Project no. 644235

REPHRASE

Research & Innovation Action (RIA)
REFACTORING PARALLEL HETEROGENEOUS RESOURCE-AWARE APPLICATIONS – A SOFTWARE ENGINEERING APPROACH

Combined Report On The Final Adaptivity System For Patterned Applications
D4.4

Due date of deliverable: 31.01.2018

Start date of project: April 1st, 2015

Type: Deliverable
WP number: WP4

Responsible institution: University Of St Andrews
Editor and editor's address: Vladimir Janjic, University of St Andrews

Version 0.1

Project co-funded by the European Commission within the Horizon 2020 Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Change Log

Rev.	Date	Who	Site	What
1	20/01/18	Vladimir Janjic	USTAN	Original version.
2	25/01/18	Vladimir Janjic	USTAN	Added section “Mapping Computations and Data To Heterogeneous CPU/GPU Architectures”
3	03/02/18	Georgios Chasparis	SCCH	Added section “Dynamic Scheduling of Patterned Applications”
4	09/02/18	Vladimir Janjic	USTAN	Added sections “Executive Summary”, “Introduction” and “Conclusions”

Executive Summary

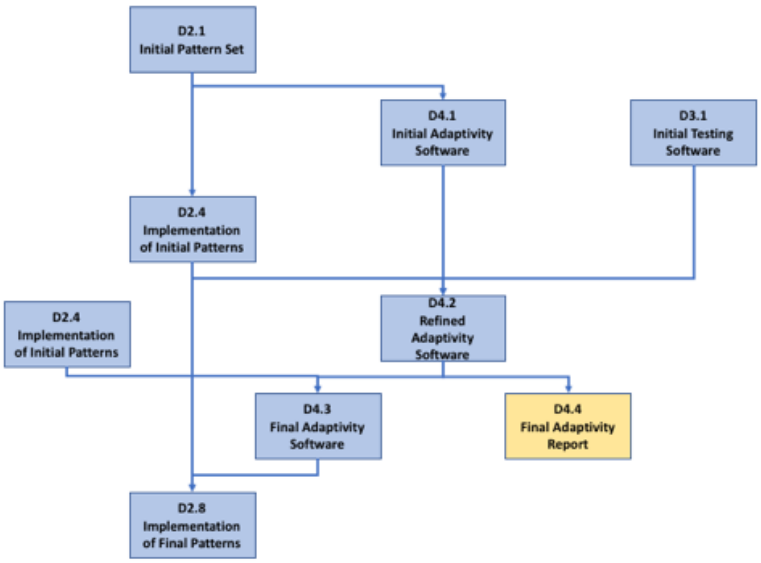


Figure 1: Dependencies between deliverables

This document is the fourth deliverable for WP4 "Dynamic Adaptation of Parallel Software". The purpose of this work package, according to the DoW, is to develop i) techniques for static mapping of components and data of patterned applications to the available hardware; ii) mechanisms for online machine-learning based scheduling for patterned applications; iii) an adaptive, Just-in-Time (JIT) compilation mechanism for patterned applications; and, iv) infrastructure for monitoring performance of patterned applications. The deliverable is the result of the first and second phases of tasks T4.1 ("Static Mapping of Software Components and Data to Hardware Resources"), T4.2 ("Adaptive Compilation of Patterned Applications"), T4.3 ("Dynamic Scheduling of Patterned Applications") and T4.4 ("Performance Monitoring of Patterned Applications"). In this deliverable, we present the final version of the adaptivity toolset, including static mapping, dynamic scheduling and performance monitoring tools. The basic and intermediate versions of the tools were described in D4.1 and D4.2 respectivelly. The final extensions of these tools address heterogeneity in the computing environment, both in terms of the type of available processors (CPUs/GPUs) and in terms of scheduling on non-uniform memory architectures (NUMA). Figure 1 shows the dependencies between the D4.4 and the related **RePhrase** deliverables.

Contents

Chapter 2

Introduction

In the deliverable D4.2 we described an intermediate version of a set of tools for dynamic adaptation of parallel software. These tools take as an input a patterned parallel application with predetermined "shape", as prepared by the tools and libraries developed in WP2 and WP3 and i) decide on an initial instantiation of the application (in terms of number and type of components) and initial distribution of data; ii) dynamically switch between different prepared versions of the same component, as a response to the changes in the application behaviour or system load; iii) allow dynamic rescheduling of application threads to the underlying resources; and, iv) monitor application behaviour and the underlying hardware environment and detect parallelism bottlenecks. The tools described in D4.1 and D4.2 provided the basic and intermediate dynamic adaptation infrastructure for homogeneous and heterogeneous computing systems and generic parallel patterns, thus addressing a crucial aspect of software engineering for parallel systems.

Most of the tools from this workpackage operate at the level of operating system threads and data as mapped to the memory banks. As such, they are independent of the actual high-level parallel used in the parallel application. Therefore, all the tools presented in D4.1 and D4.2 also work with applications that comprise advanced parallel patterns described in D2.8. In this deliverable, we present several extensions to the versions of the tools described in D4.1 and D4.2. In Chapter 3, we describe an extension to FastFlow parallel programming framework to automatically distribute computations in a parallel application to the heterogeneous CPU/GPU hardware, marshalling the data to the GPU devices as necessary and scheduling the execution of GPU kernels. All of this is encapsulated in a parallel pattern, as demonstrated on the example of *stencil-reduce* advanced pattern. We also describe the extension to the dynamic scheduler (Chapter 4) which, compared to the version described in Deliverable D4.2, addresses scheduling on NUMA architectures by providing multi-level scheduling, with different dynamic of making decisions and different learning strategies at the different levels, alowing us to make more "radical" scheduling decisions (moving threads to the processors of different NUMA nodes) less often and in response to specific runtime events. The latest ex-

tensions to the performance monitoring tool are described in more detail in D4.3.

Chapter 3

Mapping of Computations and Data to Heterogeneous CPU/GPU Architectures

3.1 Introduction

The objective of the **RePhrase** task T4.1 was to develop mechanisms for the initial mapping of the tasks and data to the available heterogeneous CPU/GPU systems. The goal of this task was to extend the parallel patterns developed in WP2 with the mapping mechanisms to better adapt to the different deployment platforms. To this end, in D4.1 we have described both the extensions of the basic parallel patterns with explicit data mapping to NUMA architectures, and the heuristics to derive near-optimal instantiation of pattern components onto CPU/GPU systems. These mechanisms considered only static characteristics of an application, i.e. pattern structure of the application (in terms of pattern nesting) and the availability of components (e.g. whether both CPU and GPU versions of a component are available). In D4.2, an extension of these mechanisms was described that also considered some runtime characteristics, such as constraints on the input data and previous history of the executions of the same application to decide, at runtime, about the most appropriate instantiation of heterogeneous pattern components. The drawback of all these mechanisms, however, was that they did not consider automatic mapping the data to the GPU devices. This had to be dealt with by the programmer, and it involved writing a complex and error-prone code for marshalling the data, synchronising the data between CPU and GPU, transferring the data to/from GPU and scheduling the execution of GPU kernels.

