TESIS DE LA UNIVERSIDAD DE ZARAGOZA

2013

101

Javier Celaya Alastrué

STaRS: a scalable task routing approach to distributed scheduling

Departamento
Informática e Ingeniería de Sistemas

Director/es
Arronategui Arribalzaga, Unai

ISSN 2254-7606

Prensas de la Universidad
Universidad Zaragoza

Universidad Zaragoza
1542

Tesis Doctoral

STaRS: A SCALABLE TASK ROUTING APPROACH TO DISTRIBUTED SCHEDULING

Autor

Javier Celaya Alastrué

Director/es

Arronategui Arribalzaga, Unai

UNIVERSIDAD DE ZARAGOZA

Informática e Ingeniería de Sistemas

2013

STaRS: A Scalable Task Routing Approach to Distributed Scheduling

PhD Dissertation

Javier Celaya Alastrué

Advised by Unai Arronategui Arribalzaga

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

Submitted in fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science and Systems Engineering with "International Doctor" Mention from Universidad de Zaragoza. June 2013.

Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza

Instituto Universitario de Investigación de Ingeniería de Aragón
Universidad Zaragoza

STaRS: A Scalable Task Routing Approach to Distributed Scheduling

Javier Celaya Alastrué

Advisor:

Unai Arronategui Arribalzaga	Universidad de Zaragoza, Spain

Reviewers:

Arnaud Legrand	Université de Grenoble, France
Loris Marchal	Ecole Normale Supérieure de Lyon, France

Committee:

José Manuel Colom Piazuelo	Universidad de Zaragoza, Spain
José Ángel Bañares Bañares	Universidad de Zaragoza, Spain
Frédéric Vivien	Ecole Normale Supérieure de Lyon, France
Rizos Sakellariou	University of Manchester, United Kingdom
Sergio Arévalo Viñuales	Universidad Politécnica de Madrid, Spain
José Miguel Alonso	Euskal Herriko Unibertsitatea, Spain
Leandro Navarro Moldes	Universitat Politècnica de Catalunya, Spain

Quiero dedicar esta tesis a Cristina y Kira,
por su infinita paciencia, amor y cariño.

Resumen

La planificación de muchas tareas en entornos de millones de nodos no confiables representa un gran reto. Las plataformas de computación más conocidas normalmente confían en poder gestionar en un elemento centralizado todo el estado tanto de los nodos como de las aplicaciones. Esto limita su escalabilidad y capacidad para tolerar fallos. Un modelo descentralizado puede superar estos problemas pero, por lo que sabemos, ninguna solución propuesta hasta el momento ofrece resultados satisfactorios. En esta tesis, presentamos un modelo de planificación descentralizado con tres objetivos: que escale hasta millones de nodos, sin una pérdida de prestaciones que lo inhabilite; que tolere altas tasas de fallos; y que permita la implementación de varias políticas de planificación para diferentes situaciones.

Nuestra propuesta consta de tres elementos principales: un modelo de datos genérico para representar la disponibilidad de los nodos de ejecución; un esquema de agregación que propaga esta información por una capa de red jerárquica; y un algoritmo de reexpedición que, usando la información agregada, encamina tareas hacia los nodos de ejecución más apropiados. Estos tres elementos son fácilmente extensibles para proporcionar diversas políticas de planificación. En concreto, nosotros hemos implementado cinco. Una política que simplemente asigna tareas a nodos desocupados; una política que minimiza el tiempo de finalización del trabajo global; una política que cumple con los requerimientos de fecha límite de aplicaciones tipo «saco de tareas»; una política que cumple con los requerimientos de fecha límite de aplicaciones tipo *workflow*; y una política que otorga una porción equitativa de la plataforma a cada aplicación.

La escalabilidad se consigue a través del esquema de agregación, que provee de suficiente información de disponibilidad a los niveles altos de la jerarquía sin inundarlos, y el algoritmo de reexpedición, que busca nodos de ejecución en varias ramas de la jerarquía de manera concurrente. Como consecuencia, los costes de comunicación están acotados y los de asignación muestran un comportamiento casi logarítmico con el tamaño del sistema. Un millar de tareas se asignan en una red de 100.000 nodos en menos de 3,5 segundos, así que podemos plantearnos utilizar nuestro modelo incluso con tareas de tan solo unos minutos de duración. Por lo que sabemos, ningún trabajo similar ha sido probado con más de 10.000 nodos.

Los fallos se gestionan con una estrategia de mejor esfuerzo. Cuando se detecta el fallo de un nodo, las tareas que estaba ejecutando son reenviadas por sus propie-

tarios y la información de disponibilidad que gestionaba es reconstruida por sus vecinos. De esta manera, nuestro modelo es capaz de degradar sus prestaciones de manera proporcional al número de nodos fallidos y recuperar toda su funcionalidad. Para demostrarlo, hemos realizado pruebas de tasa media de fallos y de fallos catastróficos. Incluso con nodos fallando con un periodo mediano de solo 5 minutos, nuestro planificador es capaz de continuar dando servicio. Al mismo tiempo, es capaz de recuperarse del fallo de una fracción importante de los nodos, siempre que la capa de red jerárquico que sustenta el sistema pueda soportarlo.

Después de comprobar que es factible implementar políticas con muy distintos objetivos usando nuestro modelo de planificación, también hemos probado sus prestaciones. Hemos comparado cada política con una versión centralizada que tiene pleno conocimiento del estado de cada nodo de ejecución. El resultado es que tienen unas prestaciones cercanas a las de una implementación centralizada, incluso en entornos de gran escala y con altas tasas de fallo.

Abstract

Scheduling many tasks in environments of millions of unreliable nodes is a challenging problem. Well-known platforms usually rely on managing the state of every computing node and application at a centralized entity. This limits their scalability and resilience. A decentralized model can overcome these problems but, to our knowledge, no solution proposed in the literature provides a satisfactory result. In this thesis, we present a decentralized scheduling model with three objectives: to scale to millions of nodes, without a loss of performance that would render it useless; to tolerates high rates of failures; and to enable the implementation of many scheduling policies for different situations. We support our claims with the results from trace-driven simulation tests on a network of up to a million nodes.

Our proposal consists of three main elements: a generic data model that represents execution node availability; an aggregation scheme that propagates this information on a hierarchical network overlay; and a forwarding algorithm that, using the aggregated information, routes tasks towards the most suitable execution nodes. These three elements are easily extensible to provide very different scheduling policies. We have implemented five policies. A policy that just allocates tasks to idle nodes; a policy that minimizes the global makespan; a policy that fulfills deadline requirements of bag-of-tasks applications; a policy that fulfills deadline requirements of workflow applications; and a policy that provides a fair share of the platform to every application.

The scalability is achieved through the aggregation scheme, that provides enough availability information to the top levels of the hierarchy without flooding them, and the forwarding algorithm, that looks for execution nodes in several branches of the hierarchy concurrently. In consequence, the communication overhead is bounded and the allocation cost shows an almost logarithmic behavior with the system size. A thousand tasks are allocated to a network of 100,000 nodes in less than 3.5 seconds, so we can consider using our model on tasks of only some minutes long. As far as we know, no other similar work has been tested on more than 10,000 nodes.

Faults are managed with a best-effort strategy. When a node failure is detected, the tasks it was executing are resubmitted by their owners and the availability information it managed is rebuilt by its neighbors. In this way, our model is able to degrade its performance accordingly to the number of failed nodes and recover its functionality. To show it, we have performed tests of churn and catastrophic failures. Even with nodes failing with a median period of only 5 minutes, our

scheduler is able to continue giving a degraded service. Meanwhile, it is able to recover from the failure of an important fraction of the nodes, as long as the underlying hierarchical overlay supports it.

