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

Georgios C. Chasparis[1], Vladimir Janjic[2], Michael Rossbory[1], and Kevin Hammond[2]

[1] Software Competence Center Hagenberg GmbH, Softwarepark 21, A-4232 Hagenberg, Austria,
{georgios.chasparis,michael.rossbory}@scch.at
[2] School of Computer Science, University of St Andrews, Scotland, UK,
vj32@st-andrews.ac.uk, kevin@kevinhammond.net

**Abstract.** This paper introduces a resource allocation framework specifically tailored for addressing the problem of dynamic placement (or pinning) of parallelized applications to many-core systems. Under the proposed setup each thread of the parallelized application constitutes an independent decision maker, which autonomously decides on which processing unit to run next. Decisions are updated recursively for each thread by a resource manager which runs in parallel to the application's threads and periodically records their performances and assigns to them new CPU affinities. We extend prior work of the authors by introducing a two-level decision making process that is more appropriate to handle many-core systems under Non-Uniform Memory Access architectures (NUMA). In particular, the first level may handle pinning of threads or memory over the available NUMA nodes, while the second level may handle pinning over the available CPU cores of the selected NUMA nodes. Under such framework, a learning process updates current estimates and decisions separately for each one of the two decision levels. Additionally, a novel performance-based learning dynamics is introduced which is more appropriate to handle measurement noise and rapid variations in the performance of the threads. Experiments are performed in a many-core Linux platform.

## 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 [1], access bandwidth and battery life in wireless communications

* 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.

[8], computing bandwidth under certain QoS requirements [2], computing bandwidth for time-sensitive applications [5], computing bandwidth and memory in parallelized applications [3].

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 [2]), 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) [4], or in the case of allocating virtual processors to time-sensitive applications [5].

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, which under certain conditions also imply shorter completion time. 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 proposed framework extends prior work of the authors [6, 12]. In particular, we propose a two-level decision making process that is more appropriate to handle resource allocation optimization in Non-Uniform Memory Access (NUMA) architectures. At the first (*higher*) level, the RM makes decisions with respect to the NUMA node which a thread should be pinned to and/or its memory should be allocated to. At the second (*lower*) level, the RM makes decisions with respect to the CPU core which each thread should be pinned to. Additionally, we propose a novel learning dynamics motivated by *aspiration learning* that is more appropriate for a) controling the switching frequency between NUMA nodes, and b) adjusting the experimentation probability as a function of the current performance. We finally validate the efficiency of the proposed algorithm to increasing the average processing speed of the application with experiments performed in a Linux platform.

The paper is organized as follows. Section 2 discusses the related work. Section 3 describes the overall problem formulation, objective and contributions of

the paper. Section 4 presents the features of the proposed Dynamic Scheduler as well as its advancement over its earlier versions. Section 5 presents experiments of the proposed resource manager in a many-core Linux platform and comparison tests with the operating system's response. Finally, Section 6 presents concluding remarks and future work.

## 2 Related work

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 [13], the non-cooperative game formulation in the problem of medium access protocols in communications [14] or for allocating resources in cloud computing [16]. 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 for an optimal placement. Unfortunately, the *exhaustive-search* type of optimization that is implemented may prohibit runtime implementation. Reference [4] 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 [15] is used to implement scheduling policies. A similar scheduling policy is also implemented by [11].

Contrary to this line of research, recent work by the authors [6, 12] has proposed a dynamic (learning-based) scheme for optimally allocating threads of a parallelized application into a set of available CPU cores. This approach implements a 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. It requires a minimum information exchange, that is only the performance measurements collected from each running thread. Furthermore, it is flexible enough to accommodate alternative optimization criteria depending on the available performance counters (e.g., average processing speed in [6, 12]). It was shown both analytically and through

experiments under the Linux operating system, that the proposed methodology learns a locally-optimal allocation, which under certain conditions also corresponds to the globally optimal solution.

This line of work was an important step towards a) *understanding the limitations of the* OS *in the presence of disturbances*, and b) *efficiently exploiting performance measurements to guide resource allocation.* However, one potential drawback of the proposed approach in [6, 12] was the fact that no special consideration was taken upon the possible *non-uniform memory access* (NUMA) architectures of the provided CPU cores. The proposed learning dynamics in [6, 12] can be applied when the provided CPU cores belong to different NUMA nodes (see, e.g., [12]). However, during the experimentation phase of the dynamics, switching between CPU cores of different NUMA nodes has equal probability to occur as switching between CPU cores within the same NUMA node. As a result, when multiple NUMA nodes are available, it is highly likely that the convergence rates to the optimal allocation will be much slower as compared to the case of a single NUMA node.

To address this potential inefficiency of the learning dynamics when multiple NUMA nodes are available, we provide a two-level resource allocation framework. The first level addresses allocation over the available NUMA nodes, while the second level addresses allocation over the available CPU cores (of the currently selected NUMA node). Such a hierarchical-based framework allows for better controlling switching between NUMA nodes, thus reducing potential inefficiencies due to memory access patterns of the running threads. Furthermore, the newly proposed framework allows for a) introducing multiple time-scale resource allocation, where allocation at the NUMA-level may take place at a slower pace than allocation at the CPU-level, b) introducing auxiliary memory allocation actuation mechanisms which may support dynamic pinning of threads.

## 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 $\mathcal{I} = \{1, 2, ..., n\}$ resulting from a parallelized application. These threads need to be pinned for processing into a set of available CPU cores $\mathcal{J} = \{1, 2, ..., m\}$ (not necessarily homogeneous).

We denote the *assignment* of a thread $i$ to the set of available CPU cores by $\alpha_i \in \mathcal{A}_i \equiv \mathcal{J}$, 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:

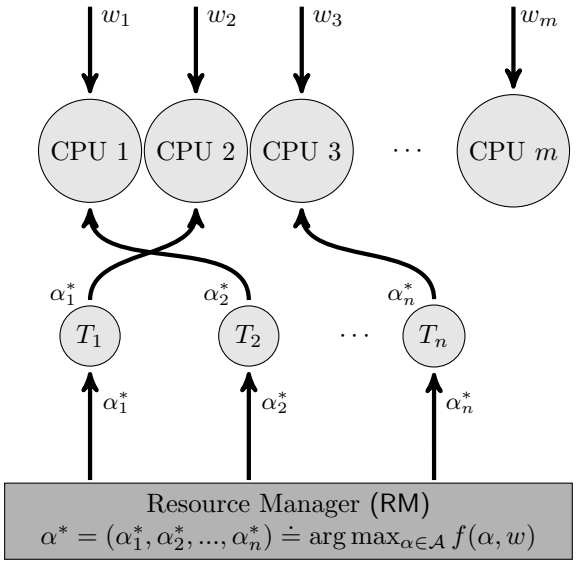



**Fig. 1.** Schematic of *static* resource allocation framework.

(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:

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

In the present work, the centralized objective will correspond to the average processing speed of the running threads, i.e.,

$$f(\alpha, w) \doteq \sum_{i=1}^{n} v_i / n. \tag{2}$$

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 $w = (w_1, ..., w_m)$ 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 $k = 1, 2, ...$ and denoted by $\tilde{f}(k)$. For example, in the case of objective (2), the measured objective takes on the form $\tilde{f}(k) \doteq \sum_{i=1}^{n} \tilde{v}_i(k)/n$. 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:

$$\{(\tilde{v}_i(1), \alpha_i(1)), ..., (\tilde{v}_i(k), \alpha_i(k))\}_i \mapsto \{\alpha_i(k+1)\}_i \tag{3}$$

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.

