Efficient Dynamic Pinning of Parallelized Applications by Reinforcement Learning with Applications*

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

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. 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 reinforcement-learning algorithm, each branch of which is responsible for assigning a new placement strategy to each thread. The proposed resource allocation framework is flexible enough to address alternative optimization criteria, such as maximum average processing speed and minimum speed variance among threads. We demonstrate the response of the dynamic scheduler under fixed and varying availability of resources (e.g., when other applications running on the same platform) in a parallel implementation of the Ant-Colony Optimization.

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 [10], computing bandwidth under certain QoS requirements [2], computing bandwidth and memory in parallelized applications [4].

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 the *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

* This work has been partially supported by the European Union grant EU H2020-ICT-2014-1 project RePhrase (No. 644235).

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, 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. Prior work has demonstrated the importance of thread-to-core bindings in the overall performance of a parallelized application. For example, [11] describes a tool that checks the performance of each of the available thread-to-core bindings and searches an optimal placement. Reference [5] 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. Hierarchical scheduling is implemented, where low-level *work stealing* [3] is used only at neighboring cores so as to maintain data locality, while at the memory-node level, the thread scheduler deals with larger groups of threads. A similar scheduling policy is also implemented by [14].

This form of scheduling strategies exhibits several disadvantages when dealing with dynamic environments (e.g., varying availability of resources). In particular, retrieving the exact affinity relations during runtime may be an issue due to the involved information complexity. Furthermore, in the presence of other applications running on the same platform, the above methodologies will fail to identify irregular application behavior and react promptly to such irregularities. Instead, in such dynamic environments, it is more appropriate to consider learning-based optimization techniques, where the scheduling policy is being updated based on performance measurements from the running threads. Through such measurement- or learning-based scheme, we can a) *reduce information complexity* (i.e., when dealing with a large number of possible thread/memory bindings) since only performance measurements need to be collected during runtime, and b) *adapt to uncertain/irregular application behavior*.

To this end, this paper proposes a dynamic (learning-based) scheme for optimally allocating threads of a parallelized application into a set of available CPU cores. In particular, the proposed methodology implements a reinforcement learning algorithm (executed in parallel by a resource manager/scheduler), according to which each thread responds to its current performance. The proposed algorithm requires minimum information exchange, that is only the performance measurements collected from each running thread. 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 demonstrate through experiments in a Linux platform that the proposed algorithm outperforms the scheduling strategies of the operating system with respect to completion time.

This work extends prior work of the authors [8] in two directions: (a) we introduce a new type of reinforcement-learning dynamics that admits faster adjustment towards better allocations, and (b) evaluation is performed over a real-world application, that is a parallelized implementation of the Ant-Colony Optimization metaheuristic.

The paper is organized as follows. Section 2 describes the overall framework and objective. Section 3 presents a reinforcement-learning algorithm for dynamic placement of threads. Section 4 presents experiments of the proposed algorithm in a Linux platform and comparison tests with the operating system's performance. Finally, Section 5 presents concluding remarks.

Notation:

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

2 Problem Formulation and Objective

2.1 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/scheduled for processing into a set of available CPU cores, denoted $\mathcal{J} = \{1, 2, ..., m\}$ (not necessarily homogeneous).

We denote the *assignment* of a thread i to the set of available CPU's by $\alpha_i \in \mathcal{A}_i \equiv \mathcal{J}$, i.e., α_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's 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. Throughout 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.

