POLITECNICO DI MILANO

DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA

COMPUTER SCIENCE AND ENGINEERING - INGEGNERIA INFORMATICA

MASTER THESIS

A DISTRIBUTED FRAMEWORK SUPPORTING RUNTIME AUTOTUNING FOR HPC APPLICATIONS

M.Sc Thesis of:

Cristiano Di Marco

Student ID:

835911

Advisor:

Prof. Cristina Silvano

Co-Advisors:

Eng. Davide Gadioli

Prof. Gianluca Palermo

Academic Year 2016/2017

Ai miei genitori
Luciana e Romano

Sommario

Applicazioni che si interessano di ricerche complesse come, ad esempio, gli studi sull'universo o sulla microbiologia, stanno proiettando le architetture ad elevate prestazioni di calcolo (*High Performance Computing*) verso il traguardo di un miliardo di miliardi di operazioni effettuate al secondo.

Queste articolate ricerche sono caratterizzate da una moltitudine di dati in ingresso e dalla presenza di parametri che ne influenzano l'esecuzione; poichè problematiche inerenti il consumo di potenza e l'efficienza energetica hanno assunto un'importanza rilevante, esistono varie tecniche che tentano di migliorare questi aspetti, mantenendo comunque accettabile il valore dei risultati ottenuti: ad esempio, strategie di approssimazione (*Approximate Computing*), applicabili sia a livello hardware sia a livello software, ricercano un equilibrio tra la qualità della computazione e lo sforzo richiesto, al fine di adempiere ai vincoli e agli obiettivi dell'applicazione e di massimizzare, al contempo, l'efficienza generale.

Per questo tipo di applicazioni, lo spazio delle possibili configurazioni è molto ampio e, pertanto, non è possibile esplorarlo esausti-

vamente: avere completa conoscenza riguardo le combinazioni dei parametri e i corrispondenti risultati delle metriche prese in esame (come, ad esempio, il numero di operazioni completate al secondo, il consumo di potenza oppure la precisione dei dati in uscita) è irrealizzabile.

Questa tesi ha sviluppato un sistema, *Agorà*, in grado di gestire l'esplorazione dello spazio delle configurazioni attraverso un sottoinsieme di esso, con lo scopo di predire, usando tecniche di apprendimento automatico (*Machine Learning*), il modello completo dei programmi in esame; questo risultato è utilizzato da un cosiddetto *autotuner* per scegliere dinamicamente la migliore combinazione di parametri che soddisfa vincoli e obiettivi correnti dell'applicazione in gestione. I principali benefici apportati da Agorà sono l'eliminazione di ogni fase antecedente l'esecuzione dei programmi e la capacità di gestire molteplici applicazioni contemporaneamente.

Abstract

Compute and data intensive problems, such as universe or microbiological studies, are pushing High Performance Computing architectures to achieve the Exascale level, that is the capability to process a billion billion calculations per second.

These applications manage huge input sets and they can be set up by multiple parameters that influence execution; since power consumption issues and energy efficiency have become essential, there exist various techniques that try to improve these aspects, keeping quality of results however acceptable: for instance, Approximate Computing strategies, both at software or hardware level, balance computation quality and expended effort, in order to achieve application objectives and to maximize overall efficiency.

The Design Space of all possible configurations, for these kind of applications, is very wide and, therefore, it can't be explored exhaustively: having full knowledge about both parameter values and corresponding metric of interest results (such as, for instance, throughput, power consumption or output precision) is almost impracticable.

This thesis has focused on the development of a framework, *Agorà*,

that is able to drive online Design Space Exploration through an initial subset of configurations, with the aim to predict application complete model through Machine Learning techniques; the result is used by an autotuner to dynamically adapt program execution with the best configuration that fulfills current goals and requirements. Main Agorà advantages are the elimination of any offline phase nor design-time knowledge and the capability to manage multiple running applications at the same time.

Table of Contents

CHAPTER 1

Introduction

1.1 Problem and Motivations

Nowadays, increasingly accurate climate forecasts, biomedical researches, business analytics, astrophysics studies or, in general, Big Data related problems require huge computing performance in order to obtain significant results; for these reasons, High Performance Computing technologies have been continuously sped up and, now, their next target is the Exascale level, that is the capability of at least one exaFLOPS, equal to a billion billion FLOPS (FLoating Point Operations Per Second), which is the order of processing power of human brain at neural level, based on H. Markram et al. research study ([1]).

In order to raise computing performances, multicore scaling era

has produced, over time, frequency and power increase; it's impossible to follow this trend anymore, due to the beginning of Dark Silicon era ([2]) and the failure of Dennard Scaling ([3]): in 1974, Robert H. Dennard provided a more granular view of the famous Moore's Law ([4]) about the doubling of number of transistors, in a dense integrated circuit, approximately every two years; this trend produces faster processors because, as transistors get smaller, their power density is constant, so that power use is proportional with area: as transistors get smaller, so necessarily do voltage and current, giving the possibility to increase frequency. The ability to drop voltage and current, in order to let transistors operate reliably, has broken down; static power (power consumed when transistor is not switching) has increased its contribution in overall power consumption, with an importance similar to dynamic power (power consumed when transistor changes its value), producing serious thermal issues and realistic breakdowns. This scenario implicates the impossibility to power-on all components of a circuit at nominal operating voltage due to Thermal Design Power (TDP) constraints.

Due to these physical problems, system energy efficiency has become essential; even if, every approximately 1.5 years computation per kilowatt-hour have doubled over time ([5]), Subramaniam et al. investigation ([6]) suggests that there is the need of an energy efficiency improvement at a higher rate than current trends, in order to reach target consumed power of 20 MW for Exascale systems, established by DARPA report ([7]).

Efficiency has become very important also for electricity consumption of data centers: in fact, just the US data centers are expected to consume 140 billion kWh in 2020, from 61 billion kWh in 2006 and 91 billion kWh in 2013 ([8]), since the amount of information managed by worldwide data centers is going to grow 50-fold while the number of total servers is going to increase only 10-fold ([9]).

It is straightforward, therefore, that green and heterogeneous approaches have to be applied in order to improve general efficiency of High Performance Computing architectures in which, for instance, multiple Central Processing Units (CPUs), General-Purpose computing on Graphics Processing Units (GPGPUs) and Many Integrated Cores accelerators (MICs) coexist and work together in a parallel way; for these reasons, *TOP500 Green Supercomputers* ([10]) demonstrates the large interest in green architectures, ranking and updating world top 500 supercomputers by energy efficiency.

There exist various approaches in order to deal with these problems, from both hardware and software point of view; for instance, Power Consumption Management consists of various techniques such as Dynamic Voltage and Frequency Scaling (DVFS) and Thermal-aware or Thermal-management technologies in order to deal with Dark Silicon issue ([11]); another important concept is Approximate Computing ([12]), that focuses on balancing computation quality and expended effort, both at programming level and in different processing units; this thesis deals with this last technique, exploited at software level, oriented to tunable High Performance Computing applications that follow the so called *Autonomic Computing* research project ([13]). The underlying structure on which this theory is based is the Monitor-Analyze-Plan-Execute feedback loop: these systems have the capability to manage themselves, given high-level objectives and application knowledge in terms of possible configurations, made by parameter values (such as, for instance, number of threads or processes involved, possible various kind of compiler optimization levels, application-specific variable values, etc.) and the corresponding metric of interest values (such as, for instance, power consumption, throughput, output error with respect to a target value, etc.); this information assembles application Design Space.

Since, for these kind of programs, the multitude of parameters

and their corresponding set of values make Design Space huge and since, in general, computation time is not negligible, an exhaustive search is practically unfeasible, so there exist Design Space Exploration techniques that aim to provide approximated Pareto points, namely those configurations that solve better than others a typical multi-objective optimization problem (for instance, keeping throughput above some value while limiting output error and power consumption within a prearranged interval).

Typically, application knowledge is built at design-time; after that, it is passed to a dynamic autotuner that has the ability to choose, from time to time, the best possible set of parameter values (also called *dynamic knobs*) with respect to current goals and requirements that might change during execution.

One of the leading researches in this area is the ANTAREX project ([14]) that aspires to express by LARA ([15, 16]), a Domain Specific Language (DSL), application self-adaptivity and to autotune, at runtime, applications oriented to green and heterogeneous HPC systems, up to the Exascale level.

1.2 Objectives

Main objective of this work is to fulfill application Design Space Exploration at runtime, collecting all information about used parameter values and related performances in terms of associated metrics of interest; through this data and Machine Learning techniques, we aim to predict application complete model, made by the whole possible configuration list; finally, obtained model is used by the dynamic online autotuner, that can finally set up related program execution in the best way, according to current goals and requirements.

Agorà feature is the ability to wisely manage multiple tunable applications that run inside a parallel architecture; we want not only to simultaneously supervise different programs, but also to share De-

sign Space Exploration phase among all nodes that run the same application, hence speeding up this process.

Any kind of prior offline phase and design-time knowledge, that separates applications from their execution in runtime autotuning mode, is therefore avoided; Agorà makes autotuning completely independent from both application type and technical specifications about the machine in which program is executed: Agorà does not have to know this information before it starts and final complete model is suitable regardless autotuner objectives.

1.3 Thesis structure

This thesis is structured as follow: in chapter 2 we focuse on the background of concepts connected to our work, clarifying what is a parallel architecture, the meaning of *Design Space Exploration* and *Design of Experiments*, the concept of *dynamic autotuning* and, finally, we present the MQTT messaging protocol and the Generalized Linear Regression (GLR) approach for application model prediction, used in our framework; chapter 3 gives an overview of the State-of-the-art about principal methodologies related to *Agorà*, in particular on Design Space Exploration and autotuning current research studies; in chapter 4 we address our intent, describing overall context, our suggested proposal, its general description and main strenghts; chapter 5 shows technical implementation of *Agorà* use cases, related workflow and framework behavior; in chapter 6 we collect all experimental results we have done, in order to validate our project and to highlight overall produced benefits; finally, chapter 7 summarizes entire work, obtained results and possible future *Agorà* developments.

CHAPTER 2

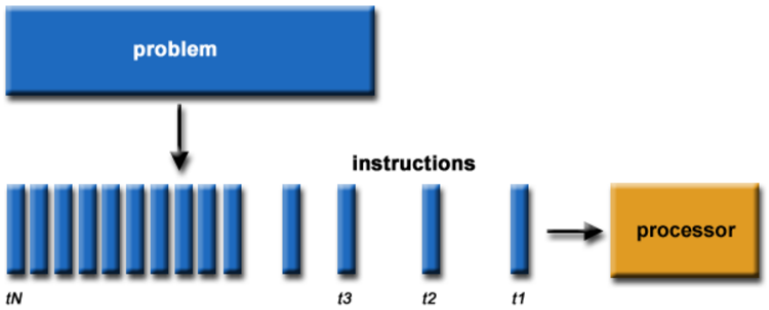
Background

In this chapter we explain main concepts and used tools, related to Agorà; as it has been revealed in previous chapter, this work has been thought for HPC applications that run in parallel architectures, so we start clarifying what are these target systems and how they are built; after that, we explain the concepts of Design Space Exploration and Design of Experiments, that represents the kernel of this work, in union with the idea of application dynamic online autotuning; finally, we present used tools for various communications among Agorà components and for application complete model prediction, highlighting their principal characteristics and features: the Message Queue Telemetry Transport (MQTT) messaging protocol and the Generalized Linear Regression interface by Apache Spark™ Machine Learning MLlib library.

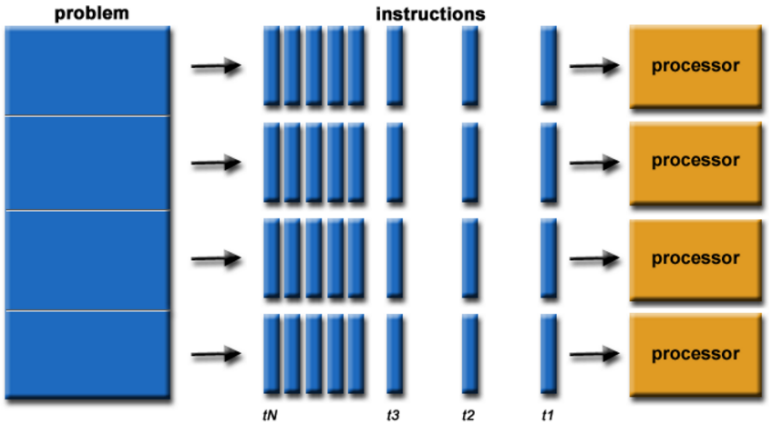
2.1 Target architecture

High Performance Computing applications have complex and massive tasks to do, so they need huge computing power in order to accomplish their objectives.

Parallel computing ([17]) simultaneously uses multiple resources in order to solve a computational problem, taking advantage of concurrency, that is the possibility to execute, at the same time, those parts that are independent each other; in figure 2.1a we can see an example of serial computation, with a single processor that executes problem instructions sequentially, one after another; figure 2.1b instead represents a possible parallel computation, where problem is split in four parts that can be executed concurrently on different processors:



(a) Serial computation



(b) Parallel computation

Figure 2.1: *Serial and parallel computation of a problem*

A typical parallel architecture is composed by an arbitrary number of computers, called *nodes*, each of them with multiple processors, cores, functional units, ect. connected all together by a network:

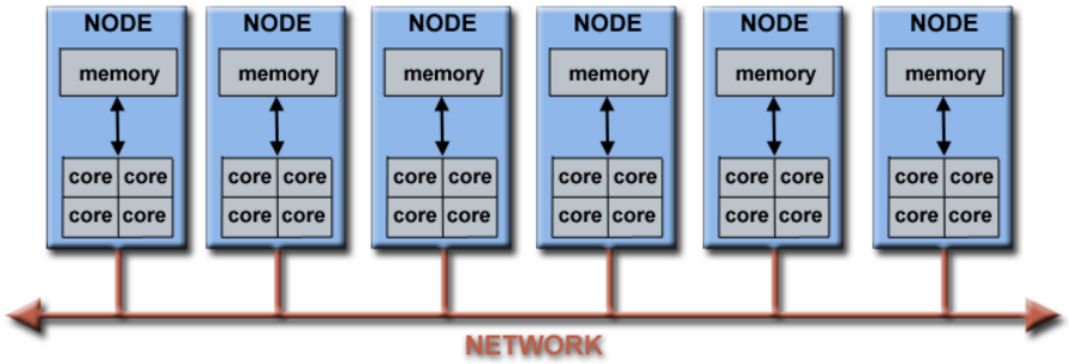


Figure 2.2: *Parallel architecture schema*

Inside a parallel architecture there can be heterogeneous nodes, with different computing techniques and configurations; according to Flynn's classical taxonomy, we can identify four different ways to classify computing architectures:

1. *Single Instruction, Single Data (SISD)*: this is the original kind of computer, in which there is only one instruction stream that is managed by the Central Processing Unit (CPU) during clock cycles; this is serial, so non-parallel, computing. Figure 2.3 shows a SISD execution example, in which every instruction is executed after the previous one:

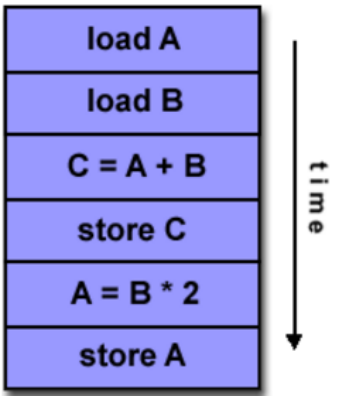


Figure 2.3: *Instruction execution in a SISD architecture example*

2. *Single Instruction, Multiple Data (SIMD)*: at any clock cycle, there is one instruction that is executed, but processing units can operate on different data elements of instruction; in figure 2.4 a classical SIMD execution is shown, where vector *A* and *B* data are used to simultaneously modify vector *C*:

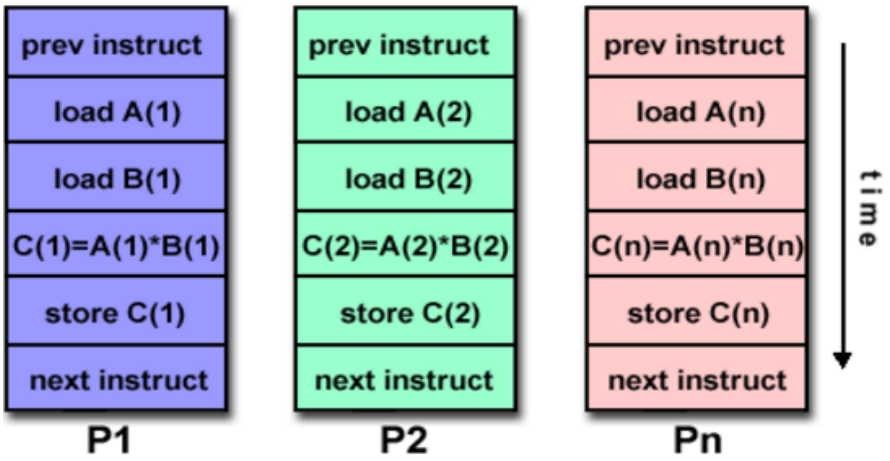


Figure 2.4: *Instruction execution in a SIMD architecture example*

3. *Multiple Instruction, Single Data (MISD)*: a single data stream is managed by multiple processing units, that can independently operate on data through separate instruction streams; figure 2.5 shows a MISD execution, where *A(1)* is simultaneously used to compute different tasks:

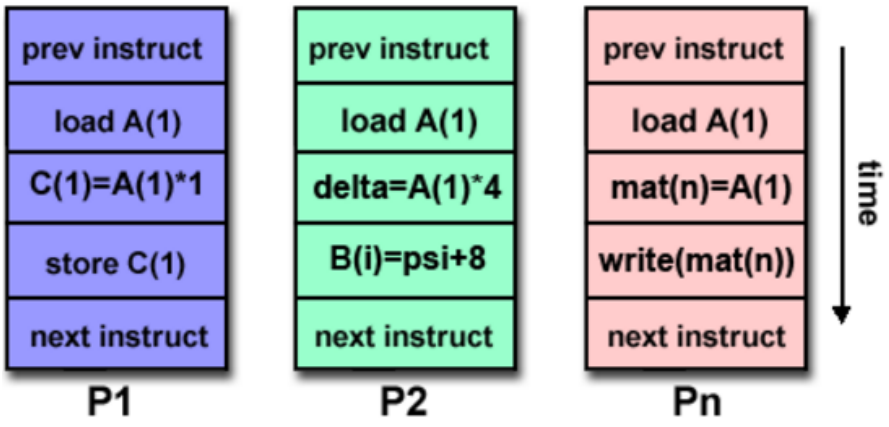


Figure 2.5: *Instruction execution in a MISD architecture example*

4. *Multiple Instruction, Multiple Data (MIMD)*: every processing

unit can execute different data and instruction streams; in figure 2.6, processors concurrently elaborate different tasks that involve different elements:

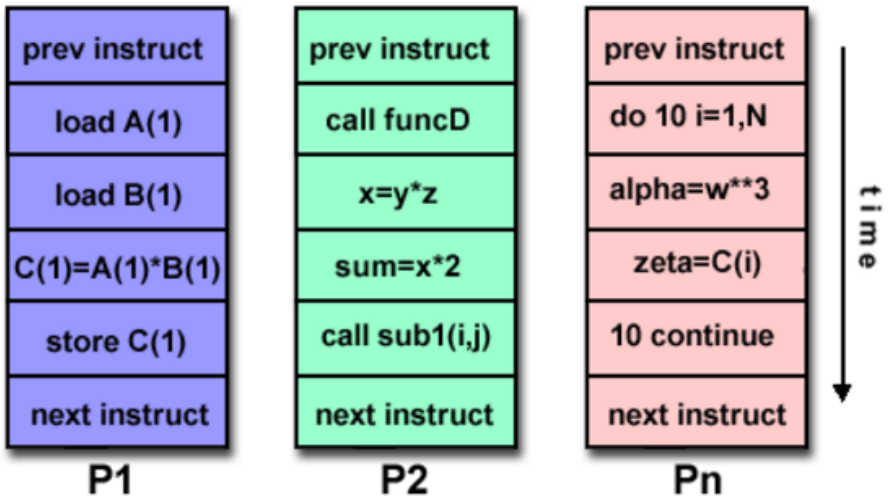


Figure 2.6: *Instruction execution in a MIMD architecture example*

Nowadays, among introduced alternatives, MIMD architectures are the most used in parallel computing, especially in HPC systems.

2.2 Design Space Exploration

In many engineering problems there are several objectives that have to be obtained, with the presence of certain constraints and with the possibility to manage some customizable parameters; objectives can involve various metrics of interest, for instance connected to application throughput, system consumed power or overall cost.

Design Space Exploration (DSE) analyzes, in a systematic way, the space of possible parameter combinations that constitute application Design Space, with the objective to find the best design point that fulfills problem goals and requirements.

HPC applications have almost always a lot of parameters and related set of values, making the list of program configurations very long; in these cases, corresponding Design Space literally explodes, making impossible to analyze it in an exhaustive way.

2.2.1 Multi-Objective Optimization (MOO) problem

When there is more than one objective function that has to be accomplished, DSE consists of a Multi-Objective Optimization (MOO) problem; taking [18] as reference, in mathematical terms a MOO problem can be defined as:

$$min(f_1(x), f_2(x), ..., f_k(x)), \quad k \geq 2 \qquad s.t. \quad x \in X \tag{2.1}$$

where k denotes the number of objective functions and X represents the feasible region, defined by some constraints functions (for instance, in an embedded architecture, the total area should not exceed a predetermined value). If some of the objective functions have to be maximized, they can be attributed to minimizing its negation.

Multi-Objective Optimization problem has a lot of analogies into a wide variety of situations and domains, even the most common ones: formerly the ancients, given a set of seeds and a plot of land, had to choose what, how and how much farm, in order to maximize harvest profit and to minimize required effort, simultaneously not exceeding a cost limit; airline companies want to augment the number of passengers on their airplanes, to increase safety, to enlarge autonomy of their vehicles by means of various technical, strategic and commercial choices; an undergraduate student would minimize his/her university career duration and maximize his/her exam average, with a predetermined time limit and with the possibility to choose some coursers than others.

As it can be understood, since some objectives are in contrast with other objectives, a unique solution does not exist; for instance, in a microcontroller, an objective focused on performance would definitely confront against a goal related to power consumption: best solution for one of them would be the worst for the other and conversely. Therefore, the aim of Design Space Exploration is almost

always to search for a meeting point, between all goals and requirements, that fulfills overall problem.

Since the concept of unique optimal solution can't be applied, it is useful to introduce the notion of Pareto optimality; pareto optimal solutions are, essentially, those ones that can't be improved without degrading at least one objective function. So, a solution x^1 is said to *(Pareto) dominate* a solution x^2 if:

$$\begin{cases} f_i(x^1) \leq f_i(x^2) \quad \forall i \in \{1, 2, ..., k\} \\ f_j(x^1) < f_j(x^2) \quad \textit{for at least one } j \in \{1, 2, ..., k\} \end{cases} \tag{2.2}$$

The set of Pareto optimal solutions is often called Pareto front, Pareto frontier or Pareto boundary.

2.3 Design of Experiments

When a Design Space of an application is huge and, consequently, there is no possibility to do an exhaustive analysis of all possible configurations, there is the need to take a subset of points of interest that represent as closely as possible system behavior. Therefore, on one hand there is the quality of representation, that should be reliable enough; on the other, the number of simulations to do, that should be small.

Taking [19] as reference, among various DoE techniques that generate initial set of design points to be analyzed, we can mention:

- *Full-Factorial DoE*: it is made by all possible combinations among parameter values, so all possible application configurations are picked up;
- *2-Level Full-Factorial DoE*: suitable for designs with two or more parameters, this DoE picks up all possible combinations among the extreme values of all parameters.

