UNIVERSIDAD DE GUADALAJARA
CENTRO UNIVERSITARIO DE CIENCIAS ECONÓMICO ADMINISTRATIVAS
DOCTORADO EN TECNOLOGÍAS DE INFORMACIÓN

Performance Prediction of Long-running Applications in Distributed and Multi-core Environments

Tesis para obtener el grado de
Doctor en Tecnologías de Información con especialidad en Sistemas de Información de Arquitectura Distribuida

Presenta
Jesús Adán Flores Contreras

Director
Dr. Héctor Alejandro Durán Limón

Co-Director
Dr. Efrén Mezura Montes

Lectores
Dra. Maria Elena Meda Campaña
Dr. Juan Carlos González Castolo
Dr. Jérôme Leboeuf Pasquier

Zapopan, Jalisco. Julio de 2016

Abstract

A distributed system is a set of independent computer systems interconnected by a network, which work in a cooperative way and behave as a single system creating an underlying platform for different kinds of applications. These platforms are usually used to execute long-running applications, such as high performance computing (HPC) applications, that demand a lot of computational resources e.g. CPU processing power, memory, and network bandwidth. Performance prediction has been used in HPC problems to improve workload management, to estimate resource provisioning, and to schedule tasks in distributed systems. However, current multicore platform-independent approaches need to make either partial executions of the application or a complete execution of an application model (i.e. a mini application that mimics the performance behaviour of the real application) on the target system. This fact can be an inconvenient when we do not have access to the target system, for example, when carrying out capacity planning for buying a new unseen cluster platform. In this work, we present a method to design a platform-independent prediction model for estimating the execution time of HPC applications in a free-workload environment. The proposed method uses classification trees as a technique to generate the prediction model. Our platform-independent prediction method avoids performing a partial or complete execution of the application on the target system. We performed our evaluation with three parallel long-running applications, namely the Weather Research and Forecasting (WRF) model, Octopus, and miniFE. Our experimental results indicate that classification trees are a suitable technique for developing performance prediction models of distributed systems exhibiting non-linear behaviour reaching an accuracy above 85%.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Introduction

High Performance Computing (HPC) systems are used on massive computational problems, due to their processing capabilities. HPC systems have been used to tackle problems that process large amounts of data or execute time consuming processing operations [119] [28]. Nowadays, different areas of study demand systems with high processing capabilities, such as meteorology, bioinformatics, astronomy, engineering, chemistry, seismology, and biology, among others areas. HPC relies on distributed systems such as Cluster or Grids. Distributed systems are defined as a set of autonomous computers interconnected by a network connection [29]. They work in a cooperative way and behave as a single computer to the user. Every computer shares its resources with other computing nodes, this increases the overall system computer power.

Distributed systems can either have single or multi-core processors. Importantly, since 2005 Intel released its first dual-core processor [89] and commodity computers. Then the commodity computers and distributed systems began to use this kind of processor architecture. Nowadays, multi-core systems are the most common architecture used in every kind of computer system. Burger et. al [121] defines a multi-core processor as a single-chip processor with two or more complete computational engines (cores). In multi-core processors every computational engine shares internal components (e.g. cache memory, and front-side-bus -FSB- bandwidth), among all the cores contained in the single-chip. The purpose of multi-core systems is to run more tasks simultaneously, which thereby achieve greater overall performance [7]. An inefficient management of those resources degrades the system performance.

Not only resource management is an issue in HPC systems, some other concerns, such as capacity planning and quality of service, have become important issues in HPC systems.

1.2. The Problem

Institutions (research centres, schools, or private companies) require HPC systems to satisfy their business needs. This is because of the need to process a large amount of information, or solving complex operations. When HPC capabilities are needed, some institutions opt to invest on buying HPC systems, or migrate to a service oriented approach to mitigate the lack of processing power.

However, these institutions face some issues when they acquire new equipment or increase their own processing power. Some of these issues are resource provisioning

and capacity planning. Capacity planning in HPC is the process of determining the processing power needed by the institution. When capacity planning is not carried out properly there are two possible outcome scenarios: over provisioning and under provisioning. In the over provisioning scenario, the institution buys an HPC system whose processing resources exceed the needs. Hence, the institution ends up investing more money for an equipment that will not be exploited to the full of its processing capabilities. On the other hand, with the under provisioning scenario, the HPC system does not satisfy the processing needs. In both cases, the institution makes unsatisfactory investments on equipment that do not fulfil its needs.

When an institution does not have enough capital to invest on HPC equipment, such an institution usually relies on migrating its processing needs to a Service Oriented Architecture (SOA), such as cloud computing, or HPC as a service. In a SOA approach, a service provider grants its costumers access to its HPC equipment. But, for a service provider it is better to know ahead the user needs to offer different scenarios that fulfil the customer needs so that the overall performance of other clients is not affected. In SOA it is usually established a contract between the customer and the service provider. This contract defines the services, also know as service level agreements (SLAs), that the provider will grant, e.g. storage capacity, capacity of processing power, network bandwidth, memory, and application execution time among others. Therefore, it is imperative for the service provider to manage efficiently the system resources in order to guarantee that the customer receives the services established in the SLA [3]. SLAs are important in service-oriented architectures [67], because they guarantee that a given Quality of Service (QoS) will be maintained [35].

Therefore, in either case, when the processing resources are bought or hired, performance prediction plays an important role in capacity planning.

1.3. The need of performance prediction

The performance prediction field has grown over the years due to its usefulness in different areas to solve real problems, e.g. they have been used to predict the water quality [91], to estimate the combustion and emission in an engine cylinder using biodiesel [90], to diagnose brain tumours [42], to identify obstacles on mobile robots [102], to predict the stock market prices [68], and to diagnose breast cancer [113], to name some examples.

Performance prediction has also been used in HPC problems to improve workload management [114], to estimate resource provisioning [103], and to schedule tasks in distributed systems [94]. It has been stated by Kundu et. al [69] that performance prediction models *"have the ability to predict application performance for a given set of hardware resources and are used for capacity planning and resource management"*. Therefore, these models are able to be used to solve HPC problems.

Each proposal makes use of different methods, such as, mathematical methods (e.g. linear and non-linear regression), and machine learning techniques (e.g. artificial neural networks, and classification trees). For example, Piro in [100] proposes a prediction method to forecast the execution time of jobs using historical information; the prediction

model is also able to predict resource usage (e.g. physical, and virtual memory). Their approach is based on predicting the resource usage of a single job, and applying a characterisation of the system to represent that behaviour as a template. The template is used to classify the incoming jobs with similar characteristics of the job used as template.

Yang et. al in [134] proposed an architecture independent prediction approach to estimate the application execution time in different architectures. Their method needs to perform a full execution on the reference system and a partial execution in the target architecture. Some disadvantages of this approach are: a) the necessity of performing a partial execution in the target system and b) the necessity of knowing the control flow of the application. Shimizu et. al in [112] developed an application agnostic and platform-independent prediction model. This approach measures the usage of several resources (e.g. cache memory, and network bandwidth communication). The values measured are used as input parameters for a linear regression-based prediction model. However, the approach was only intended for single-core processors. Gianni et. al in [39] proposed a method to predict the running time of a simulation system in a distributed environment. Their proposal is an iterative process that needs to perform a characterisation of the application. A disadvantage of this proposal is that the prediction method has to be implemented in the design phase of the application. Finally, Sanjay et. al [107] the authors developed a modelling strategy to predict the runtime of parallel applications in dedicated and non-dedicated clusters. The model takes into consideration several resources in which resources are represented as an algebraic expression. This model offers several functions and the user must select one that fits better to the application behaviour. A disadvantage of this approach is that the user should have the experience to select the function to get better accuracy.

The works mentioned above are some examples of performance prediction methods (the theoretical context related to performance prediction is fully detailed and analysed in Chapter 2). Some approaches are designed to be used on a specific platform and/or for a particular application [21, 30, 33, 54, 63]. A disadvantage of the prediction models proposed by these approaches is that the models are usually designed to be used with the same application, or same platform [97][114], which means that we need to redesign the model to use it on a different application or platform. Some other approaches are platform-independent and/or application-agnostic [127, 136, 27, 46, 58, 72, 78]. Platform-independent refers to heterogeneity in hardware configurations (i.e. different CPU speed, different amount of RAM, etc.) within the same hardware architecture (e.g. Intel). A platform-independent model is the result of considering different hardware configurations where several benchmarks are executed to evaluate the performance of an application, this resulting on a prediction model. Then the model is able to predict the application performance on unseen hardware configurations, where no benchmarks or application has been run yet. However, current multicore platform-independent approaches need to make either partial executions of the application or a complete execution of an application model (i.e. a mini application that mimics the performance behaviour of the real application) on the target system. This fact can be an inconvenient when we do not have access to the target system, for example, when carrying out capacity planning for

buying a new unseen cluster platform. Therefore, we believe further research is required in this area.

1.4. Hypothesis

Based on the statements established in early sections, we define the following hypotheses.

Hypothesis 1: It is possible to develop a method for designing a platform-independent model to predict the execution-time of a long-running scientific application (HPC application) in a free-workload cluster system with multi-core processors. This considering a maximum prediction error of 15% and without performing partial executions or executions of an application model on the target platform.

Hypothesis 2: Modelling only a subset of resource types and characteristics is enough, for such a prediction model, to obtain such a prediction error. This subset is the number of cores, number of nodes, the operation frequency of the CPU Core, RAM memory, and network bandwidth.

1.5. Research question

We defined the following research questions as a guidance for conducting this research.

- What kinds of resources have an impact on the performance execution of long-running applications in multi-core systems?
- How can the resources be modelled in order to generate a platform independent resource usage model for multi-core systems?
- What kind of regression technique would be more accurate to predict the application performance?
- What are the steps of the method to design a performance prediction model?

1.6. Research variables

Different resources are considered to estimate the application execution time; e.g. CPU, memory, I/O access, and network among others. Besides, each resource has specific characteristics that can be considered too, e.g. MIPS, clock frequency, number of floating-point operations (FLOPS), cache memory, RAM memory capacity, network latency, and network bandwidth. Many resource characteristics can be included in a prediction model. However, we consider only those resources and resource characteristics which have a greater impact on the performance of HPC applications. Therefore the independent variables are $\mu, \nu, \omega, \varphi, \beta$ and the dependent variable is τ_{app} as shown in equation 1.1.

$$\tau_{app} = f(\mu, \nu, \omega, \varphi, \beta) \tag{1.1}$$

Where:

- τ_{app} represents the application execution time in the distributed system
- μ is the number of Cores used to execute the application
- ν defines the number of nodes used
- ω indicates the operation frequency of the CPU Core
- φ is the memory available on each node
- β is the network bandwidth in the system

1.7. Research scope and assumptions

We established the following considerations and assumptions to limit the scope of this research. The prediction models generated with the proposed method have the following considerations:

- A platform-independent model can be used to predict the execution time of HPC applications on "unseen hardware configurations" (each configuration involving a different amount of CPU speed, number of processors, number of cores, RAM, etc.) where the same architecture instruction set (e.g. Intel) is assumed.
- A prediction model is designed for a free-workload environment
- A prediction model targets multicore cluster systems where the nodes are homogenous
- A prediction model does not require to perform a characterisation of the application nor implementing code instrumentation
- A prediction model needs a history dataset with the application execution time on different platform configurations sharing the same hardware architecture
- A prediction model is able to predict the execution time of an HPC application on an unseen platform configuration i.e. on a platform configuration where the application has not ever run
- A prediction model does not require performing partial executions or executions of an application model on the target platform
- A prediction model is application-dependent, i.e. such a model is only able the predict the behaviour of a single application

- A prediction model is input data-dependent. That is, the application input data for each run of a history dataset remains fixed, therefore, such a prediction model is only able to predict the behaviour of an application for such fixed input data.

Summarising, several aspects need to be considered when a prediction model is designed; we established a few considerations to design our prediction model. For instance, it is necessary to have history data of the execution time in order to generate a profile of the application. However, it is unnecessary to generate a characterisation of the application. Instead, we consider that such application profile is enough to represent the behaviour of the application in the system to predict its execution time on unseen platforms.

1.8. Research goal

The main objective of this research is to develop a method to design a platform-independent performance model of HPC applications in multicore systems, rather than designing a single model for performance prediction. The reason is, our approach assumes it is needed to produce a different model for a different application as well as for a different application input data.

The specific objectives are the following:

- The designed prediction models will not require a characterisation of the application
- The designed prediction models will be able to predict the execution time on unseen platforms configurations
- The designed prediction models will be able to predict the execution time of HPC applications running on either multi-core or single core processors
- The designed prediction models will be capable of being used by a person without expertise on performance prediction

1.9. Structure of the Thesis

This thesis is structured as follows. Chapter 2 details the theoretical context related to performance prediction methods. Chapter 3, presents the proposed method to design a performance prediction model. Chapter 4 evaluates the approach proposed in this thesis. Finally, some concluding remarks and future work are given on Chapter 5.

2. Theoretical context

2.1. Introduction

The performance prediction field has grown over the years due to its usefulness in different areas. Performance prediction methods usually provide insights for resource provisioning, workload management, and scheduling [94]. Hence, performance prediction is used to optimise the execution time of applications running in distributed systems.

There are many methods for predicting the performance behaviour of an application. These methods can be classified into two main categories: *application-oriented* and *resource-oriented* methods [139]. *Application-oriented* methods focus on estimating the runtime of an application in parallel systems whereas *resource-oriented* methods predict the performance of a specific resource such as CPU workload, memory or network bandwidth. Due to the research scope of this thesis, we focus on application-oriented performance.

We followed a systematic literature review [45] to identify the most relevant papers related on performance prediction. A systematic literature review aims to identify the most relevant papers that have been published about a specific topic [66]. The review consists of a procedural protocol to discover the most related information about the selected topic [20]. The methodology has several steps for extracting the information from the consulted articles. The process we followed for the systematic literature review is fully detailed in [38][1]

This chapter has the following structure. Section 2.2 describes the classification framework. The reviewed approaches are presented in Section 2.3. Section 2.4 presents a discussion of related work. Concluding remarks are given in Section 2.5.

2.2. Classification Framework

Performance prediction methods have been applied on a wide-range of problems in distributed systems. Hence, we defined a classification framework based on the subjects exposed in the papers reviewed. The framework classifies the methods proposed in the papers we analysed as well as the application domains in which these methods are employed. Due to the wide-range of proposals, the classification framework is divided into two main categories: *Prediction domain* and *Prediction methods*.

The *prediction domain* denotes the application domain area where performance prediction has been applied. The *Prediction methods* category classifies the approaches according to the kind of technique or method used to predict the application runtime

[1]The paper was submitted in February 2016, and until May 2016 has the status "under review".

(e.g. analytical or non-analytical methods). The following paragraphs give a more specific description about the framework structure.

2.2.1. Prediction Domain

The prediction domain category involves domains of the applications and systems that have been studied; we divided the framework into two main categories: *application domain* and *platform domain*. The prediction domain framework is presented in Figure 2.1.

The application domain branch denotes the kind of applications where performance prediction has been applied. The platform category involves the types of computer systems used to run the applications[2] e.g. Clusters, and Grid systems.

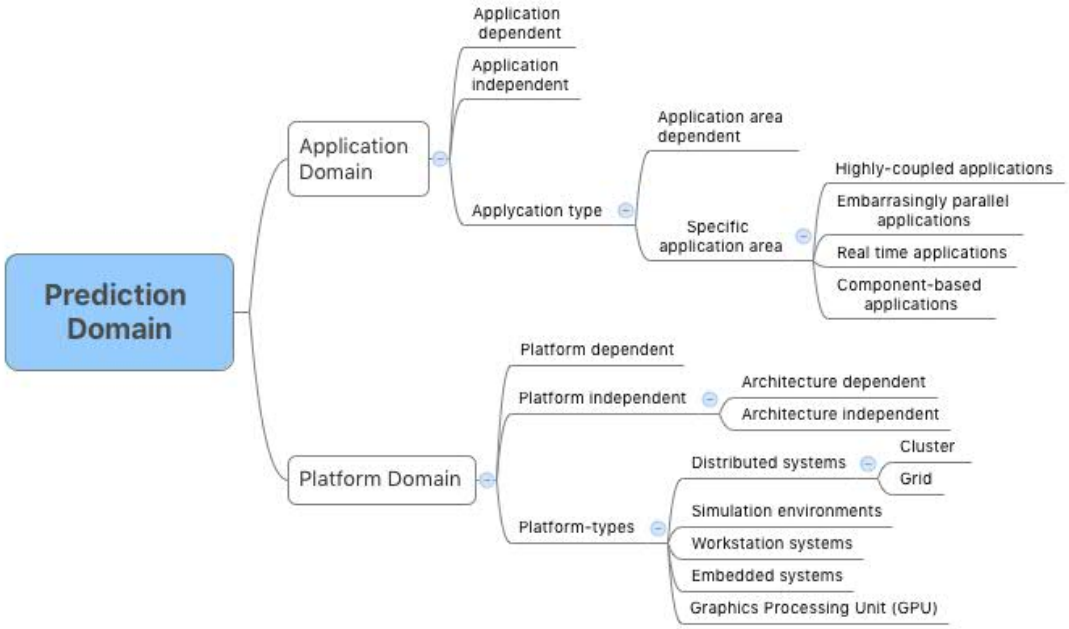

Figure 2.1.: The Prediction domain sub-category in the classification framework

The reviewed approaches are mapped to both the application branch and domain branch of the classification structure. However, some categories are mutually exclusive, such as application-dependent and application- independent. This is because application-independent means that it is not necessary to know the application structure, and the model can predict the performance of different kinds of applications without the need to make a new model representing a different application structure. Application- de-

[2]It should be noted that the simulation environments sub-category is part of the plat- form category since we considered a simulation as an executing platform. This given that a simulation approach is also mapped to a prediction method in our framework (i.e. analytical or non-analytical method)

pendent means that the model is designed to predict the performance of a specific type of application. However, elements belonging to the categories application-dependent and application-independent can be also presented in the application-type branch. The application-type category involves the types of applications used to test the approaches.

The platform domain section follows the same structure, where platform- dependent and platform-independent are mutually exclusive. This is because platform-independent refers to heterogeneity in hardware configurations (i.e. different CPU speed, different amount of RAM, etc.) within the same hardware architecture. This category does not involve architecture independence where the same prediction model can be used for architectures with a different instruction set. In the platform-independent category, a prediction model can be used on different platforms without the need to design another specific model to a given platform [112]. The platform-dependent and platform-independent categories refer to the platform-types, i.e. the type of systems that have been used to execute the applications. The architecture independent category and the platform-dependent category are mutually exclusive whereas the architecture dependent category can be inclusive of any of the remaining categories. The following paragraphs give further details about each of the categories included in the prediction domain sub-category.

The application-domain category involves the following sub-categories:

Application-dependent. Here are included approaches that are designed for being applied to specific applications. That is, a prediction method in this category cannot be applied to different types of applications without rebuilding the entire prediction model.

Application-independent. This category involves approaches that define a model that can be applied to different application types, this without the need of defining a new model for representing the application performance behaviour of different application types.

Application-type. Here are included the types of applications that are used in the reviewed approaches.

The platform-domain category includes the following sub-categories:

Platform-dependent. The approaches within this category are able to model only the specific hardware configuration (CPU speed, number of processors, number of cores, amount of RAM, etc.) where the approach is applied. That is, a prediction model cannot be employed on unseen hardware configurations.

Platform-independent. The models included in this category can be used on "unseen hardware configurations" (each configuration involving a different amount of CPU speed, number of processors, number of cores, RAM, etc.) where the same architecture instruction set (e.g. Intel) is assumed. An "unseen hardware configuration"

is a hardware platform on which the application either has never been run or has not run to completion before. One approach to produce a model regards employing different hardware configurations, which typically involves running a number of benchmarks on such hardware configurations. The produced model can then be used to predict the performance of an application on unseen hardware configurations where no benchmarks are run. Some other approaches involve running partial executions on the target platform.

Platform-type. This category involves the type of platform systems (e.g. Grids, clusters, embedded systems, etc.) used in the proposals.

Architecture-independent. Here are included the approaches that are able to predict the runtime of an application on multiple hardware architectures (e.g. SPARC, Intel, etc.). Architecture-independent approaches are also platform-independent since these approaches can also be used on unseen hardware configurations.

2.2.2. Prediction Method

The Prediction method category involves a classification of different kinds of execution time prediction methods, as shown in Figure 2.2. This category is divided into three main categories: analytic, non-analytic, and characterisation.

There are sub-categories that are mutually exclusive such as the analytical and non-analytical categories. However, each one of these categories can be combined with the characterisation branch. Below we detail the content of each sub-category.

Analytic Methods

This category groups the mathematical methods that have been used to estimate the values of the different mathematical representations; in this group we can find the following items.

Model representation. This category involves approaches that use a mathematical model to represent the performance of the system. There are two main elements in this category namely, algebraic and polynomial expressions. An algebraic expression is an expression with different mathematical terms that use algebraic operations whereas a polynomial expression involves the sum of variables raised to a power and multiplied by a coefficient [71].

Statistic methods. This category groups statistical methods used to predict performance. Some of such methods include linear methods and non-linear methods.

Isoefficiency. This area involves system scalability. Isoefficiency is a function that measures the efficiency of parallel applications [34], which requires measuring the scalability of the problem in order to determine the amount of processors that the application needs [43].

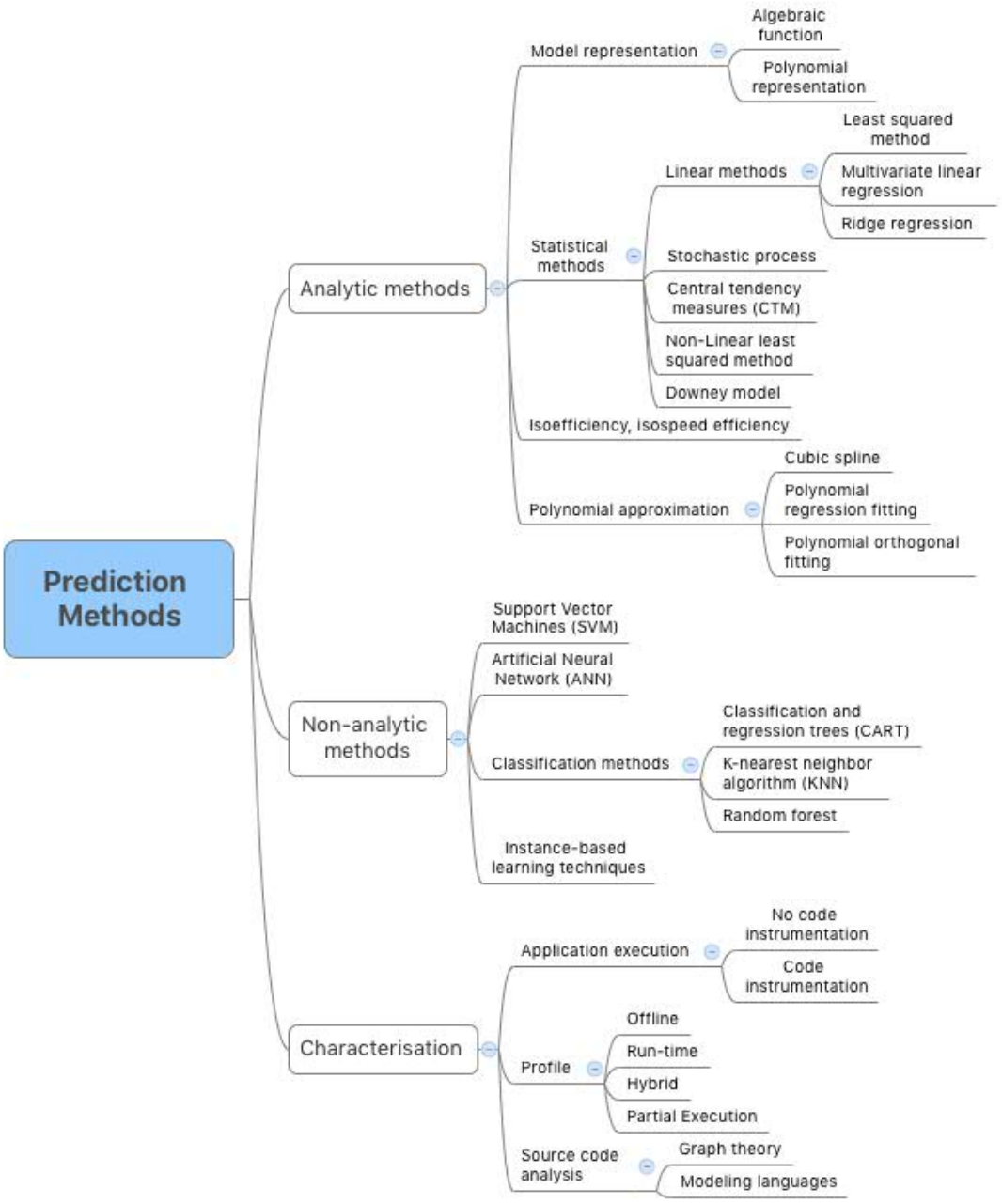

Figure 2.2.: The Prediction method category of the classification framework

Polynomial approximation. This area regards the methods that deal with functions expressed as the sum of terms of the form $a \times x^i$ [25].These expressions usually model a non-linear relationship, and the methods generate an approximation that can be used to extrapolate values that are not considered in the original data [41].

Non-Analytic Methods

Non-analytic methods group computational heuristic methods used in the performance prediction field. In this group we can find the following items.

Support vector machines. This category includes supervised learning mathematical models that are used for pattern recognition, classification and regression analysis.