2.2 Static optimization & issues

The selection of a centralized objective is open-ended. In the remainder of the paper, we will consider two main possibilities of a centralized objective in order to emphasize the flexibility of the introduced methodology to address alternative criteria. In the first case, the centralized objective will correspond to maximizing

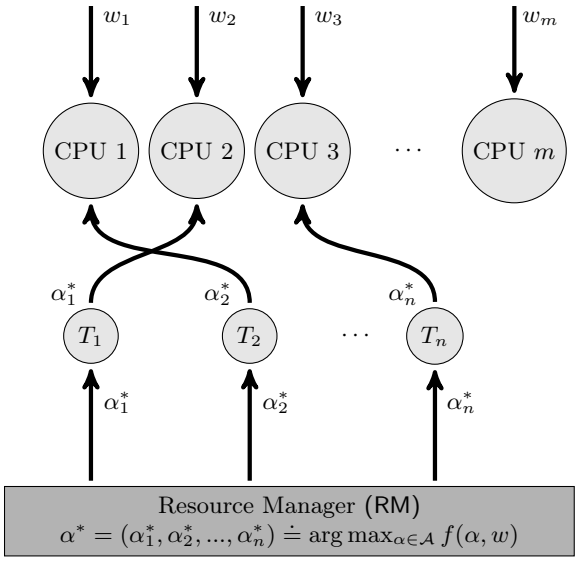

Fig. 1. Schematic of *static* resource allocation framework.

the average processing speed. In the second case, it will correspond to maximizing the average processing speed while maintaining a balance between the processing speeds of the running threads.

Let $v_i = v_i(\alpha, w)$ denote the processing speed of thread i which depends on both the overall assignment α, 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 or other irregularities of the applications. Then, the previously mentioned centralized objectives may take on the following form:

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

where

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

(O2) $f(\alpha, w) \doteq \sum_{i=1}^{n} [v_i - \gamma(v_i - \sum_{j \in \mathcal{I}} v_j/n)^2]/n$, 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) would 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 significant 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 influence w is *unknown* and may vary with time, thus the optimal assignment may not be fixed with time.

2.3 Measurement- or learning-based optimization

We wish to target the *static* objective of (1) through a *measurement-based* (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 $\tilde{f}(k)$. In the case of objective (O1), $\tilde{f}(k) \doteq \sum_{i=1}^{n} \tilde{v}_i(k)/n$. 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 function $f(\alpha, w)$. In other words, the RM employs an update rule of the form:

$$\{(\tilde{v}_i(1), \alpha_i(1)), ..., (\tilde{v}_i(k), \alpha_i(k))\}_i \mapsto \{\alpha_i(k+1)\}_i \tag{2}$$

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.

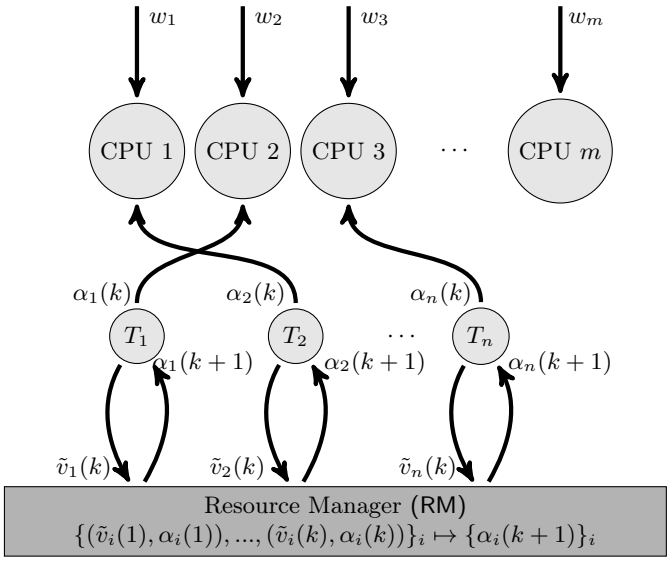

Fig. 2. Schematic of *dynamic* resource allocation framework.

The overall framework is illustrated in Figure 2 describing the flow of information and steps executed. In particular, 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 assignment of each thread i, $\alpha_i(k+1)$, and communicates it to i.

2.4 Objective

The goal of 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. It selects its own CPU assignments independently using its own preference criterion (although the necessary computations for such selection are executed by the RM).

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

In the following section, we will present such a (distributed) learning scheme.

