2019 20

Víctor Medel Gracia

Application Driven MOdels for Resource Management in Cloud Environments

Departamento
Informática e Ingeniería de Sistemas

Director/es
ARRONATEGUI ARRIBALZAGA, UNAI

ISSN 2254-7606

© Universidad de Zaragoza
Servicio de Publicaciones

ISSN 2254-7606

Tesis Doctoral

APPLICATION DRIVEN MODELS FOR RESOURCE MANAGEMENT IN CLOUD ENVIRONMENTS

Autor

Víctor Medel Gracia

Director/es

ARRONATEGUI ARRIBALZAGA, UNAI

UNIVERSIDAD DE ZARAGOZA
Informática e Ingeniería de Sistemas

2018

Application Driven Models for Resource Management in Cloud Environments

TESIS DOCTORAL

Víctor Medel Gracia

Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Agosto 2018

Application Driven Models for Resource Management in Cloud Environments

Memoria que presenta para optar al título de Doctor en Informática

Víctor Medel Gracia

Dirigida por el Doctor

Unai Arronategui Arribalzaga

Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Agosto 2018

Application Driven Models for Resource Management in Cloud Environments

Víctor Medel Gracia

Quiero dedicar esta tesis a mi mujer Cristina,
por todo su amor y cariño

Resumen

El despliegue y la ejecución de aplicaciones de gran escala en sistemas distribuidos con unos parametros de Calidad de Servicio adecuados necesita gestionar de manera eficiente los recursos computacionales. Para desacoplar los requirimientos funcionales y los no funcionales (u operacionales) de dichas aplicaciones, se puede distinguir dos niveles de abstracción: i) el nivel funcional, que contempla aquellos requerimientos relacionados con funcionalidades de la aplicación; y ii) el nivel operacional, que depende del sistema distribuido donde se despliegue y garantizará aquellos parametros relacionados con la Calidad del Servicio, disponibilidad, tolerancia a fallos y coste económico, entre otros. De entre las diferentes alternativas del nivel operacional, en la presente tesis se contempla un entorno *cloud* basado en la virtualización de contenedores, como puede ofrecer Kubernetes.

El uso de modelos para el diseño de aplicaciones en ambos niveles permite garantizar que dichos requerimientos sean satisfechos. Según la complejidad del modelo que describa la aplicación, o el conocimiento que el nivel operacional tenga de ella, se diferencian tres tipos de aplicaciones: i) aplicaciones dirigidas por el modelo, como es el caso de la simulación de eventos discretos, donde el propio modelo, por ejemplo Redes de Petri de Alto Nivel, describen la aplicación; ii) aplicaciones dirigidas por los datos, como es el caso de la ejecución de analíticas sobre *Data Stream*; y iii) aplicaciones dirigidas por el sistema, donde el nivel operacional rige el despliegue al considerarlas como una caja negra.

En la presente tesis doctoral, se propone el uso de un *scheduler* específico para cada tipo de aplicación y modelo, con ejemplos concretos, de manera que el cliente de la infraestructura pueda utilizar información del modelo descriptivo y del modelo operacional. Esta solucion permite rellenar el hueco conceptual entre ambos niveles. De esta manera, se proponen diferentes métodos y técnicas para desplegar diferentes aplicaciones: una simulación de un sistema de Vehículos Eléctricos descrita a traves de Redes de Petri; procesado de algoritmos sobre un grafo que llega siguiendo el paradigma *Data Stream*; y el propio sistema operacional como sujeto de estudio.

En este último caso de estudio, se ha analizado cómo determinados parámetros del nivel operacional (por ejemplo, la agrupación de contenedores, o la compartición de recursos entre contenedores alojados en una misma máquina) tienen un impacto en las prestaciones. Para analizar dicho impacto, se propone un modelo formal de una infrastructura operacional concreta (Kubernetes). Por último, se propone una metodología para construir índices de interferencia para caracterizar aplicaciones y estimar la degradación de prestaciones incurrida cuando dos contenedores son desplegados y ejecutados juntos. Estos índices modelan cómo los recursos del nivel operacional son usados por las applicaciones. Esto supone que el nivel operacional maneja información cercana a la aplicación y le permite tomar mejores decisiones de despliegue y distribución.

Abstract

Deploying and executing large-scale applications in a distributed system, with a feasible Quality of Service (QoS) parameters, requires an efficient resource management in such platforms. In order to decouple the functional and the non-functional –or operational– requirements of an application, we can distinguish two levels of abstraction: i) the functional level, which considers those requirements related to the application functionalities; and ii) the operational level, which is platform-dependant, and it guarantees those parameters related to QoS, availability, fault tolerance and economic cost, among others. Among the available alternatives in the operational level, in this PhD thesis, we focus on a cloud environment based on container virtualisation. Kubernetes, which is studied thoroughly in our work, is an example of such systems.

The usage of models to design applications or to describe systems in both levels guarantees that both types of requirements are fulfilled. Depending on the complexity of the model and/or on the information about applications managed by the operational layer, we distinguish three kinds of applications: i) model-driven applications, like Discrete Event Simulations, in which the model, for instance, a High Level Petri Net, describes the application; ii) data-driven applications, like the execution of analytics over data streams; and iii) system-driven applications, where the operational level makes decisions on the deployment and execution of those applications considering them as a black-box.

In the present PhD thesis, we propose the use of a specific scheduler for each kind of application and model, given different use cases. This scheduler allows the client of the infrastructure to use information about the functional model an about the specific operational platform. This solution fills the gap between both levels without losing the transparency of the cloud abstraction. In this regard, we propose several methodologies and models to deploy different kind of applications: a simulation about an Electric Vehicle system modelled with Petri Nets; analytics over a data stream graph; and the operational platform as a subject of study in itself.

In this last use case, we have analysed how certain parameters which are related to the operational level –for instance, grouping containers together, or sharing resources between containers hosted en the same machine– have an impact on performance. To analyse this impact, we propose a formal model to describe the operational infrastructure –in our case, Kubernetes–. Finally, we propose a methodology to build interference indices to characterise applications and to estimate the performance degradation produced when two containers are scheduled together. These interference indices model how the operational resources are used by deployed applications. This situation provides additional information to the operational level to take better allocation decisions.

Acknowledgments

I would like to express my gratitude to those people who have contributed in some way to this thesis. During these years, I have had the opportunity to collaborate and learn from many researchers and professionals who had make possible this work.

In the first place, I want to thank my advisor, Unai, for helping and guiding me in this process. Thanks for all the hours of discussion that have contributed to the thesis. Thanks to Rafa, José Ángel and José Manuel, for all the ideas given in coffee breaks. In the second place, thanks to Omer Rana, that dedicated three months so I could make a stay in the Cardiff University under his supervision. Your advices and collaboration have been very useful.

I would also like to thank Carlos, Guillermo, Marta and Jorge, who had carried out their end-of-degree project under my advisor. I hope you have learned something from me, as I have learned working with you.

Finally, thanks to my family for their support, specially to my brother, Diego. Thanks for spending your time helping me in the edition of the final document. Thanks to my wife, Cristina. It has been a long journey and your support has been essential to finish this work. It would not have been possible without you.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Among intelligences, however, there are some that contain more universal forms and others that contain less universal forms. This is because the forms that are in the second, lower universal intelligences in the mode of particulars are in the first intelligences in the mode of universals.

Saint Thomas Aquinas

Resource management in distributed systems enables the possibility of executing applications which describe complex systems with feasible performance and cost parameters. Computational resources –i.e., CPU, RAM, I/O file system, network– are shared by all applications executed in the distributed platform and by all users/clients which deploy applications. The manner in which the resources are shared and how each application uses them can have an impact on the performance of the other ones. Thus, the competition between resources should be regulated or controlled by using different techniques. These mechanisms ensure certain Quality of Service (QoS) parameters to the client, which are defined in a Service Level Agreement (SLA) between the platform operator and each tenant. Therefore, the platform should guarantee the isolation between applications to hide how the resources are shared. Overall, an increment in the isolation will produce a degradation in certain performance metrics, so a trade-off between both criteria should be achieved. For instance, a reservation mechanism gives a high isolation mechanism for the resource sharing problem. However, from the point of view of the cloud operator, the resource occupation metric decreases.

The deployed application might describe a complex system. A complex system is a system which can be described as many components which interact with each other. Each of these components might describe in turn a whole and meaningful system with its own subsystems or components. The concept of complex system is highly coupled with the concept of large-scale systems. The scale problem arises due to the following reasons [1]: i) the number of attributes or concepts necessary to describe the system; ii) the description of the system behaviour laws; iii) the evolution in time of the system. These reasons not only are applicable to each component, but also to the interactions between them. There are a lot of research domains which are suitable to be analysed as a complex system, namely: capacity planning and resource management in cloud environments; Internet of Things (IoT) and sensor systems [2,3]; smart grids [4–6] and Electric Vehicles (EV) systems [7]; and smart cities and smart building applications [8], among others.

Modelling complex systems allows reasoning about their properties and estimating how they evolve. The big scale of the entities and the complexity and intricacy of the relationships entail the usage of very different models. These models need computational resources to be simulated or executed in a platform, and due to their scale, the platform should be a distributed one. At this point, we can distinguish the description of the system –with its attributes and its behavioural laws–, called the functional level; and the distributed platform –with the computational resources–, called the operational level [9]. The use of models on the latter might help designing the system –for example, structural analysis, model checking or simulations might be used for this purpose.

There are different approaches to the operational level. However, the cloud computing paradigm is being widespread and it has been generalised to a variety of heterogeneous domains and applications. The nature of cloud computing abstraction, which involves transparency, fault tolerance and scalability, makes the cloud a suitable environment to deploy heterogeneous large-scale models.

The complexity in usage terms of cloud platforms, the absence of interface standardisation and the innate transparency of the cloud paradigm, prevent the development of a single framework to deploy complex models or applications easily. The solution is to analyse the properties of the given model and to exploit these properties to improve the deployment parameters.

The usage of public cloud infrastructure (e.g. Amazon EC2[1], Google Compute Engine (GCE)[2]), which are widely available, or even private solutions, attaches an intrinsic cost to the execution of applications. This cost depends on the total resources used –with a pay per use strategy– and it might be affected by how the application is deployed. For instance, if a highly coupled model is split in a distributed environment among several machines, there will be a high usage of the network infrastructure and the cost might increase. Exploiting certain properties of the application models can help to choose suitable deployment parameters to reduce the total cost or to improve the performance.

As we have said before, the cloud abstraction not only provides a platform to execute the functional description –or functional requirements– of a model, but it also has to guarantee certain QoS metrics to that application –non-functional requirements–. Moreover, those non-functional requirements related to performance are accomplished through the flexibility given by the infrastructure. The isolation between resources at this level is achieved through virtualisation techniques. In this regard, container technology is substituting Virtual Machine (VM) technology in several domains, due to its flexibility in deployment. Its usage allows the user to reduce the tarification period –for instance, GCE can be used for fractions of a second– and it allows the deployment of a new stateless computing paradigm –as

[1]https://aws.amazon.com/es/ec2/
[2]https://cloud.google.com/compute/

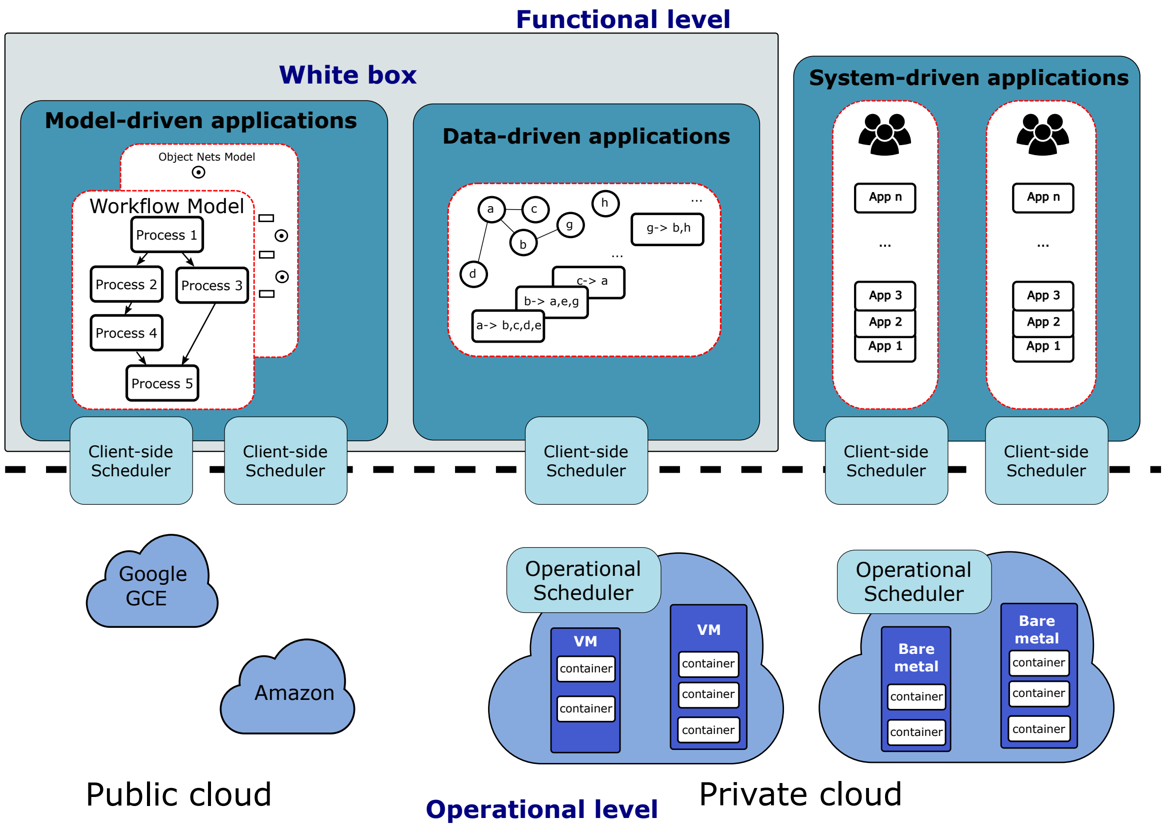


Figure 1.1: Methodology for executing complex applications in distributed environments.

Amazon Lambda[3]–. Due to the relative novelty of the container virtualisation, the field of resource allocation using containers has not been addressed by researchers thoroughly [10].

The resource efficiency is not only motivated by the economic impact, but also by the nature of the Big Data applications. The huge amount of information and the need to process the information to get its value force us to develop systems and models that can deal with it in short periods of time. This situation is nearly modelled by the Data Stream paradigm. Data –or even applications or models– arrives to the computing infrastructure with a variable rate –or a variable SLA– and they need to be analysed as soon as they arrive.

1.1 From the Functional Level to the Operational Level

From an abstract point of view, we can consider a distributed execution environment where applications arrive to be executed under certain QoS and cost requirements (Figure 1.1). The functionality requirements of the applications are modelled on the functional level and the operational level represents the infrastructure where the functional instances are mapped to be executed [9]. The manner in which these instances are mapped to instances

[3]https://aws.amazon.com/es/blogs/compute/container-reuse-in-lambda/

of the operational level might have a heavy impact on several performance and economic variables of the entire system –for instance, the QoS given to certain applications, the overall utilisation of the cloud and the size of the physical and virtual resources in the cloud, among others.

Applications deployed and executed in such a system can be classified in three categories, depending on their intrinsic characteristics and on the details given by the model:

- **Model-driven applications**. Applications arriving might be as complex as possible, so it might be impossible to execute them in a single machine with feasible QoS parameters. This means that the application should be distributed into the computational resources. On the whole, the partition decisions will be conditioned by the properties of the model. Examples of this kind of applications are the simulations of complex systems or agent applications. The application model is susceptible to be analysed and the information that it provides can be as detailed as possible –we refer to this information as micro-information or micro-parameters–. The complexity of the application makes their execution almost impossible in a single operational instance with feasible QoS guarantees. Therefore, the application –or the model– should be distributed among different instances. This distribution can be made randomly, using statistical information or exploiting certain properties inherent to the model. For example, a microservice architecture might be considered as a whole application to be deployed in a cloud. The relationships between services, either causal or non-causal, can be modelled as a graph, and we can partition the graph attending to certain properties, for example, minimising the cutting edges. For the simulation domain, High Level Petri Nets model the parallel behaviour of the application, and this information can be used to partition the model.

- **Data-driven applications**. Certain applications are high data consuming and, in fact, the data model conditions the execution of the application. For instance, since moving the data might have an important economic cost, some applications should be executed where the data is located. Furthermore, the instances arriving to the cloud might represent pre-processing or even raw data to be analysed by further applications. Bringing the data as near as possible to the operational instances where the final analysis will take place has a deep impact on the performance of those applications. Overall, the Data Stream paradigm models this behaviour and it has to deal with a nearly endless stream of data which: i) has a variable input rate; ii) arrives without any control of its order from the point of view of the system; and iii) cannot be easily retrieved [11]. Other architectural models include the Lambda architecture, where an off-line and on-line process is made to achieve certain guarantees [12]. Graph analysis over social media data is an example of this layer. The operational instance where the application is deployed –the analytic application– is mostly given by the instance where the data is located.

- **System-driven applications**. We consider that the instances arriving are meaningful applications and the system cannot make any assumption about them. Each distributed application is considered as a black box and the operational system manages how they are deployed. Therefore, how they are modelled or implemented is abstracted in this level. Models used in this layer might use macro-information to manage the instances –mainly, statistical information about execution times, deadline information, etc.–. The use of models helps to analyse how applications arrive to be processed and how the distributed processing might affect the macro-parameters. This information can be used to take decisions about QoS parameters –e.g. deadlines or throughput–, the size of the computational resources or scheduling and allocation policies. In this level, statistical models have been widely used in the literature; specially, those which model the arrival of instances with an exponential distribution function. The Petri Net abstraction might be useful to analyse the scheduling and the resource allocation problem from an analytical and a simulation perspective.

For these kind of applications, not only the partition or distribution scheme has an impact on the performance, but also the operational scheduler. Overall, we can see that there is a gap between the functional and the operational level. The transparency of the cloud abstraction hides operational information to the functional layer, and the fact that, in most of the systems, the applications are seen as a black box, hide functional or structural information to the operational layer. To keep the transparency property of the cloud abstraction, this gap needs to be filled with specific algorithms and mechanisms, related to the properties of the model, instead of using a general purpose scheduling which, in practice, will take inefficient allocation decisions. In this PhD thesis, we propose to fill this gap by bringing the scheduling algorithm near the functional level –the client-side scheduler components in the figure–. This allows the tenants of the system to use operational information.

1.2 Thesis Overview

In this PhD thesis, it is analysed the manner in which certain properties of the model, which describes an application deployed in a cloud environment, can be used to increase the performance of the execution time and/or minimise the total resource usage and the total cost. Thus, we propose different approaches to fill the gap between the functional and operational layers. Due to the heterogeneity of the models which might be used to characterise these applications, we use those which are suitable for the specific objectives of each use case. These use cases include: streamed graph partitioning for obtaining analytics, Electrical Vehicle charge and discharge applications, and a general framework for processing tasks in a multi-tenant cloud environment based on container virtualisation.

Each of these scenarios allows us to exemplify the importance of exploiting the model to improve the performance in the layers presented in the previous section:

- Electrical Vehicle charge/discharge simulation (Chapter 3). In this chapter, a system where a set of EV interacts between each other is analysed. The relationships between the modelled entities –e.g. EVs and Charging Stations– are described by using a formal model –Object Nets–. The properties of the model are used to deploy the simulation of the system among several machines. These kind of simulations can be used to make optimal decisions about the dimension of real systems.

- Streamed Graphs and graph partitioning (Chapter 4). In this chapter, we analyse how the graph partition –equivalently, the data distribution among different machines– affects the performance of the execution of analytics over that machines. We focus on how to use the streamed graph model to minimise the resource usage, namely, the total network usage and the RAM. We propose the summary graph abstraction, which compresses a graph to achieve certain resources guarantees.
- The cloud environment as a research domain. We can consider the entire system layer –where black box applications arrive to be processed– in private clouds (Chapters 6, 5, 7). We focus on a container-based infrastructure, where different tasks arrive to be executed. These tasks might be self-contained applications or they might belong to a workflow or to a bigger model which has been partitioned in a previous step –for instance, they can be pieces of a simulation (as we propose in Chapter 3) or partitions of a big graph (as proposed in Chapter 4)–. We use statistical and Object Nets models to minimise the performance degradation caused by scheduling decisions. These decisions can be made by using structural or operational information:
  - Structural information. This information is about design decisions taken by the developer; for instance, which containers are highly coupled. These decisions are close to the functional level. In Chapter 5, we propose an Object Nets model to analyse how different design and deployment parameters impact on the performance. The formal model is fed with data obtained from a real cluster, which allows us to obtain several design rules.
  - Operational information. This information deals with the manner in which the operational resources –e.g. physical machines, containers– are shared in a cluster. First, we propose an informal model to bring near the client or the application developer the scheduling decisions (Chapter 6). To accomplish this, the developer should know certain operational properties and give that information to the scheduler. For instance, this information might be which hardware resources are more intensively used in his application –i.e. CPU, Network, among others–. The idea is to help the scheduler to split applications which use the same kind of resource among different machines. In Chapter 7, we refine this informal model by proposing a statistical model to characterise applications through timed indices. These indices model thoroughly the hardware resource usage of the application, and they are used to analyse how applications interfere between them in a container cluster.

1.3 Motivation and Contributions

The process of the scientific method consists on formulating a research question, defining a hypothesis from that question, and testing experimentally the acceptance or rejection of the hypothesis, to make further analysis. The accomplishment of this PhD thesis has been contextualised inside this scheme, therefore, it is important to define the research questions and the hypothesis which determine its further development.

The underlying research question is: *Can application-driven models be used to improve the resource efficiency of the execution of applications in a distributed environment?*. The use of models to describe applications or systems has been widely addressed by the literature and there is no doubt in its utility. They can be used in the upper abstraction level –the functional level– to help to design, implement and test applications. However, in this PhD thesis, we propose to use those models to take decisions about how applications are executed in a distributed infrastructure, namely, the cloud. Of course, this premise is quite general, and we have focused on certain applications in different domains, which have specific characteristics. As the considered applications and models are heterogeneous, it is not possible to propose a general purpose analysis framework. The use of a certain model rather than other one will be determined by the specific characteristics of the application and by the expected property to be optimised; i.e., the minimisation of the overall execution time, the maximisation of the infrastructure usage, etc. All contributions aim to fill the gap between the functional and operational levels with several resource and performance restrictions.

The specific research questions and contributions regarding each domain are the following ones:

- **Q1**. *Is there a conceptual gap between large-scale models and the cloud platform where the simulation is executed?*. The simulation of large scale models in cloud environments can achieve non-functional requirements, for instance, those regarding elasticity and fault tolerance. How these high-level models –e.g. Object or Colour Petri Nets– are mapped to cloud resources is a research topic which has not been addressed thoroughly in the literature. We propose to fill this gap with the "elaboration step", where high-level primitives are elaborated into low level elements –i.e. sequential state machines with message passing– which can be used to partition the model easily.

- **Q2**. *Is it possible to distribute data-driven applications by using performance and economic criteria at the same time?*. In the context of Big Data, data-driven applications might involve the processing or the execution of analytics over a continuous flux of information, described by the Data Stream paradigm. If the incoming data models the relationship between different entities, it can be modelled as a streamed graph. At this point the distribution of this data over the operational layer can be seen as the graph partition problem. How graphs are partitioned to make further analysis has been studied previously, as we present in Section 2.3. Works focused on classical graphs and on streamed graphs. However, the use of the proposed algorithms and techniques has an impact on the performance –the partition process might have a computational cost– and on the resource usage. Moreover, if the resource usage increases, the economic cost will rise. The contributions on the streamed graph partition topic are:

  - We define the concept of "Summary graph", which enables us to determine the trade-off between the quality of the partition and the resource usage for the streamed graph partitioning problem.

  – We analyse the distributed architectures to process graph data and we set resource bounds for those architectures.

- **Q3**. *Do design decisions of applications have an impact on the performance when they are executed in a container-based environment?*. Deploying and executing applications in a container-based virtualisation system requires to adjust several deployment and configuration parameters. Moreover, new abstractions, such as Kubernetes' pod concept, arise. The impact of these parameters on the performance of containerised applications is not banal, and several design decisions might affect the performance and resource usage. In this regard, our contributions are:

  – Proposing an Object Nets model which describes the behaviour of Kubernetes, a container management system. The model is fed with data from a real cluster to make further simulations or estimations.

  – Analysing the degradation caused by the pod abstraction used in Kubernetes.

  – Proposing several rules to structure the deployment of containerised applications in Kubernetes to minimise the overhead caused by Kubernetes.

- **Q4**. *Is it useful to bring near the cloud user/developer the scheduling algorithm to improve the scheduling decisions?*. When containerised applications are deployed in multiple tenant container management systems such as Kubernetes –or alternatively, Docker Swarm–, tenants and the scheduler are not aware of the resource usage of their applications. We propose giving several primitives to clients to describe their applications. Then, the scheduler can use this information to take better allocation decisions to improve the overall performance of the cluster.

- **Q5**. *Are the isolation mechanisms available in container-based environments, such as Kubernetes, sufficient to share resources efficiently?*. Kubernetes gives some basic mechanisms which allow sharing resources in a cluster with certain guarantees. We show that certain resources cannot be completely isolated. Furthermore, the CPU reservation mechanism might lead to scenarios where a lot of resources remain unused and the overall performance of the system is worsened.

- **Q6**. *Is there a performance degradation when multiple containers are executed concurrently in the same machine?*. If so, *Is it possible to analyse it and to isolate the causative factors to minimise the degradation?*. The interference experienced among containers sharing hardware resources in the same machine, that causes a degradation in their execution, is difficult to model and to analyse due to the low level resources involved –e.g. memory hierarchy–. The state of the art proposals regard VM virtualisation, and there are few works which address this issue in container systems. The contributions of this PhD thesis in this field are:

- Proposing a model to build time-evolving indices which describe low-level resource usage.
- Proposing a methodology to characterise an application using four timed indices. The granularity of these indices can be adjusted depending on further usage. Moreover, these indices are based on statistical models and they are meaningful.
- Proposing a methodology to use these indices to estimate the overall interference between containers in the same machine.

1.4 Publications

- V. Medel and U. Arronategui,"Resource Efficiency to Partition Big Streamed Graphs", in *Parallel and Distributed Computing (ISPDC), 2015 14th International Symposium on. IEEE*, 2015, pp. 120-129.

- V. Medel and U. Arronategui, "Real-time Partition of Streamed Graphs for Data Mining over Large Scale Data", *in Advances in Information Mining and Management (IMMM), 2015 5th International conference on, 2015*, pp. 41-48.

- V. Medel, U. Arronategui, J. Á. Bañares, and J. M. Colom, "Distributed Simulation of Complex and Scalable Systems: from Models to the Cloud", in *Lecture Notes in Computer Science. Economics of Grids, Clouds, Systems, and Services. Springer, 2016* vol. 10382, pp. 278-292.

- V. Medel, O. Rana, U. Arronategui, R. Tolosana-Calasanz, J. Á. Bañares, "Modelling performance & resource management in kubernetes", in *Proceedings of the 9th International Conference on Utility and Cloud Computing. ACM, 2016*, pp. 257-262.

- V. Medel, O. Rana, J. Á. Bañares, and U. Arronategui, "Adaptive Application Scheduling under Interference in Kubernetes" in *Proceedings of the 9th International Conference on Utility and Cloud Computing. ACM, 2016*, pp. 426-427. Best Poster Award.

- V. Medel, C. Tolón, U. Arronategui, R. Tolosana-Calasanz, J. Á. Bañares, and O. F. Rana, "Client-side scheduling based on application characterization on kubernetes" in *Lecture Notes in Computer Science. Economics of Grids, Clouds, Systems, and Services. Springer, 2017* vol. 10537 pp. 162-176.

- V. Medel, R. Tolosana-Calasanz, J. Á. Bañares, U. Arronategui, and O. F. Rana, "Characterising Resource Management Performance in Kubernetes" *Journal of Computers & Electrical Engineering*, vol. 68, pp. 286-297, 2018. JCR Impact Factor 2017: 1.747 Q2.

Chapter 2

State of the Art

Not only is there an art in knowing a thing, but also a certain art in teaching it.

Cicero

In this PhD thesis, we study how we can fill the gap between the functional and the operational level. To exemplify how the deployment of model-driven, data-driven and system-driven applications can be improved by taking into account properties related to specific models, we have focused on different use cases. We have analysed two applications: the simulation of Distributed Event Systems and graph analytics over a streamed graph. The related work presented in this chapter can be classified in two blocks: works related to those application scenarios and work related to the operational level. The latter focuses on container performance comparison, container scheduling and interference in those platforms.

2.1 Models to Describe or Characterise Distributed Applications

The intrinsic characteristics of distributed applications and the properties which are going to be analysed determine the use of a certain model rather than another one. The Petri-Net model [13] allows analysing and describing the concurrent and dynamic behaviour of applications and components. The basic Petri-Net model –also called Place-Transition Nets– can be enriched with the following additional semantic:

- **Time information**. The basic Petri-Net model considers that all transitions are fired instantly, and it has no consideration regarding time. There are several models that attach time to transitions, for instance, Timed PNs (TPNs) [14] attach a deterministic time, and Stochastic PNs (SPNs) [15] attach a stochastic one. Both models give the primitives needed to build a Discrete Event Simulation system.
- **Token information**. In the basic PN model, tokens represent the current marking of the net and they determine the state of the system. In the Colored Petri-Net

model [16], tokens have types, and they can model variables moving around the net. Tokens might also be a Petri-Net, as it happens in the Net-within-Nets paradigm [17]. These models are suitable to describe high level behaviour of applications and have been used to describe agent systems and complex systems.

Although High Level Petri-Nets enable the possibility of modelling implicitly the data as tokens, it is difficult to use that kind of models to exploit properties related to the data structure. For this purpose, graph models are more suitable. These models allow reasoning about how a set of entities –called vertices– interact or are related between them –those relationships are called edges–. Among other applications, the graph model has been used widely to describe social interactions between users or the relationship between tasks in a platform –e.g. to describe a scientific workflow–. The basic semantics of graphs can be enriched with temporal or dynamic information –evolving graphs [18]– and with certain availability restrictions –streamed graphs–. Depending on the application domain, other high level models can be used to describe the interaction and/or the coordination between entities. For example, the TOSCA model (Topology and Orchestration Specification for Cloud Applications)[1], enables the description of applications composed of cloud-based web services and the relationships between them.

All of these models need detailed information about the application or about the data. How this data is obtained and the high level of the analytics carried over these models make them useless for certain application domains. In these situations, statistical or analytical models can be used to analyse certain properties of applications. Machine Learning algorithms, which need a lot of data to capture certain behaviours, are an example of these kind of models. They are useful to classify applications or to make certain estimations, but, they are generally quite opaque and it is difficult to reason about those models –this reasoning might include questions about how the system has reached certain states or how certain properties are achieved–. Other statistical models include the Exploratory Factor Analysis techniques and the Confirmatory Factor Analysis, which is explained in detail in Chapter 7. Therefore, analytical models can be used to feed high level models –e.g. transitions in Petri-Nets–, to analyse the entire system.

2.2 Distributed Discrete Event simulation

The parallel execution of Discrete Event simulation has been addressed by [19]. This work proposes splitting the sequential simulator into a set of Logical Processes (LPs) which interact exchanging timed messages. Each LP can process internal events in order; however, due to the fact that they are mapped in different processes or machines, errors resulting from out-of-order event processing might arise. These errors are referred to as *causality errors*, and this problem is called the synchronisation problem [20]. An overview of the extensive literature about the synchronisation problem can be found in [21]. To avoid causality errors, two approaches have been developed:

[1] https://www.oasis-open.org/committees/tosca/

- **Conservative simulation**. The conservative approach assumes that messages arriving on each incoming link are stored in a FIFO queue with events in a time-stamp order. The processing of local events is interleaved with the processing of messages in incoming links. Because messages in each FIFO are sorted, the LP can guarantee adherence to the local causality constraint by repeatedly processing the message containing the smallest time-stamp. The problem arises if some input link queue is empty, because the LP must wait, and a deadlock can appear if each LP is waiting on the incoming link containing the smallest time-stamp. The solution to this problem is the estimation of a lower bound on the time-stamp of messages the LP will send, and sending a *null message* with a time-stamp called *lookahead*. The lookahead values are closely related to the physical system.

- **Optimistic simulation**. The optimistic approach does not postpone simulation steps that could yield causality errors, trying in this way to accelerate the simulation. When causality errors are detected in a LP by the reception of an out-of-order *straggler message*, the LP state is restored to a previous safe state, and sends *antimessages* to cancel sent messages. The cost of this approach is in the memory to store safe states, and in sending antimessages.

The improvement on the performance of the execution of simulations can be achieved through: i) improving the simulation engine; and ii) improving how the entire model is partitioned into LPs and how these LPs are distributed into processes or machines. The majority of the literature has focused on the first approach, proposing different algorithms and techniques which estimate the lookahead value –for conservative simulations– or minimise the amount of rollbacks –for optimistic simulations–. In this regard, in [22], authors propose to ignore rollbacks. Their experiments show that the error was small –about 5%– but the used model has only 3 LPs, which makes extending their results for large scale simulations difficult. Another approach proposes to relax the event causality [23]. In [24], it is proposed to use a global progress window to adjust the number of events which a LP is allowed to execute, in order to minimise rollbacks. In [25], authors use a bulk-synchronous strategy to synchronise the execution with supersteps, where the number of events executed in a superstep is adjusted dynamically. In regard to conservative simulations, in [26], a lower bound for the degree of lookahead exposed by each LP is inferred from the Petri-Net structure and from timed firing transitions.

To the best of the author's knowledge, the partition problem for simulation models has been addressed by few works [26, 27].

2.3 Streamed Graphs and Graph Partitioning

Graph algorithms have been widely treated in literature. Graph streaming model has been described in [28]. It represents the sequential access to graph elements instead of random access, due to the size of the graph. In this regard, several papers propose how to adapt graph algorithms to the streaming paradigm [29, 30]. They take considerations about the complexity of different typical algorithms –triangle count [31], property checking, connectivity, etc.– and they calculate the required space bound and the number of times an element of a stream is processed. In several works, some data stream restrictions are relaxed to obtain more flexible models: Semi-stream [28], W-Stream [32], Best-Order Stream [33], Sort-Stream, etc.

Some theoretical works have analysed the relationship between the number of passes over an off-line stream and the memory size needed for some specific algorithms and applications for graph processing (e.g. PageRank [34]). As we explain in Chapter 4, real time considerations cannot be solved with these solutions.

Graph partitioning has been proposed in Metis [35] and Scotch [36]. However, they are not suitable to partition real time streamed graphs because they need the whole graph and their execution time is too slow for large graphs.

On the other hand, the graph partition problem in a streaming environment is a NP-Complete problem [37]. To the best of our knowledge, there are a few works which focus on this problem [38, 39]. In these works, they get approximated solutions by using some heuristics with low processing time. In Fennel [39], the obtained partition is as good as the Metis solution for some graphs. In their experiments, the worst case is for *amazon0312* dataset and they get 6% more cutting edges. The problem of these heuristics is the $O(n)$ memory bound and the underlying architecture, which does not scale well for big graphs.

Finally, the summarisation of large scale graphs has been addressed by using different data mining and sampling techniques [40–42]. The use of these techniques is not suitable for a streamed environment with real time constraints, as it entails the identification of certain graph structures –e.g. cliques– and/or the use of algorithms which need several passes over the graph.

2.4 Container Scheduling

Container scheduling in cloud environments is an emergent research topic. Google has developed and used several schedulers for large scale infrastructures over past years based on a centralised architecture [43, 44]. They are oriented towards internal global use or as a global service provider. Some works have been proposed to improve the algorithms available as a standard in practical cloud infrastructures, such as Kubernetes[2], Mesos[3] [45] and Docker Swarm[4]. However, in [46], the authors point out the lack of works about resource management with containers, and they propose a scheduling framework that provides useful management functions that can be used to apply customised scheduling policies, mainly, in local environments. We can find a few more works such as [47]. In this work, authors propose a generational scheduler to map containers to different generations of servers based on the requirements and properties learned from running containers. It shows an improved energy efficiency over Docker Swarm built-in scheduling policies. They assume that containers can

[2]https://kubernetes.io/
[3]http://mesos.apache.org/
[4]https://docs.docker.com/swarm/,https://github.com/docker/swarmkit

be migrated, which may not be consistent in all container management platforms. The work in [48] uses an ant colony meta-heuristic to improve application performance, also over Docker Swarm base scheduling policies. Authors in [49] address the problem of scheduling micro-services across multiple VM clouds to reduce overall turnaround time and total traffic generated. Finally, in [50], the authors introduce a container management service which offers an intelligent resource scheduling that increases deployment density, scalability and resource efficiency. It considers an holistic view of all registered applications and available resources in the cloud. The main difference from our approach is that we focus on the client side requirements to optimise a subset of applications and resources. The work we propose in Chapter 6 improves the performance of deployed containers based on the interference made by other containers scheduled in the same machine. The idea is to use non-formal information given by the developer –or the client– of that application.

Several approaches exist outside container technologies, however it is not straightforward how they work inside a container management system. For instance, CASH [51] is a Context Aware Scheduler for Hadoop. It takes into account the heterogeneity of the computational resources of a Hadoop cluster as well as the job characteristics, whether they are CPU or I/O intensive. In [52], the authors use k-means as a classification mechanism for Hadoop workloads (jobs), so that jobs can be automatically classified based on their requirements. They plan to improve the performance of their scheduler by separating data intensive and computation intensive jobs when the classification is performed. On the other hand, job interference was also studied in Hadoop, and acknowledged as one of the key performance aspects. In [53], the authors analyse the high level of interference between interactive and batch processing jobs and they propose a scheduler for the virtualisation layer, designed to minimise interference, and a scheduler for the Hadoop framework. Similarly, the authors in [54] analyse the interference occurring among Apache Spark jobs in virtualised environments. They develop an analytical model to estimate the interference effect, which could be exploited to improve the Apache Spark Scheduler in the future.

2.5 Container Performance

We summarised the most important work in performance evaluation of cloud environments in Table 2.1. Most of it focused on performance comparison of Virtual Machines rather than on containers as the unit of computation. A set of workloads were developed to obtain the usage of memory, CPU, networking and storage [61, 62, 67]. However, all these works are based on experimental results that do not have an analytical model which supports reasoning about performance decisions.

To the best of our knowledge, no work has tackled the container performance topic from a rigorous analytical perspective. Even in more traditional cloud technologies, few works are based on formal models [64]. An analytical model, based on Continuous Time Markov Chains, is used in [65] to study the provisioning performance of microservices. This model is validated with experiments deployed in a specific microservice platform, which is based on the execution of Docker containers, with Docker Swarm providing cluster management support and Amazon EC2 as the virtual machine backend. However, the proposed analytical

Work	**Model**	**Virtualisation infrastructure**	**Experimental framework**
[55]	Experimental approach	VMs	Hyper-V, KVM, vSphere, Xen
[56]	Experimental approach	VMs	KVM, Xen, vBox
[57,58]	Experimental approach	Containers and VMs	Docker, KVM, Xen, LXC
[59]	Experimental approach	Containers and VMs	LXC, OpenVZ, VServer, Xen
[60]	Experimental approach	Containers	LXC
[61,62]	Experimental approach	Containers	Docker, KVM
[63]	Experimental approach	Containers	Docker, Weave
[64,65]	Continuous Markov Chains (Exponential PDF)	Containers inside VMs	Docker Swarm inside Amazon EC2
[66]	Net-within-Nets (any PDF)	VMs	Simulations
Our work	Net-within-Nets (any PDF) Experimental approach	Containers	Kubernetes over bare metal

Table 2.1: Summary of the related work for container performance analysis. The table includes the kind of model they proposed, the assumptions of the model, the virtualisation infrastructure that they used and the experimental framework.

model assumes that the rate of workload generation follows a Poisson distribution –the time between arrivals has an exponential probability density function (PDF)– which may yield to non-realistic scenarios. In our work, we can link transitions on the Petri Net model with any PDF or with functions obtained from real application benchmarking, so more realistic scenarios can be modelled.

In [66], the researchers propose an iterative and step-wise refinement methodology that begins with the modelling of functional requirements. Afterwards, they model control and data flow together with the specification of the computational resources available. As a result the performance of the distributed system can be formally analysed by considering all the aspects that can affect performance.

All these models need temporal information to be fed. Previous works, mainly in the context of execution of traditional VMs in Clouds, were focused on obtaining this kind of information [55, 56]. Several works provide a performance comparison between virtual machines and containers [57, 58, 62]. These works show a better performance and resource usage in containers. Scalability performance in Kubernetes was studied in [68], where the results show that containers perform 22 times faster than VMs for the provisioning action.

The container platform evaluation, as Docker and LXC in [59], shows performance issues being improved and this approach being considered for High Performance Computing

(HPC). The researchers conclude that the performance is near-native for both technologies. In [60], the authors present several variables –e.g. Linux kernel version– which have an impact on the performance of containers. However, both works focus on the execution time as a performance metric and they do not consider other measures, which can be significant –e.g. deployment time–. They also assume that the image is pre-loaded in the machines and they do not provide a full methodology to use their results.

As we have described in our work, Kubernetes introduces a new abstraction, the pod. To the best of our knowledge, the performance degradation of this abstraction has not been studied by literature. Analysis of nested containers is the closest research field. In [61, 63], network performance degradation was observed in some configurations due to a deployment based on full nested containers. This degradation is caused by the usage of network virtualisation technologies –Linux Bridge or OpenvSwicht– twice or by the usage of Software Defined Networks and encryption. However, the Kubernetes pod abstraction gives a common space port to all containers –and therefore the same IP address to all services– so the performance degradation may be different.

Finally, several performance metrics of Kubernetes[5] were reported by the Kubernetes team. In a cluster of 100 nodes, their results show a 99th percentile pod startup time below three seconds. This value is consistent with our results. Their experiments show how a Kubernetes cluster behaves when the scale of the deployment is increased for several simple metrics –pod start time, end-to-end response time and response time of different API operations–; however, how the deployment is structured and the impact of grouping containers inside a pod is not analysed.

2.6 Interference in Container Platforms

The isolation in the performance aspects –not in the security ones– is a concept similar to the interference. It has been studied in the context of cloud computing in [69]. However, this work focuses on multi-tenant applications and it does not take into account the infrastructure level –e.g. container virtualisation or VM virtualisation– and their results are obtained from simulations. Prior works focus on the interference of VMs inside the same physical host or multicore processors [70]; specially for cache memory and memory bandwidth resources [71–73]. The majority of these works use the LLC counter to analyse the cache behaviour. Other analysed resources are the I/O file system [74, 75] and the network [76]. [77] analyses the performance isolation of Virtual Machines on the same physical host and make some estimations based on similarity between applications. In [78], a technique called Bubble-Up is used to estimate the execution time under contingency in VM clusters. They model each application as a sensitivity profile which is normalised to a single score, called bubble score. Additionally, Hubbub-Scale [79] calculates an interference index based on the value of the degradation to propose a elasticity controller which is aware of VM interference.