Artificial neural networks. This category involves approximate functions used to perform regressions on non-linear systems and are inspired by the biological neural network system [76].

Classification methods. This category regards machine learning techniques that are used to make predictions models based on classifying information.

Instance-based learning or Lazy-learning methods. Learning methods involve the study of historical information related to an event. Learning how a system behaves involves the analysis of historical information of the system. With this information we acquire the knowledge and we are able to predict its future behaviour. Instance-based learning techniques involve different kind of algorithms that focus on the data instead of building rules or decision trees in order to extract the acquired knowledge [10].

Characterisation Methods

Characterisation methods are the techniques that were used by some of the reviewed approaches that performed a profile of the application in order to determine its behaviour, as described below:

Application execution. This group includes all approaches in which it is necessary to execute the application in order to determine its behaviour and deduce the model. This category can involve either, code instrumentation and non-code instrumentation techniques.

Profile. This category involves approaches that employ some analysis to perform a profiling of the application. The profile of the application can be made offline, at runtime, or in hybrid mode by combining both. Offline techniques perform the analysis of the application before executing the application in the system. Runtime techniques require running the application to analyse the system.

Source code analysis. This category includes approaches in which the source code of the application is analysed in order to determine its structure. Source code analysis techniques make use of mathematical knowledge in order to estimate the scalability of the system.

2.3. Approaches description

This section describes the approaches related to performance prediction. After describing these approaches. The papers related to the scope of our research are listed in Table 2.1. Such papers were grouped according to the Prediction methods category. For the description of the reviewed approaches, we grouped the papers according to the subcategories of Analytic methods and Non-analytic methods i.e. Model representation, Statistical methods, Isoefficiency, Polynomial approximation, and Non-analytic methods. We did not take into account the Characterisation category because the elements in such a category also form part of the categories mentioned above, as shown in the position column of Table 2.1, in such a column, the letter *"D"* denotes the position in the Prediction Domain category whereas the letter *"M"* defines the position in the Prediction Method category.

Author(s)	Paper	Framework position
Method Cathegory: Model representation		
Bauer et al.	[13]	**D:** App-dependent, arch-independent, Cluster **M:** Algebraic, linear least-sqrt, profile hybrid.
The paper presents a model to predict the performance of an application on different architectures. The process of creating the model involves analysing the application structure, and executing the application under different input parameters, and different machine settings. The application structure analysis is made in order to identify the parallel kernel(s)[3]of the application, the communication pattern, the input data involved in the computation, and the amount of data involved in the communication. The execution of the application is carried out to analyse the application behaviour in the system under specific input parameters and machine settings. The analysis generates two expressions, one is a linear equation to determine the computation performance, the other one represents the communication performance. To estimate the computation performance is applied the linear least-squared method, and to estimate the communication is used the LogGP model. The authors do not mention details on the method followed to evaluate the performance expressions on different architectures; however, the authors mention that they take into account the information of the execution analysis.		
Boullon et al.	[15]	**D:** App-dependent, Grid **M:** Code instrumentation, profile→run-time

Continued on next page

[3]A kernel is a function that is executed on each independent Core [123]

Table 2.1 – continued from previous page

Author(s)	Paper	Framework position
The authors propose a tool to estimate the execution time of MPI applications in a Grid environment. The tool generates an analytic prediction model for the kernel; the parameters of the models are the computational power, and the network latency and bandwidth. The tool generates a profile of the application; the profile involves executing the kernel several times under different scenarios that let them analyse the kernel behaviour in order to generate the model based on the empirical analysis of the information collected. However, the described procedure does not provide information about the accuracy and the kind of the model generated, because they only describe the characterisation method to generate the models.		
Carrington et al.	[17]	**D:** App-independent, workstation **M:** Stochastic, profile→run-time
This paper proposes a prediction framework for scientific applications. The framework has several tools, and several analytical models that are able to be applied on different application programs. The proposed framework involves a two-step process. The first one performs a profile of the application at runtime to determine the application execution time. The second step measures the rates at which the system is able to perform the operations in the system. In this step, the proposal measures the memory I/O rates, the number of floating point operations, and the number of messages sent. These parameters, which are gathered through a benchmark, determine the capabilities of the system. The application profile and the processing capacity rate employ a convolution method to predict the performance of an application by performing a match of the application profile with the capacity rate.		
Cornea et al.	[27]	**D:** App-independent, embarrasingly, Cluster, simulation **M:** Algebraic, no code instrumentation, Graph theory
This proposal involves a performance prediction tool, which is designed for high performance computing (HPC) applications written in C, C++, and Fortran. The method requires to perform an analysis of the application's source code. Such an analysis generates a graph structure of the application that is used to build an algebraic model of the application.		
Davis et al.	[30]	**D:** App-dependent, plat-dependent, Grid **M:** Algebraic, central tendency measurements
The paper proposes a prediction method designed for an application called Hydrodynamic benchmark. This application simulates the processing of a several mixed materials under stress. The application's input data is arranged into a 3D grid of cells. Their method considers three elements in the distributed system: the local computation, near-neighbour communication and collective communication. One important characteristic is that the method should know how the 3D grid of cells is divided and distributed in the system's nodes. The model consists of several algebraic expressions that represent the local computation, near-neighbour communication, and collective communication.		
Gualandris et al.	[44]	**D:** Highly-coupled, Cluster, app-dependent **M:** Algebraic, source code analysis

Continued on next page

Table 2.1 – continued from previous page

Author(s)	Paper	Framework position
The authors evaluate and predict the performance of the N-body algorithm [77], which simulates the motion of bodies that interact with other ones in a distributed system. Their prediction model takes into account the following elements: the speed of each node, the network bandwidth, and the latency of the application start-up execution. Several algebraic equations are proposed to model such elements.		
Hudik et al.	[52]	**D:** App-dependent, Cluster **M:** Algebraic
The authors propose an application specific performance prediction mathematical model. Such model is able to predict the execution time of algorithms to solve linear equations in shared and distributed memory systems. The model takes into account two kind of resources: CPU and network bandwidth. The approach estimates the communication delay as well as CPU time. However, the authors focus on evaluating the performance of the algorithms whereas the evaluation of the prediction model is omitted.		
Ivannikov et al.	[57]	**D:** App-dependent, Grid **M:** Algebraic, profile offline
The authors propose a model for MPI Java parallel programs in order to determine its performance in a distributed environment. The model represents all the components of the Java program, it also analyses the communication components of the MPI applications. The model generates an algebraic representation of the elements to determine the application's future behaviour.		
Jarvis et al.	[58]	**D:** App-independent, Grid **M:** Profile→offline, graph theory
The authors define a performance prediction method that analyses the interconnection in both private and public networks. The method performs a source code analysis and the data traces of the resource usage obtained while the application is running in the system. They used the PACE tool to analyse the information collected and predict the application runtime.		
Kerbyson et al.	[63]	**D:** App-dependent, workstation **M:** Algebraic, source code analysis
The authors present a profiling method to analyse the performance of a numerical simulation application that evaluates the turbulence and fluid motion of an aerospace system. A characteristic of their prediction model is that it needs to know how the application operates and processes the information. The application profiling allows them determining the processing time on each node, and the time required to transfer the information to all the nodes in the system. For each parameter that considers the model, an algebraic expression is proposed in order to generate a mathematic representation of every parameter.		
Kestor et al.	[64]	**D:** GPU, app-dependent **M:** Profile→offline, graph theory

Continued on next page

Table 2.1 – continued from previous page

Author(s)	Paper	Framework position
The authors propose an emulation framework for task-based applications, the framework is also able to predict their performance on large many-core architectures. The framework involves creating a direct acyclic graph (DAG) of the application structure; the graph contains all the tasks that compose the application. It is necessary to define the resource contention access of the many-core architecture, thus, the authors employ a driver library that contains the information of the resource contention of different hardware architectures. The framework has a prediction engine, however the authors do not focus on defining how such an engine is designed, they only describe that they use a polynomial equation of grade 16.		
Lee et al.	[70]	**D:** Embarrassingly, arch-independent, workstation, GPU **M:** Algebraic, code instrumentation, modelling language
The authors propose a framework able to model a parallel application able to predict the runtime on different environments. The framework analyse the application source code in order to generate a parameterised model of the application structure. The framework is divided in two stages, the first one regards analysing the source code in C language and modelling the source code into the Aspen performance modelling language. The second stage involves using the application model in the Aspen performance prediction tool to estimate the number of instructions needed to execute the application.		
Midorikawa et al.	[87]	**D:** App-independent, Cluster **M:** Algebraic, profile→run-time, graph theory
The authors present a method to evaluate the performance of MPI programs. Their method considers three techniques, graphical representation in order to represent the application code and identify the relation among processes. The second one measures the execution of the application and the third one performs an analytical modelling of the application with the information gathered in the previous steps.		
Panadero et al.	[96]	**D:** App-independent, Cluster, plat-independent **M:** Code instrumentation, profile runtime, modelling language
The authors propose a method and a toolkit to predict the performance of an application on different distributed system platforms. The method involves characterising an MPI application, in order to identify the key parts of the application. This let the authors creating a mini program, called signature, that it is executed on the target platform. The execution of the signature allows them to analyse the behaviour of an application and predict its execution time. The toolkit automates the procedure of creating the mini program. The prediction model is based on identifying the key parts of the application code and determining the number of executions of each of these parts.		
Parakh et al.	[97]	**D**: App-independent, plat-independent, embedded systems **M:** Algebraic, no code instrumentation

Continued on next page

Table 2.1 – continued from previous page

Author(s)	Paper	Framework position
This paper proposes a prediction method for applications executed in Graphic Processing Units (GPUs) with cache memory. The proposal counts the number of instructions and calculates the memory access cycles. It is necessary to determine the system behaviour through several benchmark tests. That information is used to generate several algebraic expressions that model the elements that have an impact on the application performance.		
Sahuquillo et al.	[106]	**D:** Real-time, embedded systems, simulation, plat-dependent **M:** Algebraic
The authors propose an execution time model for real time applications based on heterogeneous multi-core systems, where each core has different operating frequency. The model is an algebraic expression with parameters: CPU for any operating frequency, memory and overlapping. An overlap is the time the processor executes non-dependent instructions while memory requests are served. The method to generate the model measures the computation time that each core needs to execute the tasks, the waiting time for accessing memory, and the overlapping time of computation and memory. The execution time is expressed as the time in cycles for any frequency. This equation estimates the execution time in cycles by varying the operating frequency according to the target platform.		
Seneviratne et al.	[109]	**D:** App-independent, workstation **M:** Algebraic, no code instrumentation
The authors developed a task profiling model for parallel applications that predict their execution time. The method needs to execute the application in order to determine the CPU usage. The CPU must not have any other workload when the application is executed.		
Sharkawi et al.	[111]	**D:** App-dependent, workstation **M:** Algebraic, No code instrumentation
The authors proposed a framework to evaluate the computation and communication components in order to make projections about the application runtime. The framework needs to perform a profiling of the application. Such a profiling is constructed by using the information gathered from previous application executions.		
Sun et al.	[116]	**D:** App-dependent, plat-dependent, GPU **M:** Algebraic, no code instrumentation
The authors present a method to estimate the execution time of a set of instructions in a GPU processor. The method considers the processing data and the number of instructions needed to process it. The method needs information of previous executions and that information is used together with Amdahl's law [5] to predict the runtime.		
Wong et al.	[127]	**D:** Plat-independent, Cluster **M:** Code instrumentation, profile→ partial execution

Continued on next page

Table 2.1 – continued from previous page

Author(s)	Paper	Framework position
The authors present an approach to platform independent performance prediction. Their approach is based on code instrumentation and partial execution (execution of the relevant parts of the program) on the target platform. A log of execution traces, that is produced by the execution of the instrumented code, is analysed to obtain a machine-independent application model. Such a model characterises the computation and communication behaviour of an application. The model, which is also a program, is run on the target machines to predict the execution time. Although the instrumentation of the code is automated, the application program needs to be recompiled using a dynamic library that the authors created to produce a trace of the binary application.		
Yero et al.	[135]	**D:** App-dependent, workstation **M:** Linear least-sqrt, modelling language
The authors present a performance prediction method that considers the time that a job needs to wait for using a resource. The method involves a symbolic representation to represent the application structure. The PAMELA tool, which is part of their approach, uses the symbolic representation to determine the application runtime.		
Method Cathegory: Statistical Methods		
Achour et al.	[2]	**D:** Cluster, app-dependent **M:** Linear least-sqrt, profile→ hybrid
The authors proposed a prediction framework of parallel programs that are executed in hierarchical clusters (clusters that are composed of a set of clusters). Their method is based on two steps. The first one makes a profile of both the application's core characteristics and the performance of network communication, and the second step determines the completion time of the application. Their proposal uses linear regression methods to predict the application runtime.		
Barnes et al.	[12]	**D:** App-independent, Cluster **M:** Algebraic, downey model
The authors present a method to predict application run-time, and its scalability. The method generates an application profile by executing the application with a limited amount of processors. Later they analyse its performance and extrapolate the data collected with a large number of processors. The method has three phases. The parameters take into account are the computation time, and communication time, which are the resources used by the application. The fitting function is a logarithmic-model, and for estimating the run-time apply a multi-variate linear regression method.		
Deshmen et al.	[31]	**D:** Embarrasingly, Cluster, app-dependent **M:** Algebraic, downey model
The authors present a prediction tool to determine the speed up and the application runtime in a cluster environment. Their method does not require knowing the application structure, neither the machine information. The method needs the information of previous executions in order to figure out how the application behaves in the system.		

Continued on next page

Table 2.1 – continued from previous page

Author(s)	Paper	Framework position
Gianni et al.	[39]	**D:** App-dependent, Grid **M:** Algebraic, profile→run-time, graph theory

The authors propose a method to predict the runtime of a distributed simulation system. Their method involves building a model of the system and running the model on an iterative way until reaching the prediction of the execution time. The proposal builds the model using an Execution Graph that represents the structure of the system and is obtained from the design documents. Their method also uses the system parameters such as CPU and network interconnection. After building the model, this is evaluated to obtain the performance prediction.

Happe et al.	[46]	**D:** App-independent, component-based, workstation **M:** Stochastic, modelling language

This approach evaluates the performance of a software component-based application. The approach uses UML diagrams to describe the behaviour of a parallel application. The UML diagrams allow for developing a meta model that represents the application structure. Such a meta model contains information about classes for symbols, sequences, loops, branches, and parallelism. There is a probability expression for each software component.

Huh et al.	[53]	**D:** App-dependent, real-time **M:** Stochastic, profile→run-time

This paper proposes a method to evaluate the performance of real-time applications. The author's method performs a profile of the application and applies a stochastic method to evaluate the application performance.

Mello et al.	[84]	**D:** Highly-coupled, Cluster, app-dependent **M:** non-linear least-sqrt, code instrumentation

The paper proposes a prediction model based on the analysis of the network communication, processing power, and I/O access behaviour. Their method uses chaos theory [120] and the non-linear least square method [62] to predict the application performance.

Pfeiffer et al.	[98]	**D:** App-dependent, workstation **M:** Linear least-sqrt, no code instrumentation

The authors propose a method to model the performance of an application in parallel computers. The method takes into account the latency in the system (such as communication latency) and the computation time. The information of those resources is expressed as a linear model, and they apply the least squared method [65] to predict the application's behaviour.

Sadjadi et al.	[105]	**D:** App-independent, plat-independent, Grid **M:** Algebraic, linear least-sqrt

The authors propose a performance prediction method for scientific applications. The method employs the resources usage information to determine the application performance and it does not require analysing the application source code to make the prediction. The model is able to be used on different platforms. The model is linear and they apply a linear regression method.

Continued on next page

Table 2.1 – continued from previous page

Author(s)	Paper	Framework position
Sanjay et al.	[107]	**D:** App-dependent, plat-independent, highly-coupled **M:** Algebraic, linear least-sqrt

The paper describes a modelling strategy to predict the runtime of parallel applications in dedicated and non-dedicated clusters. The model takes into consideration several resources, which are modelled as an algebraic expression. Such an expression determines the application behaviour.

Shimizu et al.	[112]	**D:** App-independent, plat-independent, Cluster **M:** Algebraic, linear least-sqrt, no code instrumentation

The authors present a performance prediction method that considers the resource application usage. The method is application agnostic and platform independent. The model considers several resources such as the CPU clock speed, cache memory, and communication bandwidth. Those resources are represented in an algebraic expression with a linear regression method.

Sodhi et al.	[115]	**D:** App-independent, embarrassingly, Cluster **M:** Code instrumentation, profile→partial execution, modelling language

The authors propose a framework to build a skeleton of an application that is used to predict the application performance. A skeleton is a short running program that represents the behaviour of the application. The skeleton executes the most representative operations of the application, since the resource usage of the skeleton is similar to that of the application. The estimated execution time of the application is the product of the skeleton execution time and the corresponding scale ratio of the application.

Wu et al.	[129]	D: App-independent, Grid M: Central tendency measurements, Code instrumentation

The authors proposed a scheduling algorithm to improve the distribution of jobs and enhance the performance of a Grid system. The main component of the scheduling algorithm is that it needs to predict the job's runtime to improve the performance. The prediction method involves the analysis of the resource usage pattern. The resource usage analysis is made with queue theory [73]. In addition, a number of algebraic expressions are proposed representing some of the parameters that are considered to predict the job's runtime.

Wu et al.	[130]	D: App-dependent, embarrasingly, dist-systems[4] M: Algebraic, Stochastic

The authors proposed a prediction method for parallel independent task in a distributed system. Their method evaluates the execution time of every single task. They apply a stochastic analysis to the information obtained form previous executions of a task.

Continued on next page

[4]It did not clearly define whether its approach is designed for a Cluster or Grid system, it only defined as distributed system.

Table 2.1 – continued from previous page

Author(s)	Paper	Framework position
Yang et al.	[134]	D: App-independent, plat-independent, highly-coupled, Cluster M: Algebraic, moving average, partial execution
The authors propose an architecture independent prediction approach based on the relative performance between different architectures. They assume that the execution of an application is ruled by three phases. The first phase is initialisation, which is the beginning stage when the application prepares the data and makes the distribution to the nodes if necessary. The second phase involves the computation phase and the IO stage. The former regards the stage where the timestep[5]is recorded. The latter involves the stage when the application performs all the operations to process the data. In addition, in this stage the output data can be written periodically as an intermediate checkpoint. The third stage is a finalisation phase, where the application ends its execution. The authors mainly focused on the study of the timestep computation and the IO phase in the model. The proposal uses two kinds of information. The first one is related to the information gathered after performing a full application execution on a reference system. The second one is the information gathered from a partial execution in the target architecture. A partial execution measures a limited number of timesteps, when such a number is reached, the application ends prematurely. The full and partial executed information is used in two models, one of them uses cumulative average or running average, and the second one employs a filter model.		
Method Cathegory : Isoefficiency		
Chen et al.	[21]	**D:** App-dependent, embarrasingly, plat-dependent, Grid **M:** Isoefficiency
The authors propose an algorithm to predict the performance of a parallel and distributed system. The method needs to perform a characterisation of all the features of a computer system. Each computer node has a quantitative value that represents the computing processing power. A scalability method is used to determine the performance of a parallel application in the system.		
Zhai et al.	[136]	**D:** Embarrasingly, simulation, plat-independent **M:** Isoefficiency, code instrumentation, profile→partial execution

Continued on next page

[5]Timestep is defined as one step of computation followed by inter-processor communication to update the data.

Table 2.1 – continued from previous page

Author(s)	Paper	Framework position
The authors propose a method to predict the application runtime based on analysing the sequential computation time of each process of the application. Parallel applications are divided into processes; each one is a group of sequential operations. The method involves executing the most relevant processes in order to analyse its sequential behaviour as well as executing the application to generate a log of the computation and communication of the application. They put together the analysis of the sequential parts, and the log information to determine the runtime. In order to predict the application runtime in a different platform, they execute representative processes of the sequential parts on the target platform to generate a log record of the execution. This log record is used as input data in the SIM-MPI simulation tool, which estimates the execution time.		
Method Cathegory : Polynomial approximation		
Chtepen et al.	[24]	**D:** Embarrasingly, Grid, app-dependent **M:** Polynomial, profile→run-time
Their method is used to predict the runtime of an application in a Grid Environment. Their proposal uses a prediction method that gathers some information when the application is executed in the system. Such information is about the jobs' execution including the job's name, prediction history, progress timestamps, and some parameters provided by the user. The method proposed by the authors employs a polynomial fitting of the application behaviour.		
Jayakumar et al.	[59]	**D:** App-independent, plat-independent, Cluster **M:** Polynomial, profile, cubic spline
The authors proposed a prediction framework for HPC applications. The framework analyses the application source code to identify the significant parts of the code, i.e. those parts that demand more computing resources. The framework also involves a database where several profiles of parallel kernels are stored. The framework creates a profile of each phase of the application and adds such a profile to the database. Each reference kernel is executed in a system under different scenarios; this let them to generate a polynomial function that represents the application behaviour in the system. The profile of the small execution let them to match between the reference kernels and the phases of the application, after finding the correct match, they are able to predict the performance.		
Lobachev et al.	[78]	**D:** App-independent, embarrassingly, workstation **M:** Algebraic, linear least-sqrt, cubic spline, polynomial regression
The authors propose a method to predict the application runtime for new input processing data, and new number of processors for the same computer system. The proposal analyses the sequential and parallel application behaviour. They evaluate different methods such ash polynomial regression fitting, cubic spline, polynomial orthogonal fitting, and simple linear regression. The parameters used to predict the performance are the number of CPUs, the sequential execution time, and the number of instructions used to execute the application. The prediction method is evaluated with five different applications.		
Method Cathegory: Non-Analytic Methods		

Continued on next page

Table 2.1 – continued from previous page

Author(s)	Paper	Framework position
Akay et al.	[4]	**D:** Simulation, plat-dependent **M:** Support vector machines
This paper presents a prediction performance method for a distributed shared memory multiprocessor, connected through an optical multiprocessor exchange bus. The proposal uses a Support Vector machine technique [61] to predict the performance. The method uses the information provided by a simulation tool.		
Dodonov et al.	[33]	**D:** App-dependent, Grid **M:** Artificial neural networks, no code instrumentation, profile→hybrid
The authors present a prediction method based on neural networks [76] and chaos theory [120]. Their method involves both an off-line and on-line monitoring evaluation of the application to determine and predict the application performance. The off-line evaluation analyses the process execution traces, and the on-line evaluation relies on monitoring the application state while executing.		
Hutter et al.	[54]	**D:** App-dependent, Cluster **M:** CART, random forest, artificial neural networks, ridge regression
The paper describes a process to predict the run-time of three specific algorithms (propositional satisfiability problem, travel salesman, and mixed integer programming) for new input data. The proposal models the performance of each algorithm by using the following techniques: random forest, ridge regression, neural networks, Gaussian process regression, and classification and regression trees. The method executes each algorithm using different input data. The experiments were carried on a cluster with dual core processors.		
Ipek et al.	[56]	**D:** App-dependent, plat-dependent **M:** Artificial neural networks
The authors present a performance prediction method based on a multilayer artificial neural network. The prediction method is designed for the Semicoarseni game multi grid solver (SMG2000). The neural network is a feed forward system with sigmoid function as activation functions. The model is evaluated on two different platforms.		
Li et al.	[72]	**D:** App-independent, workstation **M:** CART, no code instrumentation, profile→run-time
This paper provides a performance prediction approach for a single-core processor and a chip multiprocessor system. The proposal is based on performing an application profile at runtime. Their model is based on using the classification and regression technique [19] to predict the performance.		
Marin et al.	[80]	D: Workstation, arch-independent **M:** Profile→hybrid, code instrumentation, graph theory

Continued on next page

Table 2.1 – continued from previous page

Author(s)	Paper	Framework position
The authors present a tool to model the application structure in order to predict the L1, L2, TLB cache miss counts, and the application execution time. The model is able to predict the run-time in different architectures. The method performs a static and dynamic analysis of the application. The static stage analyses the binary file in order to both model the control flow and generate a data flow graph of the application. The binary analysis allows for generating a language-independent model. The dynamic analysis collects information about the frequency of operation of each control flow graph. The paper does not describe the prediction model clearly, it only describes the resources taken into account to predict the execution time on different machine targets.		
Oyamada et al.	[95]	**D:** App-dependent, embedded systems **M:** Artificial neural networks, no code instrumentation
The authors' approach consists on determining the application performance in an embedded system. The method uses artificial neural networks [76] as this kind of systems present a non-linear behaviour. The method needs to perform a profile of the application, and the neural network is trained for the specific architecture of the platform system.		
Phinjaroenphan et al.	[99]	**D:** Embarrasingly, Grid, app-dependent **M:** K-nearest neighbor
The paper presents a method to estimate the execution time of a parallel task with inter-task communication on Grid nodes. They assume that the input data, the number of tasks, and the topology of the application are known. The authors evaluate their approach with an application involving matrix multiplication, which is divided into several task; the execution time of each task is predicted for each node. They used the k-nearest-neighbours (knn) algorithm to estimate the run-time.		
Prem et al.	[101]	**D:** App-independent, Cluster **M:** Support vector machines, no code instrumentation
The authors propose a predictor approach based on support vector machines [61]. Support vector machines use historical data of different parameters such as available CPU, and network bandwidth to determine future values of the application execution. The paper also compares their proposal with the predictions made by the Network Weather Service system [126].		
Smith et al.	[114]	**D:** App-dependent, plat-dependent, Grid **M:** Instance based, no code instrumentation.
The authors present several prediction techniques for specific elements such as the application execution time, the time of transferring a file, and queue waiting time. Their proposal involves using historical information of computing resources events. In the case of predicting execution time, they use the workload traces of the system.		