After checking the feasibility of implementing policies with very different objectives on our scheduling model, we have also tested their performance. We have compared each of them with a centralized version that has full knowledge of every execution node state. The result is that they perform very close to a centralized implementation, in large-scale environments and with high rates of failures.

Acknowledgements

It has been a long way since I finished my master thesis in December 2005, the work that set the beginning of my research career. During this time, I have met many people that have helped me, in some way or another, and I would like to thank.

First Unai, for advising me in all the tasks that the writing of a PhD thesis implies. Thank you for all the hours of discussion and all the blackboards full of annotations. Thanks also to Loris Marchal, that dedicated three months of his time so that I could make a stay at the ENS in Lyon. Your collaboration and advices are very much appreciated.

I would also like to thank all my friends and colleagues. The people at the 'Becarios' office, that shared so many good moments with me: Carlos Bobed, Rosario Aragüés, Estíbaliz Fraca, Dorian Gálvez, Jorge Álvarez, Ignacio Requeno, Diego Pérez and María José Ibañez. Special thanks to Jorge Álvarez, that helped me administering the computers I used for the simulation tests. Thanks to Marian Giménez, Héctor Blanco, Víctor Catalán, David Ceresuela and Víctor Medel, that accepted the challenge of carrying out their master thesis under my advisory. Their work contributed significantly to my research. Also, many thanks to all the people at the ENS Lyon for their warm welcome.

And last, but not least, many thanks to my girlfriend and my family for their patience and support. I would have never finished this thesis without them.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1.

Introduction

"In all matters, before beginning, a diligent preparation should be made."
— Cicero

1.1. Motivation

Computers get more powerful every day. They get faster, they get more storage and they reach more devices. Today, a smartphone has the same capabilities as a personal computer ten years ago. With such amount of ubiquitous computing power, we are able to run applications that solve problems with heavy requirements in a fraction of the time they needed before. Common examples (Figure 1.1) are applications that search through an enormous state space, like chemical combination models; that need to process huge amounts of data, like outer space signal processing; or that simulate complex systems, like climate change simulations. In many cases, these problems present *coarse-grained parallelism*. That is, they can be divided into multiple tasks so that all or part of them are executed concurrently, with few or no communication among them. For instance, several tasks executing the same code may process different parts of the data set, or they may process the same data with different code. When there exists no dependency between any two tasks, all of them can be started at once. This kind of application is called a *bag-of-tasks*. When the execution of some tasks depends on the outcome of other tasks, it is called a workflow. This thesis is mostly focused on bag-of-tasks applications, but Chapter 6 deals with workflow applications.

Since there is little communication among coarse-grained parallel applications, they usually perform well on slow networks or through high-latency links. So, they are the best candidate for distributed computing platforms. These platforms consist of a set of geographically distributed, loosely-coupled and unreliable execution nodes. On top of them, a set of common services allocate tasks to execution nodes and monitor their successful result. A resource discovery service finds the execution nodes that match certain criteria. A scheduling service decides how to perform the allocation so that an objective is optimized. Other services provide additional functionality, like fault tolerance, data storage and user interaction. Distributed computing platforms are very flexible because they can be built using commodity hardware, connecting different geographical locations and administrative domains.

In this thesis, we tackle the problem of scheduling applications to a very large distributed computing platform. We present a *distributed scheduling model* that is able to allocate many tasks in environments of millions of nodes in a matter of seconds. A distributed scheduling model describes the details and concepts that a distributed computing platform uses to allocate applications to nodes: application

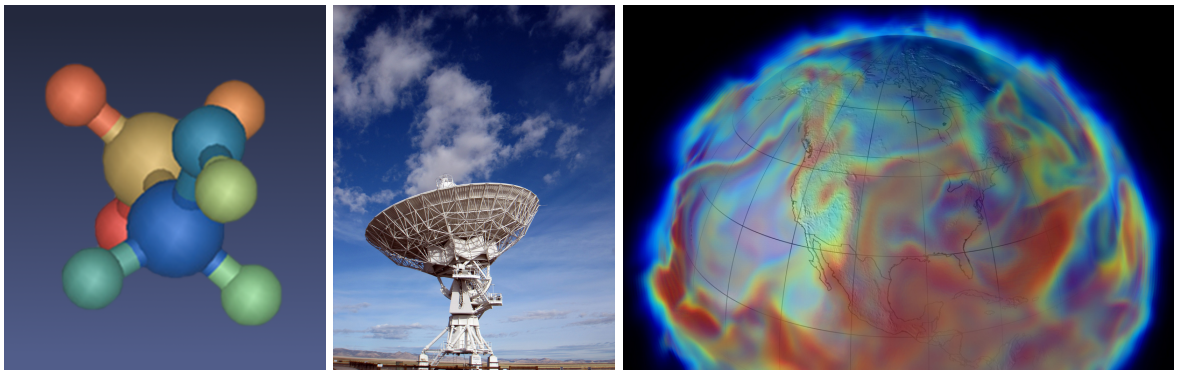

Figure 1.1.: Many applications require high amounts of computational resources. For instance, chemical combination models, outer space signal processing and climate change simulations. Pictures shared with a Creative Commons Attribution License by users wasoxygen, karenandbrademerson and NASA Goddard Photo and Video, from Flickr (http://www.flickr.com).

parameters, node properties, algorithms, objective, etc. Well-known platforms usually implement a scheduling model that relies on managing the state of every execution node and application at a centralized entity. This includes Condor [90], BOINC [8], Falkon [77] and Hadoop [73], among others. The shift of these technologies towards newer environments, like cloud computing [11, 22], and applications, like many-task computing [76, 53], still maintain a dependency on full knowledge of the system. If that knowledge is very detailed, as in the case of Condor and its ClassAds system, the cost of its management limits the scalability of the centralized manager, which can only cope with some hundreds of nodes. But not enough detail limits its scheduling capabilities, as in the case of BOINC. In both cases, the failure of the centralized manager means the failure of the whole system.

A decentralized solution is able to overcome these problems. Accessing the state of the computing nodes in a decentralized way can provide a good level of detail without putting all the stress on a single machine. Likewise, this design is much more resilient to individual node failures. With these benefits, a decentralized design is still able to manage most of the scheduling policies of a centralized one. Many works claim to implement these properties to some degree [58, 12, 98, 75]. However, to our knowledge, no one fully provides all of them at the same time. Their tests comprise less than a few thousand nodes, failure scenarios are scarcely found and they only compare heuristics for the same policy.

This thesis presents STaRS (**S**calable **Ta**sk **R**outing approach to distributed **S**cheduling), a novel distributed scheduling model that aims to deal with millions of nodes, tolerate high rates of failures and support policies with very different objectives. This design it suitable for the most demanding applications (e.g. many-task computing applications with thousands of tasks and several scheduling restrictions) in a variety of scenarios (from volunteer computing platforms to cloud data centers).

1.2. Hypothesis and Goals

A distributed computing infrastructure consists of a set of nodes connected through a communication network. Nodes are independent, in the sense that they have heterogeneous resources, independent clocks and a user behind each of them that may act at will. They have an availability difficult to

predict, either due to human decisions or to unexpected failures. Besides, the network introduces a variable delay in the message transmission that cannot be ignored.

