2023 IEEE 30th International Conference on High Performance Computing, Data, and Analytics (HiPC) | 979-8-3503-8322-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/HIPC58850.2023.00044

Oikonomos-II: A Reinforcement-Learning, Resource-Recommendation System for Cloud HPC

J.L.F. Betting[1], C.I. De Zeeuw[1,2], and C. Strydis[1,3]
[1]Department of Neuroscience, Erasmus Medical Center, Rotterdam, The Netherlands
[2]Netherlands Institute for Neuroscience, Amsterdam, The Netherlands
[3]Quantum and Computer Engineering Department, Delft University of Technology, Delft, The Netherlands

***Abstract*—The cloud has become a powerful and useful environment for the deployment of High-Performance Computing (HPC) applications, but the large number of available instance types poses a challenge in selecting the optimal platform. Users often do not have the time or knowledge necessary to make an optimal choice. Recommender systems have been developed for this purpose but current state-of-the-art systems either require large amounts of training data, or require running the application multiple times; this is costly. In this work, we propose Oikonomos-II, a resource-recommendation system based on reinforcement learning for HPC applications in the cloud. Oikonomos-II models the relationship between different input parameters, instance types, and execution times. The system does not require any preexisting training data or repeated job executions, as it gathers its own training data opportunistically using user-submitted jobs, employing a variant of the Neural-LinUCB algorithm. When deployed on a mix of HPC applications, Oikonomos-II quickly converged towards an optimal policy. The system eliminates the need for preexisting training data or auxiliary runs, providing an economical, general-purpose, resource-recommendation system for cloud HPC.**

***Index Terms*—High-Performance Computing, resource recommendation, cloud computing, prediction, middleware**

I. Introduction

High-Performance Computing (HPC) refers to the use of high computational power to solve complex calculations at high speed. Examples of HPC applications include brain simulations, weather and climate modeling, and genome sequencing. Traditionally, such computations take place on computing clusters and supercomputers, where HPC jobs are executed on a statically defined hardware allocation after being placed in a queue. However, modern cloud environments such as Amazon EC2 and Microsoft Azure offer a wide range of computing resources on demand, often without waiting times, and on a pay-per-hour basis. This makes them an attractive alternative for HPC calculations.

At the time of writing, Amazon EC2 offers 637 different instance types. The instance types are categorized into families and the available hardware and costs per hour are available on Amazon's website. Nevertheless, it remains challenging for users to make an optimal choice for their application. Recognizing this problem, Amazon offers instance type recommendation, which recommends an instance type to a user based on historical use over a 14-day period [1]. However, application execution time is affected by input size and parameters, in combination with the hardware characteristics, in a typically hard-to-predict way. This means that the optimal instance type can be different, even for the same application. Smaragdos et al. [2] showed that for a simulation model of the human brain, a CPU version, a GPU-optimized version, and a FPGA version of the application can all be optimal choices, depending on the input parameters such as size of the neural network and connectivity.

We present Oikonomos-II, a reinforcement-learning resource-recommendation system for cloud HPC. Oikonomos-II approaches cloud-resource recommendation as a contextual multi-armed bandit problem. It uses incoming jobs from researchers to both explore different cloud instance type options, while also exploiting the knowledge it gains in the process. As opposed to earlier work, which was either search-based or prediction-based, Oikonomos-II combines the best elements of both approaches, and eliminates their main weaknesses. It can therefore be seen as the first hybrid system for this purpose. This work is a novel approach over our previous work, Oikonomos [3], which used an MLP, but was still purely prediction-based.

The contributions of this work are as follows:

- A novel, reinforcement-learning instance recommender for HPC applications in heterogeneous cloud environments. By using a deep contextual bandit algorithm, it overcomes several limitations of earlier approaches.
- An improvement of the Neural-LinUCB algorithm by Xu et al. [4]: applying the principle of soft update makes it possible to use much deeper artificial neural networks, and thus the representation of much more complex context-reward relationships.
- A performance analysis of Oikonomos-II on four diverse HPC applications, showing the robustness of its reinforcement-learning approach and its potential for general (re)use.

The paper is organized as follows: in Section II, we give an extensive overview of related works, addressing the strengths and weaknesses of the various publications. In Section III, we describe our system and the underlying algorithms in detail. In Section IV, the four applications that we used for evaluation are described as well as the relevant implementation-specific

This paper is supported by the European Union's Horizon Europe research and innovation programme under projects SEPTON (Gr. Agr. No. 101094901) and SECURED (Gr. Agr. No. 101095717) and by the Dutch Research Council's Gravitation programme under project DBI^2 (No. 024.005.022).

2640-0316/23/$31.00 ©2023 IEEE
DOI 10.1109/HiPC58850.2023.00044

details. In Section V, the performance of Oikonomos-II is evaluated on four existing HPC applications. We show that Oikonomos-II explores the available options effectively and exploits the knowledge it gains, successfully selecting the best instance type for incoming jobs in the vast majority of cases, for all applications. In Section VI, we present a discussion of our findings as well as potential improvements. Section VII concludes the work.

II. Related work

Work in the field of cloud-HPC resource recommendation generally falls in one of two categories: searched-based algorithms and prediction-based algorithms. Search-based algorithms evaluate different hardware combinations in succession to find the optimal choice. These algorithms do not rely on earlier data but usually need to run a job multiple times to find an optimal instance type; this leads to extra costs. Prediction-based algorithms use offline evaluation of data to predict performance and can immediately suggest an optimal instance type. This removes the need to actively search, but these algorithms require either prior knowledge about the behavior of the application in the form of a model or historical data. A summary of related work can be found in Table I.

Venkataraman et al. [5] proposed Ernest, a prediction-based framework that works with a non-negative least-squares solver, using historic data about the size of the input data, the number of virtual machines used and the execution time to fit four parameter values to a formula. This formula is then used to predict execution times, and can be extended to include more parameter values. However, Ernest is less suitable if the application behavior is unknown. It is also unsuitable for heterogeneous hardware configurations, since it only takes the number of machines into account.

Samreen et al. [6] presented Daleel, a prediction-based framework to support decision making in Infrastructure-as-a-Service (IaaS) environments, such as clouds. Daleel uses a multivariate polynomial model to predict execution times, which is fit to the training data through different regression methods. In this respect, Daleel is similar to Ernest's formula-fitting approach. The amount of vCPUs, RAM, and the day of the week are used as input parameters. Even though Daleel achieved low Mean Square Error, like Ernest, it is less suitable for heterogeneous-hardware configurations or complex relationships between input parameters and execution time.

Yadwadkar et al. [7] proposed PARIS, another a prediction-based approach, for selecting the best Virtual Machine (VM) among multiple clouds. A central innovation of PARIS is the decoupling of instance performance characterization from the workload-specific resource requirements. It does this by profiling the instance types using a set of benchmark workloads – this has to be done only once for each instance type. It then lets the user choose and run a representative workload to analyse the resource usage patterns and create a fingerprint of the application. It uses this fingerprint to recommend an instance type based on the user's needs. The decoupling of application characteristics from instance-type characteristics is important. However, PARIS burdens the user with choosing a representative workload, and does not take the influence of application parameter values on resource usage patterns into account.

Alipourfard et al. [8] presented CherryPick, a search-based approach that uses Bayesian optimization to build a performance model for applications. A central insight of CherryPick is that a recommendation system does not need to predict the execution time as accurately as possible; it just needs to be good enough to recommend an optimal cloud configuration. The user is asked to give the objective (e.g. minimizing costs or execution time) and constraints (budget, maximum execution time), as well as a workload representative of the application. CherryPick then finds a list of candidates for the optimal hardware configuration in multiple clouds, and finds an optimal cloud configuration in an iterative manner. The authors compared CherryPick to Ernest, and found that CherryPick performed similarly when it comes to running costs, but with lower search time and cost. However, it still needs to run a workload several times, and like PARIS, burdens the user with providing representative workloads.