3 Reinforcement Learning (RL)

The question that naturally emerges is *how agents may choose assignments based only on their available measurements so that eventually an efficient assignment is established for all threads.*

To this end, we employ a learning framework (namely, *perturbed learning automata*) that is based on the reinforcement learning algorithm introduced by the authors in [6, 7]. It belongs to the general class of *learning automata* [13].

The basic idea behind reinforcement learning is rather simple. Each agent i keeps track of a strategy vector that holds its estimates over the best choice (in this case, the CPU core). We denote this strategy by $\sigma_i = [\sigma_{ij}]_{j\in\mathcal{J}} \in \Delta(|\mathcal{J}|)$.

To provide an example, consider the case of 3 available CPU cores, i.e., $\mathcal{A}_i = \mathcal{J} = \{1, 2, 3\}$. In this case, the strategy $\sigma_i \in \Delta(3)$ of thread i may take the following form:

$$\sigma_i = \begin{pmatrix} 0.2 \\ 0.5 \\ 0.3 \end{pmatrix},$$

such that 20% corresponds to the probability of assigning itself to CPU core 1, 50% corresponds to the probability of assigning itself to CPU core 2 and 30% corresponds to the probability of assigning itself to CPU core 3. Briefly, the assignment selection will be denoted by

$$\alpha_i = \mathsf{rand}_{\sigma_i}[\mathcal{J}].$$

Note that if σ_i is a unit vector (or a vertex of $\Delta(|\mathcal{J}|)$), say e_j, then agent i selects its jth action with probability one. Such a strategy is usually called *pure strategy.*

3.1 Strategy update

According to the *perturbed reinforcement learning* [6, 7], the strategy of each thread at any time instance $k = 1, 2, ...$ is as follows:

$$\sigma_i(k) = (1 - \lambda)x_i(k) - \frac{\lambda}{|\mathcal{A}_i|} \tag{3}$$

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

$$x_i(k+1) = \begin{cases} x_i(k) + \epsilon \cdot u_i(\alpha(k)) \cdot [e_{\alpha_i(k)} - x_i(k)], & u_i(\alpha(k)) > \bar{u}_i(k) \\ x_i(k), & u_i(\alpha(k)) \leq \bar{u}_i(k), \end{cases} \tag{4}$$

for some constant step-size $\epsilon > 0$. According to this recursion, the new nominal strategy will increase in the direction of the currently selected action $\alpha_i(k)$ and proportionally to the utility received from this selection, $u_i(\alpha(k))$ (which depends on the whole assignment profile). We define the utility of each thread as $u_i(\alpha(k)) = \tilde{f}(k)$, i.e., each thread is assigned a performance index that coincides with the overall objective function (*identical interest*).

In comparison to [6, 7], the difference here 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.

In comparison to [8], the difference lies in the reinforcement direction. As Equation (4) dictates, the strategy vector is only adjusted when a performance is higher than the running-average performance $\bar{u}_i$, which provides a faster adjustment towards better assignments.

The perturbation term λ provides the possibility for the nominal strategy to escape (suboptimal) pure strategy profiles. Setting $\lambda > 0$ is essential for providing an adaptive response of the algorithm to changes in the environment.

The convergence properties of this class of dynamics can be derived following the exact same reasoning used for the learning dynamics presented in [8]. In fact, it can be shown that the dynamics approach asymptotically a set of allocations which includes the solutions of the centralized optimization (1). Such a set may in fact include sub-optimal allocations, however as we shall see in the forthcoming evaluation section, they are significantly better compared to the OS's performance.

3.2 Discussion

The reinforcement-learning algorithm of Equation (4) provides a performance-based optimization. No a-priori knowledge of the type of the application or the underlying hardware is necessary. Furthermore, its memory complexity is minimal, since at any update instance of the RM only the strategy vectors of

each one of the threads needs to be kept in memory, whose size is linear to the number of CPU cores. Furthermore, for each thread, the dynamics exhibit linear complexity to the number of CPU cores, which results in minimal overhead under a reasonable number of CPU cores.

4 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×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's, Node 2: 10-19 CPU's).

