# Learning-based Dynamic Pinning of Parallelized Applications in Many-Core Systems

Georgios C. Chasparis Vladimir Janjic Michael Rossbory Kevin Hammond

***Abstract*—This paper introduces a learning-based framework for dynamic placement of threads of parallel applications to the cores of Non-Uniform Memory Access (NUMA) architectures. Adaptation takes place in two levels, where at the first level each thread independently decides on which group of cores (NUMA node) it will execute, and on the second level it decides to which particular core from the group it will be pinned. Naturally, these two adaptation levels run on different time-scales: a low-frequency switching for the NUMA-node adaptation, and a high-frequency switching for the CPU-node level adaptation. In addition, the learning dynamics have been designed to handle measurement noise and rapid variations in the performance of the threads.**

**The advantage of the proposed learning scheme is the ability to easily incorporate any multi-objective criterion and easily adapt to performance variations during runtime. Our objective is to demonstrate that this framework is appropriate for supervising parallel processes and intervening with respect to better resource allocation. Under the multi-objective criterion of maximizing total completed instructions per second (i.e., both computational and memory-access instructions), we compare the performance of the proposed scheme with the Linux operating system scheduler. We have observed that performance improvement could be significant especially under limited availability of resources and under irregular memory-access patterns.**

## I. Introduction

Efficient resource allocation for multi-threaded applications in NUMA architectures has attracted significant scientific attention due to a) the involved complexity of the decision-making process, and b) the need to incorporate alternative optimization criteria that go beyond standard maximization of execution speed. This statement is further reinforced by the recent advancement of tools for parallelizing complex applications, that gave birth to non-trivial and highly advanced parallel and data patterns [1], [2], [3], [4]. As expected, the problem of efficiently utilizing resources, while concurrently maximizing a multi-objective criterion, cannot be treated by standard heuristic-based techniques.

To this end, this paper proposes and investigates the potential of a learning- or measurement-based scheme that could operate in a supervisory manner on top of the existing operating system (OS) by regularly correcting/improving allocation decisions given the observed application's performance.

This work has been supported by the European Union grant EU H2020-ICT-2014-1 project RePhrase (No. 644235). It has also been partially supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH.

G. C. Chasparis and M. Rossbory are with the Software Competence Center Hagenberg GmbH, Softwarepark 21, A-4232 Hagenberg, Austria.

V. Janjic and K. Hammond are with the School of Computer Science, University of St Andrews, Scotland, UK.

In particular, this paper proposes a distributed learning scheme specifically tailored for addressing the problem of dynamically assigning/pinning threads of a parallelized application to the available processing units. The proposed scheme is flexible enough to incorporate any multi-objective optimization criterion and provide convergence guarantees to at least suboptimal assignments. Given the fact that it is measurement-based, it is computationally efficient with a linear-complexity with the number of threads. Since it is iterative in nature, it also exhibits minimal memory requirements.

In our previous work [5], [6], we have proposed a reinforcement-learning-based distributed scheduling framework (PaRLSched), adapted to Uniform Memory Architectures (UMA). In this paper, our goal is to provide a generalized methodology that also extends to Non-Uniform Memory Architectures (NUMA). Such framework should be considered as a supervisory scheme that acts on top of any OS scheduling and performs either low- or high-frequency allocation corrections possibly subject to alternative multi-objective criteria. For example, when optimizing with respect to both computational and memory-access instructions completed per second, the learning scheme should find the right balance between computing bandwidth and memory affinities. In this paper, we are not concerned with memory migrations.

Lastly, it is worth noting that we target an *online* learning framework where allocation decisions are taken during runtime, and without requiring any prior application knowledge.

The paper is organized as follows. Section II discusses related work and contributions. Section III describes the problem formulation and objective of the paper. Section IV presents the main features of the proposed Dynamic Scheduler (PaRLSched) and Section V presents a comparison with benchmark applications. Finally, Section VI presents concluding remarks and future work.

## II. Related Work and Contributions

Prior work has demonstrated the importance of thread-to-core bindings in the overall performance of a parallelized application [7]. The task of discovering such optimal bindings is rather complex, given the structure of NUMA architectures [8]. This task becomes even harder given the need for developing tools that can easily generalize to any architecture and they are application independent.

For example, [9] describes a tool that checks the performance of each of the available thread-to-core bindings and searches for an optimal placement. Unfortunately, the *exhaustive-search* type of optimization that is implemented

may prohibit runtime implementation. Reference [10] combines the problem of thread scheduling with *scheduling hints* related to thread-memory affinity issues. A similar scheduling policy is also implemented by [11].

At the same time, given that no prior knowledge of the application's details is available, a centralized optimization formulation is prohibitive. Such design restrictions give rise to learning-based techniques, where scheduling decisions are taken based only on performance measurements. This need for learning from data has been recognized in [12], where a machine learning based mechanism is designed for transactional applications. In this case, each instance of the application has to be run and profiled before any learning process is to be implemented.

Even such learning processes could be computationally complex given the quite large search space. For this reason, distributed or game-theoretic optimizations have been attempted in the past for related problems, including cooperative game formulation for allocating bandwidth in grid computing [13], the non-cooperative game formulation in the problem of medium access protocols in communications [14] or for allocating resources in cloud computing [15]. These approaches can significantly reduce the involved computational complexity and also allow for the development of online selection rules based on performance measurements. However, such modeling techniques have not yet been implemented in the context of pinning of parallelized applications.

Recognizing this need for both learning- and distributed-based optimization, and contrary to the aforementioned references on pinning of parallelized applications, our recent work [5], [6] proposed a scheduling scheme for optimally allocating threads of a parallelized application that combines both a learning- and a distributed-based optimization. It requires a minimum information exchange, where only measurements collected from each running thread are needed. Furthermore, it is flexible enough to accommodate alternative optimization criteria depending on the available performance counters. It was shown both analytically and through experiments under the Linux operating system, that the proposed methodology can learn a locally-optimal assignment, which under certain conditions also corresponds to the global optimum [5], [6]. However, one potential drawback was the fact that no special consideration was taken upon the possible *non-uniform memory access* (NUMA) architectures, as it did not distinguish between moving a thread to a "local" (within the same NUMA node) and "remote" (from a different NUMA node) core.