Hsu et al. published three search-based approaches; Scout [9], Arrow [10], and Micky [11]. Scout is a pair-wise-comparison approach that uses past performance information to search efficiently. A key insight from Scout is that any search-based algorithm has a trade-off between exploration and exploitation. Historical data can be used to optimize the exploration process in order to exploit more. Arrow, like CherryPick, uses Bayesian optimization, but augments it with low-level metrics in order to reduce search costs. The authors found that including this information led to enhanced performance compared to CherryPick's original Bayesian approach. Building on the insights from Scout and Arrow, the authors propose Micky, which casts the problem of finding the best VM as a multi-armed bandit problem and uses the Upper Confidence Bound (UCB) algorithm to optimize rewards. Micky optimizes for a batch of workloads, rather than a single workload, and aims to find a cloud configuration that is near-optimal for the majority of workloads. The authors suggest combining Micky with Arrow or Scout to find the best cloud configuration for individual workloads. Even though all of these approaches address some of the problems of search-based algorithms, all of them require running a workload multiple times to find the best configuration, which implies additional costs.

Recently, more prediction-based systems were published. Samreen et al. presented Tamakkon [12]. A key insight from Tamakkon is that historical performance data can be used for resource recommendation of new applications or VM types, if we can determine their similarity. Tamakkon does this using a Kolmogorov-Smirnov test. Based on the degree of similarity, Tamakkon adapts a machine-learning model to a specific task by using profiling data from similar applications. This makes the algorithm useful for different applications and hardware configurations. The systems does require the production of auxiliary data in the cloud, which entails additional costs. Also, Tamakkon simply labels workloads as 'similar' or 'partly

TABLE I
RELATED WORK IN THE FIELD OF CLOUD INSTANCE TYPE SELECTION [3]

Publication	Name	Approach	Input	Mechanism	Limitation vs this work
Venkataraman et al. (2016)	Ernest	Prediction-based	Number of machines (CPU cores)	Non-Negative Least-Squares fitting	Requires preexisting knowledge about application behaviour
Samreen et al. (2016)	Daleel	Prediction-based	Number of vCPUs, amount of RAM, and days of the week	Multivariate polynomial model regression	Less suitable for complex relationships between parameters and execution time
Yadwadkar et al. (2017)	PARIS	Prediction-based	Application ‘fingerprint’, VM configuration	Random-Forest Model	Burdens the user with providing a representative workload
Alipourfard et al. (2017)	CherryPick	Search-based	Representive workload	Bayesian optimization	Burdens the user with providing a representative workload
Hsu et al. (2018a)	Scout	Search-based	Low-level metrics and historical data from other workloads	Search-space exploration through relative ordering, pairwise comparison & transfer learning	Requires running a workload multiple times
Hsu et al. (2018b)	Arrow	Search-based	Low-level performance information, such as CPU utilization and memory and I/O pressure	Tree-based learning method (Extra-Trees algorithm)	Requires running a workload multiple times
Hsu et al. (2018c)	Micky	Search-based	Execution time & operational cost	Upper Confidence Bound	Requires running a workload multiple times
Samreen et al. (2019)	Tamakkon	Prediction-based	Auxiliary data and historical data	Multivariate Polynomial Regression, Support Vector Regression, and Random Forests, using transfer learning	Requires the production of auxiliary data
Samuel et al. (2020)	A2Cloud-RF	Prediction-based	PERF traces of HPC application, cloud traces of benchmark applications, historical data	Random-Forest Classifier	Decoupling applications and instance types limits the representation of their complex interplay
Ai et al. (2021)	A2Cloud-H	Prediction-based	PERF traces of HPC application, cloud traces of benchmark applications, historical data	Variety of supervised and non-supervised ML algorithms	Requires an additional algorithm to select a recommender algorithm
Betting et al. (2023)	Oikonomos	Prediction-based	Instance type, hardware & application parameters	Multi-Layer Perceptron DNN	Requires large amounts of historical data
This work	Oikonomos-II	Hybrid (Reinforcement Learning)	Instance types, hardware & application parameters	Variant of Neural-LinUCB with MLP	

similar', but does not further specify in which way the workloads are similar.

Samuel et al. [13] proposed A2Cloud-RF, a prediction-based approach which, like PARIS, decouples the characteristics of the applications and cloud instances. This is done by profiling them separately: the instances for performance using standard benchmark applications, and the applications for resource usage with the Linux `perf` application. A Random-Forest Classifier (RFC) is used to recommend an instance. The RFC can directly classify instance types as 'excellent', 'good', 'okay', or 'bad', based on these profiles. It can also classify applications as either computationally intensive, balanced, or memory-intensive, and use historical data of similar applications to create the aforementioned classification. Even though this classification of instance types is useful, but given the huge amount of available instance types, a classification in four categories is rather rough. Decoupling applications and instance types reduces the need for test runs, but also makes it more difficult to capture the complex interplay between application performance, resource use, and available hardware. It remains unclear how the `perf` traces generated for each application account for the differences in behavior that applications may have on various heterogeneous types of architecture.

Ai et al. [14] presented an expanded version of A2Cloud-RF, named A2Cloud-H (Hierarchy). Rather than only using a RFC, A2Cloud-H uses a variety of machine learning algorithms, divided into two modules: an unsupervised learning module (USL), and a supervised learning module (SL). Both modules are contained in a decision module. When a job request comes in, the decision module selects an algorithm from both modules, based on the popularity of the model (measured by the number of publications and the number of citations), the historical accuracy, and the F1 score. Users themselves get the final say as to whether the want to use the algorithm from the USL or the SL module. Even though offering a variety of algorithms might make the system more generalizable, it also makes it more complex: it creates the need for an additional algorithm to select a recommender algorithm.

Our previous work, Oikonomos [3], is a prediction-based algorithm that works in an opportunistic, data-driven fashion. Starting with the assumption that a single HPC application is executed a myriad times with different parameter values, Oikonomos consists of a Multi-Layer Perceptron (MLP) artificial neural network that takes the specific parameter values of the job and the hardware characteristics of a specific instance type as its input, and returns a prediction of the execution time. The network is trained using historical data. The users themselves only have to provide the parameter values they want to use. Oikonomos showed that a general MLP can be used as a general-purpose solution for cloud recommendation. The main weakness of Oikonomos is that it relies on a large amount of historical data, which might not be practical or available, especially for new applications. This is a problem that Oikonomos shares with other data-driven, prediction-based algorithms but neural networks tend to be especially vulnerable to it. Furthermore, the application was tested on balanced datasets; in reality, the datasets will not be balanced.

In summary, in the prediction-based approaches, there tends to exist a trade-off between more specific modeling (for instance by fitting a formula or by fingerprinting) and a reliance on considerable amounts of data. For search-based approaches, there is a trade-off between exploration and exploitation: more exploration might lead to better recommendations, but will also lead to higher overhead costs, whereas early exploitation will lead to lower overhead costs but might make the recommendations less accurate.

The work we present in this paper, Oikonomos-II, like Oikonomos, has an MLP at its core, and uses historical data. However, like Micky, we approach the problem of cloud resource recommendation as a multi-armed bandit problem, in order to explore the search space for giving better recommendations. Oikonomos-II can be seen as a hybrid approach, combining the advantages of search-based and prediction-based algorithms. In this way, it overcomes the limitations of earlier approaches.

