Efficient Dynamic Pinning of Parallelized Applications by Distributed Reinforcement Learning

Georgios C. Chasparis · Michael Rossbory

Abstract This paper introduces a resource allocation framework specifically tailored for addressing the problem of dynamic placement (or pinning) of parallelized applications to processing units. Under the proposed setup each thread of the parallelized application constitutes an independent decision maker (or agent), which (based on its own prior performance measurements and its own prior CPU-affinities) decides on which processing unit to run next. Decisions are updated recursively for each thread by a resource manager/scheduler which runs in parallel to the application’s threads and periodically records their performances and assigns to them new CPU affinities. For updating the CPU-affinities, the scheduler uses a distributed reinforcement-learning algorithm, each branch of which is responsible for assigning a new placement strategy to each thread. The proposed framework is flexible enough to address alternative optimization criteria, such as maximum average processing speed and minimum speed variance among threads. We demonstrate analytically that convergence to locally-optimal placements is achieved asymptotically. Finally, we validate these results through experiments in Linux platforms.

Keywords Dynamic pinning · Reinforcement learning · Parallel applications

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.
1 Introduction

Resource allocation has become an indispensable part of the design of any engineering system that consumes resources, such as electricity power in home energy management [7], access bandwidth and battery life in wireless communications [8], computing bandwidth under certain QoS requirements [1], computing bandwidth for time-sensitive applications [4], computing bandwidth and memory in parallelized applications [2].

When resource allocation is performed online and the number, arrival and departure times of the tasks are not known a priori (as in the case of CPU bandwidth allocation), the role of a resource manager (RM) is to guarantee an efficient operation of all tasks by appropriately distributing resources. However, guaranteeing efficiency through the adjustment of resources requires the formulation of a centralized optimization problem (e.g., mixed-integer linear programming formulations [1]), which further requires information about the specifics of each task (i.e., application details). Such information may not be available to neither the RM nor the task itself.

Given the difficulties involved in the formulation of centralized optimization problems in resource allocation, not to mention their computational complexity, feedback from the running tasks in the form of performance measurements may provide valuable information for the establishment of efficient allocations. Such (feedback-based) techniques have recently been considered in several scientific domains, such as in the case of application parallelization (where information about the memory access patterns or affinity between threads and data are used in the form of scheduling hints) [3], or in the case of allocating virtual processors to time-sensitive applications [4].

To this end, 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 alternative optimization criteria. In particular, we demonstrate its utility in maximizing the average processing speed of the overall application. The proposed scheme also reduces computational complexity usually encountered in centralized optimization problems, while it provides an adaptive response to the variability of the provided resources.

The paper is organized as follows. Section 2 discusses the related work and contribution of this paper. Section 3 describes the overall problem formulation and objective. Section 4 introduces the concept of multi-agent formulations and discusses their advantages. Section 5 presents the proposed reinforcement-learning algorithm for dynamic placement of threads and Sect. 6 presents its convergence analysis. Section 7 presents experiments of the proposed resource manager in a Linux platform and comparison tests with the operating system’s response. Finally, Sect. 8 presents concluding remarks.

Notation:

- $|x|$ denotes the Euclidean norm of a vector $x \in \mathbb{R}^n$.
- $\text{dist}(x, A)$ denotes the minimum distance from a vector $x \in \mathbb{R}^n$ to a set $A \subset \mathbb{R}^n$, i.e., $\text{dist}(x, A) = \inf_{y \in A} |x - y|$.
26 Int J Parallel Prog (2019) 47:24–38

- $B_δ(A)$ denotes the $δ$-neighborhood of a set $A \subset \mathbb{R}^n$, i.e., $B_δ(A) \doteq \{ x \in \mathbb{R}^n : \text{dist}(x,A)<δ \}$.
- For some finite set $A$, $|A|$ denotes the cardinality of $A$.
- The probability simplex of dimension $n$ is denoted by $\Delta(n)$ and defined as $\Delta(n) \doteq \{ x = (x_1, \ldots, x_n) \in [0,1]^n : \sum_{i=1}^n x_i = 1 \}$.
- $e_j \in \mathbb{R}^n$ denotes the unit vector whose $j$th entry is equal to 1 while all other entries are zero.
- For a vector $\sigma \in \Delta(n)$, let $\text{rand}_\sigma[a_1, \ldots, a_n]$ denote the random selection of an element of the set $\{a_1, \ldots, a_n\}$ according to the distribution $\sigma$;