A dynamic (measurement-based) counterpart of the static framework of Figure 1 is shown in Figure 2. According to such scheme, at any given time instance $k = 1, 2, ...$, 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 Multi-level decision-making and actuation

Recent work by the authors [6, 12] has demonstrated the potential of such dynamic (measurement-based) control of the CPU affinity of the running threads. However, when an application runs on a Non-Uniform Memory Access (NUMA) machine, additional information can be exploited to enhance scheduling of a parallelized application. Consider, for example, the case that the RM periodically makes a decision about the NUMA-CPU affinity pair over which a running thread should run. If the running thread is currently restricted to run on a specific NUMA node, then altering it may result in significant performance degradation (due to, e.g., shared memory with other threads). Although such a

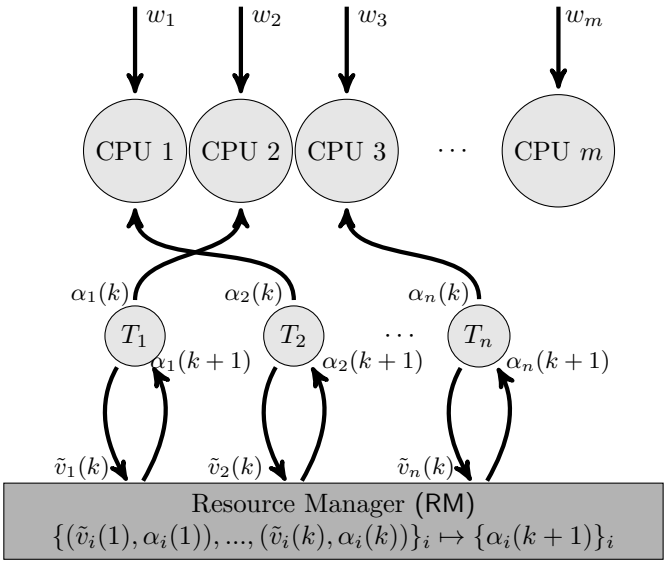



**Fig. 2.** Schematic of *dynamic* resource allocation framework.

decision could be corrected at a later evaluation interval, it would be preferable that NUMA affinities are decided through a different optimization process than the one considered for altering the CPU affinities of a thread.

To this end, a multi-level decision-making and actuation process is considered. The proposed framework builds upon the PaRLSched scheduler presented in [6], which was essentially concerned only with the efficient mapping of a parallelized application within a single NUMA node.

The proposed extension of the PaRLSched dynamic scheduler consists of two nested decision processes depicted in Figure 3. At the *higher level*, 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 (possibly involving memory affinities). At the *lower level*, 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). The details of the scheduler will be described in detail in the forthcoming sections.

The main objective of the updated PaRLSched scheduler is to exploit appropriately the available hardware resources (i.e., processing bandwidth and memory), so that it increases the *performance* of a parallelised application during run-time. Additionally, we should also increase the robustness and resilience of the parallelised application, since a) the application should meet high performance standards relatively to the provided hardware configurations and b) other applications might share the same resources, which may lead to unpredictable variations in the performance of the running applications.

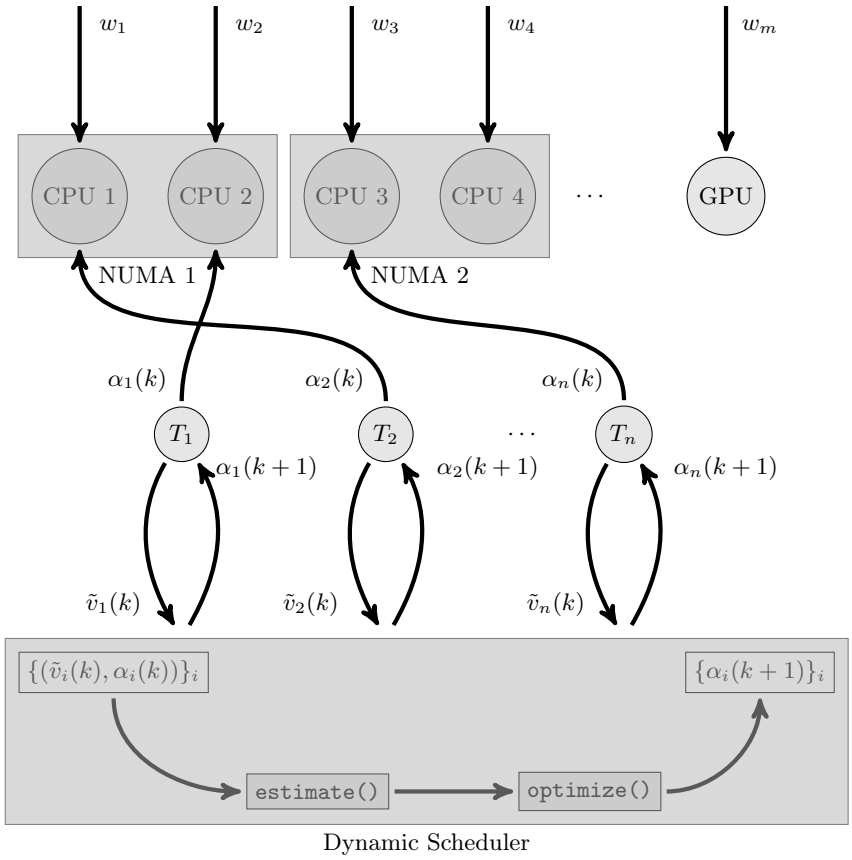



**Fig. 3.** Schematic of a multi-layer *dynamic* resource allocation framework.

### 3.5 Contributions

In prior work of the authors [6, 12], the Dynamic Scheduler addressed the problem of automatically and dynamically discovering the optimal placement of threads (pinning) into a homogeneous hardware platform (in fact, a set of identical CPU processing units). It was based on a distributed Reinforcement-Learning algorithm that learns the optimal pinning of threads into the set of available CPU cores based *solely* on the performance measurements of each thread. The proposed methodology emphasized the fact that the performance of a parallelised application can increase significantly under a) dynamic changes in the availability of resources and b) dynamic changes in the application's demand. These two points seem to be the two major weaknesses of modern operating systems, i.e., the ability to re-adjust under dynamic changes.