III. Design

As the extensive related-work section demonstrates, there is a need for a middleware layer for resource recommendation in a heterogeneous HPC system that aids the user in selecting the hardware platform that is best-suited for the job they need to run. We avoid performance-model construction, as the complex interplay between the application parameters and the execution platform call for an application-agnostic approach. We also avoid running the same job more than once: we assume a stream of incoming jobs with different parameter values each time. Oikonomos-II gets to make only one decision regarding the instance type per job, and gets to observe the execution time and costs of only that particular job execution. In contrast to recommenders like Oikonomos, we assume the absence of any preexisting historical execution data. Therefore, the decisions that Oikonomos-II makes not only influence the costs and execution time of one particular job but also the available data to base future decisions on.

Because of the absence of preexisting historical data, Oikonomos-II is forced to take risks by recommending instance types it has not encountered before. At the same time, though, Oikonomos-II has to optimize performance for its users. This dilemma is known as the exploration-exploitation dilemma, which is a general problem to be found in data-driven, decision-making processes where a feedback loop exists between data gathering and decision making [15]. This becomes most clear in a class of problems known a multi-armed bandit problems.

A. The Contextual Multi-Armed Bandit Problem

Lattimore and Szepesvári [16] describe the bandit problem as a sequential game between a *learner* and an *environment*. Played over n rounds, for each round $t \in [n]$, the learner picks an *action* a_t from a set of actions $\mathcal{A}$. After the action is chosen, the environment reveals a *reward* $r_t \in \mathbb{R}$. The learner does not get to see the rewards associated with the other actions.

The learner cannot see into the future, so in the classical multi-armed bandit problem, it has to rely on the *history* $H_{t-1} = (a_1, r_1, \ldots, a_{t-1}, r_{t-1})$ in order to make decisions. The learner is expected to adopt a policy π: a mapping from histories to actions. Most commonly, the goal is for the learner to find a policy that maximizes the cumulative reward over all rounds $\sum_{t=1}^{n} r_t$. The *regret* of a policy π is defined as the difference between the cumulative expected reward using policy π and the cumulative reward of an optimal policy π^*. Cumulative regret will often grow in a logarithmic fashion for good policies: cumulative regret will increase relatively fast in the beginning, when there is little historical data and a strong need for exploration, and will slow down with time, as the amount of historical data grows, allowing for more exploitation. Bandit algorithms are part of the wider class of reinforcement-learning algorithms.

The bandit problem has been studied since the 1930s [17], but interest has skyrocketed over the last two decades because of its applicability in online environments. Dynamic pricing of online airplane bookings is a good example of a bandit problem: when a visitor searches for a flight, the website picks a price to offer to the visitor. The reward is revealed when the customer either books the flight or leaves without booking. The goal of the algorithm is to maximize total cumulative profit over all visitors [18].

Contextual knowledge can be essential for adopting a policy to make decisions. For instance, in the airplane booking example, it might be useful to know the IP address of the visitor. After all, the visitor's location might be correlated to the price they are willing to pay. Context can also consist of similarity information regarding the actions in $\mathcal{A}$. A visitor might be willing to book a flight on a different date, or to a different airport, and showing them such options could lead to a higher chance of booking. Multi-armed bandit problems where context plays a role are known as *contextual bandits*.

Two of the most widely used algorithms for solving the exploration-exploitation dilemma in multi-armed bandit problems are Upper Confidence Bound (UCB) and Thompson Sampling (TS). UCB was first proposed by Auer et al. [19], and is based on the principle of *optimism in the face of uncertainty*. This means that the algorithm estimates the expected reward, as well as a confidence bound for each action, and chooses the action that has the highest upper confidence bound. Whereas UCB is aimed at estimating the reward (see Figure 1), TS builds a probability model based on previous rewards, and then samples from this model to choose an action [17]. Both TS and UCB are widely used and have strong theoretical guarantees on the regret bound.

The original UCB and TS algorithms do not take contextual information into account. However, they have been used as bases for algorithms that do work with contextual information. One of the most popular contextual bandit algorithms is LinUCB, proposed by Li et al. [20]. The algorithm assumes a linear relationship between the context parameters and the rewards. The relationship is represented by a vector θ, which is to be learned. LinUCB was presented in two versions: a disjoint version (where only one vector of context parameters in used) and a hybrid version (where two context vectors are used: one for parameters describing the context in round t, and one for parameters that describe the actions in $\mathcal{A}$). Li et al. applied the algorithm to personalized news-article recommendation and showed that it performs better than the original UCB algorithm.

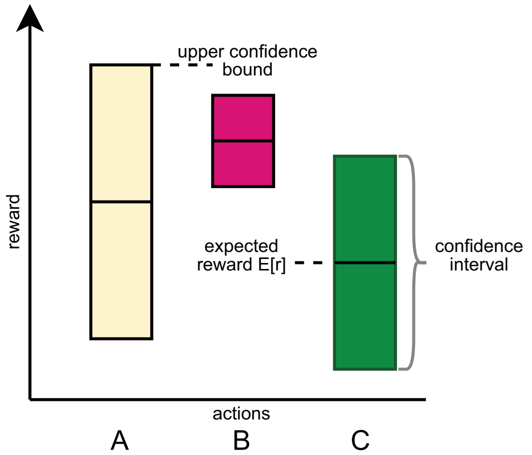

Fig. 1. *The UCB algorithm*: The expected reward E$[r]$ is assessed for each option, as well as the confidence interval. The algorithm will choose the option with the highest upper confidence bound. Even though E$[r]$ is the highest for action B, the algorithm will choose action A, as its upper confidence bound is higher: *optimism in the face of uncertainty*.

The requirement of a linear relationship between context parameters and rewards in LinUCB is restrictive. For instance, in the case of cloud HPC, the relationship between application parameters, hardware, and execution time is potentially complex. This requirement, however, can be overcome using an artificial neural network (ANN). We will mention two relevant publications. Zhou et al. presented NeuralUCB, which feeds the context vector to a neural network [21]; NeuralUCB is a generalized version of LinUCB, achieving the regret bound of LinUCB without the aforementioned requirement. However, as the whole network is used for exploration, the algorithm is very complex and computationally expensive for large neural networks. Addressing this issue, Xu et al. presented an adaptation where representation is decoupled from exploration [4]. Their algorithm, Neural-LinUCB, is based on the principle of *deep representation and shallow exploration*: it uses the entire ANN to learn the relationship between the context vectors and the rewards, but only uses the last layer for exploration. In this way, deeper and wider ANNs can be used, allowing for the representation more complex context-reward relationships. Additionally, the way in which the relationship vector θ is calculated after each round is highly parallelizable, allowing for better performance. The authors showed that Neural-LinUCB achieves similar performance to NeuralUCB, while being much less computationally expensive.

Betting et al. [3] showed with Oikonomos that a deep MLP can be used to recommend an optimal cloud-instance type for HPC applications, based on the input-parameter values. However, as Oikonomos was purely prediction-based, it relied

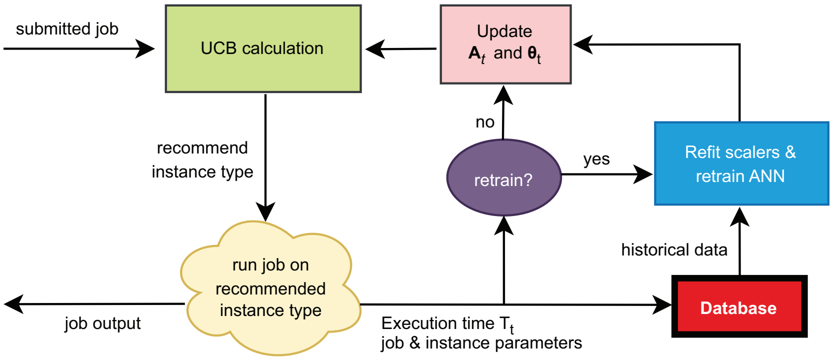

Fig. 2. *Schematic overview of Oikonomos-II*: While the user gets the job output they want, the algorithm saves the parameters of the job and its execution time, saves it in a database, and uses Neural-LinUCB to make better choices over time.