Compared to a monolithic design, such as multiprocessor supercomputers, this distributed design is harder to manage, but also has many advantages. It is easy to increase the available resources when needed. New nodes can be connected to the network, old links can be replaced with faster ones. In this way, its overall performance grows incrementally. Moreover, these platforms can often be deployed over commodity hardware, so they are cheaper. However, hardware malfunction, natural disasters and human negligence become more probable as scale increases. But due to its intrinsic independence, a distributed design is able to stop using failed parts until they are replaced.

To exploit these advantages, we consider that a distributed scheduling model requires the following properties:

- Scalability: It must deal with a huge number of nodes, from hundreds to millions, without a loss of performance that would render it useless.
- Fault-tolerance: Failures must be expected with such scales of operation. The model must degrade its performance and recover its normal operation after the failure of a node. Moreover, it must tolerate the failure of several nodes, even with high rates of failures.
- Versatility: It must enable the implementation of many scheduling policies for different situations. This covers application types (bag-of-tasks, workflows, etc.), requirements (memory, disk, deadline, etc.) and platform configurations (loosely- or tightly-coupled platforms).

The objective of this thesis is to design such a distributed scheduling model. To support these claims, we drive several simulation tests. We simulate networks of different size, with up to a million nodes, to study the behavior of our model with increasing scale. We also evaluate if our model is able to recover and maintain its operation with different rates of failures. Finally, we test the performance of five different policies. To provide a context, we compare them with a centralized implementation and a random allocation, and evaluate how near we get to the former and how far from the later.

1.3. Context and Contributions

The research that originates this thesis has been carried out within the Group of Discrete Event Systems Engineering (GISED)[1], from the Aragon Institute of Engineering Research (I3A)[2] and the Departamento de Informática e Ingeniería de Sistemas (DIIS)[3] of the Universidad de Zaragoza[4]. It has been funded by project CICYT DPI2006-15390 of the Spanish Government, grant B018/2007 (for the formation of researchers) of the Aragonese Government, and the GISED as a group of excellence of the Aragonese Government. Additionally, part of this research has been carried out in collaboration with the Laboratoire de l'Informatique du Parallélisme (LIP)[5] of the École normale supérieure de Lyon[6], as a three-month stay funded by grant TME2008-01125 of the Spanish Government.

[1] http://webdiis.unizar.es/GISED
[2] http://i3a.unizar.es
[3] http://diis.unizar.es
[4] http://www.unizar.es
[5] http://www.ens-lyon.fr/LIP
[6] http://www.ens-lyon.eu

As a result, the main contributions of this thesis are:

- An aggregation scheme that propagates information about the availability of nodes on a tree-based overlay network. It consists of a generic data model that represents the execution node availability, and a common set of operations to aggregate this information. Each policy specializes the data model with the actual node properties it depends on, which may range from scalar values (like available memory space) to functions (like available computation before a deadline). The aggregation algorithms can be tuned to find a tradeoff between accuracy and cost.

- A forwarding algorithm that routes tasks towards the most suitable execution nodes. The aggregated availability information is used to find the correct route. Using the policy-specific availability information, it is able to perform stateless resource discovery and allocation simultaneously.

- Five example policies that illustrate the use of the scheduling model in specific situations. The first one is a simple policy to allocate idle nodes that meet memory and disk space requirements, and is used to explain the aggregation scheme and the forwarding algorithm. Then, we also present a policy that minimizes the global makespan, finishing all the work as soon as possible; a policy that fulfills application deadlines; a policy that looks for a fair share of the platform among applications, using an approximation of the stretch; and a policy for applications with deadlines, but using workflows of tasks instead of bags of tasks.

The foundations of the tree overlay, the aggregation scheme and the simple idle/busy policy have been published in the proceedings of the 7th International Conference on Grid Computing [33] and in volume 4208 of the journal Lecture Notes in Computer Science, titled "High Performance Computing and Communications" [32]. This last paper was awarded the **best paper** of the 2nd International Conference on High Performance Computing and Communications, Munich, September 2006. The development of those ideas, along with further simulation and comparison with similar work, has been published in the journal Future Generation Computer Systems [36]. The policy with deadlines and its results have been published in the proceedings of the 12th International Conference on Grid Computing [35]. A first approach to the fair share policy has been published in the proceedings of the 10th International Symposium on Cluster, Cloud and Grid Computing [37]. Finally, the policy for workflows with deadlines has been published in the proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing [34].

1.4. Thesis Overview

First, Chapter 2 presents the current state of the art in scheduling in distributed computing platforms. Then, Chapter 3 explains the details of our main contribution to that state of the art, the decentralized scheduling model. We give an overview of its architecture and its four main components: the scheduling model, the nodes, the overlay and the fault-tolerance mechanisms. We explain how these components work together to build up a generic aggregation scheme for availability information and a generic task routing process. Both can be instantiated through specific policies, so we give an example

of a simple policy, the IBP policy. It allocates tasks to idle nodes with enough available memory and disk space.

Then, Chapters 4 through 7 present four additional policies with different scheduling objectives. In Chapter 4, the MMP policy tries to minimize the makespan of all the scheduled applications in the system. In Chapter 5, the DP policy tries to successfully schedule applications with deadlines. In Chapter 6, the WDP policy explores the use of workflow applications with our decentralized model, extending the DP policy to this kind of applications. In Chapter 7, the FSP policy schedules applications so that every one obtains an equal share of the system.

Finally, Chapter 8 analyzes the performance of our model with the proposed policies. It presents the accuracy and scalability results of the generic model, comparing its behavior among the different policies. Then it evaluates the performance of each policy individually, compared with a centralized implementation with full knowledge and a random allocation. Chapter 9 discusses these results and summarizes the conclusions of this thesis.

Chapter 2.

State of the Art

STaRS provides a scalable scheduling model to virtually any type of distributed computing platform, from loosely coupled desktop grids to dense clusters. Its design decisions are motivated by the shortcomings we have observed in the related literature. Centralized designs have a structurally limited scalability and resilience, so many decentralized alternatives have been proposed. Most use aggregated information in some way, but they fail to provide good scalability, fault-tolerance and flexibility at the same time.

2.1. Centralized Designs

Centralized scheduling services are common in cluster, grid and cloud computing, so their limitations are well known. The most popular desktop grid platform is BOINC [8], with millions of volunteers in projects like SETI@home [9]. However, it reaches these high scales by having the global scheduler manage very little detail about each execution node. On the contrary, Condor [90] provides a powerful scheduling engine to cluster and grid platforms at the expense of needing ad-hoc solutions [43] to reach scales over the thousand nodes. Falkon [77] aims for higher scales sacrificing some of the functionality provided by Condor, like multiple queues and priorities. However, its centralized dispatcher still scales to a maximum number of managed tasks and resources. In their tests, they reach two million tasks on 54,000 nodes. Something similar happens with Mesos [50]. It does not perform any scheduling, it just manages a set of resources and provides a fair share among several distributed computing frameworks. It offers resources to the platforms, and they perform the actual scheduling. In this way, it should be able to scale to more than the 50,000 nodes the authors advertise, but a maximum is inevitable. STaRS circumvents these limitations because it aggregates rich information about execution nodes to avoid centralizing the scheduling decisions. It reaches higher scales than Condor or Falkon, without an expected maximum, while being able to implement a wider range of scheduling policies than BOINC.