This paper extends the scheduling framework of our previous work [5], [6] with respect to the following contributions:

(C1) We propose a novel two-level scheduling process that is appropriate for NUMA architectures. At the higher level, the scheduler decides on which NUMA node each thread should be assigned, while at the lower level it decides on which CPU core (within that NUMA node) to execute the thread.

(C2) We propose a novel learning dynamics motivated by *aspiration learning* for making decisions at the higher NUMA-node level.

(C3) We advance the low-level learning dynamics for CPU-node selection by also incorporating satisfaction levels which significantly reduce unnecessary switching between CPU nodes within a NUMA node.

(C4) We demonstrate the efficiency of the proposed approach on several benchmark applications with different characteristics with respect to their computational and data intensity.

## III. Problem Formulation and Objective

Let a parallel application comprises $n$ threads, $\mathcal{I} = \{1, 2, ..., n\}$. We denote the *assignment* of a thread $i$ to a set of available NUMA nodes $\mathcal{J}_{\text{NUMA}}$ by $\alpha_i \in \mathcal{J}_{\text{NUMA}}$. Within the selected NUMA node $\alpha_i$, thread $i$ should be assigned to one of the available CPU cores $\mathcal{J}_{\text{CPU}}(\alpha_i)$, denoted by $\beta_i \in \mathcal{J}_{\text{CPU}}(\alpha_i)$. Let also $\alpha = \{(\alpha_i, \beta_i), i \in \mathcal{I}\}$ denote the overall *assignment profile*, and let $\mathcal{A}$ be the set of all profiles.

The Resource Manager (RM) periodically checks the performance of a thread and makes decisions about its assignment for the next scheduling iteration. ***For the remainder of the paper***, we will assume that: a) The internal properties and details of the threads are not known to the RM. Instead, the RM may only have access to measurements related to their performances; b) Threads may not be idled or postponed by the RM. Instead, the goal of the RM is to assign the *currently* available resources to the *currently* running threads.

*1) Static optimization and issues:* A possible centralized objective that we may consider is:

$$\max_{\alpha \in \mathcal{A}} \quad f(\alpha, w) \doteq \textstyle\sum_{i=1}^{n} u_i(\alpha, w)/n, \tag{1}$$

where, for example, $u_i$ may represent the processing speed of thread $i$. In general, $u_i$ will depend on the assignment profile $\alpha$ and exogenous disturbances (e.g., other applications) summarized within $w$. Any solution to the optimization problem (1) corresponds to an *efficient assignment*. However, there are two practical issues when posing an optimization problem in this form: a) the function $u_i(\alpha, w)$ is unknown and it may only be evaluated through measurements of the objective, denoted by $\tilde{u}_i$; and, b) $w$ is also unknown and may vary with time.

*2) Measurement- or learning-based optimization:* We wish to address a *static* objective of the form (1) through a *measurement-* or *learning-based* methodology. That is, the RM reacts to measurements of $f(\alpha, w)$, periodically collected at time instances $k = 1, 2, ...$ and denoted by $\tilde{f}(k)$. The measured objective may take on the form $\tilde{f}(k) \doteq \sum_{i=1}^{n} \tilde{u}_i(k)/n$. Given these measurements and the current assignment $\alpha(k)$ of resources, the RM selects the next assignment of resources $\alpha(k+1)$ so that the measured objective approaches the true optimum of the unknown performance function $f(\alpha, w)$.

*3) Multi-agent formulation:* We further *distribute* the decision-making process into a thread-based optimization, where the RM makes decisions *independently* for each thread. Equivalently, we may assume that each thread makes its own independent decisions as in multi-agent formulations. Such distribution reduces the complexity of the decision-making

process, since each thread has a reduced number of choices as compared to the number of choices of the group of threads. Furthermore, it increases robustness, since any performance degradation noticed in a group of threads can immediately be treated by the affected threads, thus avoiding the complexity of centrally designed assignment corrections.

*4) Multi-level decision-making and actuation:* Recent work by the authors [5], [6] has demonstrated the potential of learning-based optimization in UMA architectures. However, when an application runs on a NUMA machine, additional information can be exploited to enhance scheduling of a parallelized application. To this end, a multi-level decision-making and actuation process is considered. We extend the PaRLSched dynamic scheduler of [5], [6] by introducing two nested decision processes depicted in Figure 1. At the *higher level* (Level 1), the performance of a thread is evaluated with respect to its own prior history of performances, and decisions are taken with respect to its NUMA placement. At the *lower level* (Level 2), the performance of a thread is evaluated with respect to its own prior history of performances, and decisions are taken with respect to its CPU placement (within the selected NUMA node).

## IV. Dynamic Scheduler

Each one of the two levels of the decision process will take place at different frequencies and possibly based on different reasoning. In particular, NUMA-node switching may be costly, especially when performed with high frequency due primarily to memory affinities, while CPU-node switching within the same NUMA node may be costless (with respect to its impact to the processing speed). For this reason, we have introduced two measurement-based learning algorithms specifically tailored to accommodate these different needs (Figure 1):

- ***(Level 1) Aspiration learning for NUMA-node switching***, that responds only to significant performance variations and does not require frequent migrations.
- ***(Level 2) Perturbed learning automata for CPU-core pinning*** within a given NUMA node, that allows frequent CPU-core switches.