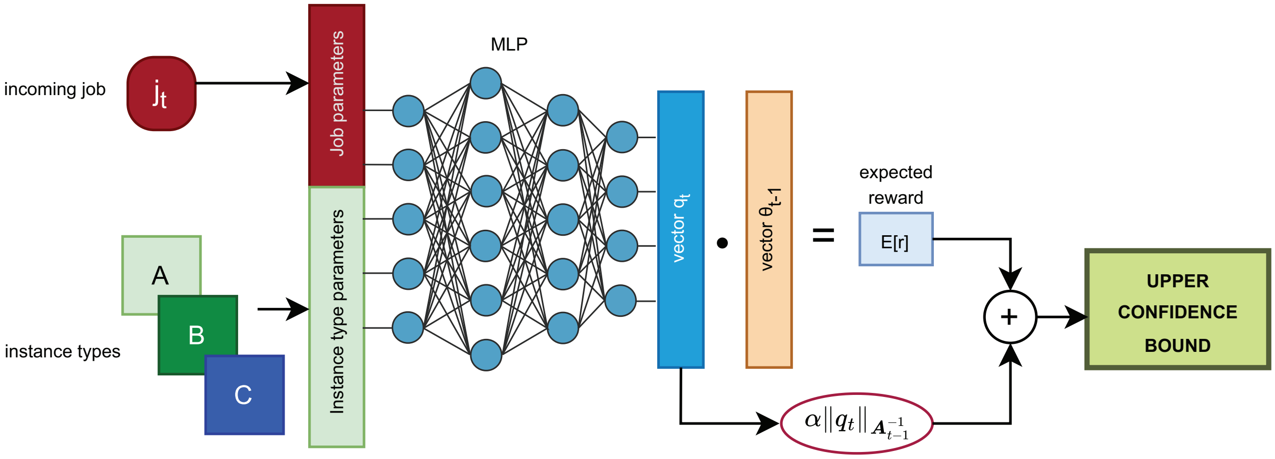

Fig. 3. *Oikonomos-II UCB calculation*: The job parameters and the instance type parameters are concatenated, and passed through an MLP. The output vector q_t is multiplied with vector θ_{t-1} to find the expected reward, and $\boldsymbol{A}_{t-1}^{-1}$ is used to calculate the confidence bound. For a detailed description of how θ_{t-1} and $\boldsymbol{A}_{t-1}^{-1}$ are determined, see Algorithm 1.

on a large amount of preexisting training data. The absence of this data creates a contextual multi-armed bandit problem. $\mathcal{A}$ consists of all possible instance type recommendations, whereas the rewards are a function of execution time and/or usage costs. Each round t involves a decision to recommend an instance type to a specific job. We define 'job' as the (requested) execution of the application with specific parameter values. The context, therefore, consists of both round-specific context (the input parameters of the job), as well as action-specific context (the hardware parameters of the instance types). The non-linear relationship between context and rewards rules out traditional LinUCB. Because of the complexity and computational costs of NeuralUCB for deeper neural networks, as well as the opportunities for parallelism that Neural-LinUCB offers, Neural-LinUCB was chosen to solve the multi-armed bandit problem that Oikonomos-II faces.

B. Oikonomos-II design

Figure 2 shows the overall architecture of Oikonomos-II, and a detailed description of its workings can be found in Algorithms 1 and 2. We assume a sequential stream of jobs, with each round t corresponding to the recommendation of an instance type and subsequent execution of job j_t. Job j_t is defined by a vector $\boldsymbol{p}_t$ of parameter values. Furthermore, we assume a set of S available instance types s; for every s, there is a vector $\boldsymbol{h}_s$ containing hardware-parameter values of the instance type, such as the number of vCPU cores, memory size, GPU type, etc. A matrix $\boldsymbol{A}_0$, and vectors $\boldsymbol{b}_0$ and $\boldsymbol{\theta}_0$ are initialized before any jobs are processed.

We make the assumption that each job j_t can start only when job j_{t-1} has finished. Each action $a \in \mathcal{A}$ is the act of assigning a job to an instance type. Action $a_{t,s}$ signifies the act of assigning job j_t to instance type s for execution. The context vectors $\boldsymbol{x}$ for Oikonomos-II consist of both the application parameters and the hardware parameters of the instance type. Here, we simply concatenate the vectors $\boldsymbol{p}_t$ and $\boldsymbol{h}_s$ to create $\boldsymbol{x}_{t,s}$. Theoretically, it is possible to implement a hybrid version of Neural-LinUCB to evaluate $\boldsymbol{p}_t$ and $\boldsymbol{h}_s$ separately, but this is to a large extent non-parallelizable and computationally much more expensive, to the extent that we consider it infeasible. Furthermore, combining $\boldsymbol{p}_t$ and $\boldsymbol{h}_s$ in a single context vector allows the MLP to learn possibly complex interplays between hardware and application parameters.

We use the combined context vector $\boldsymbol{x}_{t,s}$ to calculate the

Algorithm 1 Oikonomos-II adaptation of Neural-LinUCB

```
Input: regularization parameter $\lambda > 0$, number of jobs $J$, vector of retraining rounds $\boldsymbol{k}$, exploration parameter $\alpha > 0$,
MLP $\phi(x, w)$, context scaler function $\sigma_x(x)$, reward scaler function $\sigma_r(r)$, custom reward function $r(T)$
Initialization: $\mathbf{A}_0 = \lambda\mathbf{I}$, $\boldsymbol{b}_0 = 0$, and vector $\boldsymbol{\theta}_0$ of length $d$ filled with values $\frac{1}{d}$,
MLP weights $\boldsymbol{w}_{\mathrm{L}}$ initialized in a randomized way, empty database $\mathcal{D}$
for $t = 1, \ldots, J$ do
    receive job parameter vector $\boldsymbol{p}_t$
    concatenate $\boldsymbol{p}_t$ with vectors $\{\boldsymbol{h}_0, \ldots, \boldsymbol{h}_S\}$ to obtain unscaled context vectors $\{\boldsymbol{x}_{t,0}, \ldots, \boldsymbol{x}_{t,S}\}$
    scale each context vector with scaler function $\sigma(x)$ to obtain scaled context vectors $\{\boldsymbol{x}_{t,0}^{(\sigma)}, \ldots, \boldsymbol{x}_{t,S}^{(\sigma)}\}$
    choose action $a_t = \operatorname{argmax}_{s \in [S]} \sigma_x^{-1} \left( \boldsymbol{\theta}_{t-1}^{\mathsf{T}} \phi(\boldsymbol{x}_{t,s}^{(\sigma)}; \boldsymbol{w}_{\mathrm{L}}) + \alpha_t \| \phi(\boldsymbol{x}_{t,s}^{(\sigma)}; \boldsymbol{w}_{\mathrm{L}}) \|_{\boldsymbol{A}_{t-1}^{-1}} \right)$, and obtain execution time $T_t$
    calculate reward $r_t$ from $T_t$, using reward function $r(T)$
    store tuple $\{\boldsymbol{x}_{t,a_t}; T_t\}$ in $\mathcal{D}$
    if $t \in \boldsymbol{k}$ then
        $\sigma_x(x)$;   $\sigma_r(r)$;   $\mathbf{A}_t$;   $\boldsymbol{b}_t \leftarrow$ outputs of Algorithm 2
    else
        $\mathbf{A}_t = \mathbf{A}_{t-1} + \phi(\boldsymbol{x}_{t,a_t}^{(\sigma)}; \boldsymbol{w}_{\mathrm{L}}) \phi(\boldsymbol{x}_{t,a_t}^{(\sigma)}; \boldsymbol{w}_{\mathrm{L}})^{\mathsf{T}}$   ,   $\boldsymbol{b}_t = \boldsymbol{b}_{t-1} + r_t \phi(\boldsymbol{x}_{t,a_t}^{(\sigma)}; \boldsymbol{w}_{\mathrm{L}})$
    end if
    update $\boldsymbol{\theta}_t = \mathbf{A}_t^{-1} \boldsymbol{b}_t$
end for
Output: $\boldsymbol{w}_{\mathrm{L},J}$;   $\mathcal{D}$
```