Other platforms with application-specific policies rely on centralized resource management. Nimrod [6, 4, 5, 3] performs parameter exploration experiments acting as a resource broker: it first gathers the available resources that fit the experiment requirements and then schedules the tasks on them. MapReduce [39] and Hadoop [73] process large-scale data sets, centralizing data management and job scheduling on dedicated nodes. Policies based on economic models [19] set prices and offers independently from each other, but inflation adjustment and price gathering problems have been solved in practice with centralized market managers [48, 21] or non-scalable algorithms [95, 93]. Many-task computing platforms [76, 53] focus on the scheduling of a large number of short, data-intensive tasks. Raicu et al. [76] think that they must relax some constraints found in other platforms to be able to

cope with this amount of fine-grained workload. This is the case of Falkon commented before. All these works would benefit from a decentralized model like STaRS.

2.2. Decentralization with Aggregated Information

Several proposals adapt grid schedulers to use aggregated information describing each domain. They are still not fully decentralized, but claim to increase their scalability by using less detailed information with similar capabilities. Rodero et al. [84] use this information to rank brokers of each domain by suitability. They define a distributed meta-broker architecture that distributes the aggregated information to every domain, so that the best broker for a job can be selected. Kokkinos and Varvarigos [59] use aggregation to reduce the amount of information exchanged between domains, limit the exposure of sensitive information and improve interoperability. They focus on the aggregation accuracy for different kinds of attributes and clustering operators. Unfortunately, the architecture is only a two-level hierarchy, with a centralized meta-scheduler at the top level, that decides which domain a task is allocated to. Brunner et al. [18] propose the use of a conceptual clustering algorithm to summarize resource capabilities in different grid domains. They focus on finding suitable execution nodes in near locations to reduce data transmission times. There is no centralized scheduler, but all domains must know each other to maximize the matchmaking performance. Rahman et al. [75] map a d-dimensional logical index to a distributed hash table (DHT). Execution nodes are published in the index, using capabilities as coordinates, and they are found with point and range queries.

Many decentralized solutions have been proposed to date, some of them also using aggregated information. In the field of the resource discovery with aggregated information, Cai and Hwang [23] build distributed aggregation trees on a Chord-like DHT[88], that aggregate node information for Grid resource monitoring. SWORD [7] allows complex queries to search for computational resources. The authors propose a decentralized implementations based on partitioning the availability information space into ranges and map them to a DHT. Each node is responsible for the availability information of the set of resources in the same range. However, partitioning must be done carefully to reach good load-balancing. Cohesion [85] is a decentralized, tree-based information aggregation system on an unstructured peer-to-peer grid platform. They define two ways of building the tree from the set of nodes, one focused on efficiency and another focused on scalability. NodeWiz [12] implements a Grid information service specialized on range queries. It uses a binary balanced tree to partition an attribute space and is able to look for nodes that match certain criteria, but it is limited to scalar attributes. Cardosa and Chandra [26] present an hierarchical aggregation method that clusterizes resource capacity distribution functions into so called “resource bundles”. These bundles provide statistical information about resource properties, with a level of confidence. This implies that there is a certain probability of discovering nodes that may not fulfill the requirements of the requested application.

Other works also try to decentralize the scheduling component, but we have seen no other work that combines scalability, fault tolerance and flexibility as we do. Diet [28, 27] uses a static, ad-hoc hierarchy to route requests to execution nodes, but no aggregated information is used for forwarding. Instead, the root node sends each request to all the execution nodes, and they respond whether they can execute it. Intermediate nodes use stateful queues to control the flow of requests. This architecture is difficult to scale to millions of nodes. It is not very resilient, either, but they maintain

links to some ancestors other than the father to manage failures. WaveGrid [98] uses a CAN [80] overlay to organize nodes by timezone. It allocates work to idle nodes whose timezone is currently in the night. Resources are discovered by randomly selecting some initial nodes and then using an expanding ring search. While it scales fairly well, the information used for scheduling is too limited and random to allow more complex policies. Kim et al. [58] also use CAN to organize nodes by their resource availability and aggregate queue length information. They justify this overlay by its superior scalability and fault-tolerance properties over a tree overlay, but its non-hierarchical structure is less suited for the aggregation of data. Their solution is, at each cell of the CAN overlay, to aggregate the information coming only from the same row in each dimension, which discards most information to make scheduling decisions. Kwan and Mupala [61] use a super-peer network to schedule bag-of-tasks application in volunteer desktop grids. A gossip protocol scatters aggregated information about computing power and location of resources between super-peers. The scheduling policy consists in just assigning tasks to the fastest idle node in the selected super-peer. Unstructured networks are more resilient to failures, as long as super-peers have a very low failure rate.

2.2.1. Aggregating Information on Trees

All these works highlight that aggregating resource information in a tree is a popular approach to decentralize the resource discovery process. It scales well by reducing the amount of information that needs to be managed. A similar approach is often found in sensor networks [52, 71], where less transmitted information between nodes leads to less consumed energy. Several general-purpose aggregation and indexing frameworks also use hierarchical aggregation. Astrolabe [82] uses a gossip protocol inside a user-defined hierarchy, with arbitrary aggregation functions. In [38, 74], the authors describe a generic aggregation protocol for network management purposes. It creates a tree structure on top of an overlay network, and aggregates network state variables – bandwidth, delay, number of nodes – using operators like sum, average and min/max. Mortar [65] focuses on data stream management. For each query, it builds several static tree overlays to provide resilience and faster aggregation. It is also very common to build a virtual tree on top of a more resilient structure, like a DHT. That is the case of SDIMS [96], which improves the tree performance by treating read-dominated and write-dominated parameters separately. These works inspired our aggregation scheme, but there are important differences. We are interested in the individual state of each execution node. For this reason, the aggregation is not actually performed until enough samples are collected. Then, a clustering algorithm selects which samples should be aggregated together. This scheme prioritizes the accuracy of the aggregation in the lowest levels, where the forwarding algorithm needs to be more precise.

2.3. Hierarchical Overlays

The use of a hierarchical overlay network is important in our design. Both the aggregation scheme and the forwarding algorithm assume that nodes are organized in a binary tree (see Section 3.1.3). We also expect the overlay to recover from node failures. Many works propose fault-tolerant hierarchical overlays. NodeWiz builds a k-d-tree in which nodes are sorted by their properties. Each branch divides the search space in two parts by one of the properties. Its objective is to maintain the tree and the load of its nodes well-balanced. It also considers several failure situations and their solution.

P-Grid [2] builds a virtual trie structure on top of a DHT, for storage and search of data. By using self-organization, the tree balances its load even with non-uniform key distributions. However, it does not take fault-tolerance into account. VBI-Tree [54] builds a virtual balanced tree on top of a DHT. It uses multi-dimensional indexing that allows range queries to be performed with cost $O(\log_2 N)$. There are references to failure management, but its mechanisms are not properly described. Finally, Caron et al. [29] contribute a distributed lexical placement table (DLPT), which is a trie that allows exact, partial and range queries. In this case, it uses replication to provide resilience.

Chapter 3.

A Common Scheduling Model for Many Policies

"All models are false but some models are useful."
— George E. P. Box

In this chapter we present the details of STaRS. It is an online distributed scheduling model, customizable with different policies:

- **Distributed scheduling:** STaRS receives request from users for the execution of applications. Each application consists of a set of tasks, and STaRS allocates them to many, independent execution nodes.
- **Online:** Applications are allocated as they arrive to the system, they are not known in advance.
- **Different policies:** A scheduling policy defines the objective of the allocation of tasks: fulfill certain requirements, finish them as soon as possible, provide fairness to users, etc. It is possible to implement different policies on STaRS with a common architecture.
- **Model:** STaRS is a model that describes a set of tools, protocols and algorithms. When implemented on a distributed scheduling platform, it provides with scalability, fault-tolerance and versatility.