2 Related Work and Contribution

To tackle the issues of centralized optimization techniques, resource allocation problems have also been addressed through distributed or game-theoretic optimization schemes. The main goal of such approaches is to address a centralized (global) objective for resource allocation through agent-based (local) objectives, where, for instance, agents may represent the tasks to be allocated. Examples include the cooperative game formulation for allocating bandwidth in grid computing [14], the non-cooperative game formulation in the problem of medium access protocols in communications [15] or for allocating resources in cloud computing [17]. The main advantage of distributing the decision-making process is the considerable reduction in computational complexity (a group of $n$ tasks can be allocated to $m$ resources with $m^n$ possible ways, while a single task may be allocated with only $m$ possible ways). This further allows for the development of online selection rules where tasks/agents make decisions often using current observations of their own performance.

Prior work has demonstrated the importance of thread-to-core bindings in the overall performance of a parallelized application. For example, [9] describes a tool that checks the performance of each of the available thread-to-core bindings and searches an optimal placement. Unfortunately, the exhaustive-search type of optimization that is implemented may prohibit runtime implementation. Reference [3] combines the problem of thread scheduling with scheduling hints related to thread-memory affinity issues. These hints are able to accommodate load distribution given information for the application structure and the hardware topology. The HWLOC library is used to perform the topology discovery which builds a hierarchical architecture consisting of hardware objects (NUMA nodes, sockets, caches, cores, etc.), and the BubbleSched library [16] is used to implement scheduling policies. A similar scheduling policy is also implemented by [13].

Contrary to this line of research, this paper proposes a dynamic (learning-based) scheme for optimally allocating threads of a parallelized application into a set of available CPU cores. The proposed methodology implements a distributed reinforcement learning algorithm (executed in parallel by a resource manager/scheduler), according to which each thread is considered as an independent agent making decisions over its own CPU-affinities. The proposed algorithm requires minimum information exchange, that is only the performance measurements collected from each running thread. In the current setup, we provide a design that maximizes average processing
speed, however it can be easily modified to accommodate alternative criteria. Furthermore, it exhibits adaptivity and robustness to possible irregularities in the behavior of a thread or to possible changes in the availability of resources. We analytically demonstrate that the reinforcement-learning scheme asymptotically learns a locally-optimal allocation, while it is flexible enough to accommodate several optimization criteria. We also demonstrate through experiments in a Linux platform that the proposed algorithm outperforms the scheduling strategies of the operating system with respect to its average processing speed and completion time.

3 Problem Formulation and Objective

3.1 Framework

We consider a resource allocation framework for addressing the problem of dynamic pinning of parallelized applications. In particular, we consider a number of threads \( I = \{1, 2, \ldots, n\} \) resulting from a parallelized application. These threads need to be pinned for processing into a set of available CPU cores \( J = \{1, 2, \ldots, m\} \) (not necessarily homogeneous).

We denote the assignment of a thread \( i \) to the set of available CPU cores by \( \alpha_i \in \mathcal{A}_i \equiv J \), i.e., \( \alpha_i \) designates the number of the CPU where this thread is being assigned to. Let also \( \alpha = \{\alpha_i\}_i \) denote the assignment profile.

Responsible for the assignment of CPU cores into the threads is the Resource Manager (RM), which periodically checks the prior performance of each thread and makes a decision over their next CPU placements so that a (user-specified) objective is maximized. 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 performance (e.g., their processing speed).
(b) Threads may not be idled or postponed. Instead, the goal of the RM is to assign the currently available resources to the currently running threads.
(c) Each thread may only be assigned to a single CPU core.

3.2 Static Optimization and Issues

Let \( v_i = v_i(\alpha, w) \) denote the processing speed of thread \( i \) which depends on both the assignment profile \( \alpha \), as well as exogenous parameters aggregated within \( w \). The exogenous parameters \( w \) summarize, for example, the impact of other applications running on the same platform (disturbances). Then, the previously mentioned centralized objectives may take on the following form:

\[
\max_{\alpha \in \mathcal{A}} f(\alpha, w). \tag{1}
\]

We consider two objectives:
Fig. 1 Schematic of static resource allocation framework

(O1) \( f(\alpha, w) = \sum_{i=1}^{n} v_i / n \), corresponds to the average processing speed of all threads;

(O2) \( f(\alpha, w) = \sum_{i=1}^{n} [v_i - \gamma (v_i - \sum_{j \in \mathcal{I}} v_j / n)^2] / n \), for some \( \gamma > 0 \), corresponds to the average processing speed minus a penalty that is proportional to the speed variance among threads.