Table 2.1.: Brief description and location in the classification framework of the papers considered in the review

2.4. Discussion

This section presents some key elements found after analysing the information collected on the reviewed approaches. The first element is about how resources affect the application execution. The most common resources that are used to determine the application runtime are CPU, network, memory, and disk I/O. However, CPU has many characteristics that determine its performance, some of these characteristics are the number of instructions per second, CPU clock frequency, the number of float or integer operations per second, and cache memory among others. These characteristics or CPU parameters determine how the application executes in the system. Most of the proposals have not made an analysis to determine the correlation between the resources and the application execution time. However, Carrington in [17] estated that only two resources are necessary to estimate the application execution time on systems with single-core processors. Such resources are the CPU and network bandwidth. Carrington also estated that if another resource were added to the model, the prediction accuracy would only be improved in 2%. We also observe that most of the proposals use different types of resources, which were chosen based on the application type or domain area. Consequently, there is not a consensus on the specific resource characteristics needed to determine the performance behaviour of an application.

The resources determine how an application behaves in a system, and those are represented on the prediction methods. We found that 41 approaches uses analytic methods, those approaches provide an abstract representation of the resources in the model of their proposals. 61% [12, 13, 27, 30, 31, 39, 44, 52, 57, 63, 70, 87, 96, 97, 105, 106, 107, 109, 111, 112, 116, 127, 130, 134, 135] of the approaches represent resources as a term in an algebraic equation. 7% [24, 59, 78] of approaches use polynomial approximation whereas the remaining 32% [2, 15, 17, 21, 46, 53, 58, 64, 84, 98, 115, 129, 136] of approaches do not define a specific way about how the resources are represented in the proposed model. For example Happe et al. in [46] use stochastic analysis and non-linear methods, and no mathematical expression involving a representation of the resources is defined by the authors. In addition, there is not a standard manner to define resources; each proposal defines its own way of representing resources. It is well known that when analytic methods, such as regression, are used, the most feasible way to deal with those methods is by means of defining algebraic equations. However, in some cases, instead of generating an algebraic expression that fits to the data, a polynomial expression that adjusts to the data is used. Finally, some of the approaches involve the analysis of the execution behaviour of the system by observing how the application's resource demand varies over time. In this case, the proposals employ either a polynomial expression or a heuristic method. 8% [2, 4, 106, 136] of proposals are based on simulation. These proposals usually focus on both analysing the application source code in order to make a profile of the application's characteristics and specifying some parameters that define the characteristics of the system where the simulation will be held. Once these elements

are defined, the simulator generates a data set of information about how the application behaves; this is used by either analytical or no analytical methods to generate a model of the application behaviour in the system.

Another finding on those approaches was about how they generate the prediction methods, some approaches analyse the application source code. 31% [13, 27, 44, 46, 57, 58, 59, 63, 64, 70, 87, 96, 99, 115, 135, 136] of the approaches perform an analysis of the application source code, and the prediction error ranges from 4% to 25% and has an average of 12.91%. On the other hand, 69% [2, 4, 12, 15, 17, 21, 24, 30, 31, 33, 39, 52, 53, 54, 56, 72, 78, 80, 84, 95, 97, 98, 101, 105, 106, 107, 109, 111, 112, 114, 116, 127, 129, 130, 134] of proposals did not analyse the source code, and the prediction error accuracy ranges from 1% to 30% and has an average of 14.53%. It should be noted that the accuracy of the prediction models is not necessarily better when source code analysis methods are applied since the prediction accuracy average of proposals that employed source code analysis is lower than the other group of proposals.

Following element to analyse was the proportion of research dedicated to both application agnostic and platform-independent methods. We found that 18% [56, 59, 96, 105, 107, 112, 127, 134, 136] of the proposals are platform-independent, 35% [12, 17, 27, 46, 58, 59, 72, 78, 87, 96, 97, 101, 105, 109, 112, 115, 129, 134] are application-independent, 6% [13, 70, 80] are architecture-independent, and 10% [59, 96, 105, 112, 134] are platform- and application-independent. Although, we made our review until September 2015, we only found 6 approaches [13, 80, 96, 107, 134, 136] identified as a platform-independent using multi-core processors, and one platform-independent approach [70] employing many-core processors. However, we noticed that all the platform-independent approaches need to make either partial executions of the application or a complete execution of an application model (i.e. a mini application that mimics the performance behaviour of the real application) on the target system. This fact can be an inconvenient when we do not have access to the target system, for example, when carrying out capacity planning for buying a new unseen cluster platform.

2.5. Conclusions

This chapter presented the proposed approaches on predicting the application execution time of parallel applications. We were able to define a taxonomy of the approaches into a classification framework. The framework is divided into two main categories. The first category involves the prediction domain, which denotes the systems and the type of applications where performance prediction has been applied.

An advantage of our classification structure is that we are able to map a proposal to both categories; application domain category and the prediction method category. Our classification framework makes it clear that runtime prediction has been applied to a wide-range of application domain areas and that a wide-range of prediction methods has been developed.

However, we found many approaches are application-dependent. For instance, there is a lack of prediction methods focusing on a specific HPC application domain whereby a

wider range of applications is covered. More over, there is a lack of platform-independent methods able to predict execution time on unseen multi-core and many-core platforms.

3. Method for generating a platform-independent prediction model

3.1. Introduction

This chapter describes a method to design a platform-independent prediction model based on classification trees. At the beginning of this chapter, it is defined the phases of the method process. Later, these phases are detailed, and it is given an example about how the method is applied.

3.2. Overall approach

There exist different approaches to generate prediction models, which can be classified into two main categories: exact methods [82] and approximate methods [40]. Exact methods include Statistic Mathematical Methods (SMMs) that are commonly used to predict the behaviour or future values of a system; among them the most commonly used method is simple linear regression. However, in certain cases finding a model with SMMs that fits to the behaviour of a given system can become a difficult task; in the case a model is found, it could derive in a computational complex model. This is a common case in distributed systems, which usually present non-linear behaviours. In fact, the performance of multicore systems exhibits non-linear behaviour. That is, the rate at which an application decreases its execution time is not proportional to the increment in the number of cores used to run the application. The reason is a number of resources used by the cores are shared such as the LLCs (last-level caches, e.g. L2 or L3), FSB (front side bus), memory controller, and I/O controller.

Besides exact methods, there are approximate techniques such as machine learning [55], which also allow us to analyse and predict the performance of a given system [22, 88, 50] and are generally computational simpler. Among machine learning techniques, a classification tree [118] is a hierarchical structure based on the dividing and conquer strategy which can be used for classification and regression [75]. A classification tree separates the entities contained within a dataset into several segments called classes.

Classification trees have been applied to different prediction problems. For example, the authors in [117] predicted a disease occurrence in an specific area, in [36] estimated the passage of drug molecules through blood-brain barrier to reach the nervous system, in [49] the significant activity of avalanche in an specific area, in [81] predicted a dementia disease in people with cognitive complaints, in [74] predicted hard disk failures considering the information provided by Self-monitoring Analysis and Reporting Tech-

nology (SMART), finally in [72] and [83] proposed two prediction methods focused on predicting the resource usage in a multi-core environment.

Our method to design a performance prediction model is based on a classification tree and involves five phases, which are depicted in Figure 3.1. A brief description about the goal of each phase is given below.

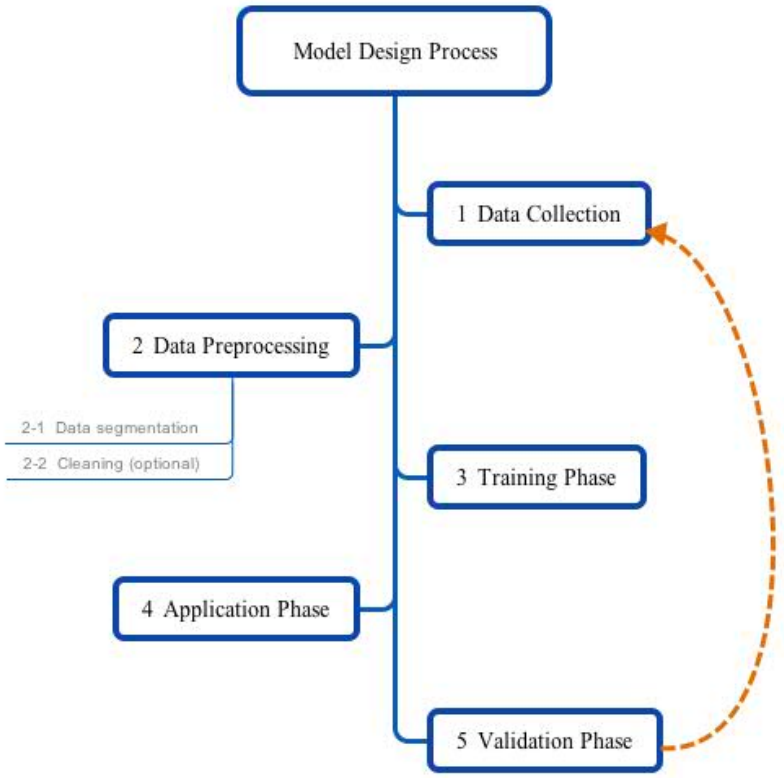

Figure 3.1.: Method to design a performance prediction model

Data collection. Data collection is the process of measuring and gathering data involving the performance behaviour of the application that we want to predict its execution time on unknown platforms. A dataset with the application execution time on different platform configurations is produced as output of this stage.

Data Preprocessing. Before a machine learning algorithm is used, in some cases, it is necessary to apply a pre-processing step. The pre-processing consists on transforming (e.g. segmenting or grouping the execution time in time intervals), cleaning (e.g. removing outliers), or normalizing the data collected, to name some data pre-processing processes. The input data of this stage is the collected data. Then, the output of this stage involves a dataset with the data segmented. However in some cases, it is necessary to remove outliers before segmenting the data.

Training phase. In this stage the pre-processed collected data is received as input and a classification tree (i.e. the performance prediction model) is produced as output. For this purpose, the training algorithm decides how the variables are selected to be weighted and combined to identify the classes associated with the instances[1].

Application phase. The input of this stage is the classification tree and the original dataset of the collected data (the collected data without any segmentation). The classification tree determines the classes of each instance in the dataset. The output is a new dataset containing the estimated class for each instance.

Validation phase. This stage determines the accuracy of the classification tree. This stage receives the new dataset as input; it compares the estimated class with its real class value. After comparing all the values, it is possible to estimate the percentage of accuracy.

It is important to note the loop back between step 5 (Accuracy validation) and step 1 (data collection). This loop back is used when the prediction model does not satisfy the accuracy established. In this case, it may be, for example, necessary to collect and add more instances to the dataset, or applying the cleaning process. This loop back process makes the classification algorithm to learn more about the application performance. This process would be repeated until the model achieves the desired prediction accuracy.

The steps of the method are more accurately defined in Algorithm 1.

Algorithm 1 Steps to generate the prediction model

```
1: if exist (model) then
2:     Λ ← load(model)                          ▷ Λ ⇒ tree model
3: else
4:     classes ← numClasses                     ▷ classes ≥ 2
5:     repeat
6:         R − ds ← Data collection
7:         discreteRuntime ← discrete(classes, R − ds)
8:         kfold ← 10
9:         Λ ← j48(discreteRuntime, kfold)
10:        pctAccuracy ← validate(R − ds)
11:    until pctAccuracy ≥ 0.85
12:    save(Λ)
13: end if
14: newClass ← findInstance(Λ, μ, ν, ω, φ, β)
15: τ_app ← determineInterval(newClass)
```

In case there exists a prediction model for the application, such a model is loaded and used to predict the application execution (steps 1 and 2), otherwise, the next steps

[1] An instance is a set of variable values (i.e. variables of the system) corresponding to the record measured, which represent the concept to be learned [16]. A dataset consists of a group of instances.

are followed to generate the prediction model. It is defined the number of segments or classes in which the application execution time will be discretized (step 4). Then, in the data collection phase, it is provided a dataset of the application performance behaviour on different platform scenarios (Step 6). The data pre-processing phase requires the collected data, and the number of classes in which the application execution time will be segmented (Step 7). This step calls the "discrete" function, which is responsible for segmenting the dataset according to the number of classes defined by the user. This function is detailed in Section 3.5.

The training phase follows when the j48 algorithm is called (Step 9). This algorithm receives two parameters, the number of k-folds to segment the data, and the segmented dataset. The j48 algorithm is an implementation of the C4.5 algorithm that is used to generate a classification tree, and is implemented in the Java programming language [8, 124]. We use an implementation of the algorithm included in the Waikato Environment for Knowledge analysis (WEKA) tool [9] (step 9).

In the validation phase, the "validate" function is used to determine the classes for a group of instances and to compare the estimated class with the real class value (Step 10). This function returns the percentage of accuracy. This function is detailed in section 3.8. The calculated percentage is compared with the desired percentage of accuracy (step 11). While this condition is not met, the design process is repeated back from step 5. Once the condition is satisfied, the classification model is saved for its future use (Step 12). Steps from 5 to 12 represent the method to design the prediction model.

Afterwards, the produced prediction model can be employed for predicting the execution time of an application on unknown platforms (Steps 14 and 15). An instance, which contains information about the resource characteristics $(\mu, \nu, \omega, \varphi, \beta)$ of the unknown platform, is introduced to estimate its corresponding class (Step 14), where the resources characteristics represent the number of cores, number of nodes, CPU frequency, memory, and network bandwidth, respectively. The estimated class has associated an interval value of the predicted execution time. The interval of the execution time associated with the estimated class is returned (Step 15).

3.3. Case Example

We use the Weather Research and Forecasting model (WRF) [85] to illustrate the method. WRF is a cpu- and communication-intensive application; it forecasts the weather in a selected geographical region. The input data needed by WRF is a region of interest (ROI). This ROI can vary according the region selected by the user. We are aware that the input data of each application has influence in the application execution time. Although, there are approaches that focus on developing prediction methods based on input data [88, 26], our approach lies outside such kind of methods. Therefore, the input data of the WRF model remained constant. The input data for the WRF model was fixed to 24 hours of land surface of 5625 Km^2 of the data obtained on March 1st, 2012.

It is important to note that the experiments were carried out in a free-workload

environment. As we established on early chapters, our prediction approach does not focus on predicting the application execution time in workload systems.

The following sections describe how the method is used to design a performance prediction model of WRF. However, the same process can be applied to design a prediction model of any other scientific application.

3.4. Data collection

Predicting a future value of an event, in our case the application execution time, requires having previous values of such an event. Therefore, it is necessary to record the execution time of an application in order to estimate its value on different platforms. This data collection allows us to generate a performance profile of the application in the base system, which leads to estimate its performance on an unknown target platform.

In the area of data mining, each independent data recorded is known as an instance. Henceforth, each record of the application execution time is refereed to as an instance. An instance is composed of the values of the dependent variable (application execution time) and independent variables (core frequency, number of cores, number of nodes, network bandwidth and system's memory).

In order to collect the data we require running the HPC application on different platform scenarios i.e. different CPU clock operation frequency, number of cores, nodes, memory, and network bandwidth. Machine learning techniques learn from the data provided. Then, it is necessary to provide enough information so that the classification tree learns different patterns. In case of providing a small number of platform scenarios, the model might not be able to generalize for unknown data. Consequently, we decided to use as many as possible platform scenarios. In any case, the validation phase determines if the collected data is sufficient. For our case example we used the platform scenarios depicted in Table 3.1.

Now, for each platform scenario, we need to perform a run of the HPC application and obtain its execution time for each possible combination of the number of cores, number of processors, and number of nodes. We only take into account those combinations whose performance behaviour is considerable different from each other. To achieve this, we follow a specific pattern of combinations, which involves two steps. In the first step, we combine n number of cores per m processors within a single node where q is the number of cores that a processor posses, and where $n \geq 1$, $m \geq 1$, and $q \geq 2$.

In the case we have n cores, m processors, $n \geq m$, and $q \geq 2$, there are $m + 1$ combinations. For $n = 2$, we use three combinations; the first one uses core 0 and 1 of processor 1, the second one uses core 0 and 1 of processor 2, and the third one uses core 0 of processor 1 and core 0 of processor 2. Hence, for n cores we have that the first combination uses core 0, core 1, core 2, ..., and core n of processor 1. The second combination uses core 0, core 1, core 2, ..., and core n of processor 2. The *m-th* combination uses core 0, core 1, core 2, ..., and core n of processor m. Given that $Q(y, z)$ gives the quotient of $\frac{y}{z}$, the $m + 1$ combination uses core 0, core 1, core 2,..., and core $Q(\frac{n}{m})$ of processor 1; core 0, core 1, core 2, ..., and core $Q(\frac{n}{m})$ of processor

Table 3.1.: Hardware configurations for WRF

Platform Scenarios	CPU frequency	Network bandwidth	Memory	Number of Cores	Number of Nodes
PS1	1.60 GHz	100 Mbps	16GB	24	2
PS2	1.60 GHZ	40 Mbps	16GB	8	1
PS3	1.60 GHZ	60 Mbps	16GB	8	1
PS4	1.73 GHz	100 Mbps	16GB	24	3
PS5	1.73 GHz	60 Mbps	16GB	8	1
PS6	2.00 GHz	60 Mbps	16GB	8	1
PS7	2.00 GHz	80 Mbps	16GB	8	1
PS8	2.00 GHz	100 Mbps	16GB	24	3
PS9	2.13 GHz	100 Mbps	16 GB	24	3
PS10	1.20 GHz	1000 Mbps	32 GB	32	2
PS11	1.40 GHz	1000 Mbps	32 GB	32	2
PS12	1.60 GHz	1000 Mbps	32 GB	32	2
PS13	1.80 GHz	1000 Mbps	32 GB	32	2
PS14	1.30 GHz	40000 Mbps	32 GB	32	2
PS15	1.50 GHz	40000 Mbps	32 GB	32	2
PS16	1.90 GHz	40000 Mbps	32 GB	32	2

2, ..., and core 0, core 1, core 2, ..., and core $Q(\frac{n}{m})$ of processor m if the residue of $n \div m = 0$, otherwise we use up to core $Q(\frac{n}{m}) + 1$ of processor m.

In the second step, we combine different number of nodes. For this, we apply the same procedure described in the first step by first considering one node, then two nodes, then tree nodes and so on.

We did not consider using other core combinations within the same processor because the execution time of any two different cores on the same processor is quite similar. This is because our experiments showed a coefficient of variation[2] of only 0.15% in the Quad-core processor and 0.37% in the Eight-core processor. However, we found that the execution time achieved by any two processors within the same node is considerably different in the case of the Quad-core processor where we obtained a coefficient of variation 5.30%. For this reason, step one considers combining the processors within the same node. Although, in the case of the Eight-core processor we obtained a coefficient of variation of only 0.43%, step one considers performing the combination of processors regardless of the number cores per processor. This does not have negative implications on the data collected but only might include additional data that is not strictly necessary.

Table 3.2 defines an example of the combinations of cores, processors, and nodes that we defined in our case example for the platform scenario 1.

[2]The coefficient of variation determines the ratio variation of the standard deviation respect to the media value. A higher value represents higher heterogeneity in the data, a lower value indicates higher homogeneity.

Table 3.2.: Example of combinations for four cores, processors, and nodes in the case of Platform scenario 1.

	Node 1								Node 2							
Number of	Processor 0				Processor 1				Processor 0				Processor 1			
Cores	C0	C1	C2	C3	C0	C1	C2	C3	C0	C1	C2	C3	C0	C1	C2	C3
1	x															
1					x											
1									x							
1													x			
2	x	x														
2	x				x											
2					x	x										
2	x								x							
2	x												x			
2					x								x			
3	x	x	x													
3			x	x	x											
3					x	x	x									
3	x	x							x							
3	x								x	x						
3					x				x	x						
4	x	x	x	x												
4			x	x	x	x										
4					x	x	x	x								
4	x	x							x	x						
4	x	x											x	x		
4					x	x							x	x		

After applying the data collection process, we obtained a dataset. Table 3.3 presents an extract of such a dataset.

3.5. Data pre-processing

Data pre-processing is an important process in machine learning techniques. Data pre-processing involves segmenting, or cleaning the dataset. In some cases, the data collected need to be adapted before applying the processing method. In our approach, the pre-processing technique relies on segmenting the application execution time from a continuous domain to a discrete domain. This process is based on the procedure to build a histogram, this is presented in Algorithm 2.

The segmentation process receives a dataset and the number of classes defined by the user. We performed a number of experiments to find the most suitable number of class, which happens to be eight (see Appendix A). It is necessary to sort in ascending order the collected execution times (Step 2). Then, it is determined the maximum and minimum value of the execution time (Steps 3 and 4). The amplitude of segments is calculated by subtracting the minimum value from the maximum value and dividing the result by the number of classes (i.e. eight) (Step 5). The following steps involve defining the lower and upper threshold of the first segment of data (Step 6 and 7). An identifier of the first segment is defined in Step 8. Then it is compared the execution time of every single instance, beginning from the execution time with its lower value,

Table 3.3.: Samples of the data collected for the execution of the WRF application in different Platform scenario

Cores	Nodes	CPU frequency (GHz)	Memory (Bytes)	Network bandwidth (bps)	Time (seconds)
5	2	2.13	1.6E+10	100000000	407.521
6	1	2	1.6E+10	60000000	407.839
6	1	2	1.6E+10	60000000	407.925
1	1	2	1.6E+10	80000000	1496.768
1	1	1.73	1.6E+10	100000000	1645.357
1	1	1.73	1.6E+10	100000000	1647.78
2	2	1.6	1.6E+10	100000000	1014.489
2	2	1.6	1.6E+10	100000000	1016.055
2	2	1.6	1.6E+10	100000000	1017.845
1	1	2.13	1.6E+10	100000000	1378.406

Algorithm 2 Data segmentation process (Step 7 from Algorithm 1)

1: **procedure** DISCRETE(classes, $R - ds$)
2: $runtime \leftarrow sort(runtime, asc)$
3: $minVal \leftarrow min(runtime)$
4: $maxVal \leftarrow max(runtime)$
5: $amplitude \leftarrow \frac{maxVal - minVal}{classes}$
6: $lowThreshold \leftarrow minVal$
7: $upThreshold \leftarrow minVal + amplitude$
8: $numClass \leftarrow 0$
9: **for** each $instance$ in $R - ds$ **do**
10: **if** $(instance \geq lowThreshold)$ & $(instance < upThreshold)$ **then**
11: $instance \Leftarrow class.numClass$
12: **else**
13: $numClass \leftarrow numClass + 1$
14: $lowThreshold \leftarrow upThreshold$
15: $upThreshold \leftarrow lowThreshold + amplitude$
16: **end if**
17: **end for**
18: **end procedure**

that fits within the amplitude of the segment (Step10). When the instance fits within the interval, it is assigned the label of the associated class (Step 11). Otherwise, the identifier is changed by increasing one value (i.e. the first identifier is class0, the second one is class1, the third one is class2, etc.), and the lower, and upper threshold is updated to the new segment (Steps 13, 14, and 15). Steps 10 to 16 are repeated until all instance are assigned a label (i.e. a class).

In our case example, the segmentation process considering 8 classes is performed as follows. The dataset instances are sort, then the minimum (239.564 sec), maximum (1790.294 sec) and amplitude (194.955 sec) of the execution time are determined. Table 3.4 shows the class label and its corresponding period of time.

Table 3.4.: Amplitude of the intervals for a segmentation process using 8 classes

Label	Threshold (seconds) Lower	Upper
Extremely fast	230.65399	425.559
Too Fast	426.044	618.262
Very Fast	620.789	809.619
Fast	816.987	995.666
Slow	1006.047	1017.845
Very Slow	1378.406	1399.145
Too Slow	1400.248	1484.386
Extremely slow	1645.357	1790.294

Another pre-processing process is to reduce noise on the collected data. Noise is usually injected during the data collection process. Noise may affect the efficiency of the machine learning technique [132]. Hence, it is necessary to apply a pre-processing method to reduce the noise. The cleaning process is focused on reducing the outliers of the execution time for a specific number of cores. Algorithm 3 presents the cleaning process.

The cleaning process receives a dataset. It is a repetitive procedure that needs to be applied to any set of core combinations. It is necessary to select the execution time for a defined number of Cores from the dataset (Step 2). The following steps determine the central tendency measurement of the execution time (Steps 3 to 7). Step 8 compares the estimated range (rng) with the maximum range permitted (rngMax). rngMax is left for user consideration, there is not a specific value, because it changes on every core combination selected. When this condition is satisfied, it is determined a new range value (newRng), which is half the range (Step 9). The new maximum upper and lower thresholds are defined (Step 10 and 11). Finally, the execution time above of the upper threshold (half+), and the execution time below of the lower threshold (half-) are excluded from the dataset. This process reduces the outliers on every core combination.

It is not commonly required to apply the cleaning process to all HPC applications. It is only applied to applications that present noise in its data collection. Because of that,

Algorithm 3 Cleaning process algorithm

Require: Execution time dataset ▷ $R - ds$

for each Core (μ) combination **do**
  $time \leftarrow select(R - ds, \mu)$
  $minVal \leftarrow min(time)$
  $maxVal \leftarrow max(time)$
  $avg \leftarrow average(time)$
  $std \leftarrow stdDev(time)$
  $rng \leftarrow range(time)$
  if $rng \geq rngMax$ **then**
    $newRng \leftarrow \frac{rng*0.5}{2}$
    $half+ \leftarrow avg + newRng$
    $half- \leftarrow avg - newRng$
    exclude execution time above $half+$
    exclude execution time below $half-$
  end if
end for

this process is not part of the base process defined in Algorithm 1.