In the following subsections, we consider a parallelized implementation of the so-called Ant-Colony Optimization. The proposed reinforcement learning dynamics is implemented in scenarios under which the availability of resources may vary with time. We compare the overall performance of the algorithm with respect to the completion time of the application.

4.1 Ant Colony Optimization (ACO)

Ant Colony Optimisation (ACO) [9] 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 τ, where $\tau[i, j]$ is the preference of assigning job j to the ith 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 Table 1.

4.2 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 RM collects measurements of the total instructions per sec (using the PAPI library [12]) for each one of the threads separately. Given the provided measurements, the update

```
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);
}
```

Table 1. Metaheuristic of Ant-Colony Optimization.

rule of Equation (4) under (O2) is executed by the RM. 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 RL scheme in comparison to the Operating System's (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 experiments, the RM is executed by the master thread which is always running in the CPU with the first index. Furthermore, in all experiments, only the first one of the two NUMA nodes are utilized, since our intention is to demonstrate the potential benefit of an efficient thread placement when the effect of memory placement is rather small.

4.3 Experiment 1: ACO under Uniform CPU Availability

In the first experiment, we consider the ACO parallelized application consisting of 20 threads and utilizing 7 CPU cores. Table 2 shows the completion times under OS and RL for different values of $\gamma > 0$.

Run #	OS	RL ($\gamma = 0$)	RL ($\gamma = 0.02$)	RL ($\gamma = 0.04$)
1	138.39 sec	142.08 sec	142.69 sec	141.69 sec
2	138.57 sec	143.28 sec	141.69 sec	141.27 sec
3	138.80 sec	142.87 sec	142.10 sec	140.92 sec
4	138.38 sec	144.08 sec	143.47 sec	142.71 sec
5	138.78 sec	143.28 sec	142.65 sec	141.28 sec
aver.	**138.58 sec**	**143.12 sec**	**142.52 sec**	**141.57 sec**
s.d.	**0.20 sec**	**0.73 sec**	**0.68 sec**	**0.69 sec**

Table 2. Statistical results regarding the completion time of OS and RL under Experiment 1.

We observe that the dynamic scheduler can match the speed of the OS. The dynamic scheduler (RL) is slower by about 2.12%. This difference can be attributed to the necessary experimentation incorporated into the scheduler ($\lambda >$

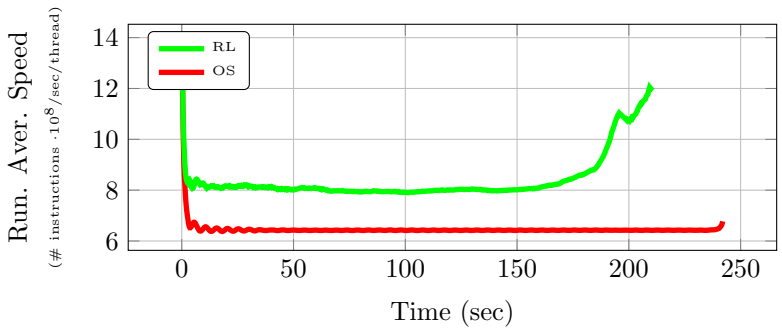

Fig. 3. Running average execution speed for OS and RL ($\gamma = 0.04$) under Experiment 2.

0). Such experimentation is absolutely necessary for the dynamic scheduler to be able to react to variations in the availability of resources.

4.4 Experiment 2: ACO under Non-Uniform CPU Availability

In the second experiment, the bandwidth speed that can be offered by the assigned CPU cores is not uniform. To achieve this variation, we have another (*exogenous*) application running on some of the available CPU cores. In particular, this exogenous application places equal work-load to the first three CPU cores. The exogenous application already runs when the ACO starts running. Figure 3 shows the running average processing speed under OS and RL, which is further supported by the statistical data of Table 3. The RL achieves a significant speed improvement that results in about 12% reduction in completion time.