We would like to emphasize though that the methodology proposed in [6, 12] could further be improved with respect to the following aspects:

– *Multiple resources.* The proposed approach in [6, 12] was concerned with the optimization of a *single* resource (i.e., the processing bandwidth through the allocation of CPU cores to threads). The question that naturally emerges is the following: *How the proposed methodology can be modified to accommodate multiple and possibly non-uniform resources?* For example, in NUMA architectures we may be concerned of both processing bandwidth as well as memory.
– *Hierarchical structure.* Resources in NUMA architectures may involve hierarchical structures. For example, placement of a thread into a CPU core constitutes a fine-grained allocation of the processing bandwidth. Allocation may instead be performed into NUMA nodes, which can be thought of as a higher-level allocation of processing bandwidth.
– *Non-uniform constraints/requirements.* Multiplicity in the number and nature of optimized resources may additionally impose non-uniform constraints and/or requirements. For example, switching the placement of a thread to a different CPU core may be performed more often compared to, for example, switching the placement of its memory pages among different NUMA nodes. Such differences in the constraints of placing non-uniform resources require special treatment and the algorithms provided should be able to accommodate alternative criteria.
– *Estimation & Optimization.* The approach in [6, 12] provided a unified methodology for concurrent *estimation* and *optimization*. In particular, *estimation* was performed through the reinforcement-learning updates of a strategy/probability vector that summarizes prior experience of a thread over the most beneficial allocation. Moreover, *optimization* was achieved by randomly selecting the destination of a thread according to the corresponding probability vector. However, it might be desirable that estimation and optimization are separated from each other, in order for the designer to be able to incorporate alternative methodologies (either for estimation or for optimization). Furthermore, alternative estimation methods might be available at the same time and the role of the optimizer should be to optimally integrate their predictions.

To this end, in this paper we provide an extension of the Dynamic Scheduler (PaRLSched) developed in [6, 12] in two main directions:

(F1) ***Advancement of architecture.*** We provide a Dynamic Scheduler (PaRLSched) that may easily accommodate more than a single resource at the same time (e.g., both processing bandwidth and memory). However, resources may not necessarily be uniform in nature, optimization criteria and constraints, while they may be organized in a hierarchical structure. To this end, we introduced a rather abstract structure in the dynamic scheduler, which is characterized by the following features:
  (a) *Multiple resources.* The user may define alternative resources to be optimized (i.e., processing bandwidth in the form of thread placement, and memory allocation).

(b) *Hierarchical structure.* The resources may accept *child* resources, a term introduced to establish hierarchical dependencies between the resources. For example, thread placement may be performed with respect to NUMA nodes, however therein a subsequent placement may also be performed with respect to the available CPU cores.
(c) *Distinct optimization criteria.* Each one of the optimized resources and/or their child resources, may accept a distinct method for estimation and optimization, as well as a distinct optimization criterion.

(F2) ***Separating estimation from optimization.*** We advanced our framework for generating strategies for threads by separating the role of *estimation/prediction* from the role of the *optimization*. The reason for this distinction comes from the need to incorporate alternative prediction schemes over optimal allocations without necessarily imposing any constraint in the way these predictions are utilized in the formulation of an optimal strategy.

(F3) ***Advancement of learning dynamics.*** When optimizing memory placement in run-time, we wish to minimize the number of placement switches necessary for approaching an optimal allocation. At the same time, we wish to increase the reaction speed to rapid variations in the performance. To this end, we introduced a novel learning dynamics that is based upon the formulation of benchmark performances/actions. This class of dynamics closely follows the evolution of the performance and triggers the appropriate responses (e.g., experimentation).

## 4 Dynamic Scheduler

Parallelized applications consist of multiple threads that can be controlled independently with respect to their NUMA/CPU affinity (at least in Linux machines). Thus, decisions over the assignment of CPU affinities can be performed independently for each thread, allowing for the introduction of a *distributed learning* framework. This implies that performance measurements can be exploited at the thread-level allowing for the introduction of a "local" learning process, without however excluding the possibility of any information exchange between threads. A schematic of the architecture of the dynamic resource allocation framework is provided in Figure 3.

Below we provide a detailed description of the new features of the updated PaRLSched Dynamic Scheduler.

### 4.1 Advancement of architecture

As briefly discussed in Section 3.5, the main goal of the updated architecture is to provide a straightforward integration of (a) *multiple resources*, (b) *hierarchical structure of resources*, and (c) *alternative optimization criteria*.

According to the new architecture, the user may define the resources to be optimized as well as the corresponding methods used for establishing predictions and for computing optimal allocations. In particular, the initialization of the scheduler accepts the parameters depicted in the following table.

RESOURCES={"NUMA_BANDWIDTH","NUMA_MEMORY"}
OPT_CRITERIA={"PROCESSING_SPEED","PROCESSING_SPEED"}
RESOURCES_EST_METHODS={"RL","RL"}
RESOURCES_OPT_METHODS={"RL","AL"}

In the above example, we have defined two distinct resources to be optimized, namely "`NUMA_BANDWIDTH`", which refers to the placement of threads to specific NUMA nodes, and "`NUMA_MEMORY`", which refers to the placement/binding of the thread's memory pages into specific NUMA nodes. For each one of the resources to be optimized, there might be alternative optimization criteria, which summarize our objectives for guiding the placements. The selection of the optimization criteria is open-ended and directly depends upon the available performance metrics. Currently, we evaluate allocations based on their impact in the *average processing speed* over all running threads of the parallelized application.

In parallel to the selection of the optimized resources, we need to also define the corresponding "methods" for establishing predictions (which are used under the `estimate`() part of the Dynamic Scheduler, Figure 2). For example, we may use the Reinforcement-Learning (RL) algorithm (some alternatives of which were developed in [6, 12]) to formulate predictions based on prior performances. In this case, the outcome of the `estimate`() part of the scheduler will be a probability or strategy vector over the available placement choices that represents a prediction over the most beneficial placement.

Similarly, we need to define the corresponding "methods" for the computation of the next placements (which are used under the `optimize`() part of the Dynamic Scheduler, Figure 2). For example, we may use again the Reinforcement-Learning (RL) perturbed selection criterion (cf., [6]) which is based upon the strategy vectors developed in the `estimate`(). For the case of "`NUMA_MEMORY`", we may use an alternative optimization method, *aspiration learning* (AL), briefly described in the forthcoming Section 4.4 as more appropriate for less frequent decision processing.

### 4.2 Hierarchical structure

Apart from the ability to optimize over more than one resources, it might be the case (as evident in NUMA architectures) that placement of resources can be specialized over several levels. For example, a thread may be bound for processing into one of the available NUMA nodes, however placement can further be specialized over the underlying CPU cores of this node. Thus, the nested hardware architecture naturally imposes a nested description of the resource assignment. That is, the decision $\alpha_1(k)$ of thread $T_1$ in Figure 3 may consist of two levels: in the first level, the NUMA node is selected, while in the second level, the CPU-core of this NUMA node is selected.