Although most of the previous work has analysed the contention problem for VM virtual-

[5] https://kubernetes.io/blog/2016/03/1000-nodes-and-beyond-updates-to-kubernetes-performance-and-scalability-in-12/

isation, there are several interesting works which analyse the interference between containers. In Paragon [80], they use a collaborative filtering algorithm to determine the influence of several Sources of Influence (SoIs) in applications. With this information, the scheduler tries to choose the optimal machine to allocate the application. In contrast to our work, they consider the interference of a container to be constant over the entire execution, which might lead to non-optimal decisions for the scheduling algorithm. Additionally, they consider that the host machines in the cluster are heterogeneous. However, they do not give any meaningful metric like our interference profiles. Their machine learning model gives several variables without a direct correspondence to physical variables, which makes analysing the model and using it to reason about different situations difficult.

Furthermore, our interference indices or the estimated degradation might be used by the scheduler to give some priorities –or penalties– for certain hosts. For instance, in ARQ [81], they introduce the concept of Quality of a Resource needed by an application. Low values of that metric correspond to insensitive influence applications. This metric is used as a priority for the scheduler. Moreover, the interference in cloud environments not only has performance consequences, but also security ones. In Bolt [82], they propose a methodology to determine which application is co-scheduled with another one based on the interference between them.

Chapter 3

Distributed Simulation of Complex and Scalable Systems: from Models to the Cloud

Uncertainty is worse than all.

Alexandre Dumas

In this chapter, we present an example of applications describing a complex system, which has to be executed in a distributed system. The application is considered as a white box, so we assume that there is a model which describes certain properties of the application and we use certain properties of the model to guide the partition and distribution into operational instances. Specifically, we consider the simulation of complex Discrete Event Systems (DES) using High Level Petri Nets. The Petri Net model is an executable one, which allows describing directly the simulation application. The simulation domain chosen in this chapter is an Electric Vehicle charge and discharge system. The system is modelled as a set of state machines –the EVs sequential activities– which use conservative and non-conservative resources –e.g. the street capacity and the electric power, respectively–. The interaction between EVs is defined by sharing those resources in the system and by exchanging messages.

Other complex applications that can be seen as a white box are the description of computer systems using a micro-service architecture or business logic models which describe the relationship between processing steps. In both cases, the most suitable model to describe relationships is the graph model, and several partition techniques –with several performance criteria such as cost or performance metrics– should be used to improve further executions or analytics over the model. Nevertheless, the High Level Petri Net model presented in this chapter not only describes the relationships between entities, but also their concurrent execution and the communication protocol between them. This kind of model allows us to describe richer scenarios where other parameters should be considered.

The chapter is structured as follows. Section 3.1 introduces the need of using models for describing complex systems. Section 3.2 presents basic notions of distributed simulation of

Timed Petri Nets. Section 3.3 describes the proposed approach for modelling complex systems. Then, the elaboration process to obtain a simulation model in the cloud is illustrated in Section 3.4. Finally, concluding remarks are discussed in Section 3.5.

3.1 Models to Describe Complex-Systems

The IoT and Cyber-Physical Systems (CPS) allow tightly interconnected and coordinated physical and computational processes to work together effectively. As environments become more complex, it is not viable anymore to engineer individual self-contained systems. Thus, the necessity of integrating large-scale Systems of Systems (SoS) involving several domains arises –e.g. Smart Factories, Smart Cities, Smart Power Grid, Intelligent Transportation Systems, etc. [83].

A fundamental challenge caused by the IoT, CPS and resulting large-scale SoS, is the need of models to cope with the complexity of current techno-socio-economic systems. This systems describe physical and computational interactions and integrate human behaviour into the processes with sustainability and economical requirements. Separation of concepts and lifting the level of abstraction have proven to be effective software engineering strategies to afford complexity. However, the interrelationship of these separate models is part of the essential complexity of the problem domain, and an early separation between different facets of the system design makes it difficult to assess the impacts and trade-offs of alternatives that affect all involved processes in complex domains [84].

In addition to the modelling of different facets and the interplay between them, a rigorous model-based approach is required to use them for a formal verification of design, coding and testing phases to detect defects in the requirements compliance. However, formal-model based analysis tools are only useful under certain assumptions or are insufficient to afford the study of even simple software systems. On the other hand, simulation may be useful to discover some (un)desirable behaviours; but, generally speaking, proving the (in)existence of some properties is not allowed. Therefore, the synergistic combination of simulation and formal models for functional, performance and economical analysis is necessary for an efficient and reliable design and/or optimisation. In our approach, we give a central role to Petri Net models describing the behaviour of the system including timing and cost information. The goal is to use these models in an intensive way for analysis and simulation purposes.

Independently of the formal model, as systems become increasingly sophisticated, the state space of the system becomes larger and the use of analysis and simulation techniques becomes impossible by a single-processor machine. Distributed simulation and cloud computing to scale to a huge amount of resources that can be rented (and disposed) in every moment seems the natural way to afford this scalability problem [85]. The intention of a *cloud-based simulation* service is to migrate the simulation software into the cloud, providing users with appropriate tools to hide low level details of this migration process from the modeller considering cost and performance requirements.

The main challenges of deploying a simulation in the cloud come from the nature of the SoS models. Each model view can represent a facet/subsystems describing, in a precise way, some behaviour. However, models resulting from the composition of different subsystems and facets tend to become an spaghetti-like specification. This characteristic is a serious handicap because parallel abstractions are based on exploiting some regular structures and declarative specifications that may be materialised on distributed systems [86]. Additionally to the difficulty for finding the maximum concurrency to scale the model, the partitioning of the model must preserve its original semantic taking into account low level details –e.g. synchronisation and state-management– [20], and must be a trade-off between performance and cost of resources.

3.2 Distributed Event Simulation of Complex Systems

Complex and scalable Discrete Event Systems (DES) require simulation and verification techniques during the design process, to prevent bad behaviours, to ensure that certain good properties hold, and to evaluate performance. Petri Nets (PNs) [13] have been pointed out as a good modelling tool, since many properties may be easily analysed in a great number of cases. Moreover, when formal analysis becomes impracticable, the model may be simulated. As a simulation tool, PNs allow the formulation of models with realistic features –as the competition for resources– absent in other paradigms, such as queueing networks.

Given a PN model of a DES, we simulate the system by *playing the token game* on that PN, i.e. by firing transitions as a result of the available tokens. This is also referred to as implementing the PN. The basic PN paradigm can be extended by attaching a deterministic or stochastic time interpretation –Timed PNs (TPNs) [14] or Stochastic PNs (SPNs) [15], respectively–. The implementation of the TPN or SPN yields a Discrete Event Simulation system. In fact, SPNs have been proposed as the minimal discrete event notation, and both TPNs and SPNs are in the scope of works on DESs and Parallel and Distributed DES (PDES) simulation [26, 27]. Observe that tokens in places represent the state, transition firings represent events, and timed transitions represent the duration of activities.

The construction of an application to simulate PNs requires: i) a representation of the net structure and the marking –state of the system that will be updated during the simulation–; and ii) the *simulation engine*, also called the Simulation Machine. The simulation engine follows a repetitive cycle that involves three stages: i) to test the enabling of transitions; ii) to fire some enabled transitions –which may include the simulation of some associated activity–; and iii) to update the marking –the state– of the PN, taking tokens out of the input places and putting new tokens in the output places of each fired transition. The first stage –the enabling test of transitions– may be such a time-consuming operation that it is worth being lightweighted. To reduce the costs of the enabling test, two main approaches exist:

- **Place-driven approaches**. Only the output transitions of some representative marked places are tested for firability. This gives a characterisation of the partial enabling of transitions.
- **Transition-driven approaches**. A characterisation of the enabling of transitions is supplied, and only enabled transitions are considered. The firing of a given transition

modifies the enabling conditions of the transitions connected to its input and output places. Overall, an explicit representation of the marking is not necessary.

A DES models or emulates the operation of a system that changes its state over time at the occurrence of discrete events. Therefore, a simulation model must represent states, state changes –events– and time. Between consecutive events, no change in the system is assumed to occur; thus the simulation can directly jump in time from one event to the next, accelerating in this way the emulation of the system. At this point, it is important to distinguish three different notions of time that appear in model simulation: i) *physical time* refers to time in the physical modelled system; ii) *simulation time* is the representation of physical time in the model, and finally, iii) *wallclock time* refers to the time when the execution of the simulation is executed [20].

Distributed simulations require to decompose a sequential simulation into a set of *logical processes* (LPs) that interact by exchanging time-stamped messages. Each LP guarantees that its internal events are processed in time-stamp order; however, errors resulting from out-of-order event processing due to the LP distribution over machines can arise. They are referred to as causality errors. As we have discussed in Section 2.2, two classical approaches have been proposed to guarantee *causal safety*: *conservative* and *optimistic* approaches [21].

Distributed simulation of Petri Nets will be based on many identical simulation engines –Simulation Machines– distributed over the execution platform, and each one devoted to the simulation of a subnet of the original one. Each subnet is represented in the corresponding simulation engine as a data structure and a set of variables for the local state. Therefore, each simulation engine plays the role of a LP in the context of Discrete Event Simulation, and the time-stamped messages will be the tokens generated by the firing of a transition that must update its output places belonging to other simulation engines. This means that the previous considerations about conservative/optimistic approaches must be taken into account in the context of simulation of PNs.

The efficiency of the distributed simulation of a PN is strongly dependent on the partition of the original model into subnets, each of them assigned to identical simulation engines which compose the distributed application. Partitioning requires, a priori, the identification of the *good* subnets in which the original one is divided. In this regard, strategies based on the identification of sequential state machines –computing for example p-semiflows [87] in an incremental way– or minimising the number of tokens to be interchanged between subnets are necessary to obtain efficient applications to simulate the PN. Nevertheless, during the execution, it is possible to observe congestion in the message flow between simulation engines; mutual exclusions in the execution of several simulation engines; or other kind of phenomena against the efficiency because of the interchange of messages. In these cases, thanks to a simulation based on identical simulation engines working on data structures and variables representing PNs, it is possible to achieve a dynamic reconfiguration of the initial partition: i) by fusion of the data structures of two simulation engines in only one; or ii) by splitting the data structure of a simulation engine into two separated data structures over two distinct simulation engines. This dynamic reconfiguration is not possible in simulation contexts where the system to be simulated is not a data structure –e.g. the system is a program that must be compiled.

Furthermore, the use of ordinary Timed PNs in the modelling of large complex DESs can lead to models of unmanageable size. This drawback has been reduced by using Object PNs which provide more compact and manageable descriptions. Nevertheless, this high level models introduce two additional characteristics to the simulation of Place/Transition Nets: modularity and hierarchy. In order to obtain an efficient simulation, instead of the direct emulation of the high level model, we propose to transform the original model to be simulated into a flat model composed by sequential state machines, each one simulated into a simulation engine, and interchanging tokens. This transformation process is called *elaboration* of the simulation model. It will be illustrated with the elaboration of an Object PN to the flat model of sequential state machines, but it can be developed from languages as that presented in [88,89] with semantics based on Object PNs.

3.3 A Petri Net based Modelling Methodology for Complex Systems. A Case Study

We propose an example in order to illustrate the kind of applications that can be modelled with the methodology presented in [88,89]. The chapter will be focussed in the construction of an Object PN. This methodology can be used with the component-oriented language designed by [89]. This language supports modularity and hierarchy components and it is designed to model large and complex systems.

Let us consider an *Electric Vehicle* (EV) hiring service to travel in a city without air and noise pollution (like London, Paris or Madrid). Users of the service take a vehicle in a so-called *EV station*, and they travel through the city until they leave the vehicle in other EV station. This basic scheme can lead to an unbalanced number of available vehicles in the EV stations which can reduce the quality of service to customers of EVs in certain service stations or, worse, cause a denial of service. The parking of the EV station has a limited capacity that cannot change. A control strategy which distributes the vehicles among the parking is needed to solve the unbalance problem.

The aim of the simulation is to analyse the behaviour of this system of EV service and the control policies to guarantee the quality of service for a given interval time –for example, a day–. To do that, it is necessary to know *a priori* what the program of scheduled activities of the customer who rents an EV for that interval is. For instance, the places and the arrival time of those places the user wants to visit. This abstraction will be called *agenda*. The nature of this program of activities is strictly *sequential* because there is only one user following a sequence of activities, but the agenda can contain flexible adaptations. This means that the user can program alternative behaviours that are taken depending on the availability of resources, or simply, depending on internal decisions of the driver that are taken with a certain probability.

The considered example throughout the chapter first requires the definition of the topology of the city, as shown in Figure 3.1. This definition is composed by a set of connected streets represented in the figure where the arrows represent the traffic direction. For traffic in each direction there is only one lane. If it is a one way street, then the traffic lane is

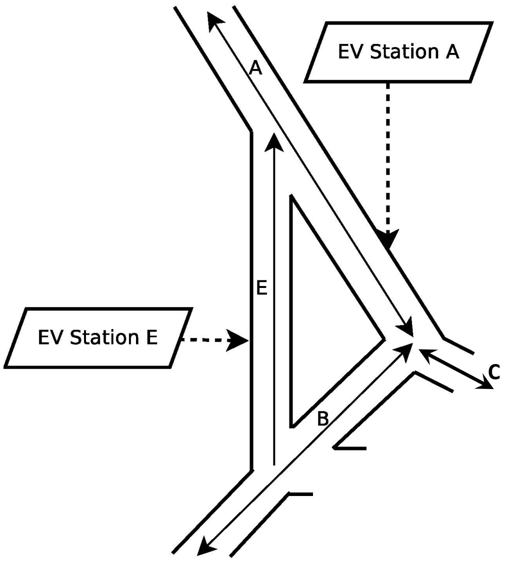


Figure 3.1: Topology of the city represented as a set of streets and the traffic direction.

designated by the letter shown in the drawing. If it is a two way road, the traffic in one direction will be designated with the letter that appears in the drawing and the incoming traffic with the same letter with the prime notation. There are *EV stations* in streets A and E that are identified with the name of the street. These two parkings have a limited capacity, where customers can find parked EVs that can be rented for their displacement, or leave the EV at the end of its ride whenever there is space in the parking station for it. The number of available EVs depends on the behaviour of the users over time and on the balance strategy between stations.

Note that the model to be constructed is hierarchical, meaning that we have a first level where we have a set of EVs, each one represented by means of its agenda of activities; and a second level corresponding to the urban space where the EVs move, interact, perform activities and so on. Therefore, a good choice for the construction of the model is the formalism of Object PN [90]. In this model, the Object Nets –the nets that are tokens inside a global Petri Net of a higher level, named System Net– are the agendas corresponding to the EVs moving in the urban space of streets and EVs stations. The System Net will represent the set of connected streets and EV stations through which the EVs can move. The Object Nets corresponding to agendas can be modelled directly by means of a state machine – in essence an automata representing the sequential program of activities to be performed by a user renting the EV–. The System Net will be constructed in a modular way from elementary modules representing the streets of the urban space and modules representing the EV stations.

The Petri Net module representing the street model is depicted in Figure 3.2. We begin modelling a conservative resource –with a **get** and **release** transitions– and we use it to build the street. These resources are acquired by processes and, when they are released, they can

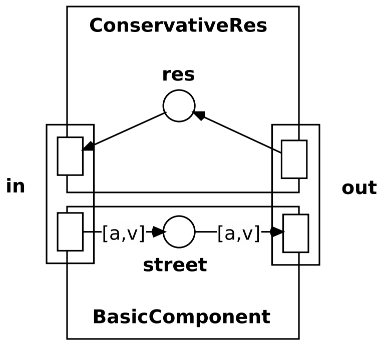


Figure 3.2: Petri Net module corresponding to a model of a street.

be used by other processes. The place **street** represents the state of a user –conceptually an agenda a– going through the street by EV v. In other words, this place will contain a token corresponding to the Object Net representing the agenda of the user of v when this vehicle is traversing the street. The model of a street could be refined by including the parking space in each street or additional parking, but for legibility and illustrative purposes, we present the unrefined model.

Instantiation of this basic module for each street of the urban topology, and a further connection of the instances according to the topology, enables to build the entire topology net where the nets representing the agendas of the EVs move. Figure 3.3 shows, for example, the composition of modules for street A. The **getA′fromA** transition represents to enter in street A' from street A. Each transition is synchronised with the agenda through an interaction, in this case $getA'A$. The number of nodes of the connectivity graph is the number of resources in the topology net, and the number of edges is the number of transitions between streets. For illustrative purposes, we have not shown the interaction between the EVs; however, with the proposed methodology it is straightforward. We would have to include the battery consumption in each transition, and to synchronise them with the EV model.

Figure 3.4 depicts the module corresponding to the model of the EV stations, with the corresponding resources –parking space and available PNs– and the output street. When a user takes an EV, the user releases a parking space and he takes an EV resource. The fusion between transitions is shown in Figure 3.4. For simplicity, the figure only shows station A. The **enterA** transition represents entering in the EV station and the **leaveA** transition represents leaving the station. The enter and leave transitions are synchronised with the user's agenda through inscriptions in the transitions.

The previous rules allow constructing the System Net of the full model of the system. The last part of the modelling task corresponds to the construction of the PN representing the agendas. These PNs will be the Object Nets that will act as tokens inside the System Net previously described in a succinct way. A user, throughout his agenda, appears in the system in the station place and his aim is to travel from that place to other station. However, during his route, the user may do some activities. A user's agenda represents a sequential

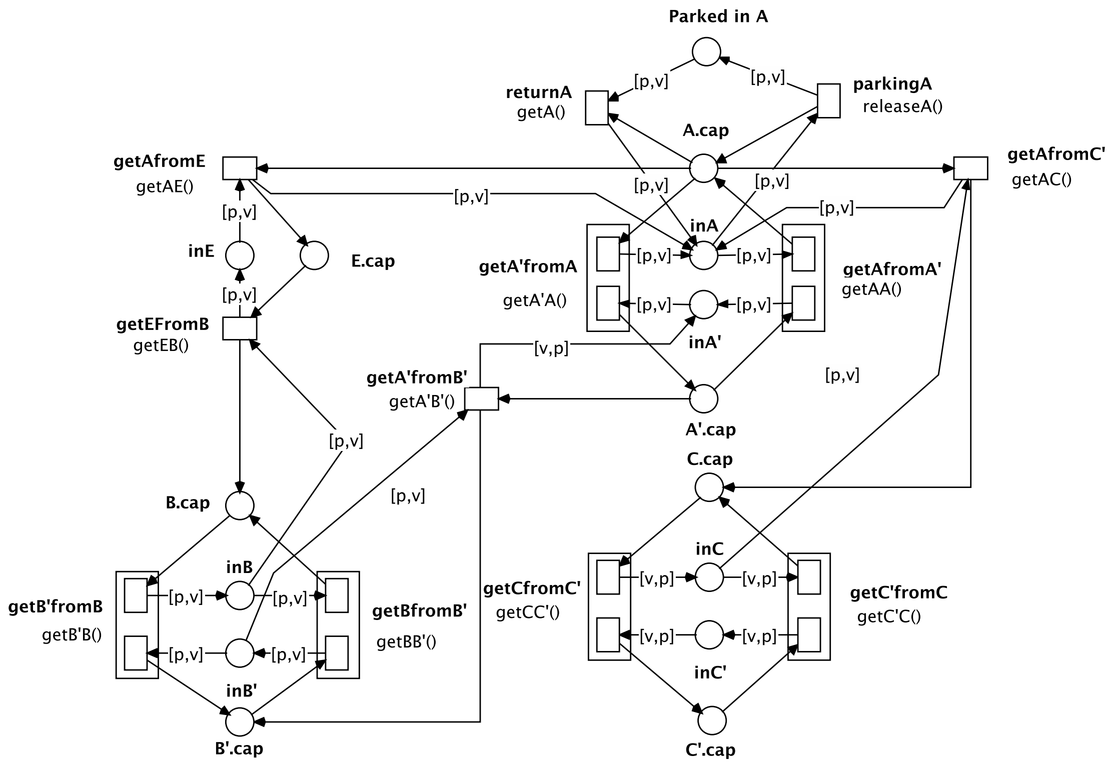


Figure 3.3: Petri Net model of the urban topology presented in Figure 3.1 connecting instances of the module of Figure 3.2.

program of activities that a user of the service has planned to do and the route that he follows. When a user begins his agenda, he takes an EV in a certain station and, when the agenda is finished, he leaves the EV in the same or in another station. The correspondence between the agenda and the EV is the token that is moving through the net –the $< a, v >$ token in Figures 3.2 and 3.4.

In Figure 3.5, we illustrate a user's agenda. In the example, the user starts in EV station A and he travels to street C to do an activity; then, he travels to EV station E and he leaves the EV. As he leaves the EV, we are not interested in the activity that the user does in that place or what he does next. The agenda nets are synchronised with the topology that we have presented before, through the inscriptions in the transitions –e.g. with the inscription *enterD* or *getA*–. More sophisticated routes and activities can be modelled with this methodology.

With this approach, we have to model every agenda that is going to happen in the simulation. If we suppose that users only travel between stations, the possible number of agendas depends on the number of stations –in our example, three stations and six agendas–. In this case, the token inside each agenda is the EV, and an agenda could be followed by more than one user. In any case, when we deploy the entire model the total number of agendas deployed is the same, because we have to represent every possible combination.

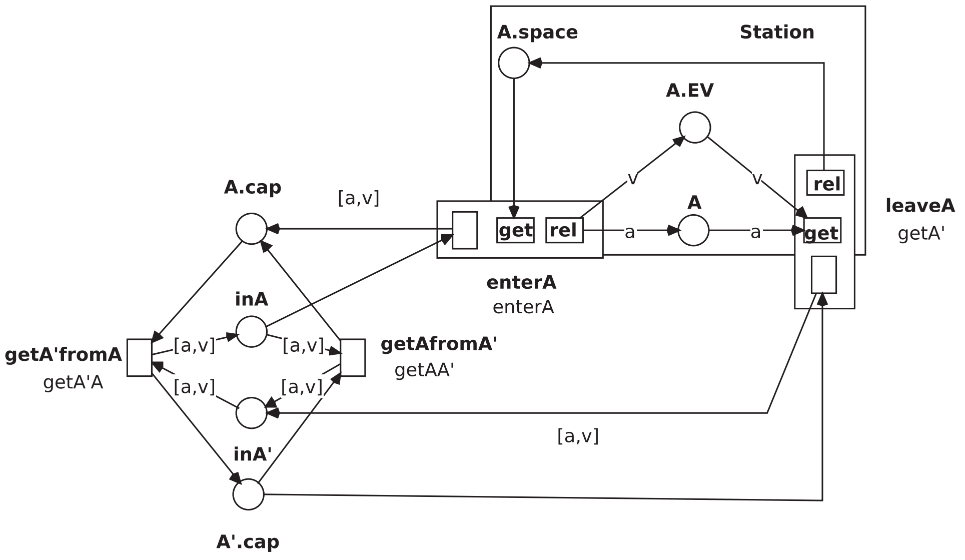


Figure 3.4: Module corresponding to the model of the EV station in street A.

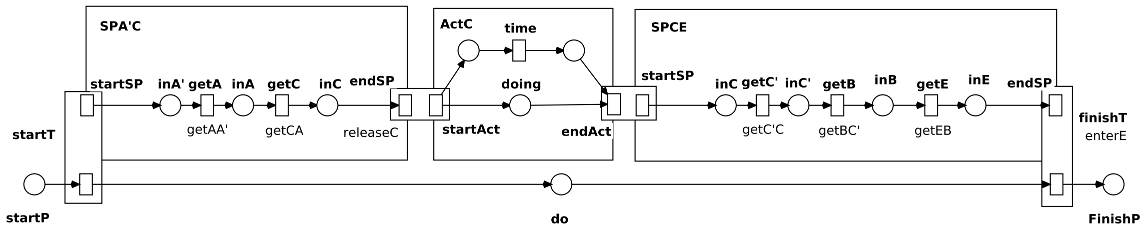


Figure 3.5: A user's agenda with two routes and one activity.

The composition of routes and agendas could be done automatically.

Finally, we model the control scheme as it is presented in Figure 3.6. Conceptually, it has three separate components: the controller, the set of controlled agendas and the resources.

The controller evaluates an assertion f. If the assertion is true, the controller will produce a message to the **outMsg** place. The message is produced by a function g which encapsulates the balance algorithm. The g function should return the pair $< src, dst >$, where src is where an EV should be taken and dst is the station where the EV should be left. The assertion might depend on external events or on the current state of the simulation, among other factors. The way this information is obtained is implementation dependant; for example, the state of the simulation could be in a shared memory or the communication could be made through a dedicated point-to-point channel. The controller is continuously iterating the previous steps. The set of controlled agendas represents the activity to travel from an EV station to another. They are used to move vehicles between the stations. Additionally, in our example, the resources in the control algorithm are the operators which move the vehicles.

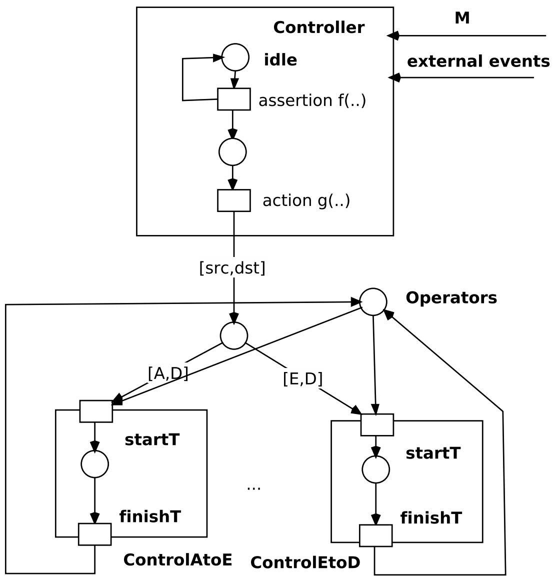


Figure 3.6: Model of the controller to balance the EVs in stations.

The communication between the controller and agendas is made by message passing. The controller puts in a place the pair $< src, dst >$ and, if there are enough resources, the place in the agenda which is waiting that pair will take the message and the agenda starts. The scheme presented is very simple and general. Other complex solutions can be implemented; for example, there could be some operators reserved to some plans to establish a priority policy.

3.4 Elaboration Process of Modular and Hierarchical Petri Net Models for Simulation in the Cloud

The result of the modelling process is an Object PN as in the example of the previous section. This PN can be obtained directly or by deploying the component-based model that has been described in a textual language as presented in [89]. The simulation of this model will consist in *playing the token game* on that Object PN. Proceeding in this way gives rise to several drawbacks that are related to the particular semantics of this kind of PNs. This is because the necessary simulation engine is a very particular algorithm that requires to be carefully analysed in order to be distributed in, for example, a cloud platform. This task is not trivial because the simulation engine must cope with modularity and hierarchy properties of the model that are not very well-adapted for distributed programming.

Instead of the direct emulation of the high level model, a transformation of the original model is proposed to obtain a model well-adapted for distributed execution, by using only primitive concepts that can be easily adapted to possible changes in the execution platform. We consider as primitive concepts those of sequential processes that communicate/syn-

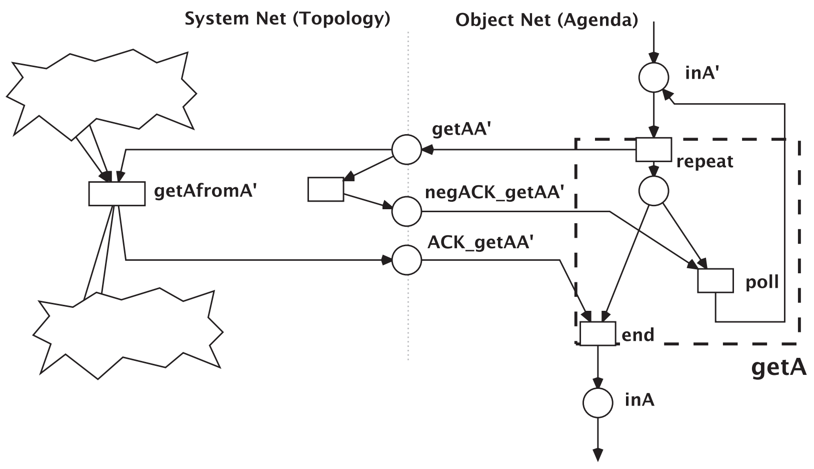


Figure 3.7: Elaboration of the synchronisation of transition **getA** of the agenda in Figure 3.5 and transition **getAfromA′** of the system net in Figure 3.2, through the label *getAA′*.

chronise by message passing. This transformation process will be called *elaboration* of the simulation model, and the transformed model will be called *elaborated model*, composed by sequential state machines and interchanging tokens by message passing mechanisms. Therefore, the elaboration process will be performed by:

1. Transformation of each synchronisation transition between a Token Net and its System Net into a subnet implementing a protocol for this synchronisation but based on message passing mechanisms.

2. Algorithms that identify and extract sequential state machines from the original Object Petri Net covering all transitions of this net, i.e. every transition of a Token Net or System Net must belong to one and only one sequential state machine.

3. Algorithms for the identification of places such that its set of connected transitions belongs to more than one sequential state machine previously identified, because the flow of tokens throughout these places will be implemented by the message passing mechanism –the messages will be the tokens– in the distributed platform of execution.

In the sequel, the elaboration process is briefly illustrated through the example presented in the previous section. The first step is the elaboration of the synchronisation transitions between the Token Nets and the System Net. Each synchronisation transition in a Token Net has a label used to identify the transitions to be synchronised –i.e. a transition of the System Net and a transition of the Object Net sharing the same label–. For example, let us consider the transition **getA** of the agenda –a Token Net– depicted in Figure 3.5. This transition is synchronised with transitions of the System Net sharing the label *getAA′*, for example, the transition **getAfromA′** of the System Net depicted in Figure 3.3. The elaboration of this synchronisation yields the transformations shown in Figure 3.7. In the elaborated model, three new places associated to the label appear for this synchronisation:

- Place **getAA′**: a token in this place represents a request from transition **getA** of the agenda to synchronise with a transition in the System Net with the label $getAA'$. Initially, it is empty, and it has as output transitions all transitions of the System Net sharing the same label and an special transition representing the denial of synchronisation.

- Places **negACK_getAA′** and **ACK_getAA′** represent the answer of some transition of the System Net for denial or approval of the synchronisation through the label. The next transformation –see Figure 3.7– is based on the substitution of transition **getA** in the Token Net by a *polling algorithm* that sends synchronisation requests with some transition of the System Net while receiving denials of synchronisation.

The timing of these new transitions is implementation dependent but the value must be much lower than the time of the original transitions.

After the elaboration of the synchronisation transitions, the next step is the identification of state machines in the Object Petri Net. The goal is to identify a set of state machines covering all transitions of the Token Nets and System Net, in such a way that a transition belongs to one and only one of the identified state machines. In the example depicted in Figure 3.5, the reader can observe that the Token Nets are user agendas and therefore, they are state machines. Therefore, the first set of state machines is composed of the nets corresponding to the Token Nets, and so all transitions in the Token Nets are covered by this set. State machines of this first set are not connected to each other by message passing. However, there may be communication between these state machines and the System Net because of the transitions synchronised by labels as in the example of Figure 3.7.

The same process of state machine identification must be applied to the System Net. In this case, by observing the net in Figure 3.3, it is easy to see that this net is not a state machine and then general methods for state machine decomposition must be applied. For example, in [91] an algorithm based on OBDDs is used to decompose a general place transition net into a set of state machines. It is also possible to use algorithms based on linear algebra [92] that compute minimal marking linear invariants that in many cases are associated to state machines. Many other methods to perform this identification operation exist, and all of them exploit the structure of the net by using different strategies.

The last step to obtain the elaborated model is the identification of the messages that must be interchanged by the different state machines distributed in the execution platform. Messages are the tokens generated by the occurrence of a transition that must be added to a place that does not belong to the state machine of the fired transition. This place will be called communicating place. If the input and output transitions of a communicating place belong to the same state machine, the messages will be internal; but if this place is connected to at least two state machines, then two things happen: i) all state machines containing at least one output transition of the communicating place will be grouped in the same simulation engine in order to avoid the distribution of a conflict between the output transitions of the communicating place; and ii) the firing of an input transition of a communicating place causes the sending of the produced tokens from the state machine containing this transition to all the state machines where this communicating place has an output transition. The communication process is performed by the message passing mechanism of the platform.

The tokens to be sent as a message can be of two types:

- **Black tokens**, that only require to include in the message the name of the destination place of the token and the identification of the state machine where the place must be present.

- **Coloured tokens**, that require to include in the message, in addition to the information included for black tokens, the identifier of the Token Nets to which refers the transmitted token. This is because the coloured tokens only appear in the System Net and they are references to Token Nets that have been deployed somewhere in the distributed platform.

The heterarchy of processes resulting from the elaboration of our specification is the starting point to provide a raw description for a cloudificable model. Partition schemes over the flattened model pursue a trade-off between the performance and the resource cost for the distributed execution of simulations in cloud environments. The TPN formalism imposes some partitioning constrains. Arbitrary arc cutting can decrease performance due to the overhead of involved messages in the required protocols between LPs with distributed conflict resolutions, i.e., transitions sharing input places. This overhead of messages can ruin the potential advantages of a distributed simulation. From now on, we will assume that a Virtual Machine is assigned to each LP. A LP consists of: a TPN region; an interpreter engine that simulates the token game of the region and preserves causality with events simulated in other LPs; and an interface. Some solutions of Petri Net partition algorithms [26, 27] have improved the arbitrary strategy of arc cutting, with methods and rules for minimum region extraction, aggregation of suitable regions and automated mapping of regions to nodes.

The LP interface is defined by the arcs from LP_k to LP_i. Therefore, the input interface is defined by means of places and the output interface by means of transitions of adjacent LPs. The first idea to develop the partitioning is to minimise the number of tokens –messages– to be exchanged between subnets. In this way, all tokens transferred from LP_k to LP_i are sent over the same channel, and communication complexity is reduced considerably. Additionally, we will use the idea of *coupled conflict*, which can be inferred from structure of the TPN in order to define the minimal region assigned to a LP. Two transitions t_i, t_j are in *coupled conflict*, $t_i CC t_j \Leftrightarrow {}^\bullet t \cap t^\bullet \neq \emptyset$ or $\exists t_R$ such that $t_i CC t_R \wedge t_R CC t_j$. The CC relationship is an equivalence relation that defines a partitioning in a set of distinct equivalence classes called Coupled Conflict Class (CCC). The set of distinct CCC defines a partition with the minimal possible regions.

Depending on the communications overhead, this partition on minimal regions may be too fine grained. The size and elements included in regions shape execution and communication results. Too small regions entail too many nodes with high communication and short execution of LPs, that can hurt overall performance and economic cost. Big regions need few nodes that can help to curve the economic cost, but also reduce performance. The computation/communication threshold gives the upper bound for the maximum number of computation nodes to maximise the speed up. The second idea to afford this trade-off is to cover with sequential state machines the partition defined by transitions in conflict. Therefore, the problem is a graph partition problem, where the set of logical processes –$LP = \cup_i LP_i$– together with the directed communication channels –$CH = \cup_{i,j}(LP_i, LP_j)$– constitutes the graph of logical processes –$GLP = (LP, CH)$–. The coverture that defines the partition is produced by taking into account the objective function to minimise, which is the cost given a simulation time t_s.

Formally, if $\mathscr{P}$ is a partition of GLP with N logical processes –LP_i–, and $R(LP_i)$ is the set of sequential processes included in LP_i, we can estimate the economic cost associated to partition $\mathscr{P}$ by using Equation 3.1, where $\hat{T}_{wc}$ represents the estimated wallclock time.

$$\hat{C}(\mathscr{P}) = N \cdot \hat{T}_{wc} \tag{3.1}$$

$\hat{T}_{wc}$ is defined by the maximum bottleneck estimated wallclock time of all LP_i –denoted by $\hat{T}_{wci}$–. To estimate this value, we need to associate an execution weight e_i to each LP_i by the addition of all e_j corresponding to the sequential processes included in LP_i, which represents the amount of computation time per simulation time unit, and a communication weight w_{ij} between every pair of LPs. In [26], it is proposed to define e_i as the number of places and transitions in LP_i, and to equal w_{ij} to the number of arcs crossing between LPs i and j. In our case, we propose to estimate $\hat{T}_{wci}$ in a similar way as a function of e_i, w_{ik}, specific cloud parameters γ, and the simulated time t_s (Equation 3.2).

$$max_{1 \leq k \leq N} \left\{ \hat{T}_{wci}\Big(\sum_{j \in R(LP_i)} (e_j), \sum_{j \in R(LP_i), k \notin R(LP_i)} w_{jk}, \gamma, t_s \Big) \right\} \tag{3.2}$$

Initially, $\hat{T}_{wci}$ values can be approximated by simulations to get computing and communication values. These values can be grossly approximated through: i) a partition requiring the minimum number of VMs to store the biggest admissible regions in memory of each VM; or ii) the maximum number of VMs allowed with an initial budget for estimating these values. The objective function is a minimisation of the cost considering initial constraints, budget and quality of simulation.

Graph partitioning is a fundamental problem in many domains in computer science. Important applications of graph partitioning include scientific computing, partitioning various stages of a VLSI design circuit and task scheduling in multi-processor systems. In Section 2.3, we have presented a review of the graph partition problem in streaming environments. If we remove the streaming restrictions, there is a vast literature about graph partitioning –see for a recent review [93]–. The most promising partitions divide the graph in equal sets while minimising edges between the sets. In the case of TPN partitions, structural information and time labels can also be exploited to define balanced lookahead values in all Logical Processes. More information about the cloud infrastructure, such as the coefficient of variation of execution time and communication, can also be included in the objective function to improve the estimated cost of a partition, namely, $\hat{C}(\mathscr{P})$.

3.5 Conclusions

Distributed simulations allow understanding and analysing complex and scalable systems. We have proposed a PN-based modelling methodology for complex DESs and a way to automatically translate the high level specification to an executable model suited to be partitioned on the cloud. The high level model provides a specification language suited for the designer of the system that enables a formal description of different facets of the system, and a hierarchical and component decomposition. On the other hand, we use the TPN formalism, as a low level model, to simulate the system. This low level model can be automatically generated, distributed and executed in an efficient and cost effective way. An elaboration process translates the high level specification in a flat TPN model. Finally, a partitioning step is proposed to deal with the decomposition of the flat low level model in regions to be distributed in cloud nodes. The obtained regions are evaluated in relation to the economic cost of their execution to decide the effective distributed deployment on the cloud.

The graph partition problem –which is similar to the partitioning step proposed in this chapter– has been addressed in the following chapter. Although the streaming paradigm induces heavy restrictions on the partition algorithm, several of its conclusions can be extrapolated to this work, especially if we consider simulations which need a lot of data or models that can be built as data arrives –e.g. Process Mining using Petri Nets [94].

Chapter 4

Resource Efficiency to Partition Big Streamed Graphs

Science may be described as the art of systematic over-simplification – the art of discerning what we may with advantage omit.

Karl Popper

Describing systems –or complex applications– has focused on the relationship between entities and on their internal behaviour. One problem of this approach is that data is modelled implicitly and it is difficult to use the model to exploit properties related to the data structure. Certain applications are heavily conditioned by data, in special those related to the Big Data domain. What differentiates data-driven applications –see Section 1.1– is that several characteristics of data –e.g. how data arrives, its size, where it is located– condition where and even how the application is executed. In this chapter, we focus on the management of data which arrives following the Data Stream paradigm. The incoming stream models an entire big graph, for example, a social graph. How data is processed when it arrives or how it is distributed has an impact on further analysis.

In contrast to the previous chapter, the partitioned model is not a high level abstraction model; thus, it can be partitioned without the elaboration process. Moreover, as a Petri Net can be seen as a graph with special semantics, the resulting PN from the elaboration process can be partitioned with the use of these techniques. However, as we present in the following sections, the fact that data arrives in a stream manner has a deep impact on the resource usage of the distributed environment and on the partition algorithms that can be used.

The chapter is structured in eight parts. We first introduce the problem of graph analytics –Section 4.1–. Section 4.2 includes a synopsis of the fundamental concepts addressed in the chapter. In Section 4.3, we analyse current streaming architectures to process streaming graphs; and, in Section 4.4, we present our model and the designed algorithm and we analyse their resource needs. We show experimental results in Section 4.5 and we discuss the model and its results in Section 4.6. Finally, the conclusions and the future directions of the research are presented in Section 4.7.

4.1 Graph Analytics in Streaming Environments

Real time streaming of big graphs is a relevant and challenging application in Cloud Computing. In a distributed infrastructure, these applications deal with the sequential arrival of large size graphs and the execution of processing tasks over them. Graph analytics in social networks, web searching or bioinformatics are relevant domains with these requirements.

For example, in the web searching context, as information is discovered sequentially –e.g. a crawler which follows links between pages–, the underlying graph can be modelled as a stream. The graph is received sequentially and it is created in a distributed infrastructure, in order to execute processing tasks –or analytics– over it.

In a distributed environment, computations over a stream have to make an efficient usage of memory and network resources, due to the large scale of the information. In a cloud scenario like Amazon EC2, these variables are easily translated into economical terms following the classical *pay per use* model.

The large size of the graph obliges us to partition it along several machines. For example, the Yahoo! Dataset [95] takes up 120 GB and a web network graph, like the one used by Google in Pregel experimentation [96], needs 25 TB of free storage space. The information stored represents page links collected by a crawler.

The graph partition problem is a well studied problem in graph theory [35,36]. However, with the resource limitations introduced by a streaming environment, classical solutions are not suitable. Proposed architectures address different heuristics which try to minimise the number of edges between partitions. These heuristics are not feasible in a distributed and realistic environment because they have only one streaming input and they manage global information.

In our work, we analyse the architectural problem and we propose a solution. We reduce the amount of needed memory and network traffic generated by using a novel concept called summary graph. This model reduces the memory bound needed to store the streamed graph. In our architecture, each parallel partitioner accesses its local memory, which is updated periodically in a feedback scheme. In this way, incoming elements are processed as fast as they are generated, processing time is decreased and, therefore, the number of processing machines is significantly lower. In the experimental section, we compare our architecture to those proposed in literature. We conclude that with our solution, the quality of the obtained partition solution only depends on the chosen resources.

4.2 Preliminaries

In this section we present those preliminary concepts related to streamed graphs and graph partitioning problem. The used notation for graphs in this chapter is presented in Table A.1, in Appendix A.

4.2.1 The Data Stream Model

We consider that the graph arrives in a Data Stream way. A Data Stream A [11] is an ordered sequence of $a_1, a_2, ..a_n$ elements. Its main characteristics are:

- The system has no control over how data arrives. Although in theory, the system cannot make any assumptions about the arrival order, in practice, in several application domains, how data is retrieved –e.g. using a web crawler– can affect the arrival order. Our model does not take into account the arrival model of data; however, in the experimental section, we show how several performance metrics are influenced by it. One additional fact about this characteristic is that the arrival rate of data may be variable and unknown by the system.