Algorithm 2 Update ANN weights and refit scalers

```
Input: Database $\mathcal{D}$, weights $\boldsymbol{w}_{\mathrm{L}}$, MLP $\phi(x, w)$, current round $t$, soft update parameter $\tau \in (0, 1]$
Initialization: For each tuple $\{\boldsymbol{x}_{t,a_t}; T_t\} \in \mathcal{D}$, load all $\boldsymbol{x}_{t,a_t}$ into feature set $\boldsymbol{X}$. Calculate rewards $r_t$ from $T_t$ and load
these into target set $\boldsymbol{Y}$. Copy $\boldsymbol{w}_{\mathrm{L}}$ into $\boldsymbol{w}_{\mathrm{Tr}}$. Define a loss function $\mathcal{L}$. Initialize $L_{\mathrm{min}} = \infty$
Take a sample from $\boldsymbol{X}, \boldsymbol{Y}$, and divide into training set $\boldsymbol{X}_{\mathrm{tr}}, \boldsymbol{Y}_{\mathrm{tr}}$ and validation set $\boldsymbol{X_v}, \boldsymbol{Y_v}$
Refit $\sigma_x$ to $\boldsymbol{X}_{\mathrm{tr}}$ and $\sigma_r$ to $\boldsymbol{Y}_{\mathrm{tr}}$, and scale vectors $\boldsymbol{X}$, $\boldsymbol{X}_{\mathrm{tr}}$, $\boldsymbol{X_v}$, $\boldsymbol{Y}$, $\boldsymbol{Y}_{\mathrm{tr}}$, $\boldsymbol{Y_v}$ accordingly. Divide the sets into mini-batches.
for each epoch do
    Use $\phi(x, \boldsymbol{w}_{\mathrm{L}})$ to recalculate $\mathbf{A}$ and $\boldsymbol{b}$ for each data point $\in \boldsymbol{X}, \boldsymbol{Y}$ (see Algorithm 1)
    Recalculate $\boldsymbol{\theta}_t$ for each data point in $\boldsymbol{X}_{\mathrm{tr}}, \boldsymbol{Y}_{\mathrm{tr}}$ and $\boldsymbol{X_v}, \boldsymbol{Y_v}$
    Update $w_T$ by performing backpropagation using the training set and loss function $\mathcal{L} \left( \boldsymbol{\theta}_{t-1}^{\mathsf{T}} \phi(\boldsymbol{x}_{t,s}^{(\sigma)}; \boldsymbol{w}_{\mathrm{Tr}}), r_t^{(\hat{\sigma_r})} \right)$
    Soft update step: $\boldsymbol{w}_{\mathrm{L}} \leftarrow \tau \boldsymbol{w}_{\mathrm{Tr}} + (1 - \tau) \boldsymbol{w}_{\mathrm{L}}$
    Calculate validation loss $L_v$, using $\boldsymbol{w}_{\mathrm{L}}$, the validation set, and loss function $\mathcal{L}$
    if $L_v < L_{\mathrm{min}}$ then
        $\boldsymbol{w}_{min} = \boldsymbol{w}_{\mathrm{L}}$
        $L_{\mathrm{min}} = L_v$
    end if
end for
$\boldsymbol{w}_{\mathrm{L}} = \boldsymbol{w}_{min}$
Use $\phi(x, \mathrm{w}_{\mathrm{L}})$ to recalculate $\mathbf{A}_t$ and $\boldsymbol{b}_t$
Output: $\boldsymbol{w}_{\mathrm{L}}$;   $\sigma_x(x)$;   $\sigma_r(r)$;   $\mathbf{A}_t$;   $\boldsymbol{b}_t$
```

upper confidence bound for the reward of each job-instance type combination, as is done in the original Neural-LinUCB algorithm. The vector $q_t = \phi(\boldsymbol{x}_{t,s}; \boldsymbol{w}_{\mathrm{L}})$ is obtained by passing $\boldsymbol{x}_{t,s}$ through the MLP. q_t is multiplied with vector θ_{t-1} to find the expected reward, and the inverse of $\boldsymbol{A}_{t-1}$ is used to calculate the confidence bound. Following the notation used by Xu et al., we use $[k]$ to denote a set $\{1, \ldots, k\}, k \in \mathbb{N}^+$. For a semi-definite matrix $\mathbf{A} \in \mathbb{R}^{d \times d}$ and vector $\mathbf{x} \in \mathbb{R}^d$, the Mahalanobis norm is denoted as $\|\mathbf{x}\|_{\mathbf{A}} = \sqrt{\mathbf{x}^{\mathsf{T}} \mathbf{A} \mathbf{x}}$. The process is visualized in Figure 3. The action with the highest upper confidence bound is recommended to the user.

As shown in Figure 2, after the algorithm recommends an instance type, the job is executed there. The job output is then returned to the user. The execution time T_t, as well as vectors $\boldsymbol{p}_t$ and $\boldsymbol{h}_s$ are stored in a database. The ANN is retrained periodically; it would be computationally expensive to retrain after every round. However, $\boldsymbol{A}_t$, $\boldsymbol{b}_t$, and $\boldsymbol{\theta}_t$ are calculated after every round, and are used for UCB calculation and instance-type recommendation in the next round.

The Oikonomos-II algorithm is described in detail in Algorithms 1 and 2. For a more detailed explanation of Neural-LinUCB and proof of the regret bound, we refer the reader to the original paper. We made several adaptations to the Neural-LinUCB algorithm to make it suitable for our application. The

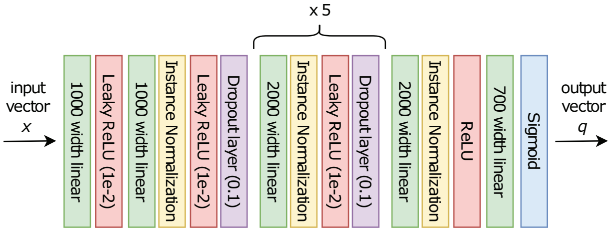

Fig. 4. The architecture of the MLP used in Oikonomos-II

original Neural-LinUCB algorithm retrains the ANN every k steps. We noticed that the network requires frequent retraining in the beginning, and requires less frequent retraining later, when there is more data available. Therefore, rather than defining k as an integer, we define $\boldsymbol{k}$ as a vector of positive integers. If $t \in \boldsymbol{k}$, the ANN is retrained after round t. The size and content of $\boldsymbol{k}$ can be chosen by the user.