The Dynamic Scheduler has been redesigned so that it accepts a nested description of resources. The depth of such description is not limited, although in the current implementation we have been experimenting with a single type of a child resource. An example of how child resources can be defined by the user is depicted in the following table.

```
CHILD_RESOURCES={"CPU_BANDWIDTH","NULL"}
CHILD_OPT_CRITERIA={"PROCESSING_SPEED","PROCESSING_SPEED"}
CHILD_RESOURCES_EST_METHODS={"RL","RL"}
CHILD_RESOURCES_OPT_METHODS={"AL","AL"}
```

We have defined a child resource for the NUMA bandwidth resources, which corresponds to a CPU-based description of the placement. On the other hand, for the NUMA memory resources, we have not defined any child resources. For each one of the child resources, we may define separate estimation and optimization methods. Note that the decisions and algorithms over child resources may not coincide with the corresponding methods applied for the case of the original resources.

### 4.3 Separating estimation from optimization

An additional feature of the updated PaRLSched Dynamic Scheduler is the separation between *estimation/prediction* and *optimization*. These two distinct functionalities of the scheduler are depicted in Figure 3. The reason for this separation in the scheduler was the need for incorporating alternative methodologies for the establishment of both predictions and optimal decisions.

For example, in the learning dynamics (Reinforcement Learning) implemented for optimizing the overall processing speed of a parallelized application in [6, 12], we have implicitly incorporated the functionalities of estimation and optimization within a single learning procedure. Recall that the first part of the Reinforcement Learning methodology is devoted to updating the strategy vectors for each one of the threads (which summarize our predictions over the most beneficial placement), while the second part is devoted to randomly selecting a decision based on a slightly perturbed strategy vector. Under the updated architecture of the scheduler, these two functionalities (estimation and optimization) are separated.

### 4.4 Aspiration-learning-based dynamics

As briefly described in Section 3.5, we wish to provide a class of learning dynamics which provides a better control over two important features: a) speed of response to rapid performance variations, and b) experimentation rate.

The Reinforcement Learning dynamics presented in [6], provide a class of learning dynamics that requires an experimentation phase controlled through a rather small perturbation factor $\lambda > 0$. Essentially, this factor represents a very small probability that a thread will not be placed according to the formulated estimates, rather it will be placed according to a uniform distribution. Such perturbation is essential for establishing a search path towards the current best placement.

However, under this learning dynamics, the times at which the experimentation phase occurs are independent from the current performance of a thread. Thus, situations may occur at which a) experimentation occurs frequently at

allocations at which rather high performance is currently observed (something that would not be desirable), and b) experimentation may be delayed when needed the most (e.g., when performance is dropping). In such cases, it would have been desirable to also exploit the available performance metrics.

To this end, we developed a novel learning scheme that is based upon the notions of benchmark actions/performances and bears similarities with the so-called *aspiration learning*. In words, the basic steps of this learning scheme can be summarized as follows: Let $k$ denote the time index of the optimizer update (it may or may not coincide with the update index of the scheduler).

1. **Performance update.** At time $k$, update the (discounted) running average performance of the thread (with respect to the optimized resource). Let us denote this average performance by $\bar{v}_i(k)$. It is updated according to the the following update rule:

$$\bar{v}_i(k+1) = \bar{v}_i(k) + \epsilon \cdot [v_i(k) - \bar{v}_i(k)], \tag{4}$$

   where $v_i(k)$ is the current measurement of the processing speed of thread $i$.
2. **Benchmark update.** Define the *upper benchmark performance* $\bar{b}_i(k)$ as follows:

$$\bar{b}_i(k) = \begin{cases} \bar{v}_i(k), & \text{if } \bar{v}_i(k) \geq \bar{b}_i(k-1) \\ \bar{b}_i(k-1), & \text{if } \underline{b}_i(k-1) < \bar{v}_i(k) < \bar{b}_i(k-1) \\ \eta \underline{b}_i, & \text{else,} \end{cases} \tag{5}$$

   for some constant $\eta > 1$. The low benchmark performance is updated as follows:

$$\underline{b}_i(k) = \begin{cases} \bar{v}_i(k), & \text{if } \bar{v}_i(k) \leq \underline{b}_i(k-1) \\ \underline{b}_i(k-1), & \text{if } \underline{b}_i(k-1) < \bar{v}_i(k) \leq \bar{b}_i(k-1) \\ (1/\eta)\bar{b}_i, & \text{else.} \end{cases} \tag{6}$$

3. **Action update.** Given the current benchmark and performance, a thread $i$ selects actions according to the following rule:
   (a) if $\bar{v}_i(k) < \underline{b}_i(k)$, i.e., if the current average performance is smaller than the low benchmark performance, then thread $i$ will perform a random switch to an alternative selection according to a uniform distribution.
   (b) if $\underline{b}_i(k) \leq \bar{v}_i(k) < \bar{b}_i(k)$, then each thread $i$ will keep playing the same action with high probability and experiment with any other action with a small probability $\lambda > 0$.
   (c) if $\bar{v}_i(k) \geq \bar{b}_i(k)$, i.e., if the current average performance is larger than the high benchmark performance, then thread $i$ will keep playing the same action.

It is important to note that the above learning scheme will react immediately when a rapid decrease is observed in the performance (thus, we indirectly increase the response time to large performance variations). At the same time,

the small probability of experimentation is necessary under situations of rather constant performance in order to explore a more beneficial allocation. However, we may direct the search of the experimentation towards allocations which we believe will provide a better outcome. This can be done by directly incorporating the outcome of an estimation method directly in step (3a) of the learning scheme. Thus, such learning scheme can easily be incorporated in the updated architecture of Figure 3 and make use of the outcome of any estimation method.

## 5 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 `28×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: 0-13 CPU cores, Node 2: 14-27 CPU cores).

In the following subsections, we consider a parallelized implementation of the so-called Ant Colony Optimization. The proposed PaRLSched Dynamic Scheduler is implemented in scenarios under which the availability of resources may vary with time. We compare the overall performance of the algorithm with that of the OS scheduler, where comparison is performed on the basis of the processing speed and completion time of the application.

### 5.1 Ant Colony Optimization (ACO)

Ant Colony Optimisation (ACO) [7] 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). We are given $n$ jobs. Each job, $i$, is characterised by its processing time, $p_i$ (p in the code below), deadline, $d_i$ (d in the code below), and weight, $w_i$ (w in the code below). The goal is to find the schedule of jobs that minimises the total weighted *tardiness*, defined as $\sum w_i \cdot \max\{0, C_i - d_i\}$ where $C_i$ is the completion time of the job, $i$.

The ACO solution to the SMTWTP problem consists of a number of iterations, where in each iteration each ant independently computes a schedule, and is biased by a *pheromone trail* (t in the code below). The pheromone trail is stronger along previously successful routes and is defined by a matrix $\tau$, where $\tau[i, j]$ is the preference of assigning job $j$ to the $i$th place in the schedule. After all ants having computed their solutions, the best solution is chosen as the "running best"; the pheromone trail is updated accordingly, and the next iteration is started. The main part of the program is given in Algorithm 2.