Any solution to the optimization problem (1) will correspond to an efficient assignment. Figure 1 presents a schematic of a static resource allocation framework sequence of actions where the centralized objective (1) is solved by the RM once and then it communicates the optimal assignment to the threads.

However, there are two practical issues when posing an optimization problem in the form of (1). In particular,

1. the function \( v_i(\alpha, w) \) is unknown and it may only be evaluated through measurements of the processing speed, denoted \( \tilde{v}_i \);
2. the exogenous disturbances \( w = (w_1, \ldots, w_m) \) are unknown and may vary with time, thus the optimal assignment may not be fixed with time.

3.3 Measurement- or Learning-Based Optimization

We wish to address a static objective of the form (1) through a measurement- or learning-based optimization approach. According to such approach, the RM reacts to measurements of the objective function \( f(\alpha, w) \), periodically collected at time instances \( k = 1, 2, \ldots \) and denoted \( \tilde{f}(k) \). For example, in the case of objective (O1), \( \tilde{f}(k) = \sum_{i=1}^{n} \tilde{v}_i(k) / n \). 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 \( f(\alpha, w) \). In other words, the RM employs an update rule of the form:
Fig. 2  Schematic of dynamic resource allocation framework

\[
\{(\tilde{v}_i(1), \alpha_i(1)), \ldots, (\tilde{v}_i(k), \alpha_i(k))\}_i \mapsto \{\alpha_i(k + 1)\}_i
\]  

according to which prior pairs of measurements and assignments for each thread \(i\) are mapped into a new assignment \(\alpha_i(k + 1)\) that will be employed during the next evaluation interval.

The overall framework is illustrated in Fig. 2 describing the flow of information and steps executed. In particular, at any given time instance \(k = 1, 2, \ldots\), each thread \(i\) communicates to the RM its current processing speed \(\tilde{v}_i(k)\). Then the RM updates the assignments for each thread \(i\), \(\alpha_i(k + 1)\), and communicates this assignment to them.

### 3.4 Objective

The objective in this paper is to address the problem of adaptive or dynamic pinning through a distributed learning framework. Each thread will constitute an independent decision maker or agent, thus naturally introducing a multi-agent formulation. Each thread selects its own CPU assignment independently using its own preference criterion (although the necessary computations for such selection are executed by the RM). The advantages are two-folded: (a) it reduces computational complexity, since each thread has only \(m\) available choices (instead of \(m^n\) available group choices), and (b) it allows for a faster response to changes in resource availability.

The goal is to design a preference criterion and a selection rule for each thread, so that when each thread tries to maximize its own (local) criterion then certain guarantees can be achieved regarding the overall (global) performance of the parallelized application.

In the following sections, we will go through the design for such a distributed scheme, and we will provide guarantees with respect to its asymptotic behavior and robustness.
4 Multi-agent (or Game) Formulation

The first step towards a distributed learning scheme is the decomposition of the decision making process into multiple decision makers (or agents). Naturally, in the problem of placing threads of a parallelized application into a set of available processing units, each thread may be considered as an independent decision maker.

4.1 Strategy

Since each agent (or thread) selects actions independently, we generally assume that each agent’s action is a realization of an independent discrete random variable. Let $\sigma_{ij} \in [0, 1], j \in \mathcal{A}_i$, denote the probability that agent $i$ selects its $j$th action in $\mathcal{A}_i$. If $\sum_{j=1}^{\left|\mathcal{A}_i\right|} \sigma_{ij} = 1$, then $\sigma_i \equiv (\sigma_{i1}, \ldots, \sigma_{i\left|\mathcal{A}_i\right|})$ is a probability distribution over the set of actions $\mathcal{A}_i$ (or strategy of agent $i$). Then $\sigma_i \in \Delta (\left|\mathcal{A}_i\right|)$. To provide an example, consider the case of 3 available CPU cores, i.e., $\mathcal{A}_i = \{1, 2, 3\}$. In this case, the strategy $\sigma_i \in \Delta (3)$ of thread $i$ may take the following form: $\sigma_i = (0.2, 0.5, 0.3)$, such that 0.2 corresponds to the probability of assigning itself to CPU core 1, 0.5 corresponds to the probability of assigning itself to CPU core 2 and 0.3 corresponds to the probability of assigning itself to CPU core 3. Briefly, the assignment selection will be denoted by $\alpha_i = \text{rand}_{\sigma_i} \left[\mathcal{A}_i\right]$.