It was also noted that, when training the ANN, there exists a feedback loop between the weights of the ANN and the feature vector $\boldsymbol{\theta}$: after all, $\boldsymbol{\theta}$ depends on $\boldsymbol{A}$ and $\boldsymbol{b}$, and $\boldsymbol{A}$, $\boldsymbol{b}$ are updated after every round using the MLP output vector q. This led to instability and reduced performance during backpropagation, as $\boldsymbol{\theta}$ is used in the loss function (see Algorithm 2). We resolved the issue by applying soft updating, as described by Lillicrap et al. [22], where the target network is used to recalculate θ for each data point at the start of each epoch, and backpropagation is applied to the training network. A soft-update step is performed at the end of each epoch. This allowed us to use deeper neural networks, which makes it possible to represent more complex relationships between inputs and rewards. We also improved the MLP training process by applying best-practise techniques, such as data scaling, training with mini-batches, and early stopping with separate training and validation sets in Oikonomos-II.

IV. Implementation

The system was implemented in Python. As machine-learning framework, we used PyTorch [26]. An MLP of nine linear layers was used, with a maximum width of 2,000. Normalization and dropout layers were used to enhance performance. (Leaky) ReLU functions were used as activation layers, except for the last layer, where a sigmoid function was used (which was observed to improve performance). The full architecture is summarized in Figure 4. The length d of the output vector (which corresponds to the length of vector θ) is 700. As loss function $\mathcal{L}$, we used the Mean-Squared Error (MSE) loss function. The data set (X, Y) used for training consists of a maximum of random 3000 samples from $\mathcal{D}$; for $t \leq 3000, (X, Y) = \mathcal{D}$. The minibatch size was initialized at 1, and slowly increased as $\mathcal{D}$ increased, to a maximum of 16.

The MLP was retrained after 50 episodes, and then at intervals of 500 episodes. Of the dataset, 85% was used as a training set, and the remaining 15% as a validation set. Backpropagation is performed using the Adam optimizer [27]. Training is done for 500 episodes, the weights of the episode with the lowest validation loss are retained. As for the reward function: as LinUCB strives to maximize the reward value, and our goal is to minimize either costs or execution time, we defined the reward function $r(x) = \frac{1}{x}$, with x the costs in dollars, or the execution time in seconds. The parameter set was scaled using `StandardScaler`, the rewards were scaled using `PowerTransformer`, both from the `sklearn.preprocessing` library for Python.

TABLE II
HPC applications used for the Oikonomos-II evaluation, and their parameter ranges

(E) HPCC

parameter	range
MPI-procs	4-8
N	20,000 - 40,000
NB	100 - 20,000

(F) simHH

parameter	range
time-steps	1 - 110,000
connectivity	0.5 - 1.0
neurons	1000 - 10,000

(G) MNIST (MLP)

parameter	range
epochs	1 - 1000
Training batch size	1 - 5000
Test batch size	1 - 5000
Layer 1 size	1 - 750
Layer 2 size	1 - 1000

(H) CIFAR-10 (CNN)

parameter	range
epochs	1 - 1000
Training batch size	1 - 3000
Test batch size	1 - 3000
Layer 1 size	1 - 500
Architecture type	1 - 4

V. Evaluation

A. Experimental setup

For the evaluation of Oikonomos-II, we used four different benchmark applications. Three of these are real HPC applications, the other one is a synthetic benchmark from the ARCHER suite. The first is simHH, a neurosimulator developed at the Erasmus Medical Center, Rotterdam [23]. It simulates a wide range of biologically plausible, conductance-based Hodgkin-Huxley neural models. These models work on non-embarrassingly parallel workloads and uses basic solvers operating on short time intervals. The second application involves training an MLP Deep Neural Network with Google TensorFlow using the widely-used MNIST database [24]. MNIST is a standard dataset in TensorFlow for testing AI classification. Training time varies based on adjustable hyper-parameters. The third application is training a TensorFlow-based Convolutional Neural Network (CNN) using the CIFAR-10 database. CIFAR-10 consists of color images from ten classes; the CNN is trained to classify images. The training time is influenced by variable parameters such as the number of convolutional layers, fully-connected layer size, epochs, and test/training minibatch sizes. HPCC [25] is a collection of synthetic benchmarks that measure the range of memory-access patterns. The application has MPI and OpenMP support. There are no available GPU or FPGA implementations of HPCC, but the number of MPI processes can be varied. The application parameters we varied are stated in Table II. As for instance type parameters, we used the number of vCPUs, the instance memory in MiB, and the number of GPUs.

The Amazon instance types that we have used for our evaluation are shown in Table III. Without loss of generality, they were selected so as to create a diversity of hardware

TABLE III
AMAZON EC2 INSTANCE TYPES USED FOR OIKONOMOS-II EVALUATION

Instance type	CPU type	vCPU no.	Memory (GiB)	GPU type	GPU mem. (GiB)
t2.2xlarge	Intel Xeon Family @ 3.3 GHz	8	32	–	–
c5a.4xlarge	AMD EPYC 7R32 @ 2.8 GHz	16	32	–	–
m5a.4xlarge	AMD EPYC 7571 @ 2.5 GHz	16	64	–	–
m5a.8xlarge	AMD EPYC 7571 @ 2.5 GHz	32	128	–	–
c5a.8xlarge	AMD EPYC 7R32 @ 2.8 GHz	32	64	–	–
c5a.12xlarge	AMD EPYC 7R32 @ 2.8 GHz	48	192	–	–
g3.4xlarge	Intel Xeon E5-2686 v4 @ 2.3 GHz	16	122	Tesla M60	8.0
g4dn.4xlarge	Intel Xeon Family @ 2.5 GHz	16	64	Tesla T4	16.0

TABLE IV
DISTRIBUTION OF BEST INSTANCE TYPE IN ORACLE SETS (TIME / COST)

	simHH	MNIST	HPCC	CIFAR-10
t2.2xlarge	1.66% / 2.44%	0.00% / 1.52%	0.34% / 1.60%	0.00% / 0.00%
c5a.4xlarge	4.66% / 25.58%	1.06% / 34.00%	0.00% / 98.40%	0.00% / 0.00%
m5a.4xlarge	2.70% / 1.10%	0.00% / 0.00%	0.00% / 0.00%	0.00% / 0.00%
m5a.8xlarge	0.16% / 0.00%	0.00% / 0.00%	0.00% / 0.00%	0.00% / 0.00%
c5a.8xlarge	6.70% / 0.18%	0.34% / 0.00%	75.44% / 0.00%	0.00% / 0.00%
c5a.12xlarge	26.28% / 0.00%	2.10% / 0.00%	24.22% / 0.00%	0.00% / 0.00%
g3.4xlarge	5.02% / 18.06%	0.00% / 0.00%	0.00% / 0.00%	0.00% / 0.00%
g4dn.4xlarge	52.82% / 52.64%	96.50% / 64.48%	0.00% / 0.00%	100.00% / 100.00%

options, but we also selected some instance types from the same family. This allows us to test if Oikonomos-II can discern between instances that are relatively similar. Two instance types have a GPU available, three are compute-optimized, and three are general-purpose instances.

To evaluate the performance of Oikonomos-II, we used datasets where all the jobs have been executed fully on each of the eight instance types. We call these datasets *oracle sets*, since they provide us with full insight into the best possible policy and the regret of each different policy.[1] By using these sets as a simulation environment, it was possible to evaluate the regret for each application. For our four applications, we used oracle sets of 5,000 jobs. The sets were created by executing jobs on the Amazon EC2 instances, and then augmenting the data by manually studying the behavior of these applications, in order to create sets that reliably represent the application behavior on the cloud instances. We randomized the order of each of the jobs and presented the jobs one by one to Oikonomos-II. The algorithm only gets to see the execution time of a job for the instance type it has chosen, and it cannot see into the future.