In this deliverable, we describe a mechanisms that automates the mapping of data to the GPU devices and execution of GPU kernels. We describe the design of a *stencil-reduce* domain-specific pattern, and we also provide its implementation in the FastFlow pattern-based programming framework. The programmer only needs to provide a problem-specific *kernels* for CPU and GPU, and the pattern imple-

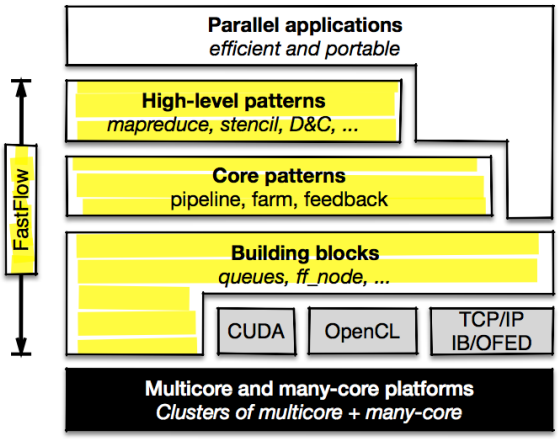


Figure 3.1: Architecture of FastFlow framework.

mentation takes care of memory management and offloading. We demonstrate this on a parallel visual data restoration use case from the image processing domain.

3.2 Stencil-Reduce Pattern

Let **map** $f[a_0, a_1, \dots] = [f(a_0), f(a_1), ...]$ and **reduce** $\oplus\,[a_0, a_1, ...] = a_0 \oplus a_1 \oplus \dots$, where f is the elemental function, $\oplus$ is the combinator (i.e. a binary associative operator), and $[a_0, a_1, ...]$ an array (e.g. the pixels of an image). These parallel paradigms have been proposed as patterns for multicore and distributed platforms, GPGPUs, and heterogeneous platforms [7, 8]. Let **stencil** be a map, with the difference that each instance of the elemental function accesses also *neighbours* of its input (e.g. crossshaped 4-neighborhood of a pixel) [10]. Stencil is a well-known example of a data-parallel pattern; since the elemental function of a *map/stencil* can be applied to each input element ai independently from each other as well as applications of the combinator to different pairs in the reduction tree of a reduce can be done independently, thus naturally inducing a parallel implementation. FastFlow parallel pattern framework [1] (Figure 3.1) provides constructors for these patterns, in form of C++ macros that take as input the code of the elemental function (combinator) of the *map/stencil* (reduce) patterns, together with an eventual read-only structure (i.e. the environment) that can be regarded, under a pure functional semantics perspective, as a function parameter. The language for the kernel code implementing the elemental function, which is actually the business code of the application, has clearly to be platform-specific (i.e. CUDA/OpenCL for GPGPUs, C++/OpenCL for multicore CPUs), since FastFlow is a plain C++ header-only library.

In order to use the proposed pattern, the programmer has to write three macros to

1. define the parameters and the elemental CUDA function that will be applied to each element using the desired stencil shape;

2. that defines the CUDA combinator function that will be applied after the map phase;

3. the stencil-reduce macro that takes as arguments the user-defined datatype that will encapsulate all needed data and macros as described in 1) and 2).

The user can additionally choose the convergence criteria that will decide when the iterative mechanism should stop, based, for example, on the result obtained from the combinator function or by simply defining an upper bound to the number of iteration.

3.3 Example: Two-Phase Image Restoration

Image restoration is a key module of any machine vision system and aims at removing noise (often generated by image sensor failures) and restoring the original visual content. Variational methods are well known for their effectiveness in image restoration [12], but they cannot be employed for real-time video and image analysis because of their high computational cost, due to function minimisation over all the image pixels, and complexity of tuning. An efficient image restoration approach for images corrupted by Salt and Pepper noise based on variational methods is described in [9]. The problem of image restoration for edge preserving is an inverse problem and it has been tackled mainly with variational based approaches **??**. This type of methods identify the restored image by minimising an energy function based on a data-fidelity term, related to the input noise, and a regularisation term where a-priori knowledge about the solution is enforced. More specifically, this kind of approaches identifies the restored image u through an optimisation problem in the form of

$$\min_{u \in N} F(u) = \alpha \int R(u) + \mu \int D(u, d) \quad (1)$$

where d is the image corrupted by the noise, $D(u, d)$ is a data-fidelity, which depends on the kind of noise and provides a measure of the dissimilarity between d and the output image u, $R(u)$ is a regularisation term where a-priori knowledge about the solution is enforced, μ and α are the regularisation parameters that balance the effects of the two terms. This optimisation problem is restricted only to the noisy pixels N.

In detail, a two-phase parallel image restoration schema running on both multicore and GPGPUs is described:

– in the first step, an adaptive median filter is employed for detecting noisy pixels (Detect phase);

– in the second step, a regularisation procedure is iterated until the noisy pixels are replaced with values able to preserve image edges and details (Restore phase).

The restoration method is implemented using the *stencil-reduce* pattern described above and deployed as a sequence of detect and restoration stages that are defined according to the map parallel paradigm and the map-reduce parallel paradigm, respectively.

3.3.1 Detect Phase

The well-known adaptive median filter is first applied to the noisy image with the goal to identify corrupted pixels. Let $\hat{y}$ be the map obtained by thresholding the difference between the output of the adaptive median filter and the noisy original image. This map has ones for candidate noisy pixels and zeros for the remaining pixels. Each pixel of the image, at this stage, is processed independently, provided the processing element can access a read-only halo of the pixel. Hence the set of noisy pixels can be defined as follows

$$N = \{(i, j) \in A : \hat{y}_{i,j} = 1\}$$

The set of all uncorrupted pixels is $N^C = N \setminus A$, where A is the set of the all pixels and N is the set of the noisy pixels.

