an orchestrator engine for containerised and virtualized workloads across distributed and heterogeneous cloud platforms. It creates a thin layer of aggregation on top of the different platforms (such as OpenStack, Kubernetes or OpenShift) and presents them through a single interface to the cloud user. The user's workloads are scheduled depending on both user requirements (hardware, latencies, cost) and platform characteristics (energy efficiency, load). Krake can be leveraged for a wide range of application scenarios such as central management of distributed compute capacities and application management in Edge Cloud infrastructures. In this paper settings, Krake performs PyCOMPSs training application migration across Kubernetes clusters that reside in different sites characterised, possibly, by different energy profiles. Furthermore, it ascertains the "best" level of resource usage based on user-defined parameters and re-evaluates the deployment periodically.

3.4 SPACE4AI-R

SPACE4AI-R[1] (System PerformAnce and Cost Evaluation on Cloud for AI applications Runtime) [8] is an optimization framework that deals with the runtime management of AI applications executed in the Computing Continuum. The SPACE4AI-R optimizer implements several heuristic algorithms to effectively tackle the Resource Selection and Component Placement problem. It determines the minimum-cost configuration that guarantees user-imposed Quality of Service constraints, in response to load variations or whenever a component migration determines the need to re-consider the current deployment.

In this work, among the available optimization algorithms, we used a dichotomous search function to determine the resources needed on the new site, following the migration. The motivations behind this choice are manifold. Firstly, in the regime of profiled configurations, the execution time decreases monotonically as the number of workers increases. In addition, the dichotomous function allows us to balance our two objectives, i.e., to minimise the cost, and therefore the number of used cores, while guaranteeing a global constraint on execution time.

4 Experimental Results

In this section, we present the experimental results of our PyCOMPSs application. In particular, Section 4.1 describes the setting from which we extracted the data used

[1] https://github.com/ai-sprint-eu-project/space4ai-r-optimizer

in our simulations which constitutes the application ”history”, Section 4.2 shows the related results, while in Section 4.3 we show the results of a real migration of our application from a site *A* to a different site *B*, to show the accuracy of our models.

4.1 Experimental setting

The PyCOMPSs application has been tested on a machine Intel(R) Xeon(R) CPU, 2.20 GHz, utilizing 1 to 10 VMs, each equipped with 4 VCPUs, 15 GB of memory, and 50 GB of storage size, operating on an Ubuntu OS. This involves considering configurations with 4, 8, 12, . . . , 40 cores, by varying the number of active workers. To ensure robustness, three independent runs were conducted for each setting. It is noteworthy that while the profiling campaign was executed on a single machine, the models are easily extendable to machines of different types, as indicated by the following results.

PyCOMPSs generates a log file that records scheduling, start, and finish times of the execution of each function, enabling the extrapolation of a snapshot of the training status at any moment. To simulate the partial execution, the log files were truncated after a fixed number of steps, from the total number of 11,655.

With regard to estimating the number of cores on the new site, following the migration, as discussed previously we perform a dichotomous search on all available configurations on the new machine, selecting the smallest number of workers that guarantees satisfaction of the global time constraint.

As introduced before, the PyCOMPSs runtime has the ability to save the state of the application. In this case, the application is started on a cluster and prepares a dataset, trains a Random Forest model with it and evaluates the accuracy of the resulting model. Checkpointing sets a tradeoff between the amount of data values saved and the amount of computation lost in the case of an error. To that end, several checkpointing mechanisms able to work with different granularity have been defined. The first mechanism, Instantiated Tasks (IT), decides to save the output values of a set of N tasks according to their creation order. Finished Tasks (FT), does the same as IT but considering the order in which tasks finish; the checkpointing will save all the values every N tasks finished. Finally, the third mechanism, Periodic Time (PT), triggers the saving of all the values periodically after N seconds.

Table 1 shows the different impact of the different checkpointing mechanisms on a PyCOMPSs execution and compares their execution time to an execution without checkpointing (NC).

Table 1 Execution times of running a complete execution of COMPSs when applying different checkpointing policies

No checkpoint	Instantiation (10 tasks)	Tasks Finished (10 tasks)	Task Periodic Time (15 seconds)
112.97s	144.58s	153.06s	145.58s

4.2 Simulation results

First, we analyse the accuracy of our model (1) in estimating the residual training time of our application, in the four different scenarios described in Section 3.2, every 100 steps of the execution. The error between the estimate and the actual residual time is calculated as follows:

$$\% \; error = \frac{residual_{estimated} - residual_{real}}{total_{real}} \tag{2}$$

This gives an non-biased evaluation of the error on the total execution time.