**ALGORITHM 1:** Metaheuristics of Ant Colony Optimization.

```
for (j=0; j<num_iter; j++) {
  for (i=0; i<num_ants; i++)
    cost[i] = solve (i,p,d,w,t);
  best_t = pick_best(&best_result);
  for (i=0; i<n; i++)
    t[i] = update(i, best_t, best_result);
}
```

**ALGORITHM 2:** Pseudocode of metaheuristics in ACO.

**Data:** this text
**Result:** *best_result*
initialization;
**for** $j = 0$ *to* $j < num\_iter$ **do**
    read current;
    **if** *understand* **then**
        go to next section;
        current section becomes this one;
    **else**
        go back to the beginning of current section;
    **end**
**end**

### 5.2 Parallelization and experimental setup

Parallelization of the ACO metaheuristic can naturally be implemented by assigning a subgroup of ants to each one of the threads. We consider a uniform division of the work-load to each one of the threads (farm pattern). Parallelization is performed using the `pthread.h` (C++ POSIX thread library).

Throughout the execution, and with a fixed period of 0.2 sec, the PaRLSched collects measurements of the total instructions per sec (using the PAPI library [10]) for each one of the threads separately. As described in detail in Section 4, the decision over the pinning of a thread is taken into two levels. *At the first level*, the scheduler decides which NUMA node the thread will be assigned to, following the aspiration-learning-based algorithm presented in Section 4.4. *At the second level*, the scheduler decides which CPU core the thread will be assigned to, within the previously selected NUMA node. This part of the learning dynamics follows the reinforcement-learning rule presented in [6, 12].

The learning process over the NUMA node assignment takes place at a faster pace as compared to the CPU core assignment. Placement of the threads to the available CPU's is achieved through the `sched.h` library (in particular, the `pthread_setaffinity_np` function). In the following, we demonstrate the response of the PaRLSched scheme in comparison to the Operating System's (OS) response (i.e., when placement of the threads is fully controlled by the OS).

In all the forthcoming experiments, the RM is executed by the master thread which is always running in a fixed CPU core (usually the first available CPU core of the first NUMA node).

In Table 1, we provide an overview of the investigated experiments with the ACO case study. As we see, we consider four main sets of experiments (A, B, C, and D), where each set differs in the amount of provided resources and their temporal availability. In the first set (Exp. A.1–A.3), we essentially restrict the scheduler into a single NUMA node (*small availability*). In the second set of experiments (Exp. B.1–B.3) we provide equal number of CPU cores in both NUMA nodes (*medium availability*). In the third set of experiments (Exp. C.1–C.3), we further increase the number of available CPU cores in both NUMA nodes (*large availability*). Finally, in the fourth set of experiments (D), we provide a time-varying availability of resources alternating between the available NUMA nodes.

Our goal is to investigate the performance of the scheduler under different set of available resources, and how the dynamic scheduler adapts to exogenous interferences. To this end, in each one of these sets, we also vary the temporal availability of the provided bandwidth. In particular, under the *non-uniform*

**Table 1.** Brief description of ACO experiments.

| **Exp.** | **Ants** | **Threads** | **# CPU's/NUMA** | **Conditions** |
|---|---|---|---|---|
| A.1 | 5000 | 40 | 8/0, 2/1 | Uniform CPU availability. |
| A.2 | 5000 | 40 | 8/0, 2/1 | Non-uniform CPU availability. |
| A.3 | 5000 | 40 | 8/0, 2/1 | Time-varying CPU availability. |
| B.1 | 5000 | 40 | 8/0, 8/1 | Uniform CPU availability. |
| B.2 | 5000 | 40 | 8/0, 8/1 | Non-uniform CPU availability. |
| B.3 | 5000 | 40 | 8/0, 8/1 | Time-varying CPU availability. |
| C.1 | 5000 | 40 | 12/0, 12/1 | Uniform CPU availability. |
| C.2 | 5000 | 40 | 12/0, 12/1 | Non-uniform CPU availability. |
| C.3 | 5000 | 40 | 12/0, 12/1 | Time-varying CPU availability. |
| D | 5000 | 40 | 6/0, 6/1 | Time-varying CPU availability alternating between NUMA nodes. |

*CPU availability* condition, other applications occupy a constant number of the available CPU cores throughout the whole duration of the experiment. On the other hand, under the *time-varying CPU availability* condition, other applications occupy a non-constant part of the available bandwidth (i.e., exogenous applications start running 1 min after the beginning of the experiment). In both the *non-uniform CPU availability* and the *time-varying CPU availability* case, the exogenous disturbances (other applications) comprise computational tasks often equally distributed among the available CPU cores. In the experiment sets A, B, and C, these exogenous applications occupy the first 6 CPU cores of both NUMA nodes.

Furthermore, in set D, we alternate the exogenous interferences between the two NUMA nodes. Our goal is to investigate the effect of the (stack) memory of threads in the overall performance of the application. In particular, in this experiment, an exogenous application alternates between the first 6 CPU cores of the two available NUMA nodes (with a switching period of 5 min).

### 5.3 Thread Pinning

**Experiment Set A: Small CPU availability** In this experiment set, we would like to test the performance of the dynamic scheduler under conditions of small CPU availability (as compared to the number of running threads). As depicted in Table 1, 8 CPU cores are available from the first NUMA node and only 2 CPU cores are available from the second one. We would like to investigate the completion time of the application under three possible conditions (A.1) uniform CPU availability (i.e., no other application is utilizing the platform), (A.2) non-uniform CPU availability (i.e., other applications constantly occupy some of the available CPU cores constantly over the duration of the experiment), and (A.3) time-varying CPU availability (i.e., other applications start running after the first minute of the experiment).

The statistical analysis of the performance of the OSand the PaRLSched dynamic scheduler are depicted in Table 2. Furthermore, in Figure 4, we have plotted the sample

**Table 2.** Statistical results regarding the completion time (in *sec*) of OS and PaRLSched under Experiment Group B ($\epsilon = 0.3/\bar{v}_i/10^8$, $\lambda = 0.1/\bar{v}_i/10^8$).

| **Run** | **A.1** | | **A.2** | | **A.3** | |
|---|---|---|---|---|---|---|
| # | OS | PaRLSched | OS | PaRLSched | OS | PaRLSched |
| 1 | 1075.21 | 1078.35 | 1730.38 | 1499.02 | 1449.87 | 1398.02 |
| 2 | 1056.01 | 1079.44 | 1760.73 | 1444.15 | 1472.76 | 1401.65 |
| 3 | 1060.62 | 1066.12 | 1753.34 | 1456.40 | 1468.28 | 1399.02 |
| 4 | 1060.18 | 1069.92 | 1745.90 | 1433.08 | 1451.86 | 1409.59 |
| 5 | 1073.21 | 1083.59 | 1771.97 | 1446.96 | 1453.15 | 1401.76 |
| **aver.** | **1065.05** | **1075.48** | **1752.46** | **1455.92** | **1459.18** | **1402.00** |
| **s.d.** | **7.68** | **6.45** | **14.00** | **22.80** | **9.42** | **4.06** |

This set of experiments is rather interesting since they provide an uneven number of CPU cores in each one of the available NUMA nodes. Some remarks are the following:

– *Completion-time under small interference:* The PaRLSched scheduler is able to almost match the performance of the OS when there is no interference. In particular, the completion time of the scheduler is about 1% larger than that of the OS. This small difference should be attributed to the following factors:

1. *Experimentation:* The PaRLSched implements a necessary (non-zero) experimentation probability that affects both the selection of the NUMA node as well as the selection of the CPU core.
2. *Load-balancing:* Another reason for this difference might be the potentially more efficient load-balancing incorporated by the OS. Note that the PaRLSched scheduler optimizes with respect to the speed, and in fact, we may observe in Figure 4 (A1), that the running average speed of the OS scheduler coincides with the corresponding one of the PaRLSched scheduler. Thus, the shorter completion time under OS should only be the outcome of the load-balancing algorithm implemented within the Linux kernel. We will revisit this remark in the forthcoming experiments as well.