- Data Streams are potentially unbound in size. A Data Stream is potentially unbound because it models a continuous flow of information. This means that the system cannot make any assumption about when the data arrival will finish. The analytics or applications running over the Data Stream should not wait to have all the data to produce results. Moreover, the complexity of these algorithms should not belong to $O(n)$, due to the fact that n might be unknown and the algorithm might be blocked while waiting for the entire data. Most algorithms address this issue by executing the algorithm over a data window.

- Once an element has been processed it cannot be retrieved. The fact that the incoming data is unbound makes storing each arrival element impossible. Algorithms can make use of the incoming element and of a summary of the previous data. Additionally, a system cannot store each arriving element; thus, developed algorithms cannot have $O(n)$ memory complexity.

The Data Stream paradigm is suitable to deal with Big Data problems when information is consumed or processed at the time it is generated. In this regard, there are a lot of data stream applications; for example, social network analysis or web links analysis.

4.2.2 Graphs in Data Stream Model

We model a graph as a Data Stream in the following way. We denote $G = (V, E)$, an undirected and unweighted graph with a vertex set $V = \{v_1, v_2, v_3, ...v_n\}$ and an edge set $E = \{e_1, e_2, e_3, ..., e_m\}$. Note that n is the number of vertices, m the number of edges and $e_i = (v_j, v_k), \quad v_j, v_k \in V$. We represent each vertex with its adjacency list. For instance, web pages can be the vertices and links are the edges.

A vertex graph stream, T, is a sequence of $t_1, t_2, ..., t_n$ where $t_i = (v_j, v_{j_1}, v_{j_2}, ..., v_{j_d})$ such as, for $i = 0..d$, $(v_j, v_{j_i}) \in E$, where $d = deg(v_j)$.

Consequently, the size of a tuple t_i depends on the number of edges on a vertex, d. Therefore, the processing time per tuple is variable because it depends on the number of adjacent vertices. In addition, as we consider undirected graphs, each edge appears implicitly twice. In storage terms, the graph size is bounded by $\Theta(n + 4m)$.

In the general Data Stream model, no consideration can be made about the order in which elements arrive; however, some specific arriving models have been proposed for graph problems [38]. In Breadth First Search arriving model (BFS), one vertex of the graph is selected. A breadth first search strategy is performed from that vertex to generate the subsequent ones. If a depth first search strategy is used, we define the DFS model. In graph applications, these orders make full sense; for example, in the network context, the graph could be generated by one crawler which follows links with one of these strategies. In our model, elements arrive in a random manner because that is the worst case. However, in the experimental section –Section 4.5–, we have tested the system with BFS graphs, to compare the suitability of different summary functions depending on the order of arrival. In addition, our work is focused on vertex graph streams, because of the fact that the information we have per tuple is significantly bigger than in edge graph stream.

4.2.3 The Graph Partition Problem

In [39], the graph partition problem is modelled in the following way. Given a graph $G = (V, E)$, we define a k partition set P, where $P = \{S_1, ..., S_k\}$ such as $S_i \subset V$, $\bigcup_{i=1}^{k} S_i = V$. We define the graph partition problem as finding an optimal P^* such that for all possible partitions P such that $|P| = k$, $f(P^*) \geq f(P)$, for a certain function f. Our objective is to obtain a partition P which minimises the communication cost among partitions and the processing time of an application algorithm.

Definition 1. *Cutting edge. Given a graph $G = (V, E)$ and a partition set $P = \{S_1, .., S_k\}$, we say that an edge $(v_i, v_j) \in E$ is a cutting edge if $v_i \in S_q \wedge v_j \in S_r \wedge r \neq q$, with $i, j \in \{i...n\}$, and $q, r \in \{1...k\}$. The set of all cutting edges is denoted by Λ.*

In order to measure the quality of the obtained partitions, we will use the λ and ρ values –Equations 4.1 and 4.2.

$$\lambda = \frac{\textit{number of cutting edges}}{\textit{total edges}} = \frac{|\Lambda|}{m} \tag{4.1}$$

$$\rho = \frac{Max\{|S_i|, i = 1...k\}}{\frac{n}{k}} \tag{4.2}$$

The λ metric gives the possible overhead of the communication between partitions when graph processing tasks are executed. As the graph is unweighted, if the number of edges between partitions decreases, the amount of communications also decreases. The ρ value is the balanced factor of the solution partition P. In the processing phase, having too many unbalanced partitions might increase the processing time –i.e. some machines have to do a heavy process and others might be idle–. Function f, which determines the optimal partition solution, will depend on λ and ρ parameters. This problem is NP-Complete [37].

4.2.4 Real Time Processing

We want to process each element as it arrives, without decreasing the incoming rate of the stream or without blocking the entire system due to load issues. Therefore, only one pass has to be made over the stream. In addition, process made per element has to be fast, so it cannot belong to $O(n)$.

The fact that we develop a single-pass algorithm with such hard memory and network usage restrictions, entails that our partition solution is approximate. We compute an ϵ-approximation of λ, which means that our result, λ', satisfies Equation 4.3.

$$\lambda' \leq \epsilon\lambda \tag{4.3}$$

4.3 Architectural Approaches

Figure 4.1 depicts different distributed architectures for partitioning streamed graphs. The simplest one is in Figure 4.1a. It has been used in [38,39]. It has only a single entry point, so the number of incoming elements per time unit, σ, is bounded by the network capacity and by processing time per tuple in the partitioners. If we analyse this solution, we can observe that the partition algorithm, for each incoming tuple, stores its vertex in local memory. This situation is not suitable for real time scenarios due to the fact that this architecture has a $O(n)$ memory bound, with n being the number of elements in the stream, in our case, the number of vertices in a very large graph.

This amount of memory could be obtained in two ways, which are presented in Figures 4.1b and 4.1c. However, these solutions have some problems. The latency introduced by a distributed memory is high and variable; the processing time per element is incremented, so the amount of tuples processed per time unit is bounded. In other words, σ decreases. We can try to solve this limitation with a local internal buffer in each partitioner, avoiding delaying the partitioner for each new element. This solution may be unfeasible due to the high variability of the degree of the vertices in the graph. Furthermore, we can consider the additional traffic generated by querying the distributed memory. In the best situation, the distributed memory would work as a single machine. Query size is $1 + \frac{m}{n}$ on average, and there are two possibilities for the communication with distributed memory:

- **Synchronous communication.** The distributed memory answers with the number of neighbours and free space in each partition. The size of the answer is $3k + 1$ and the network traffic per second is $\sigma(\frac{m}{n} + 3k + 2)$. However, processing time per tuple in each partitioner is significantly higher, so the number of partitioners has to increase. A possible partial solution is storing the pending tuple in an internal buffer.

- **Asynchronous communication.** The distributed memory answers by sending the entire tuple over the network. In this situation, the vertex with all of its neighbours is sent twice, and local partitioner does not need to store the tuple in a buffer nor wait for the answer from the distributed memory. The answer has to contain the same information as before, so the answer size is $3k + 1 + \frac{m}{n}$. The network traffic generated per second is $2\sigma(\frac{m}{n} + \frac{3k+1}{2})$ elements.

In both cases we do not take into account the additional messages generated by the distributed memory to locate the needed information.

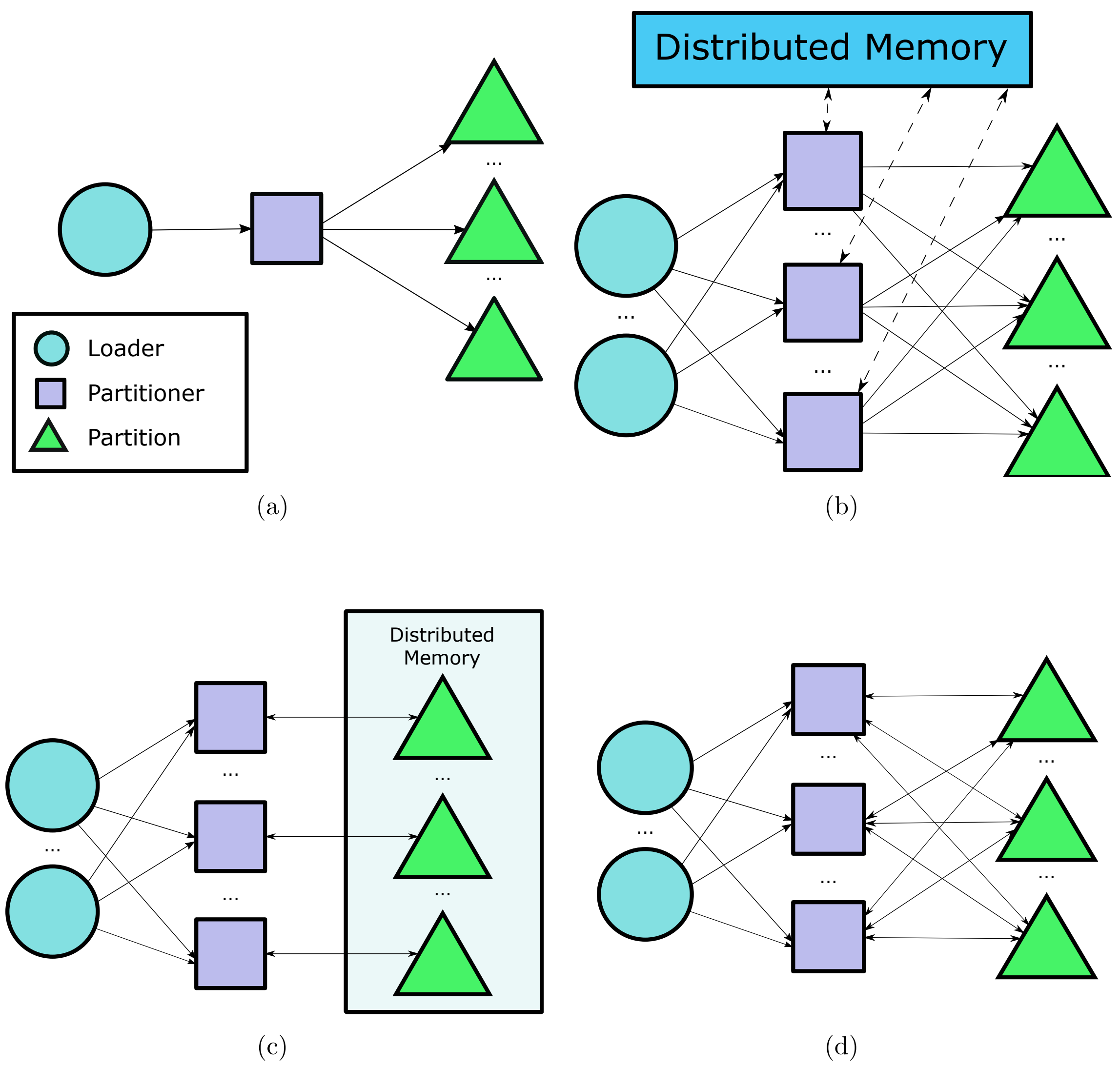


Figure 4.1: Distributed architectures to partition streamed graphs: centralised architecture (Subfigure a), distributed architecture with Distributed Memory (Subfigure b), distributed architecture with *ad-hoc* Distributed Memory on partitions (Subfigure c) and our proposed architecture (Subfigure d).

The second possibility is shown in Figure 4.1c. In this situation, we take advantage of the partitions, so they can work as a distributed memory. Each partitioner has to query every partition for each incoming element in order to determinate which one is the best. This leads to the same situation as before. The number of messages per second is $2k\sigma$ and the same analysis can be done to estimate the message size.

As we have seen in this overview, current architectures have some limitations when they have to scale to process large scale graphs. Their problem is the memory size, which is bounded by the graph size. Solving this problem through traditional approaches increases the processing time per element; thus, σ is decreased by a factor γ. This situation implies that $\gamma \cdot s$ partitioner machines are needed instead of s, which may have a big impact in economic terms.

Our proposed solution reduces the memory bound and the amount of additional traffic generated, in order not to decrease the number of incoming items per time unit, σ.

4.4 Summary Graph

We propose the decentralised architecture represented in Figure 4.1d. We have uncoupled the different processing stages in order to distribute them. There are several loaders which continuously send elements to partitioners. The partition algorithm is executed in these partitioners, and it has to be simple in computational terms. In addition, it has to choose the partition based on partial information. To maintain a consistent distributed state, the partitions update the local information of partitioners periodically.

We reduce the required memory by using the summary graph abstraction.

Definition 2. *Summary Graph. Let be a graph $G = (V, E)$ and a summary size l; we define a summary graph $G' = (\Psi, \Phi)$, where $\Psi = \{\Pi_1 ... \Pi_u\}$, $\Pi_i \subset V$, $\Phi = \{(\Pi_r, \Pi_q) \in \Psi,\ r, q = 1...u\}$ with $u = \frac{n}{l}$.*

Definition 3. *Summary functions. Given a summary graph $G' = (\Psi, \Phi)$, where $u = |\Psi|$, we can define a pair of surjective functions, g and h, called summary functions, such as:*

$g : V \rightarrow \{1...u\}$

$h : E \rightarrow \Phi$

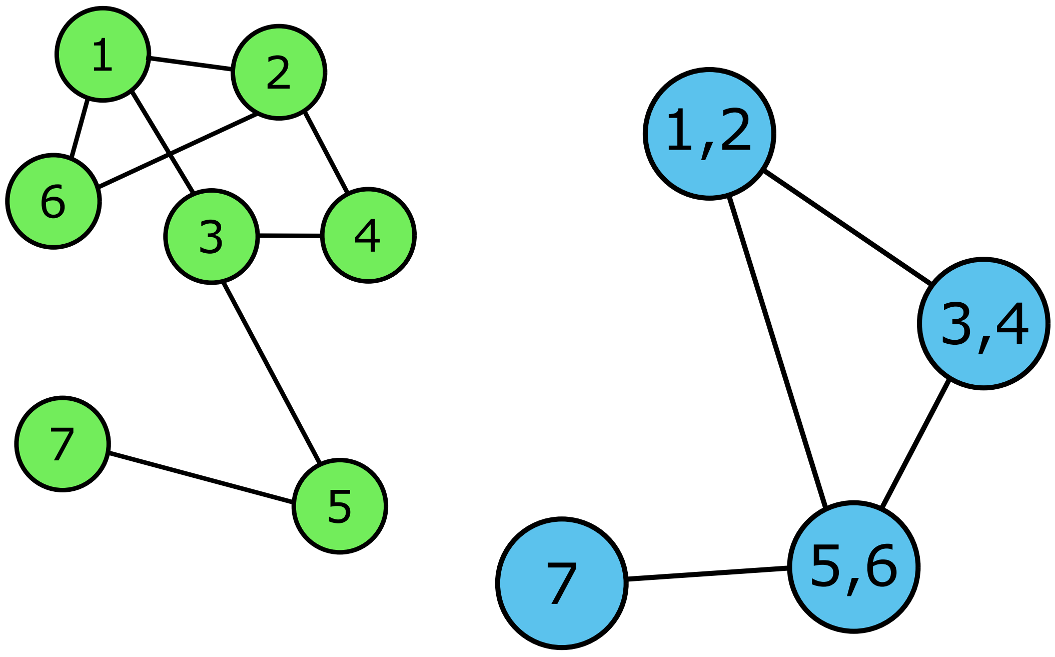
These functions meet the following conditions:

i $\forall v \in V,\ \exists \Pi_q \in \Psi \mid g(v) = q \Leftrightarrow v \in \Pi_q, q = 1...u.$

ii $\forall v_i, v_j \in V,\ \exists \Pi_r, \Pi_q \mid g(v_i) = \Pi_q,\ g(v_j) = \Pi_r,\ i, j = 1..n,\ q, r = 1..u,$ *and* $(v_i, v_j) \in E \Leftrightarrow h((v_i, v_j)) = (\Pi_q, \Pi_r)$

As a result, given $i, j = 1...n,\ q, r = 1..u\ \ \forall(\Pi_r, \Pi_s) \in \Phi,\ \exists v_i, v_j \in V$ such us $g(v_i) = \Pi_r,\ g(v_j) = \Pi_q, h((v_i, v_j)) = (\Pi_r, \Pi_q)$

In the summary graph model, l vertices are grouped in a single set –Π_i–, which is called summary. The larger the summary size l is, the less summaries there would be and, consequently, the less memory would be used. The transformation between the graph G and the summary graph G' is made by the summary functions g and h. The function g computes the summary which a vertex belongs to. As we propose a distributed framework with streaming considerations, summaries have to be created without having knowledge of the graph structure. Moreover, the functions g and h should not use any distributed state since it may not be consistent in all partitioners –which are the instances in charge of



(a) Original Graph (b) Summary Graph

Figure 4.2: Transformation between a graph (Subfigure a) and a summary graph (Subfigure c). $l = 2$ and the summary function is the consecutive assignation.

computing g–. Figure 4.2 depicts an example of how a graph (Subfigure 7.3a) is transformed into a summary graph (Subfigure 7.3c) with a summary size of two.

The memory size required to store the summary graph does not only depend on the number of vertices n of the original graph. The memory is bounded by $O(\frac{n}{l})$, where l is the number of vertices summarised in the same set. The memory reduction is done at the expense of the accuracy of the algorithm executed over the summary graph. In streaming scenarios, where the incoming graph might be too large, it might be the only available solution.

4.4.1 Summary Functions

The g and h functions are light functions –their computational time belongs to $O(1)$–. The information used to assign one vertex v to a set Π_i has to be known *a priori* for each partitioner. Hence, as we want fast partitioning of graphs with low memory usage, the partition decision has to be made without taking into account prior items in the stream. The summary function cannot depend on the arriving order. Additionally, we consider the number of elements in a summary constant, that means that $l = |\Pi_1|= |\Pi_2|= ... = |\Pi_{|\Psi|}|$

With these assumptions, the simplest summary function is based on the vertex identifier. We propose the following summary functions:

- **Hash function** (Equation 4.4). Each vertex $v_i \in V$ goes to a summary set depending on its identifier. If the vertex identifier is a non-numerical value, it should be transformed beforehand.

$$\begin{aligned} \forall i,j \in \{1,...,n\},\ g(v_i) &= i\ mod\ l \\ h((v_i, v_j)) &= (\Pi_{i\ mod\ l}, \Pi_{j\ mod\ l}) \\ |\Psi| &= \frac{n}{l} \end{aligned} \tag{4.4}$$

- **Consecutive assignation** (Equation 4.5). If the node identifier is numerical, vertices can be summarised sequentially. In some situations –in a BFS or DFS model– this summary function makes more sense because, as items arrive following a specific criteria, connected elements go to the same partition; so the summary function tries to minimise the cutting edges generated by a random summary.

$$\begin{aligned} \forall i,j \in \{1,...,n\}\ g(v_i) &= \left\lfloor \frac{i}{l} \right\rfloor \\ h((v_i, v_j)) &= \left(\Pi_{\lfloor \frac{i}{l} \rfloor}, \Pi_{\lfloor \frac{j}{l} \rfloor}\right) \\ |\Psi| &= \frac{n}{l} \end{aligned} \tag{4.5}$$

- **Property summary.** If the vertex satisfies some kind of property, summarising vertices with the same property value in the same set can be suitable. For instance, if the vertices are labelled, vertices with the same label will belong to the same set.

4.4.2 Partition Problem with the Summary Graph Model

The summary graph model is a compact representation of a graph, which requires less memory to be stored. If we have a certain graph algorithm which has a $O(f(n))$ bound for a certain resource –e.g. memory or execution time–, the same algorithm executed over a summary graph will have a $O(f(n)/l)$ bound for that resource.

In the case of the streaming partition problem, we propose the decentralised architecture represented in Figure 4.1d. The partition algorithm uses the summaries to calculate the best partition, instead of using vertices. As g gives the same value regardless of the partitioner which computes it, all vertices in a summary are assigned to the same partition. That is to say that we are partitioning summaries –with l vertices– instead of individual vertices.

Algorithm 1 illustrates partition algorithm with the summary notion. We represent the main memory with the M set, where $M = \{(q, j),\ q \in \{1...u\},\ j \in \{1...k\}\}$. Note that we assign a summary of vertices Π_q to the partition S_j by using its index q.

When an element t of the unbound stream T arrives, we obtain its vertices. If we assigned the summary which that vertex belongs to in previous steps, we would find that

Algorithm 1 Vertex Partition Algorithm

Input: Unbound stream T

```
M = ∅
for all $t \in T$ do
    let $v$ = get vertex $v \in V$ from $t$
    let $q = g(v)$
    if $\exists i \in \{1..|\Psi|\} \mid (q,i) \in M$ where $S_i \in P$ then
        Send $t$ to partition node $i$
        In partition node $i$, $S_i = S_i \cup \{t\}$
    else
        $i = BestPartition(t, P)$
        $M = M \cup \{(q,i)\}$
        Send $t$ to partition node $i$
        In partition node $i$, $S_i = S_i \cup \{t\}$
    end if
end for
```

summary in M. So, we have to send the vertex and its edges, t, to the already assigned S_i –$S_i = S_i \cup \{t\}$–. If it is the first time a vertex of that summary arrives, we will compute the optimal partition for that vertex with the *BestPartition* function, and we will add the corresponding summary to M. Moreover, our computational model decouples the summary function from the partition heuristic, which is a desirable property for a scalable system.

The analysis of *BestPartition* heuristic is out of the scope of the present chapter. In Chapter 2 (Section 2.3), we have presented some of the most recent works in this field. In the experimentation phase, we have selected Fennel [39] as partition heuristic with its best parameters. As we compare ourselves with Fennel, the comparison with Metis is straightforward.

The use of the architecture presented on Figure 4.1d generated additional traffic to keep consistent the distributed state in the s partitioners. This traffic is generated with a period f, and it is needed to update the local information of each partitioner. Thus, per second, $s \cdot k \cdot f$ messages are generated. The total size of the messages is $s \cdot k \cdot f(\frac{n}{lk} + 1)$. The main advantage is that incoming rate σ is not bounded by the network or by any parameter of the model, so the network traffic only affects on the accuracy of the solution, just like the used memory size.

4.4.3 Resource Analysis

4.4.3.1 Memory

A general partition algorithm executed over a graph has a $O(n)$ memory bound, which might not be suitable for a streaming scenario. The use of the summary graph instead of the entire graph relaxes the memory bound to $O(\frac{n}{l})$, where l is the size of the summary.

However, this memory relaxation is achieved at the expense of the quality of the partition λ.

Theorem 1. *Given a graph, $G = (V, E)$, a sample size, l, and a single-pass algorithm, ALG, which produces λ cutting edge fraction with a $O(f(n))$ memory bound; there exists an ϵ-approximation of cutting edge fraction λ', with a $O\left(\frac{f(n)}{l}\right)$ memory bound in each parallel partitioner and $\epsilon = \frac{l-1+\lambda}{l\lambda}$.*

Proof. Memory reduction is achieved by sampling incoming vertices into sets, and the sets are used as the input of the algorithm ALG. With a summary size of l, the memory bound belongs to $O(\frac{f(n)}{l})$.

In order to calculate the accuracy of the solution, we can consider that given an edge $e \in E$, the probability of being a cutting edge (Definition 1) is λ under ALG. If we transform the graph G into a summary graph G', the probability of being a cutting edge is given by Equation 4.6, where the law of total probability is applied, Par_{RND} denotes that the edge has been partitioned by a random partition algorithm RND, and Par_{ALG} means that it has been partitioned by ALG.

$$\begin{aligned}\lambda' = P(e \in \Lambda) = & P(e \in \Lambda | Par_{RND}) P(Par_{RND}) + \\ & + P(e \in \Lambda | Par_{ALG}) P(Par_{ALG}) \leq \\ & 1\frac{l-1}{l} + \lambda\frac{1}{l} \leq \frac{l-1+\lambda}{l}\end{aligned} \tag{4.6}$$

Using Equation 4.6, we can say that λ' is an ϵ-approximation of λ if Equation 4.7 is satisfied.

$$\lambda' \leq \epsilon\lambda \implies \frac{l-1+\lambda}{l} \leq \epsilon\lambda \implies \epsilon \geq \frac{l-1+\lambda}{l\lambda} \tag{4.7}$$

□

In Equation 4.6, we have set the probability of e_{ij} being a cutting edge when the RND algorithm is used as 1, because that is the worst case and we want to calculate an upper bound for ϵ. Equation 4.8 gives the theoretical probability assuming independence between the variables, for a certain number of partitions k. We can see that as the number of partitions increases, the probability gets closer to one.

$$\begin{aligned}P(e_{ij} \in \Lambda | Par_{RND}) = & 1 - \sum_{r=1}^{k} P(v_i \in S_r) \cap P(v_j \in S_r) = \\ & 1 - k\frac{1}{k}\frac{1}{k} = \frac{k-1}{k}\end{aligned} \tag{4.8}$$

However, for certain graphs, the empirical probability of the RND partitioner is greater than the theoretical probability given by Equation 4.8. One reason of this behaviour is that the independence assumption is not met due to the graph structure. Nevertheless, the bound given by Equation 4.7 is still satisfied. An option to improve that bound is to use an empirical value for the RND partitioner in Equation 4.6.

4.4.3.2 Distributed Complexity

The fact that we are using a distributed state which might not be consistent in a certain execution time, can cause that several partition decisions are not optimal. The frequency of these non-optimal decision will depend on the update frequency f. If the update frequency is increased, the probability of having an inconsistent distributed state will be reduced. The relationship between the distributed complexity –the amount of messages sent by the partitions and the partitioners– is given by the following theorem.

Theorem 2. *Given a graph, $G = (V, E)$, s distributed partitioners, a sample size, l, an update frequency, f, and a single-pass algorithm, ALG, which produces λ cutting-edge fraction with a $O(f(n))$ memory bound; there exists an ϵ-approximation of cutting edge fraction λ', with a $O\left(\frac{f(n)}{l}\right)$ memory bound in each parallel partitioner and a $O\left(\frac{nsk}{\sigma f}\right)$ global distributed complexity, where σ is the incoming elements per time unit and $\epsilon = \frac{l - \exp\left(\frac{-l\sigma f(\sigma f - 1)}{2n}\right)(1-\lambda)}{l\lambda}$.*

Proof. The number of sent messages from partitions to partitioners depends on the update period f and on the number of partitioners s; thus, in a f period, the system sends $k \cdot s$ messages. If σ elements arrive in one time unit, the entire graph arrives in n/σ time units. Then, in n/σ periods, $\frac{kns}{\sigma f}$ messages are sent, which gives the distributed complexity. The memory bound has been discussed in the previous section.

Now let us calculate the accuracy of the solution. We consider that ALG will produce less cutting edges than a random partition algorithm, RND $-\lambda' \leq \frac{k-1}{k}-$. Therefore, the non-optimal decision is taken when a vertex v_i whose $g(v_i)$ set has been assigned in the same period f is assigned again by the random partition algorithm. The probability of getting $\sigma \cdot f$ unique elements from n/l groups is given by Equation 4.9. The expression is given by the birthday paradox and the upper bound is given by a Taylor expansion of the terms.

$$\frac{\frac{n}{l} - 1}{\frac{n}{l}} \times \frac{\frac{n}{l} - 2}{\frac{n}{l}} \times \ldots \times \frac{\frac{n}{l} - (\sigma f - 1)}{\frac{n}{l}} \leq \exp\left(\frac{-l\sigma f(\sigma f - 1)}{2n}\right) \tag{4.9}$$

Given an edge $e \in E$, the probability of being a cutting edge (Definition 1) by using ALG is λ. If we transform the graph G into a summary graph G', the probability of being a cutting edge is given by Equation 4.10. For simplicity, we consider the probability of not being a cutting edge as the probability of having unique elements in that period and the probability of not being a cutting edge by using ALG algorithm $-1 - \lambda$.

$$\lambda' = P(e \in \Lambda) = 1 - P(e \notin \Lambda) \leq 1 - \frac{\exp\left(\frac{-l\sigma f(\sigma f-1)}{2n}\right)(1-\lambda)}{l} \tag{4.10}$$

$$\lambda' \leq \epsilon\lambda \implies 1 - \frac{\exp\left(\frac{-l\sigma f(\sigma f-1)}{2n}\right)(1-\lambda)}{l} \leq \epsilon\lambda \implies$$
$$\epsilon \geq \frac{l - \exp\left(\frac{-l\sigma f(\sigma f-1)}{2n}\right)(1-\lambda)}{l\lambda} \tag{4.11}$$

Dataset	**Vertices**	**Edges**
WS10000	10000	134944
WS100000	100000	3997464
BA10000	10000	134841
BA100000	100000	3548775
PL10000	10000	134766
PL100000	100000	4047486
amazon0312	400727	2349869
amazon0505	400727	2439437
amazon0601	400727	2443311
LiveJournal1	4843953	42845684
Wiki-talk	2388953	4656682

Table 4.1: List of used Datasets.

By using Equation 4.10, we can say that λ' is an ϵ-approximation of λ if Equation 4.11 is satisfied.

$\square$

As we have said before, the number of sent messages per time unit is ks/f. If we consider the distributed architectures of Figures 4.1b and 4.1c, the number of messages between the memory and the partitioners per time unit is 2σ. Thus, the incoming arrival rate is bounded by the network capacity, and for big values of σ the bound of our system is better.

4.5 Evaluation

4.5.1 Experimental Setup

We have implemented the model presented in the previous sections in a real environment in order to test it. There are several open-source distributed Data Stream Management Systems. The most important are Storm, S4 [97], and Data Stream over Spark [98].

We have chosen Storm [99], a Java based implementation, due to its flexibility to deploy the distributed infrastructure over available machines. It allows users to make a query, called *topology* in its terminology, as an operator combination in a Directed Acyclic Graph (DAG) scheme. The DAG nodes represent the operators, and the arcs model how data flows from an operator to another one.

Since we propose a complete graph processing framework, we need to test the influence of the quality of the partition solution in the execution time of an analytic algorithm over the graph. We execute a PageRank application implemented in a GraphLab cluster [100]. PageRank [101] is a typical graph application and it is very used in industrial environments, so it is interesting to analyse the behaviour of the graph with different partition schemes.

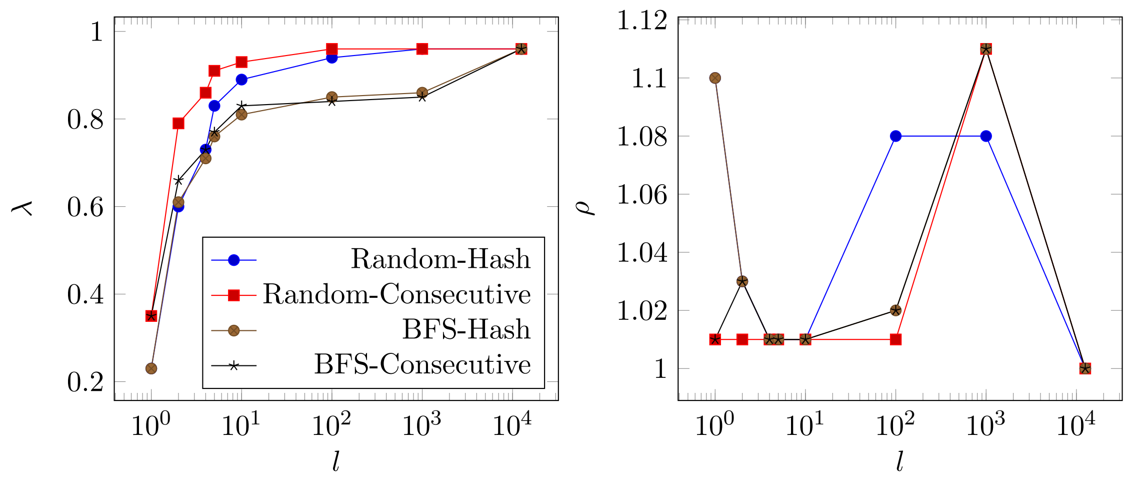


Figure 4.3: Fraction of cutting edges λ (Subfigure a) and Balance Factor ρ (Subfigure b) of *amazon0312* dataset versus summary size l with one single partitioner, 32 partitions and different incoming orders –BFS and Random– and summarise functions –Hash and consecutive.

The Storm cluster has eight workers and one master. We have one Virtual Machine for each worker. Each VM has one real core and 2 GB of memory size. The CPU is an Intel Xeon E5530 CPU with 2.40GHz.

4.5.2 Used Datasets

The datasets that we have used to test our system are in Table 4.1. PL, WS and BA datasets are synthetic, created by the Networkx[1] package. We have used these datasets because the web network and social graphs can be modelled by a power law graph [102]. PL dataset is a graph whose degree distribution follows a power law function. In an informal way, it is a graph where the number of high degree nodes is small. The degree distribution function is $y = cx^{-\alpha}$. WS is a Watts-Strogatzs graph model [103] and BA is a Barabasi-Albert graph [104].

Real datasets that we have used are: *amazon**, *Wiki-Talk* and *LiveJournal1*. The first one represents co-purchasing information of Amazon. Vertices are Amazon products and there is an edge from product i to j if i has been co-purchased with j. With this information Amazon can make personal suggestions to consumers. The information was collected in March 12 2003, May 05 2003 and June 01 2003. In *Wiki-Talk* dataset, each vertex represents an user. An edge from i to j represents that the user i has edited at least one time the talk page of user j. The information was recollected in January 2 2008. In *LiveJournal1*, each link between vertices –users– represents friendship relations.

[1] https://networkx.github.io/

l	λ	ρ
2	0.8	1.23
5	0.81	1.1
10	0.83	1.12
100	0.85	1.19
1000	0.91	1.18
12532	0.96	1

Table 4.2: Fraction of cutting edges λ and Balance Factor ρ in *amazon0312* dataset with $k = 32$, $s = 6$.

4.5.3 Experimentation

In Subsection 4.2.3, we have presented λ and ρ values. We have used them as measures of the quality of the obtained partition.

In our experiments, we have used Fennel [39] as the partition heuristic. This heuristic tries to minimise λ and ρ factors by assigning a vertex v to the partition S which maximises: $\Gamma(v, S) - \alpha\gamma x^{\gamma-1}$, where, $\Gamma(v, S)$ is the number of neighbours of the vertex v in partition S, $\alpha = m * \frac{k^{\gamma}-1}{n^{\gamma}}$, $\gamma = 1.5$ and x is the number of vertices assigned to S. We have taken these values as the best ones from their paper. However, since in our architecture we are partitioning summaries instead of vertices, we consider $\Gamma(g(v), S)$.

The aim of our experimentation is to measure the quality of obtained partitions, the total memory reduction and the additional traffic added by our architecture for different summary sizes and summary strategies. The last experiment shows the real impact of the quality of the partition solution when analytics are executed over the partitioned graph.

4.5.3.1 Quality of Partition

In Figure 4.3, we observe how summary size l affects λ and ρ values. We have partitioned *amazon0312* dataset into 32 partitions. The experiment has been made with BFS incoming order and with Random incoming order. We have implemented two summary functions, Hash and consecutive assignation. The first measure, $l = 1$, is equivalent to Fennel partition algorithm and the last one corresponds to the situation where there is only one summary per partition, $l = \frac{n}{k}$. This situation is equivalent to a random partition strategy. With two vertices per summary, the number of cutting edges increases compared to Fennel, but it is better than Random partitioner. In our results, the kind of summary function used affects the quality of the partition solution. In a BFS incoming order, the results are better for a consecutive assignment function. This kind of order is naturally obtained in social and web graphs because they are obtained by crawlers. The relationship among ρ and l, in Figure 4.3b, is distorted by the Hash partition strategy –last value– and the Fennel results –first value–. The random partition strategy always generates partitions with $\rho = 1$ and the Fennel strategy reduces λ increasing ρ. For the rest of values, as the size of the group

l	λ	ρ	Memory used (MB)
1	0.5	1.01	245.99
2	0.56	1.01	135.63
10	0.68	1.01	27.60
605495	0.96	1	0

Table 4.3: Fraction of cutting edges λ, Balance Factor ρ and used memory for different summary sizes l in *LiveJournal1* with $k = 8$.

increases, ρ also increases, because we are distributing a few big groups.

As we have discussed previously, the number of partitioners s, affects the quality of the partition, because they manage local information. In Table 4.2, we show the effect of changing the number of partitioners to six. We have used a contiguous grouping strategy and the BFS arrival order. Experimental results show that with a big summary size, the performance is similar to the single loader. The reason is that the Balance Factor decreases when the number of partitioned elements is small.

4.5.3.2 Used memory

We can see that the maximum used memory will depend on l, when l is high. In Table 4.3, the results for the *LiveJournal1* dataset are shown. In this case, we have partitioned into eight partitions with a BFS order and a consecutive assignment function.

We have also measured the RAM required to store the summaries in partitioners. The results are shown in Figure 4.4. As it is natural, when the number of elements per group increases, the needed memory decreases. Note that with $l = 1$, the total memory used is 20,8 MB. Approximately, this is 0.052 kB per element, so we cannot process a web network graph –50 billion vertices– with the Fennel algorithm.

4.5.3.3 Network traffic between Partitions and Partitioners

We have measured network traffic added by a distributed memory architecture and by our model. We have calculated the quotient between the total network traffic and the stream size. This metric is independent of the platform, in our case, Storm, and of the graph size.

We have simulated state of the art heuristics [38, 39] with a distributed memory. Our simulation does not take into account the latency and the internal messages generated by the memory. Its aim is to estimate the total network traffic. The simulated model is the asynchronous communication model presented in Section 4.3, where partitioners are continuously sending elements to the distributed memory and it answers with the entire tuple. As we have said, in a distributed memory approach, the total traffic in partition stage depends on the number of partitions k. As k is small enough, we have calculated the average value. In Figure 4.5, it is represented by a dashed line and it is about twice the streamed graph size. As we have analysed earlier, the reason is that we are sending each

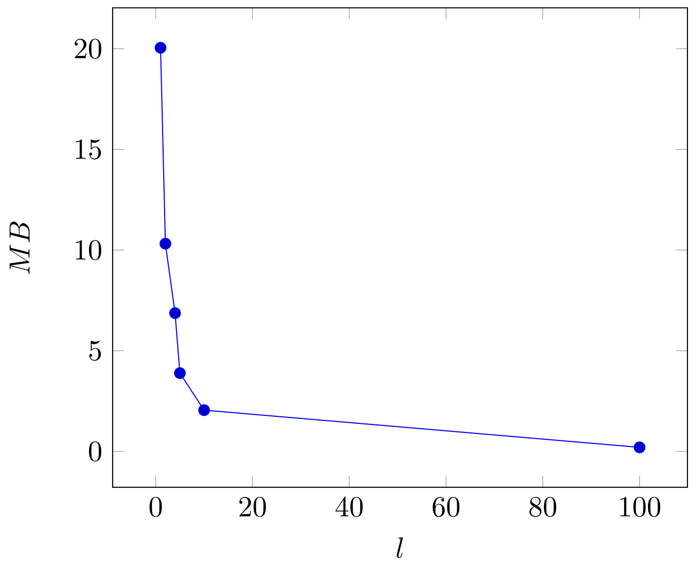


Figure 4.4: Required RAM memory to process *amazon0312* dataset versus group size l.

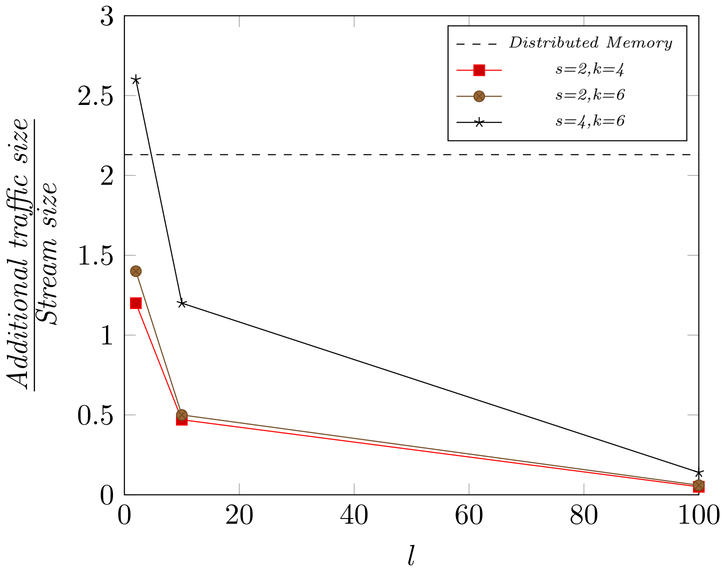
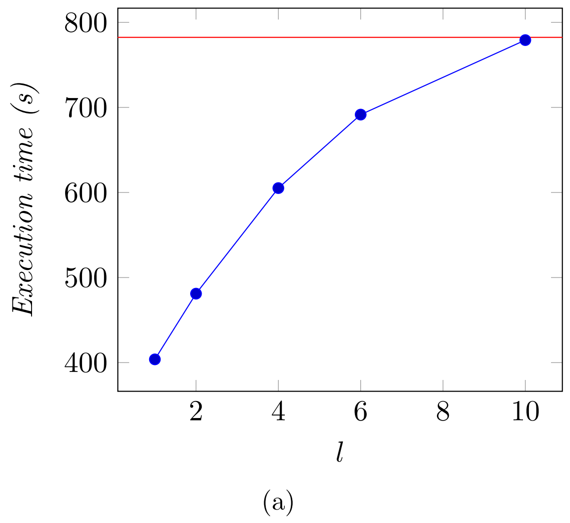


Figure 4.5: Additional network traffic in partition stage versus summary size l.

tuple twice with a bit more information.

In our architecture, the number of partitions and partitioners determines the total traffic between them. In Figure 4.5, we can see the relationship between these variables. If we had multicast communication, additional traffic would be divided by s. What is more, our architecture does not bound incoming tuple rate, as we have discussed in analysis section.



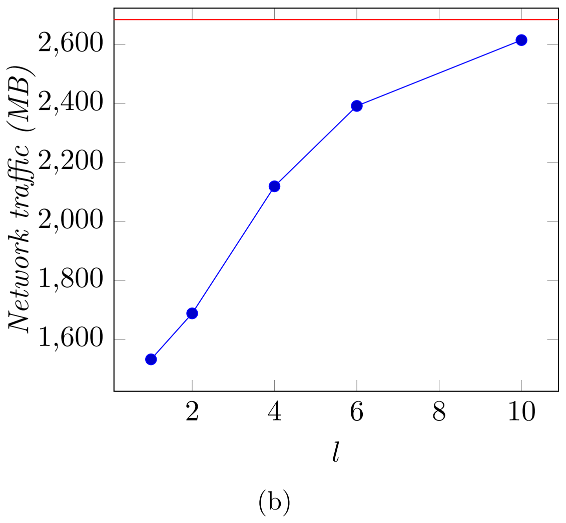


Figure 4.6: Execution time (Subfigure a) and network traffic between partitions (Subfigure b) for a PageRank execution over *LiveJournal1* dataset versus summary size l. Top line is the limit established by Hash partition strategy.