In order to validate our model, we first observed the predictive ability of the model with the same configuration between history and actual setting. Figures 1 and 2 show, respectively, the residual time estimated by our model in the four different scenarios and the relative error, calculated as in (2), with a 20-cores configuration.

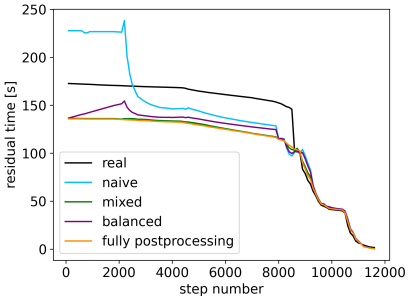

Fig. 1 Residual time for a 20-core configuration, using the same configuration history.

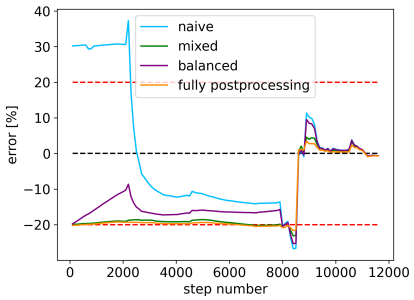

Fig. 2 Error for a 20-core configuration, using history with the same configuration.

The results show that all four models are able to estimate a residual time within 20% error. Only the naive model deviates more, especially in the early stages of training, due to the inaccurate estimation of the average execution time of those functions that have been performed a few times.

In a more general scenario, where a larger number of machines should be profiled, it is not possible to imagine performing such a large number of simulations for each machine. For this reason, we tried to estimate the residual time for all settings

considered, using data from the profiling of a run with 40 cores as history. This configuration takes into account the resource contention of the functions to be executed. The results for the 20-core configuration are shown in Figures 3 and 4.

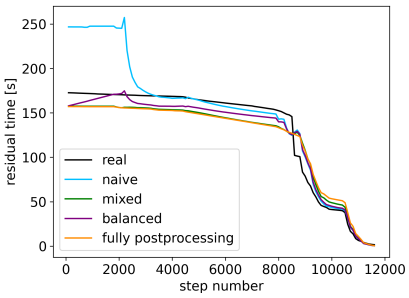

Fig. 3 Time remaining for a 20-core configuration, using the 40-core profiling data as history.

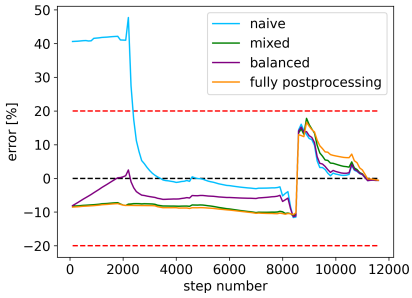

Fig. 4 Error for a 20-core configuration, using the 40-core profiling data as history.

Also, the model is sufficiently accurate for all the four scenarios but for the *Naive* one, observing an error of less than 20% with respect to the total run-time.

The second part of our analysis involves computing the number of cores on the new VM cluster required to meet a global time constraint, following the migration of the application. Building upon previous analyses, we have chosen the *Full post-processing* model to estimate the number of cores after migration, and the 40-core configuration as history. This choice for the treatment of t_f in (1) is motivated by the fact that the "Fully post-processing" model best estimates the average time remaining over the entire training process. Moreover, in case of migration, the number of cores usually needs to increase to recover the migration delay.

The estimation of the minimum number of cores required to satisfy the global time constraint is carried out by means of a dichotomous search between 1 and 40 cores, based on the model chosen for residual time estimation. The global time was set, for each simulation, equal to the real execution time on the given profiled configuration, which is considered as initial deployment, plus 60 seconds of migration. In this way, we expect an optimal number of cores following migration usually equal to the one pre-migration. Furthermore, the estimation of the number of cores post-migration was carried out by partitioning the logs into intervals corresponding to 5% of the steps and randomly extracting the point of migration. This estimation was performed 20 times for each interval, for each trial and for each setting for which we have profiling, for a total of 60 values for each interval and for each number of cores. The percentage of extra cores through the overall execution time is computed as follows:

$$\%\ cores_{extra} = \frac{cores_{estimated} - cores_{real}}{cores_{real}} \cdot \frac{T_{total} - T_{elapsed}}{T_{total}} \tag{3}$$

where T_{total} is the real total execution time in the starting configuration, while $T_{elapsed}$ is the elapsed time before migration. In this formula, the percentage of additional cores calculated for execution on the new VM cluster is normalized with respect to the remaining time. This normalization yields an estimate of the number of additional cores used during the entire training, and consequently, the actual energy costs due to model estimation errors. Figure 5 shows the percentage of extra cores estimated by our model for all the initial configurations.