Comparing Oikonomos-II to other work is challenging, since each author uses their own HPC application to evaluate performance – the absence of a good benchmark set for cloud HPC recommendation is a persistent issue in this field. Even when the same applications are used, differences in parameter ranges can lead to vastly different data sets. Most standard HPC benchmark sets are unsuitable for our purpose: they are designed to characterize specific HPC platforms, and fail to capture the complex interplay between application characteristics, individual job input parameter values, and hardware. We therefore decided to evaluate the performance of Oikonomos-II in its own right.

We employed three metrics. The first metric is the percentage of all rounds for which the best instance type was recommended. This shows the performance of Oikonomos-II, including the exploration phase. The second metric is the percentage of the last 1,000 rounds for which the best instance type was recommended. By this time, the algorithm has had the opportunity to explore and should be mostly exploiting. The last metric is the regret. Regret is usually defined as the difference between the optimal policy and the actual policy. The unit and size of the regret differs for every application. In order to compare the applications, it was decided to express regret as a percentage of the regret of random policy.

B. Results

We evaluated the performance of Oikonomos-II on all four applications, optimizing for both execution time and costs. We analyzed the oracle sets to determine the distribution of the best option. The most interesting cases are those where the best choice of instance type depends on the parameter values. As shown in Table IV, to varying degrees, this is the case for all benchmarks except for CIFAR-10, where g4dn.4xlarge is the overall best choice for both cost and time.

Table V shows the evaluation results for all four applications, employing the three metrics. Despite the fact that Oikonomos-II starts without any knowledge about any of the applications, over 5,000 episodes it is able to recommend the best instance type for jobs it has not seen before. The percentage of optimal recommendations becomes even higher when only the last 1,000 episodes are considered. This is expected: after all, the later the episode, the more the algorithm can rely on previous observations. However, the numbers are not much different, as Oikonomos-II has likely converged much earlier. It is interesting to compare these two metrics to the data in Table IV. For example, simHH shows a lot of variation regarding the optimal instance type: the best choice is heavily dependent on job parameters. The high percentage of optimal recommendations shows that Oikonomos-II is able to effectively learn the relationship between input parameters and optimal instance type. Oikonomos-II appears to perform less well in predicting the instance with the fastest execution time for HPCC. We found that this was caused by the fact that two

[1] The oracle sets that were used for evaluation can be found at: https://gitlab.com/c7859/neurocomputing-lab/oikonomos-II_data.

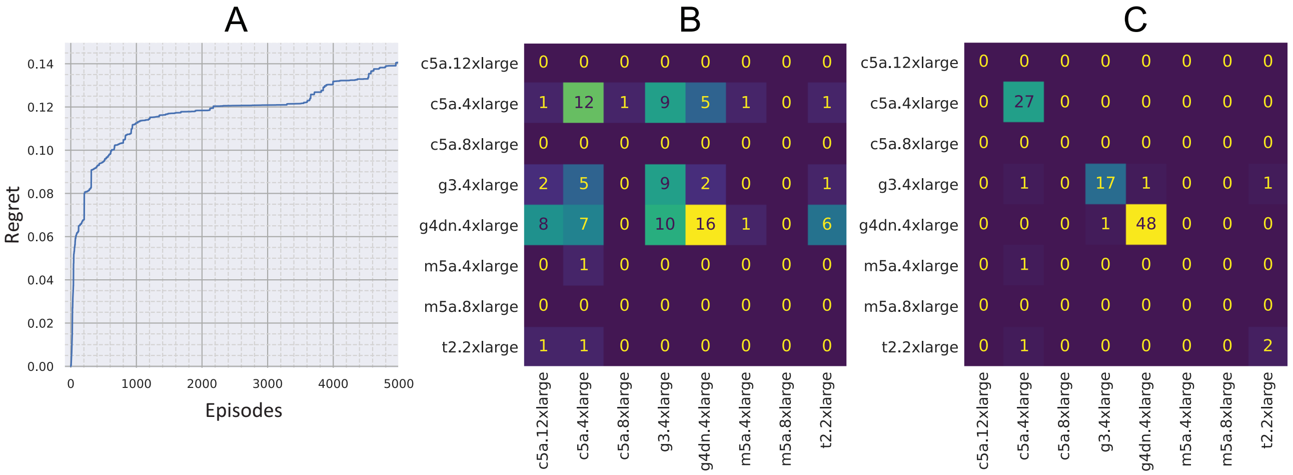

Fig. 5. **A**: Cumulative regret for simHH (cost-optimized). Regret increases rapidly in the beginning, but then mostly flatlines. **B**: Confusion matrix of recommendation choices for the first 100 episodes for simHH (cost-optimized). True labels are on the x-axis, whereas recommendations are on the y-axis. Since Oikonomos-II has not explored the space yet, it is forced to explore and make suboptimal choices. **C**: Confusion matrix for the last 100 episodes for the same application. Now that Oikonomos-II has explored the relationship between parameters and performance, it mostly exploits and makes optimal choices: most recommendations coincide with the true best option.

TABLE V
Oikonomos-II evaluation results (time / cost)

	simHH	MNIST	HPCC	CIFAR-10
Optimal action over all episodes	79.34% / 91.28%	92.92% / 87.16%	54.56% / 97.40%	95.88% / 95.82%
Optimal action over last 1000 episodes	81.40% / 94.24%	95.60% / 89.50%	68.50% / 97.50%	99.00% / 95.00%
Regret as perc. of random policy regret	2.12% / 1.57 %	4.42% / 10.59%	2.46% / 1.42%	0.99% / 2.16%

instance types have similar performance for this application. Therefore, which of those two performs for a particular job is in part determined by chance. For all applications, the regret is only a small percentage of the regret of a random policy, which shows that Oikonomos-II far outperforms a random policy.

Figure 5 gives a more detailed look into the performance of one of the applications, simHH, optimized for costs. Figure 5A shows the cumulative regret over time. Cumulative regret increases rapidly in the beginning as Oikonomos-II is forced to make sub-obtimal choices in order to explore. However, it rapidly flatlines. However, regret seems to increase faster again after about 3,500 episodes. This was likely due to the fact that, after this point, only a sample of $\mathcal{D}$ is used to train the MLP, in order to increase training speed – when we reran the experiment without sampling, the sudden jump in regret disappeared. Even though the original Neural-LinUCB paper states that performance loss is limited, this figure suggests that it is not negligible.

Figure 5B shows the confusion matrix for the first 100 episodes for simHH, and Figure 5C shows the confusion matrix for the last 100 episodes. In the first 100 episodes, Oikonomos-II has not explored the space yet, but is forced to explore and make suboptimal choices, which is why the confusion matrix is rather scattered. However, in the last 100 episodes, Oikonomos-II has explored the relationship between parameters and performance, and is able to exploit, which is shown by the fact that almost all points in the matrix lie along the diagonal.

VI. Discussion

So far, we made the assumption that cloud instances are up and running, and are readily available; *start-up times* were not taken into account. In a real-life scenario, it might be useful to keep instances running in some situations (for example, when there is a continuous stream of jobs), whereas in other situations, it would be better to shut them down between runs. Developing an algorithm that takes this into account would be useful, but requires additional information about usage patterns, which was outside the scope of Oikonomos-II. Still, this could be an interesting extension of the current work, when combined with a suitable scheduling algorithm.

Oikonomos-II uses a deep neural network which needs to be retrained regularly. *Retraining* can be a time-consuming task that is best done on specific hardware types, such as a GPU; this might incur extra costs. However, in our evaluation, we showed that Oikonomos-II can deliver outstanding results with a long retraining interval of 500 episodes. Furthermore, it is possible to reduce the training time by retraining on only a sample of the data. Xu et al. argued that this is possible for Neural-LinUCB without significant reduction in performance, and we expect the same for Oikonomos-II.