STaRS tries to fill a gap. It simultaneously provides with the scalability, fault-tolerance and versatility properties that most distributed computing platforms lack. Scalability, as its ability to manage as many execution nodes, users and applications as needed. Fault-tolerance, as its ability to gracefully degrade its performance when part of its components fail, and later recover. Versatility, as its ability to schedule different application types, with different objectives and in different environments. We have seen no other distributed computing platform that provides all of them at the same time.

To achieve these goals, we build up STaRS on the principles of decentralization and partial knowledge. In this way, we avoid the bottleneck and single point of failure of a centralized design, and no algorithm needs to know the full state of the system at once. Then, we propose three generic tools: a data model to represent the availability of execution nodes; an aggregation scheme that propagates the availability information on a tree-based overlay network, and that can be tuned to find a tradeoff between its accuracy and cost; and a forwarding algorithm that, using that information, routes tasks towards the most suitable execution nodes, performing stateless resource discovery and allocation

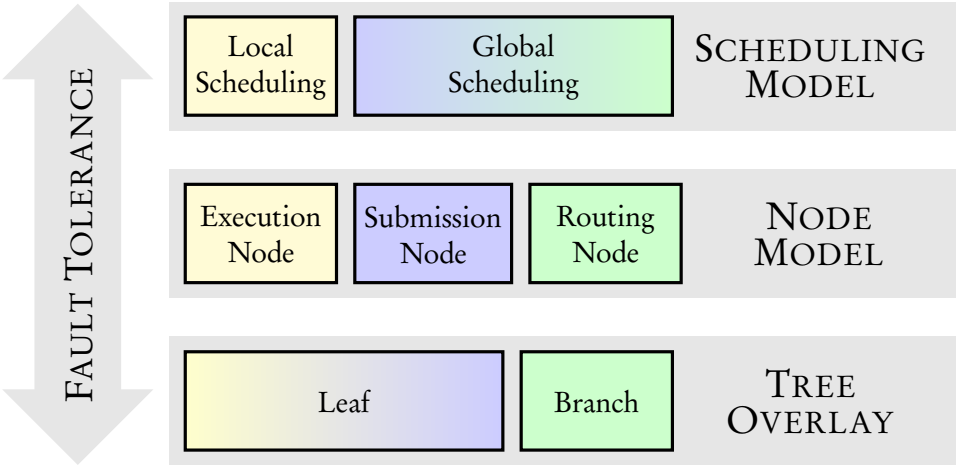

Figure 3.1.: STaRS architecture. The scheduling model is divided into a local and a global part. Execution nodes provide the local scheduling, while submission and routing nodes provide the global one. Execution and submission nodes are placed at the leaves of a tree-based overlay, while routing nodes occupy the branches.

simultaneously. Many policies can be implemented by providing a suitable instantiation of these three elements. In this chapter we present an example policy, the *Idle/Busy Policy* (IBP), that allocates tasks to idle nodes.

In order to facilitate the reading of this and the following chapters, Appendix A contains a brief reference of the most common notation used throughout it. It also presents the dot notation used in algorithms to refer to the fields of compound values. Nevertheless, notation is further detailed as it is used in the thesis. In particular, the notation that only appears in certain chapters.

3.1. Architecture Overview

Figure 3.1 presents STaRS architecture. From top to bottom, the scheduling model consists of a local and a global part. The local scheduling manages the task execution, while the global scheduling manages the resource discovery and task allocation. The functionality of each part is implemented through the node model. Each physical node may play up to three roles: Execution nodes provide the local scheduling, while submission and routing nodes provide the global scheduling. These nodes are organized in a logical tree-based overlay, so that routing nodes occupy the branches and execution and submission nodes hang from the leaves. Finally, the fault tolerance is taken into account at every level. The tree overlay must recover from physical node failures. Then, the node roles that a failed physical node was playing can be reassigned to another one, and the scheduling model can maintain its functionality, with a proportional degradation if needed.

3.1.1. Scheduling Model

STaRS presents a common scheduling architecture for different policies. Among other things, the policy defines the type of application that is being scheduled. We have mainly focused on policies for

bag-of-tasks applications, but other kinds of application could be accepted by the system. For instance, the workflow of tasks is commonly found in many scientific applications. Chapter 6 presents a policy for workflow applications.

A bag-of-tasks application A_i consists of a set of n_i independent, identical tasks. Additionally, depending on the policy being used, an application may have other parameters that describe the properties and requirements of its tasks. Common ones are the length of the tasks a_i, measured in millions of floating point operations (FLOPs), and their required memory and disk space, m_i and d_i, measured in megabytes. Let $\mathbb{PR}_i = (n_i, a_i, m_i, d_i, \ldots)$ be the tuple of parameters that describes application A_i. Then, a user submits an applications to the system as an *application scheduling request* containing $\mathbb{PR}_i$. This kind of configuration is very common in distributed scheduling platforms due to its high degree of parallelism.

The scheduling model is divided into a local and global part. The former describes the local scheduler and its policy. The local scheduler of a node manages a queue with the received remote tasks. The local policy decides whether new tasks can be accepted into the queue, and their order of execution. Common examples of local policies are First Come First Served (FCFS) or Earliest Deadline First (EDF).

The global part describes the availability information, the aggregation scheme and the forwarding algorithm. They are deeply discussed in Sections 3.2 and 3.3. Every local scheduler periodically calculates its availability to execute different types of applications, and exports this information. The aggregation scheme distributes it among the nodes of the system in a hierarchical fashion. It is then used by the forwarding algorithm to route application scheduling requests towards the most suitable execution nodes. The global scheduling policy determines the implementation of these three elements.

Naturally, the global and local policies must match. For instance, a global policy which tries to fulfill application deadlines will be used along with an EDF local policy. Thus, throughout this thesis we refer to both of them as just the scheduling policy, without distinction. Five common policies are presented in the following chapters.

3.1.2. Node Roles

Every physical node P_u of the system plays three node roles that provide the scheduling model functionality:

- The *execution node* E_u contains a local scheduler and an execution environment. The execution environment provides the platform-dependent mechanisms for the safe execution of remote tasks.
- The *submission node* S_u is the interface between the user and the platform. It manages the submission of application requests, and monitors the activity of any remote task that has been successfully allocated to an execution node.
- The *routing node* R_u is the component that implements the global scheduling policy rules. First, it receives the availability information from the execution nodes and distributes it to its neighbors. Then, it forwards application requests towards the most suitable execution nodes using that information.

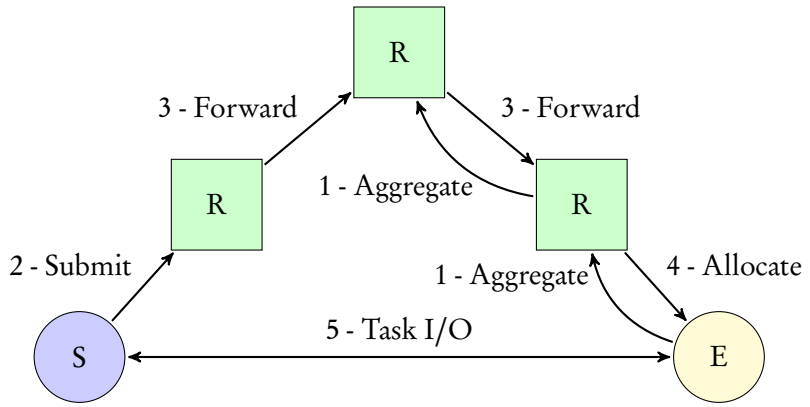

Figure 3.2.: Interactions between the node roles. First, availability information from execution nodes is aggregated by routing nodes. Then, submission nodes issue a request, that is routed to the execution nodes. When tasks are allocated to execution nodes, they communicate directly with the source submission node.