We will also use the term strategy profile to denote the combination of strategies of all agents $\sigma = (\sigma_1, \ldots, \sigma_n) \in \Delta$ where $\Delta \equiv \Delta (\left|\mathcal{A}_1\right|) \times \cdots \times \Delta (\left|\mathcal{A}_n\right|)$ is the set of strategy profiles.

Note that if $\sigma_i$ is a unit vector (or a vertex of $\Delta (\left|\mathcal{A}_i\right|)$), say $e_j$, then agent $i$ selects its $j$th action with probability one. Such a strategy will be called pure strategy. Likewise, a pure strategy profile is a profile of pure strategies.

4.2 Utility Function

A cornerstone in the design of any measurement-based algorithm 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 $\alpha$ selected by all threads, i.e., it represents a function of the form $u_i : \mathcal{A} \rightarrow \mathbb{R}_+$ (where we restrict it to be a positive number). Often, we may decompose the argument of the utility function as follows $u_i(\alpha) = u_i(\alpha_i, \alpha_{\bar{i}})$, where $\bar{i} = I \setminus i$. The utility function introduces a preference relation for each decision maker where $u_i(\alpha_i, \alpha_{\bar{i}}) \geq u_i(\alpha'_i, \alpha_{\bar{i}})$ translates to $\alpha_i$ being more desirable/preferable than $\alpha'_i$.

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 $v_i$. Other options may include more egalitarian criteria, where the utility function of each thread corresponds to the overall global objective $f(\alpha, w)$. The definition of a utility function is open-ended.
5 Reinforcement Learning (RL)

We employ a distributed learning framework (namely, perturbed learning automata) that is based on the reinforcement learning algorithm introduced in [5,6]. It belongs to the general class of learning automata [12].

The basic idea behind reinforcement learning is rather simple. If agent $i$ selects action $j$ at instance $k$ and a favorable payoff results, $u_i(\alpha)$, the action probability $\sigma_{ij}(k)$ is increased and all other entries of $\sigma_i(k)$ are decreased.

According to the perturbed learning automata [5,6], the strategy of each thread at any time instance $k = 1, 2, \ldots$ is as follows:

$$\sigma_i(k) = (1 - \lambda)x_i(k) + \frac{\lambda}{|A_i|} \mathbf{1}$$

(3)

where $\lambda > 0$ corresponds to a perturbation term (or mutation), $x_i(k)$ corresponds to the nominal strategy of agent $i$ and $\mathbf{1}$ is a vector of ones of appropriate size. The nominal strategy is updated according to the following update recursion:

$$x_i(k + 1) = x_i(k) + \epsilon \cdot u_i(\alpha(k)) \cdot [e_{\alpha_i(k)} - x_i(k)],$$

(4)

for some constant step size $\epsilon > 0$. Note that according to this recursion, the new nominal strategy will increase in the direction of the action $\alpha_i(k)$ which is currently selected and it will increase proportionally to the utility received. Finally, each agent updates its action by randomizing over the strategy $\sigma_i$, i.e.,

$$\alpha_i(k + 1) = \text{rand}_{\sigma_i}[A_i].$$

In comparison to [5,6], the difference lies in the use of the constant step size $\epsilon > 0$ (instead of a decreasing step-size sequence). This selection increases the adaptivity and robustness of the algorithm to possible changes in the environment. This is because a constant step size provides a fast transition of the nominal strategy from one pure strategy to another.

Furthermore, the reason for introducing the perturbation term $\lambda$ is to provide the possibility for the nominal strategy to escape from pure strategy profiles, that is profiles at which all agents assign probability one in one of the actions. Setting $\lambda > 0$ is essential for providing an adaptive response of the algorithm to changes in the environment.

6 Convergence Analysis

In this section, we establish a connection between the asymptotic behavior of the nominal strategy profile $x(k)$ with the Nash equilibria\(^1\) of the induced assignment game, that is the set of locally stable strategy profiles.