We introduce periodic time instances with period $T > 0$, and indexed by $k = 1, 2, ...$, at which decisions at Level 2 (CPU-core pinning) are revised. Decisions at Level 1 (NUMA-node switching) are performed less frequently, at periodic time instances of period $mT$, for some $m \in \mathbb{N}$, which will be indexed by $\ell = 1, 2, ....$

### A. Utility Function

A cornerstone in the design of any such multi-agent formulation is the *preference criterion* or *utility function* $u_i$ for each thread $i \in \mathcal{A}$. The utility function captures the benefit of a decision maker (thread) resulting from the assignment profile $\alpha$, i.e., it represents a function of the form $u_i : \mathcal{A} \to \mathbb{R}_+$ (where we restrict it to be a positive number). The action profile (i.e., the selections of all threads) constitutes a "state" of the environment that directly determines the performances

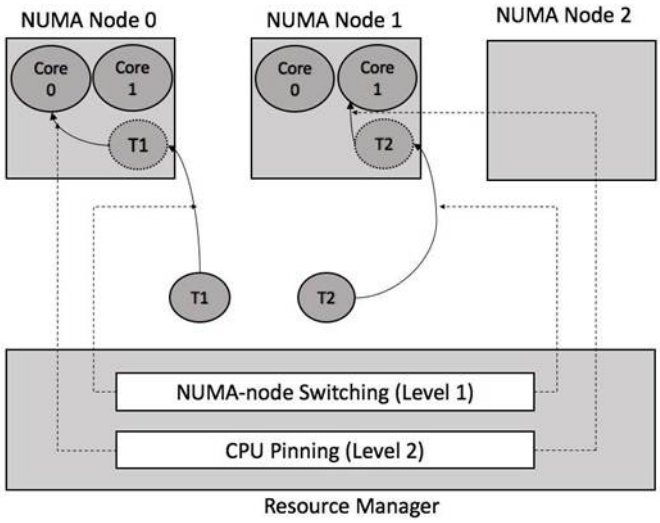


Fig. 1. Two-level scheduling where the RM decides firstly the NUMA node and secondly the CPU core at which each thread should be pinned on.

of all threads. We are interested in building *learning-based reflex* agents that respond only to current measurements in an effort to "eventually" learn to play efficient assignments.

It is important to note that the utility function $u_i$ of each agent/thread $i$ is subject to *design* and it is introduced in order to guide the preferences of each agent. Thus, $u_i$ may not necessarily correspond to a measured quantity, but it could be a function of available performance counters. For example, a natural choice for the utility of each thread is its own execution speed, which can be measured by the number of executed instructions per unit of time. This may also be combined with other counters, e.g., the number of memory-access instructions, the number of cache misses, etc., to give a better representation of the performance of a thread.

### B. Aspiration learning for NUMA-node switching

We developed a novel learning scheme for NUMA-node switching that is based upon the notions of benchmark actions/performances and bears similarities with the so-called *aspiration learning* [16]. The novelty here lies in the introduction of two benchmark levels in order to handle the possibility of noisy measurements. Such type of learning dynamics try to gradually reach assignment profiles where all threads perform well. They have the advantage that exploration (of new assignments) can be performed selectively (e.g., when a significant reduction in performance is observed). In this way, a low-frequency NUMA-node switching can be attained. The specific steps are depicted in Table I.

It is important to note that the above learning scheme will react immediately to a rapid drop in performance. In particular, when the performance drops below the lower benchmark, then with high probability the action will change, while in any other case, the action will change with a small probability $\zeta > 0$. The reason for maintaining both an upper and lower benchmark is in order to minimize the effect of noise in the decision process.

TABLE I
ASPIRATION LEARNING FOR NUMA-NODE SWITCHING

At fixed time instances denoted by $\ell = 1, 2, ...$, the following steps are executed recursively for each thread $i$ in parallel.
(1) **Performance measurement.** For the currently selected NUMA-node $\alpha_i(\ell)$ thread $i$ retrieves its current performance measurement, $\tilde{u}_i(\ell)$.
(2) **Aspiration-level update.** Given the current performance measurement $\tilde{u}_i(\ell)$, update the discounted running average performance of the thread, as follows:

$$\rho_i(\ell+1) = \rho_i(\ell) + \nu \cdot [\tilde{u}_i(\ell) - \rho_i(\ell)], \tag{2}$$

where $\tilde{u}_i(\ell)$ is the current measurement of the utility of thread $i$.
(3) **Benchmarks update.** Define the *upper benchmark performance*, $\overline{b}_i(\ell)$, as a performance threshold over which a performance is considered *satisfactory*, and the *lower benchmark performance*, $\underline{b}_i(\ell)$, as a performance threshold under which a performance is considered *unsatisfactory*, with $\underline{b}_i(\ell) < \overline{b}_i(\ell)$. They are updated as follows:

- if $\rho_i(\ell+1) \geq \overline{b}_i(\ell)$, then

$$\overline{b}_i(\ell+1) = \rho_i(\ell+1)$$
$$\underline{b}_i(\ell+1) = \rho_i(\ell+1)/\eta$$

- if $\underline{b}_i(\ell) \leq \rho_i(\ell+1) < \overline{b}_i(\ell)$, then

$$\overline{b}_i(\ell+1) = \overline{b}_i(\ell)$$
$$\underline{b}_i(\ell+1) = \underline{b}_i(\ell)$$

- if $\rho_i(\ell+1) < \underline{b}_i(\ell)$, then

$$\overline{b}_i(\ell+1) = \eta \cdot \rho_i(\ell+1)$$
$$\underline{b}_i(\ell+1) = \rho_i(\ell+1)$$

for some constant $\eta > 1$.
(4) **Action update.** A thread $i$ selects actions according to the following rule:

a) if $\rho_i(\ell+1) < \underline{b}_i(\ell)$, i.e., if the updated discounted running average performance is unsatisfactory, then thread $i$ will perform a random switch to a better reply, i.e.,

$$\alpha_i(\ell+1) \in \mathrm{rand}_{\mathrm{unif}}\left[\mathrm{BR}_{\mathrm{NUMA},i}(\alpha)\right],$$

where $\mathrm{BR}_{\mathrm{NUMA},i}(\alpha)$ denotes the better-reply of thread $i$ to the assignment $\alpha$, defined as

$$\mathrm{BR}_{\mathrm{NUMA},i}(\alpha) \doteq \left\{\alpha_i' \in \mathcal{J}_{\mathrm{NUMA}} : \rho_i(\ell) < \gamma \frac{\sum_{\{j \in \mathcal{I}: \alpha_j(\ell) = \alpha_i'\}} \rho_j(\ell)}{|\{j \in \mathcal{I} : \alpha_j(\ell) = \alpha_i'\}|}\right\}$$

for some $\gamma \in (0,1)$. The set $\{j \in \mathcal{I} : \alpha_j(\ell-1) = \alpha_i'\}$ includes all those threads that selected action $\alpha_i'$ in the previous time instance. In other words, an action $\alpha_i' \in \mathrm{BR}_{\mathrm{NUMA},i}(\alpha)$ if the average of the threads selecting $\alpha_i'$ did better on average than thread $i$.