3.3.2 Restore Phase

The restore phase is carried out by means of regularisation and, among all the variational functions identified by Formula 1, we adopt the following one, as it proved [**?**] to operate effectively for impulse noise:

$$F_{d|N}(u) = \sum_{(i,j)\in N} \left[|u_{i,j} - d_{i,j}| + \frac{\beta}{2}(S_1 + S_2) \right]$$

$$S_1 = \sum_{(m,n)\in V_{i,j}\cap N^C} 2 \cdot \phi(u_{i,j} - d_{m,n})$$

$$S_2 = \sum_{(m,n)\in V_{i,j}\cap N} \phi(u_{i,j} - u_{m,n})$$

where N represents the noisy pixels set and where $V_{i,j}$ is the set of the four closest neighbours of the pixel with coordinates (i, j) of image d and u is the restored image. $F_{d|N}(u)$ is given by the sum of two terms: a data fidelity term which represents the deviation of restored image u from the data image d, which is marred by noise, and a regularisation term that incorporates a function that penalises oscillations and irregularities, although does not remove high level discontinuities.

The method proposed in [14] is used for functional minimisation. The regularisation term is given by the function ϕ and different forms of ϕ have been employed according to the type of noise affecting the image. In this paper we employed:

$$\phi_S(t) = |t|^\alpha, 1 < \alpha \leq 2$$

$$\phi_G(t) = \sqrt{(t^2 + \alpha)}, \alpha << 1$$

for dealing with Salt and Pepper (a specific case of impulse noise) and Gaussian noise, respectively.

The variational method based on minimisation is iteratively applied until a convergence is reached, thus resulting into a fixed-point procedure. Let $u^{(k)}$ be the output of the k-th iteration of the procedure and $\Delta^{(k)} = |u^{(k)} - u^{(k-1)}|$ the pixel-wise residual of two successive iterations of the procedure. We used the following convergence criterion:

$$\frac{\Sigma_{i,j}|\Delta_{i,j}^{(k)} - \Delta_{i,j}^{(k-1)}|}{N}$$

Notice that the computation of the convergence criterion involves three different (successive) outcomes of the procedure, since the naive version of the criterion, based on the comparison of the two most recent outcomes, results in a strongly oscillatory convergence curve.

In this work, we only consider a two-copy approach for the Restore stage, in which the procedure at the (k)-th iteration takes as input the output of the $(k-1)$-th iteration in a read-only manner.

3.3.3 Parallel Implementation

The parallel processing in the Detect phase could in theory be exploited in some data-parallel fashion (e.g. stencil-reduce pattern). However, for images of standard resolutions (up to HD quality), as those considered in the experimental section of this work, the Detect phase has a negligible execution time w.r.t. the Restore stage, so the parallelisation of the Detect phase is not taken into account.

In the Restore phase, the parallel processing is described as an iterative stencil-reduce pattern. Within a single iteration, for each couple of outlier coordinates (i, j), corresponding to an outlier pixel (i.e. a pixel in the noisy set N, which is the outcome of the Detect phase), the stencil accesses the eight neighbour pixels (with replication padding for borders). For each outlier (i, j), the procedure computes a new value as the one minimising the value of the functional $F_{d|N}(u)$ in Formula 1. We remark that in this scenario the user has to provide only the code of the computation over a single pixel – which is actually the business code of the Restore iteration – and the framework takes care of applying the computation over all the (noisy) pixels and iterating the procedure until the convergence condition is reached. In the reduce pattern, the global convergence criterion is computed as a simple average of the three-iteration residuals, as discussed above. In this case the

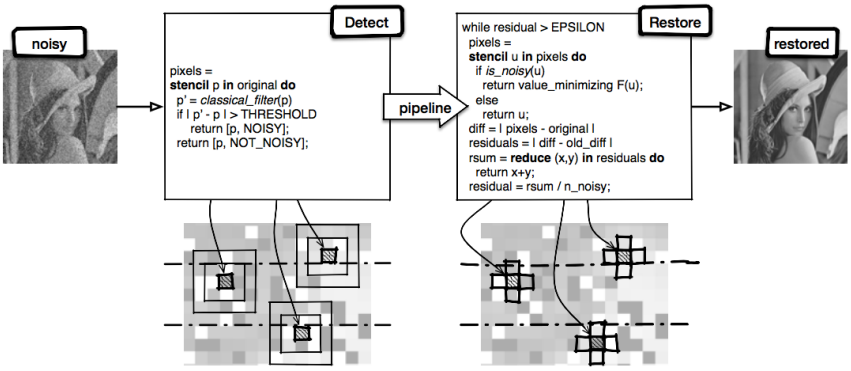


Figure 3.2: Two-phase restoration for visual data streams

user has to provide only the code of the binary combinator, which is turned into a tree-shaped parallel computation by the framework.

From the considerations above, it is clear that the Restore phase is particularly well suited for being implemented via the stencil-reduce pattern since the computation of the functional on each pixel requires accessing to a neighbourhood of the input pixel. Alternatively, the stencil phase could be implemented via other high-level patterns, but some workaround would be required in order to express the convolution-shaped computation. For example, to compute a convolution with a map pattern, the input matrix (frame) should include also the index of each value (pixel) in order to access the neighbours from the code of the elemental function. Finally, the two-phase filter methodology naturally induces a high-level structure for the algorithm, which can be described as the successive application of two filters as described in Figure 3.2. The two phases can operate in a pipeline in the case where the two-phase restoration process is used on a stream of images (e.g. in a video application). In addition, both filters can be parallelised in a data-parallel fashion.

3.3.4 Performance on Images and Video Sequences

This section presents the obtained performance of the new visual data restoration process, implemented using the FastFlow stencil-reduce pattern. The performance has been evaluated on two different environments and compared with the multicore version (also implemented using FastFlow):

1. 16-core-HT + 2 x NVidia-M2090 platform. Intel workstation with 2 eight-core double-context Xeon E5-2660 @2.2GHz, 20MB L3 shared cache, 256K L2, 64 GBytes of main memory and 2 NVidia Tesla M2090 GPGPU, Linux x86 64 and NVidia CUDA SDK 6.0;

2. 4-core-HT + NVidia-K40 platform. Intel workstation with quad-core double-context Xeon E3-1245 v3 @3.4GHz, 8MB L3 shared cache, 256K L2, 16

		Space (4096 × 4096 pixels)			Baboon (2048 × 2048 pixels)		
		10%	50%	90%	10%	50%	90%
4-core-HT+ NVidia-K40 platform	Time (s) – 1CPU	623.60	3084.36	5556.83	154.73	770.50	1385.86
	Time (s) – 1CPU+1GPGPU	4.55	17.03	28.73	1.22	4.33	7.26
	Speedup vs sequential – 1CPU+1GPGPU vs 1CPU	137.05	181.15	193.41	126.73	178.02	190.90
	PSNR	30.30	30.27	19.54	51.62	36.97	22.10
	MAE	0.32	2.34	20.49	0.13	1.63	14.37

Figure 3.3: Performance results for restoring two sample images corrupted with Salt and Pepper noise: Space (4096 x 4096 pixels) and Baboon (2048 x 2048 pixels), using the multicore version (with 1 denoiser thread) and the version built upon the new stencil-reduce pattern. "1CPU" stands for the execution time using only one Restore thread and the other pipeline stages (input/output operations and noise detection) are performed in parallel.