4.5.3.4 PageRank Processing

The last experiment we have made is to execute an off-line algorithm over our partitions to show the relationship between the quality of the partition and the execution time of graph analytics. We have chosen GraphLab system [100]. GraphLab is a distributed graph processing engine which provides several graph analytics algorithms. We have used the *LiveJournal1* dataset and the PageRank algorithm. We have compared our system to Hash strategy and Fennel, which has entire knowledge of the graph. As figure 4.6 shows, network traffic between partitions and total execution time increases as the partitioning quality worsens with bigger groups. In the experiment, we have used contiguous grouping strategy and six partitions. We see that for $l = 10$, the results are similar to the Hash partition strategy. Besides, the best situation is achieved with $l = 2$. Approximately, decreasing the used RAM to the half, as we can see in Table 4.3, only increases total execution time of PageRank algorithm by 25%.

4.6 Discussion

We have shown the impact of the memory and the network traffic on a realistic scenario in previous sections. Storm and its Java implementation use 0.052 kB for each vertex, so we can calculate the total memory needs for bigger datasets. For the Yahoo! Dataset, with 1.4 billion vertices, we would need 74.39 GB and for the Pregel experiment, with 50 billion vertices, we would need 2479.55 GB. Current cloud platforms offer nodes with a maximum 244 GB memory size –Amazon EC2– and 104 GB –Google Compute Engine–. Although in the future these values may increase, the Yahoo! dataset is dated in 2002 and Pregel experiment in 2010, so the size of graphs has increased too.

We have seen that the CPU usage is not very high, because the computation time for the light heuristics is in the order of a few microseconds per tuple. The input rate is bounded by the processing time –we can approximately process 200.000 tuples per second, a total of 43.5 MBs–. If we introduce a distributed memory, with high latency, we increase significantly the processing time per vertex. For example, with a 1 Gigabit Ethernet connection, we can estimate a latency of 0.350 ms, so the number of vertices processed per second will be approximately divided by seven. In economic terms, this situation entails multiplying the number of partitioner machines by seven.

Moreover, we can observe that the machines offered by cloud providers have high memory size and high CPU resources. This situation is not suitable for our processing. It would be more adequate to have less expensive nodes, with high memory and low CPU resources. In cloud infrastructures, network traffic between inner machines is cheaper than traffic with external ones. If the distributed memory is outside the cloud, the total network traffic might be high and it will be more expensive than accessing local memory.

With our work we approximate the partition problem to available cloud infrastructure. We provide an architecture suitable to partition large streamed graphs which makes a more efficient use of the available resources.

4.7 Conclusions

In our work, we have analysed distributed system architectures to partition streamed graphs. The use of a distributed memory to obtain the neighbours of a node bounds the system by the network capacity. Thus, the number of partitioners should increase. From this analysis, we have proposed a scalable and decentralised architecture which allows partitioning large scale streamed graphs efficiently. To reduce memory usage, we have made vertex summaries of the graph to compose a subgraph which is partitioned by a one-pass generic algorithm. The information consistency is maintained by updating local state in each partitioner. Furthermore, we have decoupled the partition algorithm and the summarising function. Because of the fact that we access local memory, instead of distributed, the processing time is lower, so the number of processing machines needed is divided by seven. This situation has a significant impact on economic cost.

In the experimental section, we have tested our system with synthetic and real datasets. Moreover, we have compared our model to the best –Fennel algorithm– and worst –Random paritioner– competitors. The first one has knowledge of the whole partitioned graph while the second one does not have it at all. With our summary functions, not having a global knowledge of the graph does not cause a significant loss in performance terms. In our experiments we show that by decreasing the RAM memory needed to process the *LiveJournal1* dataset to the half, only increases the processing time of PageRank algorithm by 25%. Finally, with our architecture, we propose a decentralised model which allows adjusting the quality of the partition solution to the desired amount of resources.

The simulation of complex systems and graph processing in streaming environments are two examples of applications which demand a high amount of computational resources due to their QoS requirements. As we have presented in the Introduction, these requirements have to be fulfilled in the operational level. In the next chapters, we study how several of these aspects can be addressed in a specific operational infrastructure –a Kubernetes private cloud.

Chapter 5

Characterising Resource management Performance in Kubernetes

The knowledge of anything, since all things have causes, is not acquired or complete unless it is known by its causes.

Avicenna

In the previous chapters, we have considered applications or data as a white box, and we have used properties of the model to improve the distribution of the application. In this chapter, we will consider a general purpose cloud processing environment where applications arrive to be executed as black boxes. We have called these kind of applications system-driven applications –see Chapter 1–, due to mapping decisions –where functional instances are deployed or executed– being taken by the system scheduler, and the information about the application being quite limited. The applications which arrive to the system might represent self-contained applications or pieces of applications obtained in previous processing steps as proposed in Chapter 3 and in Chapter 4.

As the decision is influenced by the specific operational system, the scheduling algorithms and decisions should not be aware of the operational platform. In the next chapters, we focus on container management systems; thus, the operational instances are containers executed in physical machines. This means that incoming applications are executed inside containers, and containers model how the resources are shared in a physical machine. The aim of this chapter is to analyse how a Kubernetes cluster behaves by using a formal model. The model is fed with experimental data to abstract several deployment rules which can improve the performance of incoming applications. In other words, instead of focusing on where applications –or application components– are deployed, we focus on how the deployment parameters can be optimised.

The chapter is structured as follows. We begin by introducing the container as a virtualisation technique in Section 5.1. Then, in Section 5.2, we present our model and its use with real data. Section 5.3 presents our characterisation of the pod abstraction overhead, discussing deployment results in Section 5.4.

5.1 Containers as Infrastructure as a Service

Cloud systems enable computational resources to be adjusted on-demand and in accordance with changing application requirements. Applications can rent computational resources of different types: virtual machines, containers, specialist hardware –e.g. GPU or FPGA–, or bare-metal resources, each having their own characteristics and cost. An effective automated control of cloud resource (de-)provisioning needs to consider [105]: i) resource utilisation; ii) economic cost of provisioning and management; and iii) resource management actions that can be automated.

Increasingly, many cloud providers support resource provisioning on a per second or even millisecond basis, such as GCE[1], or Amazon Lambda[2] –referred to as "serverless computing"–. A lambda function is provisioned through a container-based deployment, whose execution is billed at 100ms intervals. Therefore, understanding performance associated with the deployment, termination and maintenance of a container that hosts that function is significant, as it affects the ability of a provider to offer more fine grained charging options for users with stream analytics/processing application requirements.

Provisioning and de-provisioning actions are subject to a number of factors [105], mainly: i) the *overheads* associated with the action –e.g. launching a new VM can often take minutes [106]–; ii) the actual processing time required can vary due to resource contention –leading to uncertainty for the user.

There are several platforms which support a container-based deployment, for instance, Docker Swarm, Marathon over Apache Mesos or Kubernetes [44], among others. In this work, we have used Kubernetes because it offers a new abstraction layer in the deployment of containers –the pod concept, which is explained later– and the implications of this new abstraction have not been addressed by literature. Additionally, Docker Swarm is an extension of the Docker API, and it does not provide natively the support for certain non-functional requirements, such us application scale. Finally, Kubernetes –or systems designed over Kubernetes, such as Red Hat OpenShift– has become the industrial reference container management system; not only as a solution for private clouds, but also as an alternative deployed in public ones –for instance, Google Kubernetes Engine[3], Azure Kubernetes Service[4] and the Amazon Elastic Container Service for Kubernetes[5].

Kubernetes can provide a Cloud-Native Application (CNA) [107], a distributed and horizontally scalable system composed of services or microservices, with operational capabilities such as resilience and elasticity support. Kubernetes enables deploying multiple pods across

[1]https://cloud.google.com/
[2]https://aws.amazon.com/lambda/
[3]https://cloud.google.com/kubernetes-engine/
[4]https://azure.microsoft.com/es-es/services/kubernetes-service/
[5]https://aws.amazon.com/es/eks/

physical machines and scaling out of an application with a dynamically changing workload. Each pod can allocate multiple containers, which can make use of services –e.g. file system and I/O– associated with a pod. Any OCI compliant container runtime engine could be used, but we chose Docker as it is the most popular engine for Kubernetes.

We research the performance of deploying, terminating and maintaining containers with Kubernetes, identifying operational states that can be associated with a pod and container in this system. This is achieved through Object Nets with Reference Semantic –a kind of Petri Net (PN) [13]– based models. The models can be further annotated and configured with deterministic time, probability distributions, or functions obtained from monitoring data acquired from a Kubernetes deployment. It can also be used by an application developer/designer: i) to evaluate how pods and containers could impact their application performance; ii) to support capacity planning for application scale-up/scale-down.

5.2 Kubernetes Overhead Analysis & Performance Models

The Kubernetes architecture (Figure C.2) incorporates the concept of a pod, an abstraction that aggregates a set of containers with some shared resources at the same host machine. It plays a *key* factor in the overall performance of Kubernetes. We make use of Reference Nets to model pods and containers and to conduct a performance analysis. Object PNs can be interpreted by Renew[6] [108], a Java-based Reference Net interpreter and graphical modelling tool.

5.2.1 Kubernetes Performance Model

The pod abstraction allows grouping coupled containers which share the access and the use of certain physical resources –e.g. I/O file system, network–. The containers of a pod are scheduled in the same machine, that is to say that the pod is the minimal shedulable unit. Conceptually, Kubernetes supports two kinds of pods:

- *Service Pods*. They are run permanently, and can be seen as a background workload in the cluster. Two key performance metrics are associated with them: i) availability –influenced by faults and the time to restart a pod/container–; and ii) utilisation of the service –which impacts on the response time to clients–. For example, high utilisation leads to an increased response time. Several Kubernetes system services –e.g. container network or DNS– and high level services –e.g. monitoring, logging tools– are provided by Service Pods.

- *Job/Batch Pods*. They are containers that execute tasks and terminate on task completion. For a Job Pod, both deployment and total execution time –including restarting, if necessary– are important metrics. The restart policy of these containers can be *onFailure* or *never*.

[6]http://www.renew.de

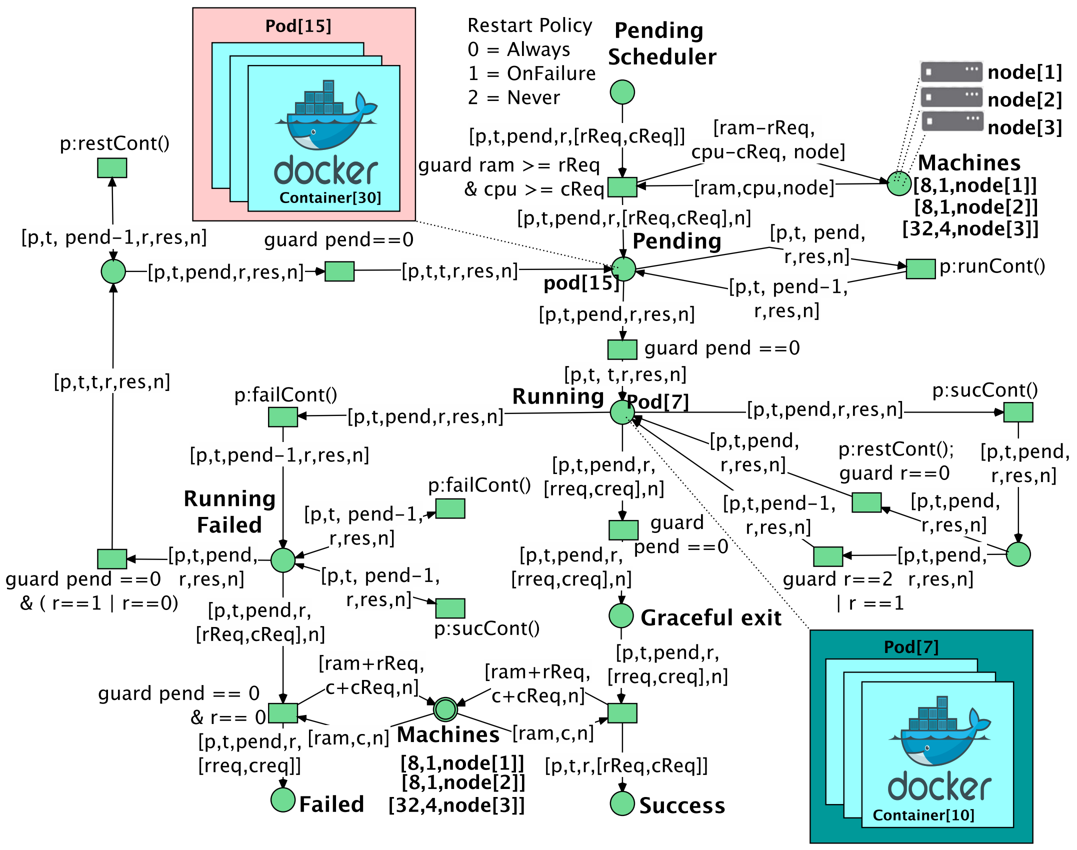


Figure 5.1: Model of the life-cycle of pods in Kubernetes.

When a pod is launched in Kubernetes, it requests resources –RAM and CPU– to the Kubernetes scheduler. If enough resources are available, the scheduler chooses the *best* node for deployment. The requested CPU could be considered as a reservation in contingency situations. For instance, when a container is idle –e.g. it is inside a service pod and the service has low utilisation–, other containers can use the CPU. With this resource model, the overall performance of the pod depends on its resource requests and on the overall workload. We could define a CPU usage limit, but then some resources might remain unused.

We model a pod's life-cycle in order to estimate the impact of different scenarios on the deployment time and the performance of the applications running inside a pod. In Kubernetes, a pod's life-cycle depends on the state of the containers that are inside it. For instance, a pod has to wait until all of its containers are created. With the Reference Nets abstraction, we can provide an unambiguous hierarchical representation of the Kubernetes manager system as the System Net and the pods with the containers as the Token Net. The tokens inside our Token Net represent containers and the tokens inside our System Net represent pods, as illustrated in Figures 5.1 and 5.2. The models were derived from

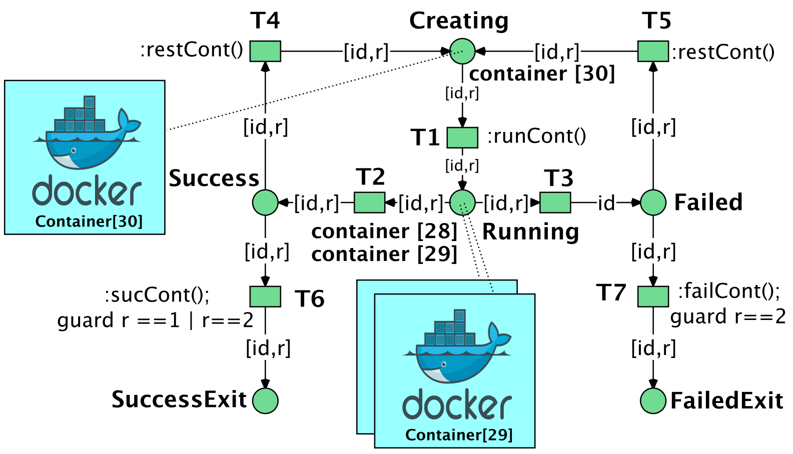


Figure 5.2: Model of the life-cycle of containers inside a pod. r models the restart policy of a container –*Always* = 0, *OnFailure* = 1, *Never* = 2.

the Kubernetes documentation[7], specifically, from the Pod Lifecycle section [8] and from the Resource Management section[9]. Details about places and transitions, needed to specify the initial marking, are hidden to improve legibility. In addition, we assume that the scheduler assigns a pod to a single node arbitrarily, as long as the machine has enough resources available. If there are not enough resources in the cluster, the pod waits in **Pending Scheduling** place. This behaviour could be refined by introducing more sophisticated policies and a rejection place for pods. The **Machines** place[10] represents the resources managed by the scheduler. For each machine, there is a tuple token with the identification of the node, the available RAM size and the number of available cores. Figure 5.1 shows three machines ranging from 8GB to 32GB, with 1 to 4 cores. The resources assigned to a pod are only released when the pod restart policy is *Never* or *onFailure*.

Once the pod has been assigned to a machine, Kubernetes starts creating the containers while the pod waits in its **Pending** place. Both nets are synchronised through the inscription *runCont*. In this way, when a container in a pod enters the **Running** place (Figure 5.2), the number of pending containers in this pod is decremented in the **Pending** place (Figure 5.1). When all containers are running in the pod, the transition with the **guard pend==0** is fired and the pod state changes to **Running**.

While the pod is in the **Running** state, it is waiting for its containers to terminate. If a container fails, the pod enters the **RunningFailed** place where it waits for the termination of all containers –with a potential restart action–. If there are no failures, the pod will be in

[7] https://kubernetes.io/docs

[8] https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/

[9] https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/

[10] It should be noted that the **Machines** place appears twice: one with a single circle –actual definition– and with a double circle –a duplication to simplify the model–. Reference Nets support double circle notation to simplify the model and to improve its legibility. If it were not used, several arcs would cross the model with their corresponding arc labels

Transition	Variable
T1	Time to create a container
T2	Execution time of a container
T3	Time until next failure in a container
T4,T5	Time to restart a container
T6, T7	Time to finish gracefully a container

Table 5.1: Timed transitions in the model.

the **Running** place or eventually it will reach the **Success** place when all containers have finished.

Figure 5.2 illustrates the behaviour of a container. A token in that net represents a container. A pod's restart policy is included in the net. A created pod enters the **Running** place, and may reach the **Success** or **Failure** place. The firing of the corresponding transitions –**T1**, **T2** and **T3**– is synchronised with the System Net. According to the restart policy, the containers might return to the **Running** place or they might finish in **SuccessExit** or in **FailedExit** places. We include several timed transitions, as summarised in Table 5.1. By default, the firing of **T2** and **T3** is arbitrary and non-deterministic; however, with Renew, it is possible to simulate any probability distribution for them in order to simulate a failure. Additionally, it is possible to assign different random distributions for each timed transitions. In the next sections, we describe different experiments to obtain the real value of these metrics. The termination time –**T6** and **T7**– and the termination time when a container is restarted –**T4** and **T5**– do not depend on the success of the container, so both transitions are modelled with the same distribution. When a container is restarted, the total restarting time can be calculated as **T4** + **T1** –or **T5** + **T1**.

5.2.2 Experiments to Feed the Performance Model

We conducted several experiments to estimate the value of transitions in Table 5.1 by deploying Kubernetes on a cluster with eight physical machines, $n = 8$, each with 32GB of RAM and 4 Intel i5-4690 (3.500GHz) cores. The results are shown in the next subsections. The performance metrics of the high level model (Figure 5.1) are determined by the firing sequences of transitions in the Token Net (Figure 5.2). For example, if there is a pod with three containers, the **T1** transition of this pod is fired three times. The pod is waiting this time in the **Pending** place.

5.2.2.1 Benchmarking Starting Time

To estimate the value of transition **T1**, we launched a variable number of containers, whose image was preloaded on all machines, and measured the total deployment time. Each experiment was repeated 30 times, and we calculated the mean, the standard deviation, and the confidence intervals with $\alpha = 0.05$. In order to calculate the confidence interval, we assumed –by the Central Limit Theorem– that the underlying distribution of the sampled mean follows a normal distribution.

In Figure 5.3, we show how different variables influence the deployment time. These variables are:

- The *number of machines* available in the cluster, n. We observe that the scheduler launches pods sequentially, without multi-threading (red line in Figure 5.3), showing a linear total deployment time with the number of deployed containers.

- The *number of containers C inside a pod*, ρ factor. The ρ factor is calculated as follows: $\rho = \frac{\#Pods}{C}$. For instance, a ρ factor of 0.25 means that there are 4 containers inside each pod. We can see that the time to deploy 10 pods with 4 containers in a single machine is 25.89s, which is higher than the time to deploy 10 pods inside a single container on a single machine –16.01s.

- *Cluster and infrastructure constraints* as the number of nodes or the network physical devices used.

The total provisioning time, T_t, is calculated by using Equation 5.1, where T_d is the time to deploy pods and containers on physical machines and T_{down} is the time to download the needed container image to the involved machines.

$$T_t = T_d + T_{down} \tag{5.1}$$

Our experiments show that T_d has a linear behaviour. Therefore, T_d depends on the number of deployed containers C and on the number of deployed pods, $\#Pods$ (Equation 5.2). As the scheduler manages the pod as the minimal schedulable unit, the maximum number of pods deployed in parallel in a cluster is given by $\min\{\#Pods, n\}$. T_c is a function that returns the time to create a single container. This value depends on how the deployment is structured –ρ and C parameters– and the number of machines in the cluster, n.

$$T_d = \frac{C\ T_c(\rho, n, C)}{\min\{\#Pods, n\}} \tag{5.2}$$

In Figure 5.4, we show different values for T_c, obtained experimentally. We can see that for large C values, and as ρ approaches 0, T_c becomes constant. Therefore, we can write:

$$\lim_{(\rho,n,C)\to(0,\infty,\infty)} T_c(\rho, n, C) = t_c$$

Under these assumptions, the T_c value could be considered as a constant –attached to transition **T1**.

In order to characterise the impact of the image download time, we repeated the previous experiments without pre-loading the image. The image is downloaded from a machine located inside the cluster connected directly to the same switch. The results are shown in Figure 5.5. The results of the homogeneous latency scenario are better than the ones of the variable latency scenario –as ρ tends to zero, the latency impact on the results decreases–. We

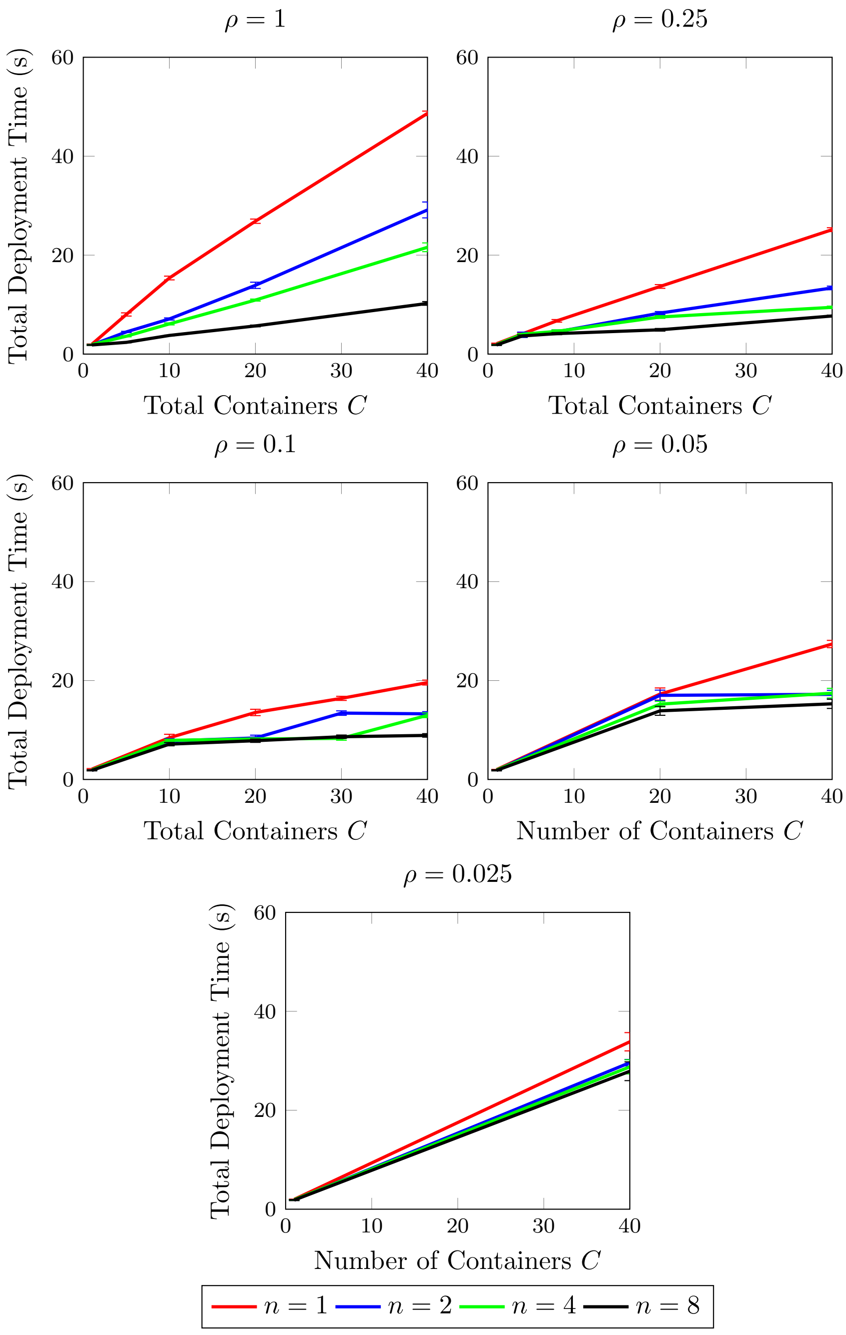


Figure 5.3: Total deployment time, T_d, versus number of deployed containers, C. Each graph shows: mean time and confidence interval for the mean for a variable number of machines in the cluster, n. The results are grouped by the number of containers inside a pod, $\frac{1}{\rho}$.

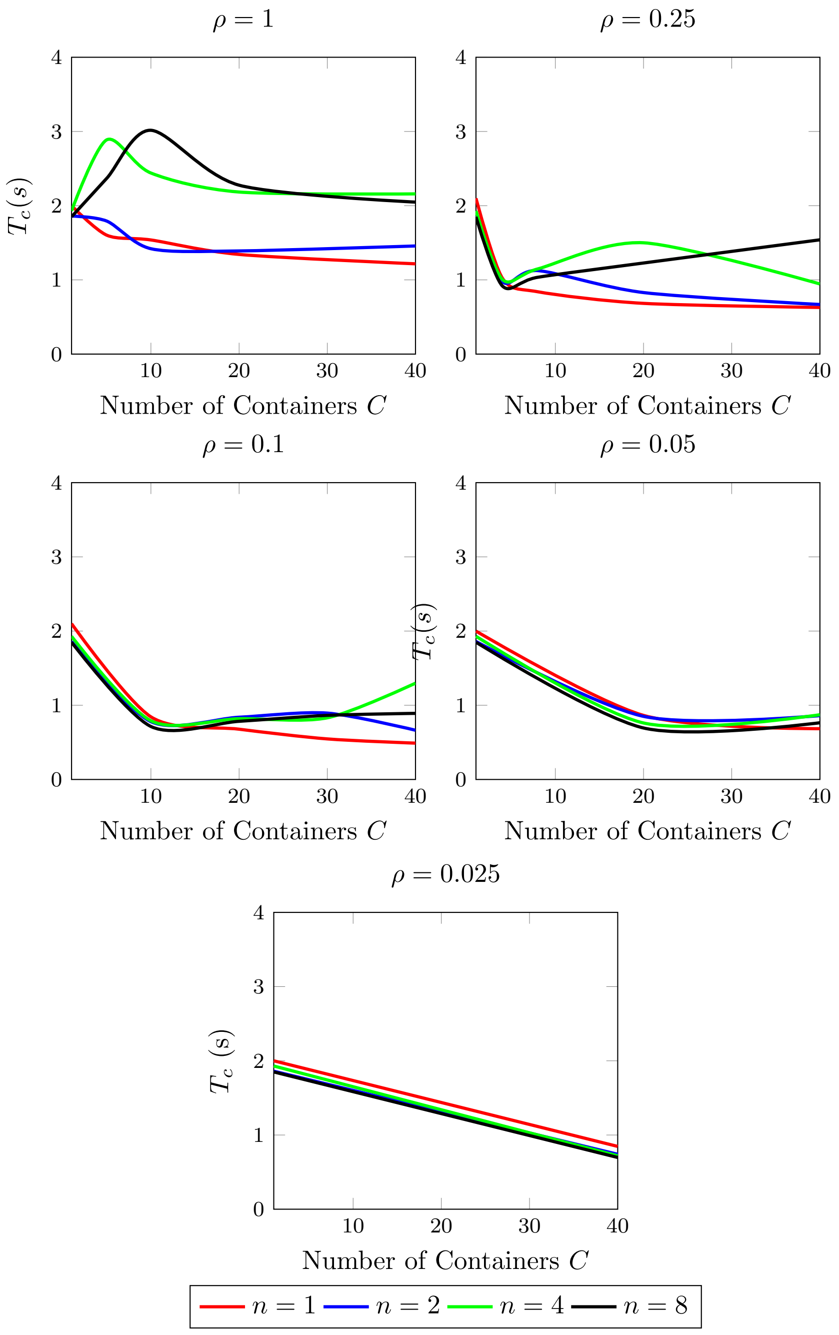


Figure 5.4: Time to create a *single* container, T_c function, versus number of deployed containers, C. Each graph shows: mean time and confidence interval for the mean, for a variable number of machines in cluster, n. The results are grouped by the number of containers inside a pod, $\frac{1}{\rho}$.

	Homogeneous RTT		**Heterogeneous RTT**	
C	T_d	T_t	T_d	T_t
1	1.85	31.09	1.94	33.26
5	2.37	67.28	2.66	66.79
10	3.77	83.44	3.87	82.66
20	5.69	83.81	5.56	86.93
40	10.24	85.47	10.21	91.02

Table 5.2: T_d and T_t values from a Kubernetes cluster with homogeneous RTT –0.25ms– and from a Kubernetes cluster with heterogeneous RTT. $\rho = 1$ and $n = 8$. The container image is 1.225 GB. The results are in seconds.

can see that as the number of machines increases, the total deployment time also increases, because the image needs to be downloaded by all the machines in the cluster, which are connected to the same server. When the number of deployed pods is greater than the number of machines, the deployment time remains stable, so we can conclude that Kubernetes only downloads the image once per machine.

Several variables related to the cluster architecture impact the deployment time, such as parameters of the physical machines and the network topology. To assess the network topology impact, we repeated the experiments in a cluster with heterogeneous latency. We simulated that half of the machines are in a different network area, so that their Round Trip Time (RTT) is about 100ms. The RTT for the rest of the machines is 0.25ms. Table 5.2 depicts the results for $\rho = 1$ and $n = 8$. The results for other values of ρ and n are quite similar. We can see that the latency does not have a significant impact on T_d –and neither on T_c–. As in T_t is included the time to download the container image, this value is higher. However, the size of the image mitigates the latency impact.

5.2.2.2 Benchmarking Termination Time

A pod is expected to be terminated sometime. If it is a service, and consequently it has to be running all the time, the termination may be due to a failure and the pod has to be restarted. This philosophy is also applied to containers, as we discussed previously in the Container net model –transitions **T4**, **T5**, **T6**, **T7**–. We consider **T3** to be dependent on the application, and it represents the failure rate –or the time between failures–. When a pod terminates, Kubernetes waits for a grace period –which, by default, is 30 seconds– until it kills any associated container and data structures.

As far as we have tested, the only variable that affects termination time of a pod is the number of containers in that pod. This occurs because when a pod finishes, all of its containers have to be finished; however, in a normal scenario, pods finish –or restart– asynchronously. Therefore, the overhead caused by terminating several pods on several machines is negligible. As ρ tends zero, the mean time to stop a single container inside a pod remains constant. The way in which these times are aggregated and synchronised depends on the scenario, and the specific performance metrics can be derived from the complete Petri Net model.

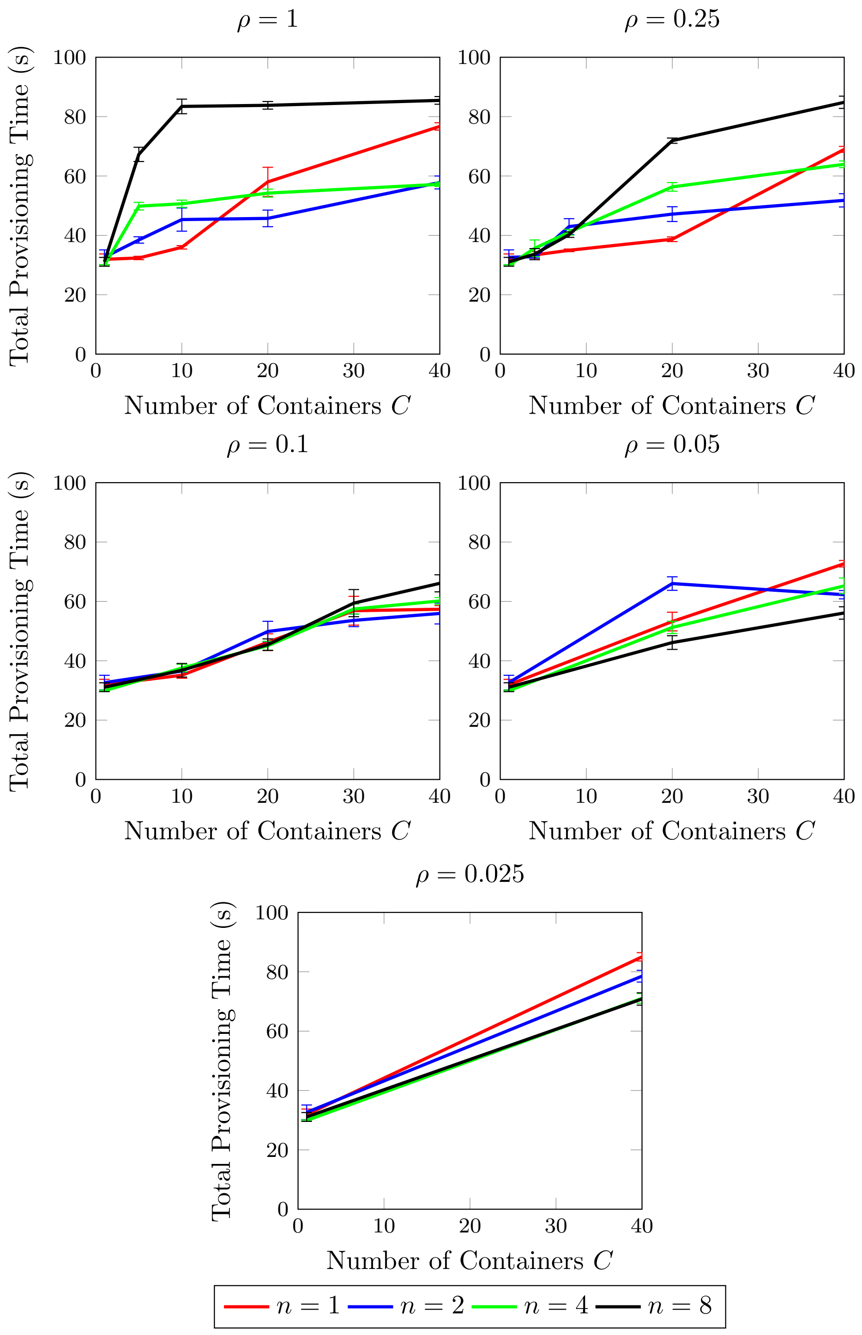


Figure 5.5: Total provisioning time, T_t, versus number of deployed containers, C. Each graph shows: mean time and confidence interval for the mean for a variable number of machines in the cluster, n. The results are grouped by the number of containers inside a pod, $\frac{1}{\rho}$. The container image –1.225 GB– is not present in the machines.

C	ρ	**T4 (T5) per Container**	**T6 (T7) Graceful termination**	**T6 (T7) per Container**
1	1	0.01	30	0
10	0.1	0.11	30.99	0.10
20	0.05	0.12	32.8	0.14
40	0.025	0.15	34.69	0.11
60	0.016	0.16	37.04	0.11

Table 5.3: **T4** and **T6** experimental results, in seconds.

In order to associate the corresponding metric for the transitions, we perform the experiments shown in Table 5.3 in the same cluster, as in the previous section. We present the results for **T4** and **T6**, which correspond to a successful scenario. Without taking into account the time to detect the failure, the behaviour of transitions **T5** and **T7** is similar.

- *Transitions **T4** and **T5***: these transitions measure the time to stop a container when it is going to be restarted. We have deployed pods with a variable number of containers to measure the time. The results are shown in column "**T4** per Container" in Table 5.3. When we decrease ρ, the mean time to terminate a container remains constant. Additionally, the highest measured mean time is ~ 0.3s and 80% of sampled times are < 0.22s.

- *Transitions **T6** and **T7***: these transitions model the normal behaviour of Kubernetes. On successful completion, Kubernetes waits for the grace period and deletes all the data structures associated with a container. We measured these variables in columns "**T6** Graceful termination" and "**T6** per container" in Table 5.3. For these experiments, we set the grace period to 30s –the default value–. We can observe that the stopping time remains constant for more than 10 containers in a pod –column "**T6** per container"– and for low values is negligible. It is interesting to note that the time to stop a container is higher when the container is going to be restarted. This overhead is about 10ms.

5.3 Overhead Analysis of the Pod Abstraction

The pod abstraction allows several containers to be grouped together and share different resources. However, the way in which resources are shared between containers in the same pod and the impact on the performance of a container are not easy to determine. In this section, we analyse this performance change based on how the deployment is structured –e.g. the number of containers inside each pod– by using the total execution time as a metric. We conducted several experiments to measure the overhead induced by the pod abstraction. The aim is to measure how transition **T2** is affected by the deployment configuration.

Let us consider the following scenarios:

C	μ_1	σ_1	μ_2	σ_2	$\mu_1 - \mu_2 = 0$?
1	123.47	0.43	123.38	0.39	Yes
4	473.65	0.96	475.15	0.62	No
8	946.90	0.72	946.63	0.69	Yes
12	1417.76	1.67	1420.40	1.35	No
20	2370.21	1.16	2374.36	3.89	Yes

Table 5.4: `Pov-ray` experiment. Comparison between the execution time (s) for Scenarios 1, μ_1, and for Scenario 2, μ_2, and hypothesis testing.

- *Scenario 1*: A pod is deployed and all the containers are inside that pod, $\rho = 1/c$.
- *Scenario 2*: Several pods are deployed and there is exactly one container inside each one, $\rho = 1$.

The total number of containers deployed is given by C and all pods are deployed on the same machine. The machine has 12 Intel Xeon E5-2620 (2.00GHz) cores and 32GB of RAM. Each experiment, one for each scenario, was repeated 30 times, so that we can consider that the probability distribution of both means follows a normal distribution –by the Central Limit Theorem–. We present the mean execution time, μ_i, and the standard deviation, σ_i. In order to compare both scenarios, we propose the following statistical hypothesis test:

$$\begin{cases} H_0 : \mu_1 - \mu_2 = 0 \\ H_1 : \mu_1 - \mu_2 \neq 0 \end{cases}$$

As we assume that both means follow a normal distribution and they have the same variance, we can use the Student t test [109]. Since there are several resources shared between containers, we can expect different behaviours for each one. In the following subsections, we accomplished a hypothesis test for applications with high CPU usage –`Pov-Ray 3.7`–, high I/O usage –`IOzone` benchmark– and high network usage –`netperf`.

5.3.1 CPU Intensive applications

We used the multi-threaded `Pov-ray` application as a benchmark to measure the overhead of pods for CPU intensive use. Kubernetes inherits from Docker the CPU quota reservation. This contingency mechanism allows a container to reserve a maximum CPU quota. However, the quota is only used when there is contingency in the machine; otherwise, every available CPU is used. The comparison between Scenarios 1 and 2 is presented in Table 5.4. We can see that when the number of containers increases, the null hypothesis should be rejected. Additionally, when H_0 is rejected, Scenario 1 is faster than Scenario 2. The overhead caused by having one container inside each pod is about 0.01%.

C	μ_1	σ_1	μ_2	σ_2	$\mu_1 - \mu_2 = 0$?
1	23.52	0.82	23.19	0.64	Yes
4	60.85	1.45	65.02	1.25	No
8	85.98	2.25	91.36	2.24	No
12	108.54	4.14	91.36	3.40	No
20	153.51	6.47	170.99	5.28	No

Table 5.5: `IOzone` experiment –*iozone -a -i 0 -i 1 -g 4M*–. Comparison between the execution time (s) for Scenario 1, μ_1, and for Scenario 2, μ_2, and hypothesis testing.

C	μ_1	σ_1	$\sum \frac{BW_i}{C}$	μ_2	σ_2	$\sum \frac{BW_i}{C}$	H_0?
1	1.88	0.06	1.88	1.90	0.04	1.90	Yes
4	8.61	0.21	2.15	8.82	0.05	2.20	Yes
8	15.53	0.12	1.94	16.26	0.20	2.03	No
12	14.99	0.21	1.25	16.42	0.38	1.37	No
20	15.10	0.19	0.75	18.32	0.91	0.91	No

Table 5.6: Hypothesis test for network bandwidth (GB) for C `iperf` Clients. `Iperf` server & client are on the same machine.

5.3.2 I/O Intensive Applications

We used `IOzone` as a representative benchmark of an I/O application. Table 5.5 depicts the results –in seconds– for the execution of the `IOzone` benchmark[11]. If we compare both scenarios, we can conclude that there is enough statistical evidence to accept H_0: as the number of pods in a machine increases, the caused overhead is higher. The conclusion of these experiments is that it is better to group all containers in the same pod.

5.3.3 Network Intensive Applications

The network infrastructure of a machine is shared by all the containers inside a pod. All the containers in a pod share the port space and the pod has only one IP address. Sharing the access to the network between several containers might cause an overhead on the container performance. To measure that overhead, we deployed an `iperf` server inside a pod and several clients with the previous scenario configuration. All tests measure the network bandwidth for a 30 second interval of TCP traffic.

The first experiment schedules all the containers at the same machine. In a real scenario, this situation might arise when the scheduler groups pods or containers together with high network traffic among them. In Table 5.6, we show the average bandwidth per container and the hypothesis test. When the number of containers inside the pod is above 4, there is

[11]The `IOzone` benchmark was executed as follows: *iozone -a -i 0 -i 1 -g 4M*

C	μ_1	σ_1	$\sum \frac{BW_i}{C}$	μ_2	σ_2	$\sum \frac{BW_i}{C}$	H_0?
1	108.26	0.04	108.26	108.26	0.04	108.26	Yes
4	110.53	0.39	27.63	110.17	0.84	27.54	Yes
8	113.81	0.47	14.23	115.64	0.51	14.46	No
12	117.42	0.92	9.78	117.53	4.01	9.79	Yes
20	124.74	1.22	6.24	126.52	2.09	6.33	No

Table 5.7: Hypothesis test for network bandwidth (MB) for C `iperf` Clients. `Iperf` Clients are on a different physical machine from the server one.

enough statistical evidence to reject H_0. The best results are achieved when each pod has an isolated container (Scenario 2).

We repeated the experiments with the `iperf` server placed on a machine and the clients scheduled in another machine. Table 5.7 shows the results, which are similar to the previous ones. The bandwidth values from Scenario 2 are higher than those from Scenario 1. From these experiments, we can conclude that deploying several pods with a few coupled containers is better than a single pod with a large number of containers.

5.4 Discussion

We demonstrated that the deployment of an application on a specific infrastructure can impact its overall performance. We used results from our experiments to derive rules that try to improve it. Since in Kubernetes the minimal schedulable unit is the pod, then ρ is the parameter which has the highest impact on performance. We assume that the Kubernetes nodes are homogeneous and that all containers can be distributed across physical nodes, improving the performance of the application –i.e. there is no coupling between containers, and this is considered as a design restriction–. Figure 5.6 summarises the rules to choose the best ρ from our experiments.