b) if $\rho_i(\ell+1) \geq \underline{b}_i(\ell)$ , then each thread $i$ will keep playing the same action with high probability and experiment with any other action with a small probability $\zeta > 0$, i.e.,

$$\alpha_i(\ell+1) = \begin{cases} \alpha_i(\ell), & \text{w.p. } 1-\zeta \\ \mathrm{rand}_{\mathrm{unif}}[\mathrm{BR}_{\mathrm{NUMA},i}(\alpha)], & \text{w.p. } \zeta \end{cases} \tag{3}$$

When the thread needs to select a new NUMA node, it will select among the set of better replies, i.e., nodes at which other threads perform better so far. Note that a thread may not have a-priori knowledge of the exact impact an action switch has on his own utility (until this action switch is performed). However, we may use prior data of the performances of other threads, as defined in $\mathrm{BR}_{\mathrm{NUMA},i}(\alpha)$. Thus, at step (4a), we may direct threads that currently do not perform well to the NUMA nodes where threads perform better.

Similarly to the analysis presented in [16], the process will gradually approach (in probability) allocations with higher utility (even suboptimal). The introduced perturbation $\zeta$ in the action selection provides the necessary randomness for the process to continuously adjust to performance variations.

### C. Perturbed Learning Automata for CPU-core pinning

Let us assume that, at Level 1, and for each one of the running threads $i \in \mathcal{I}$, the RM has already selected a NUMA node $\alpha_i \in \mathcal{J}_{\mathrm{NUMA}}$. Then, at Level 2, the RM needs to decide which CPU-core each thread should be pinned to. Given that CPU-core switching within the same NUMA node is usually costless, we have designed a learning algorithm that allows frequent switching and therefore a faster convergence rate. To this end, we employ a variation of *perturbed learning automata* [17], namely *aspiration-based perturbed learning automata* [18], developed by the authors. Such dynamics perform well in the presence of noise contrary to alternative schemes, as discussed in [17], and can guarantee convergence to at least locally optimal assignments [6].

The basic idea behind learning automata is rather simple. Each agent $i$ keeps track of a strategy vector that holds its estimates over the best choice. We denote this strategy by $\sigma_i = [\sigma_{ij}]_j$, where $j \in \mathcal{J}_{\mathrm{CPU}}(\alpha_i)$, $\sigma_{ij} \geq 0$ and $\sum_j \sigma_{ij} = 1$. To provide an example, consider the case of 3 available CPU cores, i.e., $\mathcal{J}_{\mathrm{CPU}}(\alpha_i) = \{1, 2, 3\}$. In this case, a vector of the form $\sigma_i = (0.2, 0.5, 0.3)$ is a strategy vector, such that $20\%$ corresponds to the probability of assigning itself to CPU-core 1, $50\%$ to CPU-core 2 and $30\%$ to CPU-core 3. Briefly, the CPU-core selection will be denoted by $\beta_i \in \mathcal{J}_{\mathrm{CPU}}(\alpha_i)$. Note that if $\sigma_i$ is a unit vector, say $e_j$, then agent $i$ selects its $j$th action with probability one.

In particular, the steps executed in each iteration of the perturbed learning automata learning dynamics are depicted in Table II. According to this recursion, if the current performance is *satisfactory*, $\tilde{u}_i(k) \geq \rho_i(k)$, i.e., better than the discounted running average performance, then $\phi(\tilde{u}_i(k) - \rho_i(k)) = 1$, and the new nominal strategy will increase in the direction of the currently selected action $\alpha_i(k)$ and proportionally to the utility received from this selection, $\tilde{u}_i(k)$. In case of an *unsatisfactory* response, i.e., $\tilde{u}_i(k) < \rho_i(k)$, then the corresponding strategy will increase proportionally to $h > 0$, which should be taken sufficiently small. In general, we would like that unsatisfactory responses are not reinforced.

Finally, we should note that the above recursion consists of a two time-scale dynamics, the slow update of $x_i(k)$ and the fast update of $\rho_i(k)$ (by selecting $\mu$ to be larger than $\epsilon$). This is because we would like that the discounted running average performance keeps close track of the strategy updates.

In comparison to our previously considered dynamics in the context of dynamic pinning [6], the difference lies in the reinforcement direction. As Equation (5) dictates, the strategy vector is only adjusted when a performance is higher than the discounted running average performance $\rho_i$, which provides a faster adjustment towards better assignments.

TABLE II
PERTURBED LEARNING AUTOMATA FOR CPU-CORE PINNING

At fixed time instances denoted by $k = 1, 2, ...$, the following steps are executed recursively for each thread $i$ in parallel.
(1) **Performance measurement.** For the currently selected CPU-core $\beta_i(k)$ thread $i$ retrieves its current performance measurement, $\tilde{u}_i(k)$.
(2) **Strategy update.** Given that $\alpha_i$ is the current NUMA-node assignment of thread $i$, and $|\mathcal{J}_{\mathrm{CPU}}(\alpha_i)|$ is the number of the available CPU cores, the strategy of thread $i$ with respect to its CPU-core pinning is defined as:

$$\sigma_i(k) = (1 - \lambda)x_i(k) - \frac{\lambda}{|\mathcal{J}_{\mathrm{CPU}}(\alpha_i)|} \quad (4)$$

where $\lambda > 0$ corresponds to a perturbation term (or *mutation*) and $x_i(k)$ corresponds to the *nominal strategy* of agent $i$. The nominal strategy is updated according to the following update recursion:

$$x_i(k+1) = x_i(k) + \epsilon \cdot \tilde{u}_i(k) \cdot [e_{\beta_i(k)} - x_i(k)] \cdot \phi(\tilde{u}_i(k) - \rho_i(k)) \quad (5)$$

for some constant step-size $\epsilon > 0$, where

$$\phi(y) \doteq \begin{cases} 1, & \text{if } y \geq 0 \\ \max\{h, 1 + y/h\}, & \text{if } y < 0. \end{cases}$$

and $\rho_i(k)$ is the running average performance of thread $i$ that is updated according to the recursion

$$\rho_i(k+1) = \rho_i(k) + \mu \cdot (\tilde{u}_i(k) - \rho_i(k)),$$

for a constant step size $\mu > 0$.
(3) **Action update.** The action of each thread $i$ is updated as follows:

$$\beta_i(k+1) = \mathsf{rand}_{\sigma_i}[\mathcal{J}_{\mathrm{CPU}}(\alpha_i)].$$