4.5 Experiment 3: ACO under Time-Varying CPU Availability

This is an identical experiment to Experiment 2, except for the fact that the exogenous application starts running 30 seconds after ACO starts running. This form of test examines the ability of RL to respond after a significant variation in the availability of some of the CPU cores. Figure 4 illustrates the evolution of the running-average processing speed under OS and RL for this experiment.

It is evident in Figure 4 that the RL dynamic scheduler is able to better react to variations in the availability of resources, and achieves a shorter completion time by about 10%. This is also supported by the statistical data of Table 3.

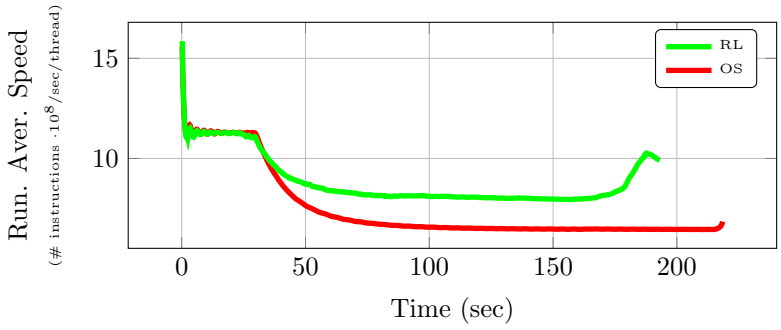

Fig. 4. Running average execution speed for OS and RL ($\gamma = 0.04$) under Experiment 3.

	Experiment 2		Experiment 3	
Run #	OS	RL ($\gamma = 0.04$)	OS	RL ($\gamma = 0.04$)
1	241.30 sec	207.33 sec	218.48 sec	193.30 sec
2	239.10 sec	201.92 sec	218.70 sec	196.45 sec
3	240.90 sec	220.11 sec	218.88 sec	201.92 sec
4	241.11 sec	221.54 sec	219.27 sec	195.88 sec
5	241.51 sec	210.09 sec	218.52 sec	193.41 sec
aver.	**241.06 sec**	**212.20 sec**	**218.77 sec**	**196.19 sec**
s.d.	**0.99 sec**	**8.42 sec**	**0.33 sec**	**3.50 sec**

Table 3. Statistical results of completion time under OS and RL in Experiments 2 and 3, respectively.

5 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. Besides its reduced computational complexity, the proposed scheme is adaptive and robust to possible changes in the environment. We further demonstrated that in the ACO metaheuristic algorithm, the proposed scheduler may reduce the completion time up to 10% under varying resource availability.

Bibliography

[1] Angelis, F.D., Fuselli, D., Squartini, S., Piazza, F., Wei, Q.: Optimal home energy management for residential appliances via stochastic optimization and robust optimization. IEEE Transactions on Smart Grid 3(4) (2012)

[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] Blumofe, R., Leiserson, C.: Scheduling multithreaded computations by work stealing. In: Proc. SFCS'94. pp. 356–368 (1994)

[4] 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)

[5] 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)

[6] Chasparis, G.C., Shamma, J.S., Rantzer, A.: Nonconvergence to saddle boundary points under perturbed reinforcement learning. International Journal of Game Theory 44(3), 667–699 (2015)

[7] Chasparis, G., Shamma, J.: Distributed dynamic reinforcement of efficient outcomes in multiagent coordination and network formation. Dynamic Games and Applications 2(1), 18–50 (2012)

[8] Chasparis, G.C., Rossbory, M.: Efficient Dynamic Pinning of Parallelized Applications by Distributed Reinforcement Learning. arXiv:1606.08156 [cs] (Jun 2016), http://arxiv.org/abs/1606.08156, arXiv: 1606.08156

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

[10] Inaltekin, H., Wicker, S.: A one-shot random access game for wireless networks. In: International Conference on Wireless Networks, Communications and Mobile Computing (2005)

[11] 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)

[12] 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)

[13] Narendra, K., Thathachar, M.: Learning Automata: An introduction. Prentice-Hall (1989)

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