In our case example, WRF did not require this process (this process was only applied to the Octopus application). The following paragraph gives an example about how the process was performed to the Octopus application.

Octopus does not have a homogeneous behaviour when this application is executed with a specific combination of Cores. This is due to the way the programming code of Octopus performs parallelisation. Therefore it was necessary to reduce outliers from the collected data set. Table 3.5 shows the values of the execution time for a different number of Cores for the Octopus application.

Table 3.5.: Example of values for the cleaning process of Octopus

	5 Cores	6 Cores	9 Cores
Min	702.778	979.094	850.011
Max	1585.756	1900.562	1354.366
Median	1210.184	1274.128	1017.620
Average	1252.611	1257.738	1072.874
Std. Dev.	245.222	226.059	143.398
Range	882.978	921.468	504.355
Range/2	220.745	230.367	126.089
Half+	1473.356	1488.105	1198.962
Half-	1031.867	1027.371	946.785

Table 3.5 has the estimated values of the cleaning algorithm. However, as we said

earlier, it is up to the user to decide from which Core combination outliers will be removed. For example, the execution of Octopus with 5 cores, has a standard deviation of 16.41 and a range of 30.10 sec., then it is not necessary to remove outliers, because we consider there are not significant differences between the execution times. But, when we execute the application with combinations using 6 Cores, the range reaches a difference of 204.68 seconds; in this case, it is necessary to remove outliers. Then, the instances with an execution time above of 1185.22 seconds, and below of 1170.17 seconds are removed from the dataset. The same procedure is applied with combinations using 9 Cores. In summary, the cleaning process is only applied when the dataset has noise because it affects the accuracy of the prediction model.

3.6. Training phase

In this stage the training algorithm decides how the independent variables are selected to be weighted and combined to identify the classes associated with the instances. For this purpose, the classification algorithm is trained using the segmented dataset. The algorithm extracts valid patterns from the data. These patterns generate the classification model by evaluating each independent variable to determine its weight into the model.

Training or designing a machine learning algorithm requires three internal stages: training, testing, and validation. These stages make the algorithm learn the application behaviour. After learning this behaviour, the algorithm is able to generalise it for any new input. Therefore, in this internal process, it is necessary to divide the dataset for the internal training process. Two main techniques are widely used to divide and train machine-learning algorithms: bootstrap and cross-validation. Bootstrap is a method that relies on random sampling with replacement [23] whereas cross-validation divides the training dataset into several mutually exclusive data portions [14]. However, the cross-validation technique has become the most suitable method to train machine-learning algorithms [138].

There are three representative cross-validation methods [93]: holdout, k-fold, and leave-one-out. Holdout divides the data set into two subsets, training, and testing dataset. The K-fold method divides the data set into k subsets (k-folds), one of the subsets is used as the testing dataset, the remaining subsets are used as training, and this process is repeated k iterations. Finally, leave-one-out is a special case of the k-fold method, where one instance is used as testing data on each iteration; the remaining instances are used for training. K-fold and leave-one-out are applied according to the dataset size. The Leave-one-out method is usually used on small datasets, unlike the k-fold that is used on large datasets [128].

We assume that our dataset is not small, and because the k-fold validation method has shown a slightly improvement of the results over the other methods [128], we decided to use the k-fold validation method. It has been shown that using a 10-fold value is the most optimal value to train the algorithm [125].

Finally, the classification model was designed by using the j48 algorithm. The j48

algorithm is an implementation of the C4.5 algorithm, which is written in the Java programming language [8, 124]. C4.5 has been listed in the top ten most influential data mining algorithms [131], and it has been taken as a benchmark reference for developing new approaches [104, 48]. This algorithm is provided by the Waikato Environment for Knowledge analysis (WEKA) tool [9]. The process described here relies on the algorithm provided by Weka.

In our case example, we use the segmented dataset as input to the j48 classification algorithm. Then, Weka applies the three internal stages and generates the classification tree model. Part of the tree generated for WRF is depicted in Figure 3.2.

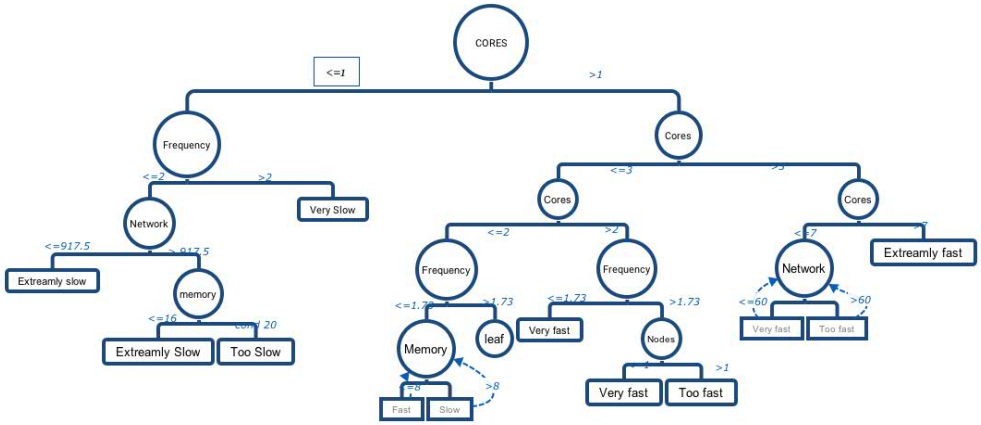

Figure 3.2.: Part of the classification tree generated by the j48 algorithm for WRF.

The elements of the classification tree of WRF are described as follows. A classification tree has a root node, internal nodes (identified by circles), and leafs or terminal nodes (identified by rectangles). Every node has a label, and outgoings or incoming arcs. In case of the root node, it only has outgoing arcs, this node denotes where the model begins. The label in the node indicates the attribute that is used with conditional statements. The root node is labelled as *"CORES"* and represents the number of cores of a platform scenario. Hence, the conditional statements define the path to follow across the classification tree. An internal node on the other hand, has an incoming arc, and two outgoing arcs. An internal node here connects to either another internal node or a leaf. A leaf is a terminal node and has only one incoming arc and no outgoing arcs. When we reach a leaf, this indicates that we reached the target value.

3.7. Application phase

The application phase of the prediction model determines the classes for a set of unknown instances. The goal of this stage is applying the prediction model generated in the previous phase so that it can be validated in the next phase (i.e. the validation phase). This step uses the same dataset of the training stage, but without the segmentation

process (i.e. the original collected data). Then, the classification tree estimates the corresponding class of each instance.

We are able to use the "same" dataset in both phases, given that the cross-validation method is designed to avoid learning just the pattern provided by the training dataset. This, given that a different subgroup (i.e. a fold) of data is used on every iteration during the training phase. This process makes the algorithm to consider this data as if it came from a new independent dataset. After completing this stage, we have a new dataset with a new estimated class for each instance. This dataset is then used in the validation phase.

3.8. Validation phase

The validation phase uses the new dataset (obtained from the previous stage) with the estimated classes for each execution instance. The process on this stage is to compare the estimated class with the class defined in the pre-processing phase. Algorithm 4 presents the process of estimating the accuracy of the prediction model.

Algorithm 4 Validation process algorithm

Require: Segmented dataset ▷ $R - ds$
Require: Estimated dataset ▷ $E - ds$
1: **for** each instance in $R - ds$ **do**
2: $estClass \leftarrow selectClass(E - ds, \mu, \nu, \omega, \varphi, \beta)$
3: $realClass \leftarrow selectClass(R - ds, \mu, \nu, \omega, \varphi, \beta)$
4: $confusionMatrix \leftarrow compare(estClass, realClass)$
5: **end for**

The algorithm requires two datasets, one with the real class assigned (determined by the segmentation process), and the estimated class (determined in the application phase). Then, a repetitive process is applied for all the instances in both datasets (Step 1 to 5). It selects the class for a specific instance in the estimated class, and the segmented dataset (Step 2, and 3). After selecting the class, it compares the estimated class with the real one (Step 4). The result is represented by a confusion matrix, which shows the instances that have been estimated correctly.

The validation phase for our case example using the WRF model gave us the Confusion Matrix presented in Table 3.6.

The information presented in the confusion matrix indicates that from the 998 with a label extremely fast, 969 instances were classified correctly, and 29 were misclassified as too fast. Too fast has 295 instances, 37 were misclassified as extremely fast, 5 misclassified as Very fast, and 253 were correctly classified. All the instances with the slow class were classified correctly.

The information provided can be resumed in percentage of accuracy. We found out, that 1458 instances were classified correctly (92.9847%), and 110 were classified incorrectly (7.0153%). The results obtained indicates that our model exceeded the desired

Table 3.6.: Confusion Matrix of estimated classes by the prediction model for WRF

a	b	c	d	e	f	g	h	⇐Classified as
969	29	0	0	0	0	0	0	a =Extreamly fast
37	253	5	0	0	0	0	0	b =Too fast
0	23	32	2	0	0	0	0	c =Very fast
0	0	0	36	10	0	0	0	d =Fast
0	0	0	0	20	0	0	0	e =Slow
0	0	0	0	0	32	0	0	f =Very Slow
0	0	0	0	0	4	44	0	g =Too slow
0	0	0	0	0	0	0	72	h = Extreamly Slow

percentage of accuracy. Hence, the model is able to predict the application execution time on unseen platforms.

3.9. Using the Generated Performance Prediction Model

Finally, after designing the prediction model, we are able to find an instance and determine the interval value of the predicted execution time. Two functions are defined in the Steps 14, and 15 of Algorithm 1, namely findInstance($\Lambda, \mu, \nu, \omega, \varphi, \beta$) and determineInterval(class). The findInstance() function uses the defined model (Λ), and the instance with the information about the resource characteristics ($\mu, \nu, \omega, \varphi, \beta$) that describes the hardware configuration of the unknown platform. Then, this function walks through the tree by following the attributes and conditional statements defined on every node in the tree. When this function finds the terminal node or leaf, the function returns its label. For example, given the tree depicted in Figure 3.2, when the parameters $\mu = 3$, $\nu = 1$,$\omega = 1.73GHz$,$\varphi = 16GB$, and $\beta = 100Mbps$ (representing the number of cores, number of nodes, CPU frequency, memory, and network bandwidth, respectively) are given as input, the returned class for these parameters happens to be "Slow". After finding the class, the determineInterval() function determines the interval values of this class (which was defined during the segmentation process). As a result, the corresponding interval of the execution time for the class "Slow" is obtained:

$$\tau_{app} = [1010.474, 1205.429] seconds.$$

Therefore the model predicts an execution time between 1010 and 1205 seconds of the WRF model in the unknown platform.

3.10. System Prototyping

The previous sections described the method to design a platform-independent prediction model based on classification trees. Our method to design a performance prediction model uses algorithms and tools provided by Weka.

However, our goal is to determine the application execution time on unseen platforms, instead of only determining its corresponding class, which is considered meaningful for our purpose. Thus, we developed a tool that fulfills the missing steps of Algorithm 1. More specifically, Weka is only able to generate a graphic representation of the model (i.e. a classification tree). But, Weka does not walk throught the classification tree to identify the class associated with a specific execution instance. Then our prototype focus on walking through any model designed, in order to make it easy to determine the class associated with an instance (Step 14). After finding the class, we are able to determine its corresponding upper and lower threshold that represents the estimated execution time of the application (Step 15).

This prototype is also focused on assisting the user during the design process of the performance prediction model. Hence, the prototype aids the user to carry out steps from 1 to 15 of Algorithm 1. However, we did not implement the segmentation, and cleaning processes. But, we focused on implementing the findInstance, and findInterval methods (Steps 14, and 15) of Algorithm 1.

Our prototyping was coded in the Java programming language. We decided to use this language, because Java allowed us to take advantage of the machine learning algorithms provided by the API of Weka.

4. Model Evaluation

4.1. Introduction

This chapter presents the results derived from evaluating the proposed method for deriving platform-independent prediction models. This chapter is organized as follows. Section 4.2 describes tools, system, and the applications used in the evaluation. Section 4.3 shows the accuracy of the prediction method in the design phase. The evaluation of the prediction model on unseen platforms is presented in Section 4.4. Section 4.5 gives a perspective about the performance of the tool support. This chapter ends with a discussion of the results in Section 4.6.

4.2. Description of system and cases of example

The experiments for evaluating the proposed method relied on two high performance clusters, both of them using Intel MPI 3.2.2, which is based on MPICH2.

The hardware specification of these systems are detailed in 4.1.

Table 4.1.: Hardware descriptions

	Clusters	
	Master	Zazil
CPU Model	Intel(R) Xeon(R) CPU E5506	Intel Xeon E5-2609 v2
Instruction set	64 bits	64 bits
Cores	4	8
Processors/Node	2	2
Processor frequency	2.3 GHz	2.0 GHz
RAM memory	16GB	32 GB
Network interconnection	Ethernet	Infiniband TruScale QDR / Ethernet
Network bandwidth	100 Mb/s	40 Gb/s /1 GB/s
Operating system	CentOS 6.4	CentOS 6.6
Number of nodes	8	4

Although, we only used two clusters, we were able to emulate different platforms scenarios by scaling the CPU clock frequency, and limiting the network bandwidth on those systems. To emulate the platform scenarios, we used two software tools: cpupower [1], and wondershaper [51]. cpupower is a tool that takes advantages of the Enhanced

INTEL Speed step Technology (EIST) on CPU processors, this technology is able to increase/decrease the frequency on every single core. On the other hand, wondershaper is a tool written by Bert Hubert, this tool provides among its features packet scheduling, rate limit, and QoS over the network connection. wondershaper takes advantages of the routing capabilities of Linux kernel.

The emulation allowed us to assess the application execution under different platform scenarios. Table 4.2 shows the values used for the platform scenarios.

Table 4.2.: Clock rate and network bandwidth values

	Clock rate (GHz)	Network Bandwidth (Mbps)
Master	1.63, 1.70, 2.00, 2.13	40, 60,80, 100
Zazil	1.20Ghz - 2.00Ghz	1000, 40000

The emulation of the scenarios was conducted by combining the CPU clock frequency, and network bandwidth. For example, in the cluster named Master, a scenario has the CPU frequency at 1.63GHz, and network bandwidth of 40Mbps. Another scenario, the same frequency (1.63GHz), and network bandwidth of 60Mbps. We tried to use the maximum number of combinations between those resources. It is important to note, that Zazil does not allow modifying the network bandwidth with wondershaper. Therefore, we use the network bandwidth provided by the network connections on the system (Ethernet and Infiniband).

We emulated 6 base platforms on the cluster Master, as shown in Table 4.3, for the data collection process. For instance BP1 involves a platform scenario emulation consisting of a CPU frequency of 1.60 GHz, a network bandwidth of 100 Mbps, 16 GB of memory, 16 cores, and 2 nodes.

Table 4.3.: Base platforms scenarios

Base Platforms	CPU frequency	Network bandwidth	Memory	Number of Cores	Number of Nodes
BP1	1.60 GHz	100 Mbps	16GB	16	2
BP2	1.73 GHz	100 Mbps	16GB	16	2
BP3	2.00 GHz	60 Mbps	16GB	8	1
BP4	2.00 GHz	80 Mbps	16GB	8	1
BP5	2.00 GHz	100 Mbps	16GB	16	2
BP6	2.13 GHz	100 Mbps	16 GB	16	2

In addition, we emulated 13 unseen platforms for evaluating the performance prediction models. With cluster Master were emulated 7 unseen platforms, namely UP1 to UP7. On cluster Zazil were emulated 6 unseen platforms, namely UP8 to UP13. These platforms are shown in Table 4.4

Three scientific applications were executed on the emulated unseen platforms. Those

Table 4.4.: Unseen platform scenarios

Unseen Platforms	CPU frequency	Network bandwidth	Memory	Number of Cores	Number of Nodes
UP1	1.60 GHZ	40 Mbps	16GB	8	1
UP2	1.60 GHZ	60 Mbps	16GB	8	1
UP3	1.60 GHz	100 Mbps	16GB	24	3
UP4	1.73 GHz	60 Mbps	16GB	8	1
UP5	1.73 GHz	100 Mbps	16GB	24	3
UP6	2.00 GHz	100 Mbps	16GB	24	3
UP7	2.13 GHz	100 Mbps	16 GB	24	3
UP8	1.30 GHz	40000 Mbps	32 GB	32	2
UP9	1.50 GHz	40000 Mbps	32 GB	32	2
UP10	1.90 GHz	40000 Mbps	32 GB	32	2
UP11	1.20 GHz	1000 Mbps	32 GB	32	2
UP12	1.40 GHz	1000 Mbps	32 GB	32	2
UP13	1.80 GHz	1000 Mbps	32 GB	32	2

applications were selected after analysing several applications listed by the HPC advisory Council [110]. The HPC advisory council is an institution focused on bridging the gap between high performance computing to any kind of institutions or users that require HPC systems. The HPC advisory council provides courses, conferences, guidelines for best practices for HPC systems, and connection with expertise professionals' on HPC area. Within the guidelines, the HPC council provides an analysis of benchmarks for scientific applications executed on HPC systems. After analysing the list of application, three HPC applications were selected for the purpose of evaluating our approach. These applications are the Weather Research and Forecasting model (WRF model) [85], Octopus [32], and miniFE [86].

These applications were selected based on the computational requirements (e.g. CPU, memory, or network bandwidth) needed to carry out its processing operations.

As mentioned previously, WRF is a cpu- and communication intensive application used to carry out weather prediction. We used WRF 3.4. The input data for WRF was fixed to 24 hours of land surface of 5625 Km2 with 20 km resolution, which contains 75000 x 75000 grid points of the data obtained on March 1st, 2012; see Chapter 3 further details on WRF. Octopus is also a cpu- and communication-intensive application, and it performs differential finite operations. The main purpose of Octopus is to simulate the electron-ion dynamics of one-, two-, and three-dimensional finite systems and it is based on time-dependent density-functional theory (TDDFT).

miniFE is a memory-intensive application. miniFe is a mini application to solve linear system operations using a conjugate-gradient algorithm, we used miniFE 1.1 The miniFE input data defines the dimension domain of the finite element. A dimension domain is defined by three parameters nx , ny , and nz, which for our experiments were set to 335,

347, and 347, respectively. Finally the Octopus, version 4.1.2, performs a simulation of the Benzene molecule. The energy units was defined in electronvolts, length unit in Armstrongs, the radius of the box shape is 8, and the space between points in the mesh was 0.10. Note that the input data of the applications remained constant in all the experiments.

These applications were executed on the platform scenarios by following the procedure of data collection described in Section 3.4. The data collection process allowed us to generate four mutually exclusive datasets for each application. One dataset represents the data collection process used to design the prediction models (one dataset per HPC application), as shown in Table 4.5. The other three datasets, Table 4.6, contain information of the execution times of the HPC applications on the unseen platforms (one dataset per platform), which were used to evaluate the accuracy of the prediction models.

Table 4.5.: Number of instances in each dataset of the base platforms

Dataset	Total Instances
DS1-WRF	1568
DS2-miniFE	1558
DS3-Octopus	1536

Table 4.6.: Number of instances in each dataset of the unseen platforms

Dataset	Total Instances
DS2-WRF	2,144
DS2-miniFE	1,967
DS2-Octopus	1,862

4.3. Validating the Accuracy of the Prediction model (design phase)

Although this procedure is carried out in the design phase, we present it as part of the evaluation of our approach given that this procedure gives an assessment of the generated prediction models. A prediction model is designed using 8 classes, as mentioned in Chapter 3. The prediction accuracy of the prediction model of each application is shown in the following paragraphs.

The accuracy of the prediction model of WRF is shown in Table 4.7, its confusion matrix in Table 4.8, and the class accuracy in Table 4.9.

Table 4.7.: Percentage accuracy of the WRF prediction model

	Instances	Pct.
Correctly classified	1458	92.9847%
Incorrectly classified	110	7.0153%

Table 4.8.: Confusion matrix of the WRF prediction model

a	b	c	d	e	f	g	h	⇐Classified as
969	29	0	0	0	0	0	0	a=Extremely fast
37	253	5	0	0	0	0	0	b=Too fast
0	23	32	2	0	0	0	0	c=Very fast
0	0	0	36	10	0	0	0	d=Fast
0	0	0	0	20	0	0	0	e=Slow
0	0	0	0	0	32	0	0	f=Very Slow
0	0	0	0	0	4	44	0	g=Too slow
0	0	0	0	0	0	0	72	h=Extremely Slow

Table 4.9.: Classification accuracy by class of the WRF prediction model

	Error
Class	percentage
Extremely fast	3%
Too fast	14%
Very fast	**44%**
Fast	22%
Slow	0%
Very Slow	0%
Too slow	8%
Extremely Slow	0%

The prediction model of WRF application has an accuracy of 92.98%, marked in bold, in the design phase. The confusion matrix of the prediction model shows that the highest error of classification is in the class labelled "Very fast" with an error of 44%. The remaining classes exhibit an error below 22%.

The following results are for the prediction model designed for miniFE. The classification accuracy is presented in Table 4.10, its corresponding confusion matrix in Table 4.11, and the percentage of class accuracy in Table 4.12.

Table 4.10.: Percentage of accuracy for the miniFE prediction model

	Instances	Pct.
Correctly classified	1404	**90.1155%**
Incorrectly classified	154	9.8845%

Table 4.11.: Confusion matrix for the miniFE prediction model

a	b	c	d	e	f	g	h	⇐Classified as
804	18	0	0	0	0	0	0	a=Extremely fast
60	386	8	0	0	0	0	0	b=Too fast
0	9	71	4	0	0	0	0	c=Very fast
0	0	4	27	5	0	0	0	d=Fast
0	0	0	8	2	0	0	0	e=Slow
0	0	0	0	0	31	4	0	f=Very Slow
0	0	0	0	0	1	28	17	g=Too slow
0	0	0	0	0	0	16	55	h=Extremely Slow

Table 4.12.: Classification accuracy by class of the miniFE prediction model

Class	Error percentage
Extremely fast	2%
Too fast	15%
Very fast	15%
Fast	25%
Slow	80%
Very Slow	11%
Too slow	39%
Extremely Slow	23%

The prediction model for the miniFE application has an accuracy of 90.11%, marked in bold in Table 4.10, the prediction model for miniFE misclassify 80% of the instances

of "Slow" class, followed by the "Too slow" class with 39% of prediction error, and the remaining classes has an error below 25%.

In case of Octopus, we present two results, the first one shows the results of the model without applying the cleaning process, the accuracy is shown in Table 4.13, its confusion matrix in Table 4.14, and the classification accuracy by class in Table 4.15.

Table 4.13.: Percentage of accuracy for the Octopus prediction model before removing outliers

	Instances	Pct.
Correctly classified	1117	72.7214%
Incorrectly classified	419	27.2786%

Table 4.14.: Confusion matrix for the Octopus prediction model before applying the cleaning process

a	b	c	d	e	f	g	h	⇐Classified as
360	171	0	0	0	0	0	0	a=Extremely fast
113	515	11	0	0	0	0	0	b=Too fast
0	37	75	17	0	0	0	0	c=Very fast
0	0	14	43	1	0	0	0	d=Fast
0	0	6	11	0	4	0	0	e=Slow
0	0	0	2	4	67	1	0	f=Very Slow
0	0	0	0	0	7	57	0	g=Too slow
0	0	0	0	0	5	15	0	h=Extremely Slow

Octopus prediction model has a classification error of 28%, which exceeds our maximum error of 15%, we see that all the classes have a classification error above of 65%. The low accuracy of the model is because the dataset of Octopus has noise, we identified that this noise is due to the parallelization model used in the application, therefore it cannot be avoided. Hence, we decided to apply the cleaning process in this application to improve its accuracy.

After applying the cleaning process, the accuracy of prediction model is presented in Table 4.16. Table 4.17 presents the confusion matrix, and Table 4.18 the classification accuracy by class.

We can observe that after applying the cleaning process in the Octopus dataset, the accuracy was improved to 83.33%. There are two classes with classification issues: the "Slow" class misclassifies all the instances, and the "Very fast" class has an error of 63%. The remaining classes have an error below of 31%.

In summary, the accuracy of the prediction model for WRF and miniFE remain above of 90%. On the other hand, Octopus requires applying the cleaning process to reduce the outliers in the dataset. After applying the cleaning process, the prediction model

Table 4.15.: Classification accuracy by class of the Octopus prediction model

	Error
Class	percentage
Extremely fast	68%
Too fast	98%
Very fast	87%
Fast	98%
Slow	81%
Very Slow	99%
Too slow	100%
Extremely Slow	100%

Table 4.16.: Percentage of accuracy for the Octopus prediction model after applying the cleaning process

	Instances	Pct.
Correctly classified	865	83.3333%
Incorrectly classified	173	16.6667%

Table 4.17.: Confusion matrix for the Octopus prediction model after applying the cleaning process

a	b	c	d	e	f	g	h	⇐Classified as
176	61	0	0	0	0	0	0	a=Extremely fast
43	489	6	2	0	0	0	0	b=Too fast
0	20	19	13	0	0	0	0	c=Very fast
0	0	10	33	5	0	0	0	d=Fast
0	0	0	2	22	0	0	0	e=Slow
0	0	1	8	2	0	0	0	f=Very Slow
0	0	0	0	0	0	58	0	g=Too slow
0	0	0	0	0	0	0	68	h=Extremely Slow

Table 4.18.: Classification accuracy by class of the Octopus prediction model

	Error
Class	percentage
Extremely fast	26%
Too fast	9%
Very fast	63%
Fast	31%
Slow	8%
Very Slow	100%
Too slow	0%
Extremely Slow	0%

improves its accuracy to 83.33%. Although, the prediction accuracy of Octopus does not reach the 85% established by our hypothesis, we decided to use this model to evaluate the prediction model on unseen platforms.