## V. EXPERIMENTS

In this section, we present an experimental study of the proposed framework. Experiments were conducted on 28×Intel©Xeon©CPU E5-2650 v3 2.30 GHz running Linux Kernel 64bit 3.13.0-43-generic. The cores are divided into two NUMA nodes (Node 1: 0-13 CPU cores, Node 2: 14-27 CPU cores).

In all experiments, the utility of each thread is defined as the *total instructions completed per second* which incorporates both the computational and memory-access instructions. This is a multi-objective criterion and it is expected that the larger the number of instructions completed, the larger the processing speed of a thread. We compared the overall performance of the application (in terms of processing speed of threads and completion time of an application) with that of the Linux OS scheduler. We considered a number of parallel applications under different levels of resource availability (i.e., number of CPU cores available for the applications) and background-load settings (i.e., number of threads of other applications running on the available cores at the same time).

### A. Benchmark applications

In particular, we have considered the following benchmark applications:

- *Swaptions* (SWA), that uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions. The HJM framework describes how interest rates evolve for risk management and asset liability management [19]. The application employs Monte-Carlo simulation to compute the prices. It is regular in terms of task sizes, with a low degree of communication between different threads. It was taken from the *Parsec* benchmark suite.
- *Blackscholes* (BLA), that calculates, using differential equations, how the value of an option changes as the price of the underlying asset changes; parallel implementation calculates values for a number of options at the same time, assigning a thread to each option (or a group of options). If the options are equally divided between threads, this results in a regular (in terms of task sizes) parallel application. In practice, similar calculations are used by financial houses to price 10-100 thousands of options. This is *computationally intensive* application as depicted in Table III. It was taken from the *Parsec* benchmark suite.
- *Ant Colony Optimization (ACO)* [20] is a metaheuristic used for solving NP-hard combinatorial optimization problems. In this paper, we apply ACO to the Single Machine Total Weighted Tardiness Problem (SMTWTP). Briefly, this is a scheduling problem of jobs that are characterized by varying processing times, deadlines and weights. The objective is to find the schedule that minimizes the total tardiness. A detailed description of this use case is provided in [5]. This is *computationally intensive* application as depicted in Table III.
- *Stochastic-Local-Search for Cutting-Stock Industrial Optimization (CSO)* that optimizes classical bin-packing and cutting-stock optimization problems using an evolutionary stochastic-local-search (SLS) algorithm. The use case and the type of parallelization (which is based on the Fast-Flow parallelization library [21]) has been described in detail in [22]. In particular, we used the Scholl 1–3 datasets for classical bin packing problems provided in [23]. According to the implemented SLS algorithm, an initial number of candidate solutions (pool) of a bin-packing/cutting-stock problem, are processed continuously through a series of heuristic based operations/modifications (optimization cycle). In each such cycle, multiple threads are assigned a portion of the candidate solutions. Since the application usually runs for a fixed time, the total number of candidate solutions processed in all optimization cycles completed constitutes an indication of the average processing speed.

TABLE III
COMPUTATIONAL/MEMORY INTENSITY OF CASE STUDIES (TOT_INS = TOTAL INSTRUCTIONS, LST_INS = LOAD/STORE INSTRUCTIONS, TLB_DM = DATA TRANSLATIONS)

| Index | BLA | SWA | ACO | CSO |
|---|---|---|---|---|
| TOT_INS / LST_INS | $\mathcal{O}(10^{+7})$ | $\mathcal{O}(10^{+6})$ | $\mathcal{O}(10^{+5})$ | $\mathcal{O}(10^{+2})$ |
| TLB_DM / LST_INS | $\mathcal{O}(10^{-7})$ | $\mathcal{O}(10^{-6})$ | $\mathcal{O}(10^{-5})$ | $\mathcal{O}(10^{-2})$ |

### B. Experimental setup

The period of the CPU pinning is fixed to $0.05$ sec, which is also the interval in which the RM collects measurements of the *total instructions completed per sec* (using the PAPI library

TABLE IV
ALGORITHM SETTINGS

| Parameter | Value |
|---|---|
| $\epsilon$ | $0.2/\rho_i/10^8$ |
| $\lambda$ | 0.005 |
| $\mu$ | 0.01 |
| $\nu$ | 0.01 |
| $\zeta$ | 0.2 |
| $\gamma$ | 0.9 |
| $\eta$ | 0.8 |
| $T$ | 50 |

[24]) for each one of the threads separately. In other words, the *utility* $u_i$ of thread $i$ corresponds to the total instructions completed per sec for thread $i$.

Pinning of threads to CPU cores is achieved through the `sched.h` library. In all experiments, the RM is executed by the master thread of an application, which is always running in a fixed CPU core (usually the first available CPU core of the first NUMA node).

In Table V, we provide an overview of the conducted experiments. We classify the experiments with respect to the resource availability and the CPU availability. We classify the resource availability as *small* (around 4 application threads per CPU core), *medium* (1 thread per CPU core) and *high* (0.2 threads per CPU core). We classify the CPU availability as *uniform*, when no background applications are running and therefore all CPU cores are fully available to the tested application, *non-uniform* where 8 threads of a background application are running on the first 8 CPU cores of the machine for the whole duration of the running of the tested application and *time-varying*, where initially the availability is uniform, but after some time background application starts running on 8 cores, as in the *non-uniform* case.