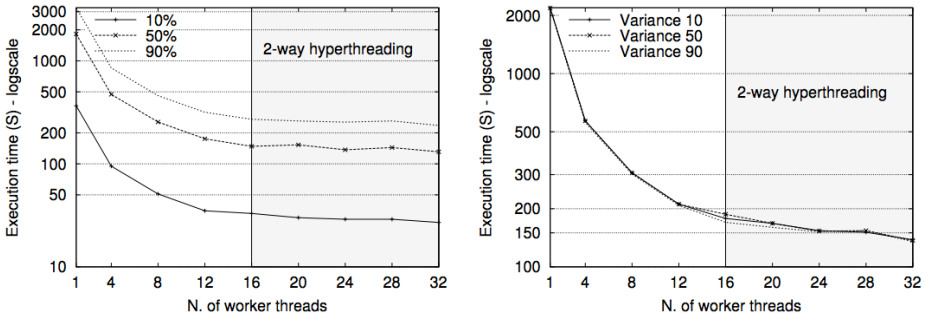


Figure 3.4: Overall denoiser execution time (multicore) of a video containing 91 frames (768 x 512 pixels): on the left 10%, 50% and 90% of Salt and Pepper noise, on the right Gaussian noise with variance 10, 50 and 90 (and mean 0).

GBytes of main memory and a NVidia K40 GPGPU, Linux x86 64 and NVidia CUDA SDK 6.0.

Figure 3.3 presents the obtained execution times when restoring images corrupted with Salt and Pepper noise using the new FastFlow stencil-reduce GPGPU

		Salt and Pepper noise			Gaussian noise (mean 0)		
		10%	50%	90%	Var. 10	Var. 50	Var. 90
16-core-HT+ 2×NVidia-M2090 platform	Time (s) – 16CPUs	27.66	131.87	236.42	138.99	137.21	135.83
	Time (s) – 1CPU + 2GPGPUs	3.94	16.86	30.43	10.33	10.30	10.14s
	Speedup vs multicore – 1CPU+2GPGPUs vs 16CPUs	7.02	7.81	7.76	13.45	13.30	13.39
	Speedup vs sequential – 1CPU+2GPGPUs vs 1CPU	92.94	107.93	107.55	211.98	211.65	212.43
4-core-HT+ NVidia-K40 platform	Time (s) – 4CPUs	62.51	309.71	552.50	237.27	236.44	236.48
	Time (s) – 1CPU+1GPGPU	1.75	5.13	8.78	3.49	3.47	3.42
	Speedup vs multicore – 1CPU+1GPGPU vs 4CPUs	35.65	60.28	62.92	67.90	68.11	69.09
	Speedup vs sequential – 1CPU+1GPGPU vs 1CPU	120.85	204.70	211.87	383.76	382.07	388.22

Figure 3.5: Performance results for restoring a video containing 91 frames (768 x 512 pixels) on two environments, using the multicore version and the version built upon the new stencil-reduce pattern. "1CPU" stands for the execution time using only one thread for Restore stage and the other pipeline stages (input/output operations and noise detection) are performed in parallel.

pattern, and the respective speedup comparing with the single-thread denoiser on CPU. The results present consistent speed improvements with all noise levels, but the GPGPU version is clearly faster when dealing with highly corrupted images. Figure 3.4 presents the multicore (CPU-only) execution times of the video denoiser on 16-core-HT + 2 x NVidia-M2090 platform. On the left (Salt and Pepper noise), the curves show that the amount of noise directly affects the execution time. On the right, however, the curves follow the same pattern, which is explained by the fact that the added Gaussian noise affects the entire image and the detection stage uses a more sensitive threshold and thus the number pixels selected as noisy are not strongly related to the variance in these three cases. The obtained speedup, using 16 physical HT cores of the Salt and Pepper denoiser varies from 13.2x to 13.8x, while the Gaussian denoisee reaches a close-to-linear speedup of 15.8x.

Figure 3.5 presents an overview of the obtained execution times and speedup on the two chosen execution environments. speedup vs multicore (1 CPU + 2 GPGPUs vs 16 CPUs and 1 CPU + 1 GPGPU vs 4 CPUs) represents the performance gain of the GPGPU version over the multicore version using all the available cores, whereas speedup vs sequential (1 CPU + 2 GPGPUs vs 1 CPU and 1 CPU + 1 GPGPU vs 1 CPU) represents the gain over a version that uses only one thread for Restore stage. Since it is much longer compared with the other stages on the pipeline, this version can be considered as an upper bound for the sequential performance. The filter has been also tested with a full length movie: a one-hour video containing 107760 frames of 400 x 304 pixels (30fps), corrupted with 10%,

50% and 90% of impulse noise. This video has been restored using a NVidia K40 GPGPU and the obtained execution times are 1446s, 2622s and 6424s respectively (yielding an average throughput of 75.52fps, 41.08fps and 16.77fps).

Chapter 4

Dynamic Scheduling of Patterned Application

4.1 Dynamic Scheduling in RePhrase

The **RePhrase** dynamic scheduling infrastructure receives as an input a patterned application extended with the static mapping mechanisms described in Chapter 3. The patterned application can be developed either using the refactoring tool described in deliverables D2.2, D2.6 and D2.10, or manually using the parallel patterns described in D2.1 and D2.5. The responsibility of the dynamic scheduler is to schedule the threads and data of the application to the computational cores/memory nodes of the heterogeneous NUMA machine. In this work, we are focusing on allocation CPU cores to the CPU threads of the parallel application, leaving the scheduling of GPU kernels on GPU devices to the operating system. In the deliverable D4.1, we described the initial version of the scheduler that focused on scheduling parallel threads on homogeneous (non-NUMA) architectures. In the deliverable D4.2, we described the extension of the scheduler that targeted the NUMA architectures and included mechanisms for re-mapping the data, as well as threads, to different memory nodes. In this last deliverable of workpackage 4, we describe further extensions to the scheduler, introducing two-level scheduling to better adapt the application schedule to NUMA architectures, and employing different scheduling dynamics and different learning methods at two different levels. We also further extend the data mapping mechanisms, and we evaluate the scheduling infrastructure on a number of both synthetic and realistic use cases, including the use cases from the **RePhrase** project partners.

4.2 Issues in Resource Allocation

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, access bandwidth and battery life in wireless communications,

computing bandwidth under certain QoS requirements, computing bandwidth for time-sensitive applications and computing bandwidth and memory in parallelized applications. In *online* allocation, where the number, arrival and departure times of the tasks are not known a priori, the role of a *resource manager* is to guarantee an *efficient* operation of all applications 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 application. Such information may not be available to either the resource manager or the applications themselves.