4.4. Evaluation of the prediction models on unseen platforms

We showed in the previous section that prediction models had positive results during the design phase. Therefore, we evaluated the accuracy of those models on unseen platforms. Our research goal is to predict the application execution time on unseen platforms, without previously making either a partial or complete application execution on the target platform, this with an error below of 15%. However, we made executions on the target platform to compare the estimated classification with the real one.

Thirteen unseen platforms (UP1 to UP13) were considered for the WRF model. The accuracy of WRF prediction model is shown in Table 4.19.

A compendium of the predictions of WRF on the 13 unseen platforms is shown in the confusion matrix in Table 4.20 as well as the percentage error by class, shown in Table 4.21.

We can see that the classification model of WRF has an overall accuracy above of 90%, as shown in Table4.22. There is only one platform with low accuracy (UP1), which obtained 69.44%. The remaining platforms had an accuracy above of 83%.

The confusion matrix of UP1 shows that the dataset has 36 instances. 25 were classified correctly, and 11 were misclassified. From the ten instances within the "Very fast" class, five were misclassified as "Too fast", which represent 50% of the instances. The "Fast" class has 2 instances, and one instance was classified as "Very fast" (50% incorrectly), and the three instances belonging to the "Slow" class, were classified as "Fast" (100% incorrectly). The misclassification errors were due to the patterns learned by the classification tree during the design phase, in which more instances were needed for training better the classification tree. For example, the model did not learn properly how to classify instances in the "Very fast" class, this is corroborated by the validation

Table 4.19.: Accuracy of the prediction model of WRF in unseen platforms

Platform	Instances	Correctly Classify	Incorrectly Classify
UP1	36	69.4444%	30.5556%
UP2	36	83.3333%	16.6667%
UP3	116	91.3793%	8.6207%
UP4	36	83.3333%	16.6667%
UP5	116	100.0000%	0.0000%
UP6	116	93.1014%	6.8986%
UP7	116	96.5517%	3.4483%
UP8	262	87.0229%	12.9771%
UP9	262	95.8015%	4.1985%
UP10	262	93.5115%	6.4885%
UP11	262	90.8397%	9.1603%
UP12	262	92.7481%	7.2519%
UP13	262	90.4580%	9.5420%

Table 4.20.: Overall confusion matrix of the WRF

a	b	c	d	e	f	g	h	⇐Classified as
1591	19	0	0	0	0	0	0	a=Extremely fast
67	258	0	0	0	0	0	0	b=Too fast
0	35	65	20	0	0	0	0	c=Very fast
0	0	1	27	8	0	0	0	d=Fast
0	0	0	9	8	0	0	0	e=Slow
0	0	0	0	0	0	8	0	f=Very Slow
0	0	0	0	0	0	0	8	g=Too slow
0	0	0	0	0	0	0	20	h=Extremely Slow

Table 4.21.: Classification accuracy by class of WRF

Class	Error Percentage
Extremely fast	1%
Too fast	21%
Very fast	46%
Fast	25%
Slow	53%
Very Slow	100%
Too slow	100%
Extremely Slow	0%

Table 4.22.: Overall accuracy of WRF prediction model.

	Instances	Pct.
Correctly classified	1969	91.8377%
Incorrectly classified	175	8.1623%

process (carried out during the design phase) which shows WRF obtained 44% of error at classifying instances within the "Very fast" class. A possible way to address this issue is customising the cross-validation method to take into account the percentage error per class instead of the average of the percentage error of all the classes. Hence, the model designer would proceed including more instances to the collected dataset until the desired accuracy is achieved for each class.

When we analyse the confusion matrix (Table 4.20), and observe the classification error by class, we can observe that the model has some issues classifying the instances labelled as "Very slow", and "Too slow" with a misclassification error of 100%, followed by the "Slow" class with an error of 53%, and "Very fast" with 46%. The remaining classes have a classification error below of 25%. In AppendixC can be found detailed information of the confusion matrixes of WRF for the 13 unseen platforms.

Next, the prediction accuracy of miniFE in unseen platforms is shown in Table 4.23.

Table 4.23.: Accuracy of the prediction model of miniFE in unseen platforms

Platform	Instances	Correctly Classify	Incorrectly Classify
UP1	35	**60.0000%**	**40.0000%**
UP2	35	**60.0000%**	**40.0000%**
UP3	116	91.3793%	8.6207%
UP4	35	**77.1429%**	**22.8571%**
UP5	116	96.5517%	3.4483%
UP6	116	88.7931%	11.2069%
UP7	116	93.9655%	6.0345%
UP8	262	93.8931%	6.1069%
UP9	262	93.8931%	6.1069%
UP10	262	84.7328%	15.2672%
UP11	436	97.7064%	2.2936%
UP12	88	**75.0000%**	**25.0000%**
UP13	88	**61.3636%**	**38.6364%**

A compendium of the predictions of miniFE on the 13 unseen platforms is shown in the confusion matrix in Table 4.24 as well as the percentage error by class, shown in 4.25.

The prediction model for the miniFE application has an overall accuracy above of

Table 4.24.: Overall confusion matrix of the model for miniFE

a	b	c	d	e	f	g	h	⇐Classified as
1357	60	0	0	0	0	0	0	a=Extremely fast
51	264	38	0	0	0	0	0	b=Too fast
0	14	78	12	0	0	0	0	c=Very fast
0	0	8	43	0	0	0	0	d=Fast
0	0	0	9	0	0	4	0	e=Slow
0	0	0	0	0	0	4	0	f=Very Slow
0	0	0	0	0	0	0	8	g=Too slow
0	0	0	0	0	0	0	17	h=Extremely Slow

Table 4.25.: Classification accuracy by class of miniFE

Class	Error Percentage
Extremely fast	4%
Too fast	25%
Very fast	25%
Fast	16%
Slow	100%
Very Slow	100%
Too slow	100%
Extremely Slow	0%

Table 4.26.: Overall accuracy of miniFE prediction model

	Instances	Pct.
Correctly classified	1759	89.4255%
Incorrectly classified	208	10.5745%

89.4255%, as shown in Table 4.26. A detailed analysis of the accuracy in every single unseen platform shows that in five unseen platforms marked in bold (UP1, UP2, UP4, UP12, and UP13) the accuracy is below 78%. A visualisation of the confusion matrix and the Table of classification by class, we see that the prediction model is not able to classify correctly the instances "Slow", "Very slow", and "Too slow". In this classes, it is reached a 100% prediction error. The remaining classes have a prediction error equal or below of 25%.

A similar behaviour, such as in the case of the WRF prediction model with the unseen platform UP1, is presented for the prediction model of miniFE for the unseen platforms UP1, UP2, UP4, UP12. This behaviour is related to the learned pattern in the design phase, the classification tree was not able to determine correctly the class of the unknown instances.

In Appendix D can be found detailed information of the confusion matrixes of miniFE for the 13 unseen platforms.

The last results belong to the prediction model of Octopus. We follow the same pattern by presenting a summary of the classification accuracy on the target platforms in Table 4.27.

Table 4.27.: Accuracy of prediction model of Octopus on unseen platforms

Platform	Instances	Correctly Classify	Incorrectly Classify
UP1		NA	
UP2		NA	
UP3	116	38.7931%	61.2069%
UP4		NA	
UP5	116	23.2759%	76.7241%
UP6	116	18.9655%	81.0345%
UP7	116	50.8621%	49.1379%
UP8	262	**80.1527%**	**19.8473%**
UP9	262	79.3893%	20.6107%
UP10	262	72.5191%	27.4809%
UP11	262	**89.3130%**	**10.6870%**
UP12	262	**80.1527%**	**19.8473%**
UP13	88	59.0909%	40.9091%

A compendium of the predictions of Octopus on the 10 unseen platforms is shown in the confusion matrix in Table 4.28 as well as the percentage error by class, shown in Table 4.29.

Table 4.27 show that there are only three platforms with an accuracy above of 80% (UP8, UP11, and UP13). Two with an accuracy above 70%, and the remaining with an accuracy below of 60%.

In addition, the classification error of the classes show that there are only two classes

Table 4.28.: Confusion matrix of the model for Octopus

a	b	c	d	e	f	g	h	⇐Classified as
1035	113	2	0	0	0	0	0	a=Extremely fast
144	179	38	14	0	15	0	0	b=Too fast
43	54	10	12	12	35	0	0	c=Very fast
2	26	6	6	26	10	0	0	d=Fast
0	4	2	0	18	0	0	0	e=Slow
0	12	0	0	0	0	4	4	f=Very Slow
0	0	12	0	0	0	0	7	g=Too slow
0	0	0	8	0	0	0	9	h=Extremely Slow

Table 4.29.: Classification error by class of Octopus

	Error
Class	Percentage
Extremely fast	10%
Too fast	54%
Very fast	94%
Fast	92%
Slow	25%
Very Slow	100%
Too slow	100%
Extremely Slow	47%

Table 4.30.: Overall accuracy of the Octopus prediction model

	Instances	Pct.
Correctly classified	1257	67.5081%
Incorrectly classified	605	32.4919%

with a prediction error below of 25%, i.e. "Extremely fast", and "Slow". The remaining has an error above of 45%, reaching the maximum error of 100% in "Very slow", and "Too slow" classes.

In summary, the overall accuracy of the prediction model is 67.5081%. We believe that this low accuracy is related to the parallelization model used by this application, which is reflected in obtaining diverse execution times for the same resource configuration (see Appendix B). Also, in Appendix E can be found detailed information of the confusion matrixes of Octopus for the 10 unseen platforms.

4.5. Performance of tool support

An important aspect to consider is the time required to build a prediction model, and then the time taken to execute the prediction model. The following paragraphs give an overview about the time needed to generate a prediction model. As we defined, in Algorithm 1 of Chapter 3, our proposed method involves five steps to design a platform-independent prediction model for an HPC application; these steps are: data collection, data pre-processing, training phase, application phase, and validation phase. The data collection process is the most time-consuming process of our method. However, we believe this case is presented in most proposed models that involve a data collection process.

Data pre-processing is the next most time-consuming process. Data pre-processing consists of two steps, data segmentation, and data cleaning. We did not implement an automatic process for the segmentation and cleaning process. At the moment this process is carried out manually and takes around about 5 minutes, which is a simple procedure based on designing a histogram. Nevertheless, this process can be automated by developing a program that creates the segmentation as it were a histogram. Such a program would only take a few seconds to execute. Similarly, the cleaning process is currently a manual process. However, we believe that given the simplicity of the proposed method it can be generated an automated process, which we consider is not a time-consuming process. Therefore, an automated version of our proposed method would be able to clean the dataset in a few seconds.

The remaining steps training, application, and validation which are supported by the Weka tool and other tool support (developed as part of this thesis), require less than 0.10 seconds altogether. For example, the time taken to build the model for Octopus is 0.02 seconds, miniFE requires 0.07 seconds, and WRF requires 0.07 seconds. Finally, it also takes about 0.10 seconds to walk through a classification tree and obtain a prediction.

4.6. Discussion

The first topic of discussion is about the system, and applications used to conduct the experiments of this thesis. The architecture of the Clusters used is based on a non-uniform memory access architecture (NUMA), where each processor can access its own local memory. This kind of architecture is the most used on these kinds of clusters.

Another aspect to consider is the architecture of the system. Both clusters use an Intel architecture. Therefore the heterogeneity relies on the hardware configurations (i.e. different CPU speed, different amount of RAM, etc.). Which were emulated by setting the CPU frequency at different rate, and limiting the network bandwidth. On the other hand, we used three scientific applications to evaluate the proposed approach. These applications use the MPI programming parallel model. Then, we assume that our proposed prediction method can be used for any parallel application based on MPI. The MPI programming model is the facto parallel model for distributed memory systems [60].

The second topic to discuss is about the accuracy of the generated prediction models. We presented the accuracy of the models at the design stage. We showed that the prediction models for WRF, and miniFE have an accuracy above of 90%. However, Octopus had an accuracy of 72%. Therefore, it was necessary to apply the proposed cleaning process in order to remove the outliers and improve the accuracy. We demonstrated that our cleaning process improves 11% the accuracy of the prediction model after removing the outliers. However, this improvement did not reach our desired value. Despite the results obtained for the prediction model of Octopus, we decided to evaluate its performance on unseen platforms.

To assess the model precision on unseen platforms, it was necessary to execute the applications on the target systems. However, the information of these executions were not used during the design phase, showing that the prediction models do not need to make a complete nor a partial execution on the target system. Every application has its own prediction model, and they cannot be used for a different application. Besides, those models are dependent on the same input data. Therefore, when an application is executed with another input data, it is necessary to apply the prediction method to obtain a new application profile model for that input data. However, given that our proposed approach requires only a few seconds to generate a new prediction model, our approach is able to generate models for new input data, or new applications in a few seconds.

We demonstrated that our method is able to generate prediction models capable of obtaining an accuracy above of 85% for the applications miniFE and WRF. We also showed, that even though the prediction models were produced with data collected from the cluster Master, such models were able to predict, with the same accuracy, the performance of applications on unseen platforms emulated on Zazil, a cluster with a different hardware configuration but same hardware architecture i.e. Intel. It should be noted that these clusters have a different Intel microarchitecture, namely Nehalem (Master) and Sandy Bridge (Zazil). This demonstrates that the generated models can even be effective across different microarchitectures belonging to the same hardware architecture, this, when the microarchitectures are not radically different. In addition, we have pointed out that the low accuracy obtained for some of the unseen platforms (in case of WRF and miniFE) was due to patterns learned during the design phase which required more instances, and this can be addressed by customising the cross-validation method. This customisation would involve validating, at the design phase, that each of

the classes achieves the desired accuracy instead of validating only the average of the classes' accuracy.

On the other hand, the prediction model for Octopus reaches its maximum accuracy on two unseen platforms, UP11, and UP12, with of 89% and 80% respectively for the target system of Zazil with an Ethernet connection, followed by Zazil with Infiniband with 80%, 79% and 72% on UP8, UP9, and UP10, the remaining unseen platforms present and accuracy below of 60%, reaching its lower value of 18% in UP6. We believe that this low accuracy is due to the noise presented in the dataset. We analysed the behaviour of the execution time of the three applications in order to identify the cause of this low accuracy with Octopus. We observed that the execution time of both WRF and miniFE were consistent, i.e. the execution time of the WRF has a coefficient of variation of 4% when a hardware configuration uses two cores at 2.13GHz, and a network bandwidth of 100Mbps. That is, when we executed these two applications on a particular hardware configuration, all resource configurations considered had a similar execution time. In contrast, the execution times of Octopus were significantly different i.e. the execution time of Octopus has a coefficient of variation of 14% when we have a hardware configuration of two cores at 2.13GHz, and a network bandwidth of 100Mbps. This fact was reflected in the low accuracy of the prediction model. In summary, the prediction method was able to generate a prediction model with an accuracy above of 85%. However, it is necessary that the applications do not present significant variations in their execution time for the same resource configurations.

Finally, despite the time needed to carry out the manual steps, which can be automated by creating tool support, the design process of the prediction model requires less than 1 second to create a new model. Although, this research has focused on off-line analysis, we believe that this method can also be used to predict the execution time of HPC applications at run-time, which can help, for example, to take scheduling decisions.

5. Conclusions and future work

5.1. Introduction

This dissertation presented a method to design platform-independent performance models for HPC applications. We have mentioned that Institutions (research centres, schools, or private companies) require HPC systems to satisfy their business needs. Such institutions choose to apply a capacity planning process to determine their computing needs, but when the capacity planning process is not carried out properly, the institutions usually do not fulfil their HPC needs. Because of that prediction methods play an importance role in the capacity planning process.

Then, we carried on a systematic review to identify the most relevant papers that have been published in the field of performance prediction. We defined the period of time of the published literature between 2005 to September 2015. We were able to identify the most common methods, and resources used in this area. But, but we found that there is a lack of platform-independent methods able to predict execution time on unseen platforms without making either a partial or complete execution on the target platforms.

5.2. Summary of Thesis

As mentioned above the research reported in this thesis regards developing a method for designing performance prediction models for HPC applications. Chapter one presented a perspective about the necessity of performance prediction in HPC systems. Other topics included in this chapter were the foundations about the conduction of this research by defining the hypothesis, research questions, the goals, and scope of this research.

Chapter two presented an analysis of the reviewed literature focused on performance prediction. In this chapter was also presented a taxonomy of the prediction models. In this taxonomy were represented the domain and the prediction methods that have been developed.

Chapter three gives a full description about the proposed method to design a platform independent prediction model. Along with the details of each phase, an example was presented to illustrate the way in which the design process is carried out.

The content detailed in Chapter four presented the accuracy of the proposed method on unseen platforms. Besides the results, it was described the environment, tools and applications considered in this research. Finally, the chapter ends with an overall discussion about the outcomes of the results.

5.3. Answer to Research Questions

Four research questions were used as guideline to conduct our research. The following paragraphs details the answer to these questions.

- **What kinds of resources have an impact on the performance execution of long-running applications in multi-core systems?** We found that the most common resources to determine the application runtime are CPU, network, memory, and disk I/O. Although, most of the reviewed approaches did not make an analysis to determine the correlation between the resources and the application execution time, it has been set by Carrington in [17] that two resources (CPU, and network bandwidth) are necessary to estimate the application execution time on systems with single-core processors, and when another resources is added to the model, the prediction accuracy improves in 2%, but it adds complexity in the analysis of the system. An important aspect about the resources used on every reviewed proposal is that most of the proposals use different types of resources, which were chosen based on the application type or domain area. Consequently, there is not a consensus on the specific resource characteristics needed to determine the performance behaviour of an application.

- **How can the resources be modelled in order to generate a platform independent resource usage model for multi-core systems?** There are different kinds of resource representation on every model proposed. We found that most of the proposals define algebraic expressions and there are many ways in which the resources are modelled. This is because each proposal defines its own model representation according to the information that is employed by each approach to generate a prediction model.

- **What kind of regression technique would be more accurate to predict the application performance?** Most of the approaches use either linear or non-linear models, being linear models the most employed. In some cases the proposals apply a method for transforming a non-linear model to a linear model. This is because linear models are easier to compute than non-linear models since the former avoid the use of complex mathematical methods such as differential partial equations and numerical methods. However, when the model deals with non-linear system proposals opt to use different machine learning techniques. Machine learning techniques have presented similar or better accuracy than exact methods not only regarding performance prediction but in other application fields as well [69, 108, 122]. Therefore we believe that machine learning techniques can suitable to use as a prediction method for estimating the application execution on unseen platforms.

- **What are the steps of the method to design a performance prediction model?** We proposed a method based on a machine learning technique, i.e. classification trees, to design a platform-independent performance model of HPC

applications in multicore systems. Our approach assumes it is needed to produce a different model for a different application as well as for a different application input data. Our proposed method consists of 5 well-defined steps. Data collection, data pre-processing, training phase, application phase, and validation phase.

Data collection describes the process of gathering data from the base platforms. Data Pre-processing involves segmenting, or cleaning the dataset. The segmentation process relies on dividing the execution time in groups. This process follows the same procedure of building a histogram. This step is needed in the proposed method.

On the other hand, the cleaning process is optional, it is used to remove the outliers of the execution time for a specific number of cores. This step is only applied when the accuracy of the prediction model is lower than the target value, and when the training dataset has noise. We determined that noise in training dataset diminishes the performance of the prediction model. The training phase generates the model by using a classification tree. We used the j48 algorithm to generate the prediction model. The following step (application phase) uses the generated model and determines the classes for a set of unknown instances. Finally, the validation phase compares the estimated classes generated in the previous step with the real ones and determine the accuracy of the prediction model.

5.4. Contributions

The main contribution of this thesis regards the proposed prediction method able to design a platform-independent prediction model in a free-workload and multicore environment. Such a designed model is capable of predicting the application execution time of HPC applications on unseen platforms with a defined subset of resource types (i.e. number of cores, number of nodes, the operation frequency of the CPU Core, RAM memory, and network bandwidth). The models generated by the proposed method have a maximum prediction error of 15%.

An important characteristic of the proposed method is that it does not require a characterisation of the application (e.g. apply code instrumentation to the application). Instead, the classification tree generates a profile of the application based on the different hardware scenarios provided in the dataset. Another important characteristic of the method is that it does not need to make a complete or a partial execution on the target system.

We used three HPC applications as case example, WRF model, miniFE, and Octopus, each application requires different computational resources (e.g. CPU, memory, or network bandwidth) to perform its processing operations. Then, we designed a prediction model for each dataset. After developing the prediction models, we evaluated the prediction accuracy of the models. The first step consisted on evaluating the prediction model accuracy in the design phase for each application. We showed that the prediction models of WRF, miniFE, and Octopus had an accuracy of 92.98%, 90.11%, 72.72%, respectively. Only the models of WRF and miniFE have a prediction error below of

15%.

The prediction error of Octopus-model was 28% because, as we established, the dataset presented noise in its instances. Therefore, it was necessary to apply the cleaning process in order to reduce the outliers (noise). Our cleaning process was able to reduce the prediction error in 11%, reaching an accuracy of 83%. We decided to use this model, and avoid continuing reducing the noise in the dataset, because we considered that by preserving more instances on the training phase, the classification tree could learn more about this particular behaviour in the Octopus application.

After designing the prediction model for each application, we evaluated the accuracy of each model on unseen platforms. WRF had an overall accuracy on the unseen platforms of 91.8337%, miniFE-model had an overall accuracy of 89.4255%, and Octopus had 67.5081%. We determined that the low accuracy of the Octopus-model was due to the noise presented in the dataset. The reason for such a noise in the dataset is the execution of Octopus is significantly different in every particular hardware configuration defined. We believe that this behaviour is because of the parallelisation model used in this application. On the other hand, our proposed method was able to predict with an accuracy above of 85% on different hardware scenarios. This is because we used a hardware configuration with Infiniband network architecture. Infiniband provides a network bandwidth of 40Gbps.

Another important contribution is showing that classification trees is a suitable technique for developing performance prediction models for distributed systems exhibiting non-linear behaviour.

Finally, a number of papers were published reporting some of the results of this research. These papers are listed in Annex F.

5.5. Future Work

The future paths for this research are the following.

5.5.1 Adapt our method to generate prediction models that support workload environments. A distributed system usually executes several applications concurrently. Then, a workload in the resources' system is generated due to the application execution. Therefore, it is essential to adapt our model to be used in these environments.

5.5.2 Include input data as a variable of the performance prediction models. As we mentioned in Chapter 4, the input data of the applications remained constant in all the experiments. However, the input data has influence on the application execution time [88, 26]. Therefore, it is desirable that a model be able to predict the performance of a system without the need to generate a new prediction model for a different input data.

5.5.3 Improve the classification accuracy with a higher number of classes. Appendix A shows how the prediction accuracy diminishes when the number of

classes increases. When the user defines a high number of classes, the amplitude of the segments decreases. This means that the amplitude of the segments between the upper and lower threshold are very small. Then, an improvement of the accuracy with a high number of classes is needed in order to reduce the amplitude segment, i.e, instead of having a prediction with an amplitude between 4 and 7 minutes, we want to reduce to an amplitude for one or two minutes without scarifying the accuracy of the prediction model.

5.5.4 Reduce the number of instances in the dataset without affecting the accuracy of the prediction model. The data collecting process is the most-time consuming phase in the design process. However, the input data used for training a machine learning technique indicates how this technique will be able to generalize for new unknown data. Then, reducing the amount of data used in the training phase will impact the accuracy of the prediction model. Thus, it is necessary to reduce the number of input data without affecting the accuracy of the prediction model.

5.5.5 Reduce the outliers in the execution time. Machine learning techniques are negatively influenced by the poor quality data. For example, Octopus presented noise in the training dataset, which generated a prediction model with a high prediction error. After removing the outliers, the prediction error was reduced. Then our cleaning process improves the prediction accuracy. However, this method requires the user assistance. Then, it is necessary to use an unsupervised cleaning process, such as, a filtering [6], functional dependency [92], or artificial neural networks [47], to name a few.

5.5.6 Improve the segmentation process. Our segmentation process is based on the method to build a histogram. Appendix A shows a comparative between our proposed method and an unsupervised clustering algorithm. Despite our method presents better performance than the clustering algorithm, we believe that an improved clustering algorithm will have a better performance than our proposed method.

5.5.7 Explore other classification methods, such as classification and regression trees. Our prediction model uses a classification tree to generate the prediction model. Our classification tree is based on the C4.5 algorithm, which has been listed in the top ten most influential data mining algorithms [131], and it has been taken as a benchmark reference for developing new approaches [104, 48]. However, there are some other approaches, such as Classification and regression trees (CART) [79], able to handle discrete and continue values. Consequently, a future venue of research includes designing a platform-independent prediction model with this method in order to compare it with our approach.

5.5.8 Provide support for runtime prediction. We exposed that our proposed method requires less than one second to create a new prediction model. Therefore,

we believe that this method can be employed in problems that require runtime prediction such as scheduling problems [24], or resource usage problems [18] where performance prediction is needed.

5.5.9 Explore how the method can provide support for cloud environments. Cloud computing is a new emerging paradigm, since it provides on-demand software, hardware, and data services [133]. In this kind of systems, resources are shared among many users that need to execute their own software application. Each application demands different resources and it is very common that the end user does not know in advance the amount of resources required by such an application [137]. This fact makes it difficult for Cloud systems to satisfy the Quality of Service demands in an efficient way. Thus, an approach that could improve the resource usage and the efficiency of a cloud system is the use of a performance prediction method [11]. That is, knowing ahead the behaviour of an application would allows us to exploit more efficiently the system resources in a Cloud environment [37]. We consider this as an area of opportunity where it is feasible to apply our proposed method model given its suitability to be employed at runtime.