\(^1\) A strategy profile $\sigma^* = (\sigma_1^*, \ldots, \sigma_n^*) \in \Delta$ is a Nash equilibrium if, no agent has incentive to change unilaterally its own strategy, i.e., no agent can increase its expected utility by altering its own strategy.
Let the utility function $u_i$ for each thread $i$ correspond to the global objective (1), i.e., $u_i(\alpha) = f(\alpha, w)$ defined by either (O1) or (O2). Let us denote $S^\lambda$ to be the set of stationary points of the mean-field dynamics (cf., [10]) of the recursion (4), defined as follows

$$S^\lambda = \{x \in \Delta : g^\lambda_i(x) = E[u_i(\alpha(k))[e_{\alpha_i(k)} - x_i(k)]|x(k) = x] = 0, \forall i \in \mathcal{I}\}.$$

The expectation operator $E[\cdot]$ is defined appropriately over the canonical path space $\Omega = \Delta^{\infty}$ with an element $\omega$ being a sequence $\{x(0), x(1), \ldots\}$ with $x(k) = (x_1(k), \ldots, x_n(k)) \in \Delta$ generated by the reinforcement learning process. Similarly we define the probability operator $P[\cdot]$. In other words, the set of stationary points corresponds to the strategy profiles at which the expected change in the strategy profile is zero.

According to [5,6], a connection can be established between the set of stationary points $S^\lambda$ and the set of Nash equilibria of the induced assignment game. In particular, for sufficiently small $\lambda > 0$, the set of $S^\lambda$ includes only $\lambda$-perturbations of Nash-equilibrium strategies [5,6].

The following proposition is a straightforward extension of [5, Theorem 1] to the case of constant step size.

**Proposition 1** Let the RM employ the strategy update rule (4) and placement selection (3) for each thread $i$. Updates are performed periodically with a fixed period such that $\tilde{v}_i(k) > 0$ for all $i$ and $k$. Let the utility function for each thread $i$ satisfy $u_i(\alpha) = f(\alpha, w)$, under either objective (O1) or (O2), where $\gamma \geq 0$ is small enough such that $u_i(\alpha(k)) > 0$ for all $k$. Then, for some $\lambda > 0$ sufficiently small, there exists $\delta = \delta(\lambda)$, with $\delta(\lambda) \downarrow 0$ as $\lambda \downarrow 0$, such that

$$P\left[\lim_{k \to \infty} \inf \text{dist}(x(k), B_\delta(S^\lambda)) = 0\right] = 1. \quad (5)$$

**Proof** The proof follows the exact same steps of the first part of [5, Theorem 1], where the decreasing step-size sequence is being replaced by a constant $\epsilon > 0$. \hfill \Box

Proposition 1 states that when we select $\lambda$ sufficiently small, the nominal strategy trajectory will be approaching the set $B_\delta(S^\lambda)$ infinitely often with probability one, that is a small neighborhood of the Nash equilibria. We require that the update period is large enough so that each thread is using resources within each evaluation period. Of course, if a thread stops executing then the same result holds but for the updated set of threads.

The following proposition provides a characterization of the stochastically stable outcomes.

**Proposition 2** (Weak convergence to Nash equilibria) Under the hypotheses of Proposition 1, the fraction of time that the nominal strategy profile $x(k)$ spends in $B_\delta(S^\lambda)$ goes to one (in probability) as $\epsilon \to 0$ and $k \to \infty$.

**Proof** The proof follows directly from [10, Theorem 8.4.1] and Proposition 1. \hfill \Box

© Springer
Proposition 2 states that if we take a small step size $\epsilon > 0$, then as the time index $k$ increases, we should expect that the nominal strategy spends the majority of the time within a small neighborhood of the Nash equilibrium strategies. Given that the utility function satisfies $u_i(\alpha) = f(\alpha, w)$, for each $i$, then the set of Nash equilibria includes the set of efficient assignments, i.e., the solutions of (1). Thus, due to Proposition 2, it is guaranteed that the nominal strategies $x_i(k), i \in \mathcal{I}$, will spend the majority of the time in a small neighborhood of locally-optimal assignments, which provides a minimum performance guarantee throughout the running time of the parallelized application.

Note that due to varying exogenous factors ($w$), the Nash-equilibrium assignments may not stay fixed for all future times. The above proposition states that the process will spend the majority of the time within the set of the Nash-equilibrium assignments for as long as this set is fixed. If, at some point in time, this set changes (due to, e.g., other applications start running on the same platform), then the above result continues to hold but for the new set of Nash equilibria. Hence, the process is adaptive to possible performance variations.

7 Experiments

In this section, we present an experimental study of the proposed reinforcement learning scheme for dynamic pinning of parallelized applications. Experiments were conducted on $20 \times$ Intel® Xeon® CPU E5-2650 v3 2.30 GHz running Linux Kernel 64bit 3.13.0-43-generic. The machine divides the physical cores into two NUMA nodes (Node 1: 0-9 CPU cores, Node 2: 10-19 CPU cores).