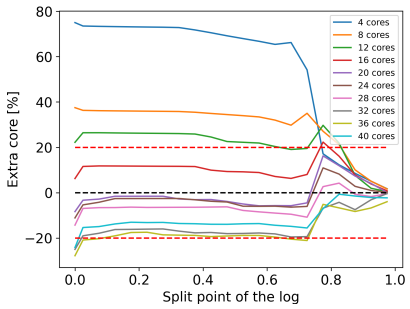

Fig. 5 Percentage of extra cores estimated.

Results show that for settings with more than 16 cores the estimate is sufficiently accurate, since the error remains below 20%. Note that for settings with only a few cores, the error is large in percentage terms, but not too large when we consider that an estimate of 75% extra cores on four cores results in an estimate of three additional cores.

In the final analysis, we look at the percentage of the number of violations of the overall time constraint and the percentage of time violation. In this case, the application migration point was randomly selected with respect to the entire application, the migration time was set at one minute and the global time constraint was randomly drawn between the average training execution times for the 32- and 4-core configurations. This experiment was carried out 1,000 times, for each available profiling run, and the results were averaged considering the initial number of cores. As for the estimation of the possible violation, the number of post-migration cores was calculated considering the 40-core profiling data, while the "real" post-migration residual time is calculated through our model on the actual number of cores we are using. This means that if initially the machine is using 40 cores and the dichotomic search function predicts that 33 cores are needed, the estimate of the "real" residual time is made on the 36 cores data, i.e., the first available data approximating by excess.

The percentage violation time for the individual experiment is calculated as follows:

$$\%\ violation\ time = \begin{cases} 0.0, \text{ if } T_{max} \geq T_{elapsed} + T_{residual} + T_{migration} \\ (T_{elapsed} + T_{residual} + T_{migration} - T_{max})/T_{max}, \text{ otherwise} \end{cases} \quad (4)$$

where T_{max} is the global time constraint and $T_{migration}$ is the migration time. Figure 6 illustrates the percentage of violations in relation to the total number of migrations, while Figure 7 depicts the percentage of time violations with respect to the overall time constraint, averaged over all experiments.

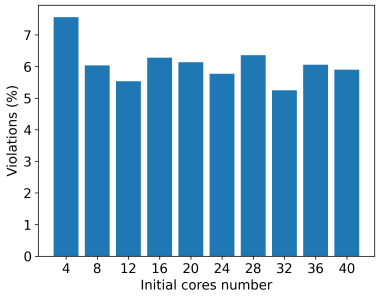

Fig. 6 Percentage number of violations.

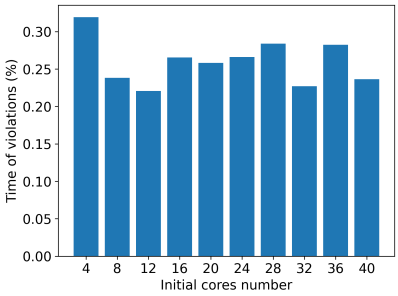

Fig. 7 Percentage violation time.

Results show that on average, about 6% of migrations involve a violation of the global time constraint, but the average time violation remains within 0.35%.

4.3 Analysis on a prototype environment

XXX Patrick please complete XXX

5 Conclusions

The challenge of energy consumption management in Cloud data centers is gaining significance as the computational power demands of ML and AI applications continue to rise. In this paper, we address a PyCOMPSs application, specifically focusing on the process of migrating computational resources from a site *A* to an alternative site *B* powered by cleaner energy sources. We have developed an integrated model within SPACE4AI-R, capable of predicting both the remaining execution time of the application, given the system configuration, and the number of cores required to meet a time constraint, using profiling data from the maximum number of cores configuration alone.

Experimental results indicate that our model can predict the remaining runtime with an error of less than 20% compared to the actual runtime. Furthermore, in estimating the resources needed at the post-migration site through dichotomous search, we observed that only 6% of migrations violate the global time constraint, with an average delay of 0.35% relative to the specified threshold.

Future works will concentrate on estimating the energy overhead resulting from migration to enhance the accuracy of our energy model.

Acknowledgements This work has been funded by the European Commission under the H2020 grant N. 101016577 AI-SPRINT: AI in Secure Privacy pReserving computINg conTinuum.