5.5.10 Customisation of the cross-validation method. The cross-validation method uses the mean of classification error to obtain the overall accuracy of the prediction model. However, when we performed an analysis of the classification error by class, we determined that some classes had a higher rate of classification error. Therefore, a venue for future work is customising the cross-validation process. This, so that instead of considering the overall instance classification error, the cross-validation process considers the classification error by class on each iteration. Then, the prediction accuracy can be evaluated based on the weight of the percentages obtained for each class. We believe this customisation of the cross-validation method can reduce the classification error in the classes with low classification accuracy.

5.6. Concluding Remarks

Capacity planning is an important task aiming to determine the amount of computing resources that an institution requires in order to satisfy its business needs. Sometimes this may involve the need to predict the performance of HPC applications on unseen platforms.

This thesis has investigated a method to design models to predict the performance of HPC applications on unseen platforms. Although there are still many unsolved issues, it is hoped that the work presented in this thesis will have some useful influence on the future development of performance prediction models for HPC applications.

A. Pre-processing method analysis

A.1. Introduction

One key element of our proposed method is the segmentation process. The segmentation process transforms the application execution time from continuous domain into discrete domain. Domain transformation relies on instance segmentation, where instances are divided into several classes defined by the user. This is an important stage within the design process, given that segmentation could greatly influence the performance of the prediction model by either increasing or diminishing it.

Hence, we evaluate this process by considering two key aspects. The first one is to compare the accuracy of our prediction method considering different number of classes; the second one is to compare our proposed method against an unsupervised prediction method.

A.2. Analysis of number of classes

We evaluate the accuracy of the prediction method of each application (WRF, miniFE, and Octopus) for 4,6,8,10,12, and 16 classes. Figure A.1 shows the accuracy of our prediction method with different number of classes.

We can observe that the accuracy of our prediction model is above of 90% when the number of classes is low. However, the accuracy is reduced when the number of classes is increased; fact that is closely related to the amplitude of the segments. The amplitude of the segments used for our evaluation for each number of classes is presented in Table A.1.

Table A.1.: Amplitude of the segments defined by different number of classes

	Segment amplitude (seconds)		
	WRF	miniFE	Octopus
4 Classes	389.910	146.930	899.736
6 Classes	259.940	97.953	599.824
8 Classes	194.955	73.465	449.868
10 Classes	155.964	58.772	359.894
12 Classes	129.970	48.977	299.912
16 Classes	97.478	36.733	224.934

Large amplitude contains greater number of instances. For example WRF its largest

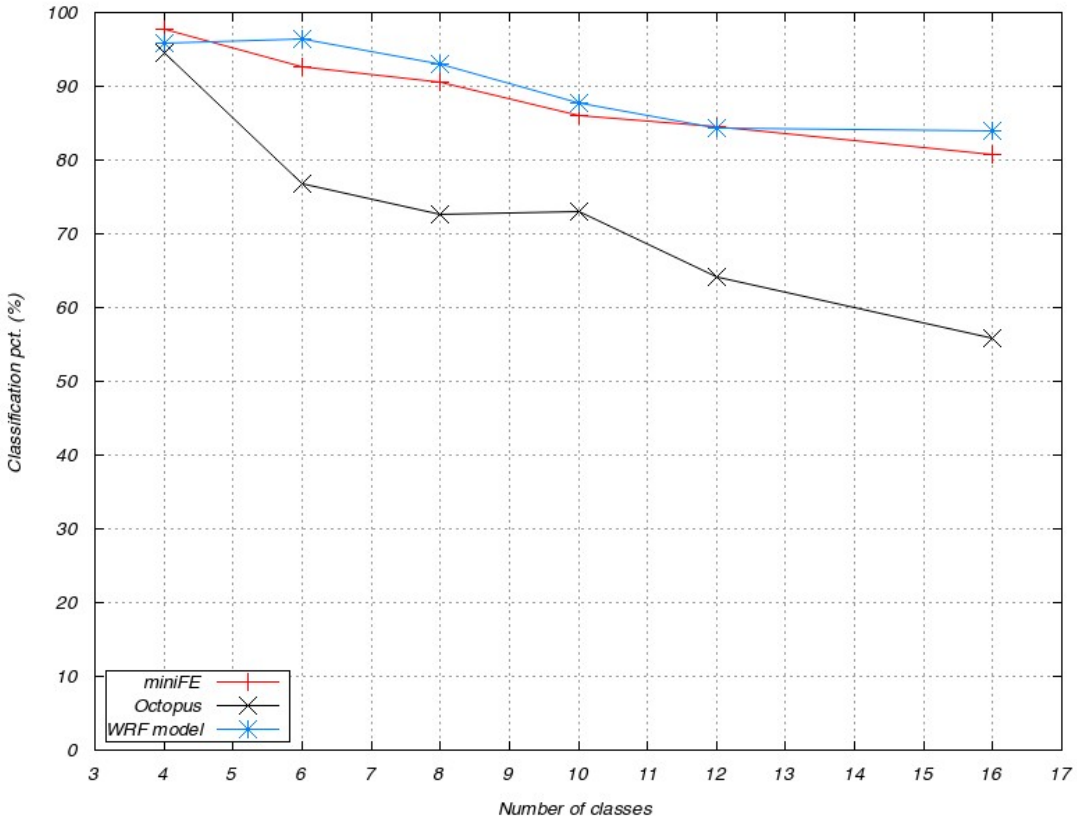

Figure A.1.: Percentage of accuracy for different number of classes

amplitude is about 390 seconds (6 minutes and 30 seconds), miniFE 147 seconds (2 minutes 27 seconds) and WRF 900 seconds (15 minutes). These values indicate a large interval of time between the upper and lower threshold. Therefore, a large execution time interval is not suitable for estimating the execution time. An advantage of increasing the number of class is that the amplitude decreases. However, when the number of classes is increased, the accuracy of the prediction model decreases. Some predictions fall below of 85%.

A.3. Unsupervised segmentation method

In order to evaluate the accuracy of our proposed method, we compare it against an unsupervised method. Unsupervised method selected to compare against is the clustering algorithm simple K-means [118]. Simple k-means algorithm was chosen after comparing five clustering algorithms: Canopy, Farthest First, Hierarchical clustering, LVQ, and Simple K-Means. Selection criteria were based on the application data distribution. A summary data distribution after applying the clustering algorithms in each application are depicted in Figure A.2, Figure A.3, and Figure A.4, which respectively corresponds to WRF, miniFE, and Octopus.

We can observe that K-means algorithm makes a better distribution of data into the different clusters. The reason of the latter is that the number of instances on each cluster generated by K-means is closer to their median value. Which means, that there is a little difference between the numbers of instances between each cluster.

After selecting the most optimal clustering algorithm, we compare the accuracy of the prediction model using our segmentation process and the selected clustering algorithm. Figure A.5 presents a comparative between our segmentation process and k-means.

From the comparison, it can be observed that our proposed method has better accuracy when compared against the unsupervised segmentation algorithm. Which can be confirmed for all considered applications and classes.

A.4. Conclusions

We evaluated the methods proposed in the data pre-processing stage by comparing the accuracy of our model for different number of classes; also by comparing our segmentation method with an unsupervised algorithm.

We found that the prediction accuracy is reduced when the number of classes is increased. Therefore, we consider eight classes as a suitable number to be used in our case example, given that such a number represents an optimal interval amplitude. Also we have showed that, regarding classification, our proposed segmentation method achieves better accuracy than the k-means algorithm.

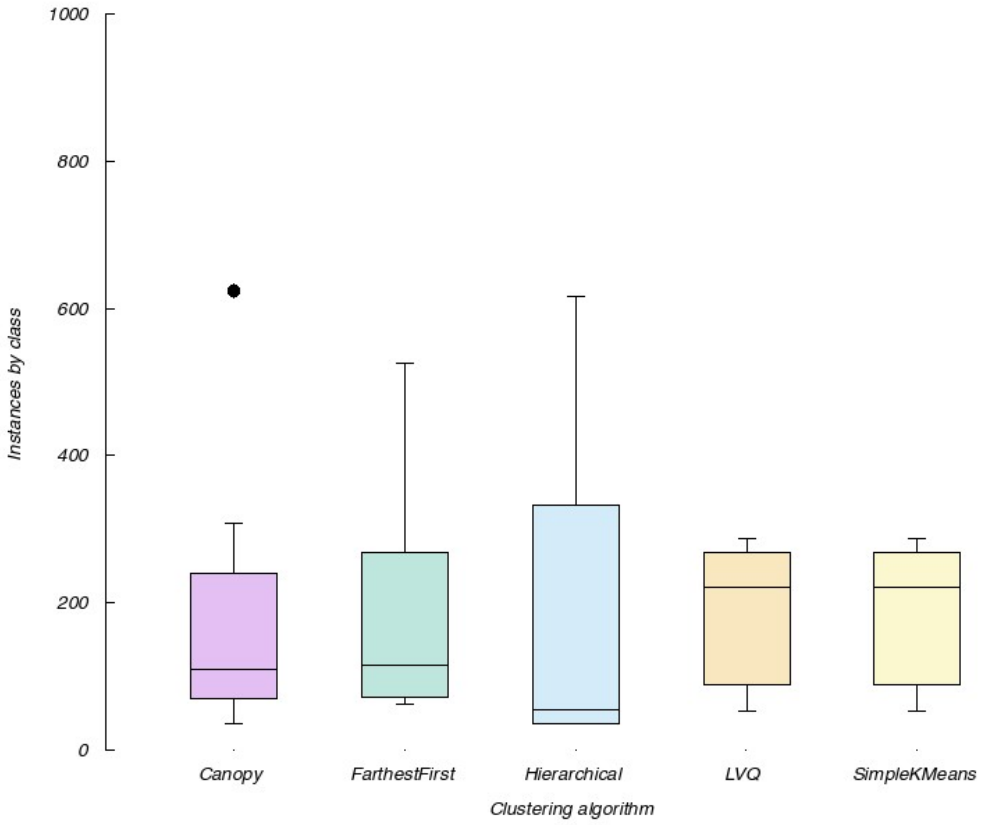

Figure A.2.: Distribution of data for WRF

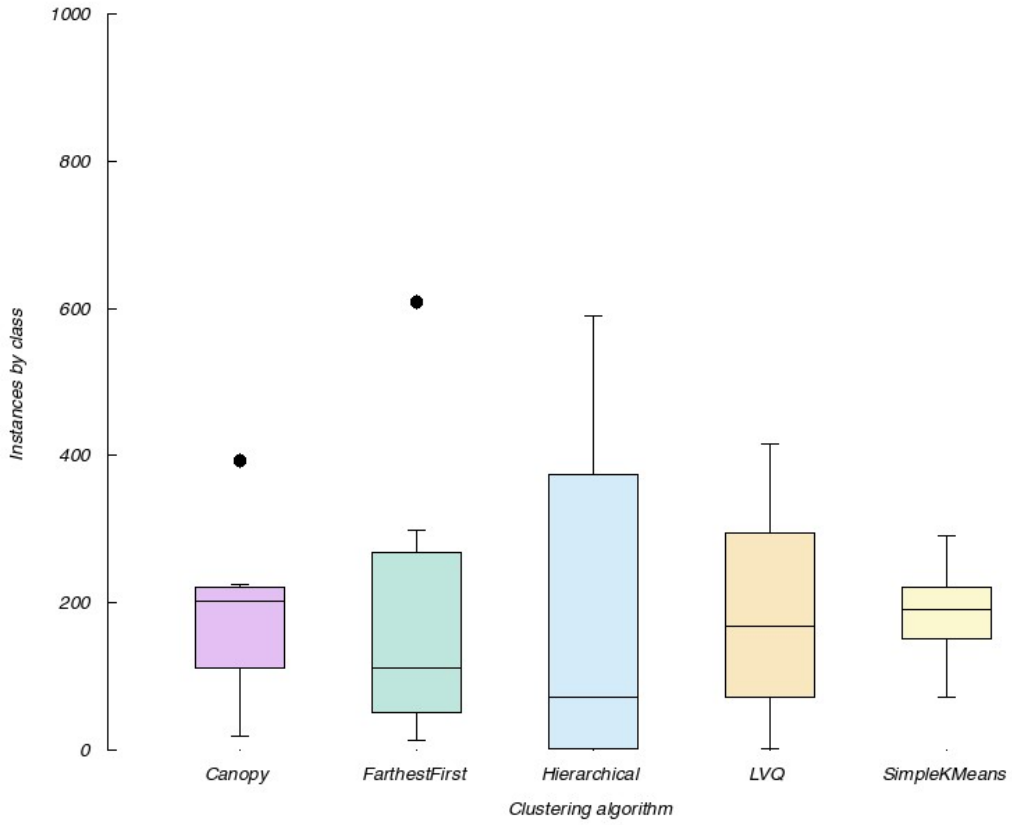

Figure A.3.: Distribution of data for miniFE

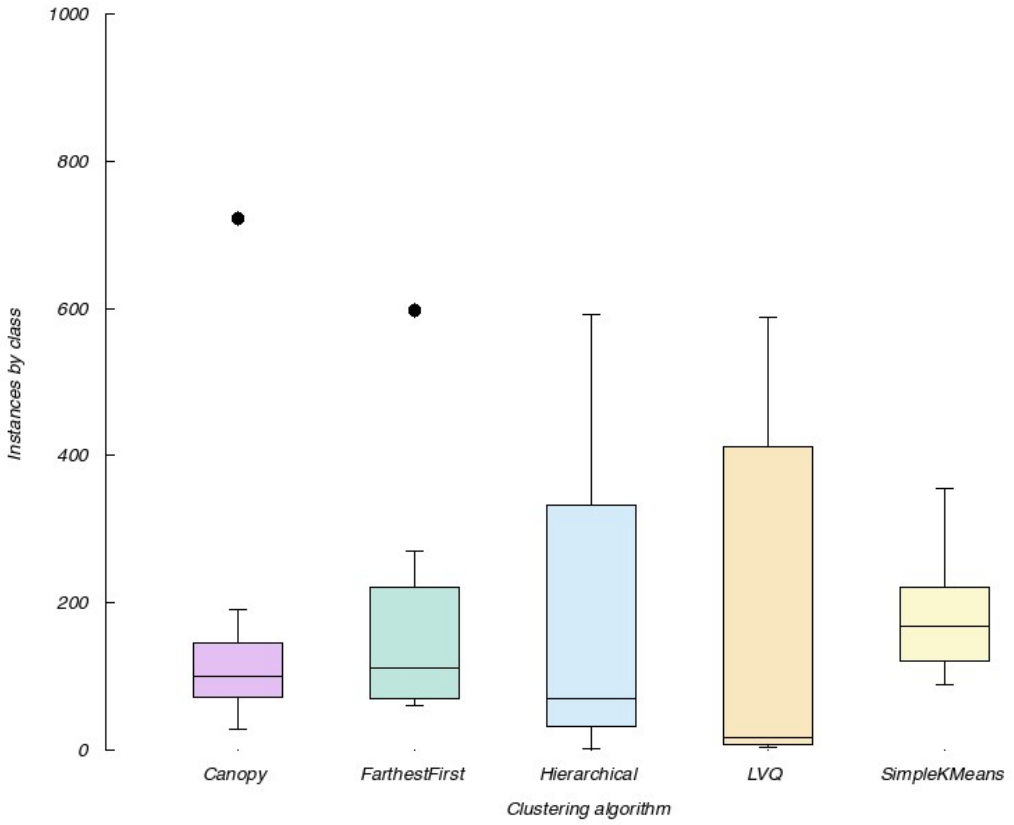

Figure A.4.: Distribution of data for Octopus

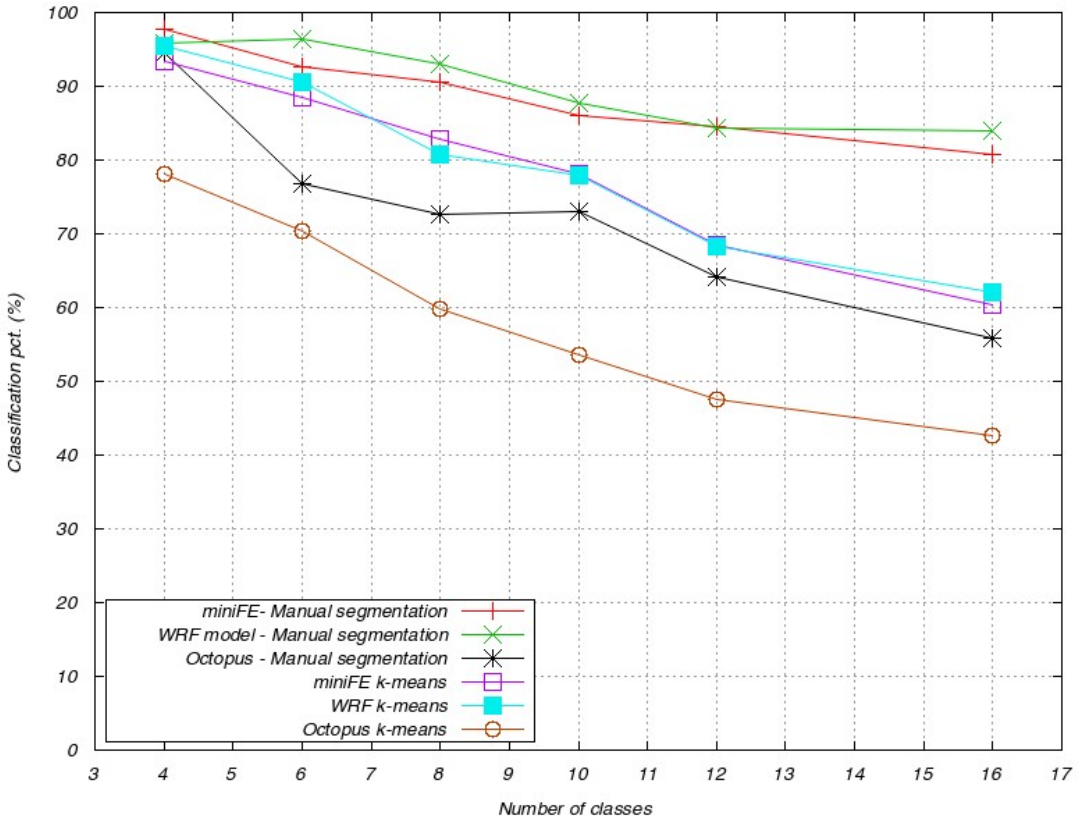

Figure A.5.: Prediction model accuracy for k-means and proposed method for different number of classes

B. Noise data in Octopus

We have established that noise in a dataset affects the prediction accuracy. However, if we apply a cleaning process the prediction accuracy is improved. As we presented in Chapter 4, Octopus is an application that presents an unusual behaviour during its execution. This unusual behaviour is presented in every combination of cores, and in all platform scenarios. Figure B.1 depicts the execution of Octopus on different platform scenarios.

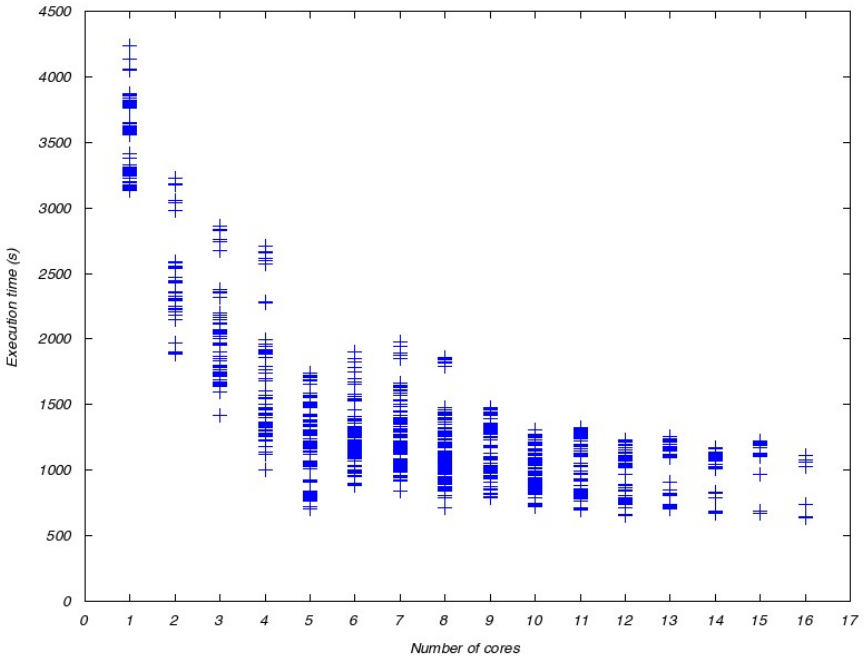

Figure B.1.: Execution time of Octopus for a different number of cores

Figure B.1 shows that there is a significant difference in the execution time of the combinations with the same number of cores. For example, the range of the execution time of Octopus with two cores is 990 seconds (16 minutes, 30 seconds) between the lowest (2232 seconds) and highest (3223 seconds) execution time, the range of the execution time with eight cores is 1054 seconds (17 minutes, 57 seconds), and so on. This difference of execution time for a specific number of cores is noise that affects the performance of the prediction model. A detail analysis of this information is presented in Table B.1.

Table B.1.: Summary of the execution time of Octopus

		CPU frequency			
		1.60GHZ	**1.73GHz**	**2.00GHz**	**2.13GHz**
One Core	Min	3758.783	3559.207	3131.360	3133.879
	Max	4237.714	4055.495	3791.851	3871.056
	Median	3773.667	3573.506	3234.372	3257.007
	Avg.	3805.001	3609.904	3244.282	3345.467
	Std. Dev.	96.825	111.661	138.370	262.796
	Range	**478.931**	**496.288**	**660.491**	**737.177**
	Range/2	**119.733**	**124.072**	**165.123**	**184.294**
	Half+	3924.734	3733.976	3409.405	3529.762
	Half-	3685.268	3485.832	3079.159	3161.173
Two cores	Min	2232.132	2146.749	1969.064	1882.835
	Max	3223.099	3182.044	3058.389	2547.207
	Median	2440.845	2433.270	2350.504	1897.423
	Avg.	2516.554	2485.015	2392.275	2104.657
	Std. Dev.	338.051	388.722	284.615	285.695
	Range	**990.967**	**1035.295**	**1089.325**	**664.372**
	Range/2	**247.742**	**258.824**	**272.331**	**166.093**
	Half+	2764.296	2743.838	2664.606	2270.750
	Half-	2268.812	2226.191	2119.944	1938.564
Three cores	Min	1658.697	1592.084	1636.190	1419.234
	Max	2864.778	2835.595	2756.738	2362.636
	Median	2020.591	1966.955	2034.973	1743.152
	Avg.	2074.894	2082.875	2006.358	1817.015
	Std. Dev.	342.873	366.502	301.026	280.624
	Range	**1206.081**	**1243.511**	**1120.548**	**943.402**
	Range/2	**301.520**	**310.878**	**280.137**	**235.851**
	Half+	2376.415	2393.752	2286.495	2052.865
	Half-	1773.374	1771.997	1726.221	1581.164
Four cores	Min	1179.579	1134.418	1117.750	998.571
	Max	2706.676	2668.503	2614.352	2278.824
	Median	1472.786	1430.769	1586.659	1263.086
	Avg.	1641.954	1646.040	1666.973	1419.411
	Std. Dev.	425.117	429.721	411.177	384.390
	Range	**1527.097**	**1534.085**	**1496.602**	**1280.253**
	Range/2	**381.774**	**383.521**	**374.151**	**320.063**
	Half+	2023.728	2029.561	2041.124	1739.474

Continued on next page

Table B.1 – *Continued from previous page*

		CPU frequency			
		1.60GHZ	**1.73GHz**	**2.00GHz**	**2.13GHz**
	Half-	1260.179	1262.518	1292.823	1099.348
Five cores	Min	702.778	803.536	770.685	724.402
	Max	1585.756	1737.059	1736.349	1301.502
	Median	1210.184	1425.270	1320.942	1053.875
	Avg.	1252.611	1323.162	1212.778	1090.459
	Std. Dev.	245.222	279.432	328.269	184.809
	Range	**882.978**	**933.523**	**965.664**	**577.100**
	Range/2	**220.745**	**233.381**	**241.416**	**144.275**
	Half+	1473.356	1556.543	1454.194	1234.734
	Half-	1031.867	1089.781	971.362	946.184
Six cores	Min	979.094	954.136	950.550	885.457
	Max	1900.562	1849.529	1900.093	1547.854
	Median	1274.128	1249.717	1214.918	1096.751
	Avg.	1257.738	1292.499	1287.515	1096.225
	Std. Dev.	226.059	241.246	208.967	166.520
	Range	**921.468**	**895.393**	**949.543**	**662.397**
	Range/2	**230.367**	**223.848**	**237.386**	**165.599**
	Half+	1488.105	1516.347	1524.901	1261.825
	Half-	1027.371	1068.650	1050.129	930.626
Seven cores	Min	946.645	920.955	1071.064	836.646
	Max	1976.148	1945.715	1877.866	1355.716
	Median	1258.940	1371.786	1306.946	1036.902
	Avg.	1276.696	1348.641	1347.435	1084.317
	Std. Dev.	237.318	231.057	195.253	130.144
	Range	**1029.503**	**1024.760**	**806.802**	**519.070**
	Range/2	**257.376**	**256.190**	**201.701**	**129.768**
	Half+	1534.072	1604.831	1549.136	1214.085
	Half-	1019.320	1092.451	1145.735	954.550
Eight cores	Min	808.484	786.884	925.981	709.366
	Max	1862.495	1813.142	1843.343	1180.736
	Median	1084.471	1190.402	1137.305	1026.766
	Avg.	1110.655	1170.566	1184.095	978.855
	Std. Dev.	200.230	203.745	188.663	106.685
	Range	**1054.011**	**1026.258**	**917.362**	**471.370**
	Range/2	**263.503**	**256.565**	**229.341**	**117.843**
	Half+	1374.158	1427.130	1413.436	1096.698
	Half-	847.152	914.001	954.755	861.013