If an application is CPU or I/O intensive, it is better to group all containers together –from experiments in Tables 5.4 and 5.5–. However, we want to distribute the pods among as many machines as possible, which is done through the ρ parameter. If the number of containers is greater than the number of machines, ρ should be equal to $\frac{n}{c}$ (**Rule no. 1**) –we have n pods with $\frac{c}{n}$ containers at each pod–. This rule tries to minimise the impact of T_d –which decreases for low values of ρ–. If the number of containers to deploy is less than the number of machines, then $\rho = 1$ (**Rule no. 2.**) –we deploy a pod with a container at each machine.

The ρ choice is different if we consider an application that makes a high use of the network. If it is a service pod and there are few failures in the scenario –equivalently, T_d is negligible– the best choice is to set $\rho = 1$. The reason is that, regardless of the machine where a pod is scheduled, the effective bandwidth is higher when there is only one container inside a pod (Tables 5.6 and 5.7). However, if T_d is relevant, we can calculate the total time

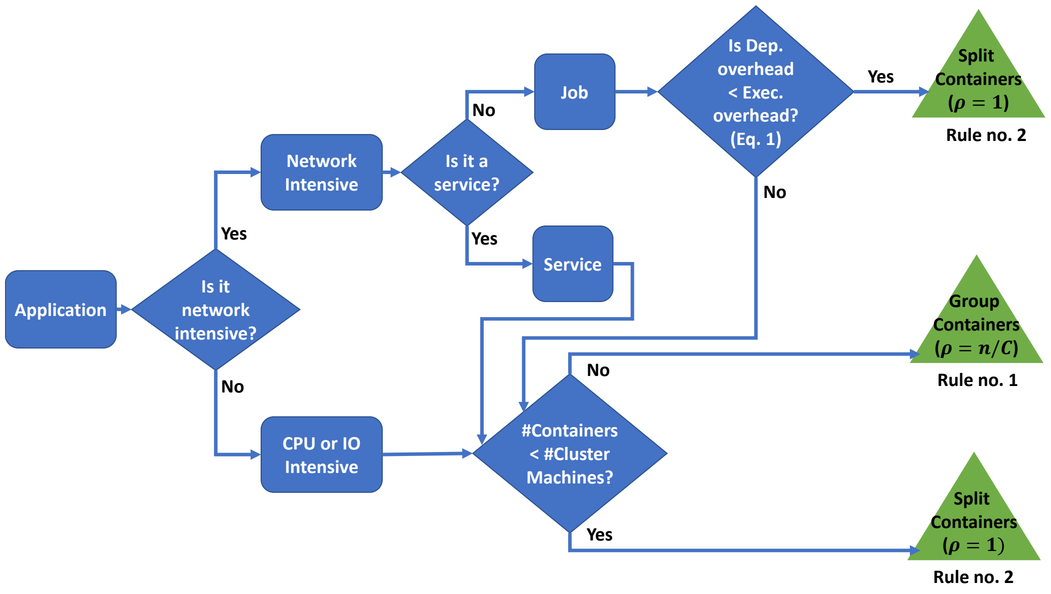


Figure 5.6: Flow diagram to choose the best ρ parameter. C is the number of containers to deploy and n is the number of machines in the cluster.

T_t –deployment time T_d plus execution time T_e– as a function of ρ (Equation 5.3), where $\alpha(\rho)$ is the overhead caused by the pod abstraction (Section 5.3) and it can be calculated as $\frac{\mu_1}{\mu_2}$, where μ_2 corresponds to a scenario with $\frac{1}{\rho}$ containers per pod. In general, as μ_2 is expected to be greater than μ_1, then $\alpha > 1$. Additionally, $\alpha(1) = 1$. Figure 5.7 depicts an example of $T_c(\rho)$, calculated when $C \to \infty$, obtained from Figure 5.4.

$$T_t(\rho) = T_d(\rho) + \alpha(\rho)T_e = \frac{c}{n}T_c(\rho) + \alpha(\rho)T_e \tag{5.3}$$

;

It is a complex task to minimise the function $T_t(\rho)$. As a simplification, we can assume that $\alpha(\rho)$ remains constant and when n tends to infinity, the mean time to create a container also remains constant. In our experiments, for low values of ρ (Table 5.7), its value is approximately 1.01. Assuming that the major improvement in the execution time is achieved by executing tasks in parallel, we can compare the situation where $\rho = 1$ versus $\rho = \frac{n}{c}$. The first one will be faster than the second one when Equation 5.4 is satisfied –**Rule no. 2** should be applied. Otherwise, **Rule no. 1** will be more suitable.

$$T_t(1) < T_t(^n/_c) \implies \frac{c}{n}T_c(1) + T_e < \frac{c}{n}T_c(^n/_c) + \alpha T_e \implies$$

$$T_c(1) - T_c(^n/_c) < \frac{n(\alpha - 1)}{c}T_e \tag{5.4}$$

These rules are based on the experiments in Section 5.3. Other container technologies –such as Linux LXC or Core OS rocket– can be abstracted in a similar way. The use of a

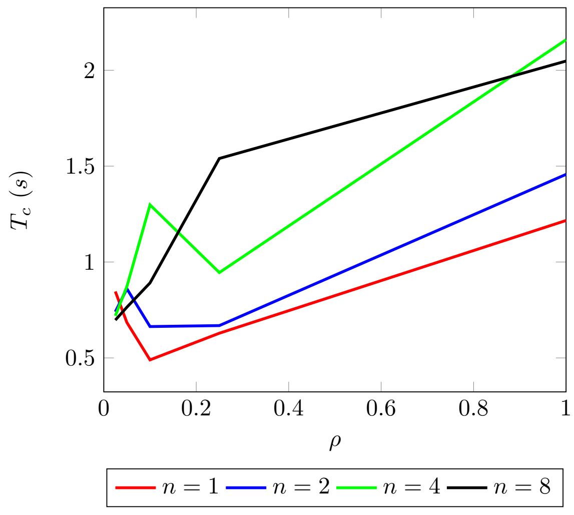


Figure 5.7: Function $T_t(\rho)$ for different values of n. The number of deployed containers is assumed to tend to infinity.

particular technology does not have an impact on our model, as many of these container frameworks will also share similar life-cycle states. However, the performance values may vary depending on the use of a particular container framework/technology. In Section 2.5, we provide a comparison of the performance of different technologies. Additionally, there are different container management systems such as Docker Swarm or Apache Mesos. Although these other platforms do not have the pod abstraction, our models and results could be relevant to them in scenarios where $\rho = 1$.

In our work, we have proposed a methodology to feed a formal model and to analyse the overhead of the pod abstraction. This methodology should be applied to different configurations in order to be generalised. For instance, all of our experiments were carried out within a private cloud and Kubernetes was deployed over a bare-metal system. This configuration allows us to avoid the additional overhead caused by the execution of Kubernetes inside VMs. On the other hand, the Google Cloud Platform allows the possibility of running a Kubernetes cluster; however, the containers are run over VMs, which may have an impact on the performance and the hypothesis tests may change. Besides, the underlying service architecture is different. For example, since the storage service is accessed over the cloud, the I/O intensive application will have a different behaviour, and the overhead caused by the pod abstraction may not be negligible.

5.5 Conclusions

An effective automated resource management in cloud computing requires to launch, terminate and maintain computing instances quickly, with a minimum overhead. In this chapter, we conduct a performance analysis over Kubernetes, achieved through a Petri Net-based performance model. It allows us to analyse the deployment and termination overheads of containers in Kubernetes, as well as understanding the performance of different configurations of a Kubernetes pod –e.g. the influence of the number of containers per pod–. We conducted our analysis in a Kubernetes cluster of 8 machines. Our model can be exploited as a basis to improve two activities: i) capacity planning and resource management; and ii) application design, specifically how an application may be structured in terms of pods and containers. From our experiments, we can see that a single container can be deployed in a time interval that ranges from less than a second to up to 3 seconds, depending on the circumstances –e.g. the number of pods per container, the number of containers deployed simultaneously, the network latency, or the number of host machines–. In contrast, the termination time is typically in the order of a tenth of a second. Moreover, we also provide a set of rules that assist in allocating the number of containers per pod to provide the best performance. These rules consider a number of characteristics of the application, such as the CPU or network usage.

Chapter 6

Client-side Scheduling Based on Application Characterisation on Kubernetes

To wisdom belongs the intellectual apprehension of things eternal; to knowledge, the rational apprehension of things temporal.

Saint Augustine of Hippo

Continuing with the Kubernetes cluster example, in this chapter we analyse how the decision of executing several containers in a machine might have an impact on the performance of those containers. Applications are treated as a black box –the system does not know anything about them–; however, we propose to bring near the client or user the scheduling process. This means that we fill the gap between the functional and operational level –see Chapter 1– with a client-side scheduling.

In practice, for our considered scenario, the client gives informal information to the system to improve the deployment of his applications. The client-side scheduler uses this information to help the specific operational scheduler –in this case, the Kubernetes scheduler–. The information managed by the scheduler is related to the operational level, for example, if the application is CPU intensive or I/O intensive. How these informal criteria could be refined is presented in the next chapter.

The chapter is structured as follows. First, in Section 6.1, we present the contention in virtualised environments. Section 6.2 shows the effects of resource contention and Section 6.3 presents our proposed architecture to deal with interference, and shows how an application characterisation can help the scheduler to improve overall performance. Finally, the chapter ends with the conclusions and future work in Section 6.4.

6.1 Contention as a Scheduling problem

With the rise of the cloud computing paradigm and the emergence of its technologies, computational power can be adjusted on-demand to the processing needs of applications. Developers can currently choose among a number of cloud computational resources such as Virtual Machines (VMs), containers, or bare-metal resources, having each their own characteristics. A VM can be seen as a piece of software that emulates a hardware computing system and typically multiple VMs share the same hardware to be executed. Nevertheless, VM utilisation can sometimes be difficult to achieve, e.g. when the applications to be run do not consume all the resources of a VM .

Containers are rapidly replacing VMs as virtual encapsulation technology to share physical machines [57, 58, 62, 110]. The advantages over VMs are a much faster launching and termination time overheads, and an improved utilisation of computing resources. Indeed, the process management of container-based systems allows users to adjust resources in a fine grained fashion, closely to the granularity of many applications enabling single containers or groups of them to be deployed on-demand [111]. Finally, container-based platforms, such as Kubernetes, also allow automating deploying and scaling of containerised applications, simplifying the scaling of elastic applications.

As happened with VMs, containers also exhibit resource contention, which leads to unexpected performance degradation. In general terms, resource contention arises when the computing demand from the applications being executed exceeds the overall computing power of the shared host machine. In particular, resource contention appears in containers, when the demand of multiple containers in the same host machine exceeds the supply, understood in terms of CPU, memory, disk or network. This phenomenon can happen in spite of the isolation mechanisms integrated with container technologies, namely Linux namespaces and Linux Control Groups, which isolate the view of the system and limit the amount of computational resources, respectively. Therefore, the development of applications on these platforms requires new research on scheduling and resource management algorithms that reduce resource contention while maximising resource utilisation. Existing platforms like Kubernetes already incorporate a reservation mechanism in order to reduce resource contention. However, such mechanism is only for CPU and for the maximum amount of memory, and can decrease resource utilisation.

In this chapter, we propose a client-side scheduling approach in Kubernetes that aims at reducing the resource contention phenomenon in container technologies. Our approach makes use of application characterisation in terms of the usage of resources, and extends the Kubernetes scheduler so that it can take better allocation decisions on containers based on such characterisation. Our application characterisation consists of dividing applications in categories, namely high and low usage of a certain resource. We propose to delegate the classification process of applications to the client or developer. Thus, he can give informal characterisation to his application to improve the scheduling algorithm.

Then, we extend the Kubernetes scheduler behaviour, in essence, we try to avoid that containers wrapping applications with high usage of a resource –e.g. CPU or disk– coincide in the same host machine. Finally, we conducted experiments with real-world applications,

Application	**Resource**	**Notes**
`Pov-ray`	CPU	Version 3.7 with default paralelism
`Stream` [115]	Memory Bandwidth	*-DSTREAM_ARRAY_SIZE=100000000* *-DNTIMES=100*
`dd`	Disk I/O Bandwidth	*dd if=/dev/zero of=/root/testfile* *bs=1G count=1 oflag=direct > dev/null*

Table 6.1: Applications used as a background workload with the resource which they use intensively and with the chosen execution parameters.

such as WordCount and PageRank, in operational stream processing frameworks, such as Thrill [112] and Flink [113], and compared the results with the standard Kubernetes scheduler.

6.2 Resource Contention on Kubernetes

When several containers are running on the same machine, they compete for the available resources. As the container abstraction provides less isolation than virtual machines, sharing physical resources might lead to a degradation in the performance of the applications running inside the containers.

To avoid this situation, Kubernetes provides a resource reservation mechanism. That mechanism has two main restrictions. The first one is that the reservation is only for CPU and for the maximum amount of RAM . However, the resources that are shared in a machine which might degrade the performance are not restricted to those. For instance, the network bandwidth is shared among all containers in the same machine, and the network access is shared for all containers inside a pod [114]. Other shared resource is the memory bandwidth. The second issue is that a reservation mechanism can lead to unused resources in the cluster. An application might reserve an entire core –CPU limit in Kubernetes terminology– but it only uses the resource sporadically.

We executed several applications on the same machine to characterise how the performance degrades. The machine has two E6750 cores and 8GB of RAM. The chosen applications are a map-reduce application, WordCount, and a webgraph application, PageRank [101], expecting that PageRank makes a higher CPU usage than WordCount. Additionally, we ran both applications inside two different frameworks for data stream processing: Flink [112] and Thrill [113]. We chose both of them because they are implemented in different programming languages –Flink is implemented in Java, whereas Thrill is implemented in C++–. We ran each experiment ten times, and we plotted their mean values.

The first set of experiments consisted in running one application per experiment – WordCount inside Flink, WordCount inside Thrill, PageRank inside Flink and PageRank inside Thrill– along with a background execution caused by another application that makes an intensive usage of a certain resource. A ray tracing program, `Pov-ray` [125], the `Stream`

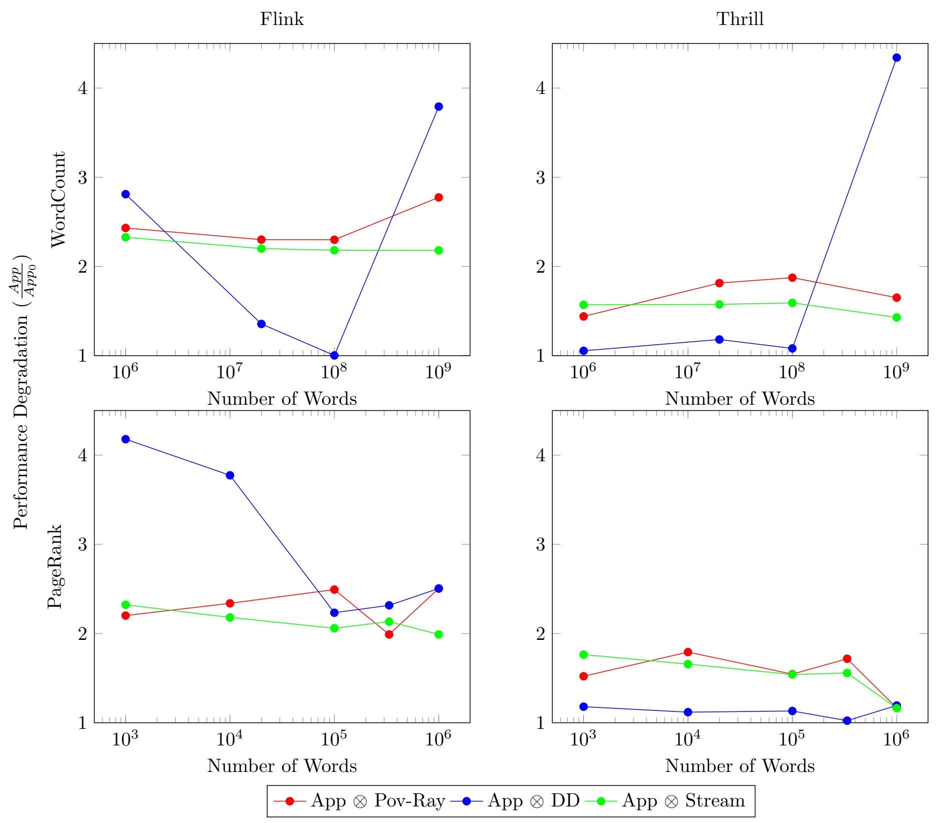


Figure 6.1: Performance degradation for several applications –WordCount inside Flink, WordCount inside Thrill, PageRank inside Flink and PageRank inside Thrill– when executed with a background workload.

benchmark [115], and a file transfer and conversion Unix command, `dd`[1], are used as workload background applications. These three applications were executed in a continuous loop. A summary of the parameters used, their version, as well as the resource they use intensively is depicted in Table 6.1.

For the experiments, we ran WordCount and PageRank and varied the input size in order to observe how their performance degrades over a long execution time. For PageRank applications, we selected the Barabasi-Albert graph which was generated using the NetworkX package[2]. As the reference time, we take the execution time of each application in isolation, without the background application, App_0. Given the execution time of that application with a certain background workload, for instance App_{pv}, we calculate the performance degradation as $\frac{App_{pv}}{App_0}$. Results are shown in Figure 6.1. From this experiment, we can see that:

- The implementation of Thrill is much more efficient than Flink in all cases. This means

[1] dd Linux user's manual (2010)
[2] https://networkx.github.io/

that Thrill uses less resources than Flink, and when it is co-scheduled with applications that use a certain resource with a high intensity, its performance is not degraded too much. For instance, the performance loss when co-scheduled with `Pov-ray` is less than two times for all experiments.

- There is a significant performance degradation –about four times– when we execute WordCount for big input sizes –one thousand million words–, when it is co-scheduled with `dd`. The explanation is that the input size is 6.76 GB, so there are a lot of page faults in the execution and the application is continuously accessing the storage system at the same time as `dd`.
- There is a significant performance anomaly when executing WordCount and Flink co-scheduled with `dd` for small input sizes. This is due to the implementation of Flink regarding I/O access and due to the computational time being negligible in comparison with the overheads for accessing disk for small input sizes.

In the second set of experiments, we have measured the degradation caused in real scenarios. We execute on the same physical machine the following scenarios for each application –WordCount and PageRank–: i) one instance of Flink/Thrill; two instances of Flink/Thrill and four instances of Flink/Thrill; and ii) one instance of Flink and one instance of Thrill; two instances of Flink and two instances of Thrill. Results are shown in Figure 6.2. We can observe that in Flink, the degradation is similar when there is another Flink or Thrill application. When there are four applications, the performance is highly degraded with Flink applications in the WordCount example. The results are very similar in Thrill experiments.

Overall, we can see that the degradation is higher when two or more instances of the same container are scheduled in the same machine. The reason for this behaviour is that both applications use the same resources at the same time, so the contention is higher. However, we can see that when two or four applications are co-scheduled together, the degradation is below two. This means that, from the point of view of the cluster operator, it is more time-efficient to co-schedule both applications together instead of executing them sequentially. From the point of view of the client, a sequential execution of applications means that one client is going to have a low degradation value –near 1– and other client has to wait for the first application to finish. Equation 6.1 gives the condition to be met when the sequential execution is better than the co-scheduling solution. The notation $t_{App_1 \otimes App_2}$ denotes the execution time of application App_1 when it is co-scheduled with application App_2.

$$max\{t_{App_1 \otimes App_2}, t_{App_2 \otimes App_1}\} \leq T_{App_1} + T_{App_2} \tag{6.1}$$

6.3 Client-side Scheduling

Pods are allocated into machines by the Kubernetes scheduler. Kubernetes provides a label-based mechanism, which allows it to place the pods in machines which satisfy certain conditions. For example, the application can request a machine with a solid disk. However,

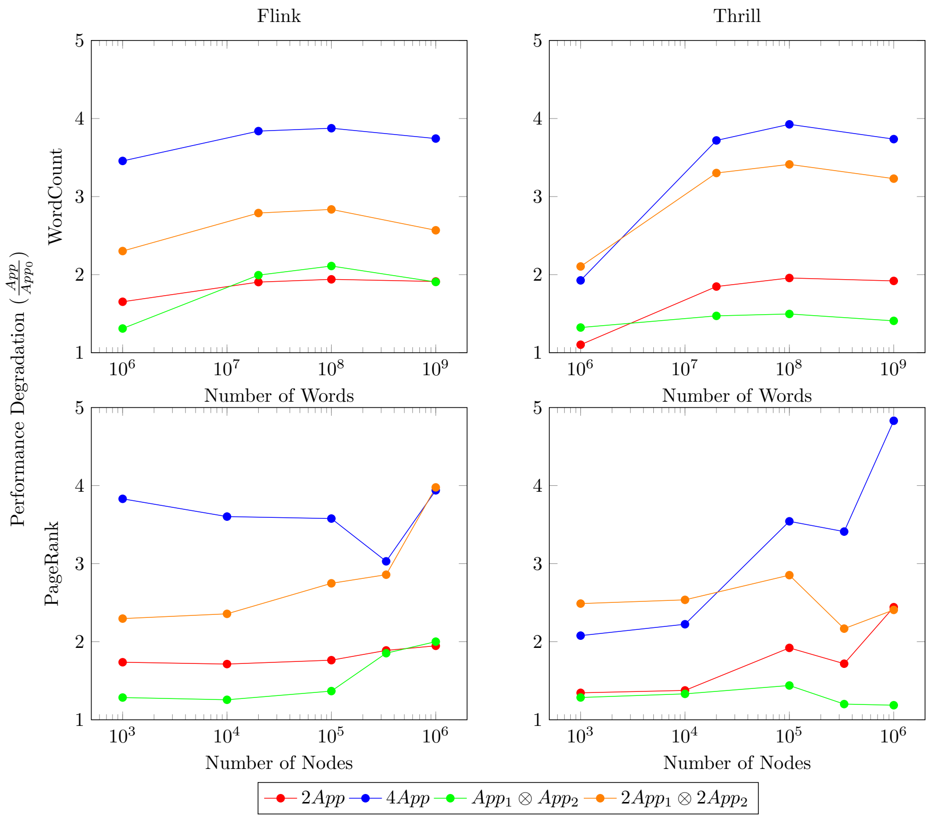
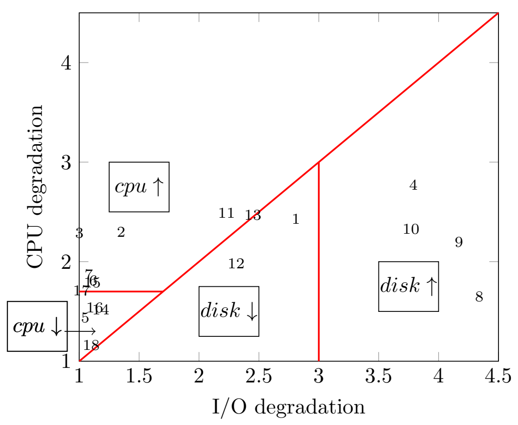


Figure 6.2: Performance degradation for several Apps –WordCount inside Flink, WordCount inside Thrill, PageRank inside Flink and PageRank inside Thrill– when executed in different configurations.

this mechanism entails that the client knows which kind of labels are provided by the cluster. This mechanism is insufficient to deal with the problem presented in Section 6.2. In this section, we introduce a methodology to characterise applications in an informal way. The implemented client-side scheduler uses the characterisation as a guideline to allocate pods inside machines.

6.3.1 Application Characterisation

In certain cases, applications can be classified depending on which resource they use more intensively –CPU, I/O disk, network bandwidth, or memory bandwidth–. An application which is writing in disk continuously has a different behaviour from another one which makes an intensive use of CPU. For illustrative purposes, we only consider applications that make an intensive use of CPU or an intensive use of I/O disk. In our previous experiments, `Pov-ray` was the application which exemplifies a high CPU application and the `dd` command exemplifies a high I/O disk utilisation. Real applications might have an intensive usage of a resource, but with different degrees. For example, the `bzip` application, used to compress



(a) CPU degradation versus I/O degradation.

App	Input Size	Id	Category
FlinkWC	$1 \cdot 10^6$	1	*CPU* ↑
FlinkWC	$20 \cdot 10^6$	2	*CPU* ↑
FlinkWC	$100 \cdot 10^6$	3	*CPU* ↑
FlinkWC	$1000 \cdot 10^6$	4	*disk* ↑
ThrillWC	$1 \cdot 10^6$	5	*cpu* ↓
ThrillWC	$20 \cdot 10^6$	6	*CPU* ↑
ThrillWC	$100 \cdot 10^6$	7	*CPU* ↑
ThrillWC	$1000 \cdot 10^6$	8	*disk* ↑
FlinkPR	1000	9	*disk* ↑
FlinkPR	10000	10	*disk* ↑
FlinkPR	100000	11	*CPU* ↑
FlinkPR	334863	12	*disk* ↓
FlinkPR	$1 \cdot 10^6$	13	*CPU* ↑
ThrillPR	1000	14	*CPU* ↓
ThrillPR	10000	15	*CPU* ↑
ThrillPR	100000	16	*CPU* ↓
ThrillPR	334863	17	*CPU* ↑
ThrillPR	$1 \cdot 10^6$	18	*CPU* ↓

(b) Application characterisation.

Figure 6.3: Application characterisation based on the CPU and the I/O degradation. Numbers in Subfigure 6.3a are application identifiers in Table 6.3b.

large files, has an I/O intensive behaviour that is less than the usage made by the `dd` example. This behaviour can be modelled by defining of several intensity usage grades. For the sake of simplicity and as we want to propose a general methodology, we are going to use here only two grades of resource usage, a high usage of the resource –with a ↑ notation– and a low usage of the resource –with a ↓ notation–. Nevertheless, we acknowledge that the number of grades is a determinant aspect for the scheduling performance that needs to be addressed, and there is a variety of approaches in literature that can be exploited to determine it better, such as classification and clustering data mining algorithms.

Therefore, in our approach, we have a total of four categories: high CPU usage ($CPU \uparrow$), low CPU usage ($CPU \downarrow$), high I/O disk usage ($disk \uparrow$), and low I/O disk usage ($disk \downarrow$). The characterisation of an application in one of these categories allows the scheduler to take better allocation decisions. As the simplest method, the client or the developer should provide the category which better fits his application. Although the categories are very intuitive, alternative sophisticated methods can be developed to classify applications automatically. In order to illustrate our methodology, in Figure 6.3, we show a possible characterisation. We have plotted the I/O degradation –the number of times the application is slower when it is scheduled in the same machine along with `dd`– versus the CPU degradation. The same procedure has been done with `Pov-ray`. We used `dd` and `Pov-ray` as benchmarking applications, however, other applications which make a high usage of a single resource can be used. The values were taken from the experiments shown in Figure 6.1. The red lines split the four categories, and they were obtained with qualitative criteria. Then, we classified each application taking as criteria the resource which caused more contention. The plotted numbers are the identifier of the corresponding application, which are shown in Table 6.3b.

Algorithm 2 Client-side scheduler

```
procedure Client-Side Scheduler($l_{app}, W$)
    $\boldsymbol{S}$ = GetClusterState()
    $minValue := \infty$
    $bestNode := 0$
    for $N$ in $\boldsymbol{S}$ do
        if $|N| \leq min\{|M|, \forall M \in \boldsymbol{S}\})$ then
            if $minValue > \sum_j^{|N|} w_{j,app}$ then
                $minValue := \sum_j^{|N|} w_{j,app}$
                $bestNode := N$
            end if
        end if
    end for
    $Allocate(l_{app}, bestNode)$
end procedure
```

$App_1 \backslash App_2$	$CPU \uparrow$	$CPU \downarrow$	$disk \uparrow$	$disk \downarrow$
$CPU \uparrow$	5	4	2	1
$CPU \downarrow$	4	3	1	0
$disk \uparrow$	2	1	5	4
$disk \downarrow$	1	0	4	3

Table 6.2: Weight matrix W for two resources and two usage grades.

6.3.2 Client-side Scheduling

We propose a scheduler which meets two criteria: i) balancing the number of applications in each node; ii) minimising the degradation in a machine caused by resource competition. Formally, let us define a node N as a multi-set of labels. Each label represents an application that is running inside that node. In our example, we have four kind of labels –l_0 corresponds to $CPU \uparrow$; l_1 corresponds to $CPU \downarrow$, and so on–. In a certain moment, the state of the cluster $\boldsymbol{S}$ can be modelled as a set of nodes. Given a new application whose label is l_{app}, the best node to allocate l_{app} is given by Equation 6.2, where $w_{k,l}$ is the weight of the *k-th* row and *l-th* column of a weight matrix W.

$$\underset{i \in 0}{\overset{|E|}{\operatorname{argmin}}} \sum_j w_{E_{i,j},app} \tag{6.2}$$

Each $w_{k,l}$ models the penalty to schedule a new application labelled as l, if in that node is running an application labelled as k. $E_{i,j}$ is the *j-th* application label of *i-th* node in E set. E is defined as $E = \{N \in \boldsymbol{S} \wedge \forall M \in \boldsymbol{S}, |N| \leq |M|\}$. The E set contains the nodes with less applications. Algorithm 2 implements the previous formalisation.

In order to obtain the weight matrix W, we provide the following rules: for each element $w_{k,l}$, we check if the labels are associated with the same resource. If that is the case, then we set high values of penalty –3, 4, or 5 in our example–. Then, we check the grade of usage. From the previous values, if both grades are high, we set the highest penalty value –5–; if only one is high, then we associate the medium value –4–; and if both are low, then the lowest penalty value is set –3–. If the labels are not associated with the same resource, we repeat the same process to associate the low values –0, 1, or 2– if i and j correspond to different resources. The resulting weight matrix W is depicted in Table 6.2.

6.3.3 Experiments

We made some experiments to compare our client-side scheduler with the default Kubernetes scheduler. The proposed scheduler was implemented in Python. The experiments were conducted in a Kubernetes cluster with 8 machines –each machine has four i5-4690 cores and 8 GB of RAM–. One of the machines acts as a dedicated master node. In the proposed scenario, we ran six applications three times –`dd` and `Pov-ray` with parameters from Table 6.1; PageRank in Thrill and Flink with 1 million nodes and WordCount in Thrill and Flink with 1,000 million words– with the default Kubernetes scheduler. The scenario was executed ten times. As the Kubernetes scheduler has a non-deterministic behaviour, we show three reference cases in Figure 6.4. Each bar represents the execution time of the application, and its colour indicates the machine where the scheduler placed the application. The vertical line shows the total time measured for the experiment –time to create the pods + execution time + time to delete the pods–. Case number 1 represents the worst case. Kubernetes allocated `WCFlink1` and `WCFlink2` in the same machine, with an execution of `dd` and `PRFlink1`. As a result, the execution time of `WCFlink1` is more than 10 minutes, due to the degradation caused by sharing the machine with `WCFlink2`. Case number 2 represents a balanced case, with an execution time of approximately 10 minutes. The scheduler placed again `WCFlink3` and `WCFlink2` in the same machine, so there is a certain degradation in the performance. The best case corresponds to Case 3. In this situation, Kubernetes allocated `WCFlink1`, `WCFlink2` and `WCFlink3` in different machines and the result is better –approximately 8 minutes–. From these experiments, we can conclude that, as Kubernetes has a non-deterministic behaviour, the execution time of the applications has a high variance. If the scheduler splits the applications with an intensive CPU usage among different machines, the results are better; however the decision is taken randomly. Additionally, we can see in Case 2 that the default Scheduler does not try to balance the number of applications among the number of machines –the scheduler places four applications in node3 and only one application in node5.

The results of the same experiment with our scheduler are shown in Figure 6.5. Its behaviour is deterministic, so under the same conditions, the scheduler allocates the applications in the same machine –in the figure we only show two of the ten executions, due to the low variance–. The overall execution time of the experiment is approximately eight minutes. This value is 20% better than the mean time of the Kubernetes scheduler –approximately 10 minutes–, and it is significantly better –33%– than the worst case –approximately 12

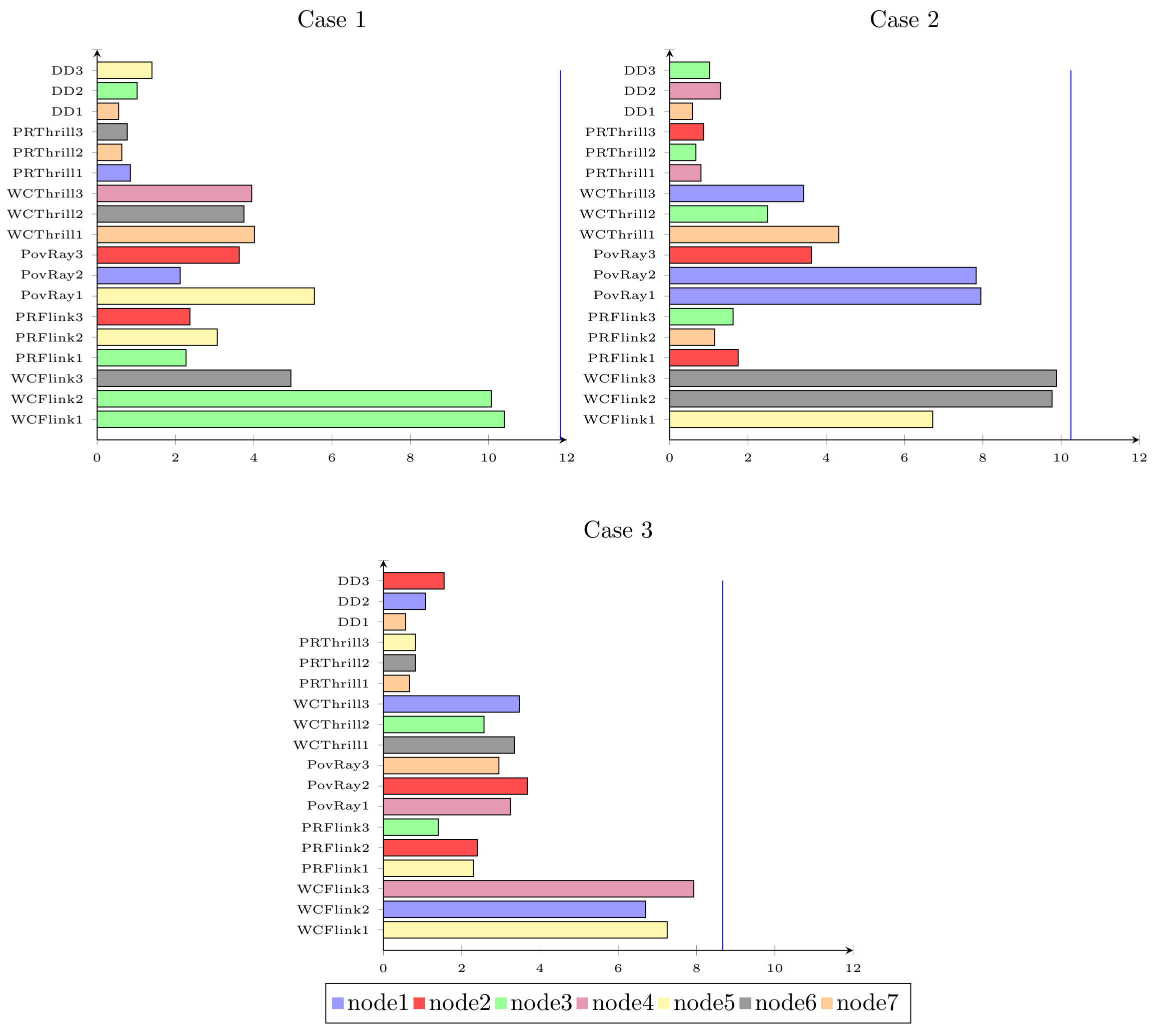


Figure 6.4: Execution time and machine allocation with the default Kubernetes scheduler. The blue line shows the total measured time –execution time + time to create pods + time to delete pods.

minutes–. The total time is similar to the best case of the default scheduler. Additionally, the variance in the execution time is lower. The improvement is achieved by deploying the applications with a high CPU utilisation –`WCFlink1`, `WCFLink2` and `WCFlink3`– on different machines.

In our last set of experiments, we executed the same batch of applications while using the reservation mechanism available in Kubernetes. As `WCFlink1`, `WCFlink2` and `WCFlink3` have the highest execution time, we reserved two cores for them. For the remaining applications, we reserved only one core. The results are displayed in Figure 6.6. The red line compares the total time to our scheduler denoted by the blue line. We can see that the total time is

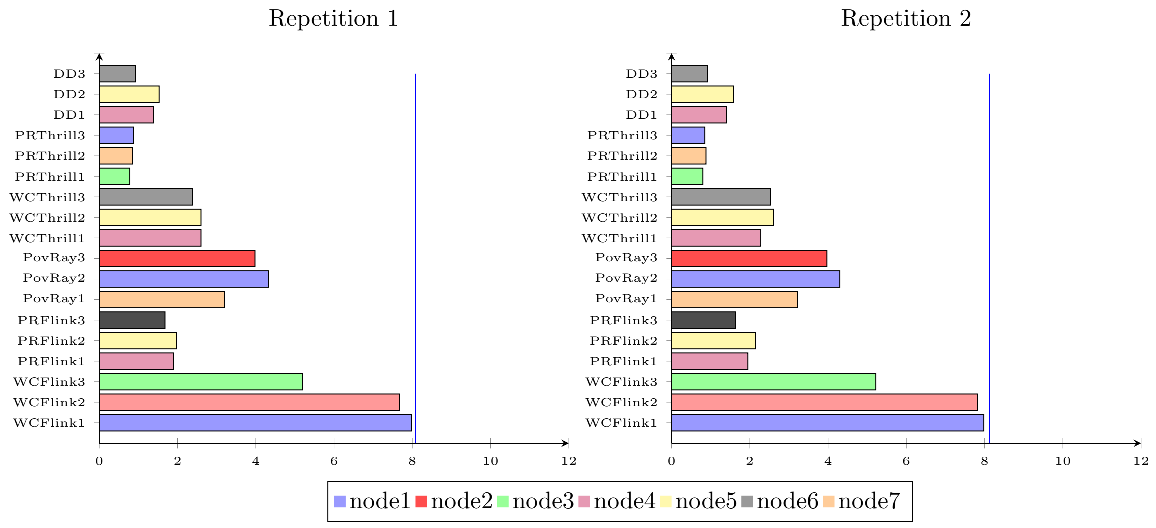


Figure 6.5: Execution time and machine allocation with the proposed scheduler. The blue line shows the total measured time –execution time + time to create pods + time to delete pods.

almost twice. The reason for this behaviour is that there are a lot of unused resources in the machines. Additionally, the variance in the execution time is very high –for instance, `Pov-ray1` has an execution time of nearly 12 minutes and `Pov-ray2` has an execution time of nearly 16 minutes–. This can be explained due to the fact that there are other resources that cause performance degradation which are not reserved.

6.4 Conclusions

Container virtualisation provides a quick and flexible mechanism to share computational resources in machines while improving resource utilisation, as compared to other cloud resources such as Virtual Machines. However, the low isolation between container-based applications can lead to performance degradation in those applications. In our work, we have shown that the default mechanisms to isolate resources between containers in Kubernetes are not sufficient to lead with the performance degradation. Although CPU is the main source of degradation, the competition for other resources –I/O disk, memory bandwidth and network– should be included in the model. Moreover, our experiments show that the CPU reservation mechanism can lead to unused resources in the cluster, and the execution time of applications might have a high variance caused by degradation produced by other sources distinct than the CPU.

As a solution to deal with the competition of resources between containers, we propose a scheduling technique based on the characterisation of applications. Clients or developers provide informal information about their applications –for instance, which resource the application uses more intensively– and in turn, the scheduler uses that information to allocate

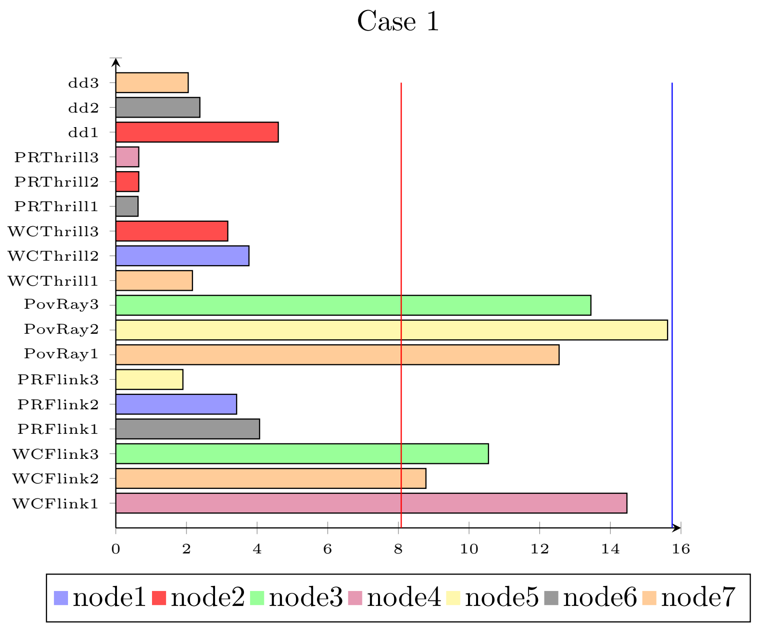


Figure 6.6: Execution time and machine allocation with the Kubernetes CPU limit mechanism. `WCFlink1`, `WCFlink2` and `WCFlink3` are executed with two cores and the remaining applications with one core. The blue line shows the total time obtained with the proposed scheduler and the red line shows the total measured time –execution time + time to create pods + time to delete pods.

the applications using the same resource in different machines. Bringing near the client the scheduling process means that we can have more information about the black box instances. In our experiments, we achieved about a 20 percent improvement in the execution time of a simple scenario compared to the default Kubernetes non-deterministic scheduler. The total execution time is about the half compared to a scenario were resources are reserved in Kubernetes. Additionally, the behaviour of our scheduler is deterministic, so it can be used for further analysis.

Finally, in the next chapter, we propose a methodology to characterise applications by using profiling techniques. The upcoming model is a refinement of the characterisation proposed in this chapter. It allows us to model changes in the resource usage patterns. Thus, instead of using fixed categories which represent a certain hardware resource with a usage pattern –e.g., high CPU or low I/O–, the new model allows the description of variations in time for the use of that resource.

Chapter 7

Timed Indices to Characterise Interference in Container Environments

Knowledge of the fact differs from knowledge of the reason for the fact.
Aristotle

In previous chapters, we have presented a hierarchical model to describe the usage of Kubernetes and an approach to deal with the interference between containers focused on the scheduler. In this chapter, we present a methodology to describe containerised applications by using four resource indices. Unlike machine-learning based models [80,81], these indices are very tight to the resources, so they are easily interpretable and they allow a better behavioural and resource analysis. Moreover, the indices represent how the resource usage varies over time.