Our goal is to investigate the performance of the scheduler under different set of available resources, and how the dynamic scheduler adapts to background load.

TABLE V
CLASSIFICATION OF THE EXPERIMENTS.

| Exp. | Resource availability | CPU availability |
|---|---|---|
| A.1 | Small | Uniform |
| A.2 | Small | Non-uniform |
| A.3 | Small | Time-varying |
| B.1 | Medium | Uniform |
| B.2 | Medium | Non-uniform |
| B.3 | Medium | Time-varying |
| C.1 | Large | Uniform |
| C.2 | Large | Non-uniform |
| C.3 | Large | Time-varying |

### C. Experimental Results

Tables VI–IX show the execution times of the four chosen applications under OS and PaRLSched scheduler and under the experimental scenarios of Table V. Below, we analyze each application separately.

TABLE VI
COMPLETION TIMES OF OS AND PaRLSched SCHEDULING FOR SWAPTIONS APPLICATION. WE SHOW THE MEAN EXECUTION TIME OF THE APPLICATION, MEAD DEVIATION AND IMPROVEMENT IN EXECUTION TIME OF PaRLSchedOVER OSSCHEDULING

| Exp/ Resources | OS | | PaRLSched | | Diff. (%) |
|---|---|---|---|---|---|
| | Mean | Dev | Mean | Dev | |
| SWO (A.1) | **225.58** | 1.28 | **225.27** | 1.41 | $+\mathbf{0.13}$ |
| SWO (A.2) | **385.75** | 17.00 | **344.53** | 3.38 | $+\mathbf{10.69}$ |
| SWO (A.3) | **337.46** | 14.62 | **311.17** | 2.98 | $+\mathbf{7.79}$ |
| SWO (B.1) | **163.40** | 0.56 | **158.10** | 2.20 | $+\mathbf{3.25}$ |
| SWO (B.2) | **289.31** | 5.93 | **285.68** | 5.28 | $+\mathbf{1.26}$ |
| SWO (B.3) | **240.81** | 5.22 | **238.05** | 4.89 | $+\mathbf{1.15}$ |
| SWO (C.1) | **122.54** | 0.79 | **129.85** | 3.25 | $-\mathbf{5.96}$ |
| SWO (C.2) | **206.68** | 1.94 | **202.85** | 2.83 | $+\mathbf{1.85}$ |
| SWO (C.3) | **164.11** | 1.49 | **161.54** | 3.19 | $+\mathbf{1.57}$ |

*a) SWA:* We observe that the PaRLSched scheduler exhibits better behavior than the OS under small and medium availability of resources (i.e., categories A and B) with or without background interference. The improvement varies between 0.13% and 10.69%. In case of large availability of resources (i.e., category C), the OS outperforms the PaRLSched but only in the case where there is no background interference. Note also that the percentages of the mean deviations are significantly smaller than the corresponding performance differences (except for the A.1 case), thus we may not attribute these differences to noise.

*b) ACO:* In this set of experiments, we see a similar behavior to the SWA experiments. The PaRLSched outperforms the OS in the case of small and medium availability of resources and in the presence of background interference (i.e., categories A.2–A.3 and B.2–B.3). The improvement may reach up to 16.92%. In the absence of any background interference, the behavior under small availability of resources (i.e., category A.1) is about equivalent, while in the remaining categories the OS outperforms the PaRLSched scheduler.

As a side note, we should mention that even under scenarios where the OS outperformed PaRLSched, such as scenario C.3, the average speed over all threads is not necessarily smaller, as Figure 2 demonstrates. In other words, the PaRLSched does indeed achieve a good level of the average speed, which is its design criterion, but apparently completion time is not only a matter of average speed. For example, a large average speed over all threads does not necessarily guarantee that all threads are running with identical speeds. Instead, there might be significant differences in the speeds between threads, which may have an impact on the overall completion time.

*c) BLA:* The performance under the Blackscholes application is not deviating significantly in comparison with the conclusions of ACO and SWA applications. In fact, we observe a constantly better performance of the PaRLSched in conditions of small resource availability which may reach up to 4.05% improvement. On the other hand, the performance under large resource availability has been up to -8.89% worse than the OS performance.

TABLE VII
COMPLETION TIMES OF OS AND PaRLSched SCHEDULING FOR ACO APPLICATION. WE SHOW THE MEAN EXECUTION TIME OF THE APPLICATION, MEAD DEVIATION (IN SECONDS) AND AVERAGE PROCESSING SPEED PER THREAD (IN $10^8$ INSTRUCTIONS PER SECOND).

| Exp/ Time(s) | OS | | | PaRLSched | | | Diff. (%) |
|---|---|---|---|---|---|---|---|
| | Mean | Dev | Avg. Spd. | Mean | Dev | Avg. Spd | |
| ACO (A.1) | **1065.05** | 7.68 | 13.37 | **1075.48** | 6.45 | 14.87 | **−0.9** |
| ACO (A.2) | **1752.46** | 14.00 | 8.54 | **1455.92** | 22.8 | 9.82 | **+16.92** |
| ACO (A.3) | **1459.18** | 9.42 | 10.29 | **1402.00** | 4.06 | 10.41 | **+3.91** |
| ACO (B.1) | **673.09** | 5.69 | 21.26 | **699.16** | 10.24 | 22.16 | **−3.87** |
| ACO (B.2) | **1106.36** | 16.71 | 12.73 | **1041.33** | 16.71 | 14.94 | **+5.87** |
| ACO (B.3) | **1066.18** | 0.88 | 13.37 | **1019.11** | 8.39 | 14.87 | **+4.41** |
| ACO (C.1) | **455.87** | 5.08 | 31.90 | **496.26** | 5.08 | 33.46 | **−8.85** |
| ACO (C.2) | **659.78** | 27.45 | 21.57 | **688.80** | 18.66 | 24.15 | **−4.39** |
| ACO (C.3) | **659.35** | 3.62 | 21.82 | **676.03** | 7.72 | 23.72 | **−2.52** |