7.1 Experimental Setup

We consider a computationally intensive routine that executes a fixed number of computations (corresponding to the combinations of $M$ out of a set of $N > M$ numbers). The routine is being parallelized using the pthread.h (C++ POSIX thread library), where each thread is executing a replicate of the above set of computations. The nature of these computations does not play any role and in fact it may vary between threads (as we shall see in the forthcoming experiments). Intentionally, the considered application does not make extensive memory use, since the main focus is the investigation of the utility of reinforcement learning in maximizing the overall processing speed.

Throughout the execution, and with a fixed period of 0.2 s, the RM collects measurements of the total instructions per sec (using the PAPI library [11]) for each one of the threads separately. Given the provided measurements, the update rule of Eq. (4) with the utility function $u_i(\alpha) = f(\alpha, w)$ under (O2) is executed by the RM. Placement of the threads to the available CPU cores is achieved through the sched.h library (in particular, the pthread_setaffinity_np function). In the following, we demonstrate the response of the RL scheme in comparison to the operating system (OS) response (i.e., when placement of the threads is not controlled by the RM). We compare them for different values of $\gamma \geq 0$ in order to investigate the influence of more balanced speeds to the overall running time. In all the forthcoming
Fig. 3 Experiment 1. Running average execution speed when 4 threads run on 3 identical CPU cores. Thread 3 requires half as much computing bandwidth as the rest of the threads, which are identical. The strategies of the threads ($x_1$, $x_2$, $x_3$ and $x_4$) capture the nominal selection probabilities and are generated by the RL scheme with $\gamma = 0.04$. The RL schemes of all threads run with $\epsilon = 0.005$ and $\lambda = 0.005$.

experiments, the RM is executed within the master thread which is always running in the first available CPU (CPU 1).

7.2 Experiment 1: Small Number of Threads and Uniform Availability of Resources

In this experiment, we consider the case of small number of threads and CPU cores. Threads are non-identical with respect to the requested bandwidth, while CPU cores are identical in the amount of provided bandwidth (uniform availability). In particular, one of the threads requires smaller CPU bandwidth than the rest. We should expect that in an optimal setup, threads that require smaller CPU bandwidth should be the ones sharing a CPU core with other threads. On the other hand, threads that require large bandwidth, they should be placed alone.

In particular, in this experiment, Thread 3 requires only half as much computing bandwidth as the rest of the threads (i.e., Threads 1, 2 and 4). The resulting performance is depicted in Fig. 3.

We observe indeed that Thread 1, 2 and 4 (which require larger computing bandwidth) are allocated to different CPU cores (CPU 1, 3 and 2, respectively). On the other hand, Thread 3 is switching between CPU 1 and CPU 3, since both provide almost equal processing bandwidth to Thread 3. In other words, the less demanding thread...
Experiment 2. Running average execution speed when 7 non-identical threads run on 3 CPU cores. Thread 1 and 2 require half as much computing bandwidth as the rest of the threads, which are identical. Thread 3 is joining after the 120s. The RL schemes of all threads run with $\epsilon = 0.005$ and $\lambda = 0.005$ is sharing a CPU core with one of the most demanding threads. Note that this assignment corresponds to a Nash equilibrium (as Proposition 2 states), since there is no thread that can benefit by unilaterally changing its strategy. It is also straightforward to check that this assignment is also efficient, in the sense that no other assignment could increase the overall average speed.

Note, finally, that the difference with the processing speed of the OS scheme is small, although a more balanced processing speed ($\gamma = 0.04$) improved slightly the overall completion time. However, given that the performance index used corresponds to the objective (O2), we may only provide a guarantee with respect to the average processing speed.

7.3 Experiment 2: Small Number of Threads and Non-uniform Availability of Resources

In this experiment, we demonstrate the robustness of the algorithm in a dynamic environment. We consider 7 threads and 3 available CPU cores. The first two threads (Thread 1 and 2) require about half as much computing bandwidth as the rest of the threads. The rest of the threads (Thread 3, 4, 5, 6 and 7) are identical. However, Thread 3 starts running later in time (in particular, after 120s).

Figure 4 illustrates the evolution of the RL-based scheduling scheme under different values of $\gamma$. Again, a faster response of the overall application can be achieved when higher values of $\gamma$ are selected. The difference should be attributed to the fact that the OS fails to distinguish between threads with different bandwidth requirements. Table 1 presents a statistical analysis of these schemes where the speed difference between the RL ($\gamma = 0.04$) and the OS reaches approximately 5% on average.