We propose using these indices to estimate the execution time of containers when they are competing with other containers in the same physical machine, namely, they cause interference between them. Although our methodology is platform agnostic, this estimation can be used directly in the Kubernetes model presented in Chapter 5 to make simulations and estimations about an entire container management system. Additionally, the characterisation of applications with this technology can be used by the client-side scheduler presented in Chapter 6 to refine how applications are characterised.

The chapter is structured as follows. We begin defining and explaining the resources that might cause interference in applications (Section 7.1). Then, we present the CFA model in Section 7.2. The methodology to define the interference indices and to characterise applications with them is presented in Section 7.3. In Section 7.4, we propose a multiple linear regression model to estimate the interference and the performance degradation between applications running in the same machine. Section 7.5 shows the experiments conducted to test our models. In Section 7.6, we discuss certain parameters and assumptions of our estimation model. Finally, the chapter ends with the conclusions and future work in Section 7.7.

7.1 Containers and Sources of Interference

The container virtualisation paradigm enables developers to share physical resources among applications. Containers decouple the application and its dependencies from the operative system and the environment where they are deployed. These characteristics allow containers to be deployed faster than Virtual Machines. Moreover, they are suitable to package services or applications with high non-functional requirements, specially, those related to elasticity or fault tolerance.

The fact that several containers are running in the same physical –or virtual– machine can cause degradation in their performance. The simplest scheme to model how containers compete for a resource is when that resource is allocated by one container and it is not released until the container has finished a certain task –it is a conservative resource–. This scheme has been studied with Resource Allocation Systems (RAS) and Petri Nets [116] in different domains. In this context, the degradation in the QoS metrics is caused by the waiting time wasted while waiting for a resource to be available, or by the existence of deadlocks. Moreover, the RAS paradigm represents high level resources, for example, computing nodes used by a scheduler. However, the resources involved in the execution of a container in a node are more complex. For instance, the allocation of the CPU resource by a container is done for a small fraction of time. If there are not other containers to be executed in that machine, the scheduler in the machine will continuously allocate the CPU to that container.

The interference between containers is measured as the performance loss caused by the execution of a container at the same time as another one in the same host. Thus, it is the difference between the execution time when the container has available all the computational resources that it needs and the execution time when there are other containers using that resources at the same time. There are different ways to measure this metric. In Section 7.4, we propose to measure it as the relative value between the time which an application takes to reach certain points of its execution when it is co-scheduled with another application; and the same time measured when it is alone.

To understand how interference between containers works, we have identified the potential Sources of Interference (SoIs). Each of them is related to a physical resource (Figure 7.1):

- **CPU usage**: in most container management systems, if there is no contention in the use of the CPU, each container uses the CPU it needs; otherwise, there is a reservation system to share the CPU proportionally.
- **Cache Memory** and **Memory bandwidth**: the cache hierarchy in a node is not isolated between containers, so a container can be continuously failing when it accesses cache because another container is making an aggressive use of cache.
- **Network usage**: the network access is shared between all containers in a node; in

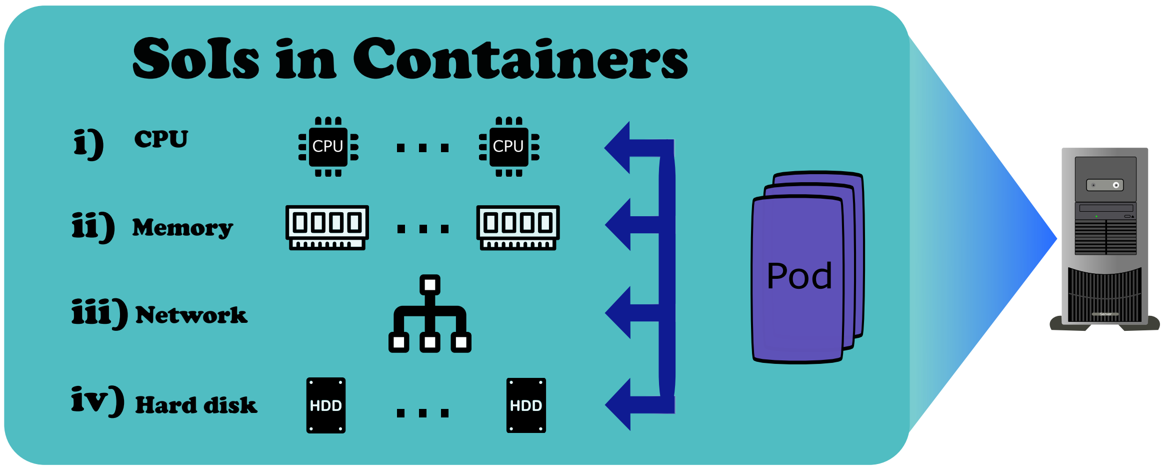


Figure 7.1: Sources of interference in a physical host.

addition, if there is no contention, a container can use the entire bandwidth.

- **I/O file system** access: like the network, the file system is shared between all containers; additionally, it could be a distributed file system and the network can be related to it.

In our work, we focus on CPU and cache memory usage, because they a have a heavy impact in the performance of containers and because cache memory is very difficult to isolate in these environments. However, the analysis of the Network and the I/O behaviour is straightforward following the proposed methodology. The main difference for these resources is that other low level events should be measured and the analysis should take into account that network usage requires to synchronise with another container.

The CPU of a container is easily isolated through a reservation mechanism. However, our studies show that this technique leads to an increase in the execution time of the container if it is not well-fitted [117]. Overall, this approximation avoids that unused resources in the machine are used by another container. An *ad hoc* solution is to overbook the available resources, but we consider that the solution should be given on the design level, not on the operational level. Moreover, other resources –namely cache memory and memory bandwidth– are difficult to isolate in hosts.

7.2 Background

Confirmatory Factor Analysis (CFA) [118] is a set of statistical techniques which identify how a set of observed –or measured– variables are affected by a set of factors –or latent variables– which are not possible to measure. CFA is a subset of Structural Equation Modelling (SEM) techniques [119]. In SEM, more sophisticated relationships between variables are allowed;

for instance, they let the modeller build hierarchical models to construct composite indices. We have chosen CFA because it allows the researcher to stablish an *a priori* hypothesis and the statistical model confirms or rejects the hypothesised model; unlike in other Factor analysis techniques, such as Exploratory Factor Analysis (EFA) [120]. In this regard, EFA can be used to determine the factor structure which explains the maximum amount of the variance of the variables. To extract meaningful factors, the solution should be rotated and an arbitrary cut-off value has to be used to determine which variable loads onto each factor. On the other hand, Principal Component Analysis (PCA) [121] only reduces the observed variables into a set of fewer factors which are difficult to interpret.

7.2.1 Confirmatory Factor Analysis

Formally, given a set of p observed variables, $\boldsymbol{X}$, and a set of m factors $\boldsymbol{F}$, we expect Equation 7.1 to be satisfied. μ_i is the intercept for x_i –it is the expected value when all factors are 0–, ϵ_i is the stochastic error and $m < p$.

$$x_i = \mu_i + \lambda_{i1}F_1 + \lambda_{i2}F_2 + ... + \lambda_{im}F_m + \epsilon_i \tag{7.1}$$

If all observed variables are affected by a single factor, we call the model a measurement model. Namely, $\forall i \in \{0, p\}, \exists k \in \{0, m\} \mid \lambda_{ik} \neq 0 \wedge \forall j \in \{0, m\}, j \neq k, \lambda_{i,j} = 0$.

We can express Equation 7.1 in matrix form, as it is done in Equation 7.2.

$$\begin{aligned}
&\boldsymbol{X} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{pmatrix} \boldsymbol{\Lambda} = \begin{pmatrix} \lambda_{11} & 0 & \cdots & 0 \\ \lambda_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_{p-1m} \\ 0 & 0 & \cdots & \lambda_{pm} \end{pmatrix} \\
&\boldsymbol{F} = \begin{pmatrix} F_1 \\ F_2 \\ \vdots \\ F_m \end{pmatrix} \boldsymbol{\mu} = \begin{pmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_p \end{pmatrix} \boldsymbol{\epsilon} = \begin{pmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_p \end{pmatrix} \\
&\boldsymbol{X} = \boldsymbol{\Lambda F} + \boldsymbol{\mu} + \boldsymbol{\epsilon}
\end{aligned} \tag{7.2}$$

If we suppose that $\boldsymbol{\Sigma} = \text{Cov}(\boldsymbol{X} - \boldsymbol{\mu})$, and we denote the covariance over the factors by $\boldsymbol{\Phi}$ and the covariance over the error by $\boldsymbol{\Psi}$, we can write Equation 7.2 in covariance form (Equation 7.3).

$$\begin{aligned}
\boldsymbol{\Sigma} &= \text{Cov}(\boldsymbol{X} - \boldsymbol{\mu}) = \text{Cov}(\boldsymbol{\Lambda F} + \boldsymbol{\epsilon}) \\
&= \boldsymbol{\Lambda}\text{Cov}(\boldsymbol{F})\boldsymbol{\Lambda}^t + \text{Cov}(\boldsymbol{\epsilon}) \\
&= \boldsymbol{\Lambda\Phi\Lambda}^t + \boldsymbol{\Psi}
\end{aligned} \tag{7.3}$$

In order to identify the model, we have to set the scale of the latent factors. Two methods can be used: i) setting the loading of the first observed variable for each factor to 1; or ii) setting the factor variance to 1, namely, $\forall i \in \{1, ..., m\}, \sigma(F_i) = 1$.

There are different methods in literature to estimate the parameters of the model [122] –$\boldsymbol{\Lambda}$ and $\boldsymbol{\Phi}$–: i) Maximum Likelihood (ML); ii) Robust ML (MLR); and iii) Weighted Least Squares (WLS). These methods are available in the most used statistical frameworks – e.g. R, STATA or MPLUS–. In this work we use the default method –Maximum Likelihood– in the lavaan package [123] on R statistical software[1].

7.2.2 Factor Scores

There are several methods to compute the value –or factor score– for the latent variables [124]. In this work we are going to use the regression approach, also known as Thurstone or Thompson scores.

Once the $\boldsymbol{\Lambda}$ and $\boldsymbol{\Phi}$ values are estimated, we can compute the Factor scores as a regression problem (Equation 7.4). $\boldsymbol{B}$ is the matrix with the coefficient values from the regression.

$$\boldsymbol{F} = \boldsymbol{B}\boldsymbol{X} \tag{7.4}$$

In the general formulation of CFA problem (Equation 7.1), we can include the n sampled values for the p observed variables (Equations 7.5 and 7.6).

$$\begin{pmatrix} x_{11}-\mu_1 & \cdots & x_{1n}-\mu_1 \\ x_{21}-\mu_2 & \cdots & x_{2n}-\mu_2 \\ \vdots & \cdots & \vdots \\ x_{p1}-\mu_p & \cdots & x_{pn}-\mu_p \end{pmatrix} = \begin{pmatrix} \lambda_{11} & 0 & \cdots & 0 \\ \lambda_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_{p-1m} \\ 0 & 0 & \cdots & \lambda_{pm} \end{pmatrix} \times \begin{pmatrix} f_{11} & \cdots & f_{1n} \\ f_{21} & \cdots & f_{2n} \\ \vdots & \cdots & \vdots \\ f_{m1} & \cdots & f_{mn} \end{pmatrix} \tag{7.5}$$

$$\hat{\boldsymbol{X}}_{p\times n} = \boldsymbol{\Lambda}_{p\times m}\hat{\boldsymbol{F}}_{m\times n} \tag{7.6}$$

We can isolate $\boldsymbol{F}$ in Equation 7.6 to compute $\boldsymbol{B}$ as follows:

$$\hat{\boldsymbol{X}} = \boldsymbol{\Lambda}\hat{\boldsymbol{F}} \implies \hat{\boldsymbol{F}} = (\boldsymbol{\Lambda}^t\boldsymbol{\Lambda})^{-1}\boldsymbol{\Lambda}^t\hat{\boldsymbol{X}}$$

As we want to minimise the error $\boldsymbol{\epsilon}$, from Equation 7.3, we get:

$$\hat{\boldsymbol{\Sigma}} = \boldsymbol{\Lambda}\boldsymbol{\Phi}\boldsymbol{\Lambda}^t \implies (\boldsymbol{\Lambda}^t\boldsymbol{\Lambda})^{-1}\boldsymbol{\Lambda}^t = \boldsymbol{\Phi}\boldsymbol{\Lambda}^t\hat{\boldsymbol{\Sigma}}^{-1}$$

Combining the previous expressions, we get Equation 7.7, where $\boldsymbol{\Phi}$ is the covariance matrix among factors, $\hat{\boldsymbol{\Sigma}}$ is the covariance matrix among observed variables and $\hat{\boldsymbol{X}}$ is the

[1] https://www.r-project.org/

matrix with the observed values centred in 0. f_{ij} represents the factor score for F_j for the ith observation.

$$\begin{aligned} \hat{\boldsymbol{F}} &= \boldsymbol{\Phi}\boldsymbol{\Lambda}^t\hat{\boldsymbol{\Sigma}}^{-1}\hat{\boldsymbol{X}} \\ \boldsymbol{B} &= \boldsymbol{\Phi}\boldsymbol{\Lambda}^t\hat{\boldsymbol{\Sigma}}^{-1} \end{aligned} \tag{7.7}$$

7.3 Developing Interference Indices

We want to build a set of indices which model the intensity in the usage of different resources for an application over its execution time. How an application makes use of the physical resources –cache, CPU, RAM, etc.– in a machine has a deep impact on the interference caused by that application to others co-scheduled in the same machine. This property is measured indirectly by the built indices. These indices can be useful to understand the behaviour of applications and to reason about how they are going to interfere between them. For instance, a high use of some resources when the application is alone will cause a high interference when it is co-scheduled with another one.

To identify these indices, we proceeded as follows: i) we conduct several experiments to measure the variability of different performance metrics and applications (Subsection 7.3.1); ii) we extract several variables from the dataset which are highly correlated (Subsection 7.3.2); and iii) we carry out a CFA to summarise the previous variables into four indices (Subsection 7.3.3). Figure 7.2 depicts the process to characterise any application with the identified indices. This process is described in detail in the following subsections.

7.3.1 Building the Dataset

We executed different applications inside a container in a machine with 4GB of RAM and 4 Intel i5-4690 (3,500GHz) cores. The chosen applications belong to different scientific domains. All of them can be seen as a job which may be executed several times with different input parameters. These applications are:

- `Pov-ray` [125] is a ray tracing application which generates an image from a scene description. We used version 3.7. The application is multi-thread and it makes an intense usage of the CPU. The input parameters used are given by the default benchmark.

- `IOzone` [126] is a file system benchmark utility. the benchmark is executed as follows: *iozone -a -i 0 -i 1 -g 4M*. The parameters have been chosen to adjust the execution time of the task and to perform random accesses to the memory.

- `Stream` [115, 127] is a benchmark to test the memory bandwidth of an architecture. To adjust the execution time of the task, the parameters used are:

  -DSTREAM_ARRAY_SIZE=100000000 -DNTIMES=100.

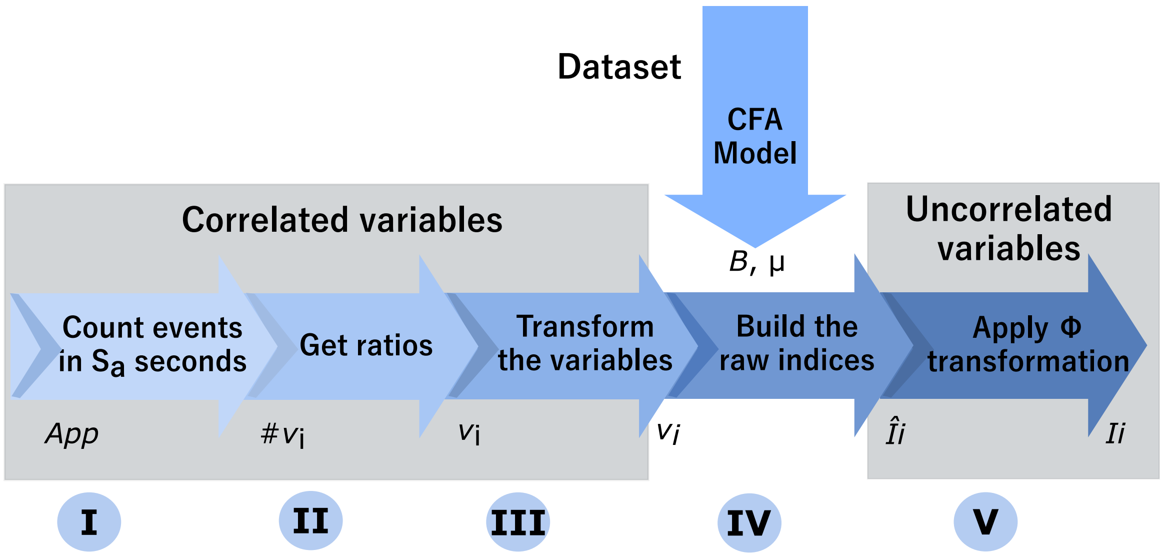


Figure 7.2: Process to get the interference indices from an application.

- `Metis` [128] is a set of graph partitioning tools. We use the `gpmetis` tool with one iteration and two partitions. The used benchmark is the *LiveJournal1* dataset[2], which has 4847571 nodes and 68993773 vertices.
- `Bzip2` and `pbzip2`. `Bzip2` is an open-source file compression program and `pbzip2` is the parallel version. The experiment compresses the *LiveJournal* dataset [2]. The size of the dataset is approximately 1.1 GB.
- `Montage` is a widely used scientific workflow which consists of the integration of several astronomical images into a single image mosaic. This task has high computational and data requirements. We use the `Montage` toolkit[3] with the provided *pleiades* example. We considered the entire *pleiades* workflow as a single task.
- `Blastn` and `Blastx` benchmarks[4] are related to the bioinformatic domain. They consist of several queries –100 for `blastn` and 24 for `blastx`– to be searched in a library or database to find biological sequences which resemble the original query.

We have chosen these applications because they have different patterns of resource usage and, consequently, the variance of the variables in the dataset is obtained from real scenarios. Additionally, we are going to use three of them –`Pov-ray`, `IOzone` and `Stream`– as benchmarking applications to estimate the interference between applications (Section 7.4).

[2]https://snap.stanford.edu/data/soc-LiveJournal1.html
[3]http://montage.ipac.caltech.edu/
[4]http://fiehnlab.ucdavis.edu/staff/kind/collector/benchmark/blast-benchmark

We consider that these three applications use certain resources intensely; thus, the dataset has values of situations with saturated resources.

The applications are containerised using Docker technology. We executed these containers with different configurations and we measured the variables in different points of their life-cycle. The objective is to get a dataset which captures the variations of the metrics to build meaningful indices. We assume that when the application is started, it has available all required data or information. Therefore, the application does not need to communicate with other applications to retrieve data or to send results. In other words, the only constraint to the progress of the application is given by the physical resources in the machine.

The dataset consists of 515 observations of 11 variables which measure low level resource usage of these applications executed inside a container. The metrics are measured with `perf` tool [129], which is available in several Linux distributions. This tool makes usage of the Performance counters of Linux to measure several hardware events. Given an application A and a sampling period of time s_A, we measure the number of occurrences –denoted by the # notation– of the following events:

- **Cycles** ($\#Cycles$). It is the number of processor cycles executed in s_A seconds. It is an indirect measure of the CPU usage in that period.
- **Cache-references** ($\#C_r$) and **Cache-misses** –$\#C_m$–. They indicate the total cache accesses and misses from the memory hierarchy in s_A seconds. They reflect the memory usage intensity and the pattern in which it is accessed.
- **LLC-loads** ($\#LLC_l$) and **LLC-load-misses** ($\#LLC_{lm}$). They indicate the number of Last Level Cache (LLC) accesses and misses for loading data in s_A.
- **LLC-stores** ($\#LLC_s$) and **LLC-store-misses** ($\#LLC_{sm}$). They indicate the Last Level Cache accesses and misses for storing data in s_A.
- **Branch-instructions** ($\#B_r$) and **Branch-instructions** ($\#B_m$). They measure the number of instructions which causes the execution of a different instruction sequence and the number of them which cause a miss in the cache hierarchy.
- **Page faults** ($\#fault$). It measures the amount of page faults when running the application. A page fault is raised when the application tries to access a virtual memory address which is not loaded in the physical memory. This event represents the lowest level of the memory which needs to access the hard drive.
- Executed **Instructions** ($\#Inst$). It counts the number of instructions executed in s_A seconds.

7.3.2 Variables from the Dataset

Instead of working with the absolute value of the events presented in the previous subsection, we are going to use their relative values as follows:

- Cache-references per instructions (v_1), branch-instructions per instructions (v_2), LLC-loads per instructions (v_3), and LLC-stores per instructions (v_4). These variables are obtained by dividing the amount of events by the number of instructions executed in that period (Equation 7.8).

$$\begin{aligned} v_1 = \frac{\#C_r}{\#Ins}, v_2 = \frac{\#B_r}{\#Ins}, \\ v_3 = \frac{\#LLC_l}{\#Ins}, v_4 = \frac{\#LLC_s}{\#Ins} \end{aligned} \tag{7.8}$$

- Cache-miss rate (v_5), branch-misses rate (v_6), LLC-load-miss rate (v_7) and LLC-store-miss rate (v_8). These variables are obtained by dividing the amount of events by the number of events of their corresponding non-miss event executed in that period (Equation 7.9).

$$\begin{aligned} v_5 = \frac{\#C_m}{\#C_r}, v_6 = \frac{\#B_m}{\#B_r}, \\ v_7 = \frac{\#LLC_{lm}}{\#LLC_l}, v_8 = \frac{\#LLC_{sm}}{\#LLC_s} \end{aligned} \tag{7.9}$$

- Standardised faults (v_9). We consider that it is not representative to reflect the number of fault events per instructions executed. Instead, we standardise the value to avoid scaling problems in further analysis (Equation 7.10). The mean and the standard deviation are calculated from the entire dataset, which is supposed to reflect the variability of the variable.

$$v_9 = \frac{\#faults - \mathrm{E}[\#faults]}{\mathrm{Var}[\#faults]} \tag{7.10}$$

- CPU usage (v_{10}). This variable indicates the intensity in CPU usage. Given the number of cycles executed in s_A seconds ($\#Cycles$), the number of cores of the machine ($\#Cores$) and the CPU speed measured in megahertz (S), Equation 7.11 shows how CPU usage is computed. The values belong to the $[0, 1]$ interval.

$$v_{10} = \frac{\#Cycles}{s_A \cdot S \cdot 10^6 \cdot \#Cores} \tag{7.11}$$

We have used the v_i notation to simplify the index expressions in the following sections.

7.3.3 Interference Indices

The high correlation between the previous variables avoids using them as raw values to describe an application and to do further analysis. We propose a theoretical model which combines the variables with the following four factors:

- I_1. **CPU usage** index. This index models the CPU usage of an application, measured as the number of cores used per time fraction.

$$I_1 = v_{10} \tag{7.12}$$

- I_2. **Memory page fault**. This index models the amount of page faults in the system. We have analysed them isolated because they have a deep impact on performance. It is interesting to note that the observed values resemble an exponential distribution. To improve the linearity of I_2, we transform the variable by taking natural logarithms (Equation 7.13).

$$\begin{aligned} I_2 &= \Phi_1(\hat{I}_2), \text{ where} \\ \hat{I}_2 &= \ln(v_9 + 1) \end{aligned} \tag{7.13}$$

- I_3. Intensity of **memory hierarchy usage** index. This index represents the number of accesses to the memory hierarchy. It is a combination of cache references –$\#C_r$, $\#LLC_l$– store references –$\#LLC_s$– and branch references –$\#B_r$– (Equation 7.14).

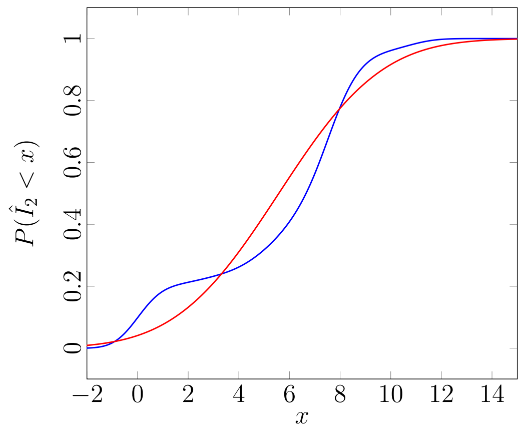
$$\begin{aligned} I_3 &= \Phi_2(\hat{I}_3), \text{ where} \\ \hat{I}_3 &= \sum_{i=1}^{i=4} b_i(\ln(v_i) - \mu_i) \end{aligned} \tag{7.14}$$

- I_4. **Aggressive intensity of memory hierarchy usage** index. This index measures the aggressiveness –in memory terms– of an application. It identifies applications that may have an intense memory usage and a low failure rate –a low value of I_4– from those with a high failure rate. The first one will have a lower impact into the performance of applications co-scheduled in the same machine.

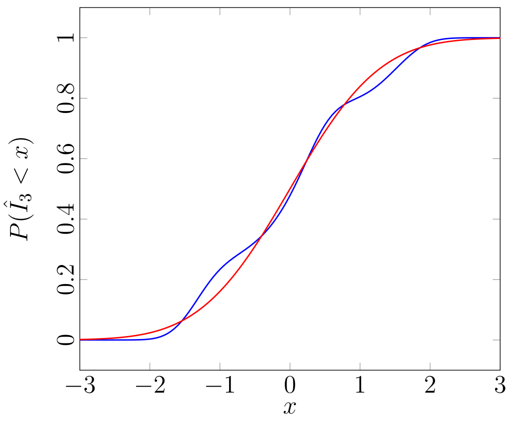
$$\begin{aligned} I_4 &= \Phi_3(\hat{I}_4), \text{ where} \\ \hat{I}_4 &= \sum_{i=5}^{i=8} b_i(\ln(v_i) - \mu_i) \end{aligned} \tag{7.15}$$

To avoid linearity problems in further analysis, we have taken logarithms for $\hat{I}_2$, $\hat{I}_3$ and $\hat{I}_4$ because they can be analysed as ratios [130]. Due to the CFA formulation (Equation 7.1), we have to centre the variable before multiplying by its load; consequently, we subtract its mean value. It is important to note that $\hat{I}_2$, $\hat{I}_3$ and $\hat{I}_4$ are values with a certain mean and a certain standard deviation and they might follow a non-normal probability distribution –theoretically, their values belong to the interval $(-\infty, \infty)$–. We can transform these indices to other ones –I_2 ,I_3 and I_4– trough their cumulative distribution function (cdf). The transformation function Φ_X is defined as shown in Equation 7.16.

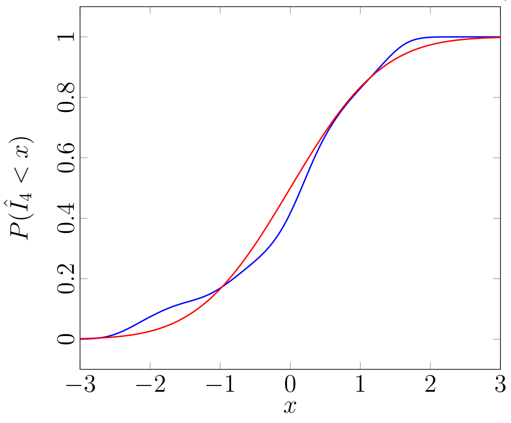
$$\begin{aligned} \Phi_X(x) : \quad & x \to P(X \le x) \\ & (-\infty, \infty) \to [0, 1] \end{aligned} \tag{7.16}$$



(a) Φ_2



(b) Φ_3



(c) Φ_4

Figure 7.3: Φ_2 (Subfigure (a)), Φ_3 (Subfigure (b)) and Φ_4 (Subfigure (c)) functions. Red line is the cumulative distribution function (cdf) of a normal distribution with the same mean and standard deviation.

We can build these functions –Φ_2, Φ_3 and Φ_4– empirically by using their histogram, as shown in Figure 7.3. We can see that the raw indices do not follow a normal distribution, which is denoted by the red line in Figure 7.3.

To compute the weight of each variable for I_3 and I_4, we performed a CFA analysis as explained in Section 7.2. We set the variance of the latent factors to 1 –$\sigma(\hat{I}_2) = \sigma(\hat{I}_3) = 1$– and the covariance among factors to 0 –$\text{Cov}(\hat{I}_2, \hat{I}_3) = 0$–, so the resulting factors are forced to be orthogonal.

The solution and further factor scores are calculated with the lavaan package [123] on R. The parameter estimation method used is ML and we used the regression method to compute the scores.

The resulting $\boldsymbol{\Lambda}$, $\boldsymbol{\Phi}$ and $\boldsymbol{\mu}$ matrices are:

$$\boldsymbol{\Lambda} = \begin{pmatrix} \lambda_{11} & 0 \\ \lambda_{21} & 0 \\ \lambda_{31} & 0 \\ \lambda_{41} & 0 \\ 0 & \lambda_{52} \\ 0 & \lambda_{62} \\ 0 & \lambda_{72} \\ 0 & \lambda_{82} \end{pmatrix} = \begin{pmatrix} 2.202 & 0 \\ 0.179 & 0 \\ 2.342 & 0 \\ 2.011 & 0 \\ 0 & 1.315 \\ 0 & -0.259 \\ 0 & 1.241 \\ 0 & 1.483 \end{pmatrix}$$

$$\boldsymbol{\Phi} = \begin{pmatrix} \sigma(F_1) & \mathrm{Cov}(F_2, F_1) \\ \mathrm{Cov}(F_1, F_2) & \sigma(F_2) \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\boldsymbol{\mu} = \begin{pmatrix} \mu_1 \\ \mu_2 \\ \mu_3 \\ \mu_4 \\ \mu_5 \\ \mu_6 \\ \mu_7 \\ \mu_8 \end{pmatrix} = \begin{pmatrix} -6.079 \\ -2.049 \\ -6.448 \\ -8.734 \\ -2.051 \\ -3.796 \\ -2.083 \\ -1.857 \end{pmatrix}$$

Then, we computed the $\boldsymbol{B}$ matrix using Equation 7.19:

$$\boldsymbol{B} = \begin{pmatrix} 1 & -0.06 & -0.48 & -0.04 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.40 & 0.05 & -0.46 & -0.14 \end{pmatrix}$$

We can see that the values corresponding to v_2, v_4 and v_6 are quite low. Hence, they might be removed from the analysis. The reason for this behaviour is the high correlation between variables and the fact that we are looking for factors that are not correlated. The variance of v_2 can be mostly explained by other variables. Some of the observed variables –e.g. v_3 and v_4– measure events which might be included in other ones –e.g. v_1–; so the scores of the factor remove the common measured elements and the correlation. Due to these situations and due to the data transformations taken, these coefficients should not be interpreted as they decrease the interference caused by using a certain resource. Since in a computational environment it is difficult to isolate these variables and to control them for a given application, an analysis about the elasticity of the variables in the indices should not be carried out.

7.3.4 Characterising Applications with Interference Indices

We want to model how the resource usage of an application changes over time. Resource usage is described by the four indices presented in the previous section. Given a sampling time s_A, to compute the value of I_1, I_2, I_3 and I_4 in that interval, we proceed as shown in Figure 7.2: i) we execute the containerised application alone to get the raw events; ii) we compute the ratios of the events; iii) we take logarithms in the $\hat{I}_2$ and $\hat{I}_3$ dependant variables; iv) we compute the factor scores for $\hat{I}_3$ and $\hat{I}_4$ through Equation 7.19; and v) we standardise and transform the variables to a $[0, 1]$ range, using Equation 7.16. Steps 2 to 5 are repeated for each set of measured events with a period of s_A seconds.

Figure 7.4 depicts the profile of `Pov-ray`, `IOzone` and `Stream` applications. These applications are used in the following sections as benchmarks to analyse the impact of application co-scheduling because they make a high usage of a certain resource. This situation is modelled by getting a high value on a certain index. For example, `Pov-ray` (Figure 7.4a) continuously uses the entire computational capacity of the machine, therefore I_1 is about one for the entire execution. On the other hand, `iozone` (Figure 7.4b) makes random accesses to the memory, hence why I_4 is about one. `Stream` application (Figure 7.4c) is a benchmark which tries to test the memory bandwidth, so the highest index is I_3.

The behaviour of the remaining applications is shown in Figure 7.5. We can see that `pbzip2` (Figure 7.5d), the parallel implementation of `bzip2`, makes a high usage of all resources in the system. When compared with the `bzip2` profile (Figure 7.5c), the usage of the extra CPU capacity leads to an increase on the faults in cache –I_4 index– while I_3 remains constant. Another interesting application is `montage` (Figure 7.5b), which consists of several tasks executed in a pipeline scheme. This situation causes that the memory usage indices –I_3 and I_4– feature some kind of patron in their distribution. Similarly, we can see that `blastn` application (Figure 7.5e) shows high values for I_2.

7.4 Interference Analytical Model

In this section, we propose a model to estimate the interference between containers and, consequently, the total execution time under contingency situations. The interference is caused by sharing physical resources which are not isolated at the container level. To describe an application and how it uses the physical resources, we have proposed an index-based representation in Section 7.3. We assume that, when two applications are scheduled in the same machine to be executed, if both make a high usage of the same resource, the degradation will be higher. Additionally, as the indices vary over execution time, the interference will not be constant. If execution time is equal to sampling time, we will have a single value for all indices an the interference will be modelled as a constant.

We propose a multi-linear regression model to estimate the interference between containers. Given an application A, we execute it with three benchmarking applications, which make a high usage of a certain resource to get reference interference values. These values will allow us to build the regression model to estimate the interference when application A is co-scheduled with another application whose resource profiles are known.

The proposed methodology (Figure 7.6) has the following steps: i) getting the interference profiles; ii) benchmarking the application; iii) defining the regression model; and iv) estimating the interference. Each phase is explained in the following subsections.

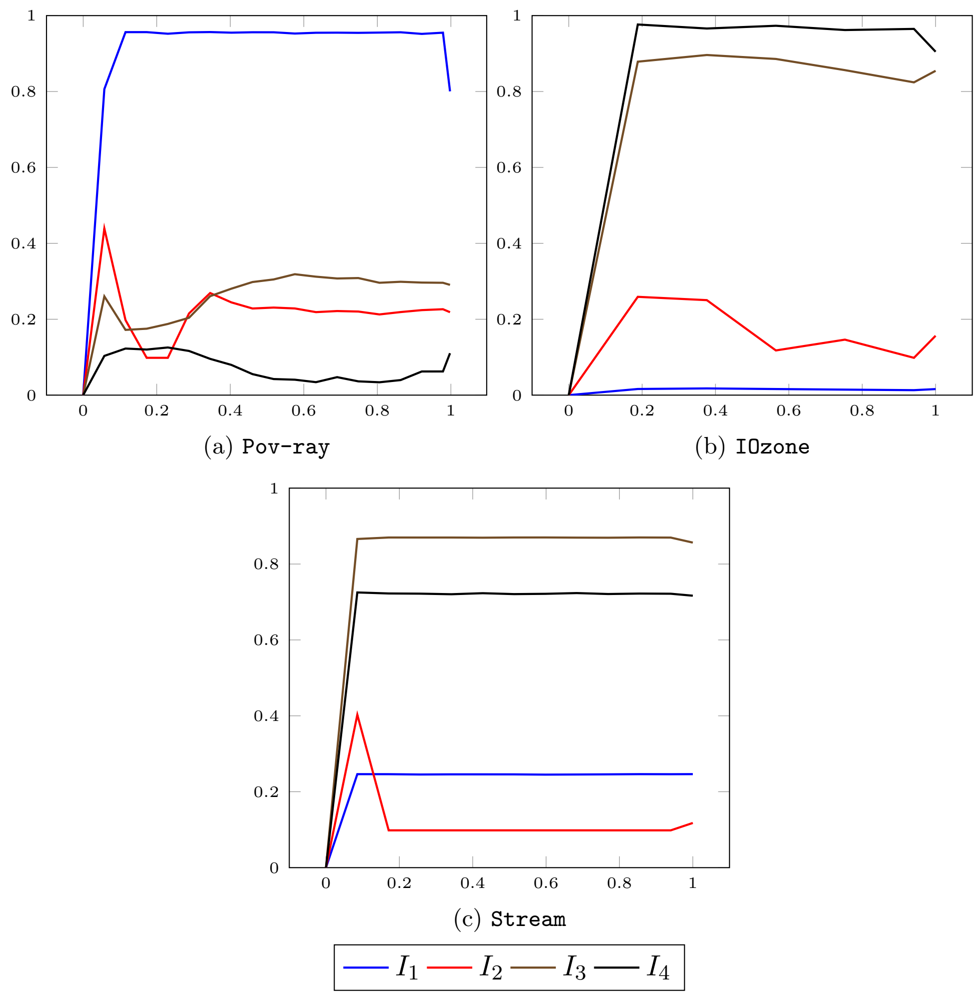


Figure 7.4: I_1, I_2, I_3 and I_4 values for benchmarking applications: (a) `Pov-ray`, (b) `IOzone` and (c) `Stream`. The execution time (x-axis) is normalised.

7.4.1 Preliminaries

We propose to model applications as a set of n sampled values for each interference index –I_1, I_2, I_3 and I_4–. Formally, an application A, is a tuple $\langle s_A, \boldsymbol{Y} \rangle$, where:

- s_a is a constant and it is called the sampling period of A.
- $\boldsymbol{Y}$ is a $n \times 4$ matrix. We denote y_{ij} an element in $\boldsymbol{Y}$. Each y_{ij} represents the sampled value for the j-th interference index at time $i \cdot s_A$.

Given the previous definition, we can define the profile function f_{Ai} for each index as an interpolation between the sampled points:

$$\forall i \in [1, n], \forall j \in [1, 4], f_{Aj}(i \cdot s_A) = y_{ij}$$

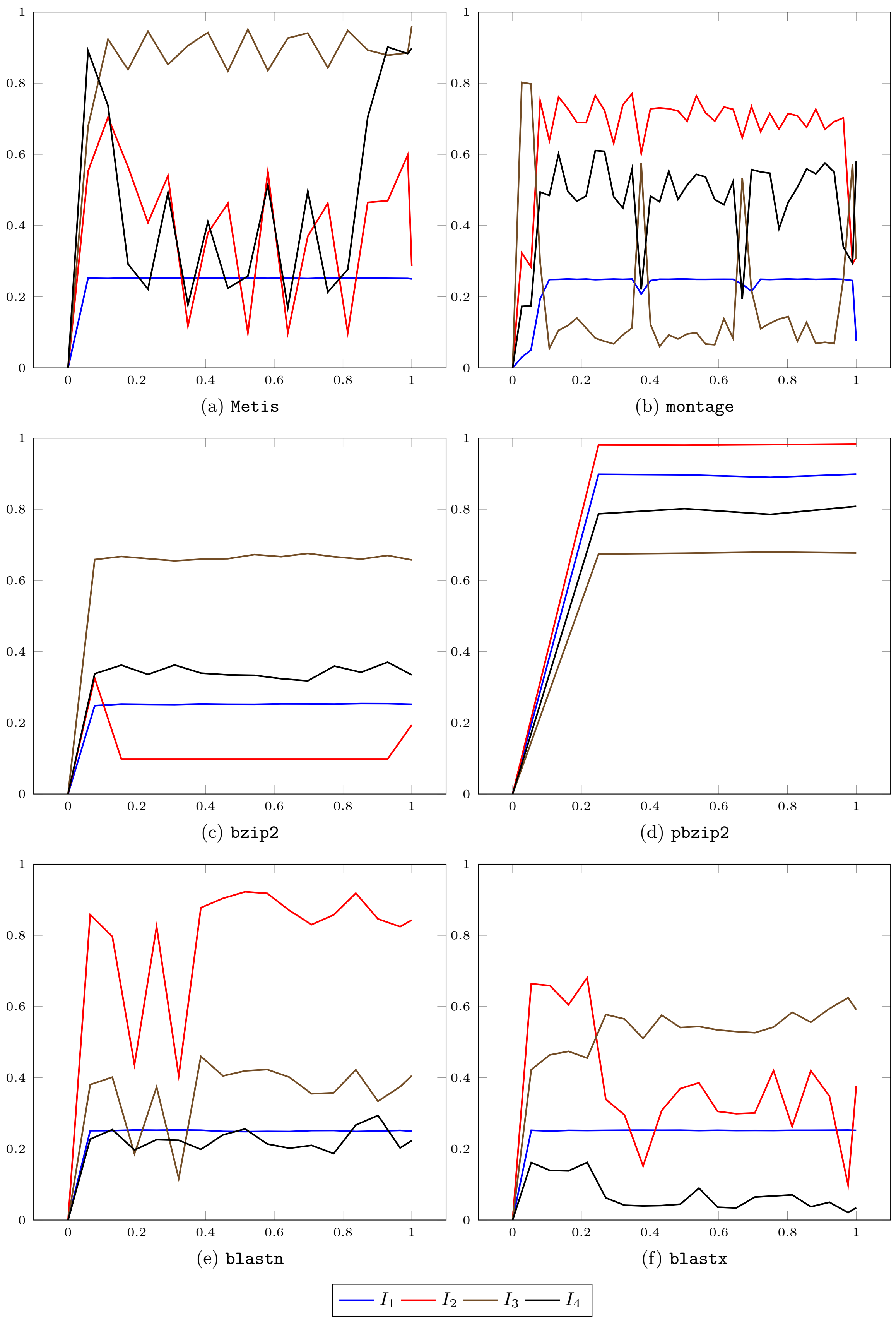


Figure 7.5: I_1, I_2, I_3 and I_4 values for some applications: (a) `metis`, (b) `montage`, (c) `bzip2`, (d) `bzip2`, (e) `blastn` and (f) `blastx`. The execution time (x-axis) is normalised.