The current *number of instance types* offered by AWS is over 600. Oikonomos-II was tested on data from eight instance types, which is only a fraction of the number of instances offered. However, as there is currently no standard benchmark set for resource recommendation in cloud HPC, it was necessary to collect our own oracle datasets to evaluate performance. This required that we limit ourselves to a small selection of instance types. Even so, the small set of eight instance types contains sufficient diversity. The fact that oftentimes, there is not one overall 'best' instance type, attests to this.

Oikonomos-II was tested on two types of *reward functions*: cost- and time-optimized. A fixed reward function for all episodes was assumed. However, in a real-life situation, some users might want the instance type that delivers the fastest results, whereas other users want to have results at the lowest cost. Yet others might prefer a balance between these two or have additional requirements. The current implementation of Oikonomos-II does not support this diversity of user requirements but its design can be easily extended to accommodate a variety of custom reward functions in the future.

Finally, the assumption was made that jobs arrive and are dispatched in a sequential manner: a new job arrives when the previous job has completed. In reality, however, jobs may arrive simultaneously, and a new job may arrive before the previous ones have finished. This might affect recommender performance. However, the problem of delayed feedback in bandits is well-studied [28], and the structure of Oikonomos-II is suitable for expansion to incorporate solutions to challenges that may arise in practice. In addition, it would also be valuable to assess how Oikonomos-II would perform on other contextual bandit algorithms, such as Thompson Sampling. However, this falls beyond the scope of the current work.

VII. Conclusion

Oikonomos-II casts the problem of cloud instance-type selection for different HPC jobs as a contextual multi-armed bandit problem. It applies a variant of the Neural-LinUCB algorithm, balancing exploration and exploitation. The system starts off without knowledge of the application behavior, and is forced to explore when recommending instances for incoming jobs. However, as it gathers knowledge, Oikonomos-II starts exploiting and converges towards optimal choices. We evaluated Oikonomos-II on four diverse HPC applications, where it was shown to converge towards optimal choices, demonstrating its effectiveness and robustness. Oikonomos-II avoids the main issues of both prediction-based and search-based recommenders. Combining the best elements of these two approaches into a reinforcement-learning recommender system, Oikonomos-II is both generalizable and accessible, making it a promising tool for researchers who want to harness the power of the cloud for their high-performance computing applications.

References

[1] AWS. ”Amazon EC2 Instance Recommendations”, *Amazon Web Services Documentation*, https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-recommendations.html, accessed 12-06-2023.

[2] G. Smaragdos, G. Chatzikonstantis, R. Kukreja, H. Sidiropoulos, D. Rodopoulos, I. Sourdis, Z. Al-Ars, C. Kachris, D. Soudris, C. I. De Zeeuw, et al., ”BrainFrame: a node-level heterogeneous accelerator platform for neuron simulations,” *Journal of Neural Engineering*, vol. 14, no. 6, 2017.

[3] J.L.F. Betting, D. Liakopoulos, M. Engelen, C. Strydis, ”Oikonomos: An Opportunistic, Deep-Learning Resource Recommendation System for Cloud HPC,” in *2023 IEEE 34th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2023.

[4] Pan Xu, Zheng Wen, Handong Zhao, and Quanquan Gu, “Neural Contextual Bandits with Deep Representation and Shallow Exploration,” in *International Conference on Learning Representations*, 2022.

[5] S. Venkataraman, et al., ”Ernest: Efficient performance prediction for large-scale advanced analytics,” in 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16), 2016.

[6] F. Samreen, Y. Elkhatib, M. Rowe, and G. S. Blair, ”Daleel: Simplifying cloud instance selection using machine learning,” in *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium*, IEEE, 2016, pp. 557–563.

[7] N. J. Yadwadkar, et al., ”Selecting the best VM across multiple public clouds: A data-driven performance modeling approach,” in Proceedings of the 2017 Symposium on Cloud Computing, 2017.

[8] O. Alipourfard, et al., ”CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics,” in 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), 2017.

[9] C.-J. Hsu, et al., ”Scout: An Experienced Guide to Find the Best Cloud Configuration,” arXiv preprint arXiv:1803.01296, 2018.

[10] C.-J. Hsu, et al., ”Arrow: Low-level augmented bayesian optimization for finding the best cloud vm,” in 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), IEEE, 2018.

[11] C.-J. Hsu, et al., ”Micky: A cheaper alternative for selecting cloud instances,” in 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), IEEE, 2018.

[12] F. Samreen, G. Blair, and Y. Elkhatib, ”Transferable Knowledge for Low-cost Decision Making in Cloud Environments,” IEEE Transactions on Cloud Computing, 2020.

[13] D. Samuel, et al., ”A2Cloud-RF: A random forest based statistical framework to guide resource selection for high-performance scientific computing on the cloud,” Concurrency and Computation: Practice and Experience, vol. 32, no. 24, 2020.

[14] X. Ai, T. Jena, S. Khan, R. Hughes, and V. K. Pallipuram, ”A2Cloud-H: A Multi-tiered Machine Learning Framework for Cost-Effective Cloud Resource Selection,” in *Proceedings of the Future Technologies Conference (FTC) 2021, Volume 3*, Springer, 2022, pp. 272–291.

[15] J. Rocca, ”The exploration-exploitation trade-off: intuitions and strategies”, published in *Toward Data Science* (2021). https://tinyurl.com/rocca2021 , accessed 20-06-2023.

[16] T. Lattimore and C. Szepesvári, *Bandit algorithms*. Cambridge University Press, 2020.

[17] W. R. Thompson, “On the likelihood that one unknown probability exceeds another in view of the evidence of two samples,” *Biometrika*, vol. 25, no. 3-4, pp. 285–294, 1933.

[18] A. Slivkins *et al.*, “Introduction to multi-armed bandits,” *Foundations and Trends® in Machine Learning*, vol. 12, no. 1-2, pp. 1–286, 2019. Publisher: Now Publishers, Inc.

[19] P. Auer, N. Cesa-Bianchi, and P. Fischer, “Finite-time analysis of the multiarmed bandit problem,” *Machine Learning*, vol. 47, pp. 235–256, 2002. Publisher: Springer.

[20] L. Li, W. Chu, J. Langford, and R. E. Schapire, “A contextual-bandit approach to personalized news article recommendation,” in *Proceedings of the 19th International Conference on World Wide Web*, pp. 661–670, 2010.

[21] D. Zhou, L. Li, and Q. Gu, “Neural contextual bandits with UCB-based exploration,” in *International Conference on Machine Learning*, pp. 11492–11502, 2020. Publisher: PMLR.

[22] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, ”Continuous control with deep reinforcement learning,” *arXiv preprint arXiv:1509.02971*, 2015.

[23] M. Engelen, ”Scalable GPU Acceleration for Complex Brain Simulations,” MSc thesis, Delft University of Technology, 2021. Available at: http://resolver.tudelft.nl/uuid:b79bbfa7-0c57-4949-b974-83a7d9ee6b39

[24] Image classification, MNIST digits, http://neupy.com/2016/11/12/mnist_classification.html, accessed 14-12-2021.

[25] P. Luszczek, et al., ”Introduction to the HPC challenge benchmark suite,” Technical report, Lawrence Berkeley National Lab (LBNL), Berkeley, CA (United States), 2005.

[26] A. Paszke et al., ”PyTorch: An Imperative Style, High-Performance Deep Learning Library,” in *Advances in Neural Information Processing Systems 32*, pp. 8024–8035, 2019.

[27] D. P. Kingma and J. Ba, ”Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[28] A. Grover et al., ”Best arm identification in multi-armed bandits with delayed feedback,” in *International Conference on Artificial Intelligence and Statistics*, pp. 833–842, 2018.