Continued on next page

Table B.1 – *Continued from previous page*

		CPU frequency			
		1.60GHZ	**1.73GHz**	**2.00GHz**	**2.13GHz**
Nine cores	Min	850.011	908.827	1027.819	792.036
	Max	1354.366	1474.423	1441.757	1008.274
	Median	1017.620	1276.927	1288.822	940.348
	Avg.	1072.874	1219.009	1274.929	927.836
	Std. Dev.	143.398	167.806	135.876	65.050
	Range	**504.355**	**565.596**	**413.938**	**216.238**
	Range/2	**126.089**	**141.399**	**103.485**	**54.060**
	Half+	1198.962	1360.408	1378.413	981.895
	Half-	946.785	1077.610	1171.444	873.776
Ten cores	Min	791.019	842.749	926.461	720.837
	Max	1263.814	1305.779	1270.066	908.506
	Median	957.777	1119.199	1070.754	862.567
	Avg.	977.628	1096.669	1094.903	832.178
	Std. Dev.	133.014	125.710	95.679	59.758
	Range	**472.795**	**463.030**	**343.605**	**187.669**
	Range/2	**118.199**	**115.758**	**85.901**	**46.917**
	Half+	1095.826	1212.427	1180.804	879.095
	Half-	859.429	980.912	1009.002	785.261
Eleven cores	Min	767.305	1027.484	1103.203	700.465
	Max	1317.934	1320.327	1281.917	857.296
	Median	932.906	1206.456	1187.563	808.623
	Avg.	1006.522	1210.575	1192.871	798.386
	Std. Dev.	177.793	95.759	62.252	44.144
	Range	**550.629**	**292.843**	**178.714**	**156.831**
	Range/2	**137.657**	**73.211**	**44.679**	**39.208**
	Half+	1144.179	1283.786	1237.549	837.594
	Half-	868.865	1137.364	1148.192	759.178
Twelve cores	Min	716.805	1030.047	1014.860	657.923
	Max	1232.491	1224.819	1170.099	782.362
	Median	930.821	1106.604	1086.257	759.847
	Avg.	963.542	1126.431	1091.202	750.791
	Std. Dev.	157.583	64.262	54.013	35.061
	Range	**515.686**	**194.772**	**155.239**	**124.439**
	Range/2	**128.922**	**48.693**	**38.810**	**31.110**
	Half+	1092.463	1175.124	1130.012	781.900
	Half-	834.620	1077.738	1052.392	719.681

Continued on next page

Table B.1 – *Continued from previous page*

		CPU frequency			
		1.60GHZ	**1.73GHz**	**2.00GHz**	**2.13GHz**
	Min	803.769	1153.031	1092.887	707.875
	Max	1237.913	1259.218	1195.912	738.943
	Median	1024.655	1196.897	1142.063	720.156
	Avg.	1019.439	1201.829	1141.208	720.957
Thirteen cores	Std. Dev.	181.753	42.438	41.595	10.060
	Range	**434.144**	**106.187**	**103.025**	**31.068**
	Range/2	**108.536**	**26.547**	**25.756**	**7.767**
	Half+	1127.975	1228.376	1166.965	728.724
	Half-	910.903	1175.282	1115.452	713.190
	Min	791.199	1074.317	1011.169	672.210
	Max	1168.628	1139.634	1104.909	685.536
	Median	958.591	1110.936	1057.936	682.020
	Avg.	969.278	1105.343	1057.252	680.502
Fourteen cores	Std. Dev.	173.718	25.702	36.310	4.791
	Range	**377.429**	**65.317**	**93.740**	**13.326**
	Range/2	**94.357**	**16.329**	**23.435**	**3.331**
	Half+	1063.635	1121.672	1080.687	683.834
	Half-	874.920	1089.013	1033.817	677.171
	Min	966.088	1169.690	1103.005	668.439
	Max	1221.941	1203.289	1127.921	690.381
	Median	1090.169	1192.625	1114.829	671.840
	Avg.	1092.092	1189.557	1115.146	675.625
Fifteen cores	Std. Dev.	143.398	14.931	10.960	10.162
	Range	**255.853**	**33.599**	**24.916**	**21.942**
	Range/2	**63.963**	**8.400**	**6.229**	**5.486**
	Half+	1156.055	1197.957	1121.375	681.111
	Half-	1028.129	1181.158	1108.917	670.140
	Min	740.762	1078.781	1029.021	638.771
	Max	1115.755	1110.382	1062.087	644.345
	Median	928.259	1094.582	1045.554	641.558
	Avg.	928.259	1094.582	1045.554	641.558
Sixteen cores	Std. Dev.	265.160	22.345	23.381	3.941
	Range	**374.993**	**31.601**	**33.066**	**5.574**
	Range/2	**93.748**	**7.900**	**8.267**	**1.394**
	Half+	1022.007	1102.482	1053.821	642.952
	Half-	834.510	1086.681	1037.288	640.165

Continued on next page

Table B.1 – *Continued from previous page*

	CPU frequency			
	1.60GHZ	**1.73GHz**	**2.00GHz**	**2.13GHz**

Table B.1 presents a summary of execution time by number of cores, and the frequency of the CPU processor. The range value (marked in bold) shows the difference of the execution time between the lowest and highest values of the combinations for the same number of cores and same CPU frequency. Some of those range values reach 1000 seconds of difference. We assume that this unusual behaviour is due to the parallelization method used during the development of Octopus. Therefore, the noise cannot be avoided during the data collection process. Hence, it is necessary to apply the proposed cleaning process in applications that presents this unusual behaviour.

C. WRF confusion matrixes

This appendix shows the accuracy of the prediction model, and their corresponding confusion matrix for WRF in the unseen platforms.

Table C.1.: WRF prediction accuracy in UP1

(a) Summary of prediction accuracy

	Instances	Pct.
Correctly classified	25	69.4444%
Incorrectly classified	11	30.5556%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
4	0	0	0	0	0	0	0	a=Extremely fast
2	11	0	0	0	0	0	0	b=Too fast
0	5	5	0	0	0	0	0	c=Very fast
0	0	1	1	0	0	0	0	d=Fast
0	0	0	3	0	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	4	h=Extremely Slow

Table C.2.: Prediction accuracy for WRF in UP2

(a) Summary of prediction accuracy

	Instances	Pct.
Correctly classified	30	83.3333%
Incorrectly classified	6	16.6667%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
6	2	0	0	0	0	0	0	a=Extremely fast
0	10	0	0	0	0	0	0	b=Too fast
0	4	6	0	0	0	0	0	c=Very fast
0	0	0	4	0	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	4	h=Extremely Slow

Table C.3.: Prediction accuracy of WRF in UP3

(a) Summary of prediction accuracy

	Instances	Pct.
Correctly classified	106	91.3793%
Incorrectly classified	10	8.6207%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
88	0	0	0	0	0	0	0	a=Extremely fast
10	14	0	0	0	0	0	0	b=Too fast
0	0	4	0	0	0	0	0	c=Very fast
0	0	0	0	0	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	0	h=Extremely Slow

Table C.4.: WRF prediction accuracy in UP5

(a) Summary of prediction accuracy

	Instances	Pct.
Correctly classified	116	100.0000%
Incorrectly classified	0	0.0000%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
98	0	0	0	0	0	0	0	a=Extremely fast
0	14	0	0	0	0	0	0	b=Too fast
0	0	4	0	0	0	0	0	c=Very fast
0	0	0	0	0	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	0	h=Extremely Slow

Table C.5.: WRF prediction accuracy in UP6

(a) Summary of prediction accuracy

	Instances	Pct.
Correctly classified	108	93.1034%
Incorrectly classified	8	6.8966%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
100	0	0	0	0	0	0	0	a=Extremely fast
4	8	0	0	0	0	0	0	b=Too fast
0	4	0	0	0	0	0	0	c=Very fast
0	0	0	0	0	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	0	h=Extremely Slow

Table C.6.: WRF prediction accuracy in UP7

(a) Summary of prediction accuracy

	Instances	Pct.
Correctly classified	112	96.5517%
Incorrectly classified	4	3.4483%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
100	0	0	0	0	0	0	0	a=Extremely fast
4	12	0	0	0	0	0	0	b=Too fast
0	0	0	0	0	0	0	0	c=Very fast
0	0	0	0	0	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	0	h=Extremely Slow

Table C.7.: Prediction accuracy for WRF in UP8

(a) Summary of prediction accuracy

	Instances	Pct.
Correctly classified	228	87.0229%
Incorrectly classified	34	12.9771%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
186	0	0	0	0	0	0	0	a=Extremely fast
16	24	0	0	0	0	0	0	b=Too fast
0	12	10	0	0	0	0	0	c=Very fast
0	0	0	0	0	0	0	0	d=Fast
0	0	0	6	4	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	4	h=Extremely Slow

Table C.8.: Confusion matrix of the WRF for UP9

(a) Summary of prediction accuracy

	Instances	Pct.
Correctly classified	251	95.8015%
Incorrectly classified	11	4.1985%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
199	0	0	0	0	0	0	0	a=Extremely fast
3	36	0	0	0	0	0	0	b=Too fast
0	0	10	0	0	0	0	0	c=Very fast
0	0	0	6	4	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	4	g=Too slow
0	0	0	0	0	0	0	0	h=Extremely Slow

Table C.9.: Prediction accuracy for WRF in UP10

(a) Summary of prediction accuracy

	Instances	Pct.
Correctly classified	245	93.5115%
Incorrectly classified	17	6.4885%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
214	3	0	0	0	0	0	0	a=Extremely fast
0	31	0	0	0	0	0	0	b=Too fast
0	0	0	10	0	0	0	0	c=Very fast
0	0	0	0	0	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	0	0	0	0	0	4	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	0	h=Extremely Slow

Table C.10.: Prediction accuracy for WRF in UP11

(a) Summary of prediction accuracy

	Instances	Pct.
Correctly classified	238	90.8397%
Incorrectly classified	24	9.1603%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
184	0	0	0	0	0	0	0	a=Extremely fast
18	30	0	0	0	0	0	0	b=Too fast
0	6	10	0	0	0	0	0	c=Very fast
0	0	0	6	0	0	0	0	d=Fast
0	0	0	0	4	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	4	h=Extremely Slow

Table C.11.: Prediction accuracy for WRF in UP12

(a) Summary of prediction accuracy

	Instances	Pct.
Correctly classified	243	92.7481%
Incorrectly classified	19	7.2519%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
197	6	0	0	0	0	0	0	a=Extremely fast
5	30	0	0	0	0	0	0	b=Too fast
0	0	10	0	0	0	0	0	c=Very fast
0	0	0	6	4	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	4	g=Too slow
0	0	0	0	0	0	0	0	h=Extremely Slow

Table C.12.: Prediction accuracy for WRF in UP13

(a) Summary of prediction accuracy

	Instances	Pct.
Correctly classified	237	90.4580%
Incorrectly classified	25	9.5420%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
209	6	0	0	0	0	0	0	a=Extremely fast
5	28	0	0	0	0	0	0	b=Too fast
0	0	0	10	0	0	0	0	c=Very fast
0	0	0	0	0	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	0	0	0	0	0	4	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	0	h=Extremely Slow

D. miniFE confusion matrixes

This appendix shows the accuracy of the prediction model, and their corresponding confusion matrix for miniFE in the unseen platforms.

Table D.1.: miniFE prediction accuracy in UP1

(a) Summary of prediction accuracy

Correctly classified	21	60.0000%
Incorrectly classified	14	40.0000%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
0	0	0	0	0	0	0	0	a=Extremely fast
2	10	0	0	0	0	0	0	b=Too fast
0	4	8	0	0	0	0	0	c=Very fast
0	0	4	0	0	0	0	0	d=Fast
0	0	0	4	0	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	3	h=Extremely Slow

Table D.2.: miniFE prediction accuracy in UP2

(a) Summary of prediction accuracy

Correctly classified	21	60.0000%
Incorrectly classified	14	40.0000%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
0	0	0	0	0	0	0	0	a=Extremely fast
2	10	0	0	0	0	0	0	b=Too fast
0	4	8	0	0	0	0	0	c=Very fast
0	0	4	0	0	0	0	0	d=Fast
0	0	0	4	0	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	3	h=Extremely Slow

Table D.3.: miniFE prediction accuracy in UP3

(a) Summary of prediction accuracy

Correctly classified	106	91.3793%
Incorrectly classified	10	8.6207%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
84	0	0	0	0	0	0	0	a=Extremely fast
8	18	0	0	0	0	0	0	b=Too fast
0	2	4	0	0	0	0	0	c=Very fast
0	0	0	0	0	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	0	h=Extremely Slow

Table D.4.: miniFE prediction accuracy in UP5

(a) Summary of prediction accuracy

Correctly classified	112	96.5517%
Incorrectly classified	4	3.4483%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
88	0	0	0	0	0	0	0	a=Extremely fast
4	20	0	0	0	0	0	0	b=Too fast
0	0	4	0	0	0	0	0	c=Very fast
0	0	0	0	0	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	0	h=Extremely Slow

Table D.5.: miniFE prediction accuracy in UP6

(a) Summary of prediction accuracy

Correctly classified	103	88.7931%
Incorrectly classified	13	11.2069%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
92	0	0	0	0	0	0	0	a=Extremely fast
12	11	0	0	0	0	0	0	b=Too fast
0	1	0	0	0	0	0	0	c=Very fast
0	0	0	0	0	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	0	h=Extremely Slow

Table D.6.: miniFE prediction accuracy in UP7

(a) Summary of prediction accuracy

Correctly classified	109	93.9655%
Incorrectly classified	7	6.0345%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
97	0	0	0	0	0	0	0	a=Extremely fast
7	12	0	0	0	0	0	0	b=Too fast
0	0	0	0	0	0	0	0	c=Very fast
0	0	0	0	0	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	0	h=Extremely Slow

Table D.7.: miniFE prediction accuracy in UP8

(a) Summary of prediction accuracy

Correctly classified	246	93.8931%
Incorrectly classified	16	6.1069%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
180	0	0	0	0	0	0	0	a=Extremely fast
10	42	6	0	0	0	0	0	b=Too fast
0	0	10	0	0	0	0	0	c=Very fast
0	0	0	10	0	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	4	h=Extremely Slow

Table D.8.: miniFE prediction accuracy in UP9

(a) Summary of prediction accuracy

Correctly classified	246	93.8931%
Incorrectly classified	16	6.1069%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
190	6	0	0	0	0	0	0	a=Extremely fast
0	36	6	0	0	0	0	0	b=Too fast
0	0	10	0	0	0	0	0	c=Very fast
0	0	0	10	0	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	4	g=Too slow
0	0	0	0	0	0	0	0	h=Extremely Slow

Table D.9.: miniFE prediction accuracy in UP10

(a) Summary of prediction accuracy

Correctly classified	222	84.7328%
Incorrectly classified	40	15.2672%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
202	24	0	0	0	0	0	0	a=Extremely fast
0	16	6	0	0	0	0	0	b=Too fast
0	0	4	6	0	0	0	0	c=Very fast
0	0	0	0	0	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	0	0	0	0	0	4	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	0	h=Extremely Slow

Table D.10.: miniFE prediction accuracy in UP11

(a) Summary of prediction accuracy

Correctly classified	426	97.7064%
Incorrectly classified	10	2.2936%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
328	0	0	0	0	0	0	0	a=Extremely fast
4	66	6	0	0	0	0	0	b=Too fast
0	0	14	0	0	0	0	0	c=Very fast
0	0	0	14	0	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	4	h=Extremely Slow

Table D.11.: miniFE prediction accuracy in UP12

(a) Summary of prediction accuracy

Correctly classified	66	75.0000%
Incorrectly classified	22	25.0000%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
48	12	0	0	0	0	0	0	a=Extremely fast
0	6	6	0	0	0	0	0	b=Too fast
0	0	6	0	0	0	0	0	c=Very fast
0	0	0	6	0	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	4	g=Too slow
0	0	0	0	0	0	0	0	h=Extremely Slow

Table D.12.: miniFE prediction accuracy in UP13

(a) Summary of prediction accuracy

Correctly classified	54	61.3636%
Incorrectly classified	34	38.6364%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
48	18	0	0	0	0	0	0	a=Extremely fast
0	6	6	0	0	0	0	0	b=Too fast
0	0	0	6	0	0	0	0	c=Very fast
0	0	0	0	0	0	0	0	d=Fast
0	0	0	0	0	0	4	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	0	h=Extremely Slow

E. Octopus confusion matrixes

This appendix shows the accuracy of the prediction model, and their corresponding confusion matrix for Octopus in the unseen platforms.

Table E.1.: Octopus prediction accuracy in UP3

(a) Summary of prediction accuracy

Correctly classified	45	38.7931%
Incorrectly classified	71	61.2069%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
45	0	0	0	0	0	0	0	a=Extremely fast
24	0	0	0	0	0	0	0	b=Too fast
21	4	0	0	0	0	0	0	c=Very fast
2	8	0	0	0	0	0	0	d=Fast
0	4	0	0	0	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	4	0	0	0	0	0	g=Too slow
0	0	0	4	0	0	0	0	h=Extremely Slow

Table E.2.: Octopus prediction accuracy in UP5

(a) Summary of prediction accuracy

Correctly classified	27	23.2759%
Incorrectly classified	89	76.7241%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
0	47	0	0	0	0	0	0	a=Extremely fast
0	25	0	0	0	0	0	0	b=Too fast
0	20	2	0	0	0	0	0	c=Very fast
0	6	6	0	0	0	0	0	d=Fast
0	0	2	0	0	0	0	0	e=Slow
0	4	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	4	0	0	0	0	h=Extremely Slow

Table E.3.: Octopus prediction accuracy in UP6

(a) Summary of prediction accuracy

Correctly classified	22	18.9655%
Incorrectly classified	94	81.0345%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
0	56	0	0	0	0	0	0	a=Extremely fast
0	22	0	0	0	0	0	0	b=Too fast
0	22	0	0	0	0	0	0	c=Very fast
0	8	0	0	0	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	4	0	0	0	0	0	0	f=Very Slow
0	0	4	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	0	h=Extremely Slow

Table E.4.: Octopus prediction accuracy in UP7

(a) Summary of prediction accuracy

Correctly classified	59	50.8621%
Incorrectly classified	57	49.1379%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
57	0	0	0	0	0	0	0	a=Extremely fast
19	2	0	0	0	0	0	0	b=Too fast
22	4	0	0	0	0	0	0	c=Very fast
0	4	0	0	0	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	4	0	0	0	0	0	0	f=Very Slow
0	0	4	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	0	h=Extremely Slow

Table E.5.: Octopus prediction accuracy in UP8

(a) Summary of prediction accuracy

Correctly classified	210	80.1527%
Incorrectly classified	52	19.8473%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
158	0	0	0	0	0	0	0	a=Extremely fast
32	32	2	0	0	0	0	0	b=Too fast
0	4	4	2	0	6	0	0	c=Very fast
0	0	0	2	0	6	0	0	d=Fast
0	0	0	0	10	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	4	h=Extremely Slow

Table E.6.: Octopus prediction accuracy in UP9

(a) Summary of prediction accuracy

Correctly classified	208	79.3893%
Incorrectly classified	54	20.6107%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
173	2	0	0	0	0	0	0	a=Extremely fast
17	34	6	0	0	2	0	0	b=Too fast
0	0	0	4	0	10	0	0	c=Very fast
0	0	0	0	10	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	3	g=Too slow
0	0	0	0	0	0	0	1	h=Extremely Slow

Table E.7.: Octopus prediction accuracy in UP10

(a) Summary of prediction accuracy

Correctly classified	190	72.5191%
Incorrectly classified	72	27.4809%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
190	0	0	0	0	0	0	0	a=Extremely fast
24	0	18	8	0	0	0	0	b=Too fast
0	0	0	2	6	6	0	0	c=Very fast
0	0	0	0	4	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	0	0	0	0	0	4	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	0	h=Extremely Slow

Table E.8.: Octopus prediction accuracy in UP11

(a) Summary of prediction accuracy

Correctly classified	234	89.3130%
Incorrectly classified	28	10.6870%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
178	0	0	0	0	0	0	0	a=Extremely fast
12	36	2	0	0	2	0	0	b=Too fast
0	0	4	0	0	6	0	0	c=Very fast
0	0	0	4	2	4	0	0	d=Fast
0	0	0	0	8	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	4	h=Extremely Slow

Table E.9.: Octopus prediction accuracy in UP12

(a) Summary of prediction accuracy

Correctly classified	210	80.1527%
Incorrectly classified	52	19.8473%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
182	8	0	0	0	0	0	0	a=Extremely fast
8	28	6	0	0	6	0	0	b=Too fast
0	0	0	4	0	6	0	0	c=Very fast
0	0	0	0	10	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	0	0	0	0	0	0	0	f=Very Slow
0	0	0	0	0	0	0	4	g=Too slow
0	0	0	0	0	0	0	0	h=Extremely Slow

Table E.10.: Octopus prediction accuracy in UP13

(a) Summary of prediction accuracy

Correctly classified	52	59.0909%
Incorrectly classified	36	40.9091%

(b) Confusion matrix

a	b	c	d	e	f	g	h	⇐Classified as
52	0	2	0	0	0	0	0	a=Extremely fast
8	0	4	6	0	5	0	0	b=Too fast
0	0	0	0	6	1	0	0	c=Very fast
0	0	0	0	0	0	0	0	d=Fast
0	0	0	0	0	0	0	0	e=Slow
0	0	0	0	0	0	0	4	f=Very Slow
0	0	0	0	0	0	0	0	g=Too slow
0	0	0	0	0	0	0	0	h=Extremely Slow

F. Publications

F.1. JCR Journals

- Jesus Flores-Contreras, Hector A. Duran-Limon. Performance Prediction of Parallel Applications: A Systematic Literature Review Parallel Computing. Elsevier, submitted to Parallel Computing Journal of Elsevier in February 2016.

- Hector A. Duran-Limon, Jesus Flores-Contreras, Nikos Parlavantzas, Ming Zhao, Angel Meulenert-Pena. Efficient execution of the WRF model and other HPC applications in the cloud. Earth Science Informatics, March 2016. pp 1-18. ISSN 1865-0481. DOI : 10.1007/s12145-016-0253-7.

F.2. Conferences

- Jesus Flores-Contreras, Carlos Ruiz, Pablo Salazar, Hector A. Duran-Limon,Efren Mezura-Montes, Nicandro Cruz-Ramirez, Héctor-Gabriel Acosta-Mesa. **A Performance Prediction Model for Database Environments: A Preliminary Analysis.** High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on, New York, NY, 2015, pp. 1707-1712. doi:10.1109/HPCC-CSS-ICESS.2015.243

Bibliography

[1] ArchLinux ArchWiki. Cpu frequency scaling, January 2016.

[2] S. Achour, M. Ammar, B. Khmili, and W. Nasri. Mpi-perf-sim: Towards an automatic performance prediction tool of mpi programs on hierarchical clusters. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, pages 207–211, 2011.

[3] Ali Afzal, A. Stephen McGough, and John Darlington. Capacity planning and scheduling in grid computing environments. *Future Generation Computer Systems*, 24(5):404 – 414, 2008.

[4] M. Fatih Akay and Ipek Abaskele. Predicting the performance measures of an optical distributed shared memory multiprocessor by using support vector regression. *Expert Systems with Applications*, 37(9):6293 – 6301, 2010.

[5] G.M. Amdahl. Computer architecture and amdahl's law. *Computer*, 46(12):38–46, Dec 2013.

[6] B. G. Amidan, T. A. Ferryman, and S. K. Cooley. Data outlier detection using the chebyshev theorem. In *2005 IEEE Aerospace Conference*, pages 3814–3819, March 2005.

[7] Binstock Andrew. Multi-core processor architecture explained. http://software.intel.com/en-us/articles/multi-core-processor-architecture-explained, 2008. Consulted May 2013.

[8] Rohit Arora. Comparative analysis of classification algorithms on different datasets using weka. *International Journal of Computer Applications*, 54(13), 2012. Copyright - Copyright Foundation of Computer Science 2012; Last updated - 2013-09-13.

[9] Machine Learning Group at the University of Waikato. Waikato environtment for knowledge analysis, 2014. Revised on November 20th 2014.

[10] Christopher G. Atkeson, Andrew W. Moore, and Stefan Schaal. Locally weighted learning. *Artif. Intell. Rev.*, 11(1-5):11–73, February 1997.

[11] A.A. Bankole and S.A. Ajila. Predicting cloud resource provisioning using machine learning techniques. In *Electrical and Computer Engineering (CCECE), 2013 26th Annual IEEE Canadian Conference on*, pages 1–4, May 2013.

[12] Bradley J. Barnes, Barry Rountree, David K. Lowenthal, Jaxk Reeves, Bronis de Supinski, and Martin Schulz. A regression-based approach to scalability prediction. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 368–377, New York, NY, USA, 2008. ACM.