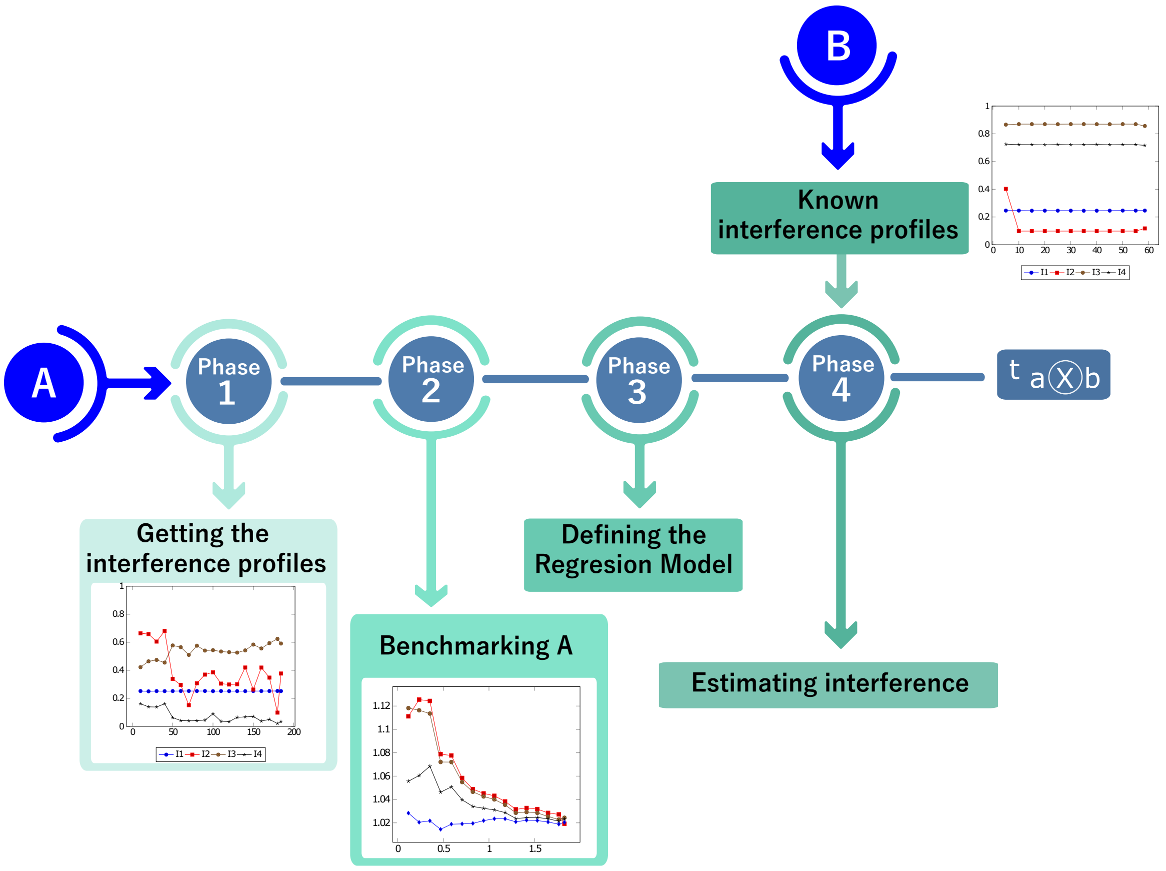

B
Known interference profiles
A
Phase 1
Phase 2
Phase 3
Phase 4
t a⊗b
Getting the interference profiles
Benchmarking A
Defining the Regresion Model
Estimating interference
I1 I2 I3 I4

Figure 7.6: Methodology to estimate the execution time of application A when it is co-scheduled with application B.

Without loss of generality, we are going to consider the linear interpolation function, which is defined as the concatenation of linear interpolants between each pair of (x_i, y_i) (Equation 7.17). Figure 7.5 depicts several examples of profile functions for different applications.

$$f_{Aj}(x) = y_{ij} + (x - is_A)\frac{y_{i+1,j} - y_{ij}}{s_A} \tag{7.17}$$

Given two applications A and B, we say that they are co-scheduled if they are executed in the same physical machine; therefore, they are going to share computational resources. We denote the co-scheduling operator as $\otimes$. The result of co-scheduling two applications can be interpreted as a compound application C. The profile function of C is a combination of the profile functions of A and B.

However, we are not interested in how the profiles are combined or how the resource profiles of a compound application C look like. Alternatively, we want to estimate the execution time of C given the applications A and B. In our model, A represents an incoming application to the system, and B can model the entire resource utilisation of applications executed in a certain machine. Application B might model an application as complex as possible.

For notation purposes, we denote by T_A the execution time of application A; and $T_{A \otimes B}$ the execution time of application A when it is co-scheduled with application B. If application B is one of the three benchmarks, we denote $T_{A \otimes B_i} = T_{AB_i}, i \in [1,3]$. In general, $T_{A \otimes B} \neq T_{B \otimes A}$.

7.4.2 Phase 1. Getting Interference Profiles

As we consider homogeneous clusters, when an application arrives in the system, it is executed alone in a machine to get the profile functions –one for each interference index–. The profile functions are built with the methodology presented in Section 7.3. Figure 7.5 shows several examples of how these profiles look like. Additionally, we get the total execution time when the application is executed with all resources available, denoted by T_A. The sample period of the profiles is given by s_A.

7.4.3 Phase 2. Benchmarking the Application

We execute the application on a machine where different benchmarks are running. First, we execute application A with application B_1 –`Pov-ray`– to obtain T_1; then, it is co-scheduled with application B_2 –`IOzone`– and we get T_2; finally, it is co-scheduled with B_3 –`Stream`– to get T_3.

Additionally, we split the execution time of application A in $\lceil T_A/s_A \rceil$ intervals. We measure the time that application A takes to execute the *i-th* interval when it is co-scheduled with benchmark B_j. This time is denoted by $\tau_{i,j},\ i \in [0, \lceil T_A/s_a \rceil],\ j \in [1,3]$. The interference $\delta_{i,j}$ can be computed by Equation 7.18. The interval bounds are measured by using the instruction observation of `perf` tool –$\#Inst$ variable–. Note that $\tau_{\lceil T_A/s_A \rceil, i} = T_{AB_i}$.

$$\begin{aligned} \forall i \in [1, \lceil T_A/s_A \rceil], j \in [1,3], \quad \delta_{i,j} &= \frac{\tau_{i,j} - \tau_{i-1,j}}{s_A} \\ \forall j \in [1,3], \quad \delta_{0,j} &= \frac{\tau_{0,j}}{s_A} \end{aligned} \tag{7.18}$$

7.4.4 Phase 3. Defining the Regression Model

We can model the interference of an application A when it is co-scheduled with an application B as a multi-linear function (Equation 7.19).

$$\begin{aligned} &\Delta(f_{A1}, ..., f_{A4}, f_{B1}, ..., f_{B4}) = \\ &y = \beta_0 + \sum_{i=1}^{4} (\beta_i f_{Ai} + \beta_{i+4} f_{Bi}) \end{aligned} \tag{7.19}$$

Where f_{A1},f_{A2},f_{A3} and f_{A4} are values from the profile functions of A (Equation 7.17) and f_{B1},f_{B2},f_{B3} and f_{B4} are values from the profile functions of B at the same time.

With the interference reference values –δ_{ij} in the previous step– and the profile functions of A, B_1, B_2 and B_3, we can estimate the parameters of the model. To improve the accuracy of the model, we propose to build a model per application. To accomplish this, we have to execute the application at the same time as each benchmark before building the model.

As the four indices range belongs to $[0, 1]$ and the interference value is positive, we can include the following restriction to the regression problem: $\beta_i > 0, \forall i \in [0, 9]$. To estimate the regression model under these assumptions we use the Non-Negative Least Squares (NNLS) approach [131]. We used the nnls package for R. The usage of NNLS algorithm leads to non-normal distributed residuals, so the classical interpretation of them should be avoided [132].

7.4.5 Phase 4. Estimating the Interference

Once the interference is modelled as a linear function, we can estimate the interference of application A when co-scheduled with any other application B whose resource utilisation functions are known.

The execution time of application A has been split in $\lceil T_A/s_A \rceil$ intervals in the second phase. At the instant $i \cdot s_A$ –the upper bound of the interval i– the interference is denoted by δ_i, and we use the hat notation for the estimation $\hat{\delta}_i$. Without loss of generality, the values inside the interval are calculated by using a linear interpolation function. Through Equation 7.19, we can compute the interference for each interval by using the profile function. Given a model Δ –or equivalently the β coefficients in the regression model–, the interference for the interval i can be estimated with Equation 7.20.

$$\begin{aligned}\hat{\delta}_i &= \Delta(f_{A1}(i \cdot s_A), ..., f_{A4}(i \cdot s_A), f_{B1}(i \cdot s_A), ..., f_{B4}(i \cdot s_A)) = \\ &= \beta_0 + \beta_1 f_{A1}(i \cdot s_A) + ... + \beta_4 f_{A4}(i \cdot s_A) + \\ &\quad + \beta_5 f_{B_1}(i \cdot s_A) + ... + \beta_8 f_{B_4}(i \cdot s_A)\end{aligned} \tag{7.20}$$

As the indices are functions which depend on the execution time of the application, namely their domain is $[0, T_A]$; we can rewrite Equation 7.20 as seen in Equation 7.21. To simplify the process, we compute the values of the function at the upper bound of each interval.

$$\begin{aligned}\Delta(t) &= \beta_0 + \beta_1 f_{A_1}(t) + ... + \beta_4 f_{A_4}(t) + \\ &\quad + \beta_5 f_{B_1}(t) + ... + \beta_8 f_{B_4}(t)\end{aligned} \tag{7.21}$$

The execution time of application A co-scheduled with application B can be computed as the integral of function $\Delta(t)$ (Equation 7.22). If the values used to compute the integral are the estimated ones, we obtain the estimated execution time, and if the values correspond to the measured one, we obtain the real execution time. In Equation 7.22, we have considered that the interference is constant in each i interval due to sampling reasons. Note that if we do not take any intermediary sample point –$T_A = s_A$–, we consider the interference as a constant and the overall estimation error might be higher.

$$T_{A\otimes B} = \int_0^{T_A} \Delta(t)dt \approx \sum_{i=1}^{n} \hat{\delta}_i s_A \tag{7.22}$$

We can use this estimation to develop a scheduler for a container management system. When an application arrives, the scheduler builds its functions and it chooses the best machine to deploy the application. When new applications arrive, the scheduler knows the profile of the application scheduled in each machine and it can estimate the machine which leads to the lowest interference and, consequently, to the smallest execution time.

7.5 Experimental Evaluation

In this section we present the experiments carried out to test our methodology. First, we present the results for the benchmarking phase and then, the results for the interference estimation.

7.5.1 Benchmarking Phase

Figure 7.7 depicts the interference of different applications when they are co-scheduled with each of the three benchmarking applications. The x-axis shows the execution time of the application normalised to one and the y-axis represents the interference as computed in Equation 7.18. The y-axis scale for `bzip2`, `blastn` and `blastx` is different in order to see the interference oscillations properly. We can see that `Metis` has a similar behaviour as `bzip2` because both of them are applications which make a high usage of the CPU with a single thread. In both cases the highest interference is achieved when they are co-scheduled with stream, which makes an aggressive usage of cache memory.

On the other hand, `pbzip2` is highly affected by `Pov-ray`, because both applications are sharing the entire CPU resources, and by `Stream`. `Montage` application (Figure 7.7b) depicts several well-differentiated phases. Each of them corresponds to a task of the montage workflow. `Blastn` and `blastx` applications are quite tolerant or resilient to the benchmark application interference. Their degradation is about 1.1 for the former and about 1.2 for the latter at most.

Table 7.1 shows the overall degradation for the six applications. It can be interpreted as the mean value of the interference shown in Figure 7.7. As expected, parallel applications –such as `pbzip2`– suffer a higher degradation when they are co-scheduled with B_1 than single-core applications. The interference caused on the latter by this benchmark is about 5%-7%.

7.5.2 Exploiting the Model

We have measured the interference suffered by applications when they are co-scheduled with other ones. Results are shown in Figure 7.8. The red line in the results is the estimated

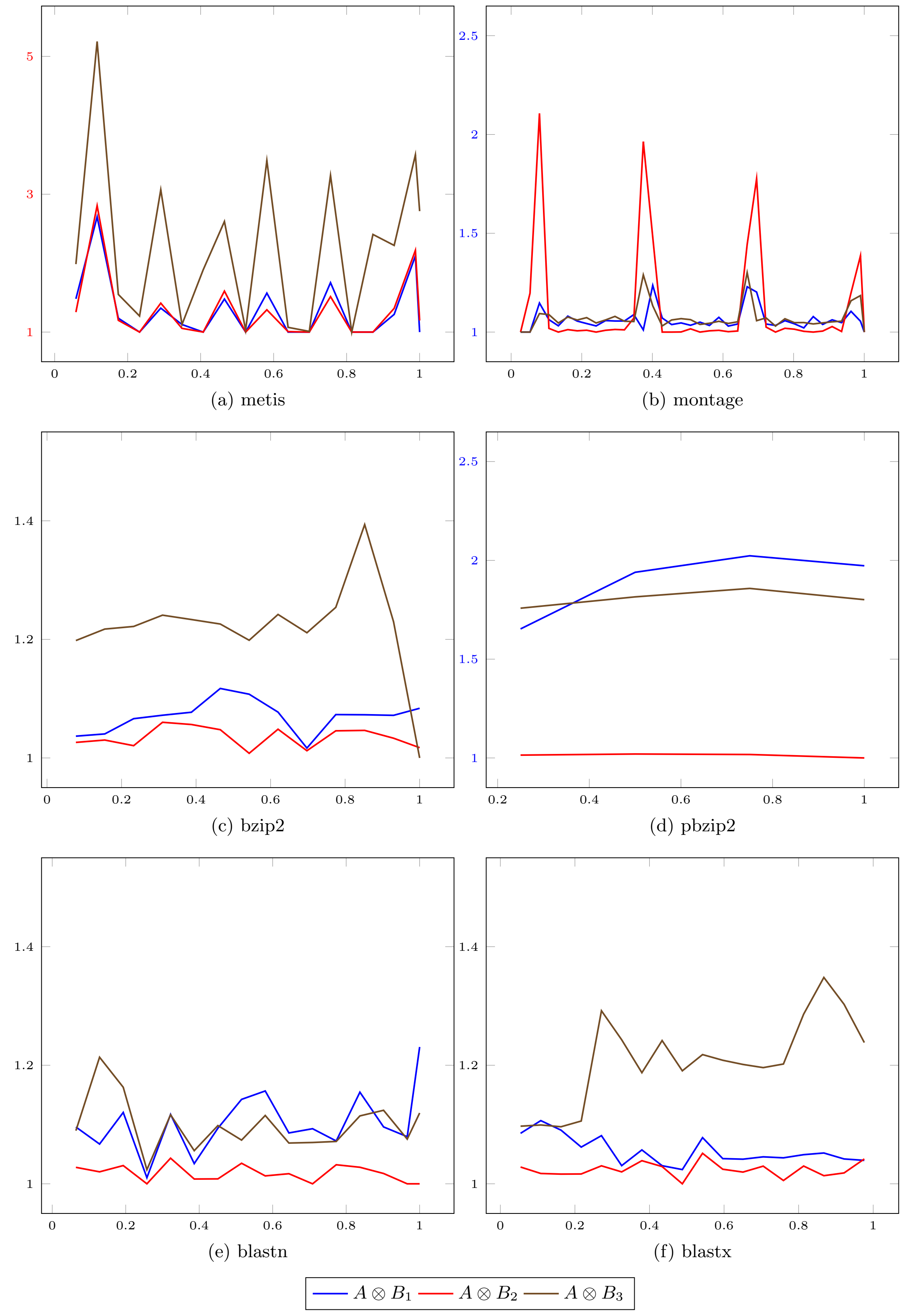


Figure 7.7: Interference values versus normalised execution time for some applications (A) –(a) `Metis`, (b) `Montage`, (c) `bzip2`, (d) `pbzip2`, (e) `blastn`, and (f) `blastx`– when they are co-scheduled with the benchmarking applications –B_1 is `Pov-ray`, B_2 is `IOzone` and B_3 is `Stream`.

App.	$T_A(s)$	$T_{AB_1}(s)$	T_{AB_1}/T_A	$T_{AB_2}(s)$	T_{AB_2}/T_A
metis	86.01	114.37	1.33	114.22	1.33
montage	374.08	397.63	1.06	421.73	1.13
bzip2	64.52	69.05	1.07	66.77	1.03
pbzip2	20.0	37.94	1.9	20.26	1.01
blastn	155.25	171.26	1.1	157.98	1.02
blastx	180.0	190.02	1.06	184.33	1.02

App.	$T_A(s)$	$T_{AB_3}(s)$	T_{AB_3}/T_A
metis	86.01	193.67	2.25
montage	374.08	401.41	1.07
bzip2	64.52	78.75	1.22
pbzip2	20.0	36.16	1.81
blastn	155.25	170.71	1.1
blastx	180.0	217.55	1.21

Table 7.1: Overall interference for applications co-scheduled with the benchmarks.

value with our methodology. Table 7.2 depicts the Mean Error (ME), the Mean Squared Error (MSE) and the accuracy of estimations (Acc). The first ones have been calculated using Equations 7.23 and 7.24. Although the regression methodology tries to minimise the absolute error, it could be interesting to report the Mean Relative Error (MRE) to analyse the accuracy (Acc) of the model (Equation 7.25).

$$ME = \frac{1}{n}\sum_{i=1}^{n}(\delta_i - \hat{\delta}_i) \tag{7.23}$$

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\delta_i - \hat{\delta}_i)^2 \tag{7.24}$$

$$Acc = 1 - MRE = 1 - \frac{1}{n}\sum_{i=1}^{n}\frac{|\delta_i - \hat{\delta}_i|}{\delta_i} \tag{7.25}$$

Once we have estimated the interference for each interval, we compute the execution time (Equation 7.22). Table 7.3 shows the values. As the estimated execution time is computed as the sum of all the estimated interference, the residual error ϵ in the estimation depends

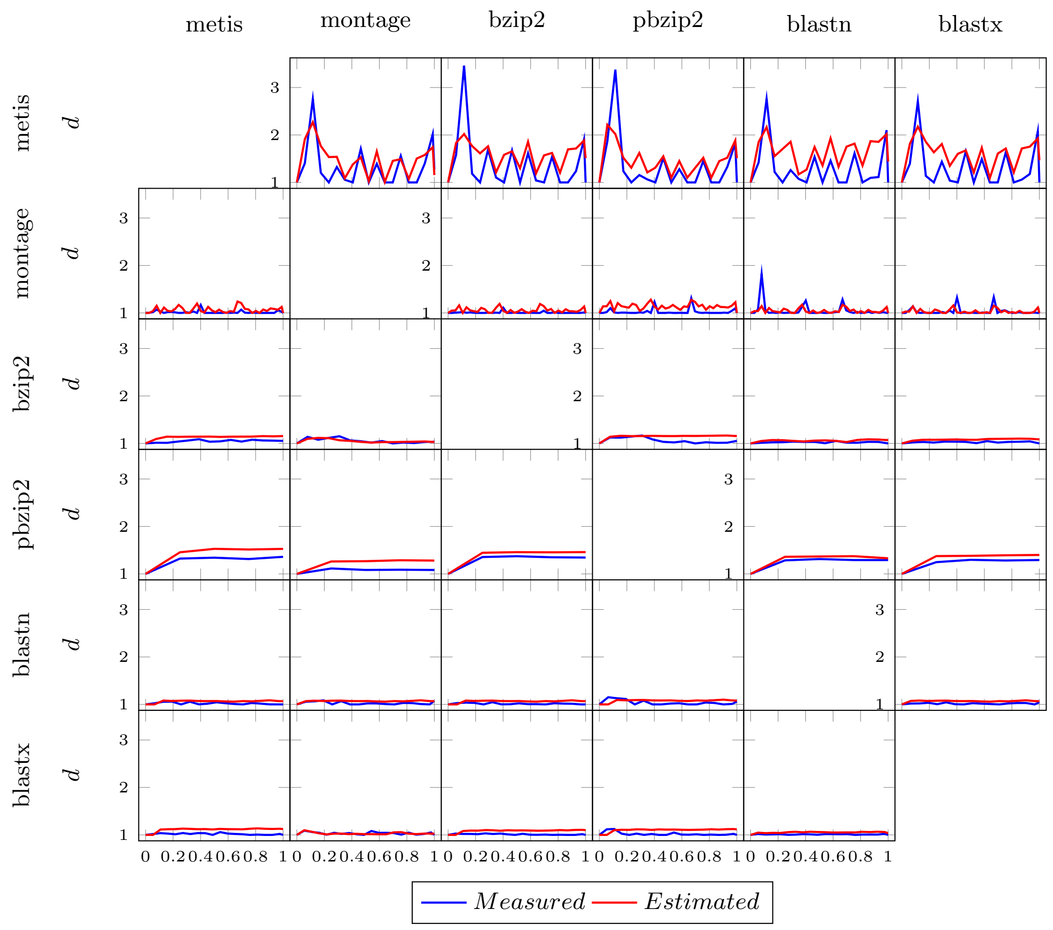


Figure 7.8: Interference values versus normalised execution time for several co-scheduled applications ($A \otimes B$). The blue line shows the measured values and the red line shows the estimated values with the proposed methodology.

on the ME and the sampling parameters, namely, the number of sampling points n and the sampling period s_A. The expression is given by Equation 7.26.

$$\begin{aligned}\epsilon = T_A - \hat{T}_A = \sum_{i=1}^{n} \delta_i s_A - \sum_{i=1}^{n} \hat{\delta}_i s_A = \\ = \sum_{i=1}^{n} (\delta_i s_A - \hat{\delta}_i s_A) = n s_A \sum_{i=1}^{n} (\delta_i - \hat{\delta}_i) = \\ = n^2 \cdot s_A \cdot ME \end{aligned} \tag{7.26}$$

		metis	**montage**	**bzip2**	**pbzip2**	**blastn**	**blastx**	**Total**
metis	ME	-	-0.14	-0.21	-0.14	-0.34	-0.29	-0.22
	MSE	-	0.1	0.25	0.16	0.22	0.18	0.18
	Acc	-	0.8	0.71	0.79	0.65	0.69	0.73
montage	ME	-0.05	-	-0.05	-0.11	0.0	-0.0	-0.04
	MSE	0.01	-	0.01	0.02	0.02	0.01	0.01
	Acc	0.94	-	0.94	0.88	0.95	0.97	0.94
bzip2	ME	-0.08	0.0	-	-0.09	-0.03	-0.05	-0.05
	MSE	0.01	0.0	-	0.01	0.0	0.0	0.0
	Acc	0.92	0.98	-	0.92	0.97	0.95	0.95
pbzip2	ME	-0.14	-0.15	-0.08		-0.05	-0.09	-0.1
	MSE	0.02	0.03	0.01	-	0.0	0.01	0.01
	Acc	0.9	0.87	0.94	-	0.96	0.93	0.92
blastn	ME	-0.04	-0.04	-0.05	-0.04		-0.05	-0.04
	MSE	0.0	0.0	0.0	0.01	-	0.0	0.0
	Acc	0.95	0.96	0.95	0.94	-	0.95	0.95
blastx	ME	-0.09	0.0	-0.07	-0.08	-0.04		-0.06
	MSE	0.01	0.0	0.01	0.01	0.0	-	0.01
	Acc	0.91	0.98	0.92	0.91	0.96	-	0.94
Total	ME	-0.08	-0.07	-0.09	-0.09	-0.09	-0.1	-0.09
	MSE	0.01	0.03	0.06	0.04	0.05	0.04	0.04
	Acc	0.92	0.92	0.89	0.89	0.9	0.9	0.9

Table 7.2: Mean Error (ME), Mean Squared Error (MSE) and accuracy (Acc) values for the experiment of Figure 7.8.

7.6 Refining the Model

In this section, we analyse how the methodology can be refined attending to the experimental results. Firstly, we evaluate the impact of the sampling time in the accuracy of the estimated execution time. Secondly, we remove the benchmarking phase and we evaluate a single model for all arriving applications. Finally, we discuss how our methodology can be exploited by the scheduler of a container management system.

7.6.1 Sampling Time

In our model, we consider that the performance losses are not constant during the execution time of an application. The accuracy of these values depends on the sampling time and the interpolation function. In the experimental section, we have considered a sampling period of five seconds for all applications, namely $s_A = 5$ for all A.

		Metis	**Montage**	**bzip2**	**pbzip2**	**blastn**	**blastx**
metis	Mea.	-	115.36	121.13	116.49	114.30	114.96
	Est.	-	128.94	138.95	127.46	144.44	140.84
montage	Mea.	379.72	-	376.66	383.55	392.83	385.16
	Est.	399.71	-	396.07	427.39	391.83	386.90
bzip2	Mea.	67.79	68.27	-	68.73	66.08	66.31
	Est.	73.59	67.94	-	74.65	68.48	69.97
pbzip2	Mea.	26.70	21.85	27.12	-	25.91	25.60
	Est.	30.13	25.49	29.10	-	27.15	27.74
blastn	Mea.	158.62	159.55	157.62	161.54	-	158.04
	Est.	165.71	166.35	165.72	167.62	-	166.45
blastx	Mea.	188.16	190.94	186.69	188.60	186.33	-
	Est.	206.02	190.19	200.94	203.92	194.44	-

Table 7.3: Measured and Estimated execution times in seconds –Mea. and Est. rows– for the experiment of Figure 7.8.

If we consider the experiment of `metis` and `Montage` ($metis \otimes Montage$), we can compute the accuracy of the estimated execution time given a certain sample period. For the sake of simplicity, we assumed that the sampling period is constant in all phases of the methodology; however, we can use a certain sampling period to compute the profiles –Scenario 1 in Table 7.4– and a different value to compute the overall interference and execution time – Scenario 2 in Table 7.4–. We can observe that the accuracy of the estimation increases as the value of sampling period decreases in Scenario 1. The reason for this behaviour is straightforward. As we increase the number of intervals, the size of those intervals decreases, so the error estimation decreases too. For Scenario 2, the behaviour is quite similar. We reduced the number of points to interpolate the interference profile of the applications and, consequently, the estimation is less fine grained.

The experimental results lead us to conclude that the sampling period should be adjusted depending on the granularity needed on the estimations and on the expected accuracy. Namely, if we have a scheduler that tries to fill the gaps of low expected interference values on the execution of long applications with smaller applications, the granularity can determine the size of these gaps and the sampling period.

7.6.2 Single Model

In our methodology, we build a model per incoming application to make the estimations. This model leads us to wait for the execution time of the application when it is co-scheduled with the three benchmarks. This is easily parallelisable, and the waiting time of the bench-

n	s_A	Measured	Scenario 1	Scenario 2
17	5	115.36	128.94	128.94
9	10	115.36	132.182	154.53
5	20	115.36	133.86	158.52
3	40	115.36	143.9	154.6
1	85	115.36	159.94	135.34

Table 7.4: Measured and Estimated execution times in seconds when `Metis` is co-scheduled with `Montage` for different sampling periods (s_A) in Phase 4 for Scenario 1 –s_A in Phases 1 and 3 is 5 seconds– and Scenario 2 –s_A is the same for all phases.

marking phase, T_{bench} is given by Equation 7.27.

$$T_{bench} = max_{i=1}^{i=3}\{T_{B_i}\} \tag{7.27}$$

This time can be negligible if we consider scenarios where a lot of similar applications arrive to the system to be processed, as it happens in a lambda-function processing architecture. However, we can analyse what happens when we build a single model by using the values of all experiments and when we use that model to make estimations about the execution time.

For example, we can consider a single model which includes the interference values of `metis`, `bzip2`, `pbzip2` and `blastn` applications when they are co-scheduled with the benchmarking applications. We excluded `Montage` and `blastx` applications to analyse the accuracy of the model when an application which is not included arrives. Results are shown in Table 7.5. Note that the accuracy value (Acc) is the mean value of the accuracy of all estimations for that experiment (Equation 7.25), not the accuracy of the total estimated time. The overall accuracy is about 0.78. This value includes the estimation of co-scheduling all applications with `Montage` and `blastx`, and the $Montage \otimes blastx$ experiment in which both applications are not in the model.

These results show that the proposed single model captures the variance of the variables quite fine and it can be used as an upper bound of the real value. Additionally, it seems that applications that are not included in the model do not have a worse behaviour than those which are included. These results can lead us to conclude that the benchmarking phase improves the overall accuracy of the model at the expense of waiting for the benchmarking phase to finish. Thus, the single model can be used if an estimation of the upper bound is needed and/or its accuracy is high enough.

7.6.3 Discussion

In the previous sections, we have focused on estimating the effects of the interference between containers. We have defined the co-scheduling operator $\otimes$ and we have calculated the execution time. These values are quite useful for the scheduling to determine which is the optimal host to deploy the container. Several scheduling techniques –e.g. the usage of priority queues– can be used to improve the performance of executed applications. However,

		Metis	**Montage**	**bzip2**	**pbzip2**	**blastn**	**blastx**	**Total**
metis	Mea.	-	115.36	121.13	116.49	114.3	114.96	-
	Est.	-	117.92	121.0	144.3	125.21	122.63	-
	Acc	-	0.8	0.74	0.59	0.73	0.74	0.72
montage	Mea.	379.72	-	376.66	383.55	392.83	385.16	-
	Est.	480.4	-	427.67	532.15	430.97	436.09	-
	Acc	0.74	-	0.87	0.62	0.87	0.85	0.79
bzip2	Mea.	67.79	68.27	-	68.73	66.08	66.	-
	Est.	87.74	76.34	-	95.18	79.39	79.48	-
	Acc	0.73	0.89	-	0.64	0.81	0.82	0.78
pbizp2	Mea.	26.7	21.85	27.12	-	25.91	25.6	-
	Est.	34.18	32.47	31.24	-	31.94	32.3	-
	Acc	0.78	0.61	0.88	-	0.81	0.79	0.77
blastn	Mea.	158.62	159.55	157.62	161.53	-	158.04	-
	Est.	200.16	177.91	179.18	219.39	-	184.61	-
	Acc	0.75	0.88	0.86	0.64	-	0.84	0.79
blastx	Mea.	188.16	190.95	186.69	188.6	186.34	-	-
	Est.	226.28	195.45	199.71	249.01	212.26	-	-
	Acc	0.81	0.95	0.93	0.68	0.87	-	0.85
Total	*Acc*	0.76	0.83	0.86	0.63	0.82	0.81	0.78

Table 7.5: Measured and Estimated execution times in seconds –Mea. and Est. rows– and Accuracy (*Acc*) with a single model which includes `metis`, `bzip2`, `pbzip2` and `blastn` applications.

with our methodology, the scheduler can also determine when is the best time to launch the application.

We can define the co-scheduling operator with a delay $\otimes^k$. $A \otimes^k B$ denotes that application A is co-scheduled with application B; however, A is delayed until B reaches the k interval. This approach can be useful to execute short applications when the long execution applications are in a low resource usage period. The scheduler can choose the machine and the delay which are optimal to minimise the interference caused on the applications in the cluster.

Additionally, the interference profiles can be useful to analyse the behaviour of applications. For example, they can be used to give penalties –or rewards– to those applications which interfere more –or less– with the remaining containers. Although some resources are difficult to isolate, our model can analyse the impact of setting up limits for those resources that can be isolated –for example, CPU, I/O disk or network usage–. In this regard, several container management systems such as Kubernetes allow setting bounds for the CPU used by a container.

7.7 Conclusions

Containers competing for resources inside a physical machine cause degradation in the performance of their executions. This degradation is caused by low level resources –e.g. cache hierarchy and disk access– which are difficult to measure and to analyse. The statistical CFA model allows us to build a set of indices which characterise how applications use the low level resources of a machine to analyse the interference caused between them. To apply the CFA model, we have built a dataset from real experiments to capture the variability of low-level events in certain situations. The proposed indices abstract the hardware events into comprehensible resources –CPU, memory access, memory aggressive access and memory page faults– and are expressed as a time series. Thus, we can analyse how they interfere over the entire execution of the application, instead of considering them as a constant attached to applications. In our work, we propose to use these indices to estimate the container interference; however, the indices have been designed as general purpose indices, and they can be used to help the scheduler to prioritise applications which cause more/less interference, to set interference bounds in machines, among other applications.

The proposed estimation model is based on a multiple linear regression model. It estimates how much an application is going to be degraded for being executed at the same time as another one, considering that both applications are deployed inside containers. This model allows the scheduler to estimate execution times to make better allocation decisions in a container cluster. In this regard, this methodology is a refinement of the characterisation proposed in the client-side scheduler presented in Chapter 6. The original methodology proposes to build a regression model for each incoming application, assuming that we have scenarios where a lot of similar instances of the same application are going to be deployed at the same time. The experiments suggest that a single model for all applications can be interpreted as an upper bound. However, as its accuracy is quite good –about 0.78–, it can be used alone for certain situations.

Chapter 8

Conclusions and Future Work

Having well polished the whole bow, he added a golden tip.

Homer, The Illiad

Classical approaches to the deployment of applications in distributed systems try to fill the conceptual gap between the functional and the operational level with the use of generic scheduling algorithms. These general-purpose mechanisms guarantee the transparency of the distributed paradigms –like the cloud– but, in general, they do not optimise the distributed resource management, as we have shown in Chapter 6. It is easy to see that a reduction of the resource efficiency –e.g. the number of unused resources increases– leads to an increase in the economic cost of those platforms.

Applications deployed in such environments can be seen as a white box or a black box by the system. The first ones might refer to applications that encapsulate complex systems and their distribution can be driven either by the model or by the data. The second ones refer to applications deployed in a system where the system has no additional information about the application model or implementation. The deployment of these applications is driven by the system, and to guarantee certain QoS parameters or cost bounds, we propose bringing the scheduling decisions near the users. In this regard, clients or application developers can use operational information to take decisions about the allocation of their applications. These decisions can be taken in accordance with the operational scheduler criteria.

In this PhD thesis, we have analysed each kind of application, using relevant use cases. First, we analyse an application which is heavily conditioned by the model, that is the distributed simulation of a High Level Petri Net. Then, we analyse streamed graph analytics from the point of view of the resource efficiency. The proposed model, based on the novel concept of summary graphs, reduces the number of processing instances by a seven factor. Finally, we analyse the operational level as a research subject by itself. In this regard, we propose several strategies to analyse black box applications with several information provided by the user. The operational infrastructure considered is based on container management systems, which in the last years has focused the attention of the research community due to their flexibility and lightweight deployment when compared with other infrastructure isolation abstractions, like VMs.

Furthermore, our experiments show that the reservation mechanism provided by those systems –in our work we have analysed Kubernetes– offers an inefficient resource management mechanism. Additionally, the new abstractions given by those platforms –for example, the concept of a pod managed by Kubernetes– increase the amount of deploying parameters. Consequently, platforms should provide several mechanisms or guides to help the client to deploy efficiently their applications. Thus, we bring near the client not only the scheduling or the allocation decision of their application, but also some operational information to help characterise its applications. Of course, this characterisation can be automated, as proposed in the last chapter of the thesis. Finally, we propose a characterisation of applications based on their resource usage profiles, which are described by timed indices created using the statistical CFA model.

8.1 Future work

In our work, we have analysed those QoS and cost parameters related to the performance mainly. We propose to analyse fault tolerance issues in the different domains. For example, for the streamed graph scenario, we can analyse how the inclusion of failures in the model has an impact on the performance metrics or in the total cost. In this regard, it could be interesting to introduce in the summary graph abstraction several mechanisms to deal with fault tolerance, such as the replication of summaries. We propose to take into account the dichotomy between the fault-tolerance requirements and the resource efficiency. Additionally, in our work we analyse the summary graph model for the partition problem. It is interesting to extend this approach by: i) using the summary graph model for other relevant graph domains; and ii) using the summary graph model to partition large scale systems as the one proposed in Chapter 3. That means that the set of LPs is summarised –only for partition purposes– into a smaller set of LPs.

Additionally, future work will involve the development of a platform as a service to support the modelling and distributed simulation of large complex systems. Involved tasks include: the full implementation of the elaboration process; the adaptation of current PN structural analysis tools for large TPNs; the refinement of objective functions to define balanced partitions attending to different criteria –size, lookahead values, etc.–; and the use of graph processing frameworks.

In the last chapters of the PhD thesis, we have considered an specific operational infrastructure, a container-based cluster. The model proposed in 5 uses several concepts from a specific software, Kubernetes. However, the transitions and places in the model can be refined to include specific policies of other container clusters. In this regard, we propose to build a model which analyses the deployment of applications in a container cluster as a whole system to make simulations. The aim of these simulations is helping to undertake decisions about the size or amount of resources of the cluster. Moreover, this kind of model can help to analyse different allocation policies.

On the other hand, we propose to apply the concept of the client-side scheduler to model-driven and data-driven applications. For instance, in the field of Discrete Event Simulation, the scheduler can use structural information about the specific simulation application to help the operational scheduling. Additionally, this concept can be extended to support more sophisticated distributed systems, such as edge computing paradigm or federation between clouds. Moreover, we propose to use our interference indices to develop better container isolation mechanisms which can reduce the cost of the infrastructure. These indices can also be used in other domains or directly as a scheduling mechanism. In this regard, they can be used as a criteria to distribute jobs or applications in cloud environments and in more specific abstractions, as osmotic computing, edge computing or fog computing.

Finally, in the last chapters, we have used reference applications to exemplify and conduct the experiments related with the operational level. As we have discussed, these applications are quite relevant in their respective domain, and they model a certain resource usage pattern. As future work, we propose to use more complex applications –as the simulation of Distributed Event Systems or graph analytics analysed in Chapters 3 and 4–, instead of isolated and self-contained applications. They might be useful to analyse more complex properties of the model such as the relationship between the different pieces of the model.

Appendix A

Notation

A.1 Notation for Chapter 4

Notation	Description
T	The entire Stream. Ordered sequence of $t_1, ..t_n$ items.
G	A Graph G, with V vertex set and E edge set.
v_i	Ith vertex in V set.
e_i	Ith edge in E set.
$deg(v)$	Degree of he vertex v. Number of adjacency edges of v.
n	Number of vertices, namely, $\|V\|$.
m	Number of edges, namely, $\|E\|$.
k	Number of partitions.
s	Number of partitioners.
Ψ	Set of summarised vertices.
Π_i	*I-th* summary of vertices in Ψ. $Pi_i \subset V$.
Φ	Set of edges between summary vertices set Ψ.
l	Number of vertices in a summarised vertex. $l = \|\Pi_1\|= \|\Pi_2\|= ... = \|\Pi_{\|\Psi\|}\|$
P	Partition solution. Set of partitions $\{S_1, ..., S_k\}$.
S_i	State of the *i-th* partition.
$\Gamma(v, S_i)$	Number of neighbours of v in partition S_i.
σ	Input rate. Number of incoming elements per time unit.

Notation	Description
Λ	Set of cutting edges.
λ	Fraction of cutting edges.
ρ	Normalised maximum load.

Table A.1: Used Notation in Chapter 4.

A.2 Notation for Chapter 6

Notation	Description
$\#Pods$	Number of pods in a deployment.
C	Number of containers in a deployment.
n	Number of machines in the cluster.
ρ	Inverse of the number of containers in a pod.
T_t	Total provisioning time.
T_d	Deployment time.
T_{down}	Time to download the container image.
T_c	Time to deploy a container.
T_e	Total execution time.
μ_1	Mean value for Scenario 1.
μ_2	Mean value for Scenario 2.
α	Ratio between μ_1 and μ_2.

Table A.2: Used Notation in Chapter 6.

A.3 Notation for Chapter 7

Notation	Description
A, C	Applications.
p	Number of factors.
m	Number of observed variables.
$\boldsymbol{X}$	Matrix with the observations.
$\boldsymbol{\Lambda}$	Matrix with the factor loads.
$\boldsymbol{\Phi}$	Covariance matrix of the factors.
$\boldsymbol{\Sigma}$	Covariance matrix of the observations.
$\boldsymbol{\mu}$	Matrix with the intercepts.
$\boldsymbol{B}$	Matrix with the factor scores.

Notation	Description
$\boldsymbol{\epsilon}$	Error matrix.
s_A	Sampling period of application A.
v_i	*I-th* variable from the dataset.
I_i	*I-th* adjusted interference index.
$\hat{I}_i$	*I-th* raw interference index.
Φ_X	Transformation function for variable X.
y_{ij}	*J-th* sampled value for the *i-th* index.
f_{Ai}	Profile function for the *i-th* index for application A.
B_i	*I-th* Benchmark.
$\otimes$	Co-scheduling operator.
$A \otimes^k B$	Application A is co-scheduled with application B delayed k time units. $A \otimes B = A \otimes^1 B$.
T_A	Execution time of application A.
n	Number of sampled points for a given application.
T_{AB_i}	Execution time of application A when it is co-scheduled with benchmark B_i. $T_{AB_i} = T_{A\otimes B_i}$.
$\tau_{i,j}$	Time to reach the i time interval when a given application is co-scheduled with benchmark B_j.
$\delta_{i,j}$	Measured interference at the i time interval when a given application is co-scheduled with benchmark B_j.
Δ	Regression model for a given application.
β_i	*I-th* β coefficient in the regression model.
$\hat{\delta}_i$	Estimated interference at the i time interval, given two co-scheduled applications.

Table A.3: Used Notation in Chapter 7.

Appendix B

Petri Nets and Object Nets

A Petri Net [13] is one of several formal models for the description and analysis of distributed, parallel or concurrent computing systems. It can be seen as a bipartite graph, where nodes can be of two types, either places or transitions; and arcs connect a transition to a place or viceversa. A place is often represented by a circle, whereas a transition by a rectangle. Besides, places can contain a –discrete– number of tokens –a token is typically represented as *[]*–. In order to model a system, all of these constituents can represent the dynamics of a system in a number of different ways. For instance, a transition can model a system action, and a place can represent a state, arcs can represent that each transition has a certain number of input and output places, modelling pre-conditions and postconditions of the system action. Tokens move from one pre-condition state to a post-condition state, when the involved transition is fired. In this way, it can be used to capture the evolution of system semantics.

In this thesis, we are making use of a particular type of Petri Nets called Object Nets [17] with reference semantic –sometimes these Petri Nets are called Reference Nets–. Reference Nets belong to the class of High-Level Petri Nets (HLPN) [108]. A HLPN is a Petri net whose tokens represent data structures or even objects. The pre-conditions of a HLPN can be labelled by expressions that identify states defined by the value of tokens. Besides, post-conditions can be labelled by expressions that define state changes by the modification of token values. In this way, HLPNs provide a more concise representation than ordinary Petri Nets. In essence, Object Nets extend High-Level Petri Nets with some characteristics that support the construction of hierarchical models, by allowing a token to be a net itself, creating hierarchies of nets. The nets forming part of such hierarchies can communicate by means of synchronous channels. Synchronous channels can be seen as a sophisticated way of message passing communication, but with a richer semantics based on the unification mechanism. A synchronous channel engages two transitions –typically from different nets– that, by means of the channel, fire –synchronize– simultaneously. The channel can also accommodate variables and it has two main roles:

- The *uplink* or callee role, at the subordinated instance net, which servers requests.
- The *downlink* or caller role, at the parent net, which makes use of the channel to both

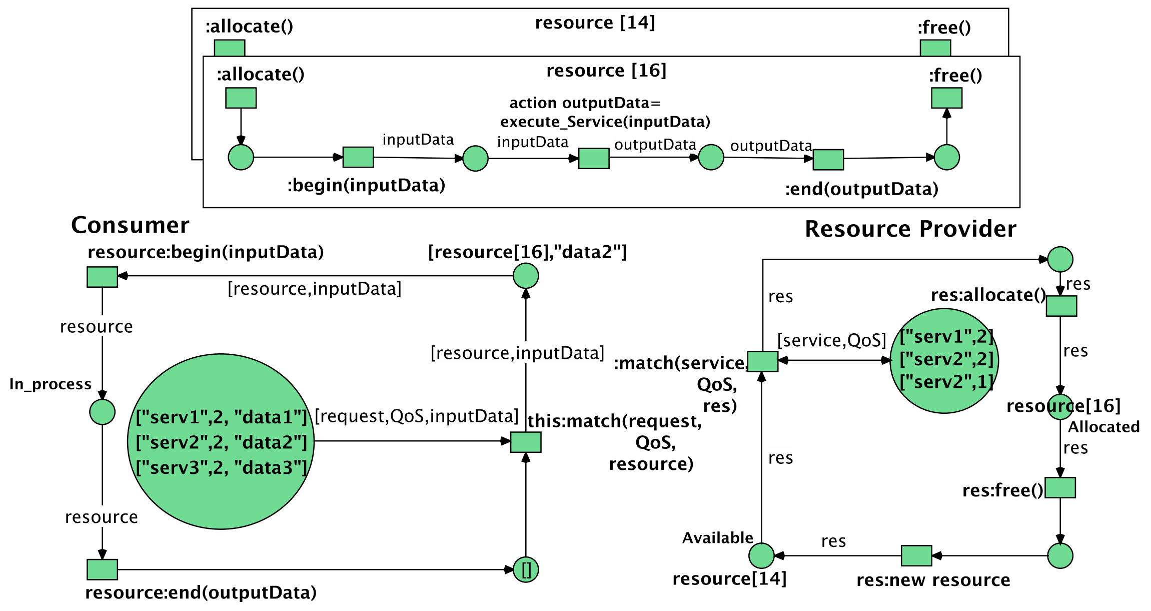


Figure B.1: A Reference Net model that represents a `Service Consumer` and a `Resource Provider`. References to `Resources` Net instances are managed by the consumer to provide data and collect results, and providers to manage resources.

synchronize and call the subordinate instance.