– *Optimization criterion:* Recall that the optimization criterion driving allocation of resources under the PaRLSched dynamic scheduler is the average processing speed of each thread. Note that the dynamic scheduler is able to achieve the same or higher average processing speed than the OS. In other words, the PaRLSched is able to meet its design specifications.
However, processing speed is only one factor that contributes to the overall completion time. Apparently, there could be additional factors that may influence the completion time, such as internal application details, as well as the load balancing algorithm of the OS discussed above.
– *Completion time under large interference:* The performance of the PaRLSched is significantly better both in experiments A.2 and A.3. This should be attributed to the fact that the PaRLSched utilizes performance measurements in order to adapt in the performance variations of each thread separately. The speed of response to such variations has also been improved by the updated learning dynamics discussed in Section 4.4.

**Experiment Set B: Medium CPU availability** In this set of experiments, we increase the CPU availability (i.e., we provide a larger number of CPU cores from each one of the available NUMA nodes). The statistical analysis of the performance of the OS and the PaRLSched is provided in Table 3. In Figure 6, we provide sample responses for the different classes of interference introduced in Table 1.

We may point out the following remarks:

– *Completion time under small interference:* The OS outperforms the PaRLSched scheduler when there are no disturbances, i.e., the parallel application is the only application running in the system. Note that this discrepancy between the dynamic scheduler and the OS was smaller in experiment set A, when essentially only the first NUMA node was available. In other words, the larger CPU availability or the smaller degree of interference, increased the performance of the OS with respect to the overall completion time. This discrepancy should be attributed primarily to the load balancing algorithm implemented by the OS (as also discussed in the experiment set A). This observation will become more clear in the forthcoming experiment set C.

**Table 3.** Statistical results regarding the completion time (in *sec*) of OS and PaRLSched under Experiment Group B ($\epsilon = 0.3/\bar{v}_i/10^8$, $\lambda = 0.1/\bar{v}_i/10^8$).

| Run | B.1 | | B.2 | | B.3 | |
|---|---|---|---|---|---|---|
| # | OS | PaRLSched | OS | PaRLSched | OS | PaRLSched |
| 1 | 669.20 | 715.47 | 1114.39 | 1038.96 | 1065.86 | 1012.14 |
| 2 | 671.67 | 698.14 | 1113.25 | 1042.97 | 1066.24 | 1013.26 |
| 3 | 684.35 | 691.66 | 1113.29 | 1031.61 | 1067.14 | 1019.89 |
| 4 | 669.98 | 704.48 | 1117.78 | 1052.01 | 1066.95 | 1015.24 |
| 5 | 670.24 | 686.04 | 1073.09 | 1041.11 | 1064.69 | 1035.02 |
| **aver.** | **673.09** | **699.16** | **1106.36** | **1041.33** | **1066.18** | **1019.11** |
| **s.d.** | **5.69** | **10.24** | **16.71** | **6.59** | **0.88** | **8.39** |

- *Optimization criterion:* As also was the case in experiment set A, the PaRLSched dynamic scheduler achieves a running average speed that is either larger than or equal to the corresponfing processing speed achieved by the OS. This is independent of the interference level, as shown in the sample responses of Figure 5.
- *Completion time under large interference:* Observe that the dynamic adaptivity of the PaRLSched scheduler is able to provide better responses with respect to the completion time in dynamic environments (i.e., non-uniform and non-constant availability of resources), i.e., when the interference is rather high. This should be attributed to the adaptive response of the dynamic scheduler to variations in the processing speed of the threads.

**Experiment Group C: Large CPU availability** In this set of experiments, we increase the CPU availability even further. In particular, we provide almost the full available bandwidth from both NUMA nodes. The statistical analysis of the performance of the OS and the PaRLSched is provided by Table 4, while in Figure 6, we provide sample responses for the different classes of interference introduced in Table 1.

A few interesting observations stem from this set of experiments, especially in comparison with the corresponding performances in sets A and B.

- *Completion time:* We observe that in set C, the benefit (initially observed in A and B) of using the PaRLSched dynamic scheduler under the presence of exogenous applications is lost. In fact, the completion time under the PaRLSched scheduler is always slightly larger than that of the OS. One reason for this change (as compared to the sets A and B) is the fact that the interference is now smaller (as a percentage of the provided resources) than that of experiment sets A or B. In particular, in experiment set C, the interference covers 50% of the available CPU cores (since the exogenous applications uses only the 6 first CPU cores of each NUMA node), while in set A and B, the interference covers 80% and 75% of the available CPU cores,

**Table 4.** Statistical results regarding the completion time (in *sec*) of OS and PaRLSched under Experiment Group C ($\epsilon = 0.3/\bar{v}_i/10^8$, $\lambda = 0.1/\bar{v}_i/10^8$).

| **Run** | **C.1** | | **C.2** | | **C.3** | |
|---|---|---|---|---|---|---|
| # | OS | PaRLSched | OS | PaRLSched | OS | PaRLSched |
| 1 | 452.94 | 500.87 | 657.74 | 665.24 | 660.70 | 685.73 |
| 2 | 451.95 | 487.71 | 657.26 | 675.26 | 660.71 | 678.85 |
| 3 | 465.63 | 510.34 | 679.96 | 706.61 | 656.76 | 665.13 |
| 4 | 452.72 | 490.86 | 692.16 | 714.24 | 664.54 | 669.07 |
| 5 | 456.11 | 491.65 | 611.78 | 682.63 | 654.04 | 681.39 |
| **aver.** | **455.87** | **496.29** | **659.78** | **688.80** | **659.35** | **676.03** |
| **s.d.** | **5.08** | **8.28** | **27.45** | **18.66** | **3.62** | **7.72** |

respectively. Thus, we may conclude that the OS is able to respond better under small interferences, most probably due to its internal load balancing of the running threads among the provided CPU cores. The load balancing algorithm of the OS is not utilized by the PaRLSched mainly due to the fact that, at any given time, each thread is restricted to run only at a single CPU core.

– *Average processing speed:* Note that in all experiments (A, B, and C) the average processing speed under the PaRLSched dynamic scheduler is either larger than or equal to the corresponding processing speed under the OS. Observe, for example, in Figure 6(C2)–(C3), that although the running average processing speed under the PaRLSched is larger than that of the OS, the OS completes the tasks at an earlier time. *This shows that the* PaRLSched *is successful with respect to its initial design specifications, that is to maximize the average processing speed of the overall application.* However, this is not necessarily enough to provide a shorter completion time (at least in situations of small or no interference).