TABLE VIII
COMPLETION TIMES OF OS AND PaRLSched SCHEDULING FOR BLACKSCHOLES (BLA) APPLICATION

| Exp/ Resources | OS | | PaRLSched | | Diff. (%) |
|---|---|---|---|---|---|
| | Mean | Dev | Mean | Dev | |
| BLA (A.1) | **193.20** | 1.89 | **190.43** | 0.62 | **+1.09** |
| BLA (A.2) | **322.32** | 4.98 | **314.73** | 8.40 | **+2.36** |
| BLA (A.3) | **285.76** | 4.17 | **274.17** | 7.30 | **+4.05** |
| BLA (B.1) | **129.98** | 1.09 | **129.88** | 1.31 | **+0.08** |
| BLA (B.2) | **236.62** | 4.18 | **245.09** | 2.64 | **−3.58** |
| BLA (B.3) | **192.45** | 5.16 | **200.15** | 4.46 | **−4.00** |
| BLA (C.1) | **98.97** | 1.11 | **107.77** | 1.25 | **−8.89** |
| BLA (C.2) | **166.50** | 1.46 | **172.65** | 3.00 | **−3.69** |
| BLA (C.3) | **130.24** | 2.13 | **135.42** | 3.87 | **−3.98** |

TABLE IX
CANDIDATE SOLUTIONS PROCESSED UNDER OS AND PaRLSched SCHEDULING FOR CSO APPLICATION WITHIN 5MIN SIMULATION TIME

| Exp/ Resources | OS | | PaRLSched | | Diff. (%) |
|---|---|---|---|---|---|
| | Mean | Dev | Mean | Dev | |
| CSO (C.1) | **494.3** | 23.46 | **534.00** | 39.22 | **+8.03** |
| CSO (C.2) | **494.2** | 36.50 | **507.9** | 33.37 | **+2.75** |
| CSO (C.3) | **521.50** | 35.58 | **517.7** | 30.65 | **−0.73** |

*d) CSO:* The CSO application is a bit different than the ones previously considered. It is characterized by scattered memory pages as Table III reflects. In particular, under large availability of resources (set C), we see a significant advantage of the PaRLSched scheduler in the absence of any background application that reaches up to $8.03\%$. This advantage dies away under the presence of background applications. A possible explanation of this type of behavior should be attributed to an allocation of a large amount of memory that the application needs to do at the beginning (in the first available NUMA node), which creates uneven processing speeds between the two available NUMA nodes.

## D. Discussion

In general, we observed that the PaRLSched scheduler was able to achieve better performance that the OS scheduler in most cases of limited availability of resources and high background (exogenous) interference load. This is somewhat ideal settings for the scheduler, since we expect the performance of individual threads to vary due to external influences and, therefore, it is important to make the correct remapping decisions. Additionally, it is not possible to predict this variation solely based on the characteristics of the application itself. Also, in the data-intensive application (CSO), the scheduler was able to better adapt to the irregularity in the memory-access speeds between the two NUMA nodes (even under large availability of resources).

On the other hand, the OS outperformed the PaRLSched scheduler in most cases of large availability of resources (e.g., category C.1). This should be attributed to the fact that the Linux scheduler is utilizing internal load balancing of threads

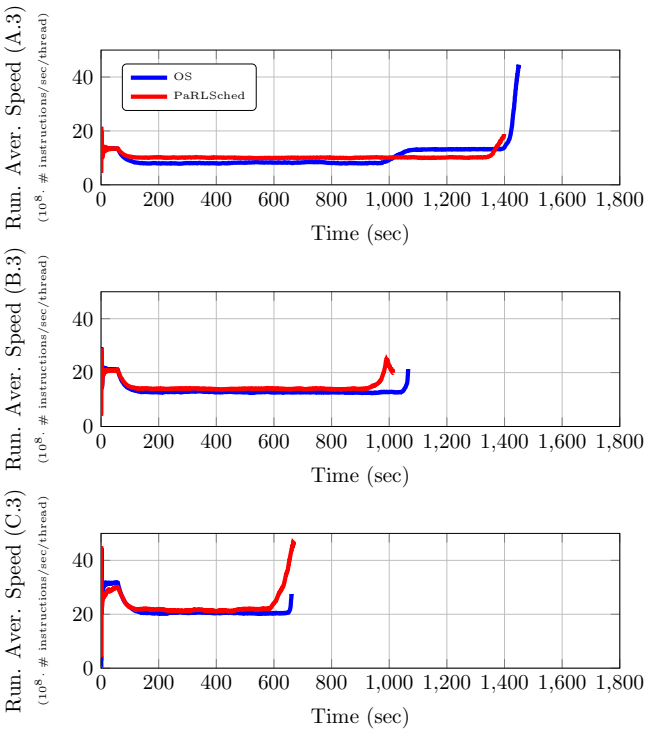


Fig. 2. Sample responses for Experiments of category 3 (i.e., under time-varying CPU availability. The running average speed is measured in ($10^8$· # instructions/sec/thread).

between cores, which has notable effect on the execution time when there is not significant background interference (in terms of additional running applications). In this case, performance of the individual threads depends exclusively on the distribution of threads of the application to cores, so there is no additional benefit in measuring external interference in the PaRLSched scheduler. The PaRLSched scheduler applies rigid pinning of threads to cores, which means that it cannot utilize any internal load balancing by the Linux scheduler.

Given the rather diverse nature of the considered applications, the observed improvements constitute a promising indication. Note that the intention and goal of this work is not to replace the OS scheduler, but instead to act on a supervisory level, and possibly under alternative multi-objective criteria. The notion of the utility function that drives the thread placement can be designed to accommodate any such multi-objective criterion, since the only assumption considered is the positivity constraint.

## VI. Conclusions and future work

We proposed a measurement- (or performance-) based learning scheme for addressing the problem of efficient dynamic pinning of parallelized applications into many-core systems under a NUMA architecture. According to this scheme, a centralized objective is decomposed into thread-based objectives, where each thread is assigned its own utility function. Allocation decisions were organized into a hierarchical decision structure: at the first level, decisions are taken with respect to the assigned NUMA node, while at the second level, decisions are taken with respect to the assigned CPU core (within the selected NUMA node). The proposed framework is flexible enough to accommodate any multi-objective criterion, while it is appropriately designed to handle noisy observations.