If, for instance, there are three tunable parameters:

1. Number of Processors $\in$ { 2, 4, 8, 16, 32 };
2. Number of Threads $\in$ { 1, 2, 3, 4, 5, 6, 7, 8 };
3. Cache size $\in$ { 2K, 4K, 8K, 16K, 32K }.

design points will be: $\langle$ #processors, #threads, cache size $\rangle \in$

{ $\langle$ 2, 1, 2K $\rangle$, $\langle$ 32, 1, 2K $\rangle$, $\langle$ 2, 8, 2K $\rangle$, $\langle$ 32, 8, 2K $\rangle$, $\langle$ 2, 1, 32K $\rangle$, $\langle$ 32, 1, 32K $\rangle$, $\langle$ 2, 8, 32K $\rangle$, $\langle$ 32, 8, 32K $\rangle$ };

- *Face Centered Central Composite DoE with one Center Point*: also this DoE is appropriate for designs with two or more parameters; design point list can be split in three sets:

  1. A 2-Level Full-Factorial Design set;
  2. A Center Point, in which each value is the median value of corresponding parameter;
  3. An Axial Point set, in which all median and extreme values of each parameter are combined.

  Considering the example in previous DoE, final design point list would be: $\langle$ #processors, #threads, cache size $\rangle \in$

  { $\langle$ 2, 1, 2K $\rangle$, $\langle$ 32, 1, 2K $\rangle$, $\langle$ 2, 8, 2K $\rangle$, $\langle$ 32, 8, 2K $\rangle$, $\langle$ 2, 1, 32K $\rangle$, $\langle$ 32, 1, 32K $\rangle$, $\langle$ 2, 8, 32K $\rangle$, $\langle$ 32, 8, 32K $\rangle$ } $\cup$

  { $\langle$ 8, 4, 8K $\rangle$ } $\cup$

  { $\langle$ 2, 4, 8K $\rangle$, $\langle$ 32, 4, 8K $\rangle$, $\langle$ 8, 1, 8K $\rangle$, $\langle$ 8, 8, 8K $\rangle$, $\langle$ 8, 4, 2K $\rangle$, $\langle$ 8, 4, 32K $\rangle$ };

- *Plackett-Burman DoE*: it might be useful to analyze, more economically, a larger number of parameters; in fact, this DoE reduces the number of potential factors, constructing very economical designs with number of points multiple of 4 (rather than power of 2, as in the 2-Level Full-Factorial DoE).

Concerning the example above, in this case final design point list would be: ⟨ #processors, #threads, cache size ⟩ ∈

{ ⟨ 2, 1, 32K ⟩, ⟨ 32, 1, 2K ⟩, ⟨ 2, 8, 2K ⟩, ⟨ 32, 8, 32K ⟩ };

- *Latin-Hypercube DoE*: this DoE randomly chooses parameter values for each design point; the number of final configurations can be set up in advance.

2.4 Dynamic autotuning

When applications expose some configurable parameters (a.k.a. *dynamic knobs*), the concept of dynamic autotuning is defined as the ability to find the best set of knob values, in an automatic and systematic way, that satisfies application goals and requirements at runtime, properly reacting to possible objective function change. For instance, a web video streaming application would be able to manage video quality according to the overload of its servers: in some situations, it could set up itself for the best possible resolution; sometimes, it should reduce quality in order to make still available its services to all connected clients.

IBM research studies on *Autonomic Computing* ([13], [20]) made a breakthrough on the concept of self-adapting systems that are able to manage themselves and to dynamically optimize their execution configuration at runtime; *Autonomic Computing* is based of the MAPE-K control loop, showed in figure 2.7:

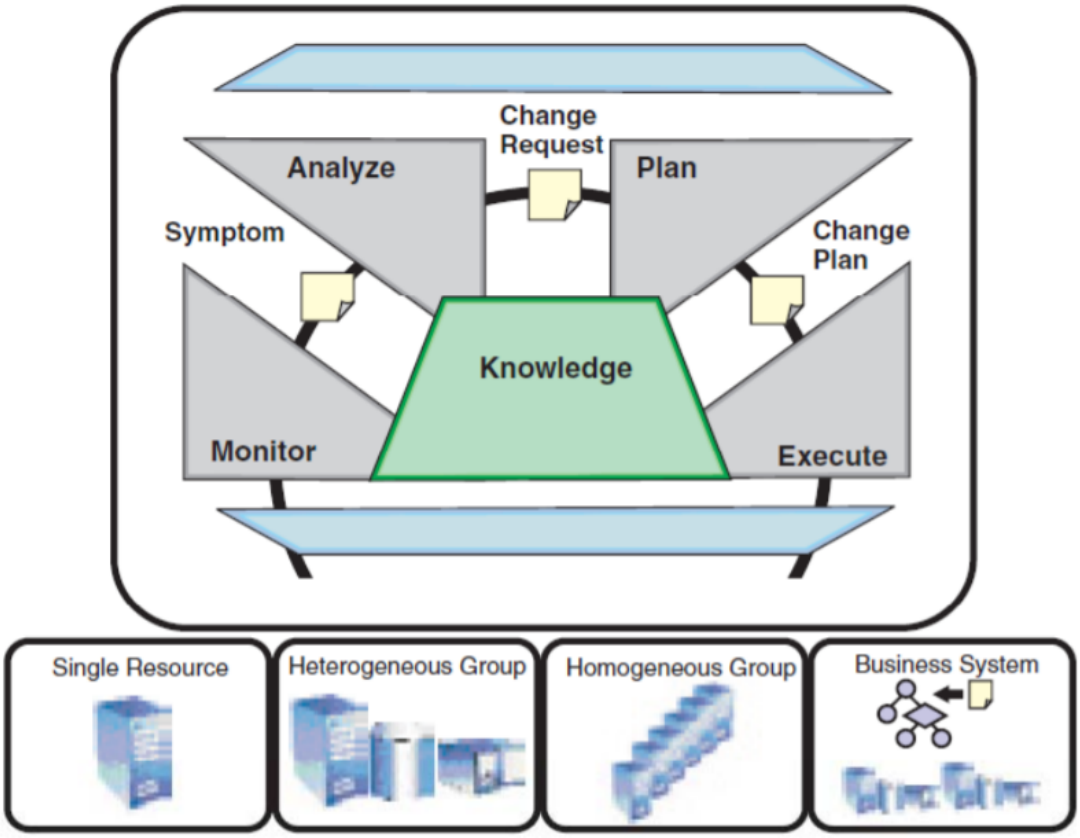


Figure 2.7: *MAPE-K control loop design*

In this control loop we can identify four principal functions:

- *Monitor*: it gathers application information about knob setting and associated metric of interest values as, for instance, throughput and consumed power;
- *Analyze*: it performs data analysis and reasoning on information provided by the monitor function;
- *Plan*: it reacts to a possible application objective change during execution and it structures the actions needed to achieve this new state;
- *Execute*: it modifies the behavior of managed resource, according to actions recommended by the plan function.

Finally, *Knowledge* source is composed by all data that is used

by the four functions; it includes information such as, for instance, topology structure, historical logs, metrics and policies.

Online autotuning, therefore, entrusts system management from people to technology, achieving self-configuration and self-optimization objectives; application requirements may change during execution and the overall system is able to properly react and to re-adapt itself.

2.5 MQTT messaging protocol

MQTT (Message Queue Telemetry Transport, [21]) is a lightweight messaging protocol that gives the possibility to establish remote communications among subjects. Its main characteristics are the minimization of network bandwidth and devices requirements; this features make MQTT ideal for machine-to-machine (M2M) or Internet of Things (IoT) world of connected devices, but in general this protocol have a large use in different projects: for instance, famous Facebook Messenger is built on top of it ([22]).

MQTT uses a client server publish/subscribe pattern: a client has the possibility to subscribe to topics and to publish messages on them (both topics and messages are strings); another component, called *broker* server, deals with the dispatch of messages to only clients that have subscribed to corresponding topic; so, publishers (clients that sends messages) and subscribers (clients that receive messages) don't know about the existence of one another: the broker, which is known by every client, distribute messages accordingly. Figure 2.8 shows a possible MQTT scenario with a sensor and two devices:

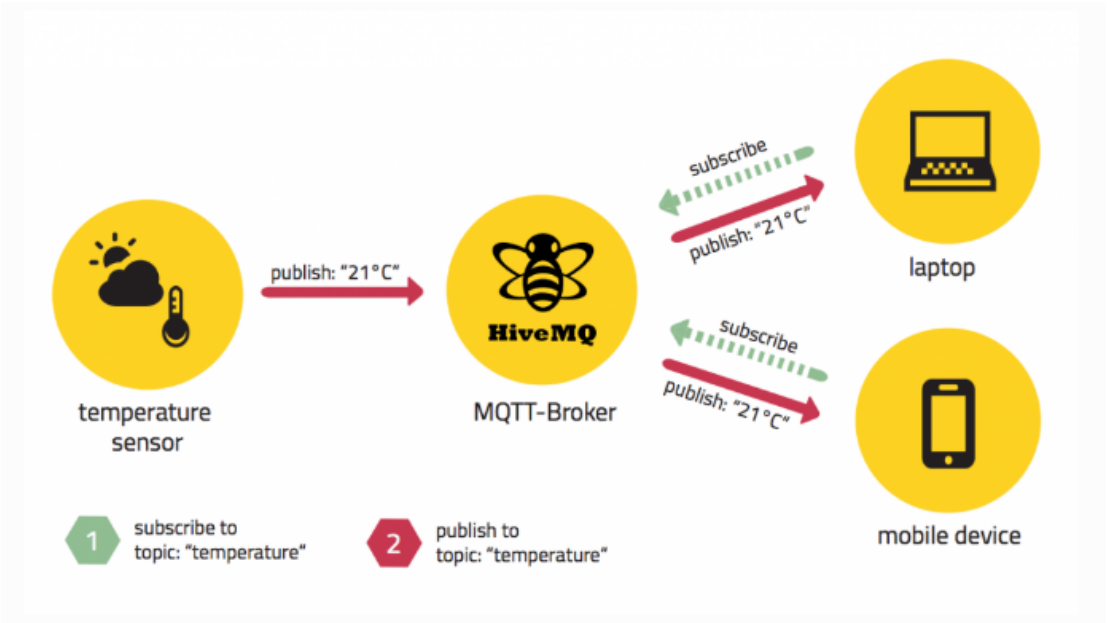


Figure 2.8: *MQTT publish/subscribe example, taken from [23]*

Topics are used by the broker to filter messages and to manage them in a correct way; they are made up of one or more levels, separated by a forward slash, as shown in figure 2.9:

Figure 2.9: *MQTT topic example, taken from [23]*

There is the possibility to subscribe to more topics at once through the use of wildcards: the single-level one (denoted with the symbol +) and the multi-level one (indicated with the symbol #).

Single-level wildcard substitutes an arbitrary level in a topic, so all topics that matches the same structure are associated to the one with single-level wildcard:

Figure 2.10: *MQTT topic with single-level wildcard example, taken from [23]*

For instance, *myhome/groundfloor/kitchen/temperature* and *myhome/groundfloor/livingroom/temperature* match topic in figure 2.10, while *myhome/groundfloor/kitchen/humidity* don't.

Multi-level wildcard is placed at the end of a topic and it covers an arbitrary number of topic levels:

Figure 2.11: *MQTT topic with multi-level wildcard example, taken from [23]*

In this case, for instance, *myhome/groundfloor/kitchen/temperature* and *myhome/groundfloor/kitchen/humidity* match topic in figure 2.11, while *myhome/firstfloor/livingroom/temperature* don't.

Another interesting MQTT feature is the Last Will and Testament (LWT): each client can specify a normal MQTT message with topic and payload. When it connects to the broker, this message is stored; if client abruptly disconnects, broker sends corresponding LWT to all subscribed clients on related topic, notifying occurred disconnection.

2.6 Apache Spark^{TM} MLlib library

Agorà takes advantage of the Machine Learning MLlib library by Apache Spark™ ([24]) in order to predict complete model of running

applications; in particular, we focuse on the Generalized Linear Regression interface.

2.6.1 (Generalized) Linear Regression

Taking [25] as reference, Linear Regression tries to model the relationship between a variable y and one or more variables $x_1, x_2, ...x_n, n \geq 1$; more rigorously, given a set of statistical units $\{y_i, x_{i1}, x_{i2}, ..., x_{ip}\}_{i=1}^{n}$, in which y_i is the variable that depends on the p-vector $[x_{i1}, x_{i2}, ..., x_{ip}]$, linear regression assumes that this relationship is linear:

$$y_i = \beta_0 \mathbf{1} + \beta_1 x_{i1} + \beta_2 x_{i2} + ... + \beta_p x_{ip} + \epsilon_i, \quad i = 1, 2, ...n \tag{2.3}$$

y_i is the *regressand*, *endogenous variable*, *response variable*, *measured variable*, *criterion variable* or *dependent variable*;

$x_{i1}, x_{i2}, ..., x_{ip}$ are called *regressors*, *exogenous variables*, *explanatory variables*, *covariates*, *input variables*, *predictor variables* or *independent variables*;

$\beta_0, \beta_1, ..., \beta_p$ are the *effects* or *regression coefficients*, whose values establish the relationship among regressand and regressors; β_0 is also called *intercept*; linear regression mainly focuses on the estimation of these parameters;

ϵ_i is called the *error term*, *disturbance term* or *noise*. It represents all other factors that influence y_i other than the predictor variables.

Linear Regression assumes that the response variable follows a Gaussian distribution; Generalized Linear Regression (GLR) gives the possibility to specify other distributions taken from the exponential family; it is useful for several kinds of prediction problems, including Linear Regression, Poisson Regression (for count data) and Logistic Regression (for binary data); moreover, GLR gives the possibility to specify a link function *g* that relates the mean of measured variable to exogenous variables. In case of Gaussian distribution for Linear Regression, link function can be equal to *Identity*, *Log* and

Inverse, with explicit meaning.

2.6.2 Regressors transformations

In order to use Linear Regression even if the relationship among dependent variable and independent variables is not linear, there is the necessity to modify input variables; Agorà implements two possible regressors transformations:

1. it transforms parameter values with inverse, square root and natural logarithmic functions; in union with the unaltered predictor variables, it tries to find the best model combining these transformations and available link functions; we refer to this strategy as "*transformations by functions*";
2. it transforms parameter values through polynomial expansion of second order: their cross-products and square values are added to the set of regressors, evaluating the best model with available link functions; we refer to this strategy as "*polynomial expansion of second order*"

Agorà chooses the model with the smallest Akaike Information Criterion (AIC) value, that is a measure of the quality, in terms of lost information, of statistical models for a given set of data; at the same AIC value, chosen model is the one with the smallest mean of the sum of coefficient standard errors, that measure how precisely predicted model has estimated regression coefficients with the given training set of data.

CHAPTER 3

State of The Art

We introduce main research studies that have represented a starting point for the design and implementation of Agorà; we divide them in two subcategories: the former is oriented to the concept of Design Space Exploration, the latter on autotuning.

3.1 Design Space Exploration related works

ReSPIR ([26]) proposes a DSE methodology for application-specific multiprocessor systems-on-chip (MPSoCs). First, a Design of Experiments phase is used to capture an initial plan of experiments that represent the entire target Design Space to be explored by simulations; after that, Response Surface Methodology techniques ([27])

identify the feasible solution area with respect to the system-level constraints, in order to refine the Pareto configurations until a predetermined criterion is satisfied.

MULTICUBE Explorer ([28]) is an open source, portable, automatic Multi-Objective Design Space Exploration framework for tuning multi-core architectures; a designer, through an existing executable model (use case simulator) and a Design Space definition through a XML file, can explore his parametric architecture.

ϵ-Pareto Active Learning (ϵ-PAL, [29]) aims at efficiently localize an ϵ-accurate Pareto frontier in Multi-Objective Optimization problems; it models objectives as Gaussian process models, in order to guide the iterative design evaluation and, therefore, to maximize progress on those configurations that are likely to be Pareto optimal.

ReSPIR and MULTICUBE are researches oriented on application-specific architecture design, while ϵ-PAL deals with the MOO problem in a general way; all these three works aim to obtain a Pareto front and their execution is done offline; Agorà uses the concept of Design Space Exploration but it is focused on Approximate Computing software strategies in executing applications; Agorà doesn't calculate Pareto frontier, but its goal is to provide complete application model through Machine Learning techniques; moreover, with respect to ReSPIR, MULTICUBE and ϵ-PAL, we want to avoid any offline DSE phase, driving it during programs execution.

Furthermore, Agorà is able to fulfill application Design Space Exploration in a shared way, among simultaneously running applications; with this improvement, DSE execution time is considerably reduced.

3.2 Autotuning related works

SiblingRivalry ([30]) proposes an always online autotuning framework that uses evolutionary tuning techniques ([31]) in order to adapt

parallel programs. It eliminates the offline learning step: it divides available processor resources in half and it selects two configurations, a "safe" one and an "experimental" one, according to an internal fitness function value; after that, the online learner handles the identical current request in parallel on each half and, according to the candidate that finishes first and meets target goals and requirements, it updates its internal knowledge about configurations just used for current objective functions. This technique is used until a convergence is reached or when the context changes and, therefore, new objectives have to be achieved. When objectives change, SiblingRivalry restarts its procedure until new result is obtained; Agorà doesn't ground its worklow on predetermined objectives: it is completely uninteresting about application goals and requirements, since it predicts the complete model for all metrics under examination; furthermore, computational power of the machine in which the tunable program is executed is not kept busy by Agorà, since application information gathering and model prediction are done remotely, on a different computer.

Capri framework ([32]) focuses on control problem for tunable applications which mostly use Approximate Computing ([12]) as improvement technique; given an acceptable output error, it tries to minimize computational costs, such as running time or energy consumption. There are two main phases: training, done offline, estimates error function and cost function, using Machine Learning techniques with a training set of inputs; the control algorithm, done online, finds the best knob setting that fulfills objectives. Also Agorà is oriented in tuning applications based on Approximate Computing techniques; it is focused on everything that precedes the control phase, supplying, to the dynamic online autotuner, metric of interest estimations for each possible application configuration; Agorà wants to eliminate any offline phase, giving the possibility to simply run

an application, driving its execution in order to collect a training set for model prediction; finally, the result is transmitted to application autotuner, that is in charge of managing the control phase.

mARGOt ([33]) proposes a lightweight approach to application runtime autotuning for multicore architectures; it is based on the Monitor-Analyze-Plan-Execute (MAPE) feedback loop ([13]), made up of a monitor infrastructure and an Application-Specific RunTime Manager (AS-RTM), based on Barbeque work ([34]): the former element captures runtime information that is used by the latter in order to tune application knobs, together with design-time knowledge and application multi-objective requirements. The user has to provide XML configuration files in which a list of mARGOt Operating Points, made by parameter values and related metric of interest values, desired monitors and application objectives are expressed; the AS-RTM module, starting from this design-time knowledge and evaluating runtime information, has the task of choosing, from time to time, the best application configuration that satisfies application goals and requirements as best as possible. This work represents the starting point for the conception of Agorà framework; mARGOt has to have the list of program Operating Points before execution: we want to produce this information while applications are running, removing to users the burden to make an offline step before taking advantage of autotuner capabilities.

There are other different autotuning frameworks in HPC context, yet based on design-time knowledge: OpenTuner ([35]) is able to build up domain-specific program autotuners in which users can specify multiple objectives; ATune-IL ([36]) is an offline autotuner that gives the possibility to specify a wide range of tunable parameters for a general-purpose parallel program; PetaBricks ([37]) is oriented on the definition of multiple algorithm implementations to solve a problem; Ananta Tiwari et al. ([38]) propose a scalable and general-

purpose framework for autotuning compiler-generated code, combining Active Harmony's parallel search backend ([39]) and the CHiLL compiler transformation framework ([40]); Green ([41]) is focused on the energy-conscious programming using controlled approximation; PowerDial ([42]) is a system that transforms static configuration parameters into dynamic knobs in order to adapt application behavior with respect to the accuracy of computation and the amount of resources. Agorà main difference is the absence of any kind of prior information about application features and the indifference on quality and quantity of metrics and objectives, in contrast with the last two mentioned works ([41] and [42]), focused on energy and accuracy goals.

Finally, among libraries for specific tasks, OSKI ([43]) provides a collection of low-level primitives that automatically tunes computational kernels on sparse matrices; SPIRAL ([44]) generates fast software implementations of linear signal processing transforms; ATLAS ([45]) builds up a methodology for the automatic generation of basic linear algebra operations, focusing on matrix multiplications. Agorà aims to generalize autotuning process.

CHAPTER 4

Proposed methodology

Tunable applications are characterized by the presence of specific parameters, also known as *knobs*, that influence program execution; their change produces different application results in terms of metric of interest values, as, for instance, throughput or power consumption. Figure 4.1 shows a typical parallel architecture with three nodes (node 1, 2 and 3) that are executing three tunable applications (application X, Y and Z respectively):

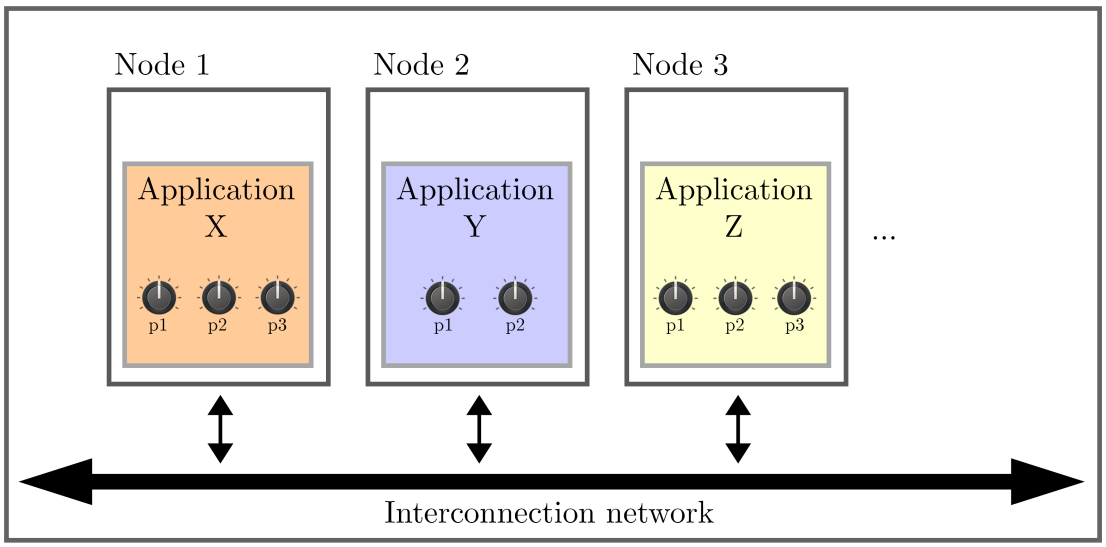


Figure 4.1: *An example of a parallel architecture with three executing tunable applications; application X on node 1 and application Z on node 3 have three knobs, application Y on node 2 two ones*

Very often, High Performance Computing applications expose a large set of parameters, making related Design Space huge and, consequently, unrealistic to explore it in an exhaustive way; in order to choose, from time to time, best program setting with the aim to improve energy efficiency with respect to power consumption and current input data, the concept of runtime autotuning is used: a class of online autotuners is able to choose, from time to time, best possible parameter values that fulfill application goals and requirements, starting from a design-time knowledge that gives information about parameter values and corresponding metric of interest values, built off-line. Figure 4.2 shows application Z interconnected with mARGOt [33], a dynamic autotuner developed by our research group:

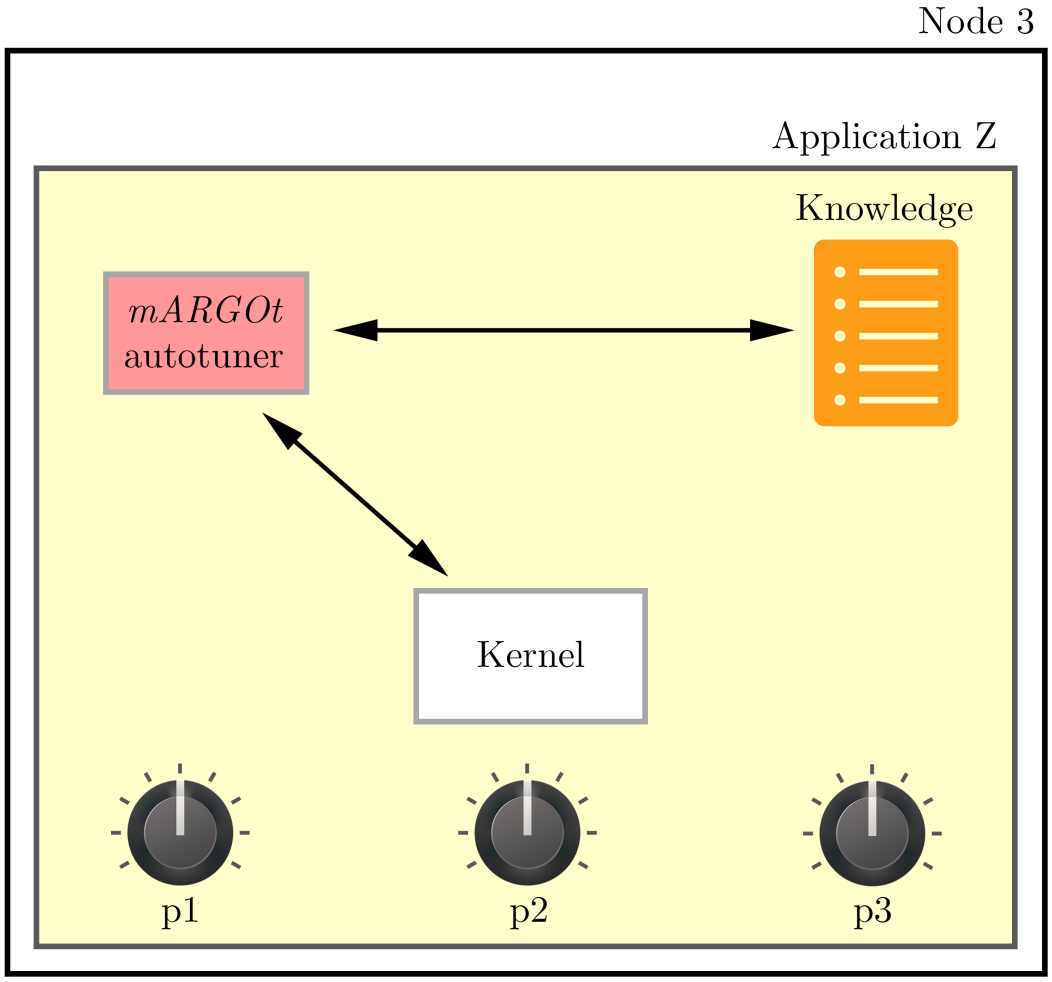


Figure 4.2: *Tunable application Z with the assistance of mARGOt autotuner schema*

As we can understand, this kind of autotuner needs application knowledge, so there have to be a preceding phase, before program start of computation, in which it is made; our improvement is to avoid this offline step, building, managing and updating application knowledge during execution itself. A local module mainly takes care of properly setting application knowledge, while a remote one manages collected information during execution, in order to predict complete application model. Figure 4.3 shows application Z interconnected with Agorà plus mARGOt autotuner:

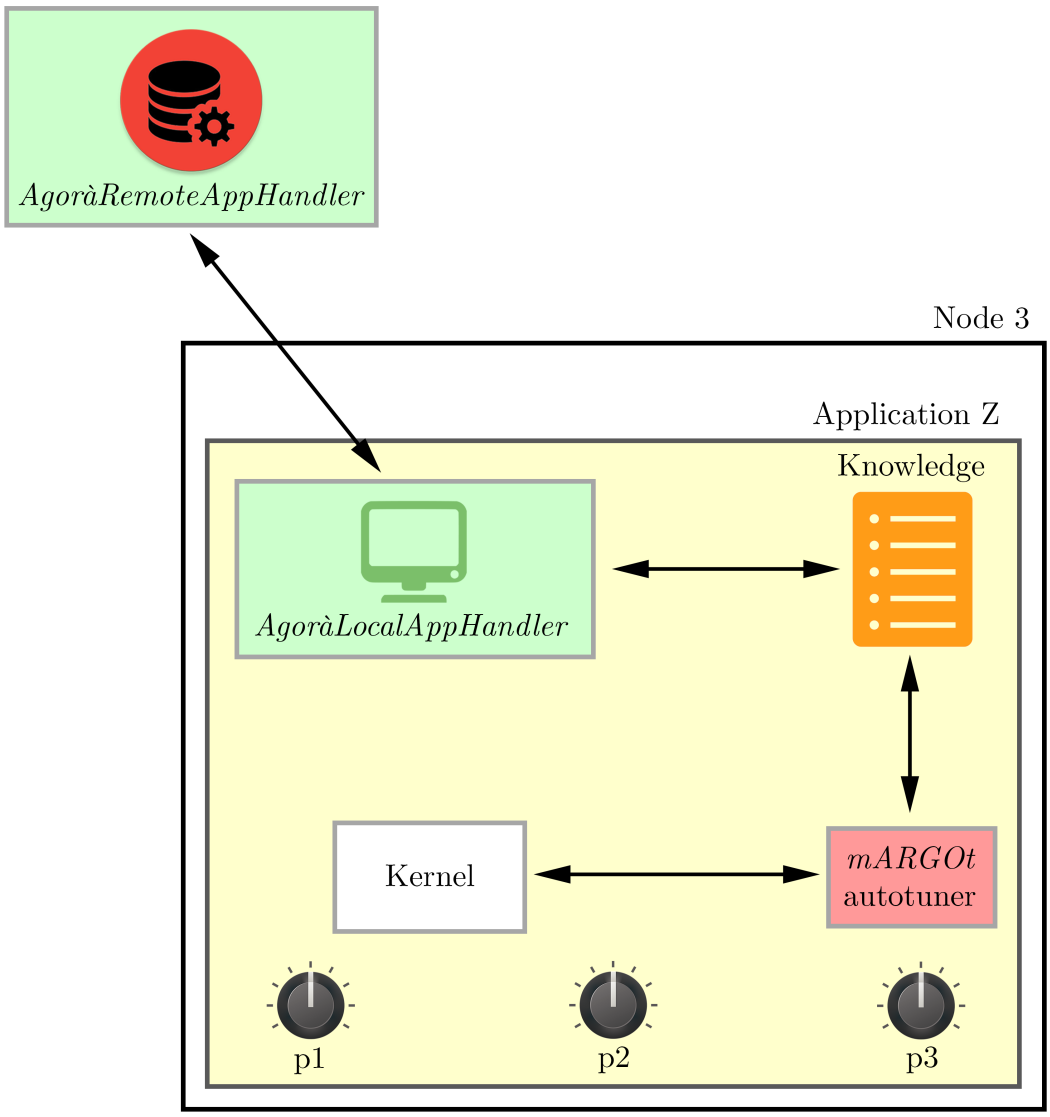


Figure 4.3: *Tunable application Z with the assistance of Agorà plus mARGOt autotuner schema*

This work, therefore, wants to address the problem of managing possible multiple applications that run, at the same time, inside a parallel architecture; main objective is to initially drive program execution with a subset of parameter configurations taken from their Design Space, in order to gather all metric of interest values associated to them; this list composes the training set for the prediction of application complete model through Machine Learning techniques.

Agorà can also correctly manage possible features; a feature is a particular application element than can not be set up like software knobs, but it contributes to the estimation of complete model; during DSE phase, feature values are observed like metric values while, during model prediction, they are considered as parameters, so their observations take part to the estimation of metric of interest values.

The typical architecture in which Agorà works is a parallel one, where there are multiple nodes, potentially heterogeneous, that execute applications; principal Agorà strenghts are:

1. the ability to drive Design Space Exploration in a distributed way, among all nodes that are running the same program, in order to considerably reduce DSE phase and to speed up overall workflow;

2. the ability to manage multiple kinds of applications, each of them separately organized by a dedicated Agorà module that is in charge of all nodes that execute the same program;

3. the out-of-band activity from parallel architecture data streams: computation of Design of Experiments configurations, collection of associated metric of interest values and complete model prediction are done in a separate node with respect to the ones that run applications inside the architecture, while the exchange of information is done using the lightweight MQTT protocol (discussed in chapter 2.5);

4. the persistence of generated knowledge: once application complete model is predicted, it is stored so, at any time, it can be reloaded and it is sent to new nodes that start running the same application, without repeating all the workflow through which the complete model has been previously predicted;

5. the capability of being fault-tolerant, with respect both to a run-

ning node crash and to the interruption of remote Agorà module that has the objective to predict application complete model: if the former situation happens, Agorà has to properly handle remaining running nodes; if the latter situation happens, running nodes inside the parallel architecture does not have to stop their execution but they react properly, according to the internal state of their related local Agorà module at that moment.

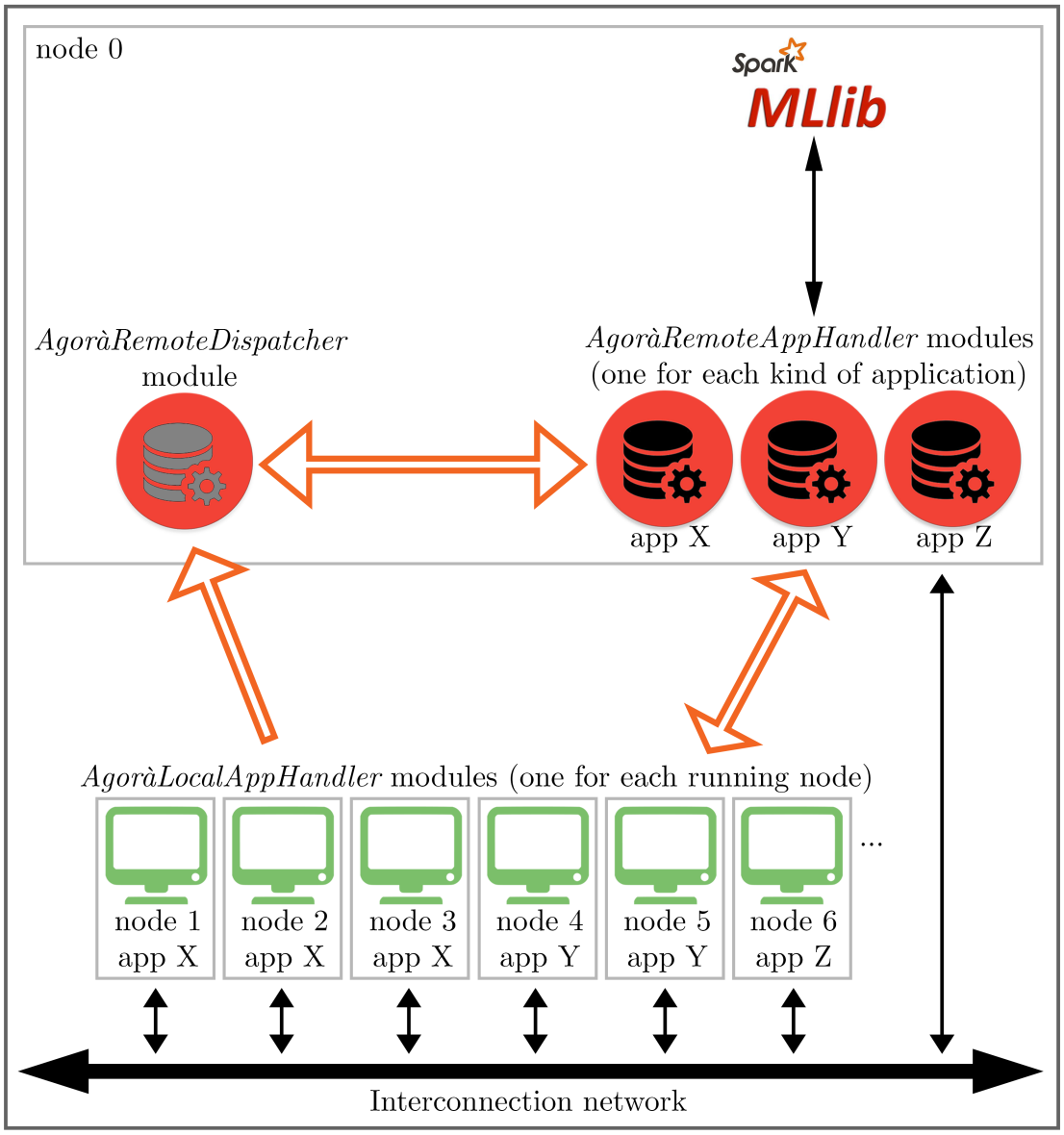


Figure 4.4: *Agorà overview in a parallel architecture*

Figure 4.4 shows all Agorà components and a possible scenario with a parallel architecture in which six nodes are running three different tunable applications; for every kind of application there exists a dedicated AgoràRemoteAppHandler module that manages it; orange empty arrows represent all possible communications among modules, made possible through MQTT subscriptions and publications on predetermined topics.

Agorà main components are:

- the *AgoràDispatcher* module, written in Python: it keeps waiting for program arrival, in order to properly manage them;
- the *AgoràRemoteAppHandler* module, written in Python: it is created by the AgoràDispatcher for every kind of application; it asks for application information such as, for instance, all parameter name and values; it computes application configurations that compose the Design of Experiments; it drives Design Space Exploration phase, distributing DoE configurations among all nodes it manages; it collects parameter values and the observed metrics of interest sent by running programs; it makes use of Machine Learning techniques in order to build application complete model; it sends result to connected nodes;
- the *AgoràLocalAppHandler* module, written in C++: it is set up in every executing program; it communicates with the autotuner that manages application behavior; it notifies the existence of related running machine to the AgoràDispatcher module; it replies to possible information request made by the related AgoràRemoteAppHandler; during Design Space Exploration phase, it receives configurations from the AgoràRemoteAppHandler module, it sets up program knowledge with this information and, after the application has done computation, it sends back all obtained information, regarding parameter values and associated

metric of interest values; it saves predicted complete model received from the AgoràRemoteAppHandler module, in order to properly update application knowledge with this data.

Principal use cases that define framework workflow and the interaction among components are the following:

1. application start of execution: when programs start running, related AgoràLocalAppHandler module notifies their existence to the AgoràDispatcher module; if the application is unknown, AgoràDispatcher creates a dedicated AgoràRemoteAppHandler module that is in charge of managing it, otherwise it communicates new node to the corresponding existing AgoràRemoteAppHandler module;

2. Design of Experiments computation: AgoràRemoteAppHandler module computes the subset of configurations, from the entire application Design Space, that compose the Design of Experiments; after that, it is ready to drive Design Space Exploration phase, distributing these configurations to requesting nodes;

3. configuration reception: AgoràLocalAppHandler module updates application knowledge with all the configurations that, from time to time, are sent by the AgoràRemoteAppHandler module; every time program computation is finished, it sends back to the AgoràRemoteAppHandler module a list of values, made by the configuration just used with the observed metrics of interest;

4. application configuration and related metric value collection: AgoràRemoteAppHandler module collects all the information it receives from running nodes; when it has all necessary data, it uses Machine Learning techniques in order to predict application complete model, made by all possible configurations associated with predicted metric values;

5. predicted model dispatch: application complete model is sent by AgoràRemoteAppHandler to associated running nodes; related AgoràLocalAppHandler modules update program knowledge with this information, so the dynamic autotuner can set up application knobs with the best configuration that fulfills current goals and requirements.

The interaction among Agorà components is implemented in an asynchronous way: program executions are independent from MQTT message exchange and all modules properly react to these events, in order to not condition application workflow and to not steal execution time, making all process as flowing as possible.

CHAPTER 5

Agorà: proposed framework

Chapter 5 shows technical implementation of Agorà framework, going into detail of all possible use cases; in the last part, a sketch application explains how AgoràLocalAppHandler module is integrated and how it works.

5.1 Introduction

Agorà makes use of the MQTT protocol ([21]) for the communications among components: it uses Eclipse Paho MQTT Python Client and Eclipse Paho MQTT C Client for managing message exchange ([46]), while it uses Eclipse Mosquitto as broker ([47]). We take addvantage of the Machine Learning library MLlib by Apache Spark™ ([24]) in order to predict application complete models.

Next technical use case implementation makes use of application *Swaptions* as reference, taken from the PARSEC benchmark suite ([48]). This application is a workload which prices a portfolio of swaptions through Monte Carlo simulations; it has two tunable parameters, the number of threads (variable *num_threads*, from 1 to 8) and the number of trials for the simulation (variable *num_trials*, from 100.000 to 1.000.000 with a step of 100.000); observed metrics of interest are: throughput (variable $avg\_throughput$) as the number of priced swaptions per second and error (variable avg_error), computed as:

$$avg\_error = \frac{\sum_{s \in pricedSwaptions} |StandDevRef(s) - StandDev(s)|}{|pricedSwaptions|}$$

where $StandDevRef(s)$ is the reference standard deviation for swaption s, $StandDev(s)$ is the evaluated one and $pricedSwaptions$ represents the set of swaptions that are priced at each computing cycle; so, metric avg_error stands for the average of differences between standard deviation of priced swaptions using evaluated configuration with respect to the reference one (standard deviation for 1.000.000 trials).

Agorà has been interconnected to mARGOt autotuner ([33]), that exploits design-time knowledge to dynamically adapt application behavior during execution; mARGOt represents this information as a list of Operating Points (OPs): an OP is made by a set of parameter values, also called software knobs, in union with the associated performance (metric of interest values), profiled at design-time; Agorà improvement is to build application knowledge at runtime, with an online distributed Design Space Exploration phase in which a subset of OPs are collected, in order to predict the complete model, made by the entire Operating Point list.

Agorà could work in union with other autotuners that, using appli-

cation knowledge in terms of configurations and associated performances, have the capability to dynamically adapt application behavior during execution.

5.2 Use case implementation

5.2.1 AgoràRemoteDispatcher module creation

The starting point is the creation of the AgoràRemoteDispatcher module, that is in charge of managing the arrival of applications; it connects to the MQTT broker and it subscribes to topic "agora/apps":

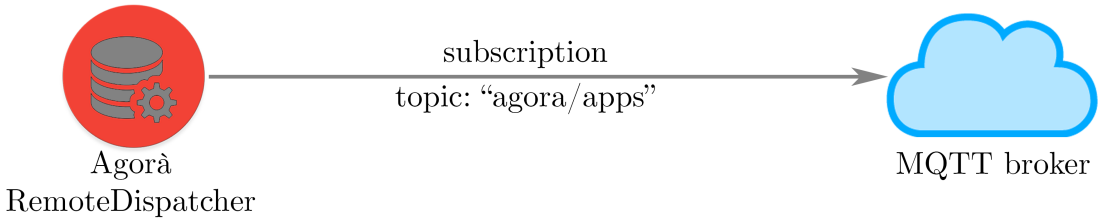


Figure 5.1: *AgoràRemoteDispatcher subscription*

5.2.2 Application arrival

An application can be already known by the AgoràRemoteDispatcher module or a program is executed by a machine for the first time.

5.2.2.1 Unknown application

A node starts running a program; the related AgoràLocalAppHandler module notifies this event, publishing on topic "agora/apps" a string composed of application name and machine hostname plus Process IDentifier (PID), with format "*[appName] [hostname]_[PID]*", so that AgoràLocalAppHandler module can be univocally recognized in the future; the message is received by AgoràRemoteDispatcher, that creates a dedicated AgoràRemoteAppHandler module for this application:

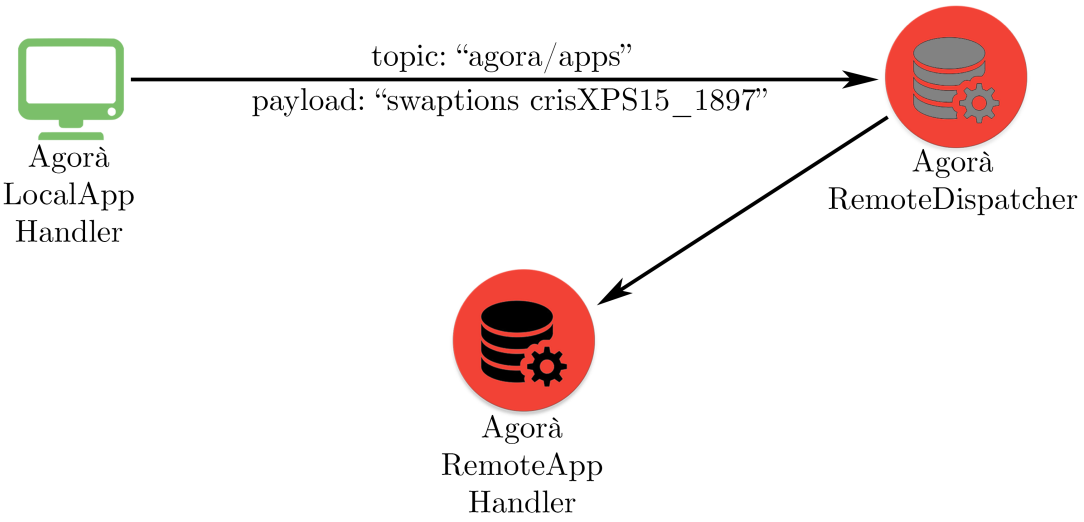


Figure 5.2: *New unknown application arrival example; a dedicated AgoràRemoteAppHandler module is created by AgoràRemoteDispatcher*

At the beginning, the AgoràLocalAppHandler module subscribes to some topics that are needed to receive communications from the related AgoràRemoteAppHandler:

1. "agora/*[appName]*", in order to understand if AgoràRemoteAppHandler has asked application information and, therefore, to reply (see 5.2.3.1); this topic is also used to understand if AgoràRemoteAppHandler has crashed and, so, to react properly (see 5.2.7);

2. "agora/*[appName]*/*[hostname]_[PID]*/conf", in order to receive configurations from AgoràRemoteAppHandler during DSE phase (see 5.2.3.4);

3. "agora/*[appName]*/*[hostname]_[PID]*/model", in order to receive a partial OP list (see 5.2.3.5) and the complete predicted model from AgoràRemoteAppHandler (see 5.2.3.6).

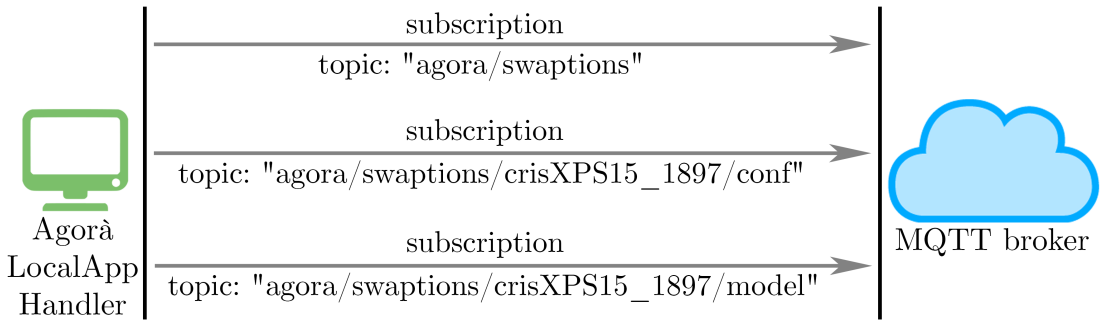


Figure 5.3: *AgoràLocalAppHandler MQTT subscriptions example*

The AgoràRemoteAppHandler module subscribes to some topics in order to manage correctly all various situations that happens:

1. "agora/*[appName]*/newHostpid", in order to manage the hypothetical notification of other AgoràLocalAppHandler modules that are supervising the same application (see 5.2.2.2);

2. "agora/*[appName]*/req", in order to manage all the requests made by AgoràLocalAppHandler modules during program execution (see 5.2.3);

3. "agora/*[appName]*/info/#", in order to receive all available application information, such as parameter name and values (see 5.2.4); real topic is "agora/*[appName]*/info/*[hostname]_ [PID]*" (see MQTT multi-level wildcard, 2.5), therefore AgoràRemoteAppHandler can store the ID of the node that is sending application information, in order to react properly to node possible crash during this phase (see 5.2.6);

4. "agora/*[appName]*/disconnection", in order to correctly react to a possible node disconnection (see 5.2.6);

5. "agora/*[appName]*/OPs", in order to receive Operating Points from AgoràLocalAppHandler modules during Design Space Exploration phase (see 5.2.5).