Note that, in general, a maximization of the running average speed of the threads should not necessarily imply a reduction in the completion time. However, in the current setup of almost identical threads, we should expect that increasing the running average speed increases the chances of improving the completion time.

In both Experiment 1 and 2, we do not utilize any initial smart placement of the threads, rather the initial strategies of the threads correspond to the uniform distribu-
Table 1. Experiment 2. Comparison between the OS performance and RL schemes when $\epsilon = 0.005$ and $\lambda = 0.005$ for different values of $\gamma$

| Run # | OS (s) | RL ($\gamma = 0$) (s) | RL ($\gamma = 0.02$) (s) | RL ($\gamma = 0.04$) (s) |
|-------|--------|----------------------|----------------------|----------------------|
| 1     | 513    | 505                  | 492                  | 489                  |
| 2     | 530    | 506                  | 489                  | 494                  |
| 3     | 536    | 517                  | 518                  | 515                  |
| 4     | 533    | 507                  | 515                  | 509                  |
| 5     | 523    | 502                  | 491                  | 496                  |
| 6     | 513    | 523                  | 501                  | 492                  |
| 7     | 520    | 514                  | 497                  | 492                  |
| 8     | 530    | 518                  | 499                  | 497                  |
| 9     | 520    | 532                  | 500                  | 497                  |
| 10    | 528    | 517                  | 493                  | 492                  |
| Aver. | 524.6  | 514.1                | 499.5                | 497.3                |
| SD    | 8.06   | 9.29                 | 9.85                 | 8.27                 |

This is the main reason that it takes some time for the RM to increase the average speed of the threads. However, we observe that it is eventually able to reach higher speed levels compared to the OS, which explains the shorter completion time.

7.4 Experiment 3: Large Number of Threads in a Dynamic Environment with Multiple NUMA Nodes

In this experiment, we would like to see how the proposed learning scheme scales when we increase the number of threads and available CPU cores. Although the considered application does not make heavy memory use, we allow placement in both available NUMA nodes, thus indirectly considering cache-memory related disturbances. Finally, we utilize an initial round-robin placement of the threads, that intends on enhancing the initial adjustment phase of the RM.

In particular, in this experiment, we consider 25 identical threads that can be placed on 12 CPU cores. The resource availability in these cores is not uniform, given that in the first 6 cores other applications are also running.

In Fig. 5, we demonstrate the evolution of the running average processing speed under the OS scheduler and the learning-based RM. First, we observe that the initial placement of CPU cores makes a significant difference with respect to the overall completion time (compared to the previous experiments). Furthermore, we observe that minimizing variance can again decrease the completion time, which is indeed reasonable in the case of identical threads.

Lastly, it is important to point out that even though we have an increased number of threads and CPU cores, the algorithm scales well. As expected, issues with respect to cache-memory use should not be so apparent in this simulation example, due to the nature of this application.
Fig. 5  Experiment 3. Running average execution speed when 25 identical threads run on 12 CPU cores. The first 6 available CPU cores are also occupied by other applications. The RL schemes of all threads run with $\epsilon = 0.005$ and $\lambda = 0.005$.

Table 2  Experiment 3. Comparison between the OS performance and RL schemes when $\epsilon = 0.005$ and $\lambda = 0.005$ for different values of $\gamma$

| Run # | OS (s) | RL ($\gamma = 0$) (s) | RL ($\gamma = 0.02$) (s) |
|-------|--------|------------------------|--------------------------|
| 1     | 244    | 201                    | 196                      |
| 2     | 243    | 191                    | 196                      |
| 3     | 263    | 203                    | 192                      |
| 4     | 255    | 195                    | 198                      |
| 5     | 254    | 194                    | 193                      |
| 6     | 252    | 199                    | 200                      |
| 7     | 259    | 190                    | 191                      |
| 8     | 276    | 192                    | 195                      |
| 9     | 250    | 197                    | 197                      |
| 10    | 248    | 200                    | 190                      |
| Aver. | 254.44 | 196.78                 | 195.44                   |
| SD    | 9.85   | 4.50                   | 3.37                     |

Table 2 also presents a statistical analysis of these schemes under the current setup. We observe that the completion time difference between the RL schemes and the OS reaches approximately 20% on average.