Channel communication and hierarchies provide the way to describe finely grained complex behaviours in the form of nested Petri Nets. The usefulness of net instances lies in the fact that they can represent behaviours that can be moved to different execution environments. The inscription language of Reference Nets have also been extended to include tuples, which can be used for representing a group of related values or variables in a single token. A net instance can be influenced by the net that holds it, called System Net, and such influence is accomplished by means of the synchronous channels mechanism.

The execution of a HLPN specification requires to find bindings, i.e. mapping of variables used in are expressions to specific values.

Object Nets with reference semantic can be interpreted by Renew [1] [108], a Java-based interpreter and graphical modelling tool. The way Renew binds variables to values is by unification, the same mechanism used by logic programming languages such as Prolog. In HLPNs, when a transition fires, then its expression is evaluated and tokens are moved according to the result. Furthermore, Reference Nets incorporate some characteristics from object-oriented languages: Reference Nets support the creation of net instances dynamically, and transitions also support the inclusion of inscriptions, including Java inscriptions. Therefore, Reference Nets also support the creation of Java objects, and Java method invocations inside a net. They also incorporate the assignment operator "=", and it can be

[1]http://www.renew.de

used to define –assign– the value of variables.

Reference Nets can hold two kinds of tokens: Valued tokens and tokens which correspond to a reference to a PN instance. By default, an arc will transport a black token, denoted by `[]`. In case an inscription is added to an arc, that inscription will be evaluated and the result will determine which kind of token is moved.

In order to illustrate the main concepts of Reference Nets, Figure B.1 depicts a Reference Net model that represents a `Resource Consumer` and a `Resource Provider` with allocating capacity of two resources. Services are required with a QoS and data to be processed when `Consumer` invokes the `match` channel. Communication happens when unification of variables is possible. In the state represented in the figure, the transition labelled with the downlink `this:match(request, Qos, inputData)` channel in the `Consumer` can synchronise with transitions labelled with uplinks `:match(service, QoS,Res)` in the `Resource Provider`. Figure B.1 shows one allocated resource providing service, and one available resource. There are two possible bindings with `data1` and `data2`, but there is only one available resource, and therefore transitions with downlink and uplink will be synchronously fired with one of the two possible bindings.

Appendix C

Kubernetes Background

Kubernetes is an open-source platform that abstracts and automates the deployment of containerised applications across a number of distributed computational nodes. Kubernetes manages these nodes and provide to applications a set of abstractions to support several non-functional requirement, such as elasticity and fault tolerance. In this appendix, we analyse the container approach used by Kubernetes, and then, we explain briefly its architecture and its scheduling principles.

C.1 Container Approach

VMs have been one of the first and most important cloud computational resources. A VM is a piece of software that emulates a hardware computing system and typically multiple VMs share the same hardware to be executed. The emulation is accomplished by a hypervisor. Hypervisors are responsible for dividing the hardware of the host physical machine, so that it can be used by the OS inside each VM. Therefore, applications that run inside a VM can accomplish calls to their own OS inside the VM, and then their virtual kernel executes instructions on the physical CPU of the host machine by means of the hypervisor.

One of the most important benefits of using VMs is the full isolation they achieve: VMs on the same host physical machine share the same hardware, but they are completely isolated. Nevertheless, VM utilization can sometimes be difficult to achieve, e.g. when the applications to be run do not consume all the resources of a VM. Developers can therefore can try to map multiple applications onto the same VM. In this case, applications would not be isolated. *Containers*, on the other hand, represent a way to solve that isolation problem by improving utilization. A container can be seen as a set of processes where an application is executed in isolation. Multiple containers typically coexist on the same host machine, and each container in it uses the resources that the application on it consumes. Nevertheless, the degree of isolation achieved by VMs is still higher than the one achieved by containers, but containers have much less overhead. The reason for it is that all containers deployed in the same host machine share the same OS kernel, and therefore virtualization is not required. Furthermore, while a VM needs to boot up first before an application can be executed on it,

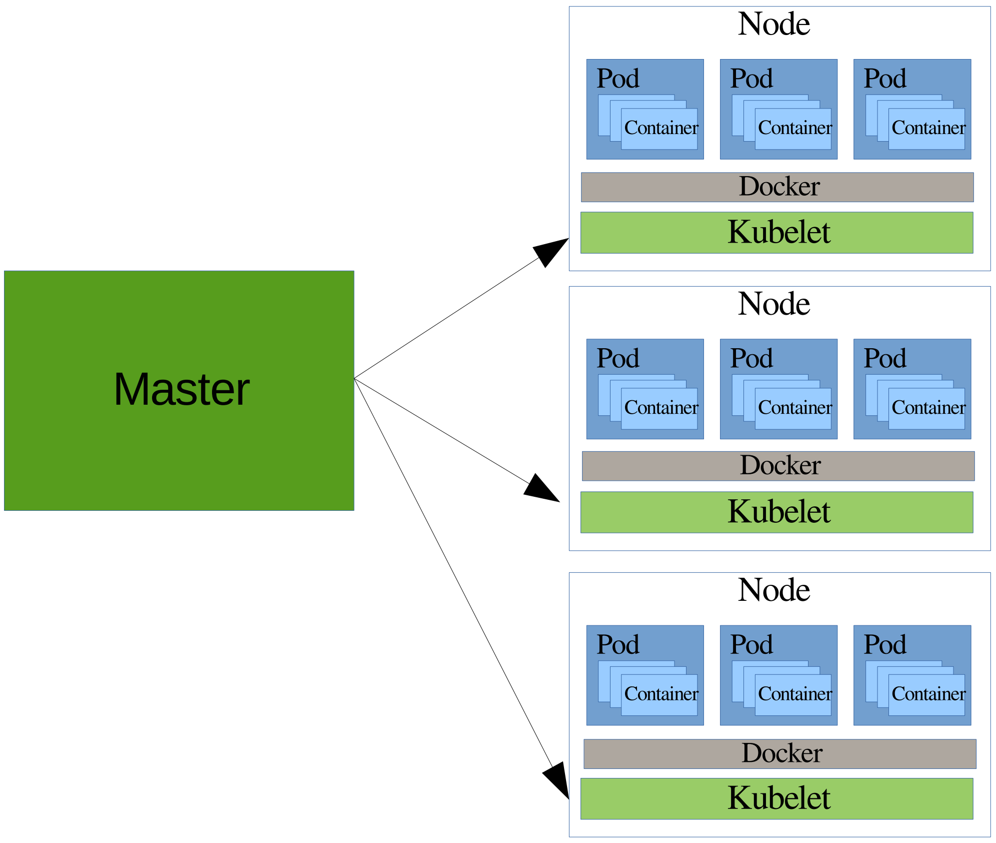


Figure C.1: Kubernetes architecture.

a container is a group of processes whose execution can be initiated almost immediately. The isolation of containers is obtained by two Linux mechanisms: Linux namespaces and Linux Control Groups, which isolate the view of the system and limit the amount of computational resources, respectively.

Although container technologies were developed many years ago, they became popular with the emergence of Docker. Docker containers became popular due to the fact that (i) they also help deploy library dependencies with the application and (ii) the high portability of containers on different platforms. A Docker container is a Docker image in execution. A Docker image contains an application and all the libraries required for its execution. In turn, Docker images are stored at the Docker registry, which also facilitates the availability of such images.

C.2 Kubernetes Architecture

Kubernetes is a system that allows developers to deploy and manage containerized applications. It is based on a master-slave architecture, with a particular emphasis on supporting a cluster of machines where containers are executed. Applications are submitted to the master and Kubernetes, in turn, deploys them automatically across the worker nodes in containers. The communication between Kubernetes master & slave nodes is realised through the *kubelet* service. This service must be executed on each machine in the Kubernetes cluster. The node which acts as master can also carry out the role of a slave during execution. The basic component architecture of Kubernetes is shown in Figure C.1. As Kubernetes often works with Docker containers, the *docker daemon* should be running on every machine in the cluster. In addition, Kubernetes makes use of the *etcd* project to have a key-value distributed storage system, in order to coordinate resources and to share configuration data of the cluster. The master node runs also an API server, implemented with a RESTful interface, which gives an entry point to the cluster. The API service is used as a proxy to expose the services which are executed inside the cluster to external applications/ users.

In order to run an application, it has to be wrapped on one or more container images, then push (submit) these images to a container service registry, and then post a description of the application in the form of an application descriptor to the API server. Then, Kubernetes will automatically retrieve the container images when the application is launched. Instead of deploying containers individually, Kubernetes deploys pods. A pod is an abstraction of a set of containers tightly coupled with some shared resources (the network interface and the storage system). Any OCI compliant container runtime engine could be used to execute containers in pods. With this abstraction, Kubernetes adds persistence to the deployment of single containers. It is important to note two aspects of a pod : (i) a pod is scheduled to execute on one machine, with all containers inside the pod being deployed on the same machine; (ii) a pod has a local IP address inside the cluster network, and all containers inside the pod share the same port space. Therefore, each pod has a unique IP address in a flat shared networking space that allows bidirectional IP communications with all other pods and physical computers in the cluster. The main implication of this is that two services which listen on the same port by default cannot be deployed inside a pod .

The Kubernetes scheduler allocates pods into nodes taking into account factors that have a significant impact on the availability, performance and capacity – e.g. the cluster topology, individual and collective resources, service quality requirements, hardware and software restrictions, policies, etc. The scheduler uses request and limits to filter the nodes that have enough resources to execute a pod , and from them, it chooses the best one. Pods can be categorise in three Quality of Service (QoS) classes: *Best effort*(lowest priority), *Burstable*, and *Guaranteed* (highest priority). The QoS classes is inferred from the request and limits manifests. A *Guaranteed* pod has all containers with limits equal to requests; a *Best effort* pod has not request or limit manifest for any container; and the rest of pods are *Burstable*. Once a pod is deployed in a node, if *pod request manifest* $<$ *pod request limits* resource requested are guaranteed by the scheduler, but it is possible to use resources beyond the request manifest if they are idle resources.

A pod can be replicated along several machines for scalability and fault tolerance purposes. When a service or a set of services are deployed over several machines, we can consider: (1) the *functional level* or application level involves exposing dependencies between the deployed services. Different services need to be coordinated in order to provide a high level functionality. An example of this kind of relationship is the deployment of a stream processing infrastructure (e.g. Apache Kafka, Storm, Zookeeper and HDFS for persistence) or the GuestBook example provided by Kubernetes, composed of a PHP frontend and a Redis master-slave system. (2) the *operational level* or deployment level involves mapping services to physical machines, VMs, pods or containers. It is platform dependant and must involve isolation between resources. Kubernetes primarily focuses on the operational/ deployment level. A pod implements a service, and some coordination between different pods is achieved through the key-value distributed store provided by *etcd*. Services running in others pods can be discovered through a DNS. This approach imposes some restrictions to Kubernetes. For instance, in the Guestbook example, Kubernetes' scheduler cannot ensure that the three pods are deployed rightly, because Kubernetes does not manage the application level.

Bibliography

Every great study is not only an end in itself, but also a means of creating and sustaining a lofty habit of mind.

Bertrand Russell

[1] V. Vemuri, *Modeling of complex systems: an introduction.* Academic Press, 2014.

[2] D. Bandyopadhyay and J. Sen, "Internet of things: Applications and challenges in technology and standardization," *Wireless Personal Communications*, vol. 58, no. 1, pp. 49–69, 2011.

[3] F. Xia, L. T. Yang, L. Wang, and A. Vinel, "Internet of things," *International Journal of Communication Systems*, vol. 25, no. 9, pp. 1101–1102, 2012.

[4] S. M. Amin and B. F. Wollenberg, "Toward a smart grid: power delivery for the 21st century," *IEEE power and energy magazine*, vol. 3, no. 5, pp. 34–41, 2005.

[5] M. Pipattanasomporn, H. Feroze, and S. Rahman, "Multi-agent systems in a distributed smart grid: Design and implementation," in *Power Systems Conference and Exposition, 2009. PSCE'09. IEEE/PES.* IEEE, 2009, pp. 1–8.

[6] V. C. Gungor, B. Lu, and G. P. Hancke, "Opportunities and challenges of wireless sensor networks in smart grid," *IEEE transactions on industrial electronics*, vol. 57, no. 10, pp. 3557–3564, 2010.

[7] J. A. P. Lopes, F. J. Soares, and P. M. R. Almeida, "Integration of electric vehicles in the electric power system," *Proceedings of the IEEE*, vol. 99, no. 1, pp. 168–183, 2011.

[8] N. Komninos, H. Schaffers, and M. Pallot, "Developing a policy roadmap for smart cities and the future internet," in *45th Hawaii International Conference on Systems Sciences*, 2011.

[9] R. Tolosana-Calasanz, J. Á. Bañares, and J.-M. Colom, "Model-driven development of data intensive applications over cloud resources," *Future Generation Computer Systems*, 2018.

[10] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: a state-of-the-art review," *IEEE Transactions on Cloud Computing*, 2017.

[11] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems.* ACM, 2002, pp. 1–16.

[12] M. Kiran, P. Murphy, I. Monga, J. Dugan, and S. S. Baveja, "Lambda architecture for cost-effective batch and speed big data processing," in *Big Data (Big Data), 2015 IEEE International Conference on.* IEEE, 2015, pp. 2785–2792.

[13] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, 1989.

[14] W. M. Zuberek, "Timed petri nets and preliminary performance evaluation," in *Proceedings of the 7th annual symposium on Computer Architecture.* ACM, 1980, pp. 88–96.

[15] M. K. Molloy, "Performance analysis using stochastic petri nets," *IEEE Transactions on computers*, no. 9, pp. 913–917, 1982.

[16] K. Jensen, *Coloured Petri nets: basic concepts, analysis methods and practical use.* Springer Science & Business Media, 2013, vol. 1.

[17] R. Valk, "Petri nets as token objects: An introduction to elementary object nets," in *Application and Theory of Petri Nets 1998, 19th International Conference, ICATPN '98, Lisbon, Portugal, June 22-26, 1998, Proceedings*, ser. LNCS, J. Desel and M. Silva, Eds., vol. 1420. Springer, 1998, pp. 1–25.

[18] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: densification laws, shrinking diameters and possible explanations," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining.* ACM, 2005, pp. 177–187.

[19] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, 1990.

[20] ——, *Parallel and distributed simulation systems.* Wiley New York, 2000, vol. 300.

[21] ——, "Parallel and distributed simulation," in *Proceedings of the 2015 Winter Simulation Conference.*, 2015, pp. 45–59.

[22] D. M. Rao, N. V. Thondugulam, R. Radhakrishnan, and P. A. Wilsey, "Unsynchronized parallel discrete event simulation," in *Proceedings of the 30th conference on Winter simulation.* IEEE Computer Society Press, 1998, pp. 1563–1570.

[23] R. M. Fujimoto, "Exploiting temporal uncertainty in parallel and distributed simulations," in *Proceedings of the thirteenth workshop on Parallel and distributed simulation.* IEEE Computer Society, 1999, pp. 46–53.

[24] S. C. Tay, Y. M. Teo, and S. T. Kong, "Speculative parallel simulation with an adaptive throttle scheme," in *ACM SIGSIM Simulation Digest*, vol. 27, no. 1. IEEE Computer Society, 1997, pp. 116–123.

[25] M. Marin, V. Gil-Costa, C. Bonacic, and R. Solar, "Approximate parallel simulation of web search engines," in *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation.* ACM, 2013, pp. 189–200.

[26] G. Chiola and A. Ferscha, "Distributed simulation of petri nets," *IEEE Concurrency*, no. 3, pp. 33–50, 1993.

[27] D. M. Nicol and W. Mao, "Automated parallelization of timed petri-net simulations," *Journal of Parallel and Distributed Computing*, vol. 29, no. 1, pp. 60–74, 1995.

[28] S. Muthukrishnan *et al.*, "Data streams: Algorithms and applications," *Foundations and Trends® in Theoretical Computer Science*, vol. 1, no. 2, pp. 117–236, 2005.

[29] Z. Bar-Yossef, R. Kumar, and D. Sivakumar, "Reductions in streaming algorithms, with an application to counting triangles in graphs," in *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms.* Society for Industrial and Applied Mathematics, 2002, pp. 623–632.

[30] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang, "Graph distances in the streaming model: the value of space," in *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms.* Society for Industrial and Applied Mathematics, 2005, pp. 745–754.

[31] D. Garcıa-Soriano and K. Kutzkov, "Triangle counting in streamed graphs via small vertex covers," *Tc*, vol. 2, p. 3, 2014.

[32] J. M. Ruhl, "Efficient algorithms for new computational models," Ph.D. dissertation, Citeseer, 2003.

[33] A. D. Sarma, R. J. Lipton, and D. Nanongkai, "Best-order streaming model," in *Theory and Applications of Models of Computation.* Springer, 2009, pp. 178–191.

[34] A. D. Sarma, S. Gollapudi, and R. Panigrahy, "Estimating pagerank on graph streams," *Journal of the ACM (JACM)*, vol. 58, no. 3, p. 13, 2011.

[35] D. LaSalle and G. Karypis, "Multi-threaded graph partitioning," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on.* IEEE, 2013, pp. 225–236.

[36] F. Pellegrini and J. Roman, "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," in *High-Performance Computing and Networking.* Springer, 1996, pp. 493–498.

[37] T. Feder, P. Hell, S. Klein, and R. Motwani, “Complexity of graph partition problems,” in *Proceedings of the thirty-first annual ACM symposium on Theory of computing.* ACM, 1999, pp. 464–472.

[38] I. Stanton and G. Kliot, “Streaming graph partitioning for large distributed graphs,” in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, 2012, pp. 1222–1230.

[39] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, “Fennel: Streaming graph partitioning for massive scale graphs,” in *Proceedings of the 7th ACM international conference on Web search and data mining.* ACM, 2014, pp. 333–342.

[40] Y. Liu, T. Safavi, N. Shah, and D. Koutra, “Reducing large graphs to small supergraphs: a unified approach,” *Social Network Analysis and Mining*, vol. 8, no. 1, p. 17, 2018.

[41] D. Koutra, U. Kang, J. Vreeken, and C. Faloutsos, “Summarizing and understanding large graphs,” *Statistical Analysis and Data Mining: The ASA Data Science Journal*, vol. 8, no. 3, pp. 183–202, 2015.

[42] J. Batson, D. A. Spielman, N. Srivastava, and S.-H. Teng, “Spectral sparsification of graphs: theory and algorithms,” *Communications of the ACM*, vol. 56, no. 8, pp. 87–94, 2013.

[43] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, “Large-scale cluster management at Google with Borg,” in *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[44] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, “Borg, omega, and kubernetes,” *ACM Queue*, vol. 14, pp. 70–93, 2016.

[45] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, “Mesos: A platform for fine-grained resource sharing in the data center.” in *NSDI*, vol. 11, no. 2011, 2011, pp. 22–22.

[46] S. Choi, R. Myung, H. Choi, K. Chung, J. Gil, and H. Yu, “Gpsf: General-purpose scheduling framework for container based on cloud environment,” in *International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData),.* IEEE, 2016, pp. 769–772.

[47] A. Havet, V. Schiavoni, P. Felber, M. Colmant, R. Rouvoy, and C. Fetzer, “Genpack: A generational scheduler for cloud data centers,” in *Cloud Engineering (IC2E), 2017 IEEE International Conference on.* IEEE, 2017, pp. 95–104.

[48] C. Kaewkasi and K. Chuenmuneewong, “Improvement of container scheduling for docker using ant colony optimization,” in *Knowledge and Smart Technology (KST), 2017 9th International Conference on.* IEEE, 2017, pp. 254–259.

[49] D. Bhamare, M. Samaka, A. Erbad, R. Jain, L. Gupta, and H. A. Chan, “Multi-objective scheduling of micro-services for optimal service function chains,” in *IEEE International Conference on Communications (ICC 2017)*. IEEE, 2017.

[50] U. Awada and A. D. Barker, “Improving resource efficiency of container-instance clusters on clouds,” in *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2017)*. IEEE, 2017.

[51] K. A. Kumar, V. K. Konishetty, K. Voruganti, and G. V. P. Rao, “CASH: context aware scheduler for hadoop,” in *2012 International Conference on Advances in Computing, Communications and Informatics, ICACCI ’12, Chennai, India, August 3-5, 2012*, 2012, pp. 52–61.

[52] A. R. Oskooei and D. G. Down, “COSHH: A classification and optimization based scheduler for heterogeneous hadoop systems,” *Future Generation Comp. Syst.*, vol. 36, pp. 1–15, 2014.

[53] W. Zhang, S. Rajasekaran, S. Duan, T. Wood, and M. Zhu, “Minimizing interference and maximizing progress for hadoop virtual machines,” *SIGMETRICS Performance Evaluation Review*, vol. 42, no. 4, pp. 62–71, 2015.

[54] K. Wang, M. M. H. Khan, N. Nguyen, and S. S. Gokhale, “Modeling interference for apache spark jobs,” in *9th IEEE Int Conference on Cloud Computing, CLOUD 2016, USA*, 2016, pp. 423–431.

[55] J. Hwang, S. Zeng, F. y Wu, and T. Wood, “A component-based performance comparison of four hypervisors,” in *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*. IEEE, 2013, pp. 269–276.

[56] A. J. Younge, R. Henschel, J. T. Brown, G. Von Laszewski, J. Qiu, and G. C. Fox, “Analysis of virtualization technologies for high performance computing environments,” in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE, 2011, pp. 9–16.

[57] R. Morabito, J. Kjällman, and M. Komu, “Hypervisors vs. lightweight virtualization: a performance comparison,” in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. IEEE, 2015, pp. 386–393.

[58] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, and B.-J. Kim, “Performance comparison analysis of linux container and virtual machine for building cloud,” *Advanced Science and Technology Letters*, vol. 66, no. 105-111, p. 2, 2014.

[59] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, “Performance evaluation of container-based virtualization for high performance computing environments,” in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*. IEEE, 2013, pp. 233–240.

[60] C. Ruiz, E. Jeanvoine, and L. Nussbaum, "Performance evaluation of containers for hpc," in *European Conference on Parallel Processing.* Springer, 2015, pp. 813–824.

[61] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance evaluation of microservices architectures using containers," in *14th IEEE International Symposium on Network Computing and Applications, NCA 2015, Cambridge, MA, USA, September 28-30, 2015*, 2015, pp. 27–34.

[62] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015, pp. 171–172.

[63] N. Kratzke and P.-C. Quint, "How to operate container clusters more efficiently? some insights concerning containers, software-defined-networks, and their sometimes counterintuitive impact on network performance," in *International Journal on Advances in Networks and Services*, vol. 8, 2015, pp. 203–214.

[64] H. Khazaei, J. Misic, and V. Misic, "Performance analysis of cloud computing centers using m/g/m/m+r queuing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 5, pp. 936–943, 2012.

[65] H. Khazaei, C. Barna, N. Beigi-Mohammadi, and M. Litoiu, "Efficiency analysis of provisioning microservices," in *Cloud Computing Technology and Science (CloudCom), 2016 IEEE International Conference on.* IEEE, 2016, pp. 261–268.

[66] A. Merino, R. Tolosana-Calasanz, J. Á. Bañares, and J. M. Colom, "A specification language for performance and economical analysis of short term data intensive energy management services," in *Economics of Grids, Clouds, Systems, and Services - 12th International Conference, GECON 2015, Cluj-Napoca, Romania, September 15-17, 2015*, ser. LNCS, vol. 9512, 2015, pp. 147–163.

[67] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 275–287, mar 2007. [Online]. Available: http://doi.acm.org/10.1145/1272998.1273025

[68] A. M. Joy, "Performance comparison between linux containers and virtual machines," in *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in.* IEEE, 2015, pp. 342–346.

[69] R. Krebs, C. Momm, and S. Kounev, "Metrics and techniques for quantifying performance isolation in cloud environments," *Science of Computer Programming*, vol. 90, pp. 116–134, 2014.

[70] A. Fedorova, S. Blagodurov, and S. Zhuravlev, "Managing contention for shared resources on multicore processors," *Communications of the ACM*, vol. 53, no. 2, pp. 49–57, 2010.

[71] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam, "Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines," in *Proceedings of the 2nd ACM Symposium on Cloud Computing.* ACM, 2011, p. 22.

[72] A. Sandberg, A. Sembrant, E. Hagersten, and D. Black-Schaffer, "Modeling performance variation due to cache sharing," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on.* IEEE, 2013, pp. 155–166.

[73] X. Chen, C. Xu, R. P. Dick, and Z. M. Mao, "Performance and power modeling in a multi-programmed multi-core environment," in *Proceedings of the 47th Design Automation Conference.* ACM, 2010, pp. 813–818.

[74] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, and C. Pu, "Understanding performance interference of i/o workload in virtualized cloud environments," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on.* IEEE, 2010, pp. 51–58.

[75] G. Casale, S. Kraft, and D. Krishnamurthy, "A model of storage i/o performance interference in virtualized systems," in *Distributed Computing Systems Workshops (ICDCSW), 2011 31st International Conference on.* IEEE, 2011, pp. 34–39.

[76] X. Pu, L. Liu, Y. Mei, S. Sivathanu, Y. Koh, C. Pu, and Y. Cao, "Who is your neighbor: Net i/o performance interference in virtualized clouds," *IEEE Transactions on Services Computing*, vol. 6, no. 3, pp. 314–329, 2013.

[77] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu, "An analysis of performance interference effects in virtual environments," in *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on.* IEEE, 2007, pp. 200–209.

[78] J. Han, S. Jeon, Y.-r. Choi, and J. Huh, "Interference management for distributed parallel applications in consolidated clusters," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 443–456, 2016.

[79] N. Rameshan, Y. Liu, L. Navarro, and V. Vlassov, "Hubbub-scale: Towards reliable elastic scaling under multi-tenancy," in *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on.* IEEE, 2016, pp. 233–244.

[80] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 77–88.

[81] C. Delimitrou, N. Bambos, and C. Kozyrakis, "Qos-aware admission control in heterogeneous datacenters," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, 2013, pp. 291–296.

[82] C. Delimitrou and C. Kozyrakis, "Bolt: I know what you did last summer," in *The Cloud. In Proc. of the Twenty Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[83] A. Carbone, M. Ajmone-Marsan, K. W. Axhausen, M. Batty, M. Masera, and E. Rome, “Complexity aided design,” *The European Physical Journal Special Topics*, vol. 214, no. 1, pp. 435–459, 2012.

[84] A. Rajhans, S.-W. Cheng, B. Schmerl, D. Garlan, B. H. Krogh, C. Agbi, and A. Bhave, “An architectural approach to the design and analysis of cyber-physical systems,” *Electronic Communications of the EASST*, vol. 21, 2009.

[85] G. D’Angelo and M. Marzolla, “New trends in parallel and distributed simulation: From many-cores to cloud computing,” *Simulation Modelling Practice and Theory*, vol. 49, pp. 320–335, 2014.

[86] L. Yu, C. Moretti, A. Thrasher, S. J. Emrich, K. Judd, and D. Thain, “Harnessing parallelism in multicore clusters with the All-Pairs, Wavefront, and Makeflow abstractions.” *Cluster Computing*, vol. 13, no. 3, pp. 243–256, 2010.

[87] J. M. Colom and M. Silva, “Convex geometry and semiflows in p/t nets. a comparative study of algorithms for computation of minimal p-semiflows,” in *International Conference on Application and Theory of Petri Nets.* Springer, 1989, pp. 79–112.

[88] R. Tolosana-Calasanz, J. Á. Bañares, and J. M. Colom, “Towards Petri net-based economical analysis for streaming applications executed over cloud infrastructures,” in *Economics of Grids, Clouds, Systems, and Services - 11th International Conference, GECON’14, Cardiff, UK, September 16-18, 2014.*, ser. LNCS, vol. 8914, 2014, pp. 189–205.

[89] A. Merino, R. Tolosana-Calasanz, J. Á. Bañares, and J. M. Colom, “A specification language for performance and economical analysis of short term data intensive energy management services,” in *Economics of Grids, Clouds, Systems, and Services - 12th International Conference, GECON’15, Cluj-Napoca, Romania, September 15-17, 2015.*, ser. LNCS, vol. 9512, 2015, p. 16 pages.

[90] R. Valk, *Lectures on Concurrency and Petri Nets: Advances in Petri Nets.* Springer Berlin Heidelberg, 2004, ch. Object Petri Nets, pp. 819–848.

[91] F. García-Vallés and J. Colom, “A boolean approach to the state machine decomposition of Petri nets with OBDD’s,” in *Systems, Man and Cybernetics, 1995. Intelligent Systems for the 21st Century., IEEE International Conference on*, vol. 4, Oct 1995, pp. 3451–3456 vol.4.

[92] J. Colom, M. Silva, and J. Villarroel, “On software implementation of petri nets and colored petri nets using high-level concurrent languages,” in *7th International Workshop on Application and Theory of Petri Nets*, 1986, pp. 207–222.

[93] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, “Recent advances in graph partitioning,” *CoRR*, vol. abs/1311.3144, 2013. [Online]. Available: http://arxiv.org/abs/1311.3144

[94] W. M. Van der Aalst and A. Weijters, "Process mining: a research agenda," 2004.

[95] "Yahoo dataset," http://webscope.sandbox.yahoo.com/catalog.php.

[96] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data.* ACM, 2010, pp. 135–146.

[97] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on.* IEEE, 2010, pp. 170–177.

[98] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing.* USENIX Association, 2012, pp. 10–10.

[99] "Storm," http://storm-project.net/.

[100] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new framework for parallel machine learning," *arXiv preprint arXiv:1006.4990*, 2010.

[101] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[102] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 4. ACM, 1999, pp. 251–262.

[103] D. J. Watts and S. H. Strogatz, "Collective dynamics of small-world networks," *nature*, vol. 393, no. 6684, pp. 440–442, 1998.

[104] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks," *Reviews of modern physics*, vol. 74, no. 1, p. 47, 2002.

[105] R. Tolosana-Calasanz, J. D. Montes, L. F. Bittencourt, O. F. Rana, and M. Parashar, "Capacity management for streaming applications over cloud infrastructures with micro billing models," in *Proceedings of the 9th International Conference on Utility and Cloud Computing, UCC 2016, Shanghai, China, December 6-9, 2016*, 2016, pp. 251–256. [Online]. Available: http://doi.acm.org/10.1145/2996890.3007868

[106] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "A performance analysis of ec2 cloud computing services for scientific computing," in *1st Intl. Conf. on Cloud Computing (CloudComp), Munich, Germany*, 2009.

[107] N. Kratzke and P.-C. Quint, "Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study," *Journal of Systems and Software*, vol. 126, pp. 1–16, 2017.

[108] O. Kummer, F. Wienberg, M. Duvigneau, J. Schumacher, M. Köhler, D. Moldt, H. Rölke, and R. Valk, "An extensible editor and simulation engine for petri nets: Renew," in *International Conference on Application and Theory of Petri Nets*. Springer, 2004, pp. 484–493.

[109] Student, "The probable error of a mean," *Biometrika*, pp. 1–25, 1908.

[110] M. Raho, A. Spyridakis, M. Paolino, and D. Raho, "Kvm, xen and docker: A performance analysis for arm based nfv and cloud computing," in *3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*. IEEE, 2015, pp. 1–8.

[111] S. Brunner, M. Blochlinger, G. Toffetti, J. Spillner, and T. M. Bohnert, "Experimental evaluation of the cloud-native application design," *IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, pp. 488–493, 2015.

[112] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[113] T. Bingmann, M. Axtmann, E. Jöbstl, S. Lamm, H. C. Nguyen, A. Noe, S. Schlag, M. Stumpp, T. Sturm, and P. Sanders, "Thrill: High-performance algorithmic distributed batch data processing with c++," *arXiv preprint arXiv:1608.05634*, 2016.

[114] V. Medel, O. Rana, U. Arronategui *et al.*, "Modelling performance & resource management in kubernetes," in *Proceedings of the 9th International Conference on Utility and Cloud Computing*. ACM, 2016, pp. 257–262.

[115] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[116] J. Ezpeleta, J. M. Colom, and J. Martinez, "A petri net based deadlock prevention policy for flexible manufacturing systems," *IEEE transactions on robotics and automation*, vol. 11, no. 2, pp. 173–184, 1995.

[117] V. Medel, C. Tolón, U. Arronategui, R. Tolosana-Calasanz, J. Á. Bañares, and O. F. Rana, "Client-side scheduling based on application characterization on kubernetes," in *International Conference on the Economics of Grids, Clouds, Systems, and Services*. Springer, 2017, pp. 162–176.

[118] T. A. Brown, *Confirmatory factor analysis for applied research*. Guilford Publications, 2014.

[119] N. K. Bowen and S. Guo, *Structural equation modeling.* Oxford University Press, 2011.

[120] L. R. Fabrigar and D. T. Wegener, *Exploratory factor analysis.* Oxford University Press, 2011.

[121] H. Abdi and L. J. Williams, "Principal component analysis," *Wiley interdisciplinary reviews: computational statistics*, vol. 2, no. 4, pp. 433–459, 2010.

[122] C.-H. Li, "Confirmatory factor analysis with ordinal data: Comparing robust maximum likelihood and diagonally weighted least squares," *Behavior Research Methods*, vol. 48, no. 3, pp. 936–949, 2016.

[123] Y. Rosseel, "lavaan: An R package for structural equation modeling," *Journal of Statistical Software*, vol. 48, no. 2, pp. 1–36, 2012. [Online]. Available: http://www.jstatsoft.org/v48/i02/

[124] C. DiStefano, M. Zhu, and D. Mindrila, "Understanding and using factor scores: Considerations for the applied researcher," *Practical Assessment, Research & Evaluation*, vol. 14, no. 20, p. 2, 2009.

[125] "Persistence of vision raytracer (version 3.7)[computer software]," http://www.povray.org/download/.

[126] "Iozone filesystem benchmark [computer software]," http://www.iozone.org/.

[127] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers," University of Virginia, Charlottesville, Virginia, Tech. Rep., 1991-2007, a continually updated technical report. http://www.cs.virginia.edu/stream/. [Online]. Available: http://www.cs.virginia.edu/stream/

[128] G. Karypis and V. Kumar, "MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0," http://www.cs.umn.edu/~metis, University of Minnesota, Minneapolis, MN, 2009.

[129] "Linux kernel `perf_event.h`."

[130] G. E. Box and D. R. Cox, "An analysis of transformations," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 211–252, 1964.

[131] D. Chen and R. J. Plemmons, "Nonnegativity constraints in numerical analysis," in *The birth of numerical analysis.* World Scientific, 2010, pp. 109–139.

[132] M. Slawski, M. Hein *et al.*, "Non-negative least squares for high-dimensional linear models: Consistency and sparse recovery without regularization," *Electronic Journal of Statistics*, vol. 7, pp. 3004–3056, 2013.

Glossary

ϵ-approximation Given a function, we say that it is an ϵ-approximation of another function if the returned value for the first one is always bound by a constant ϵ multiplied by the second function. 39, 45, 46

Balance Factor Given a partition solution of a graph, the Balance Factor ρ measures the ratio between the number of edges of the partition with more edges and the number of edges of a balanced partition. 50

BFS Breadth First Search. Breadth First Search is a search algorithm for tree and graph structures which explores all neighbour nodes at the present depth before exploring nodes at deeper levels. When referred to the arrival nodes of a streamed graph, it is the sequence of nodes given by such algorithm. 38, 43, 48–50

CFA Confirmatory Factor Analysis. Confirmatory Factor Analysis is a set of techniques which allows to identify how a set of observed variables are affected by a set of factors to accept or reject an *a priori* hypothesis. 83, 85–88, 92, 93, 108, 112, *see* EFA & SEM

client-side scheduler A client-side scheduler is an algorithm which implements a certain scheduling policy to allow the client to use operational information in a distributed system. It is deployed above the specific operational scheduler. 5, 71, 72, 79, 109, 112

CNA Cloud Native Application. A Cloud Native application is a service or an application which has been designed and developed for a cloud platform. This means that these applications are loosely-coupled and they support elasticity by design. 56

Complex System A Complex System is a system which can be described as many components which interact with each other. Each of these components might describe in turn a whole and meaningful system with its own subsystem or components. These systems are also called System of Systems (SoS). 1, 2, 4, 19, 20, 23, 111, 112

conservative resource In a system, a conservative resource is a resource that can be allocated and released by processes without being consumed. Thus, the total amount of these resources is an invariant in the system. 19, 24, 84

cutting edge Given a set of partition solution of a graph, a cutting edge is a edge whose vertices belong to different partitions. 45, 46

DAG Directed Acyclic Graph. A Directed Acyclic Graph is a directed graph (edges have a direction associated) which have no cycles. 47

Data Stream A Data Stream is a flux of data which arrives to the system as a sequence of items. The stream is unbounded in size and it is not feasible to access data with in a random manner. 3, 4, 35, 37, 38

DES Discrete Event System. A Discrete Event System is a dynamic system which is described using discrete estates and the transition between them is modelled by transitions triggered by events. 19, 21, 22, 32

DFS Depth First Search. Depth First Search is a search algorithm for tree and graph structures which explores first all nodes at deeper level depth before exploring neighbour nodes. When referred to the arrival nodes of a streamed graph, it is the sequence of nodes given by such algorithm. 38, 43

EFA Exploratory Factor Analysis. Exploratory Factor Analysis are a set of statistical techniques to discover the underlying factor structure of a set of variables. Unlike CFA, EFA assumes that any variable may be associated with any factor. 86, *see* CFA

EV Electric Vehicle. An Electric Vehicle, is a vehicle which uses one or more electric motors or traction motors for propulsion. xv, 2, 5, 19, 23–28

IoT Internet of Things. The Internet of Things is the network of physical devices, home appliances and, in general, items embedded with electronics, sensors or actuators which enables them to exchange data and information. 2, 20

LP Logical Process. A Logical Process is an abstraction which encapsulates the minimal simulation unit for a distributed or parallel simulator. 12, 13, 22, 31, 32, 112

non-conservative resource In a system, a non-conservative resource is a resource which can be allocated and released by processes. However, when a process releases it, the resource is consumed and cannot be used by other processes. 19

Object Net In the Nets-within-nets paradigm, a Object Net – also named Token Net – is a Petri Net which acts as a token of a System Net. Object nets might be synchronised through transitions with other Object Nets which are in the same place of the System Net or with the System Net. 24, 25, 58, 60, *see* System Net & Object Nets with Reference Semantic

Object Nets with Reference Semantic In the Nets-within-nets paradigm, an Object Net with Reference Semantic means that different instances of an Object Net in the System Net reference to the same Object Net. Object Nets with Value Semantic refers to the inverse behaviour. 6, 22, 23, 28, 57, *see* System Net & Object Net

PageRank PageRank algorithm is an algorithm to measure the importance of nodes in a graph. The importance of a node depends on the number of incoming edges. PageRank algorithm is used by Google Search Engine to rank websites. xv, 48, 51, 52, 73–76, 79

PCA Principal Component Analysis. The Principal Component Analysis is a statistical procedure to reduce a set correlated variables into a set of linearly uncorrelated variables called principal components. 86

PN Petri Net. A Petri Net is a formal model to describe the behaviour of distributed systems. It is a bipartite graph, in which nodes represent transitions (i.e. events) and places (i.e. conditions or states). The edges represent which places are preconditions and/or postconditions for which transitions. 5, 21–25, 28, 30, 32, 35, 57, 112

Pod A pod is an abstraction introduced by the Software Kubernetes. It represent several coupled containers that are scheduled to the same machine. xv, 56–70, 75, 76, 79–81, 112, 125, 126

QoS Quality of Service. Quality of Service is the description of the non-functional requirements of an application or a service related to performance metrics. 1–5, 54, 84, 111, 112, 125

RAS Resource Allocation System. A Resource Allocation System is a system where the competition for shared resources between concurrent processes has a relevant role. 84

RTT Round-trip Time. The Round-trip Time is the time that a message –or a signal – takes to be sent plut the time to receive the acknowledgement of that message to be received. 62, 65

SEM Structural Equation Modelling. Structural Equation Modelling are a set of mathematical models to analyse or describe constructs of data. It includes multiple techniques, such as CFA, Path analysis, latent growth models, among others. 85, *see* CFA

SLA Service-level Agreement. Service Level Agreement is the commitment between a service provider and the client which regulates certain QoS – i.e. quality, quantity or availability of the service. 1, 3

SoI Source of Interference. A Source of Interference is a computational resource which is shared between process/containers/VMs inside a physical machine. The competition for that resource make that those instances interference between them leading to a performance degradation. Examples of Sources of Interference are the CPU, the memory hierarchy, the I/O filesystem and the network. 18, 84

SPN Stochastic Petri Nets. A Stochastic Petri Nets is a kind of Petri Net where the transitions fire after a probabilistic delay determined by a random variable. 11, 21, *see* TPN & PN

System Net In the Nets-within-nets paradigm, the System Net is a Petri Net whose tokens might be Objects Nets – also called Token Nets. The System Net describe the interaction mechanisms between the Object Net and the system. 24, 25, 29, 58, 60, *see* Object Net & Object Nets with Reference Semantic

TPN Timed Petri Nets. A Timed Petri Net is a kind of Petri Net which attach a temporal duration to each transitions. Tokens are distributed among places and among firing transitions – when a transition is fired, tokens are not put into the output places until the time attached to the transition has happened. 11, 21, 22, 31–33, 112, *see* SPN & PN

VM Virtual Machine. A Virtual Machine is an isolation mechanism which allows to share resources between different tenants hosted in the same physical machine. A VM emulates a full Operative System. 2, 9, 15–17, 31, 32, 48, 56, 69, 70, 72, 81, 84, 112, 123, 126