### 5.4 Thread pinning and memory binding

In this last experiment, an additional feature has been added into the PaRLSched scheduler, that is the memory binding of a newly allocated (*stack*) memory to the selected NUMA node of the running thread. The intention here is to constrain any newly allocated memory into the selected NUMA node, which may potentially further increase the running speed of the overall application. We wish to investigate the effect of this additional degree of optimization into the overall completion time of the application.

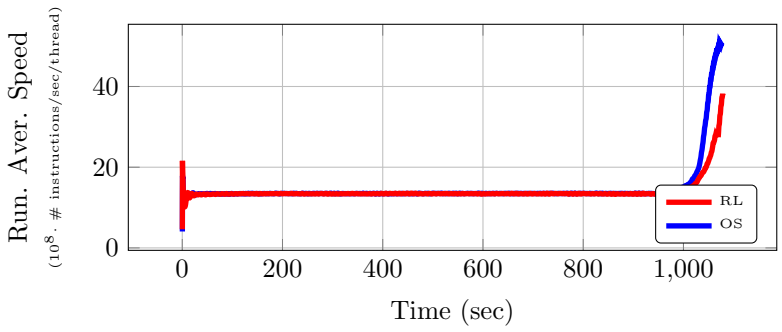
We further introduce the variable $\zeta \in [0, 1]$ that captures the minimum percentage of occupancy (among threads) requested before binding the memory of a thread into a NUMA node. For example, if $\zeta = 1/2$, it implies that a thread's memory will be bound to a NUMA node if and only if more than 1/2 of the threads also run on that node. Intuitively, the more threads occupy a NUMA node, the more likely it is that the larger part of the shared memory will be (or

should be) attached on that node. Thus, essentially, through the introduction of $\zeta$, we get an additional control variable that may potentially affect the speed of the overall application.

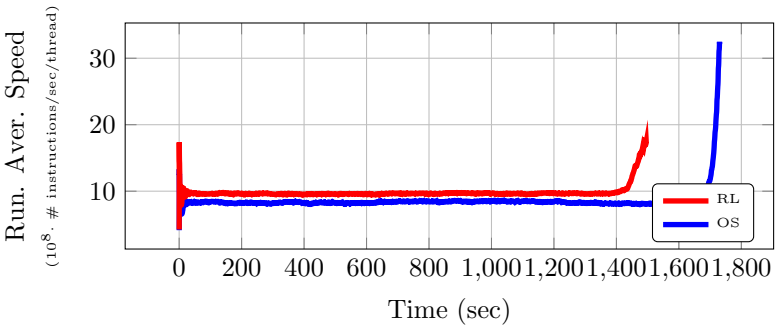
This is indeed the case as depicted in the statistical analysis of Table 5. It is observed that under $\zeta = 1/2$, a small decrease is observed in the completion time of the overall application (which is not observed under $\zeta = 0$).

**Table 5.** Statistical results regarding the completion time (in *sec*) of OS and PaRLSched under Experiment Group D ($\epsilon = 0.3/\bar{v}_i/10^8$, $\lambda = 0.1/\bar{v}_i/10^8$).

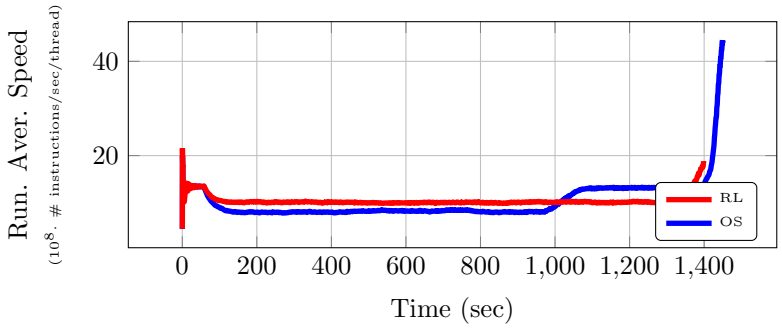
| **Run** # | **D** | | | |
|---|---|---|---|---|
| | OS | PaRLSched | PaRLSched ($\zeta = 1/2$) | PaRLSched ($\zeta = 0$) |
| 1 | 1310.93 | 1231.38 | 1217.91 | 1225.06 |
| 2 | 1322.39 | 1222.69 | 1221.69 | 1227.94 |
| 3 | 1339.97 | 1226.25 | 1220.07 | 1223.37 |
| 4 | 1315.60 | 1224.49 | 1223.50 | 1226.11 |
| 5 | 1332.67 | 1231.12 | 1220.59 | 1230.58 |
| 6 | 1303.44 | 1238.09 | 1231.45 | 1228.79 |
| 7 | 1306.84 | 1224.52 | 1227.91 | 1233.87 |
| 8 | 1322.44 | 1224.56 | 1219.17 | 1226.60 |
| 9 | 1311.75 | 1235.77 | 1225.76 | 1221.96 |
| 10 | 1309.82 | 1219.98 | 1225.94 | 1224.55 |
| **aver.** | **1317.59** | **1227.89** | **1223.40** | **1226.88** |
| **s.d.** | **11.11** | **5.613** | **4.088** | **3.365** |




(A1)

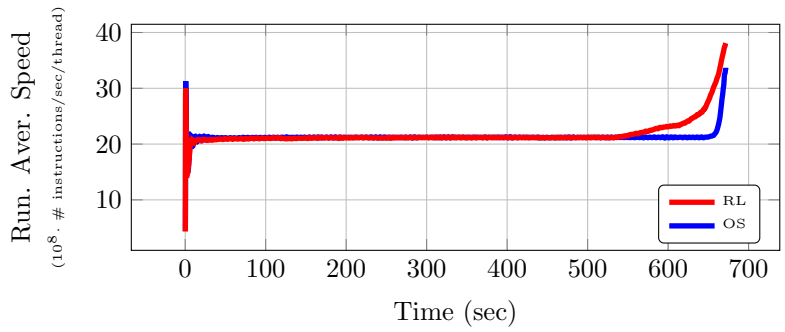



(A2)

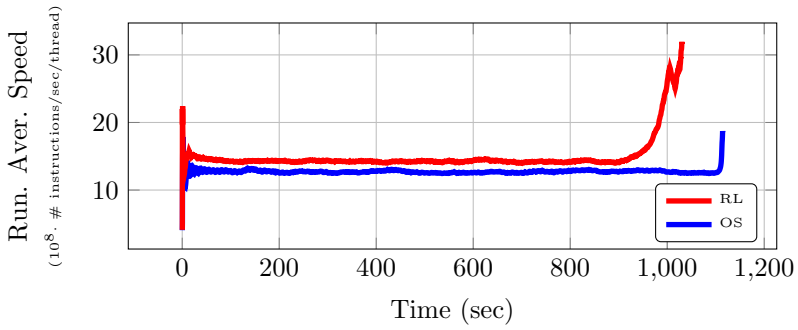



(A3)