8 Conclusions

We proposed a measurement-based learning scheme for addressing the problem of efficient dynamic pinning of parallelized applications into processing units. According to this scheme, a centralized objective is decomposed into thread-based objectives, where each thread is assigned its own utility function. A RM updates a strategy for each one of the threads corresponding to its beliefs over the most beneficial CPU placement for this thread. Updates are based on a reinforcement learning rule, where prior actions are reinforced proportionally to the resulting utility. It was shown that, when we appropriately design the threads’ utilities, then convergence to the set of locally optimal assignments is achieved. Besides its reduced computational complexity, the proposed scheme is adaptive and robust to possible changes in the environment.
We demonstrated the utility of the proposed framework in the maximization of the running average processing speed of the threads, which in the case of almost identical threads led to a significant reduction in the completion time. Alternative objectives may also be defined, as long as they consist of measured performance indicators. An interesting future direction would involve the possibility of criteria that also optimize memory placement, something that would enlarge the application domain of the proposed learning scheme.

References

1. Bini, E., Buttazzo, G.C., Eker, J., Schorr, S., Guerra, R., Fohler, G., Årzén, K.E., Vanessa, R., Scordino, C.: Resource management on multicore systems: The ACTORS approach. IEEE Micro 31(3), 72–81 (2011)
2. Brecht, T.: On the importance of parallel application placement in NUMA multiprocessors. In: Proceedings of the Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV). pp. 1–18. San Deigo, CA (1993)
3. Broquedis, F., Furmento, N., Goglin, B., Wacrenier, P.A., Namyst, R.: ForestGOMP: an efficient OpenMP environment for NUMA architectures. Int. J. Parallel Program. 38, 418–439 (2010)
4. Chasparis, G.C., Maggio, M., Bini, E., Årzén, K.E.: Design and implementation of distributed resource management for time-sensitive applications. Automatica 64, 44–53 (2016)
5. Chasparis, G.C., Shamma, J.S., Rantzer, A.: Nonconvergence to saddle boundary points under perturbed reinforcement learning. Int. J. Game Theory 44(3), 667–699 (2015)
6. Chasparis, G., Shamma, J.: Distributed dynamic reinforcement of efficient outcomes in multiagent coordination and network formation. Dyn. Games Appl. 2(1), 18–50 (2012)
7. De Angelis, F., Boaro, M., Fuselli, D., Squartini, S., Piazza, F., Wei, Q.: Optimal home energy management under dynamic electrical and thermal constraints. IEEE Trans. Ind. Inform. 9(3), 1518–1527 (2013)
8. Inaltekin, H., Wicker, S.: A one-shot random access game for wireless networks. In: International Conference on Wireless Networks, Communications and Mobile Computing (2005)
9. Klug, T., Ott, M., Weidendorfer, J., Trinitis, C.: autopin: automated optimization of thread-to-core pinning on multicore systems. In: Stenstrom, P. (ed.) Transactions on High-Performance Embedded Architectures and Compilers III. Lecture Notes in Computer Science, vol. 6590, pp. 219–235. Springer, Berlin (2011)
10. Kushner, H.J., Yin, G.G.: Stochastic Approximation and Recursive Algorithms and Applications, 2nd edn. Springer, New York (2003)
11. Mucci, P.J., Browne, S., Deane, C., Ho, G.: PAPI: a portable interface to hardware performance counters. In: Proceedings of the Department of Defense HPCMP Users Group Conference. pp. 7–10 (1999)
12. Narendra, K., Thathachar, M.: Learning Automata: An Introduction. Prentice-Hall, Upper Saddle River (1989)
13. Olivier, S., Porterfield, A., Wheeler, K.: Scheduling task parallelism on multi-socket multicore systems. In: ROSS’11. pp. 49–56. Tuscon, Arizona, USA (2011)
14. Subrata, R., Zomaya, A.Y., Landfeldt, B.: A cooperative game framework for QoS guided job allocation schemes in grids. IEEE Trans. Comput. 57(10), 1413–1422 (2008)
15. Tembine, H., Altman, E., ElAzouri, R., Hayel, Y.: Correlated evolutionary stable strategies in random medium access control. In: International Conference on Game Theory for Networks. pp. 212–221 (2009)
16. Thibault, S., Namyst, R., Wacrenier, P.: Building portable thread schedulers for hierarchical multiprocessors: the bubbleshed framework. In: Euro-Par. ACM, Rennes, France (2007)
17. Wei, G., Vasilakos, A.V., Zheng, Y., Xiong, N.: A game-theoretic method of fair resource allocation for cloud computing services. J. Supercomput. 54(2), 252–269 (2010)