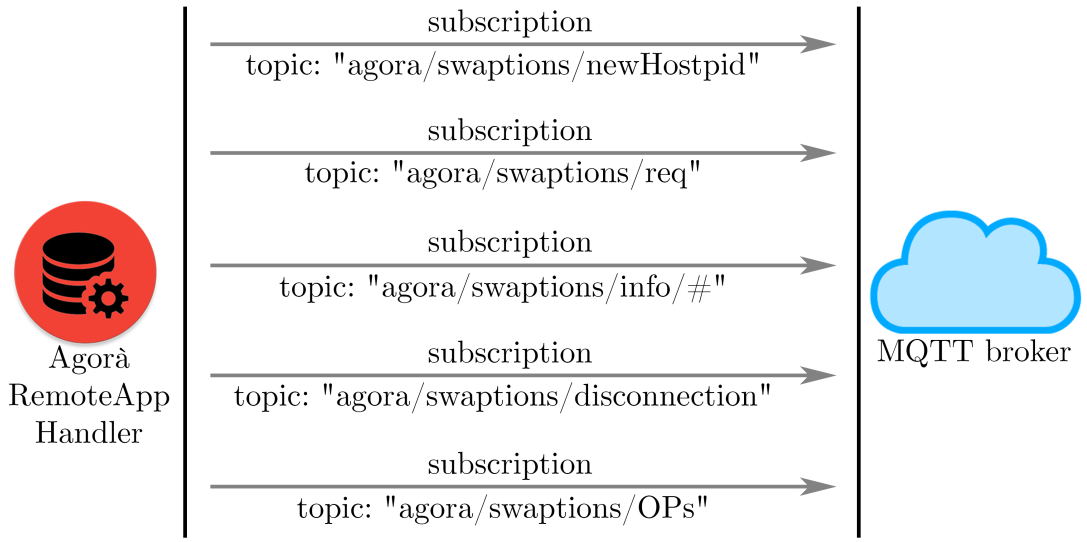


Figure 5.4: *AgoràRemoteAppHandler MQTT subscriptions example*

5.2.2.2 Known application

When the AgoràRemoteDispatcher module is informed that a new node has started running an application but there exist already an AgoràRemoteAppHandler that is managing that program, it publishes on topic "agora/*[appName]*/newHostpid" the new machine hostname plus PID, so that the corresponding AgoràRemoteAppHandler module can add the node to the pool of machines that are running the application it is supervising:

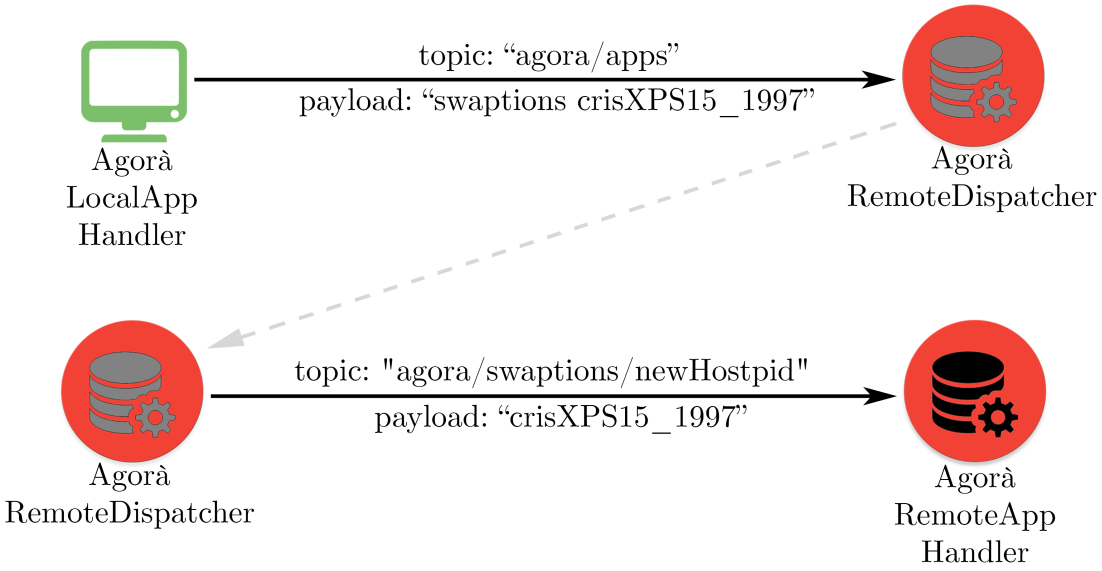


Figure 5.5: *New known application arrival example; node ID is sent by AgoràRemoteDispatcher to the corresponding AgoràRemoteAppHandler module*

5.2.3 AgoràLocalAppHandler request

At each predetermined time interval, AgoràLocalAppHandler modules make a request to the related AgoràRemoteAppHandler, publishing their hostname plus PID on topic "agora/*[appName]*/req":

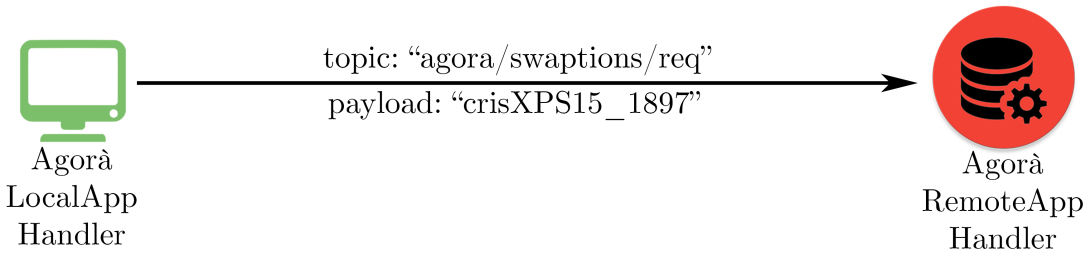


Figure 5.6: *AgoràLocalAppHandler request example*

This kind of publication is repeated until the node receives the predicted complete Operating Point list; AgoràRemoteAppHandler replies to these requests according to its internal state, that can be one of the following:

1. *unknown;*

2. *receivingInfo*;
3. *buildingDoE*;
4. *DSE*;
5. *buildingTheModel*;
6. *autotuning*.

5.2.3.1 AgoràRemoteAppHandler internal state equal to *unknown*

AgoràRemoteAppHandler doesn't know anything about the application it is managing, except its name; it asks to AgoràLocalAppHandler modules all available information, making a publication with payload "info" on topic "agora/*[appName]*":

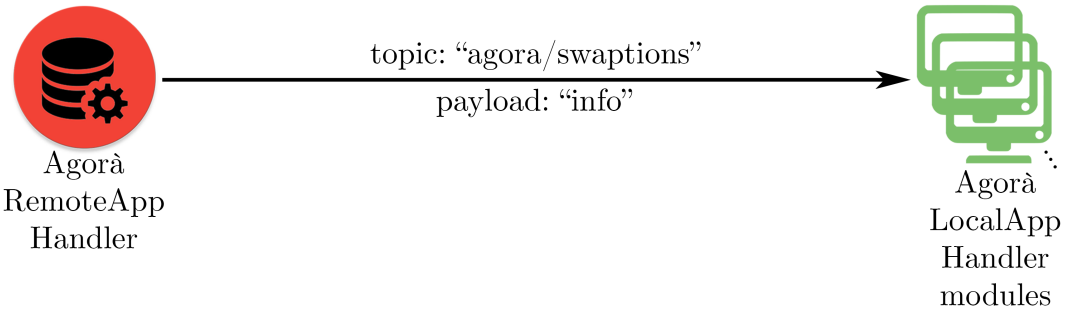


Figure 5.7: *Application information request by AgoràRemoteAppHandler example*

5.2.3.2 AgoràRemoteAppHandler internal state equal to *receivingInfo*

The AgoràRemoteAppHandler module is receiving application information, so in this case it discards all possible requests made by AgoràLocalAppHandler modules.

5.2.3.3 AgoràRemoteAppHandler internal state equal to *buildingDoE*

AgoràRemoteAppHandler, according to the received Design of Experiments type (see 5.2.4), is building the set of configurations that

are going to be distributed to AgoràLocalAppHandler modules during Design Space Exploration phase; also in this case it discards all possible AgoràLocalAppHandler module requests.

5.2.3.4 AgoràRemoteAppHandler internal state equal to *DSE*

The AgoràRemoteAppHandler module has computed DoE configurations, so it is driving Design Space Exploration phase; it picks up the first element on top of the available configuration list and it sends, in lexicographic order, the associated software knob values on topic "agora/*[appName]*/*[hostname]*_*[PID]*/conf", relative to the AgoràLocalAppHandler module that made the request; the configuration just sent is reinserted at the end of the mentioned list:

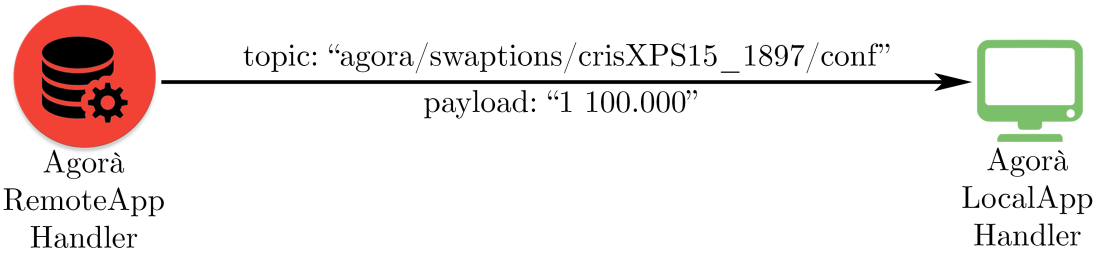


Figure 5.8: *Configuration dispatch by AgoràRemoteAppHandler example*

As shown in figure 5.8, AgoràLocalAppHandler receives a configuration with $num\_threads = 1$ and $num\_trials = 100.000$; next computation is going to be done with these parameter values.

5.2.3.5 AgoràRemoteAppHandler internal state equal to *buildingTheModel*

AgoràRemoteAppHandler has gathered all needed OPs related to DoE configurations and it is computing complete Operating Point list through Machine Learning techniques; from gathered Operating Points, a partial model is built, assembling to every DoE configuration the mean of metric values, taken from the corresponding Operating Points; this partial model in sent to the AgoràLocalAppHan-

dler module that made the request: each obtained OP is published on topic "agora/*[appName]/[hostname]_[PID]*/model" with format "*[configuration] [metric values]*"; both configuration and metric values follow lexicographic order; finally, AgoràRemoteAppHandler makes a final publication on same topic with payload "DoEModelDone":

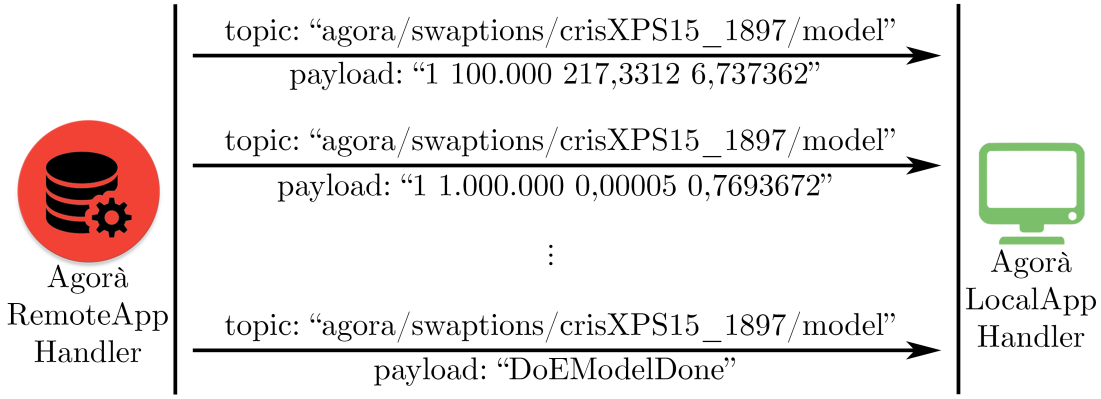


Figure 5.9: *Partial model dispatch by AgoràRemoteAppHandler example*

Taking figure 5.9 as reference, the first sent OP has parameters $num\_threads = 1$ and $num\_trials = 100.000$, with metrics $avg\_error = 217,3312$ and $avg\_throughput = 6,737362$; the AgoràLocalAppHandler module sets up mARGOt autotuner with this OP list, so the application is executed with the best Operating Point that fulfills current goals and requirements.

5.2.3.6 AgoràRemoteAppHandler internal state equal to *autotuning*

The AgoràRemoteAppHandler module owns the complete OP list, obtained through the Generalized Linear Regression interface by Apache Spark™ MLlib library; similarly to the previous case (5.2.3.5), every Operating Point is sent to AgoràLocalAppHandler on topic "agora/*[appName]/[hostname]_[PID]*/model" with format "*[configuration] [metrics values]*", respecting lexicographic order for both parameter and metric values; the final publication has payload "modelDone":

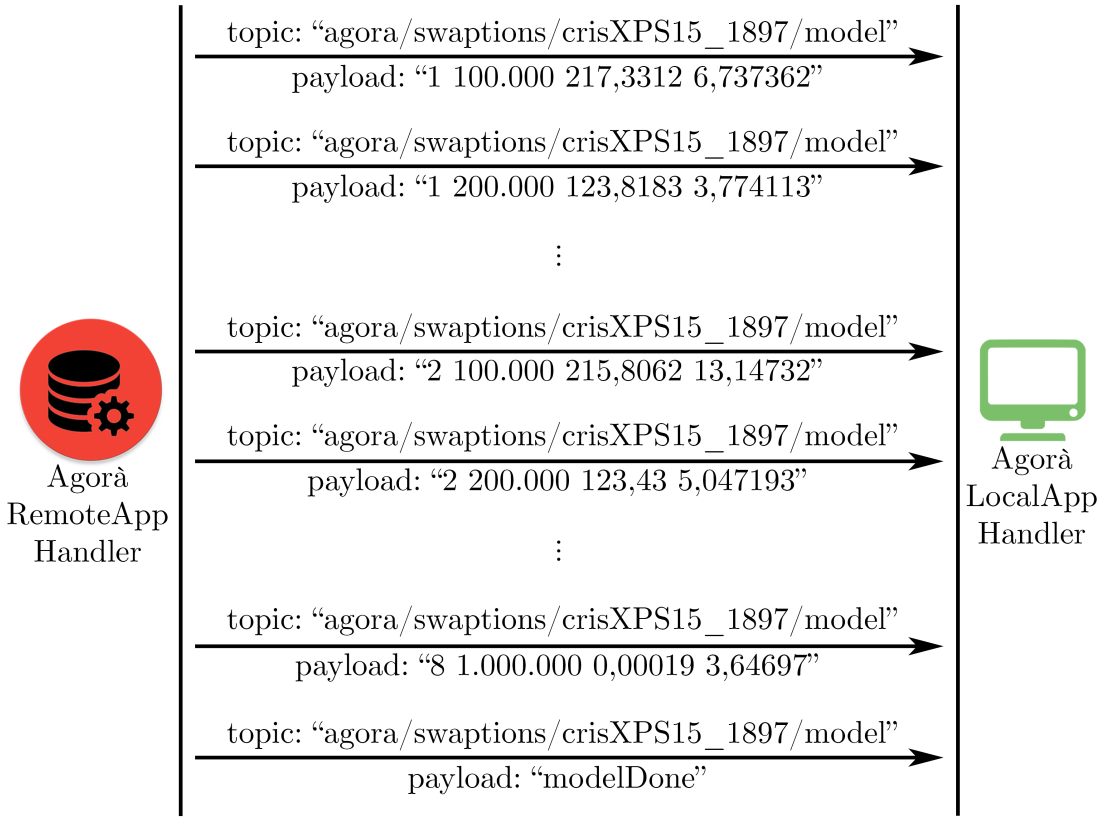


Figure 5.10: *Complete model dispatch by AgoràRemoteAppHandler example*

In Swaptions application, parameter $num\_threads$ can assume 8 different values, while parameter num_trials 10 ones, so the complete model is composed by all the 80 OPs; after mARGOt autotuner has received all the predicted Operating Points, it can set up application knobs according to current objectives.

From this point on, AgoràLocalAppHandler stops making requests to the AgoràRemoteAppHandler module.

5.2.4 Application information dispatch by AgoràLocalAppHandler

It has been shown that, if an AgoràRemoteAppHandler module receives a request from a node but its internal state is *unknown*, it requests application information (see 5.2.3.1); AgoràRemoteAppHan-

dler saves both the identifier of the first AgoràLocalAppHandler module that replies and all data it receives; other possible replies from other AgoràLocalAppHandler modules are discarded.

Mandatory information that AgoràLocalAppHandler modules have to send is:

1. metrics under examination: the keyword is *metric*, followed by metric name; there is a publication for each metric; publications have to be in lexicographic order with respect to metric name;

   e.g. payload: "metric avg_throughput"

2. application parameters: the keyword is *param*, followed by parameter name, the way in which it is transmitted and corresponding values; there is a publication for each parameter; also in this case, publications must follow lexicographic order with respect to parameter name; Agorà makes available two ways to send values:

   (a) by list: the keyword is *enum* and, in this case, all possible values are listed;

   (b) by extreme values and step: the keyword is *range* and, in this case, minimum value, maximum value and step are sent; AgoràRemoteAppHandler module, from this information, computes all possible parameter values.

   e.g. payload: "param num_threads enum 1 2 3 4 5 6 7 8"

   e.g. payload: "param num_trials range 100.000 1.000.000 100.000"

There are some optional information that AgoràLocalAppHandler modules can send to the AgoràRemoteAppHandler:

1. number of required repetitions for each Operating Point: the keyword is *numReps*, followed by a number; this value represents the number of Operating Points that the AgoràRemoteAp-

pHandler has to gather for each Design of Experiments configuration, during Design Space Exploration phase;

e.g. payload: "numReps 5"

2. Design of Experiments type: the keyword is *DoE*, followed by the term that indicates the kind of Design of Experiments that has to be used; it can be:

   (a) *fcccd*: it corresponds to the Face Centered Central Composite DoE with one Center Point;

   (b) *ff2l*: it corresponds to the 2-Level Full-Factorial DoE;

   (c) *pbd*: it corresponds to the Plackett-Burman DoE;

   (d) *lhd*: it corresponds to the Latin-Hypercube DoE; in this case, there is another optional information that can be sent to the AgoràRemoteAppHandler module: the number of configurations that have to be produced, with the word *lhdSamples* followed by the desired value; if this information is not sent, the number of random configurations is equal to the number of application parameters;

   (e) *fcccdExtra*: it corresponds to the Face Centered Central Composite DoE with one Center Point plus the addition of other configurations through the Latin-Hypercube DoE; as in the previous case, the optional keyword *lhdSamples* is used to express the number of extra configurations, otherwise they are equal to the number of parameters;

   (f) *fullFact*: it corresponds to the Full-Factorial DoE.

   We refer to chapter 2.3 for Design of Experiments detailed information.

   e.g. payload: "DoE fcccdExtra"

   e.g. payload: "lhdSamples 6"

3. Response Surface Modeling technique: the keyword is *RSM*, followed by the term that indicates the Machine Learning technique that has to be used in order to predict application complete OP list; Agorà implements two versions of the Apache Spark™ Generalized Linear Regression RSM, explained in detail in 2.6.2:

   (a) the 1st version ("*transformations by functions*"), that uses parameter values transformations, with associated word *sparkGenLinRegrTransforms*;

   (b) the 2nd version ("*polynomial expansion of second order*"), that uses parameter polynomial expansion of second order, with associated word *sparkGenLinRegr2ndPolyExp*

   e.g. payload: "RSM sparkGenLinRegr2ndPolyExp"

4. parameter value transformations for the 1st implemented version of Apache Spark™ Generalized Linear Regression RSM: the keyword is *paramsTransforms*, followed by the involved metric name and the terms that indicate the kind of parameter transformations, the family distribution and the link function.

   Transformations must follow the same order of parameter information dispatch; they can be:

   (a) *inv*: in this case, to the corresponding parameter values in the OPs, the inverse function is applied;

   (b) *ln*: in this case, to the corresponding parameter values in the OPs, the natural logarithmic function is applied;

   (c) *sqrt*: in this case, to the corresponding parameter values in the OPs, the square root function is applied;

   (d) *id*: in this case, the corresponding parameter values in the OPs are not transformed.

Agorà focuses on the prediction of continuous functions with normal distribution: corresponding family is the Gaussian one, indicated with word *gaussian*.

For Gaussian family, link function can be:

(a) *identity*;

(b) *log*;

(c) *inverse*.

If this kind of information is available, it must exist for each metric of interest. We refer to chapter 2.6.1 for more detailed information about family and link function.

e.g. payload: "paramsTransforms avg_error id sqrt gaussian log"

5. number of application features: the keyword is *numFeats*, followed by the corresponding number; in this case, a minimum number n of feature value observations can be specified through the keyword *minNumObsFeatValues*, followed by n: only those feature values that are observed at least n times, during Design Space Exploration phase, contribute to the prediction of application complete OP list; if no feature value reaches n observations, n becomes the number of observations of the most observed feature value; if this last information is missing, $n = 1$.

   e.g. payload: "numFeats 1"

   e.g. payload: "minNumObsFeatValues 5"

If not specified, default Design of Experiments type is Face Centered Central Composite DoE with one Center Point, default number of repetitions for each OP is 1, default number of program features is 0 and default RSM technique is the "*polynomial expansion of second order*" version of the Apache Spark™ Generalized Linear Regression. If the chosen RSM technique is the "*transformations by functions*"

GLR version but there is no information about parameter value transformations, Agorà tries every possible combination in union with all possible link functions, choosing the best result according to Akaike Information Criterion measure and the mean of the sum of coefficient standard errors (see 2.6.2).

All information is sent by AgoràLocalAppHandler modules on topic "agora/*[appName]*/info/*[hostname]_[PID]*"; a final publication with message "done" specifies to the AgoràRemoteAppHandler that application information is finished:

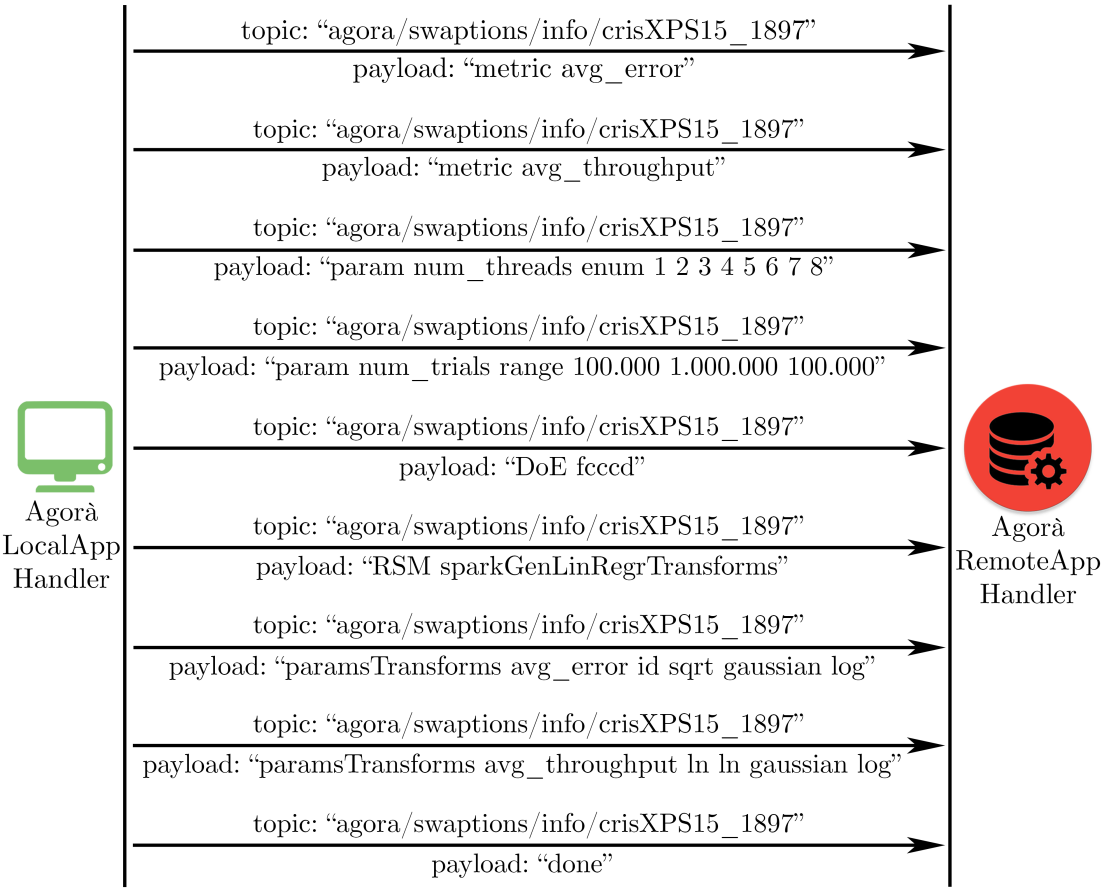


Figure 5.11: *Application information dispatch by AgoràLocalAppHandler module example*

Taking figure 5.11 as reference, the AgoràRemoteAppHandler pulls out, from topic, *crisXPS15_1897* and it saves this identifier as the AgoràLocalAppHandler that is sending program information; as we

can see, *Swaptions* application focuses on two metrics, avg_error and $avg\_throughput$; it has two parameters, $num\_threads$ and num_trials; the former is sent with the complete list of values, while the latter is sent with the keyword *range*, specifying its minimum value (100.000), its maximum one ($1.000.000$) and the step (100.000): the AgoràRemoteAppHandler computes all values, that are therefore 100.000, 200.000, 300.000, ..., $1.000.000$; the Design of Experiments that has to be used is the Face Centered DoE with one Center Point and the RSM technique is the 1st version of the Apache Spark™ Generalized Linear Regression; parameter transformations are also specified so, e.g. for metric avg_error, the first parameter ($num\_threads$) will not be transformed, while the second parameter (num_trials) has to be transformed with the square root function, applying *gaussian* as family distribution and *log* as link function; it can be noticed that there is no information about the number of Operating Point repetitions to collect during DSE phase: in this case, default value is used (1 repetition for each OP); finally, there is no payload with keyword *numFeats* either: *Swaptions* does not have features.