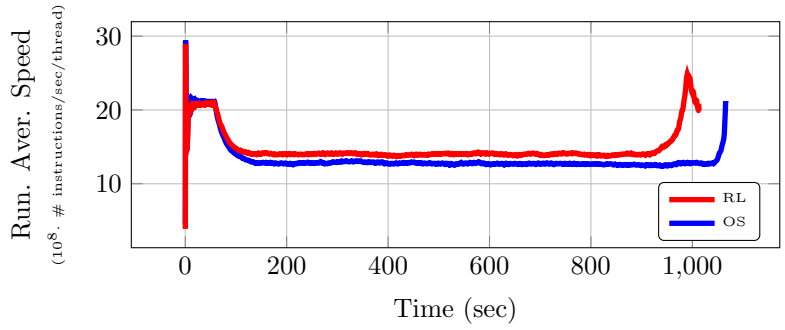
**Fig. 4.** Experiment Group A: Sample Responses




(B1)

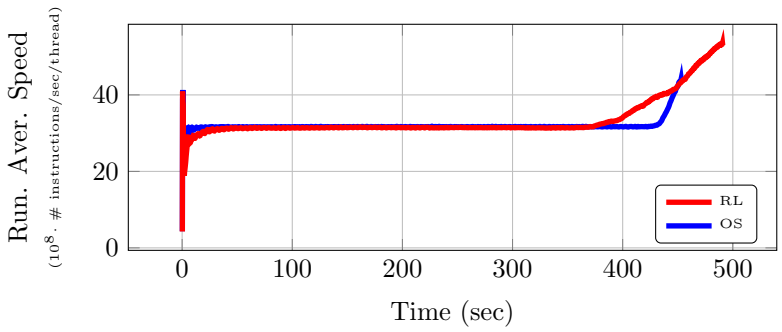



(B2)

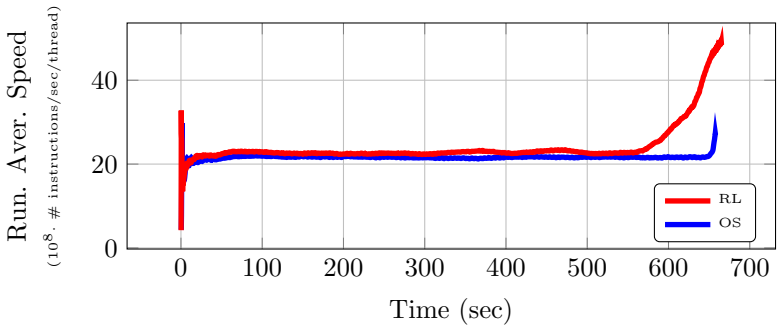



(B3)

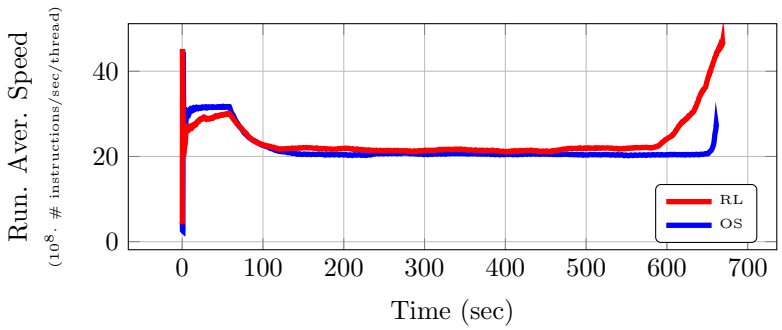
**Fig. 5.** Experiment Group B: Sample Responses




(C1)




(C2)




(C3)

**Fig. 6.** Experiment Group C: Sample Responses

## 6 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 a large set of actuation decisions, including memory placement. Moreover, we introduced a novel learning-based optimization scheme that is more appropriate for administering actuation decisions under a NUMA architecture, since a) it provides better control over the switching frequency and b) it provides better adaptivity to variations in the performance, since the experimentation probability is directly influenced by the current performance.

We demonstrated the utility of the proposed framework in the maximization of the running average processing speed of the threads. Through experiments, we observed that the PaRLSched dynamic scheduler can ensure that the running average speed of the parallelized application will be either larger than or equal to the corresponding speed under the OS's scheduler. However, (as we showed in experiment set C), this may not be sufficient to guarantee shorter completion time of the overall application. This was particularly evident under small degree of interference. One way to resolve this discrepancy in the overall completion time under the PaRLSched scheduler and under small interference is to allow the scheduler to assign more than a single CPU core to each thread. In this way, we will allow the internal load balancing algorithm to also contribute to the better bandwidth management, thus we will manage to better combine the adaptive response of the PaRLSched and the load balancing algorithm of the OS.

# Bibliography


[1] Angelis, F.D., Boaro, M., Fuselli, D., Squartini, S., Piazza, F., Wei, Q.: Optimal home energy management under dynamic electrical and thermal constraints. IEEE Transactions on Industrial Informatics 9(3), 1518–1527 (Aug 2013)

[2] 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)

[3] 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 (Jul 1993)

[4] Broquedis, F., Furmento, N., Goglin, B., Wacrenier, P.A., Namyst, R.: ForestGOMP: An efficient OpenMP environment for NUMA architectures. International Journal Parallel Programming 38, 418–439 (2010)

[5] 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)

[6] Chasparis, G.C., Rossbory, M.: Efficient Dynamic Pinning of Parallelized Applications by Distributed Reinforcement Learning. International Journal of Parallel Programming pp. 1–15 (Nov 2017), https://link.springer.com/article/10.1007/s10766-017-0541-y

[7] Dorigo, M., Stützle, T.: Ant Colony Optimization. Bradford Company, Scituate, MA, USA (2004)

[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 Heidelberg (2011)

[10] 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)

[11] Olivier, S., Porterfield, A., Wheeler, K.: Scheduling task parallelism on multi-socket multicore systems. In: ROSS'11. pp. 49–56. Tuscon, Arizona, USA (2011)

[12] Rivera, F.F., Pena, T.F., Cabaleiro, J.C. (eds.): Efficient Dynamic Pinning of Parallelized Applications by Reinforcement Learning with Applications, Lecture Notes in Computer Science, vol. 10417. Springer International Publishing (2017), http://link.springer.com/10.1007/978-3-319-64203-1, dOI: 10.1007/978-3-319-64203-1

[13] Subrata, R., Zomaya, A.Y., Landfeldt, B.: A cooperative game framework for QoS guided job allocation schemes in grids. IEEE Transactions on Computers 57(10), 1413–1422 (Oct 2008)
[14] 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)
[15] Thibault, S., Namyst, R., Wacrenier, P.: Building portable thread schedulers for hierarchical multi-processors: the BubbleSched Framework. In: Euro-Par. ACM. Rennes, France (2007)
[16] Wei, G., Vasilakos, A.V., Zheng, Y., Xiong, N.: A game-theoretic method of fair resource allocation for cloud computing services. The Journal of Supercomputing 54(2), 252–269 (Nov 2010)