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].

4.2.1 RePhrase Approach

In our work, we propose a distributed learning scheme specifically tailored to 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 (see [5, 13] and D4.2). 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 resource manager 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 resource manager 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.

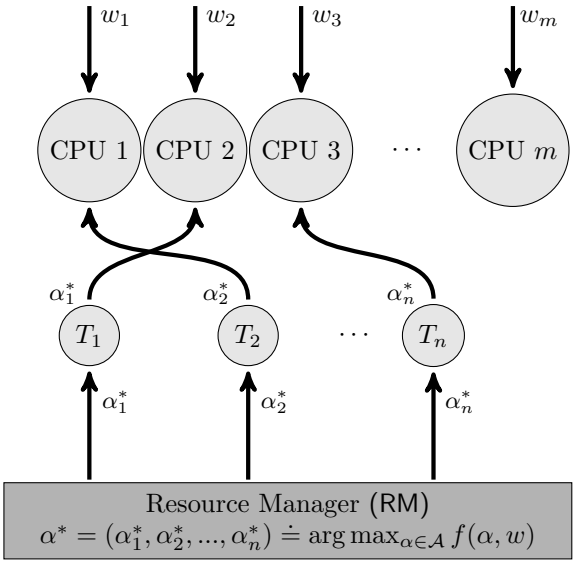


Figure 4.1: Schematic of *static* resource allocation framework.

4.3 RePhrase Scheduling 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., 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 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. We 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.

4.3.1 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 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 (*disturbances*). Then, the previously mentioned centralized objectives may take on the following form:

$$\max_{\alpha \in \mathcal{A}} \quad f(\alpha, w). \tag{4.1}$$

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

$$f(\alpha, w) \doteq \sum_{i=1}^{n} v_i / n. \tag{4.2}$$

Any solution to the optimization problem (4.1) will correspond to an *efficient assignment*. Figure 4.1 presents a schematic of a *static* resource allocation framework sequence of actions where the centralized objective (4.1) 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 (4.1). 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.

4.3.2 Measurement- or learning-based optimization

We wish to address a *static* objective of the form (4.1) 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 (4.2), the measured objective takes on the form $\tilde{f}(k) = \sum_{i=1}^{n} \tilde{v}_i(k)/n$. 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:

$$\{(\tilde{v}_i(1), \alpha_i(1)), ..., (\tilde{v}_i(k), \alpha_i(k))\}_i \mapsto \{\alpha_i(k+1)\}_i \tag{4.3}$$

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.

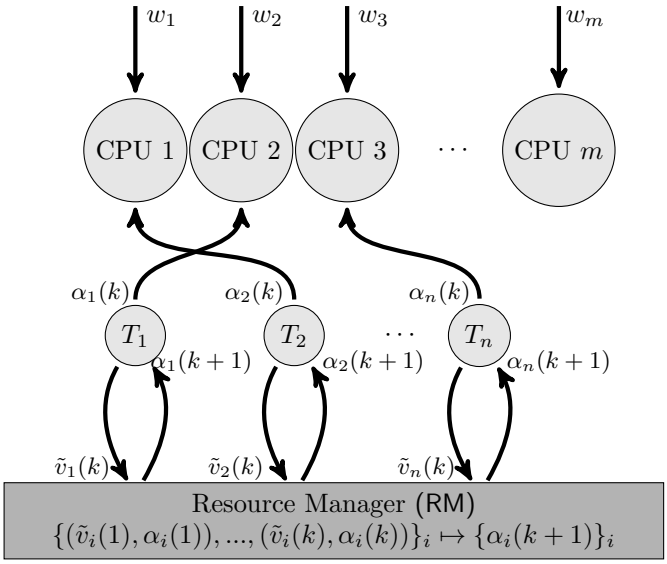


Figure 4.2: Schematic of *dynamic* resource allocation framework.

A dynamic (measurement-based) counterpart of the static framework of Figure 4.1 is shown in Figure 4.2. 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.

4.3.3 Multi-level decision-making and actuation

Recent work by the authors [5, 13] 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 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 RL scheduler presented in [5], which was essentially concerned only with the efficient mapping of a parallelized application within a single NUMA node.

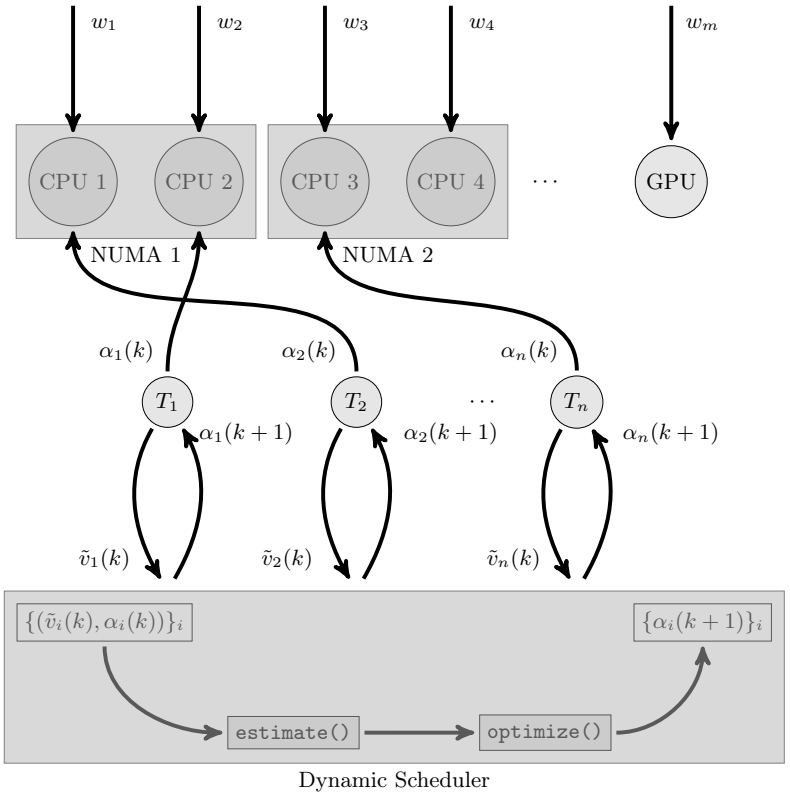


Figure 4.3: Schematic of a multi-layer *dynamic* resource allocation framework.

The proposed extension of the RL dynamic scheduler consists of two nested decision processes depicted in Figure 4.3. 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 RL 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.