The AgoràRemoteAppHandler module is now ready to compute Design of Experiments configurations; after that, it can start distributing them to nodes, driving the subsequent Design Space Exploration phase.

5.2.5 Operating Point dispatch by AgoràLocalAppHandler module

During DSE, AgoràLocalAppHandler modules store configurations that receive from AgoràRemoteAppHandler (see 5.2.3.4); every time a configuration is sent, if the one in use differs from the new one, AgoràLocalAppHandler communicates to mARGOt autotuner new parameter values, therefore next program computation is executed with new parameter values.

When execution is finished, the AgoràLocalAppHandler module arranges the obtained Operating Point, composed by the set of parameter values and the observed metrics of interest; it publishes on topic "agora/*[appName]*/OPs" a message in the form "*[configuration]*:*[metric values]*", in which both value lists have to follow lexicographic order with respect to parameter and metric name; if there are application features, payload is in the form "*[configuration]*:*[observed feature values]*:*[metric values]*":

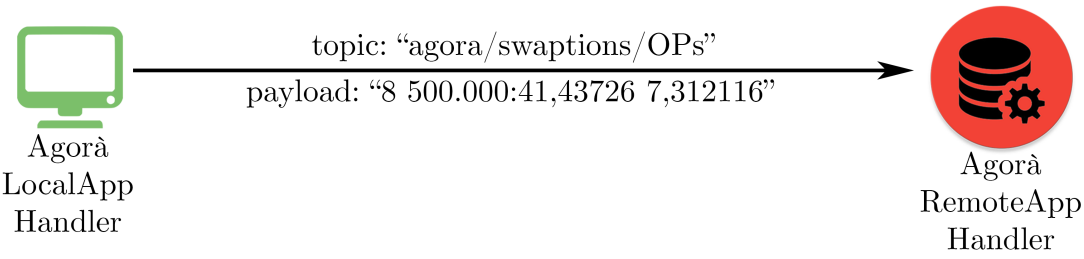


Figure 5.12: *Operating Point dispatch by AgoràLocalAppHandler module example*

In the example above, *Swaptions* has been just executed with $num\_threads = 8$ and $num\_trials = 500.000$; monitored metric values are $avg\_error = 41,43726$ and $avg\_throughput = 7,312116$.

When AgoràRemoteAppHandler receives an Operating Point, it decrements the corresponding number of needed OP repetitions: if the updated value is equal to zero, it means that there is no need of other related OPs, so the corresponding configuration is moved from the set of available ones to the set of accomplished ones.

Model prediction just starts when last needed Operating Point repetition related to last available configuration is received from a node.

5.2.6 AgoràLocalAppHandler module disconnection

If an AgoràLocalAppHandler module disconnects for some reason, related AgoràRemoteAppHandler receives a message on topic "agora/*[appName]*/disconnection" with payload "*[hostname]_[PID]*", rela-

tive to disconnected AgoràLocalAppHandler; the AgoràRemoteAppHandler module removes node identifier from the list of machines it is managing:

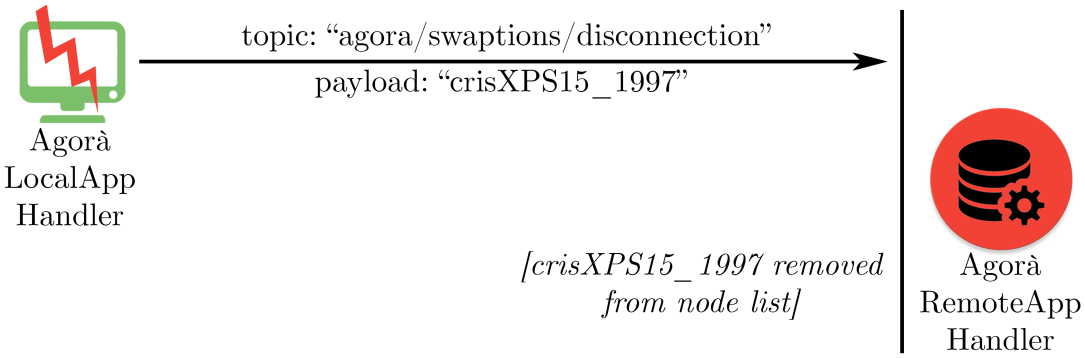


Figure 5.13: *AgoràLocalAppHandler module disconnection example*

Particular attention has to be taken if the disconnected AgoràLocalAppHandler module is the one that, at the beginning, is sending application information (see 5.2.4): in this case, AgoràRemoteAppHandler has to remove disconnected machine, it has to reset partial data (received up to that moment) and it asks again all available application information to remaining connected AgoràLocalAppHandler modules:

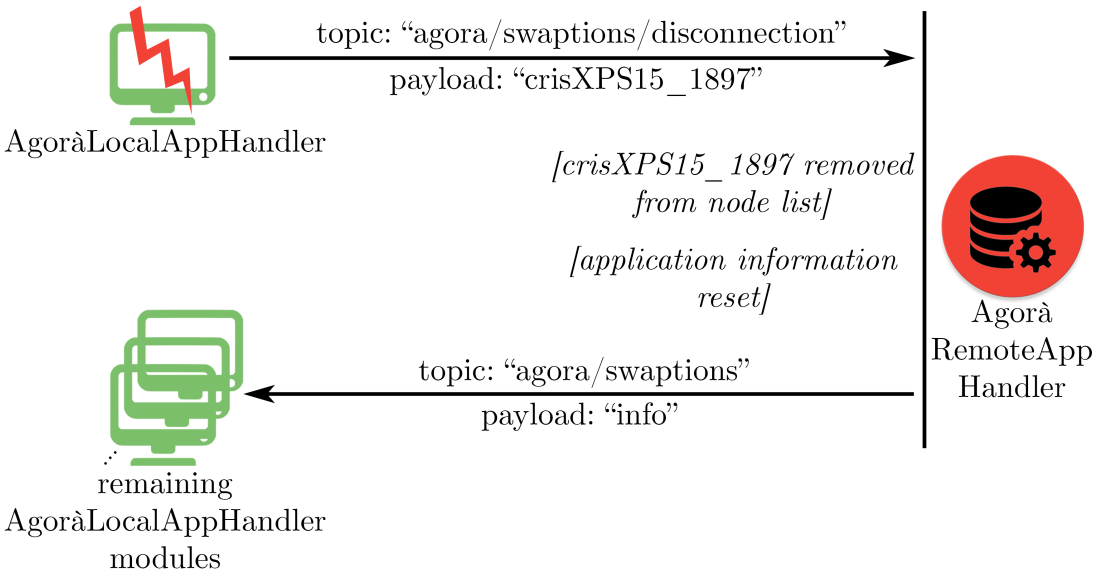


Figure 5.14: *Disconnection of AgoràLocalAppHandler module that was sending application information example*

5.2.7 AgoràRemoteAppHandler module disconnection

If the AgoràRemoteAppHandler module disconnects, related AgoràLocalAppHandler modules receive a message on topic "agora/*[appName]*" with payload "disconnection":

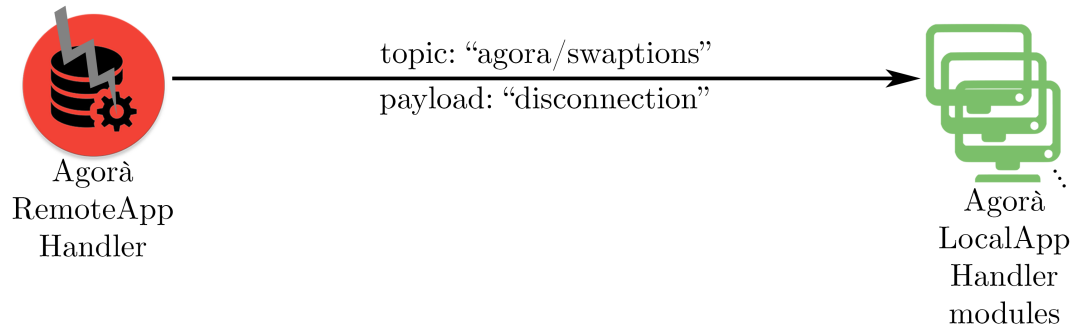


Figure 5.15: *AgoràRemoteAppHandler module disconnection example*

Each AgoràLocalAppHandler reacts to this event according to its internal state, that can be:

1. *defaultStatus*;
2. *DSE*;

3. *DoEModel*;
4. *autotuning*.

5.2.7.1 AgoràLocalAppHandler internal state equal to *defaultStatus*

When a node starts running a program, the autotuner sets up application parameter values with a predetermined default configuration; if any Design Space Exploration phase has not been started yet, AgoràRemoteAppHandler disconnection does not affect application behavior.

5.2.7.2 AgoràLocalAppHandler internal state equal to *DSE*

The AgoràRemoteAppHandler module is driving Design Space Exploration phase, sending configurations to AgoràLocalAppHandler modules; in this case, default configuration is restored and the application is executed with corresponding parameter values:

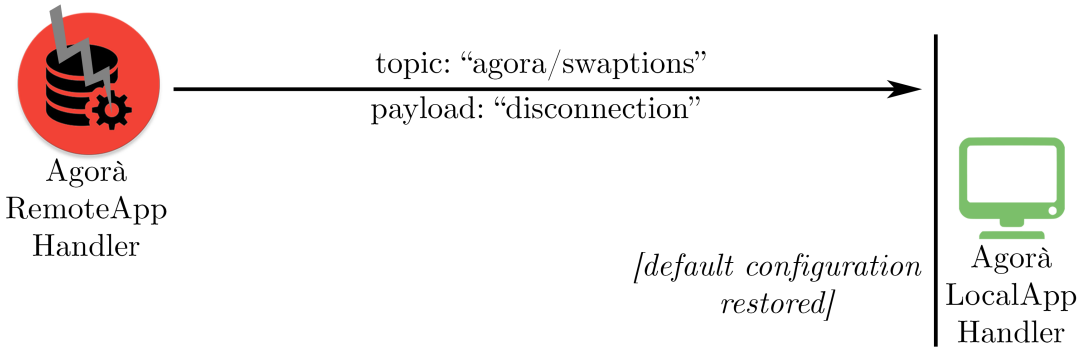


Figure 5.16: *AgoràRemoteAppHandler module disconnection with AgoràLocalAppHandler internal state equal to DSE example*

5.2.7.3 AgoràLocalAppHandler internal state equal to *DoEModel*

AgoràLocalAppHandler has received a partial OP list, related to Design of Experiments configurations (see 5.2.3.5); in this case, available OP list is not deleted, therefore the autotuner continues to work with this information.

5.2.7.4 AgoràLocalAppHandler internal state equal to *autotuning*

The AgoràLocalAppHandler module has already received predicted complete OP list from AgoràRemoteAppHandler, therefore nothing changes.

5.3 AgoràLocalAppHandler module integration

```
1  #include "agora_margot_manager.hpp"
2
3  int param1;
4  int param2;
5  int param3;
6
7  int main()
8  {
9    agora_margot_manager tmm;
10   tmm.init();
11
12   while( loop_condition() )
13   {
14       tmm.update_OPs();
15
16       margot::margot_block
17       {
18           do_computation( param1, param2, param3 );
19       }
20
21       tmm.sendResult( { param1, param2, param3 },
22                       { margot::margot_block::metric1, margot::margot_block::metric2 } );
23       }
24 }
```

Figure 5.17: *Sketch application with AgoràLocalAppHandler module plus mARGOt autotuner integration*

Figure 5.17 shows a sketch application that, until *loop_condition()* is verified (line 12), is executed; computation depends on three parameters ($param1, param2, param3$) that are set up by mARGOt autotuner at the beginning of every cycle (line 16), while two metrics of interest ($metric1, metric2$) are monitored (for mARGOt details, see related scientific publication [33]).

Integration code required to use Agorà framework with mARGOt autotuner is written in bold red; main steps during program execution are three:

1. AgoràLocalAppHandler module and mARGOt autotuner instanti-

ation and initialization (lines 9-10): the AgoràLocalAppHandler module saves all application information and sets up mARGOt autotuner with a default Operating Point; if nothing happens, the application is executed with this configuration;

2. Application knowledge update (line 14): the AgoràLocalAppHandler module updates, from time to time, its internal knowledge about application configurations that are sent by the AgoràRemoteAppHandler module; before mARGOt autotuner sets up application parameters (line 16), AgoràLocalAppHandler checks mARGOt knowledge with respect to its internal one: if they are different, mARGOt Operating Points are updated. In this way, for instance, if the AgoràLocalAppHandler module receives a new configuration during Design Space Exploration phase (see 5.2.3.4), mARGOt knowledge is set up with this data, so the application is forced to be executed with the corresponding parameter values; if, for instance, application complete model is received (see 5.2.3.6), mARGOt internal knowledge is set up with all this information, so the autotuner can choose the best Operating Point that fulfills application current goals and requirements;

3. Operating Point dispatch (line 21-22): after each computation has done, parameter values just used with the corresponding monitored metrics of interest are published on a predetermined MQTT topic, so the related AgoràRemoteAppHandler module can receive this information (see 5.2.5).

CHAPTER 6

Experimental results

In order to demonstrate Agorà validity, we do some tests on various applications and scenarios. Firstly, we present the structure of programs we use to do experiments; after that, we show test types and corresponding results.

6.1 Experimental setup

We test Agorà in three different scenarios: two versions of a synthetic application and a real one. This work has been coupled with mARGOt autotuner ([33]); from now on, we use the concept of Operating Point (OP) concerning application configurations in terms of parameter values and associated performance (observed metric of interest values).

Synthetic application has three parameters: $param_1$, $param_2$ and $param_3$; a function of these three variables calculates the amount of milliseconds of an execution cycle ($executionTime$ variable), while another function sets up an error measure; this last variable is considered as metric, together with application throughput as number of jobs per second (so, approximately equal to $\frac{1000}{executionTime}$).

For the real scenario, *Swaptions* application is used, taken from the PARSEC benchmark suite ([48]). This application solves partial differential equations through Monte Carlo simulations in order to price a portfolio of swaptions; its tunable parameters are: the number of threads (variable $num\_threads$, from 1 to 8) and the number of trials for the simulation at every cycle (variable num_trials, from 100.000 to 1.000.000 with a step of 100.000); observed metrics of interest are: throughput (variable $avg\_throughput$) as the number of priced swaptions per second and error (variable avg_error), computed as:

$$avg\_error = \frac{\sum_{s \in pricedSwaptions} |StandDevRef(s) - StandDev(s)|}{|pricedSwaptions|}$$

where $StandDevRef(s)$ is the reference standard deviation for swaption s, $StandDev(s)$ is the evaluated one and $pricedSwaptions$ represents the set of swaptions that are priced at each computing cycle; so, metric avg_error stands for the average of differences between standard deviation of priced swaptions using evaluated configuration with respect to the reference one (standard deviation for 1.000.000 trials).

Concerning used machine, we test Agorà on a Dell XPS 15 9550 with 4 core / 8 threads Intel(R) Core(TM) i7-6700HQ CPU @ 2.60 processor.

6.1.1 Synthetic application version 1

In the first version of synthetic application, the amount of execution time is calculated as:

$$executionTime = 7.35 \cdot \ln(param_1) + 38.1 \cdot param_2 + 52.96 \cdot \sqrt{param_3} + noise$$

where $noise$ simulates a disturbance; it is computed as:

$$noise = executionTime \cdot randomNumber \cdot noisePercentage$$

$randomNumber$ is a generated random value according to an exponential distribution with mean 0.3; $noisePercentage$ affects noise weight and it can be equal to $1\%, 5\%, 10\%, 15\%, 25\%$ or 50%.

Error metric is calculated as:

$$error = \frac{1}{0.015 \cdot \sqrt{param_1} + 0.033 \cdot \ln(param_2) + 0.028 \cdot \ln(param_3)}$$

Application parameters can assume these values:

Parameters	Values
$param_1$	1, 50, 150, 300, 450, 700, 800
$param_2$	1, 50, 100, 150, 200
$param_3(\cdot 10)$	1, 8, 15, 22, 29, 36, 43, 50, 57, 64, 71, 78, 85

This application has 455 Operating Points; figure 6.1 shows complete OP distribution for $noisePercentage = 15\%$: every point represents a particular configuration with an error value (on x-axis) and a throughput one (on y-axis). Since $noise$ in $executionTime$ is affected by $randomNumber$, this plot consider its expected value, equal to the mean of the chosen exponential distribution (0.3); the obtained OP list represents the reference model for this predetermined $noisePercentage$ value; of course, other reference models are similar, since

the only difference is the $noisePercentage$ value for the calculation of $executionTime$ variable.

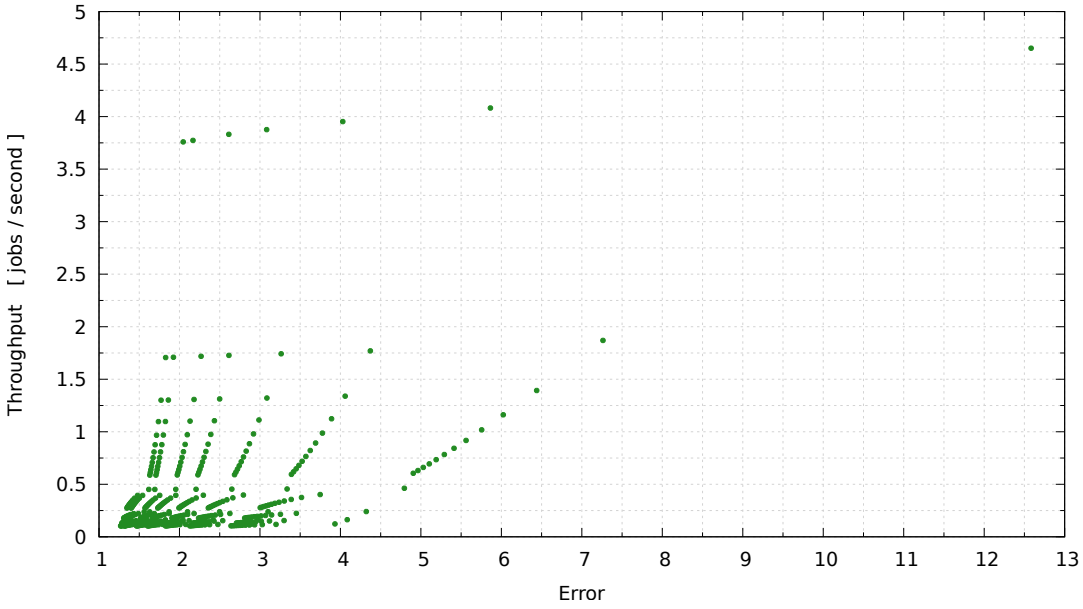


Figure 6.1: *Complete OP distribution of synthtetic application version 1 (reference model with $noisePercentage = 15\%$); every point stands for a particular configuration with associated observed metric values; on x-axis: error metric values, on y-axis: throughput metric values [jobs / second]*

Concerning model prediction quality, therefore, for every application setting with a fixed $noisePercentage$, estimated OP list is compared with the corresponding reference one.

6.1.2 Synthetic application version 2

In the second synthetic application version, execution time is equal to:

$$executionTime = 7.4 \cdot param_1 \cdot param_2 + 2.1 \cdot (param_3)^2 + noise$$

where $noise$ is simulated as in the previous application version, while the error is:

$$error = \frac{1}{0.01 \cdot param_1 + 0.7 \cdot \ln(param_2) + 0.019 \cdot param_3}$$

Parameter values are:

Parameters	Values
$param_1$	1, 10, 15, 25, 40, 65, 80
$param_2$	1, 5, 10, 20
$param_3$	10, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40, 43, 46

In this application the number of Operating Points is 364; figure 6.2 shows complete OP distribution of reference model with $noisePercentage = 5\%$:

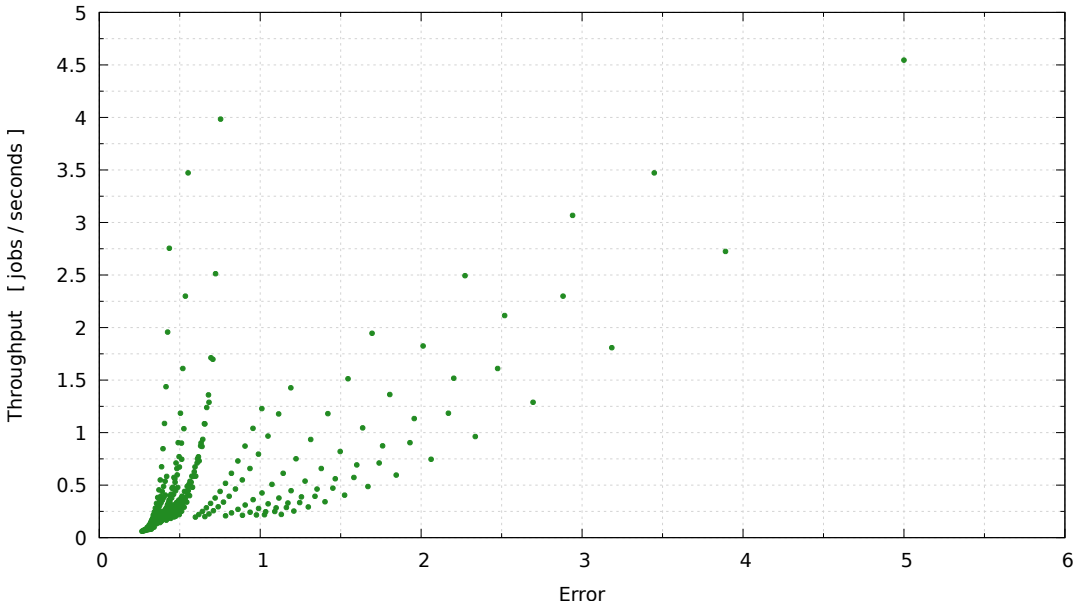


Figure 6.2: *Complete OP distribution of synthtetic application version 2 (reference model with $noisePercentage = 5\%$); every point stands for a particular configuration with associated observed metric values; on x-axis: error metric values, on y-axis: throughput metric values [jobs / second]*

Of course, regarding model prediction goodness, same reasoning as previous application version is applied.