[13] G. Bauer, S. Gottlieb, and T. Hoefler. Performance modeling and comparative analysis of the milc lattice qcd application su3_rmd. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 652–659, May 2012.

[14] C. Bergmeir and J. M. Benitez. Forecaster performance evaluation with cross-validation and variants. In *Intelligent Systems Design and Applications (ISDA), 2011 11th International Conference on*, pages 849–854, Nov 2011.

[15] M. Boullón, J.C. Cabaleiro, R. Doallo, P. González, D.R. Martínez, M. Martín, J.C. Mouriño, T.F. Pena, and F.F. Rivera. Modeling execution time of selected computation and communication kernels on grids. In PeterM.A. Sloot, AlfonsG. Hoekstra, Thierry Priol, Alexander Reinefeld, and Marian Bubak, editors, *Advances in Grid Computing - EGC 2005*, volume 3470 of *Lecture Notes in Computer Science*, pages 731–740. Springer Berlin Heidelberg, 2005.

[16] Max Bramer. *Data for Data Mining*, pages 9–19. Springer London, London, 2013.

[17] Laura Carrington, Allan Snavely, and Nicole Wolter. A performance prediction framework for scientific applications. *Future Gener. Comput. Syst.*, 22(3):336–346, February 2006.

[18] Sara Casolari and Michele Colajanni. Short-term prediction models for server management in internet-based contexts. *Decision Support Systems*, 48(1):212 – 223, 2009. ¡ce:title¿Information product markets¡/ce:title¿.

[19] Cheng-Mei Chen, Chien-Yeh Hsu, Hung-Wen Chiu, and Hsiao-Hsien Rau. Prediction of survival in patients with liver cancer using artificial neural networks and classification and regression trees. In *Natural Computation (ICNC), 2011 Seventh International Conference on*, volume 2, pages 811–815, July 2011.

[20] Lianping Chen, Muhammad Ali Babar, and Nour Ali. Variability management in software product lines: a systematic review. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 81–90, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.

[21] Yong Chen, Xian-He Sun, and Ming Wu. Algorithm-system scalability of heterogeneous computing. *J. Parallel Distrib. Comput.*, 68(11):1403–1412, November 2008.

[22] Aymen Cherif, Hubert Cardot, and Romuald Boné. Som time series clustering and prediction with recurrent neural networks. *Neurocomput.*, 74(11):1936–1944, May 2011.

[23] Michael R. Chernick. *Bootstrap methods: a guide for practitioners and researchers*. Wiley Series in Probability and Statistics. Wiley-Interscience, 2 edition, 2007.

[24] Maria Chtepen, Filip Claeys, Bart Dhoedt, Filip De Turck, Jan Fostier, Piet Demeester, and Peter Vanrolleghem. Online execution time prediction for computationally intensive applications with periodic progress updates. *The Journal of Supercomputing*, pages 1–19, 2012. 10.1007/s11227-012-0748-z.

[25] George W. Collins. Fundamental numerical methods and data analysis, 2003.

[26] B.F. Cornea and J. Bourgeois. Performance prediction of distributed applications using block benchmarking methods. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, pages 183–190, 2011.

[27] BogdanFlorin Cornea and Julien Bourgeois. A framework for efficient performance prediction of distributed applications in heterogeneous systems. *The Journal of Supercomputing*, 62(3):1609–1634, 2012.

[28] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed systems concepts and design*, chapter Chapter 1, pages 1–5. Pearson, fifth edition, 2012.

[29] Geourge Coulouris, Jean Dollimore, and Tim Kindberg. Chapter 1 - characterization of distributed systems. In *Distributed Systems Concepts and Design*, pages 1 – 25. Addison Wesley, fourth edition edition, 2009.

[30] J. A. Davis, G. R. Mudalige, S. D. Hammond, J. A. Herdman, I. Miller, and S. A. Jarvis. Predictive analysis of a hydrodynamics application on large-scale cmp clusters. *Comput. Sci.*, 26(3-4):175–185, June 2011.

[31] A. Deshmeh, J. Machina, and A. Sodan. Adept scalability predictor in support of adaptive resource allocation. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, 2010.

[32] Octopus Team Developers. Octopus, 2000. Accesed : 27/11/2014.

[33] Evgueni Dodonov and Rodrigo Fernandes de Mello. A novel approach for distributed application scheduling based on prediction of communication events. *Future Generation Computer Systems*, 26(5):740 – 752, 2010.

[34] M. Drozdowski and L. Wielebski. Isoefficiency maps for divisible computations. *Parallel and Distributed Systems, IEEE Transactions on*, 21(6):872–880, 2010.

[35] Gerhard Engelbrecht and Siegfried Benkner. A service-oriented grid environment with on-demand qos support. In *Proceedings of the 2009 Congress on Services - I*, SERVICES '09, pages 147–150, Washington, DC, USA, 2009. IEEE Computer Society.

[36] Eric Deconinck, Menghui H. Zhang, Danny Coomans, , and Yvan Vander Heyden*. Classification tree models for the prediction of bloodbrain barrier passage of drugs. *Journal of Chemical Information and Modeling*, 46(3):1410–1419, 2006. PMID: 16711761.

[37] Chih-Tien Fan, Yue-Shan Chang, Wei-Jen Wang, and Shyan-Ming Yuan. Execution time prediction using rough set theory in hybrid cloud. In *Ubiquitous Intelligence Computing and 9th International Conference on Autonomic Trusted Computing (UIC/ATC), 2012 9th International Conference on*, pages 729–734, Sept 2012.

[38] Jesus Flores-Contreras and Hector A. Duran-Limon. Performance prediction of parallel applications: A systematic literature review. *Parallel Computing*, 2016. Manuscript submitted for publication (copy on file with author).

[39] D. Gianni, G. Iazeolla, and A. D'Ambrogio. A methodology to predict the performance of distributed simulations. In *Principles of Advanced and Distributed Simulation (PADS), 2010 IEEE Workshop on*, pages 1 –9, may 2010.

[40] Manfred Gilli, Dietmar Maringer, and Enrico Schumann. Chapter twelve - heuristic methods in a nutshell. In Manfred GilliDietmar MaringerEnrico Schumann, editor, *Numerical Methods and Optimization in Finance*, pages 337 – 379. Academic Press, San Diego, 2011.

[41] Theodore Colton Gordon K. Smyth, Peter Armitage. Polynomial approximation, encyclopedia of biostatistics, 1998. http://www.statsci.org/smyth/pubs/EoB/bap064.pdf.

[42] S. Goswami and L.K.P. Bhaiya. Brain tumour detection using unsupervised learning based neural network. In *Communication Systems and Network Technologies (CSNT), 2013 International Conference on*, pages 573–577, April 2013.

[43] Ananth Y. Grama, A. Gupta, and V. Kumar. Isoefficiency: measuring the scalability of parallel algorithms and architectures. *Parallel Distributed Technology: Systems Applications, IEEE*, 1(3):12–21, 1993.

[44] Alessia Gualandris, Simon Portegies Zwart, and Alfredo TiradoRamos. Performance analysis of direct n-body algorithms for astrophysical simulations on distributed systems. *Parallel Computing*, 33(3):159–173, 2007.

[45] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic review of fault prediction performance in software engineering. *Software Engineering, IEEE Transactions on*, PP(99):1, 2011.

[46] Jens Happe, Heiko Koziolek, and Ralf Reussner. Parametric performance contracts for software components with concurrent behaviour. *Electron. Notes Theor. Comput. Sci.*, 182:91–106, June 2007.

[47] Simon Hawkins, Hongxing He, Graham Williams, and Rohan Baxter. *Outlier Detection Using Replicator Neural Networks*, pages 170–180. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[48] P. He, L. Chen, and X. H. Xu. Fast c4.5. In *Machine Learning and Cybernetics, 2007 International Conference on*, volume 5, pages 2841–2846, Aug 2007.

[49] Jordy Hendrikx, Matt Murphy, and Terry Onslow. Classification trees as a tool for operational avalanche forecasting on the seward highway, alaska. *Cold Regions Science and Technology*, 97:113 – 120, 2014.

[50] Liang Hu, Xi-Long Che, and Si-Qing Zheng. Online system for grid resource monitoring and machine learning-based prediction. *Parallel and Distributed Systems, IEEE Transactions on*, 23(1):134 –145, jan. 2012.

[51] Bert Hubert. The wonder shaper, January 2016.

[52] M. Hudik and M. Hodon. Modeling, optimization and performance prediction of parallel algorithms. In *Computers and Communication (ISCC), 2014 IEEE Symposium on*, volume Workshops, pages 1–7, June 2014.

[53] Eui-Nam Huh and Lonnie R. Welch. Adaptive resource management for dynamic distributed real-time applications. *J. Supercomput.*, 38(2):127–142, November 2006.

[54] Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206:79 – 111, 2014.

[55] Mark A. Hall. Ian H. Witten, Eibe Frank. *Data mining, Practical Machine Learning Tools and Techniques*, chapter One, pages 3–8. Morgan Kaufmann Publishers, third edition edition, 2011.

[56] Engin Ipek, BronisR. de Supinski, Martin Schulz, and SallyA. McKee. An approach to performance prediction for parallel applications. In JosC. Cunha and PedroD. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science*, pages 196–205. Springer Berlin Heidelberg, 2005.

[57] V. P. Ivannikov, S. S. Gaisaryan, A. I. Avetisyan, and V. A. Padaryan. Estimation of dynamical characteristics of a parallel program on a model. *Program. Comput. Softw.*, 32(4):203–214, July 2006.

[58] Stephen A. Jarvis, Daniel P. Spooner, Helene N. Lim Choi Keung, Junwei Cao, Subhash Saini, and Graham R. Nudd. Performance prediction and its use in parallel and distributed computing systems. *Future Gener. Comput. Syst.*, 22:745–754, August 2006.

[59] A. Jayakumar, P. Murali, and S. Vadhiyar. Matching application signatures for performance predictions using a single execution. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 1161–1170, May 2015.

[60] Sol Ji Kang, Sang Yeon Lee, and Keon Myung Lee. Performance comparison of openmp, mpi, and mapreduce in practical problems. *Advances in Multimedia*, 2015(5756873):9, March 2015.

[61] V. Kecman. Support vector machines - an introduction. In Lipo Wang, editor, *Support Vector Machines: Theory and Applications*, volume 177 of *Studies in Fuzziness and Soft Computing*, pages 1–47. Springer Berlin Heidelberg, 2005.

[62] C.T. Kelley. *Iterative Methods for Optimization.* Society for Industrial and Applied Mathematics, 1999. http://www.caam.rice.edu/žhang/caam554/KelleyBooks/fr18_book.pd.

[63] D.J. Kerbyson and K.J. Barker. A performance model of direct numerical simulation for analyzing large-scale systems. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1824–1830, 2011.

[64] G. Kestor, R. Gioiosa, and D. Chavarra-Miranda. Prometheus: scalable and accurate emulation of task-based applications on many-core systems. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pages 308–317, March 2015.

[65] Roger E. Kirk. *Statistics an Introduction*, chapter 6, pages 161–169. Thomson Wadsworth, fifth edition edition, 2008.

[66] Barbara Kitchenham. Procedures for performing systematic reviews. Technical report, Keele University and Empirical Software Engineering National ICT Australia Ltd., 2004.

[67] R. Kubert and S. Wesner. Service level agreements for job control in high-performance computing. In *Computer Science and Information Technology (IMCSIT), Proceedings of the 2010 International Multiconference on*, pages 655–661, 2010.

[68] D.A. Kumar and S. Murugan. Performance analysis of indian stock market index using neural network time series model. In *Pattern Recognition, Informatics and Mobile Engineering (PRIME), 2013 International Conference on*, pages 72–78, Feb 2013.

[69] S. Kundu, R. Rangaswami, K. Dutta, and M. Zhao. Application performance modeling in a virtualized environment. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1 –10, jan. 2010.

[70] Seyong Lee, Jeremy S. Meredith, and Jeffrey S. Vetter. Compass: A framework for automated performance modeling and prediction. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 405–414, New York, NY, USA, 2015. ACM.

[71] Charles H. Lehmann. *Algebra, 9th Edition*. Limusa, 1997.

[72] Bin Li, Lu Peng, and Balachandran Ramadass. Accurate and efficient processor performance prediction via regression tree based modeling. *J. Syst. Archit.*, 55:457–467, October 2009.

[73] Haifeng Li. A queue theory based response time model for web services chain. In *Computational Intelligence and Software Engineering (CiSE), 2010 International Conference on*, pages 1–4, Dec 2010.

[74] Jing Li, Xinpu Ji, Yuhan Jia, Bingpeng Zhu, Gang Wang, Zhongwei Li, and Xiaoguang Liu. Hard drive failure prediction using classification and regression trees. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 383–394, June 2014.

[75] R. Li, X. m. Wei, and X. w. Yu. The improvement of c4.5 algorithm and case study. In *Computational Intelligence and Design, 2009. ISCID '09. Second International Symposium on*, volume 2, pages 190–192, Dec 2009.

[76] Yuhong Li and Weihua Ma. Applications of artificial neural networks in financial economics: A survey. In *Computational Intelligence and Design (ISCID), 2010 International Symposium on*, volume 1, pages 211–214, 2010.

[77] Pangfeng Liu and S.N. Bhatt. Experiences with parallel n-body simulation. *Parallel and Distributed Systems, IEEE Transactions on*, 11(12):1306–1323, Dec 2000.

[78] Oleg Lobachev, Michael Guthe, and Rita Loogen. Estimating parallel performance. *Journal of Parallel and Distributed Computing*, 73(6):876 – 887, 2013.

[79] Wei-Yin Loh. Fifty years of classification and regression trees. *International Statistical Review*, 82(3):329–348, 2014.

[80] Gabriel Marin and John Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 2–13, New York, NY, USA, 2004. ACM.

[81] João Maroco, Dina Silva, Ana Rodrigues, Manuela Guerreiro, Isabel Santana, and Alexandre de Mendonça. Data mining methods in the prediction of dementia: A real-data comparison of the accuracy, sensitivity and specificity of linear discriminant analysis, logistic regression, neural networks, support vector machines, classification trees and random forests. *BMC Research Notes*, 4(1):1–14, 2011.

[82] Rafael Martí and Gerhard Reinelt. Heuristic methods. In *The Linear Ordering Problem*, volume 175 of *Applied Mathematical Sciences*, pages 17–40. Springer Berlin Heidelberg, 2011.

[83] A. Matsunaga and J. Fortes. On the use of machine learning to predict the time and resources consumed by applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 495–504, May 2010.

[84] Rodrigo F. Mello and Laurence T. Yang. Prediction of dynamical, nonlinear, and unstable process behavior. *J. Supercomput.*, 49(1):22–41, July 2009.

[85] Mesoscale and Microscale Meteorology (MMM) Division of NCAR. Weather research and forecasting model, 2007. Accesed : 27/11/2014.

[86] Sandia National Laboratories Michael A. Heroux. Finite element mini application minife, 2013. Accesed : 27/11/2014.

[87] EdsonToshimi Midorikawa, HelioMarci Oliveira, and JeanMarcos Laine. Pempis: A new methodology for modeling and prediction of mpi programs performance. *International Journal of Parallel Programming*, 33(5):499–527, 2005.

[88] T. Miu and P. Missier. Predicting the execution time of workflow activities based on their input features. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 64–72, Nov 2012.

[89] Samuel K. Moore. Multicore cpus: Processor proliferation. IEEE Spectrum, January 2011. http://spectrum.ieee.org/semiconductors/processors/multicorecpus-processorproliferation.

[90] K. Muralidharan and D. Vasudevan. Applications of artificial neural networks in prediction of performance, emission and combustion characteristics of variable compression ratio engine fuelled with waste cooking oil biodiesel. *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, 37(3):915–928, 2015.

[91] A. Najah, A. El-Shafie, O.A. Karim, and AmrH. El-Shafie. Application of artificial neural networks for water quality prediction. *Neural Computing and Applications*, 22(1):187–201, 2013.

[92] Kalaivany Natarajan, Jiuyong Li, and Andy Koronios. *Data mining techniques for data cleaning*, pages 796–804. Springer London, London, 2010.

[93] Robert Nisbet, John Elder, and Gary Miner. Chapter 13 - model evaluation and enhancement. In Robert NisbetJohn ElderGary Miner, editor, *Handbook of Statistical Analysis and Data Mining Applications*, pages 285 – 312. Academic Press, Boston, 2009.

[94] Adam Oliner, Archana Ganapathi, and Wei Xu. Advances and challenges in log analysis. *Queue*, 9(12):30:30–30:40, dec 2011.

[95] Marcio Seiji Oyamada, Felipe Zschornack, and Flvio Rech Wagner. Applying neural networks to performance estimation of embedded software. *Journal of Systems Architecture*, 54(12):224 – 240, 2008.

[96] Javier Panadero, Alvaro Wong, Dolores Rexachs, and Emilio Luque. A tool for selecting the right target machine for parallel scientific applications. *Procedia Computer Science*, 18:1824 – 1833, 2013. 2013 International Conference on Computational Science.

[97] A.K. Parakh, M. Balakrishnan, and K. Paul. Performance estimation of gpus with cache. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2384–2393, 2012.

[98] W. Pfeiffer and N.J. Wright. Modeling and predicting application performance on parallel computers using hpc challenge benchmarks. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12, 2008.

[99] Panu Phinjaroenphan, Savitri Bevinakoppa, and Panlop Zeephongsekul. A method for estimating the execution time of a parallel task on a grid node. In PeterM.A. Sloot, AlfonsG. Hoekstra, Thierry Priol, Alexander Reinefeld, and Marian Bubak, editors, *Advances in Grid Computing - EGC 2005*, volume 3470 of *Lecture Notes in Computer Science*, pages 226–236. Springer Berlin Heidelberg, 2005.

[100] Rosario M. Piro, Andrea Guarise, Giuseppe Patania, and Albert Werbrouck. Using historical accounting information to predict the resource usage of grid jobs. *Future Gener. Comput. Syst.*, 25(5):499–510, May 2009.

[101] Hema Prem and N.R.Srinivasa Raghavan. A support vector machine based approach for forecasting of network weather services. *Journal of Grid Computing*, 4(1):89–114, 2006.

[102] Zhang Qiuhao, Li Wei, Sun Baiqing, and Guo Hongche. Obstacles pattern recognition based on bp neural network. In *Mechatronics and Automation (ICMA), 2012 International Conference on*, pages 2574–2578, Aug 2012.

[103] Lavanya Ramakrishnan and Daniel Reed. Predictable quality of service atop degradable distributed systems. *Cluster Computing*, pages 1–14, 2009. 10.1007/s10586-009-0078-y.

[104] S. Ruggieri. Efficient c4.5 [classification algorithm]. *IEEE Transactions on Knowledge and Data Engineering*, 14(2):438–444, Mar 2002.

[105] Seyed Masoud Sadjadi, Javier Figueroa, Raju Rangaswami, Javier Delgado, Hector Duran, and Xabriel J Collazo-Mojica. A modeling approach for estimating execution time of long-running scientific applications. *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2008.

[106] Julio Sahuquillo, Houcine Hassan, Salvador Petit, Jos Luis March, and Jos Duato. A dynamic execution time estimation model to save energy in heterogeneous multicores running periodic tasks. *Future Generation Computer Systems*, pages –, 2015.

[107] H. A. Sanjay and Sathish Vadhiyar. Performance modeling of parallel applications for grid scheduling. *J. Parallel Distrib. Comput.*, 68(8):1135–1145, August 2008.

[108] A. Sarangi and A.K. Bhattacharya. Comparison of artificial neural network and regression models for sediment loss prediction from banha watershed in india. *Agricultural Water Management*, 78(3):195 – 208, 2005.

[109] Sena Seneviratne and David C. Levy. Task profiling model for load profile prediction. *Future Generation Computer Systems*, 27(3):245 – 255, 2011.

[110] HPC Advisory Council Chairman Gilad Shainer. Hpc advisory council, 2016. Revised on April 30th, 2016.

[111] S. Sharkawi, D. DeSota, R. Panda, S. Stevens, V. Taylor, and Xingfu Wu. Swapp: A framework for performance projections of hpc applications using benchmarks. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1722–1731, 2012.

[112] Shuichi Shimizu, Raju Rangaswami, Hector A. Duran-Limon, and Manuel Corona-Perez. Platform-independent modeling and prediction of application resource usage characteristics. *Journal of Systems and Software*, 82(12):2117 – 2127, 2009.

[113] S. Singh, J. Harini, and B.R. Surabhi. A novel neural network based automated system for diagnosis of breast cancer from real time biopsy slides. In *Circuits, Communication, Control and Computing (I4C), 2014 International Conference on*, pages 50–53, Nov 2014.

[114] W. Smith. Prediction services for distributed computing. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10, 2007.

[115] Sukhdeep Sodhi, Jaspal Subhlok, and Qiang Xu. Performance prediction with skeletons. *Cluster Computing*, 11(2):151–165, 2008.

[116] Enqiang Sun and David Kaeli. Aggressive value prediction on a gpu. *International Journal of Parallel Programming*, pages 1–19, 2012.

[117] Smitha. T and V. Sundaram. Article: Classification rules by decision tree for disease prediction. *International Journal of Computer Applications*, 43(8):6–12, April 2012.

[118] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining, (Classification: Basic concepts, Decision Trees, and Model Evaluation)*, pages 145–166. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[119] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed systems principles and paradigms*, chapter Chapter 1, pages 1–5. Prentice-Hall, second edition, 2007.

[120] T.J. Taylor. A tutorial on chaos in control theory. In *Decision and Control, 1992., Proceedings of the 31st IEEE Conference on*, pages 2102–2106 vol.2, 1992.

[121] Burger W. Thomas. Intel multi-core processors: Quick reference guide. http://software.intel.com/en-us/articles/intel-multi-core-processors-quick-reference-guide, 2009. Consulted May 2013.

[122] S. Upadhyaya, K. Farahmand, and T. Baker-Demaray. Comparison of {NN} and {LR} classifiers in the context of screening native american elders with diabetes. *Expert Systems with Applications*, 40(15):5830 – 5838, 2013.

[123] Robert van de Geijn Victor Eijkhout, Edmond Chow. *Introduction to High Performance Computing*, chapter 3, page 135. The Saylor Foundation, 2014.

[124] Ian H. Witten, Eibe Frank, and Mark A. Hall. Chapter 11 - the explorer. In Ian H. WittenEibe FrankMark A. Hall, editor, *Data Mining: Practical Machine Learning Tools and Techniques (Third Edition)*, The Morgan Kaufmann Series in Data Management Systems, pages 407 – 494. Morgan Kaufmann, Boston, third edition edition, 2011.

[125] Ian H. Witten, Eibe Frank, and Mark A. Hall. Chapter 5 - credibility: Evaluating what's been learned. In Ian H. WittenEibe FrankMark A. Hall, editor, *Data Mining: Practical Machine Learning Tools and Techniques (Third Edition)*, The Morgan Kaufmann Series in Data Management Systems, pages 147 – 187. Morgan Kaufmann, Boston, third edition edition, 2011.

[126] Rich Wolski, Neil T Spring, and Jim Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(56):757 – 768, 1999.

[127] A. Wong, D. Rexachs, and E. Luque. Parallel application signature for performance analysis and prediction. *Parallel and Distributed Systems, IEEE Transactions on*, 26(7):2009–2019, July 2015.

[128] Tzu-Tsung Wong. Performance evaluation of classification algorithms by k-fold and leave-one-out cross validation. *Pattern Recognition*, 48(9):2839 – 2846, 2015.

[129] Ming Wu and Xian-He Sun. Grid harvest service: A performance system of grid computing. *Journal of Parallel and Distributed Computing*, 66(10):1322 – 1337, 2006.

[130] Rongteng Wu, Jizhou Sun, and Jinyan Chen. Parallel execution time prediction of the multitask parallel programs. *Perform. Eval.*, 65(10):701–713, October 2008.

[131] Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand, and Dan Steinberg. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37, 2007.

[132] H. Xiong, Gaurav Pandey, M. Steinbach, and Vipin Kumar. Enhancing data analysis with noise removal. *IEEE Transactions on Knowledge and Data Engineering*, 18(3):304–319, March 2006.

[133] Da-yu Xu, Shan-lin Yang, and Ren-ping Liu. A mixture of hmm, ga, and elman network for load prediction in cloud-oriented data centers. *Journal of Zhejiang University SCIENCE C*, 14(11):845–858, 2013.

[134] Leo T. Yang, Xiaosong Ma, and Frank Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 40–, Washington, DC, USA, 2005. IEEE Computer Society.

[135] Eduardo Javier Huerta Yero and Marco Aurélio Amaral Henriques. Contention-sensitive static performance prediction for parallel distributed applications. *Perform. Eval.*, 63(4):265–277, May 2006.

[136] Jidong Zhai, Wenguang Chen, and Weimin Zheng. Phantom: Predicting performance of parallel applications on large-scale parallel machines using a single node. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 305–314, New York, NY, USA, 2010. ACM.

[137] Hongli Zhang, Panpan Li, Zhigang Zhou, Xiaojiang Du, and Weizhe Zhang. A performance prediction scheme for computation-intensive applications on cloud. In *Communications (ICC), 2013 IEEE International Conference on*, pages 1957–1961, June 2013.

[138] Yongli Zhang and Yuhong Yang. Cross-validation for selecting a model selection procedure. *Journal of Econometrics*, 187(1):95 – 112, 2015.

[139] Yuanyuan Zhang, Wei Sun, and Yasushi Inoguchi. Predict task running time in grid environments based on cpu load predictions. *Future Generation Computer Systems*, 24(6):489 – 497, 2008.