4.3.4 Contributions

In prior work of the authors [5, 13] and D4.2, 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 [5, 13] and D4.2 could further be improved with respect to the following aspects:

- *Multiple resources.* The proposed approach 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 [5, 13] and D4.2 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 deliverable we provide an extension of the Dynamic Scheduler (RL) developed in [5, 13] in three main directions:

(F1) ***Advancement of architecture.*** We provide a Dynamic Scheduler (RL) 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).

4.4 Dynamic Scheduler

Parallelized applications consist of multiple threads that can be controlled independently with respect to their NUMA/CPU affinity. 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 excluding the possibility of any information exchange between threads. A schematic of the architecture of the dynamic resource allocation framework is provided in Figure 4.3.

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

4.4.1 Advancement of architecture

As briefly discussed in Section 4.3.4, 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 4.2). For example, we may use the Reinforcement-Learning (RL) algorithm (some alternatives of which were

developed in [5, 13]) 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 4.2). For example, we may use again the Reinforcement-Learning (RL) perturbed selection criterion (cf., [5]) 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 4.4.4 as more appropriate for less frequent decision processing.

4.4.2 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 4.3 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.

4.4.3 Separating estimation from optimization

An additional feature of the updated RL Dynamic Scheduler is the separation between *estimation/prediction* and *optimization*. These two distinct functionalities of the scheduler are depicted in Figure 4.3. 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 [5, 13], 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.

4.4.4 Aspiration-learning-based dynamics

As briefly described in Section 4.3.4, 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 [5], 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:

$$\bar{v}_i(k+1) = \bar{v}_i(k) + \epsilon \cdot [v_i(k) - \bar{v}_i(k)], \tag{4.4}$$

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:

$$\bar{b}_i(k) = \begin{cases} \bar{v}_i(k), & \text{if } \bar{v}_i(k) \geq \bar{b}_i(k-1) \\ \bar{b}_i(k-1), & \text{if } \underline{b}_i(k-1) < \bar{v}_i(k) < \bar{b}_i(k-1) \\ \eta \underline{b}_i, & \text{else,} \end{cases} \tag{4.5}$$

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

$$\underline{b}_i(k) = \begin{cases} \bar{v}_i(k), & \text{if } \bar{v}_i(k) \leq \underline{b}_i(k-1) \\ \underline{b}_i(k-1), & \text{if } \underline{b}_i(k-1) < \bar{v}_i(k) \leq \bar{b}_i(k-1) \\ (1/\eta)\bar{b}_i, & \text{else.} \end{cases} \tag{4.6}$$

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 4.3 and make use of the outcome of any estimation method.

4.5 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 the *Corryvreckan* machine that was bought for the **RePhrase** project. It comprises a `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 RL 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 Linux (OS) scheduler, where comparison is performed on the basis of the processing speed and completion time of the application.

4.5.1 Ant Colony Optimization (ACO)

Ant Colony Optimisation (ACO) [6] 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 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

4.5.2 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 RL collects measurements of the total instructions per sec (using the PAPI library [11]) for each one of the threads separately. As described in detail in Section 4.4, 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 4.4.4. *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 [5, 13].

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 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 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 4.1, 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 CPU availability*

Table 4.1: 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.

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).

4.5.3 Thread Pinning

4.5.3.1 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 4.1, 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 OS and the RL dynamic scheduler are depicted in Table 4.2. Furthermore, in Figure 4.4, we have plotted sample responses for the different classes of interference considered here, namely (A.1), (A.2) and (A.3).

Table 4.2: Statistical results regarding the completion time (in *sec*) of OS and RL 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	RL	OS	RL	OS	RL
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 RL 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 RL 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 RL scheduler optimizes with respect to the speed, and in fact, we may observe in Figure 4.4 (A.1), that the running average speed of the OS scheduler coincides with the corresponding one of the RL 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 RL 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 RL 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 RL is significantly better both in experiments A.2 and A.3. This should be attributed to the fact that the RL 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 4.4.4.

4.5.3.2 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 RL is provided in Table 4.3. In Figure 4.6, we provide sample responses for the different classes of interference introduced in Table 4.1.

We may point out the following remarks:

– *Completion time under small interference:* The OS outperforms the RL 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

Table 4.3: Statistical results regarding the completion time (in *sec*) of OS and RL 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	RL	OS	RL	OS	RL
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**

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.

– *Optimization criterion:* As also was the case in experiment set A, the RL 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 4.5.

– *Completion time under large interference:* Observe that the dynamic adaptivity of the RL 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.

4.5.3.3 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 RL is provided by Table 4.4, while in Figure 4.6, we provide sample responses for the different classes of interference introduced in Table 4.1.

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 RL dynamic scheduler under the presence of exogenous applications is lost. In fact, the completion time under the RL 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

Table 4.4: Statistical results regarding the completion time (in *sec*) of OS and RL 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	RL	OS	RL	OS	RL
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**

(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, 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 RL 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 RL dynamic scheduler is either larger than or equal to the corresponding processing speed under the OS. Observe, for example, in Figure 4.6(C2)–(C3), that although the running average processing speed under the RL is larger than that of the OS, the OS completes the tasks at an earlier time. *This shows that the RL 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).

4.5.4 Thread pinning and memory binding

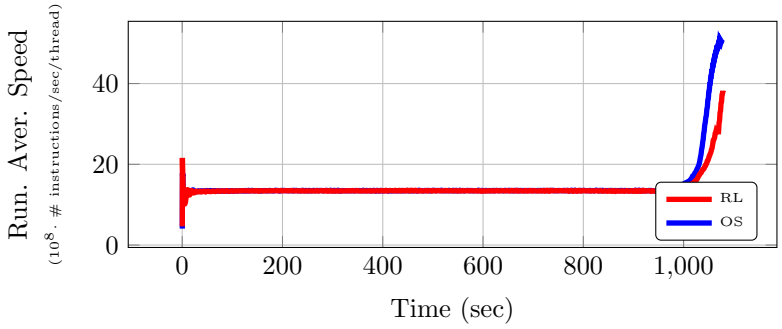
In this last experiment, an additional feature has been added into the RL scheduler, that is the memory binding of a newly allocated (*stack*) memory to the selected NUMA node of the running thread. The intention here is to constrain any newly allocated memory into the selected NUMA node, which may potentially 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 ζ, we get an additional control variable that may potentially affect the speed of the overall application.