6.1.3 Swaptions

As already stated in 6.1, this application has two parameters: $num\_threads$ and num_trials; their values are:

Parameters	Values
$num\_threads$	1, 2, 3, 4, 5, 6, 7, 8
$num\_trials(\cdot 10^4)$	10, 20, 30, 40, 50, 60, 70, 80, 90, 100

The number of Operating Points is therefore 80; figure 6.3 shows complete OP distribution with respect avg_error and $avg\_throughput$ metrics of interest:

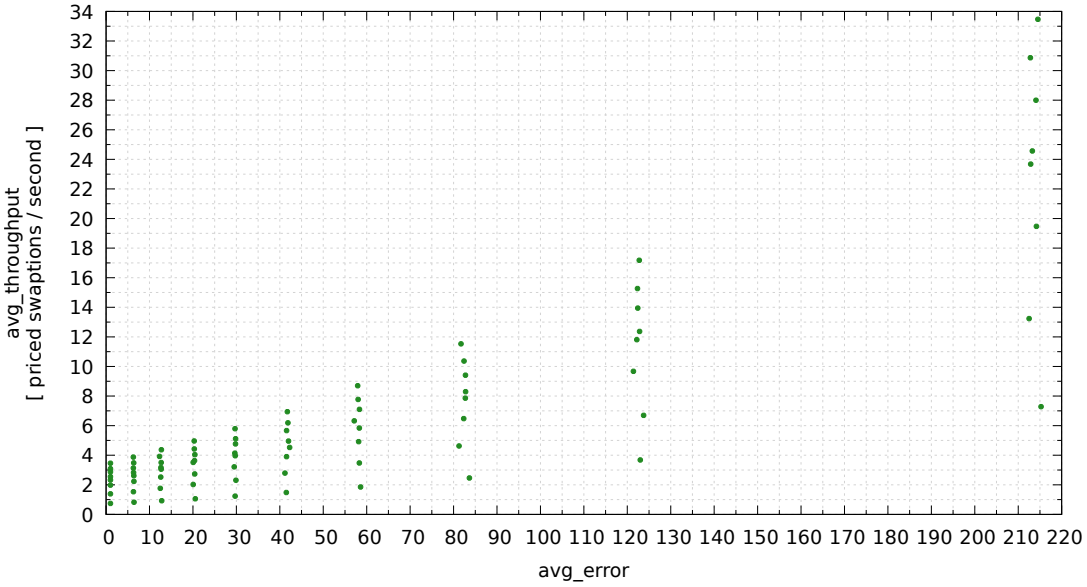


Figure 6.3: *Swaptions complete OP distribution; every point stands for a particular configuration with associated observed metric values; on x-axis: avg_error metric values, on y-axis: avg_throughput metric values [priced swaptions / second]*

6.2 Experimental campaign

We want to demonstrate Agorà validity; for both synthetic application versions and *Swaptions*, we focuse our attention on model prediction goodness in various scenarios; we study execution times, especially for Design Space Exploration phases with one and more

than one executing applications at the same time, highlighting benefits in sharing DSE; we reveal application behavior during execution on different cases; next paragraphs show results, firstly for synthetic applications, finally for the real one.

6.2.1 Synthetic application

In this paragraph we show experimental results for both versions of synthetic application, divided as explained before.

6.2.1.1 Model prediction quality

Concerning model prediction goodness, both synthetic application versions are executed several times with the introduction of all noise weights ($1\%, 5\%, 10\%, 15\%, 25\%$ and 50%), focusing on the prediction of throughput metric, that is the one affected by $noisePercentage$ value; a Face Centered Central Composite Desing of Experiments with one Center Point has been used, firstly gathering 1 repetition, then 5 repetitions and finally 10 repetitions for each DoE configuration during Design Space Exploration phase; for each predicted metric value m for each application setting, we have calculated a $deltaError$ measure:

$$deltaError(m) = \frac{|predictedValue(m) - referenceValue(m)| \cdot 100}{|referenceValue(m)|}$$

where $predictedValue(m)$ and $referenceValue(m)$ have explicit meaning; so, $deltaError(m)$ stands for how much the predicted metric value m distances itself from the corresponding value of the related reference model, in percentage; next figures summarize results:

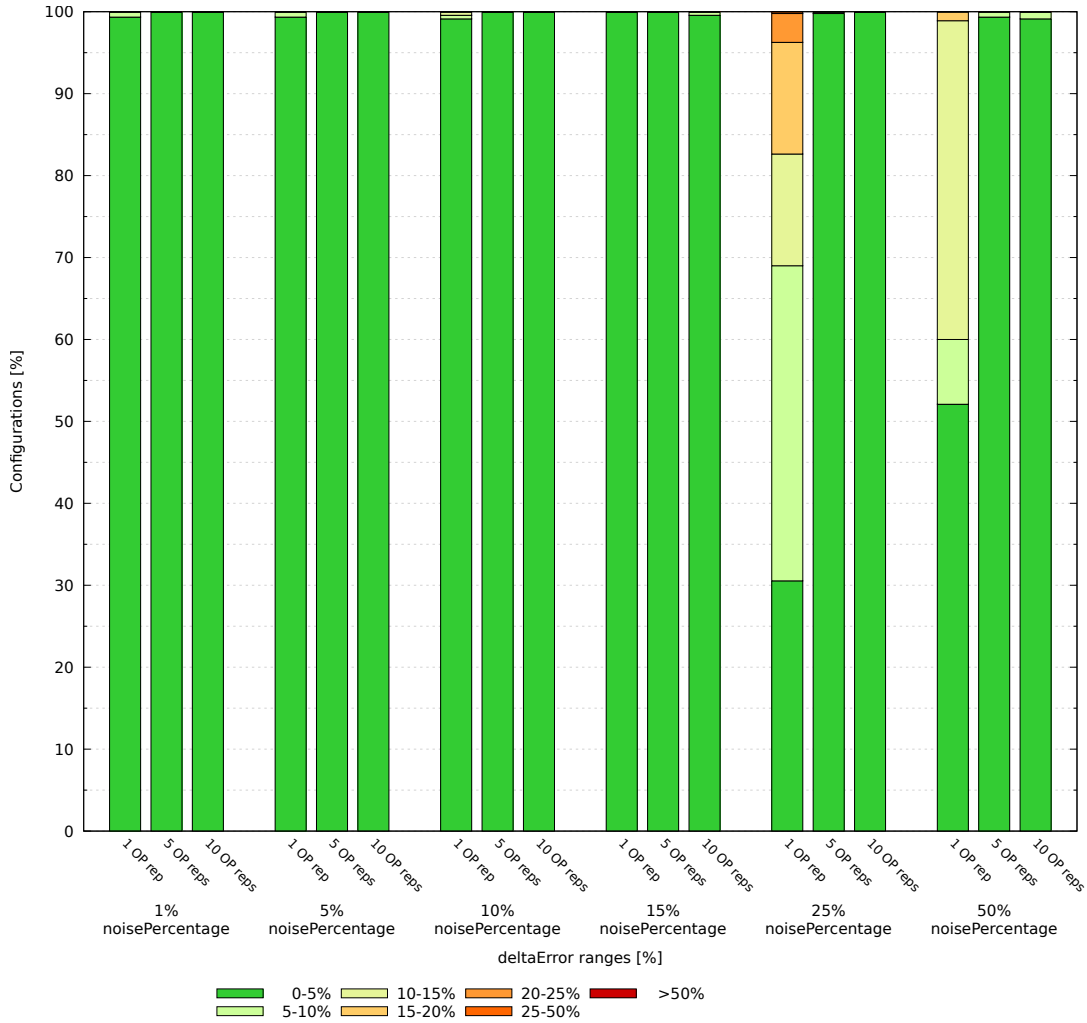


Figure 6.4: *$deltaError$ results for throughput metric of synthetic application version 1; used RSM: 1st version of implemented GLR ("transformations by functions", see 2.6.2); each stacked bar refers to a particular application configuration, with respect to $noisePercentage$ value and number of collected OP repetitions during DSE phase, showing OP percentages that have a corresponding throughput $deltaError$ within prearranged intervals; on y-axis: number of OPs [%]*

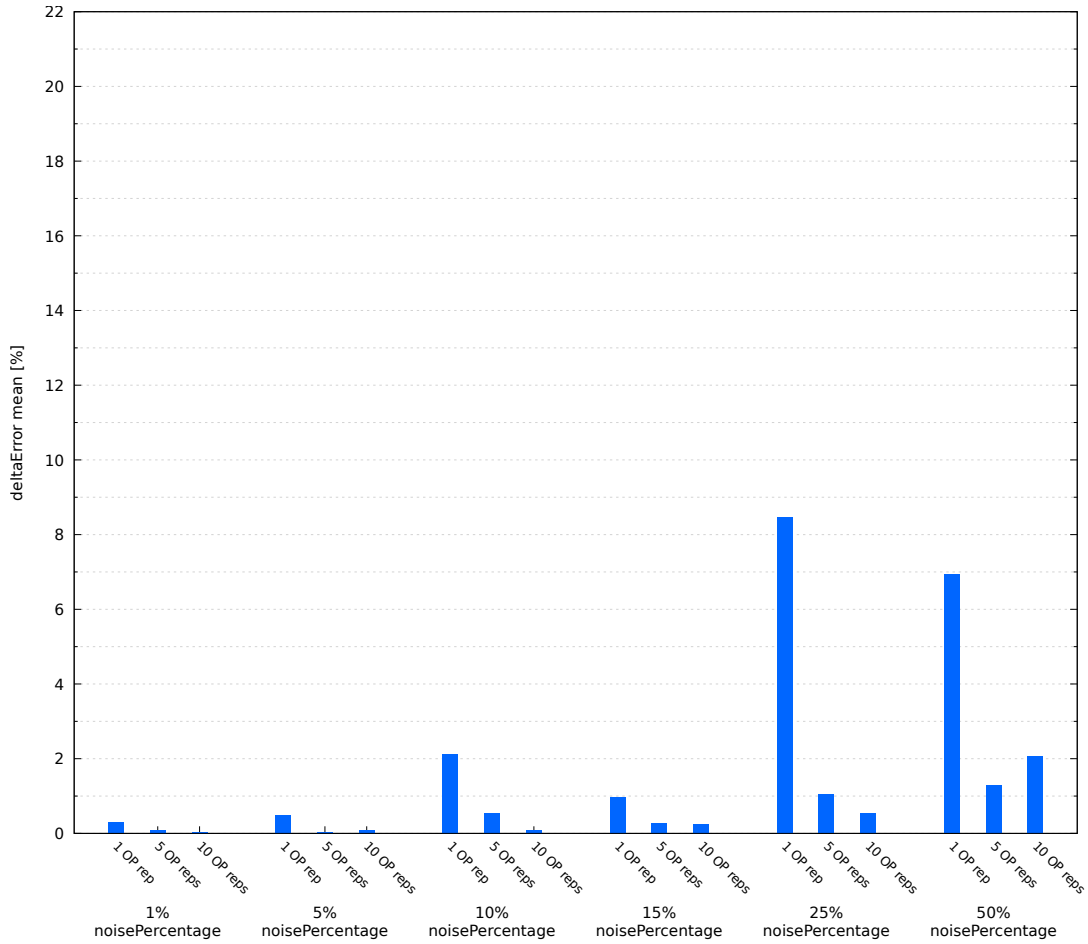


Figure 6.5: *$deltaError$ mean values for throughput metric of synthetic application version 1; used RSM: 1st version of implemented GLR ("transformations by functions", see 2.6.2); each bar shows throughput $deltaError$ mean for a particular application configuration, with respect to $noisePercentage$ value and number of collected OP repetitions during DSE phase; on y-axis: $deltaError$ mean [%]*

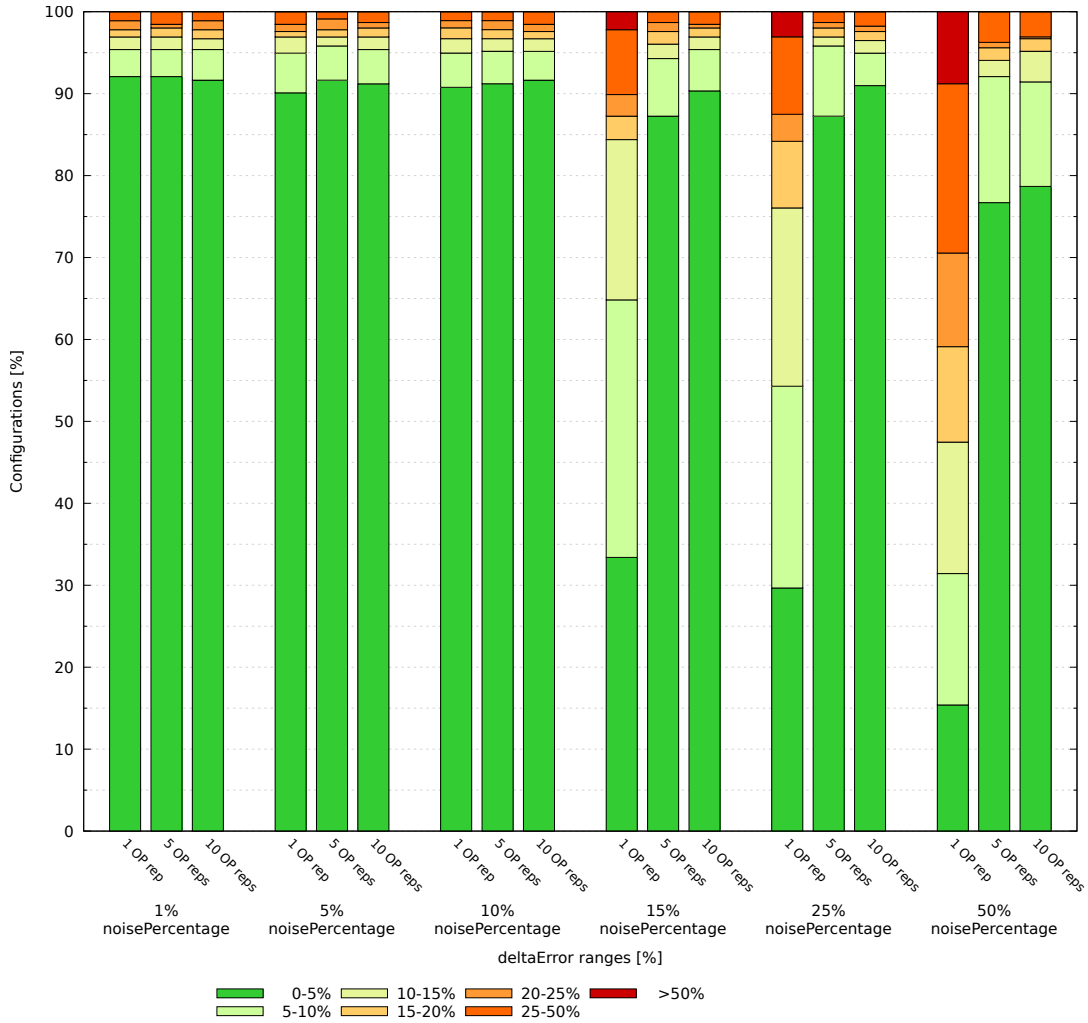


Figure 6.6: *$deltaError$ results for throughput metric of synthetic application version 1; used RSM: 2nd version of implemented GLR ("polynomial expansion of second order", see 2.6.2); each stacked bar refers to a particular application configuration, with respect to $noisePercentage$ value and number of collected OP repetitions during DSE phase, showing OP percentages that have a corresponding throughput $deltaError$ within prearranged intervals; on y-axis: number of OPs [%]*

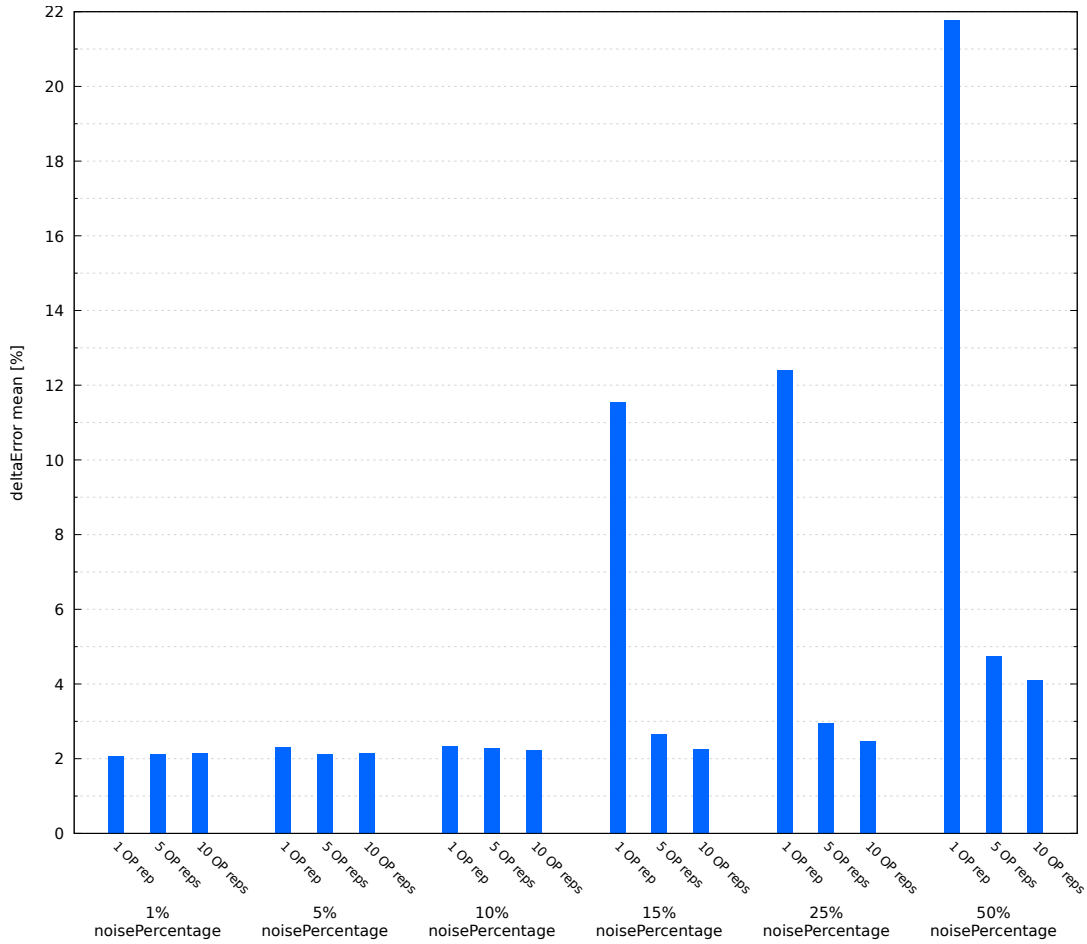


Figure 6.7: *$deltaError$ mean values for throughput metric of synthetic application version 1; used RSM: 2nd version of implemented GLR ("polynomial expansion of second order", see 2.6.2); each bar shows throughput $deltaError$ mean for a particular application configuration, with respect to $noisePercentage$ value and number of collected OP repetitions during DSE phase; on y-axis: $deltaError$ mean [%]*

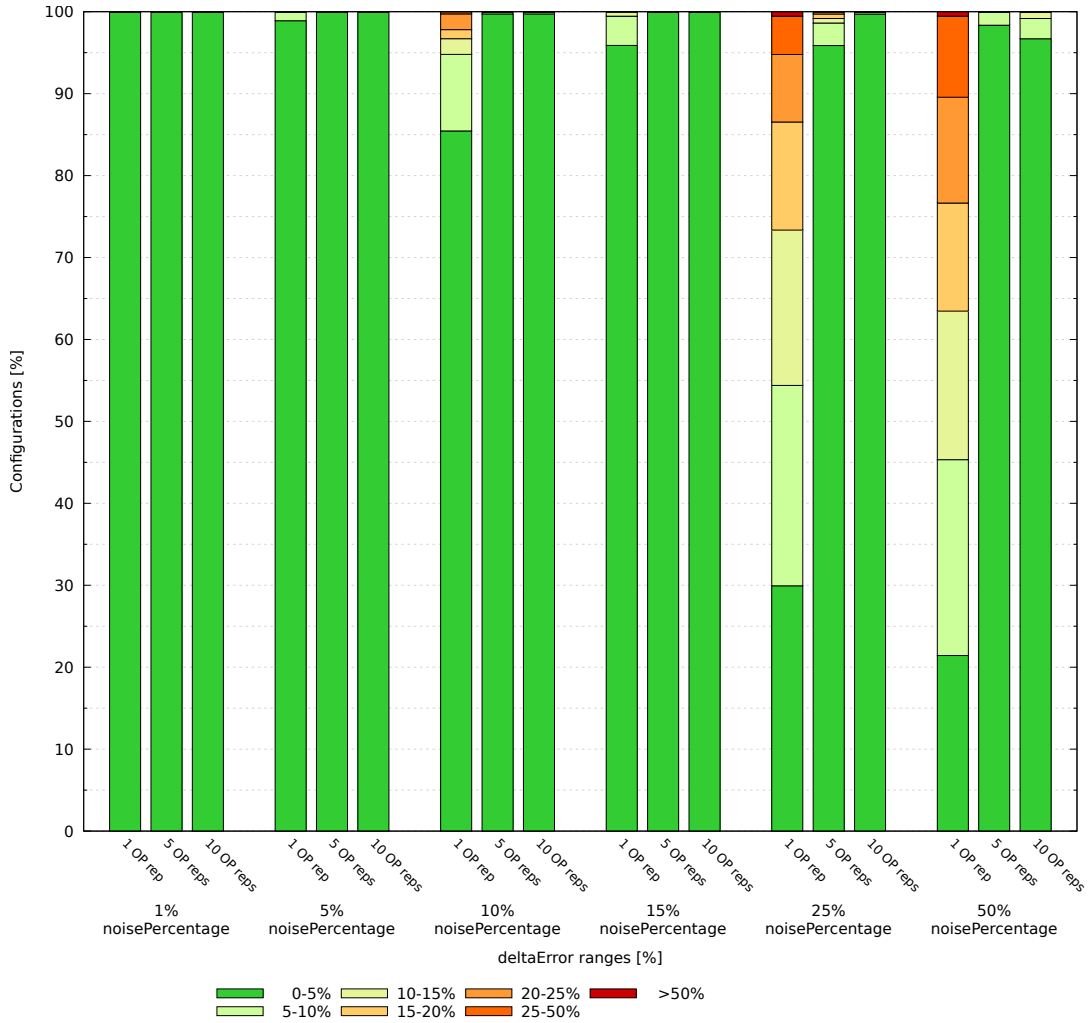


Figure 6.8: *$deltaError$ results for throughput metric of synthetic application version 2; used RSM: 2nd version of implemented GLR ("polynomial expansion of second order", see 2.6.2); each stacked bar refers to a particular application configuration, with respect to $noisePercentage$ value and number of collected OP repetitions during DSE phase, showing OP percentages that have a corresponding throughput $deltaError$ within prearranged intervals; on y-axis: number of OPs [%]*

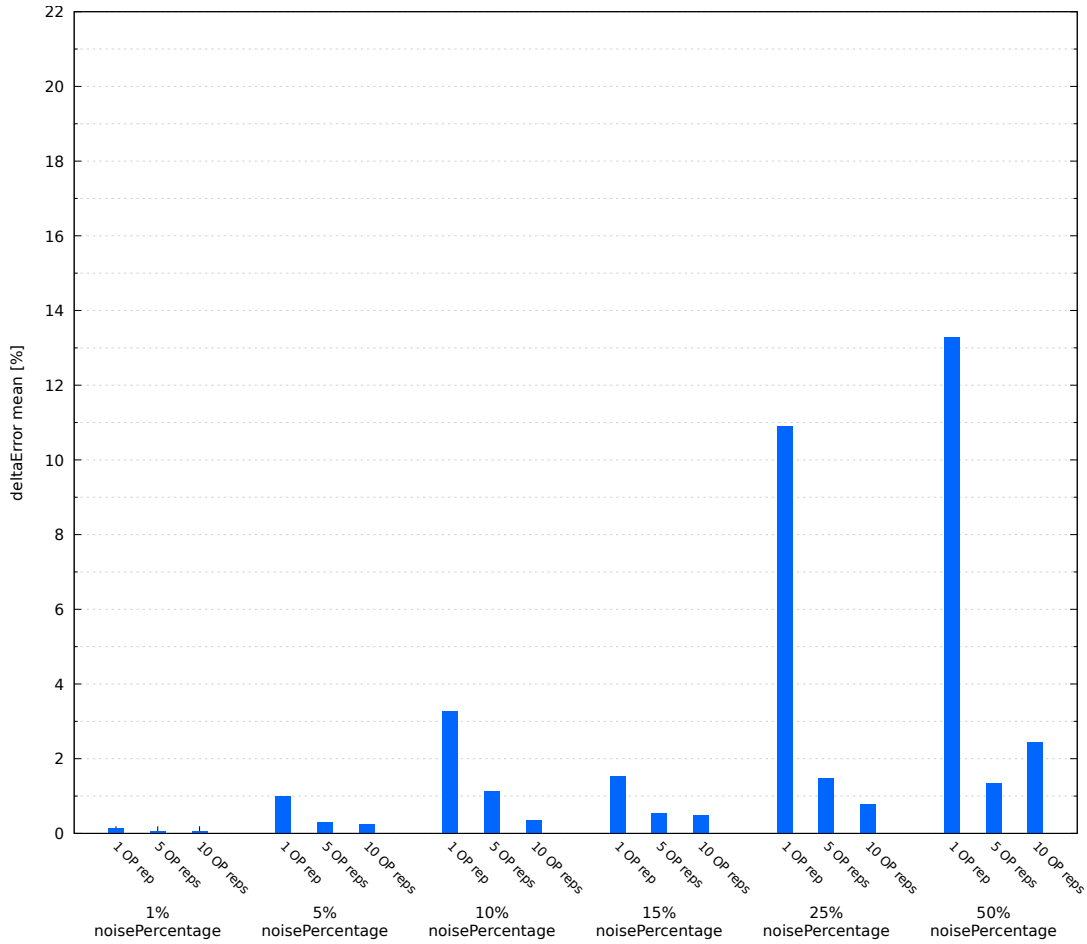


Figure 6.9: *$deltaError$ mean values for throughput metric of synthetic application version 2; used RSM: 2nd version of implemented GLR ("polynomial expansion of second order", see 2.6.2); each bar shows throughput $deltaError$ mean for a particular application configuration, with respect to $noisePercentage$ value and number of collected OP repetitions during DSE phase; on y-axis: $deltaError$ mean [%]*

In figures 6.4, 6.6 and 6.8, every stacked bar chart represent an application setting with respect to $noisePercentage$ value and the number of repetitions, for each DoE configuration, collected during DSE phase; on y-axis there are the number of predicted OPs, in percentage, grouped with respect to measured $deltaError$ of related throughput metric; related figures 6.5, 6.7 and 6.9 show $deltaError$ mean for each program setting.