These roles are organized in a hierarchical fashion (Figure 3.2). The local scheduler at each execution node sends its availability information to its father routing node. Routing nodes aggregate this information through the tree. When a user wants to get an application executed, it instructs a submission node to issue an application scheduling request. This request is forwarded by the routing nodes towards the most suitable execution nodes, using the aggregated information. Execution nodes are allocated as they are found, in a single stage. From then on, task communication is performed directly between the execution node and the source submission node.

Usually, all physical nodes play these three roles in order to balance the load among them, but other configurations may also be interesting, e.g. only some nodes playing the submission node role in a dedicated cluster. The roles played by the same physical node are placed independently within the overlay. This provides great flexibility by allowing the relocation of any of the roles of a node without affecting the other ones.

The execution environment at each execution node E_u isolates remote tasks from the rest of the system. A virtual machine is a straightforward implementation, but more lightweight ones can be found in some platforms (e.g. Linux containers). This is done mainly for security reasons, to prevent remote tasks from abusing the host computers. It also allows the node owner to arbitrarily limit the type and amount of resources that remote tasks may use. These values are often used to calculate the availability of an execution node, most common ones being:

- The computational power s_u, measured in millions of FLOPs per second.
- The available memory reserved for the execution of remote tasks M_u, measured in megabytes.
- The available disk space D_u, measured in megabytes too.

In order to be able to execute every remote task under the same conditions, we assume that the environment disallows preemption. This feature is commonly found in existing distributed computing platforms, since it prevents preempted tasks from consuming memory and disk space to store their

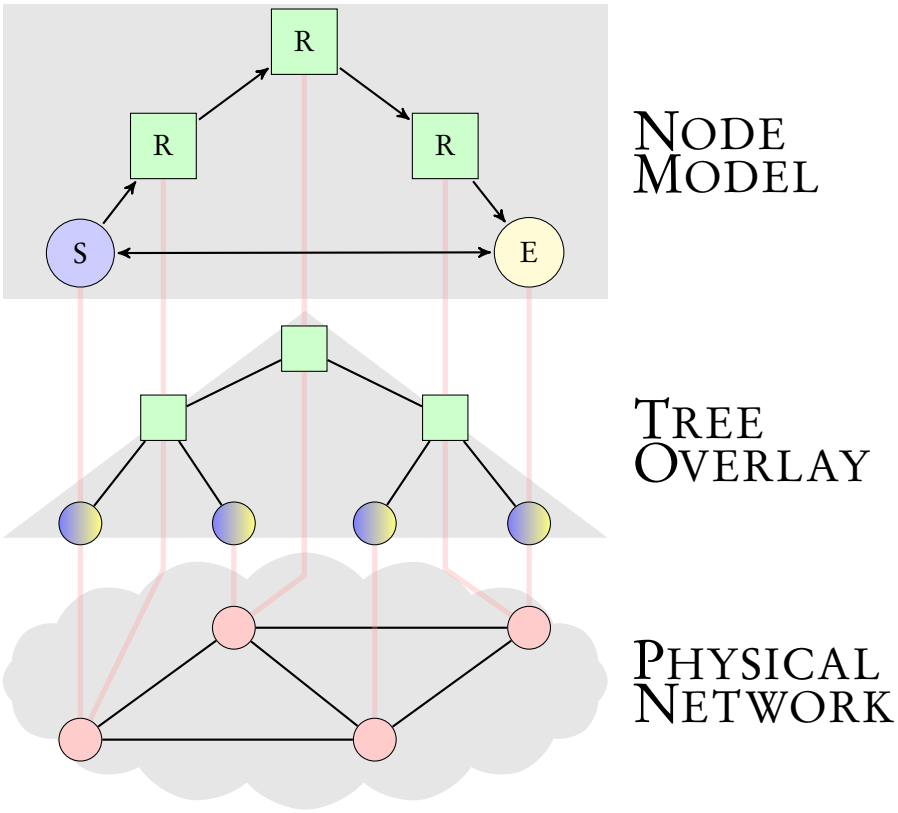

Figure 3.3.: Mapping of the different node roles. Every physical node plays as a leaf and a branch node in the tree. On top of them, execution and submission nodes are mapped to the leaves, and routing nodes are mapped to the branches.

paused state. So, at each execution node, there is only one running task at a time, which either finishes or is aborted.

3.1.3. Overlay Structure

STaRS assumes the existence of an underlying overlay network with a *balanced binary tree* organization. In fact, it assumes a design similar to VBI-Tree [54]. It is a search tree that differentiates branches from leaves. Leaf nodes contains the resources that the system looks for. Branch nodes route discovery requests to find the matching resources. Physical nodes play both roles, so that they can contain resources and route requests at the same time. In our case, physical nodes play the routing node role at the branches and the execution and submission node roles at the leaves (Figure 3.3). Every physical node may play all three roles in different positions of the tree, mutually independent from each other.

The tree is balanced so that the management, search and distribution of data operations are performed with cost $O(\log_2 N)$. This provides very good scalability properties to the system. The tree is binary because, from our experience in [32], it has a better tradeoff between computational cost and tree height than higher-degree trees. On one hand, at each node, the network traffic and the cost of most core algorithms are proportional to the number of children: maintain tree links, receive

and aggregate children information, route tasks to child nodes, etc. On the other hand, increasing the tree degree decreases the tree height, and so it shortens the path between any two nodes. This reduces the time of two important processes: routing a request to the execution nodes and distributing the availability information through the tree. However, for every tree degree over 3, while the computational cost increases by a factor of k, the tree height is reduced by a factor of less than k. So, there is no performance benefit if we increase the tree degree over 3. Then, it is much simpler to manage a binary tree than a 3-degree tree, so the binary option is preferred.

We also assume that nodes are ordered in the tree, grouping nearby execution nodes under the same branch. By doing so, certain locality information is provided to the forwarding algorithm in order to look first for execution nodes nearer to the submission node. This can be accomplished with topology-aware overlay construction, like in [81]. However, it does not meaningfully contribute to the evaluation of STaRS, so we use the simpler method of sorting nodes by network address in our tests. In order to bootstrap the overlay, we propose using a classical approach in decentralized systems, where a set of well-known nodes are used as a persistent entry point.

We proposed a first design of such an overlay in [32]. It provided the methods to build the tree structure and expose it to the scheduling components, but it had limited fault-tolerance capabilities. Later, two master thesis [31, 67] have been carried out to overcome these limitations, taking two different approaches. The first one backs up the tree structure on a DHT, and the second uses redundant links between nodes. They show the feasibility of building a scalable and fault-tolerant tree-based overlay, being able to successfully recover from multiple node failures, but they still need further development. For this reason, and since this thesis is focused on the scheduling part, the overlay behavior is simulated in the experiments.

3.1.4. Fault Tolerance

The management of faults in STaRS takes a best-effort approach. It tries its best to allocate and finish applications, but failures are admissible and should be expected by users. A failed node can make requests reach no execution node, loose the availability information of its branch and abort the tasks it was executing. Thus, every level of the model must consider resilience to faults.