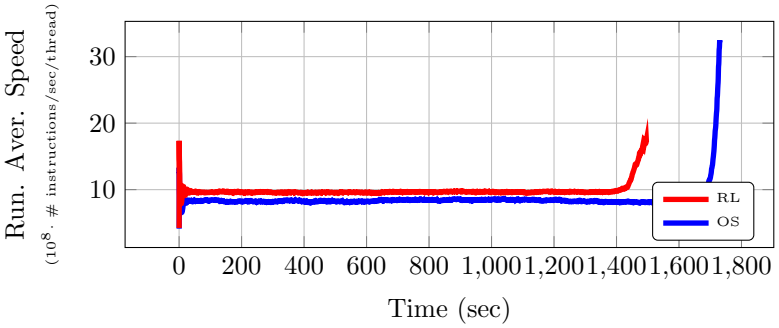
This is indeed the case as depicted in the statistical analysis of Table 4.5. It is observed that under $\zeta = 1/2$, a small decrease is observed in the completion time of the overall application (which is not observed under $\zeta = 0$).

Table 4.5: Statistical results regarding the completion time (in *sec*) of OS and RL under Experiment Group D ($\epsilon = 0.3/\bar{v}_i/10^8$, $\lambda = 0.1/\bar{v}_i/10^8$).

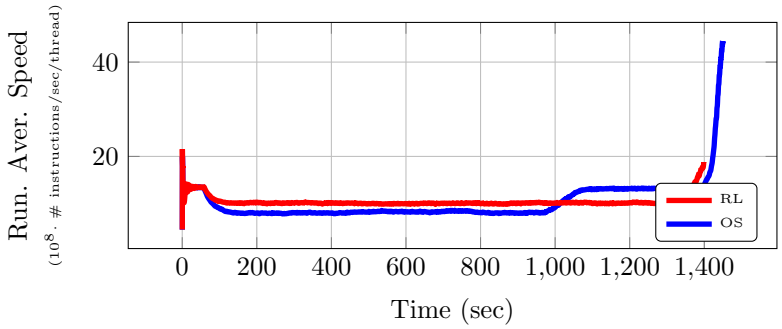
Run #	**D**			
	OS	RL	RL ($\zeta = 1/2$)	RL ($\zeta = 0$)
1	1310.93	1231.38	1217.91	1225.06
2	1322.39	1222.69	1221.69	1227.94
3	1339.97	1226.25	1220.07	1223.37
4	1315.60	1224.49	1223.50	1226.11
5	1332.67	1231.12	1220.59	1230.58
6	1303.44	1238.09	1231.45	1228.79
7	1306.84	1224.52	1227.91	1233.87
8	1322.44	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)

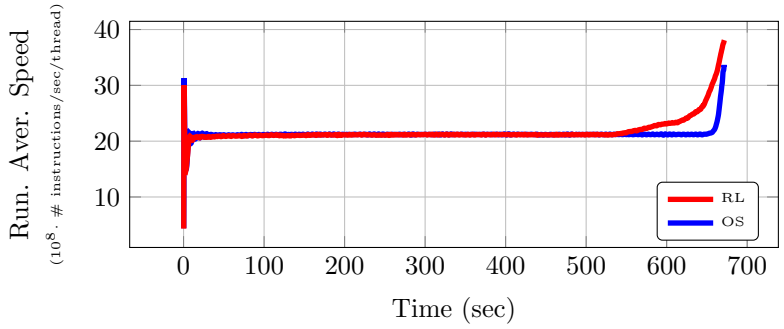


(A2)

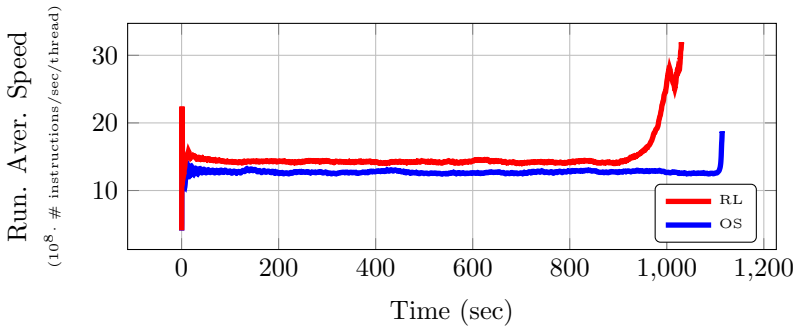


(A3)

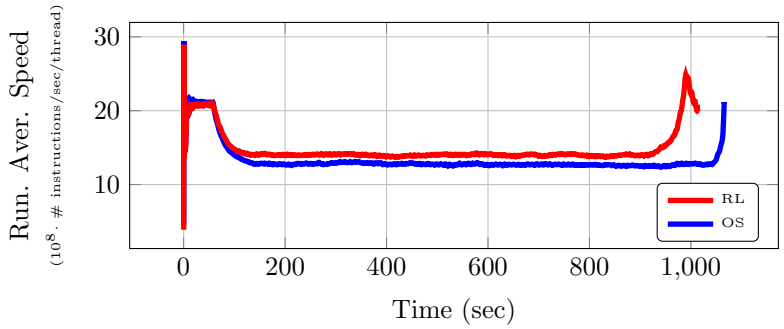
Figure 4.4: Experiment Group A: Sample Responses



(B1)

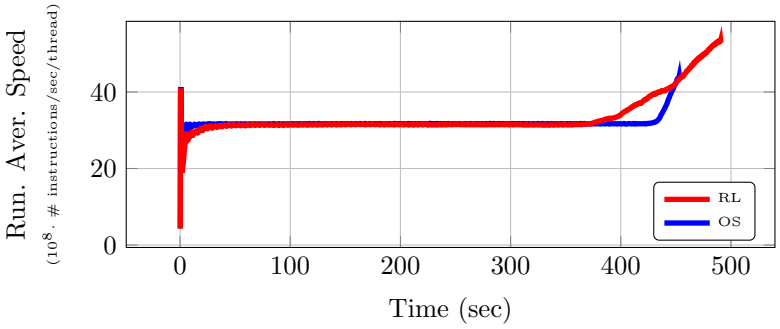


(B2)

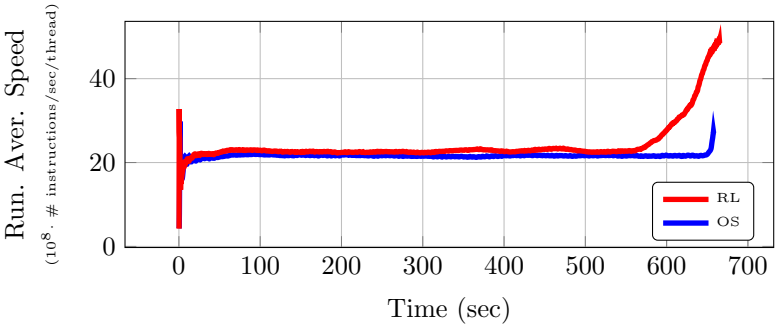


(B3)

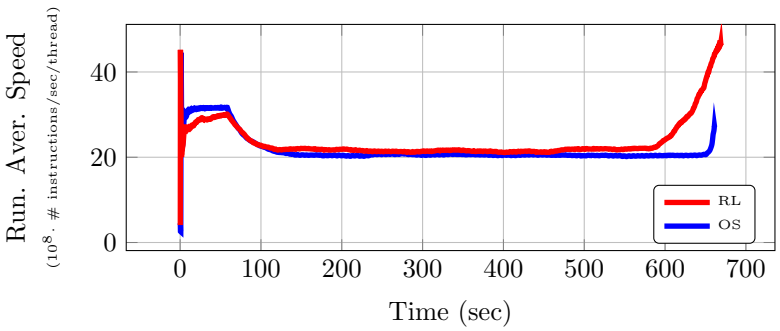
Figure 4.5: Experiment Group B: Sample Responses



(C1)



(C2)



(C3)

Figure 4.6: Experiment Group C: Sample Responses

4.6 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 RL 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 RL 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 RL and the load balancing algorithm of the OS.