We demonstrated the utility of the proposed framework in the maximization of the running average processing speed of the threads and we evaluated its performance in four benchmark parallel applications. We have concluded that the PaRLSched scheduler can achieve better running speed in certain cases, especially of small availability of resources or large background load. These observations should be further reinforced with additional benchmark tests. In addition, we plan to identify and generalize the indicators that trigger these advantageous responses of the PaRLSched scheduler and also to consider additional utility functions, such as register count of each thread.

## References


[1] M. Danelutto, "On skeletons and design patterns," in *Proc. of Intl. ParCo 2001*, ser. Parallel Computing: Advances and Current Issues, G. Joubert, A. Murli, F. Peters, and M. Vanneschi, Eds. Imperial College Press, 2001, pp. 425–432.

[2] M. Aldinucci, G. P. Pezzi, M. Drocco, C. Spampinato, and M. Torquati, "Parallel visual data restoration on multi-gpgpus using stencil-reduce pattern," *The International Journal of High Performance Computing Applications*, vol. 29, no. 4, pp. 461–472, 2015.

[3] D. del Rio Astorga, M. F. Dolz, J. Fernndez, and J. D. Garca, "A generic parallel pattern interface for stream and data processing: A generic parallel pattern interface for stream and data processing," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 24, Dec. 2017.

[4] V. Janjic, C. Brown, K. Mackenzie, K. Hammond, M. Danelutto, M. Aldinucci, and J. D. Garcia, "Rpl: A domain-specific language for designing and implementing parallel c++ applications," in *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, Feb 2016, pp. 288–295.

[5] G. C. Chasparis, M. Rossbory, and V. Janjic, *Efficient Dynamic Pinning of Parallelized Applications by Reinforcement Learning with Applications*, ser. Lecture Notes in Computer Science, F. F. Rivera, T. F. Pena, and J. C. Cabaleiro, Eds. Springer International Publishing, 2017, vol. 10417.

[6] G. C. Chasparis and M. Rossbory, "Efficient Dynamic Pinning of Parallelized Applications by Distributed Reinforcement Learning," *Int. J. Parallel Program.*, pp. 1–15, 2017.

[7] A. Podzimek, L. Bulej, L. Y. Chen, W. Binder, and P. Tuma, "Analyzing the Impact of CPU Pinning and Partial CPU Loads on Performance and Energy Efficiency," in *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2015, pp. 1–10.

[8] B. Goglin, "Managing the topology of heterogeneous cluster nodes with hardware locality (hwloc)," in *International Conference on High Performance Computing and Simulation (HPCS)*, 2014, pp. 74–81.

[9] T. Klug, M. Ott, J. Weidendorfer, and C. Trinitis, "`autopin` - automated optimization of thread-to-core pinning on multicore systems," in *Transactions on High-Performance Embedded Architectures and Compilers III*, ser. Lecture Notes in Computer Science, P. Stenstrom, Ed. Springer Berlin Heidelberg, 2011, vol. 6590, pp. 219–235.

[10] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, and R. Namyst, "ForestGOMP: An efficient OpenMP environment for NUMA architectures," *International Journal Parallel Programming*, vol. 38, pp. 418–439, 2010.

[11] S. Olivier, A. Porterfield, and K. Wheeler, "Scheduling task parallelism on multi-socket multicore systems," in *ROSS'11*, Tuscon, Arizona, USA, 2011, pp. 49–56.

[12] M. Castro, L. F. W. Goes, C. P. Ribeiro, M. Cole, M. Cintra, and J.-F. Mehaut, "A machine learning-based approach for thread mapping on transactional memory applications," in *2011 18th International Conference on High Performance Computing*, 2011, pp. 1–10.

[13] R. Subrata, A. Y. Zomaya, and B. Landfeldt, "A cooperative game framework for QoS guided job allocation schemes in grids," *IEEE Transactions on Computers*, vol. 57, no. 10, pp. 1413–1422, Oct. 2008.

[14] H. Tembine, E. Altman, R. ElAzouri, and Y. Hayel, "Correlated evolutionary stable strategies in random medium access control," in *Int. Conf. Game Theory for Networks*, 2009, pp. 212–221.

[15] G. Wei, A. V. Vasilakos, Y. Zheng, and N. Xiong, "A game-theoretic method of fair resource allocation for cloud computing services," *The Journal of Supercomputing*, vol. 54, no. 2, pp. 252–269, Nov. 2010.

[16] G. C. Chasparis, A. Arapostathis, and J. S. Shamma, "Aspiration learning in coordination games," *SIAM J. Control and Optim.*, vol. 51, no. 1, 2013.

[17] G. C. Chasparis, "Stochastic stability of reinforcement learning in positive-utility games," *(accepted to) IEEE Trans. Autom. Control*, 2017, arXiv:1709.05859 [cs].

[18] ——, "Aspiration-based perturbed learning automata," in *European Control Conference*, Limassol, Cyprus, 2018.

[19] D. Heath, R. Jarrow, and A. Morton, "Bond pricing and the term structure of interest rates: A new methodology for contingent claims valuation," *Econometrica*, vol. 60, no. 1, pp. 77–105, Jan. 1992.

[20] M. Dorigo and T. Stützle, *Ant Colony Optimization*. Scituate, MA, USA: Bradford Company, 2004.

[21] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati, "Pool Evolution: A Parallel Pattern for Evolutionary and Symbolic Computing," *International Journal of Parallel Programming*, vol. 44, no. 3, pp. 531–551, Jun. 2016.

[22] G. C. Chasparis, M. Rossbory, and V. Haunschmid, "An evolutionary stochastic-local-search framework for one-dimensional cutting-stock problems," *arXiv*, vol. 1707.08776, 2017.

[23] M. Delorme, M. Iori, and S. Martello. (2018) A bin packing problem library. [Online]. Available: http://or.dei.unibo.it/library/bpplib

[24] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A portable interface to hardware performance counters," in *Proceedings of the Department of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.