First, the overlay network must be fault-tolerant in order to provide a reliable structure. Several peer-to-peer overlays already exist that construct a tree structure with good scalability and fault-tolerance properties [12, 54, 2]. Additionally, we have the experience of the two master thesis commented in the previous section. All of them prevent the top levels of the tree from turning into a bottleneck and a single point of failure. So, in our tests, we assume that the overlay is able to recover from node failures, and we evaluate the impact of the recovery in the scheduling performance. We do not evaluate the cost of the recovery, as the authors of each referred overlay have already done it.

For failures in the node and scheduling models, we treat routing, submission and execution nodes independently. Routing node failures affect the aggregation scheme and the forwarding algorithm. A failed routing node looses the availability information aggregated from its branch. The availability information is distributed in a reactive way, it is sent whenever a routing node modifies its information or a link with a neighbor changes. Likewise, when a routing node detects that a neighbor has failed, it sends an update as soon as the link is recovered. Any missing information is quickly rebuilt.

A failed routing node also disconnects its branch from the rest of the tree. Scheduling requests cannot jump from one side to the other until the node recovers, so we assume that this will impact

the scheduling performance temporarily. However, some requests may get lost as a result of a routing node failure. To cope with it, submission nodes set timeouts to detect tasks that have not been successfully allocated in a reasonable period of time. Then, they retry sending those tasks in a new request.

Submission nodes deal with their own failures by saving the state of submitted applications to a database. It contains pending requests, allocated tasks and their current status – queued, running or finished. All the information about ongoing applications is reloaded as soon as the user restarts the software.

On the contrary, execution nodes do not save their state. When they fail, all the information about queued tasks is lost. This is done in this way because we do not known how much time an execution node will remain off. It is better to resubmit the lost tasks to different nodes. Execution nodes send heartbeat messages periodically to all the submission nodes for which they have a task in their queue. When a submission node does not receive three heartbeats in a row from an execution node, it assumes that it has failed and resubmits its allocated tasks in a new request. Currently, failed tasks must restart their execution once they are allocated to a new execution node. Well-known mechanisms could be easily implemented to avoid this, like periodically checkpointing task state or sending several replicas of each task.

In Section 8.5, we measure how failures actually impact the scheduling performance. We analyze two situations: the simultaneous failure of a group of nodes, with different group sizes; and several degrees of *churn*, which is the rate of nodes leaving the network over a period of time.

3.2. Availability Information Management

The availability information is the glue between the local schedulers and the forwarding algorithm. Local schedulers use it to describe their availability to accept new tasks. Then, routing nodes aggregate this information, so that it approximates the availability of the execution nodes in their branch. In this way, routing nodes can decide where to forward the scheduling requests they receive.

In this section we give a generic pattern for the availability information representation and its aggregation scheme. Each policy must specialize it with the node parameters and application requirements that it considers relevant. We illustrate this with a simple policy, the Idle/Busy Policy (IBP). We firstly introduced the IBP policy in [32]. An execution node may only accept one task at a time, so its state is either "idle" or "busy". Then, a task of application A_i that needs m_i memory and d_i disk space for its execution will only be accepted by an idle node with enough available memory and disk space. This configuration may be appropriate in scenarios where very little information is obtained from execution nodes. For example, in projects of volunteer computing it is very difficult or even impossible to estimate the availability of execution nodes with any useful precision.

3.2.1. Describing Node Availability

The term "availability" is very vague. It refers to the possibility of an execution node of accepting new tasks, given the objective of the policy. For instance, in the IBP policy, a busy node would have zero availability, but we need to decide which of two any idle nodes is more available. To solve this problem, we define a set of tools, common to all the policies, that *measure* the availability of the

execution nodes. The first one is the *availability function*, a primitive that describes the availability of an execution node to accept tasks of a certain application, defined as follows:

Definition 1. The *availability function* $\mathbb{AF}_u(\mathbb{PR}_i)$ describes the availability of execution node E_u at the current time, as the number of tasks of application A_i, with its parameters $\mathbb{PR}_i$, that E_u is able to execute.

The actual implementation of $\mathbb{AF}_u$ depends on the scheduling policy being used. It will take into account the parameters of application A_i and the corresponding node resources. In the IBP policy, the availability function for E_u would be $\mathbb{AF}_u(m_i, d_i)$. For any pair of values (m_i, d_i) such that $m_i \leq M_u$ *and* $d_i \leq D_u$, the function would return a value of 1, and 0 otherwise. As the definition states, it is also usual to include the current time if any aspect of the node availability depends on it, like queue state.

When receiving a request at a routing node, the forwarding algorithm must calculate the availability function of the execution nodes in its branch, to decide how to distribute the tasks among them. With the application parameters $\mathbb{PR}_i$ in the request, it needs the node resources. Execution nodes send this information up the tree, providing their state to the forwarding algorithm. For this purpose, the availability function is first discretized into a *sampled function*.

Definition 2. A *sampled function* f_u for node E_u is a tuple of samples of the availability of node E_u, for each of its resources, that is used to compute its availability function.

These samples, together with the parameters of a newly submitted application, are used to approximate the value of the availability function by interpolation. This representation is very flexible since its size is $O(kl)$, where k is the number of resources of E_u and l is the number of samples per resource. By increasing k and l, the approximation will be more accurate, but the sampled function will also be larger, so a tradeoff must be found. On one hand, the information reported to the routing nodes is used by the forwarding algorithm to look for a certain scheduling objective, so improving its accuracy directly affects the scheduling performance. On the other hand, the data sent between nodes must be compact in order to save network and computational resources. This tradeoff looks for a good *performance and scalability balance*.

The method to determine which samples are useful, and how the approximation of the availability function is actually done – e.g. linear interpolation –, heavily depends on the nature of the function and its parameters. In the IBP policy, only a single sample is needed for each node resource, so its sampled function is the tuple (M_u, D_u). The availability function can be computed comparing the application parameters m_i and d_i with the samples in the tuple, as explained before. Each policy must then define its own representation of the availability, using the concepts of availability function and sampled function. In the following chapters, more elaborate policies are analyzed.

3.2.2. Availability Aggregation Scheme

The sampled functions of execution nodes are propagated to the routing nodes in *availability summaries*, which are just sets of sampled functions. Summaries are sent from children to fathers. After receiving new information, a routing node builds the summary of its branch as the union of the summaries of its children. That is the information needed to allocate tasks to execution nodes in that

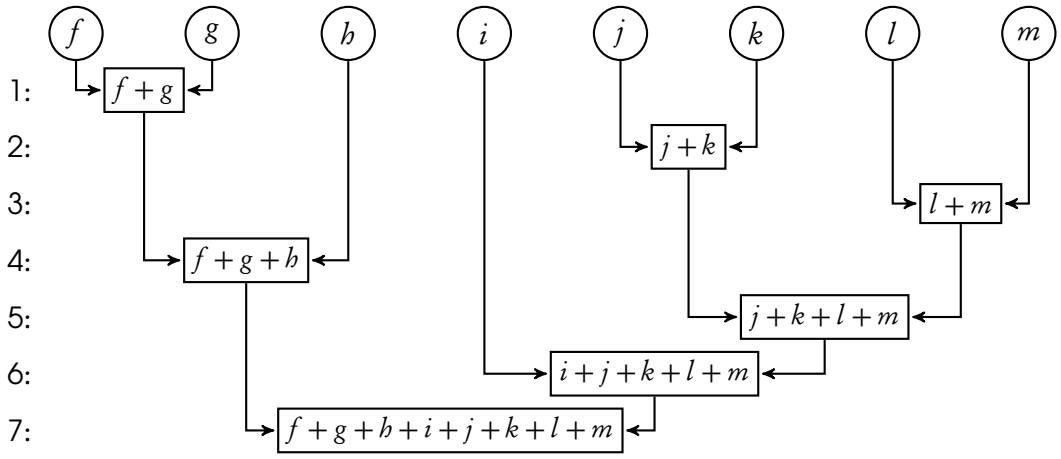

Figure 3.4.: Hierarchical agglomerative clustering of sampled functions. At each step, the two most similar functions are joined together.

branch. However, the availability summary size grows exponentially as we climb up the tree, so this solution does not scale over a few tree levels.