Chapter 5

Conclusions

In this deliverable, we have described the final versions of the adaptivity tools from WP4. In D4.3, we have described where to obtain the actual software and how it can be used with the user code. Here, we have focused on the improvements that these versions bring to those previously described in D4.1 and D4.2. In particular, in Chapter 3, we have described the extension of the static mapping mechanisms in the form of an implementation of the *stencil-reduce* parallel pattern which, in addition to the automatic mapping of the computing components to CPU/GPU systems, also does automatic mapping of the application data to the main and GPU memory, providing support for automatic offloading and memory management on GPU devices. We have demonstrated the improvements that using this pattern brings to the problem of image restoration on different CPU/GPU configurations. This demonstrated that we are able to achieve very good speedups in the execution time with relatively little programming effort, as our approach requires the programmer to only write relatively short problem-specific CPU and GPU computing kernels, where everything else is encapsulated in the implementation of the parallel pattern. In Chapter 4, we described the extension of the dynamic scheduler described in D4.1 and D4.2, targetting hierarchical non-uniform memory architectures (NUMA), providing two-level decision making process that adapts mapping of the threads and data to these systems. At the top level, we decide on which group of nodes to execute a particular thread (and, possibly, to which NUMA node to map the appropriate data) and then, at the bottom level, to which particular processing core from the chosen group to pin the thread. We employ different strategies at the two levels, with *aspiration learning* being used at the top level and *reinforcement learning* being used at the bottom level. This also allows different dynamics of making scheduling decisions, with more "radical" decisions (moving threads between different NUMA nodes) being made less often and only as a response to a dramatic fall in the performance. We have presented evaluation of our scheduling methods, demonstrating the improvement of 12% in the execution speed of applications, compared to the state-of-the-art Linux operating system scheduler. These are extremely encouraging results, considering the fact that user only needs

to make very small adjustments to their code to use our scheduler. Finally, in the D4.3 we have described improvements to the performance monitoring infrastructure described in D4.1 and D4.2, which allow us monitoring relevant run-time metrics such as CPU utilisation and energy consumption, allowing also pinpointing of potential problems in the high-level application code.

Collectively, the tools described in D4.1, D4.2, D4.3 and D4.4 provide mechanisms for adaptivity, at deployment time, of the patterned applications to different target architectures. Provided that the application is already parallelised and suitable pattern structure (in terms of possible nesting of patterns), the tools from this workpackage allow the programmer to:

- automatically derive the number of components (e.g. farm workers), as well as the type of each component (e.g. CPU or GPU), of each pattern in a patterened application, based both on the application's pattern structure (D4.1) and the input data together with the previous history of the executions of the same application (D4.2);
- automatically map/replicate the application data to the memory nodes of a NUMA machine (D4.1);
- automatically map the components and application data to the main and GPU memory, as a part of the implementation of a heterogeneous parallel pattern (D4.4);
- support dynamic recompilation of the application code while the application is running, as a response to the changes in the application behavior and/or execution environment (D4.1);
- dynamic remapping of application threads/data to the processing cores/memory nodes of both homogeneous (D4.1 and D4.2) and NUMA (D4.4) architectures;
- provide mechanisms for monitoring the execution of patterned parallel applications, both at the level of parallel patterns and at the level of operating system threads (D4.1 and D4.2).

This is critical for the *deployment* phase of the overall software engineering process, and also has a huge effect on software maintenance, allowing seamless adaptation to the new architectures. The tools that we have developed pushed the state-of-the-art in several directions, outperforming the long established and widely used alternatives both in terms of raw performance (e.g. our dynamic scheduler vs. the Linux operating system scheduler) and ease of development of parallel code (parallel pattern libraries).

Bibliography

[1] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. *Programming Multi-Core and Many-Core Computing Systems*, chapter Fast-Flow: High-Level and Efficient Streaming On Multi-Core. Wiley, May 2014.

[2] Enrico Bini, Giorgio C. Buttazzo, Johan Eker, Stefan Schorr, Raphael Guerra, Gerhard Fohler, Karl-Erik Årzén, Romero Vanessa, and Claudio Scordino. Resource management on multicore systems: The ACTORS approach. *IEEE Micro*, 31(3):72–81, 2011.

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

[4] G. C. Chasparis, M. Maggio, E. Bini, and K.-E. Årzén. Design and implementation of distributed resource management for time-sensitive applications. *Automatica*, 64:44–53, 2016.

[5] Georgios C. Chasparis and Michael Rossbory. Efficient Dynamic Pinning of Parallelized Applications by Distributed Reinforcement Learning. *International Journal of Parallel Programming*, pages 1–15, November 2017.

[6] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. Bradford Company, Scituate, MA, USA, 2004.

[7] Johan Enmyren and Christoph W. Kessler. Skepu: A multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[8] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, November 2010.

[9] M. Drocco M. Torquati M. Aldinucci, C. Spampinato and S. Palazzo. A Parallel Edge Preserving Algorithm For Salt And Pepper Image Denoising. In *Proc. IPTA 2012*, pages 97–102. IEEE, 2012.

[10] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[11] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. PAPI: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[12] Mila Nikolova. A variational approach to remove outliers and impulse noise. *Journal of Mathematical Imaging and Vision*, 20(1):99–120, Jan 2004.

[13] Francisco F. Rivera, TomÃąs F. Pena, and JosÃľ C. Cabaleiro, editors. *Efficient Dynamic Pinning of Parallelized Applications by Reinforcement Learning with Applications*, volume 10417 of *Lecture Notes in Computer Science*. Springer International Publishing, 2017. DOI: 10.1007/978-3-319-64203-1.

[14] C. R. Vogel and M. E. Oman. Fast, Robust Total Variation-Based Reconstruction Of Noisy, Blurred Images. *IEEE Trans. on Image Processing*, 7(6), 1998.