From figure 6.4 we can see that, until $noisePercentage = 15\%$, model

prediction is really precise even with only 1 repetition for each DoE configuration: almost the totality of OPs have a $deltaError$ below 5%; with 5 repetitions and 10 repetitions, all configurations have a $deltaError$ below 5%. With the introduction of a strong noise weight, 25% and 50%, prediction get worse with 1 OP repetition, even if, respectively, around 70% and 60% of configurations remains with a *deltaError* below 10%; prediction gets back really accurate with 5 and 10 OP repetitions also with these high noise values.

Figure 6.6 shows that, for synthetic application 1, 2nd version of the implemented GLR produces, in general, results less precise that the ones generated with the 1st GLR version: up to $noisePercentage = 10\%$, quality of predicted models is very satisfying, where more than 90% of OPs have a $deltaError$ below 5%; from noise weight equal to 15%, 25%, 50% and with 1 OP repetition, this $deltaError$ percentage decreases to 35%, 30% and 15% respectively and a visible number of Operating Points has a $deltaError$ greater that 25%; for these settings, model prediction quality remarkably increases collecting more training data for each Design of Experiments configuration: with 5 and 10 OP repetitions we notice very few predicted OPs with a $deltaError$ greater that 10%, so prediction quality becomes very accurate even in these awful cases.

Concerning synthetic application version 2 with the 2nd version of implemented Generalized Linear Regression as RSM (figure 6.8), results are very similar to the ones related to synthetic application version 1 with 1st version of implemented Generalized Linear Regression (figure 6.4): quality of model prediction is generally very high, even with heavy noises but, in these cases, there is the need of more OP repetitions in order to mitigate this strong disturbance.

For synthetic application version 1, as it can be understood, 1st implemented version of GLR works slightly better than the 2nd one, but their prediction times are very different, as shown in table 6.1:

	1 OP rep	5 OP reps	10 OP reps
GLR 1st version	155.52 sec	155.7 sec	157.72 sec
GLR 2nd version	23.21 sec	23.64 sec	23.8 sec

Table 6.1: *Model prediction times for each implemented Generalized Linear Regression for each number of OP repetitions scenario*

2nd GLR version takes less than 24 seconds to predict complete model, while 1st GLR version approximately 156 seconds: the former, almost to the same level of quality, is more than 6 times faster than the latter. Moreover, we have noticed that the number of OP repetitions affects very poorly model prediction time: from table 6.1 we can understand that, among 1, 5 and 10 OP repetitions for each DoE configuration case, times vary very few.

We have not disclosed model prediction quality for synthetic application version 2 with the 1st implemented version of GLR: "*transformations by functions*" strategy is not able to capture functions of second order behavior, as variable $executionTime$ is built in this case. On the contrary, "*polynomial expansion of second order*" technique has the capability to highly predict functions with, for instance, logarithmic or square root values, as demonstrated in the second model prediction analysis (figure 6.6).

From last analyses, we can assert that Generalized Linear Regression with "*polynomial expansion of second order*" is a Response Surface Methodology much more powerful than GLR that uses "*transformations by functions*": the latter version can predict a restricted set of scenarios, while the former is able to include those cases and the wide variety of quadratic functions. Agorà has been able to predict very well these metric behaviors and to properly manage strong noise disturbances, with the necessity to do, in some cases, a longer Design Space Exploration phase, collecting more OPs for each Design of Experiments configuration; lastly, figures 6.5, 6.7 and 6.9 recap all various scenarios, showing $deltaError$ mean values that increase with

high noises, going back still very low with 5 and 10 OP repetitions, even in the worst cases.

6.2.1.2 Execution times

Concerning execution times, we want to show the significant benefit of sharing Design Space Exploration phase among several nodes, running the same application: the 2nd version of synthetic application with $noisePercentage = 10\%$ is executed, using Face Centered Central Composite Desing of Experiments with one Center Point and collecting, for each configuration, 20 repetitions; for this application setting, the total number of configurations to be explored is 300. We accomplish this DSE phase with 1 up to 10 executing applications, collecting the overall executing time, represented by each bar chart in figure 6.10:

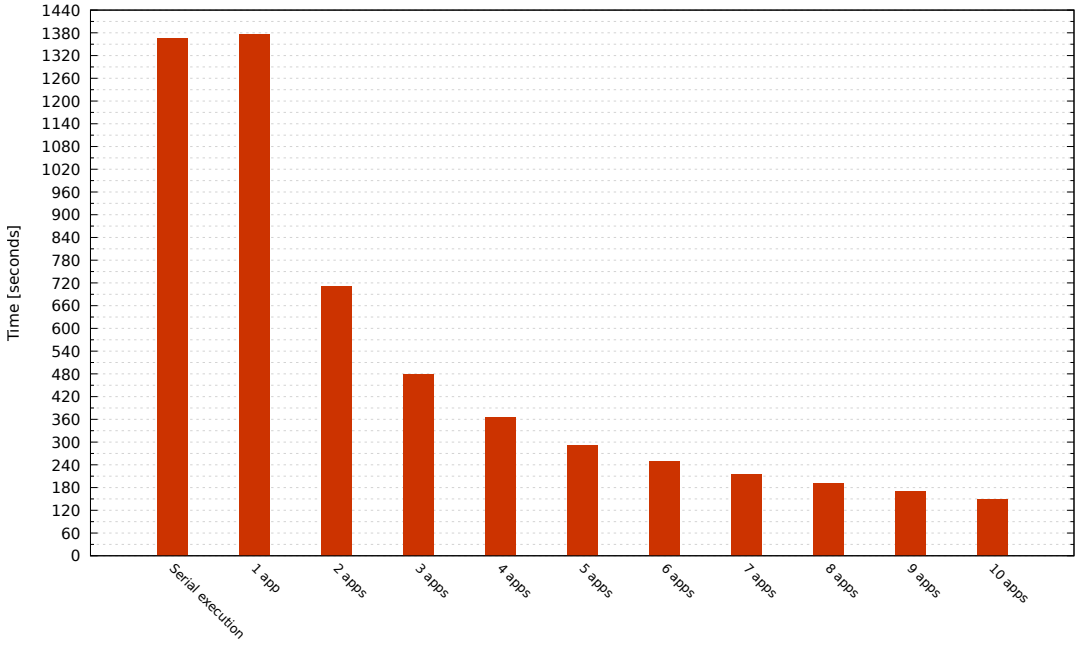


Figure 6.10: *Design Space Exploration times for several numbers of executing applications; first bar is related to DSE carried out exploring each configuration in a serial way, without Agorà; other bars are related to DSE phase done with implemented framework, through 1 executing application up to 10 ones; on y-axis: elapsed time [seconds]*

From figure 6.10 we can see that Agorà takes about 23 minutes to terminate Design Space Exploration with 1 executing application (second bar chart). The first bar chart shows the amount of time needed to execute application with all the 300 configurations in a serial way, without Agorà; this execution lasts around 10 seconds less than the first analyzed scenario, due to the lack of Agorà overheads; as we can understand, they are in any case very little: they add only approximately 0.73% of time with respect of the serial execution. From 2 executing applications on, DSE time strongly decreases: Agorà takes around 12 minutes to finish this phase with just 2 executing applications, 8 minutes with 3 ones, up to only 2 and a half minutes with 10 ones, more than 9 times lower than DSE phase with 1 application. Sharing Design Space Exploration phase among more than one executing application clearly speeds up overall time, since collection of training data duration, that is much longer than model prediction one (reported in table 6.1), considerably goes down.

Now we want to show general Agorà advantages, comparing a Full-Factorial, so exhaustive, application execution with the one supervised by this work; in figure 6.11, first bar chart shows overall time needed to compute Full-Factorial run for synthetic application version 2 with $noisePercentage = 15\%$, analyzing every possible configuration one time; next stacked bar charts show overall time, divided by DSE phase and model prediction, for the same application setting using Agorà and collecting 1, 5 and 10 OP repetitions for each DoE configuration:

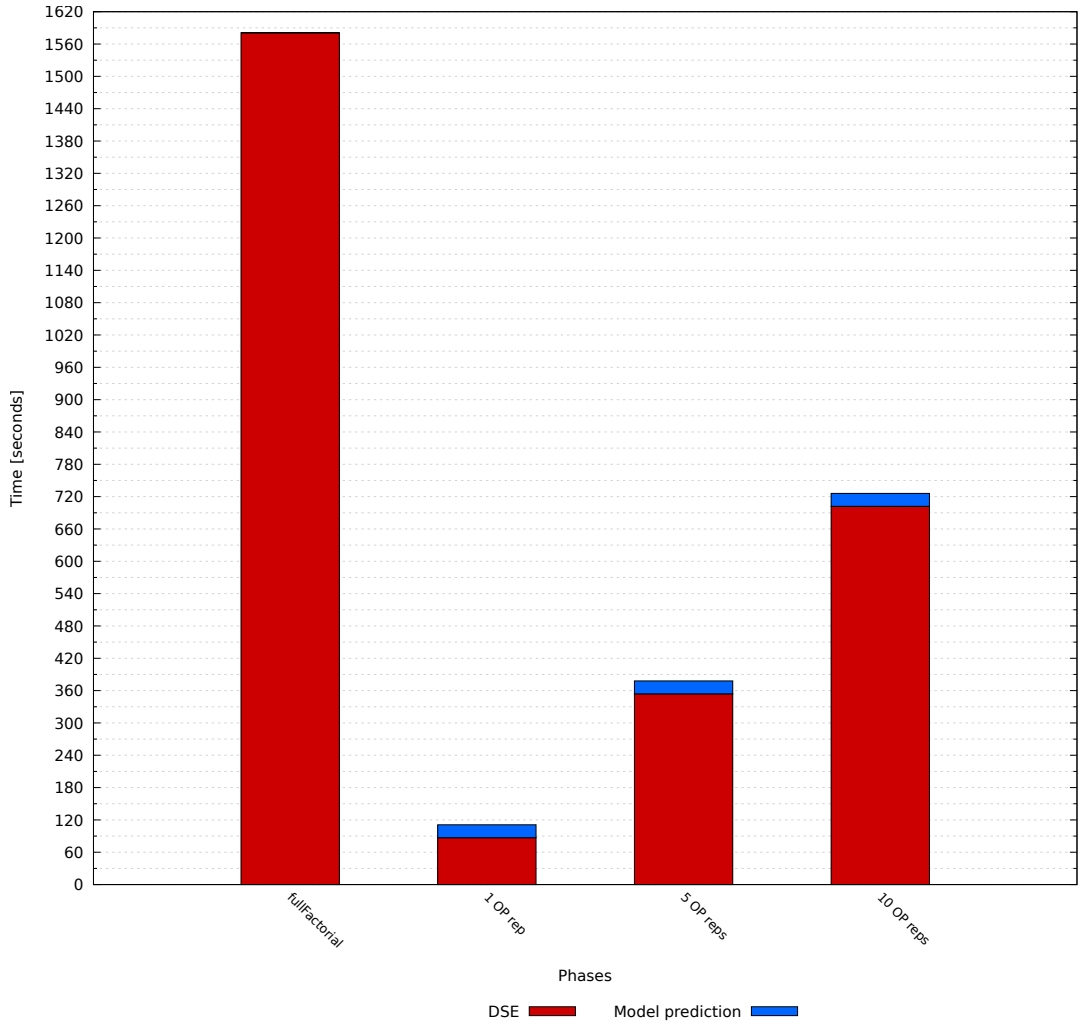


Figure 6.11: *Synthetic application version 2 (with $noisePercentage = 15\%$) execution times; first bar shows elapsed time for exhaustive execution; adjacent ones reveal overall time, until complete model is predicted, using 1, 5 and 10 DoE configuration repetitions during DSE phase; on y-axis: elapsed time [seconds]*

Figure 6.11 shows that an exhaustive execution of analyzed application takes more than 26 minutes to finish; with the introduction of Agorà, using Face Centered Central Composite Desing of Experiments with one Center Point and the 2nd version of implemented GLR as RSM, overall time becomes 12 minutes if 10 repetitions for each DoE configuration are collected, about 6 minutes with 5 repetitions and even around 2 minutes with 1 repetition; we specify that, if we want to explore every possible configuration in the exhaustive

analysis as much as we do for 5 and 10 OP repetitions of DoE configurations with Agorà, of course overall time is 130 minutes and 260 minutes respectively (5 times and 10 times the shown serial execution time), increasing much more the gap between Full-Factorial execution and the ones with the supervise of this work. Finally, we also want to remind, from figures 6.8 and 6.9, that Agorà, for this application setting, can predict a complete model with a very high quality even collecting just 1 repetition for each DoE configuration, so the final result is very similar to an exhaustive analysis. Therefore, this work can produce high benefits in the prediction of application complete model, starting from a small subset of configurations.

6.2.1.3 Application behavior over time

Last figure wants to show application behavior during execution with the supervise of Agorà and mARGOt autotuner; on x-axis there is time, while on y-axis there are observed metric of interest values during execution. We have run synthetic application version 2 with $noisePercentage = 15\%$, using Face Centered Central Composite Design of Experiments with one Center Point and the 2nd version of implemented GLR as RSM, collecting 5 repetitions for each Design of Experiments configuration during DSE phase; objective functions have been set up to $Throughput > 3$ and $Error < 1$; figure 6.12 shows results:

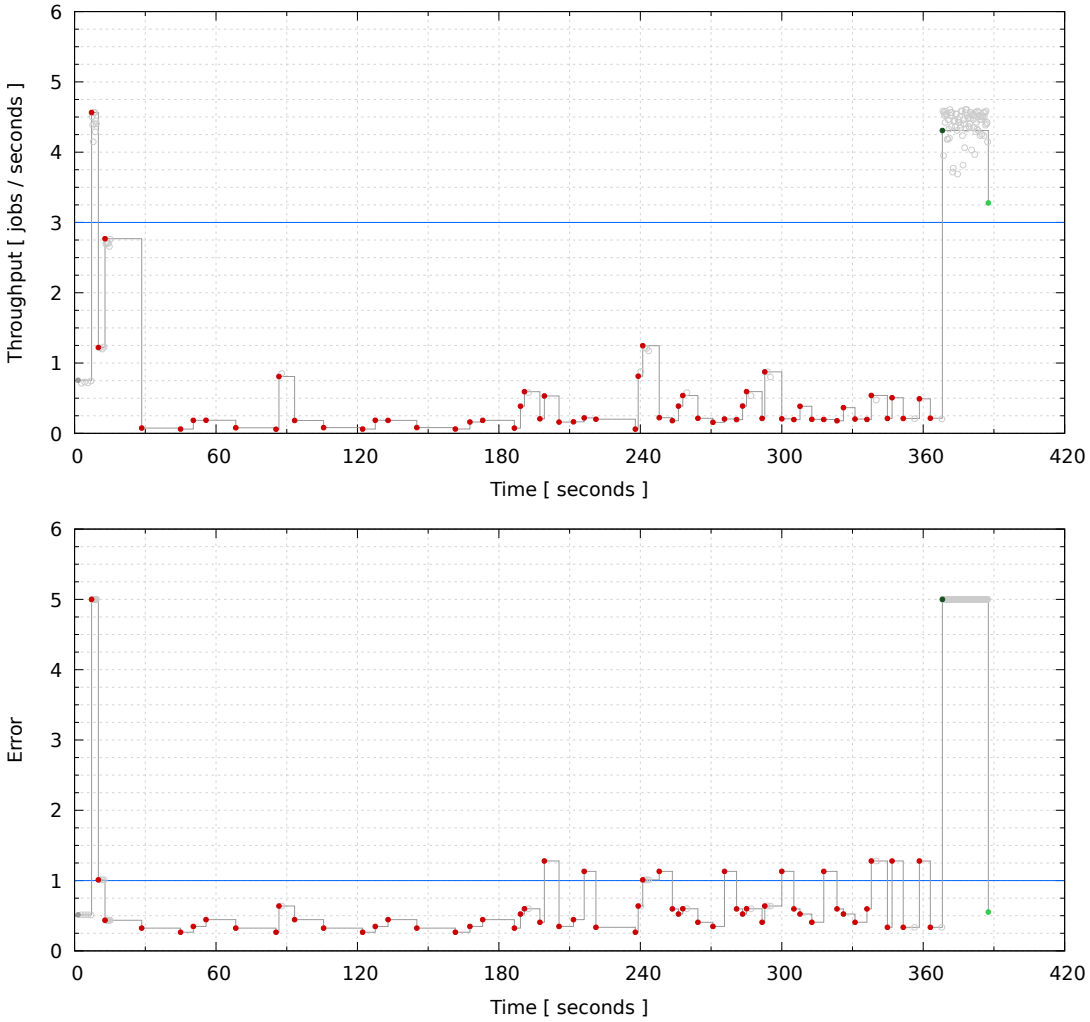


Figure 6.12: *Observed metric values during application execution; blue lines highlight boundaries of objective functions; on x-axis: time [seconds]; on y-axis: metric values*

Every couple of corresponding points in graphs of figure 6.12 represents a configuration with which the application has been executed, obtaining certain throughput and error metric values; from a colored couple of points to the following one, the application has maintained parameter values corresponding to the former couple; grey empty couples of points indicate application execution with parameter values equal to the first previous colored couple. At the beginning, application runs with default configuration, corresponding to an error of around 0.5 and a throughput equal to about 0.75 (grey

couple of points); Agorà starts Design Space Exploration, sending DoE configurations: mARGOt, from time to time, sets application parameters with these values, so observed metric of interest values vary during all this phase (red couples of points); when DSE phase terminates, Agorà sends a partial model (see 5.2.3.5) and it starts predicting the complete list of Operating Points: mARGOt chooses the best application configuration that fulfills objective functions, represented by dark green couple of points; throughput is around 4.25, so the first goal is respected, while error is about 5, so the second goal is not achieved; when application receives the complete model (around at 6 and a half minutes after starting time), it is set with another configuration (light green couple of points): obtained throughput is approximately 3.25 and error is about 0.5, so all objective functions are fulfilled; of course, if objective functions don't change, program keeps running with this Operating Point.

6.2.2 Swaptions application

We test Agorà on a real scenario; we use Face Centered Central Composite DoE with One Center Point as Design of Experiments and the 2nd version of implemented Generalized Linear Regression as Response Surface Methodology; due to application computing kernel (Monte Carlo simulations), we have decided to collect, for each DoE configuration, a meaningful number of repetitions, setting this value to 15.