References

1. Ilager, S., Toosi, A. N., Jha, M. R., Brandic, I., & Buyya, R. A Data-driven Analysis of a Cloud Data Center: Statistical Characterization of Workload, Energy and Temperature. In 2023 IEEE/ACM 16th International Conference on Utility and Cloud Computing (UCC '23) (pp. 1-10), to appear
2. Lacoste, A., Luccioni, A., Schmidt, V., & Dandres, T. (2019). Quantifying the carbon emissions of machine learning. arXiv preprint arXiv:1910.09700.
3. Filippini, F., Ardagna, D., Lattuada, M., Amaldi, E., Riedl, M., Materka, K., ... & Cicala, M. (2021, August). ANDREAS: Artificial intelligence traiNing scheDuler foR accElerAted resource clusterS. In 2021 8th International Conference on Future Internet of Things and Cloud (FiCloud) (pp. 388-393). IEEE. doi: 10.1109/FiCloud49777.2021.00063.
4. https://www.iea.org/energy-system/buildings/data-centres-and-data-transmission-networks
5. https://www.top500.org
6. https://www.top500.org/lists/green500/
7. Sedghani, H., Ardagna, D., Matteucci, M., Fontana, G. A., Verticale, G., Amarilli, F., ... & Wawruch, K. (2021, July). Advancing design and runtime management of AI applications with AI-SPRINT (position paper). In 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC) (pp. 1455-1462). IEEE.
8. Filippini, F., Sedghani, H., & Ardagna, D. (2023). SPACE4AI-R: a Runtime Management Tool for AI Applications Component Placement and Resource Scaling in Computing Continua. In 2023 IEEE/ACM 16th International Conference on Utility and Cloud Computing (UCC '23) (pp. 1–7), to appear, isbn: 979-8-4007-0234-1/23/12. doi: 10.1145/3603166.3632560.
9. Badia, R. M., Conejero, J., Ejarque, J., Lezzi, D., Lordan, F., (2022). PyCOMPSs as an Instrument for Translational Computer Science. In Computing in Science & Engineering, 24(2), (pp. 66-82), doi: https://doi.org/10.1109/MCSE.2022.3152945.
10. Patel, T., Wagenhäuser, A., Eibel, C., Hönig, T., Zeiser, T., & Tiwari, D. (2020, May). What does power consumption behavior of hpc jobs reveal?: Demystifying, quantifying, and predicting power consumption characteristics. In 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS) (pp. 799-809). IEEE.
11. Thakkar, A., Chaudhari, K., & Shah, M. (2020). A comprehensive survey on energy-efficient power management techniques. Procedia Computer Science, 167, 1189-1199.
12. D'Amico, M., & Gonzalez, J. C. (2021). Energy hardware and workload aware job scheduling towards interconnected HPC environments. IEEE Transactions on Parallel and Distributed Systems.
13. Li, J., Zhang, X., Wei, Z., Wei, J., & Ji, Z. (2021). Energy-aware task scheduling optimization with deep reinforcement learning for large-scale heterogeneous systems. CCF Transactions on High Performance Computing, 3, 383-392.

14. Zhao, J., Rodríguez, M. A., & Buyya, R. (2021, September). A deep reinforcement learning approach to resource management in hybrid clouds harnessing renewable energy and task scheduling. In 2021 IEEE 14th International Conference on Cloud Computing (CLOUD) (pp. 240-249). IEEE.
15. Ozer, G., Garg, S., Davoudi, N., Poerwawinata, G., Maiterth, M., Netti, A., & Tafani, D. (2020). Towards a predictive energy model for HPC runtime systems using supervised learning. In Euro-Par 2019: Parallel Processing Workshops: Euro-Par 2019 International Workshops, Göttingen, Germany, August 26–30, 2019, Revised Selected Papers 25 (pp. 626-638). Springer International Publishing.
16. Guim, F., Metsch, T., Moustafa, H., Verrall, T., Carrera, D., Cadenelli, N., ... & Prats, R. G. (2021). Autonomous lifecycle management for resource-efficient workload orchestration for green edge computing. IEEE Transactions on Green Communications and Networking, 6(1), 571-582.
17. Kumbhare, N., Akoglu, A., Marathe, A., Hariri, S., & Abdulla, G. (2020). Dynamic power management for value-oriented schedulers in power-constrained HPC system. Parallel Computing, 99, 102686.
18. Filippini, F., Anselmi, J., Ardagna, D. & Gaujal, B., "A Stochastic Approach for Scheduling AI Training Jobs in GPU-based Systems" in IEEE Transactions on Cloud Computing, vol. , no. 01, pp. 1-17, 5555. doi: 10.1109/TCC.2023.3336540
19. Ataie, E., Evangelinou, A., Gianniti, E., & Ardagna, D. (2022). A Hybrid Machine Learning Approach for Performance Modeling of Cloud-Based Big Data Applications. The Computer Journal, 65(12), 3123-3140.