6.2.2.1 Model prediction quality

Next figures show results on predicted model using $deltaError$ measure, as done for synthetic application analysis (see 6.2.1.1), for both avg_error and $avg\_throughput$ metrics:

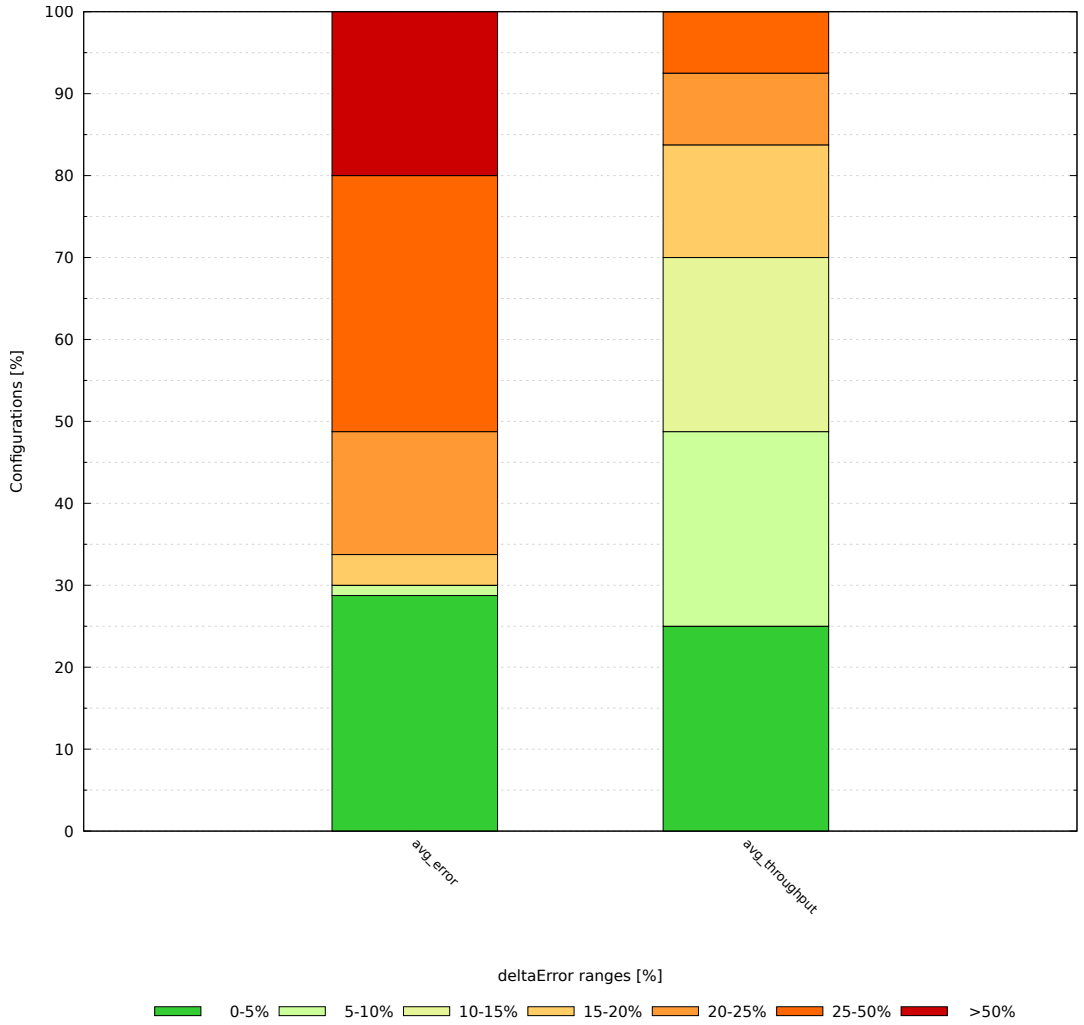


Figure 6.13: *$deltaError$ results for both Swaptions metrics of interest; each stacked bar shows OP percentages that have a corresponding metric $deltaError$ within prearranged intervals; left stacked bar focuses on avg_error metric, right one on $avg\_throughput$; on y-axis: number of configurations [%]*

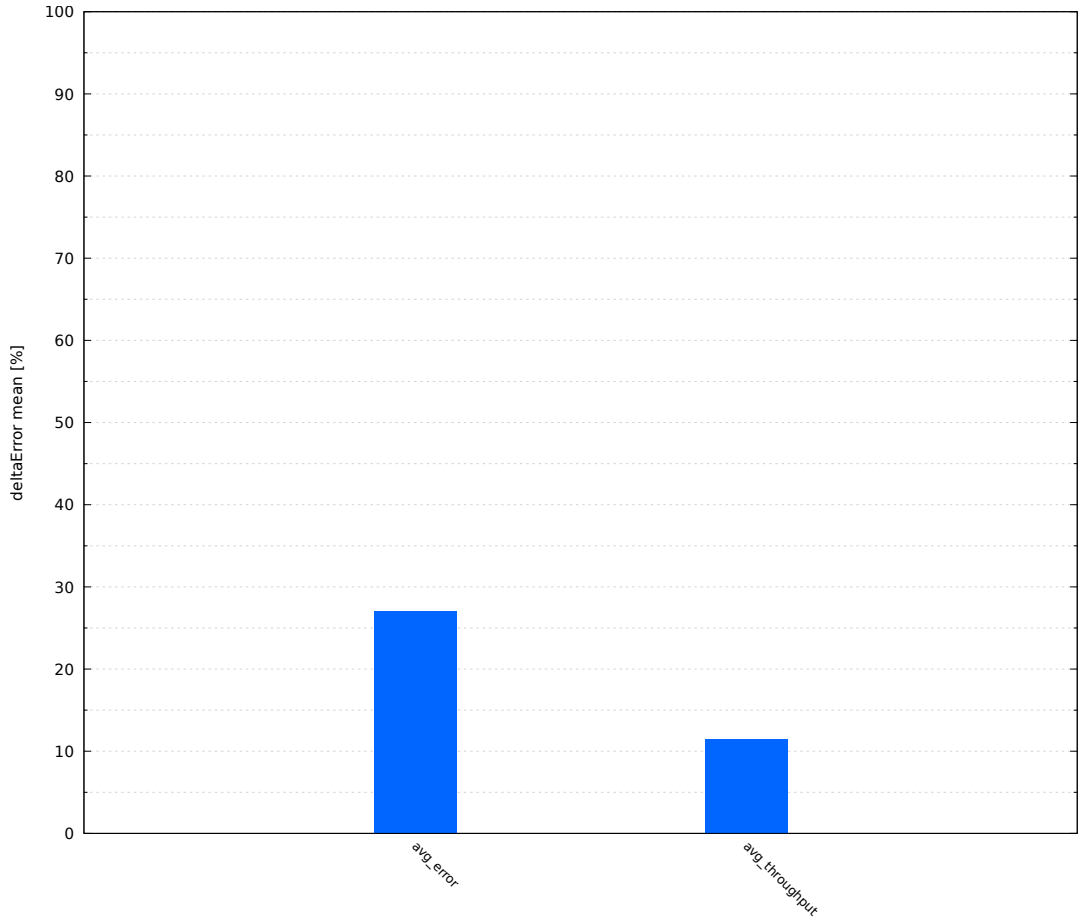


Figure 6.14: *$deltaError$ mean values for both Swaptions metrics of interest; left bar shows $deltaError$ mean for avg_error metric, right one for $avg\_throughput$; on y-axis: $deltaError$ mean [%]*

We obtain an average $deltaError$ of around 11% for $avg\_throughput$ and about 27% for avg_error, as it is shown in figure 6.14; from figure 6.13 we can see that 70% of $avg\_throughput$ predictions has a $deltaError$ below 15%, while this percentage increases to 30% for avg_error metric.

6.2.2.2 Execution times

Regarding execution times, we compare analyzed run with a Full-Factorial, so exhaustive, execution in which we collect, for each configuration, 15 repetitions for each Operating Point, as shown in figure

6.15:

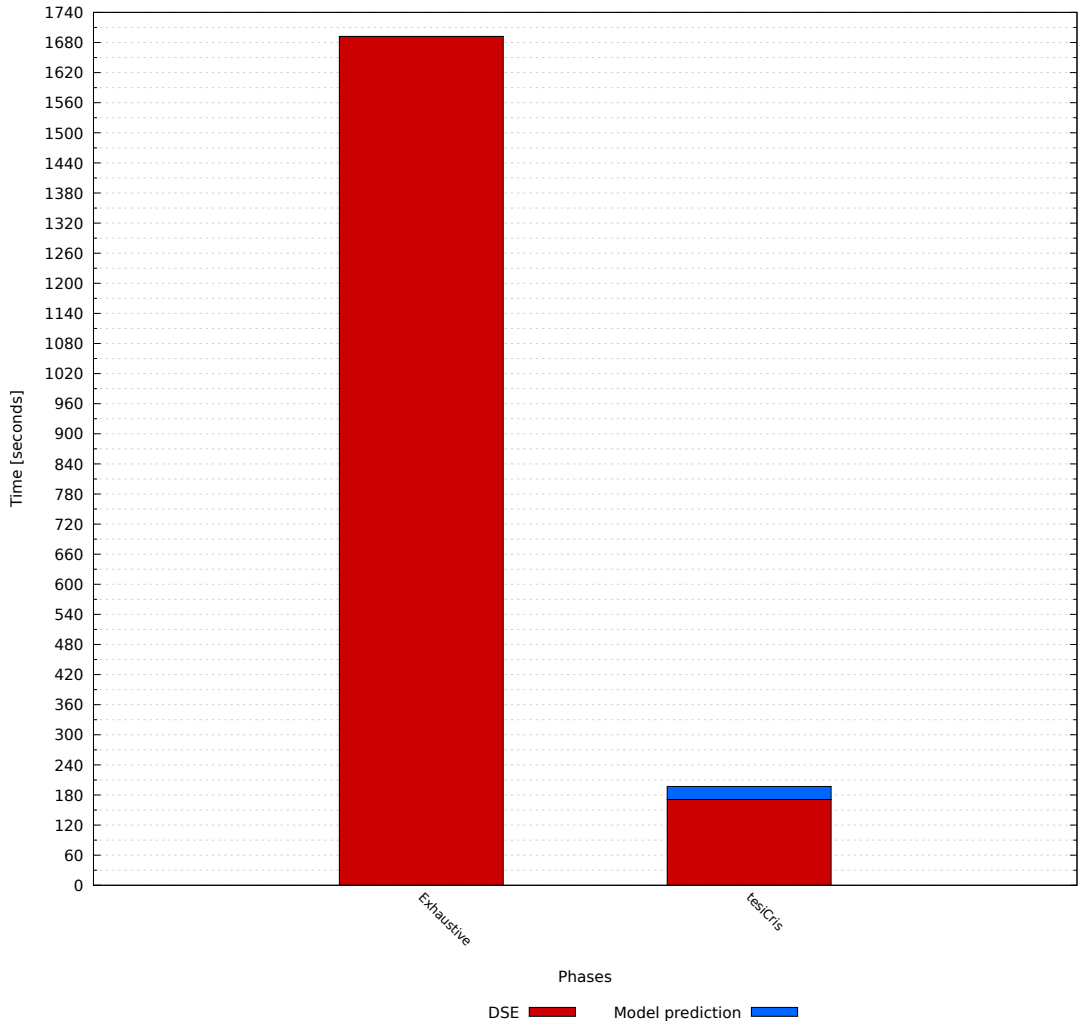


Figure 6.15: *Swaptions execution times; first bar shows elapsed time for exhaustive execution; adjacent one reveal overall time, until complete model has been predicted, with the supervise of Agorà; on y-axis: elapsed time [seconds]*

Exhaustive execution takes around 28 minutes to finish, while the analyzed one less than 3 and a half minutes, so approximately 8 times less; of course, as explained before, we do not obtain precise predictions for each metric value; nevertheless, *Swaptions* can be executed with the assistance of Agorà plus mARGOt autotuner and, finally, prearranged objective functions are achieved, also because mARGOt is able to collect feedback information that corrects application model.

6.2.2.3 Application behavior over time

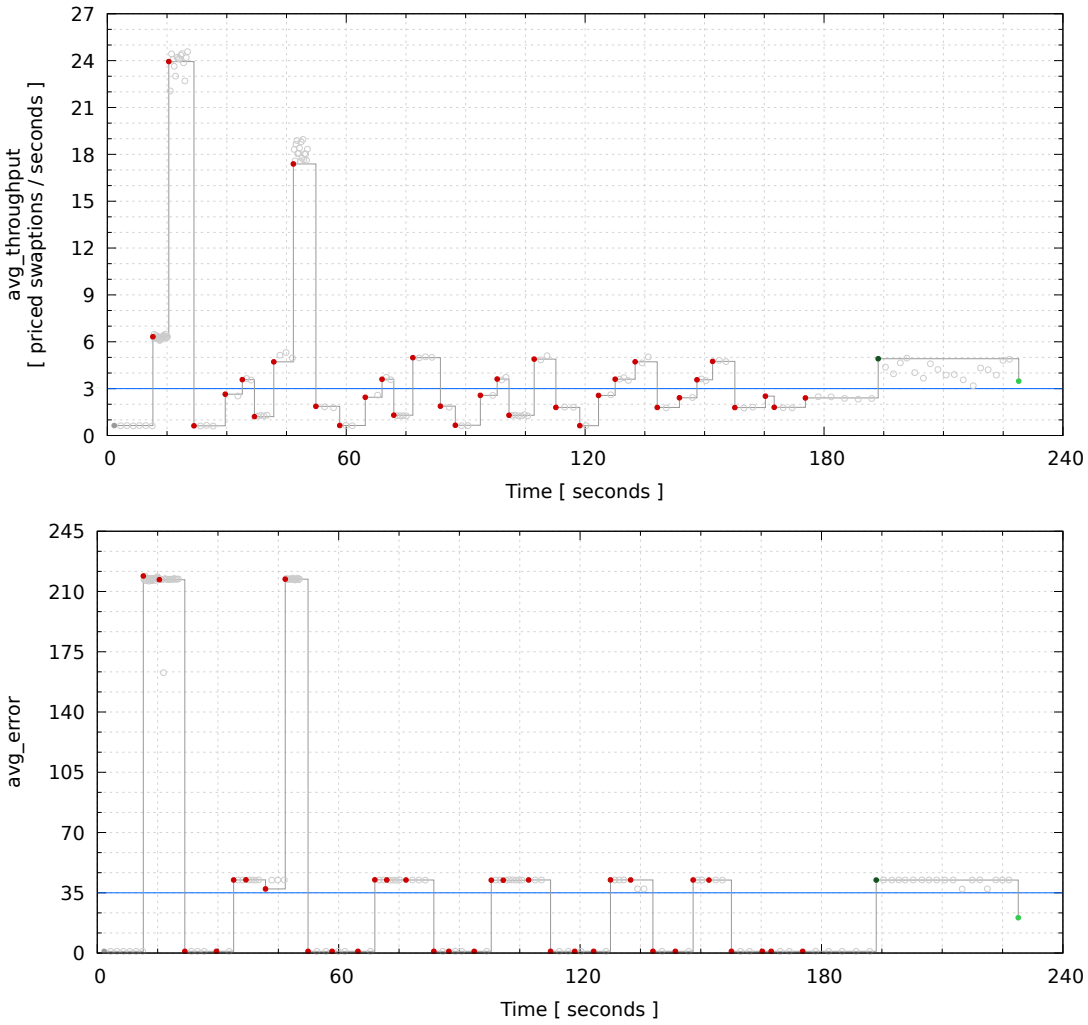


Figure 6.16: *Observed metric values during Swaptions execution; blue lines highlight boundaries of objective functions; on x-axis: time [seconds]; on y-axis: metric values*

Figure 6.16 shows application behavior during execution; we set, as objective functions, $avg\_throughput > 3$ and $avg\_error < 35$; during Design Space Exploration phase, application is executed with Design of Experiments configurations in order to collect training data (red couples of points); around 195 seconds, DSE phase finishes and application receives partial model (see 5.2.3.5); mARGOt sets up *Swaptions* with best possible parameter values with respect to goals and

requirements (dark green couple of points): during model prediction phase, $avg\_throughput$ is about 5 (so, greater than the minimum required value 3) and avg_error approximately 42 (higher than 35, so the corresponding objective function is not followed); Agorà sends complete model with predicted metric values for each application configuration: mARGOt changes current Operating Point (light green couple of points), obtaining an $avg\_throughput > 3$ and an $avg\_error < 35$, so both objective functions are achieved.

CHAPTER 7

Conclusion and future works

We have presented *Agorà*, a supporting framework that is able to provide predicted complete model of tunable applications, made by the whole list of parameter settings and related metric of interest performances. *Agorà* faces High Performance Computing application issues related to the impossibility to explore their Design Space in an exhaustive way, due to its significant dimension; main goal is to collect information about a small subset of possible application configurations and, through this data and Machine Learning techniques, to predict general program behavior in all possible paratemer value combinations.

Unique *Agorà* feature is the ability to remotely drive Design Space Exploration phase among same executing applications at runtime, distributing all configurations to be explored in an efficient way, there-

fore every running program contributes to data collection and the entire workflow is considerably sped up; we remark that analysis is not done locally in every node of a possible parallel architecture, but it is accomplished outside, not consuming, in this way, node computational capacity, except for communications among *Agorà* modules, fulfilled by the lightweight MQTT messaging protocol.

We show *Agorà* benefits, in terms of model prediction quality, execution times and ability to drive application executions in the experimental chapter (6.2), through multiple versions of a synthetic program with analytical metrics and a real scenario, using *Swaptions* application from the PARSEC benchmark suite ([48]); we demonstrate that in several scenarios, even introducing strong noises, we are able to achieve very high model prediction quality and, having the possibility to use multiple executing nodes to explore application Design Space, we demonstrate how overall time strongly decreases with respect to an individual analysis with a single application: at the best of our knowledge, this is the first attempt to share Design Space Exploration as supervised by *Agorà*.

There are, certainly, several elements of our work that can be improved; we indicate these aspects:

1. in addition to implemented Machine Learning strategies (two versions of Generalized Linear Regression, see 2.6.2), there can be added other techniques, in order to expand and to improve *Agorà* versatility; moreover, instead of entrusting to user the selection of Machine Learning strategy, the system could be able to analyze current problem and to choose itself best technique among the available ones in a proper way;

2. after application model prediction, *Agorà* could continue to collect feedback information about program execution, in order to eventually update model and send again revised complete list of

application configurations with related predicted performance in terms of metric of interest values;

3. there could be added the possibility to specify a limit duration of Design Space Exploration phase: when time is up, *AgoràRemoteAppHandler* module predicts application complete model with all information collected up to that moment;

4. instead of specifying a priori application lapse for request (see 5.2.3), *AgoràLocalAppHandler* module could dynamically adapt request frequency, according to previous execution times.

We hope *Agorà* can contribute to the research on Design Space Exploration and dynamic autotuning of High Performance Computing applications in parallel architectures.

Ringraziamenti

[TO DO]

Bibliography

[1] Henry Markram et al. “Introducing the human brain project”. In: *Procedia Computer Science* 7 (2011), pp. 39–42.

[2] Hadi Esmaeilzadeh et al. “Dark silicon and the end of multicore scaling”. In: *ACM SIGARCH Computer Architecture News*. Vol. 39. 3. ACM. 2011, pp. 365–376.

[3] Robert H Dennard et al. “Design of ion-implanted MOSFET’s with very small physical dimensions”. In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268.

[4] Gordon E Moore et al. “Cramming more components onto integrated circuits”. In: *Proceedings of the IEEE* 86.1 (1998), pp. 82–85.

[5] Jonathan Koomey et al. “Implications of historical trends in the electrical efficiency of computing”. In: *IEEE Annals of the History of Computing* 33.3 (2011), pp. 46–54.

[6] Balaji Subramaniam et al. “Trends in energy-efficient computing: A perspective from the Green500”. In: *Green Computing Conference (IGCC), 2013 International*. IEEE. 2013, pp. 1–8.

[7] Keren Bergman et al. “Exascale computing study: Technology challenges in achieving exascale systems”. In: *Defense Advanced Research*

Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep 15 (2008).

[8] *America's Data Centers Consuming and Wasting Growing Amounts of Energy*. 2015. url: https://www.nrdc.org/resources/americas-data-centers-consuming-and-wasting-growing-amounts-energy.

[9] John Gantz and David Reinsel. "Extracting value from chaos". In: *IDC iview* 1142.2011 (2011), pp. 1–12.

[10] *TOP500 Green Supercomputers*. 2017. url: https://www.top500.org/green500/.

[11] Sparsh Mittal. "Power management techniques for data centers: A survey". In: *arXiv preprint arXiv:1404.6681* (2014).

[12] Sparsh Mittal. "A survey of techniques for approximate computing". In: *ACM Computing Surveys (CSUR)* 48.4 (2016), p. 62.

[13] Jeffrey O Kephart and David M Chess. "The vision of autonomic computing". In: *Computer* 36.1 (2003), pp. 41–50.

[14] Cristina Silvano et al. "The ANTAREX approach to autotuning and adaptivity for energy efficient HPC systems". In: *Proceedings of the ACM International Conference on Computing Frontiers*. ACM. 2016, pp. 288–293.

[15] João MP Cardoso et al. "LARA: an aspect-oriented programming language for embedded systems". In: *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*. ACM. 2012, pp. 179–190.

[16] João MP Cardoso et al. "Performance-driven instrumentation and mapping strategies using the LARA aspect-oriented programming approach". In: *Software: Practice and Experience* (2014).

[17] Blaise Barney. "Introduction to Parallel Computing. Lawrence Livermore National Laboratory". In: *Available on https://computing.llnl.gov/tutorials/parallel_comp* (2012).

[18] Massimiliano Caramia and Paolo Dell´ Olmo. “Multi-objective optimization”. In: *Multi-objective Management in Freight Logistics: Increasing Capacity, Service Level and Safety with Optimization Algorithms* (2008), pp. 11–36.

[19] Mary Natrella. “e-handbook of statistical methods”. In: NIST/SEMATECH, 2013. Chap. 5.3, Choosing an experimental design. url: `http://www.itl.nist.gov/div898/handbook/pri/section3/pri3.htm`.

[20] Autonomic Computing et al. “An architectural blueprint for autonomic computing”. In: *IBM White Paper* 31 (2006).

[21] Andrew Banks and Rahul Gupta. “MQTT Version 3.1.1”. In: *OASIS standard* (2014).

[22] Lucy Zhang. *Building facebook messenger*. 2011.

[23] *HiveMQ - Enterprise MQTT Broker*. 2017. url: `http://www.hivemq.com/`.

[24] Apache Spark. *Apache Spark: Lightning-fast cluster computing*. 2015.

[25] *Learning from Data - Machine Learning course taught by Caltech Professor Yaser Abu-Mostafa*. 2012. url: `http://work.caltech.edu/telecourse.html`.

[26] Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. “ReSPIR: a response surface-based Pareto iterative refinement for application-specific design space exploration”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.12 (2009), pp. 1816–1829.

[27] André I Khuri and Siuli Mukhopadhyay. “Response surface methodology”. In: *Wiley Interdisciplinary Reviews: Computational Statistics* 2.2 (2010), pp. 128–149.

[28] Cristina Silvano et al. “Multicube: Multi-objective design space exploration of multi-core architectures”. In: *VLSI 2010 Annual Symposium*. Springer. 2011, pp. 47–63.

[29] Marcela Zuluaga, Andreas Krause, and Markus Püschel. "ϵ-PAL: An Active Learning Approach to the Multi-Objective Optimization Problem". In: *Journal of Machine Learning Research* 17 (2016), pp. 1–32.

[30] Jason Ansel et al. "Siblingrivalry: online autotuning through local competitions". In: *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*. ACM. 2012, pp. 91–100.

[31] Carlos A Coello Coello, Gary B Lamont, David A Van Veldhuizen, et al. *Evolutionary algorithms for solving multi-objective problems*. Vol. 5. Springer, 2007.

[32] Xin Sui et al. "Proactive control of approximate programs". In: *ACM SIGOPS Operating Systems Review* 50.2 (2016), pp. 607–621.

[33] Davide Gadioli, Gianluca Palermo, and Cristina Silvano. "Application autotuning to support runtime adaptivity in multicore architectures". In: *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*. IEEE. 2015, pp. 173–180.

[34] Patrick Bellasi, Giuseppe Massari, and William Fornaciari. "A RTRM proposal for multi/many-core platforms and reconfigurable applications". In: *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), 2012 7th International Workshop on*. IEEE. 2012, pp. 1–8.

[35] Jason Ansel et al. "Opentuner: An extensible framework for program autotuning". In: *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM. 2014, pp. 303–316.

[36] Christoph Schaefer, Victor Pankratius, and Walter Tichy. "Atune-IL: An instrumentation language for auto-tuning parallel applications". In: *Euro-Par 2009 Parallel Processing* (2009), pp. 9–20.

[37] Jason Ansel et al. *PetaBricks: a language and compiler for algorithmic choice*. Vol. 44. 6. ACM, 2009.

[38] Ananta Tiwari et al. “A scalable auto-tuning framework for compiler optimization”. In: *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE. 2009, pp. 1–12.

[39] I-Hsin Chung, Jeffrey K Hollingsworth, et al. “Using information from prior runs to improve automated tuning systems”. In: *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society. 2004, p. 30.

[40] Chun Chen, Jacqueline Chame, and Mary Hall. *CHiLL: A framework for composing high-level loop transformations*. Tech. rep. Citeseer, 2008.

[41] Woongki Baek and Trishul M Chilimbi. “Green: a framework for supporting energy-conscious programming using controlled approximation”. In: *ACM Sigplan Notices*. Vol. 45. 6. ACM. 2010, pp. 198–209.

[42] Henry Hoffmann et al. “Dynamic knobs for responsive power-aware computing”. In: *ACM SIGPLAN Notices*. Vol. 46. 3. ACM. 2011, pp. 199–212.

[43] Richard Vuduc, James W Demmel, and Katherine A Yelick. “OSKI: A library of automatically tuned sparse matrix kernels”. In: *Journal of Physics: Conference Series*. Vol. 16. 1. IOP Publishing. 2005, p. 521.

[44] Markus Püschel et al. “Spiral: A generator for platform-adapted libraries of signal processing alogorithms”. In: *The International Journal of High Performance Computing Applications* 18.1 (2004), pp. 21–45.

[45] R Clint Whaley and Jack J Dongarra. “Automatically tuned linear algebra software”. In: *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE Computer Society. 1998, pp. 1–27.

[46] N O’Leary. “Paho - Open source messaging for M2M”. In: *Eclipse Paho’s MQTT* (2014).

[47] R Light. “Mosquitto - An open source MQTT v3.1 and 3.1.1 broker”. In: *http://mosquitto.org* (2013).

[48] Christian Bienia et al. “The PARSEC benchmark suite: Characterization and architectural implications”. In: *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM. 2008, pp. 72–81.