To limit the amount of information distributed through the network, and to the top levels in particular, we use a *tree-based aggregation scheme* [66]. Such scheme reduces the number of sampled functions in the summary of a branch, but maintains a good approximation. When the size of a summary is over a certain maximum SF_{max}, we use a *hierarchical agglomerative clustering algorithm* [55]. As depicted in Figure 3.4, it iteratively sums up the two most similar functions into one, until the number of sampled functions reaches the maximum again. While being suboptimal, it provides a good tradeoff between cost and accuracy. Note that, unlike other tree-based aggregation schemes, that look for a single value that summarizes the global state, we look for the individual states of the execution nodes. By delaying the aggregation until the availability summary reaches a certain size, and then applying a clustering algorithm, we can control how accurately the resulting information approximates the actual availability of the execution nodes.

We think that a hierarchical agglomerative clustering algorithm is better suited than other agglomerative and partitional algorithms. Their details can be found in [55]. All of them start with a set of sampled functions and iteratively reduce them to a certain number of representatives. At each step they provide a partial solution. To calculate this partial solution, a hierarchical algorithm only uses the result of the previous step and reduces the number of sampled function by one. Other agglomerative and partitional algorithms need the original set of functions at every step. For this reason, with a hierarchical algorithm we can cluster the availability information at any level of the tree, even if it has already been clustered before. The original set of functions is not passed up the tree once the availability information is clustered for the first time, so other agglomerative and partitional algorithms produce less accurate results, or may not be applicable at all.

The Sum and the Distance

We just said that the clustering algorithm sums up similar sampled functions. From the definition of availability function, it is easy to see that the number of tasks that could be executed by a pair of

nodes is calculated as the sum of their availability functions. So we define the *sum* of two sampled functions as:

Definition 3. The sum of sampled functions f_1 and f_2 (expressed as $\text{SUM}(f, g)$), that are used to compute functions $\mathbb{AF}_1$ and $\mathbb{AF}_2$ respectively, is the sampled function that is used to compute $\mathbb{AF}_1 + \mathbb{AF}_2$.

With the associative property, this definition is extended to the sum of any number of sampled functions. It follows that a sampled function is able to describe the availability of a set of nodes, not just one. So, we also extend the definition of sampled function f to include an attribute $f.v$ for the number of nodes it describes. When an execution node provides a sampled function for its availability, it sets $f.v = 1$. Then, for $h = \text{SUM}(f, g)$, we have that $h.v = f.v + g.v$.

Obviously, how the sum operation is performed depends on the node resources the sampled function contains and how they are represented. For instance, the sampled function of the IBP policy contains samples for two resources and the v attribute: (M, D, v). We can argue that, since we want to allocate tasks to nodes that have enough available memory and disk space, the sum operation of this policy should be

$$\text{SUM}(f_1, f_2) = \big(\min(f_1.M, f_2.M), \min(f_1.D, f_2.D), f_1.v + f_2.v\big). \tag{3.1}$$

Equation 3.1 expresses that there are $f_1.v + f_2.v$ nodes with at least $\min(f_1.M, f_2.M)$ memory and $\min(f_1.D, f_2.D)$ disk space. Then, tasks with lower requirements than those can be sent to any of the execution nodes of f_1 and f_2. However, it is easy to see that a single sampled function for a set of nodes must be less accurate than their respective set of functions. Assuming $f_2.M < f_1.M$, tasks with $f_2.M < m_i < f_1.M$ could be allocated to any node of f_1, but we would not know with the information in $\text{SUM}(f_1, f_2)$. The more $f_1.M$ and $f_2.M$ differ, the more accuracy is lost. Or, in general, the more similar the parameters of the sampled functions are, the less accuracy is lost in the sum operation. This can be measured with the *distance* operator.

Definition 4. The distance of sampled functions f_1 and f_2, expressed as $\text{DIST}(f_1, f_2)$, is a measure of the accuracy lost by representing the availability of their set of nodes with $\text{SUM}(f_1, f_2)$.

Again, the implementation of the distance operation depends on the policy. Usually, it is easy to obtain a measure of the accuracy lost for each resource in the sampled function, but we need to reduce them to a single value. The solution that has shown best results is to normalize these values and perform a weighted sum. With the weight of each resource, we control how important it is to meet each of the application requirements.

To improve the aggregation performance, the clustering algorithm sums up the two closest sampled functions, those that loose less accuracy at each iteration. In this way, it groups together sampled functions that represent nodes with similar available resources. With the sum and distance operations, Algorithm 3.1 shows the aggregation of two summaries x and y.

Clustering Scalar Parameters

Scalar parameters are the most common ones to appear in sampled functions. For instance, the IBP policy includes the available memory and disk space. Other examples are the queue length, the

Algorithm 3.1 Aggregation of two availability summaries.

Pre: x and y are summaries.
Post: z is the result of the aggregation of x and y.

```
1: function AGGREGATE(x, y)
2:   z ← x ⋃ y
3:   while |z| > SF_max do
4:     Let f, g ∈ z | DIST(f, g) ≤ DIST(i, j) ∀i, j ∈ z
5:     z ← (z \ {f, g}) ⋃ {SUM(f, g)}
6:   end while
7:   return z
8: end function
```

processor power and the network bandwidth. Therefore, we have developed an implementation of the sum and distance operations on sampled functions with scalar parameters. We use it extensively in our other policies.

First, consider a scalar parameter p for a generic resource. The value for node E_u is denoted as p_u, as usual, and a sampled function with that parameter is written as the tuple $f = (p, v)$. Such a function represents a set of $f.v$ execution nodes with an amount of resource p equal to $f.p$. It is clear that, for every node E_u such that $p_u \neq f.p$, we are introducing an error in the availability information. We want to:

- Be able to compare the error introduced by two different sampled functions.
- Minimize the total error: $\sum_{u=1}^{f.v} |f.p - p_u|$.
- Minimize the maximum error among all the nodes: $\max_{u=1}^{f.v} |f.p - p_u|$.

To accomplish this, we calculate the mean square error (MSE) of parameter p introduced by a sampled function. We can use this value to compare two sampled functions. For functions that represent the same number of nodes, the one with the lowest MSE is also the one with the lowest total error. Finally, for the same total error, the function with the lowest MSE is also the one with the lowest maximum error. So, we include a new attribute mse_p in the sampled function:

$$f.mse_p = \frac{1}{f.v} \sum_{u=1}^{f.v} (f.p - p_u)^2. \tag{3.2}$$

The MSE is 0 for the sampled function f_u that an execution node creates to represent its own availability, since $f_u.p = p_u$. We now explain how the sum operation calculates the MSE of the resulting sampled function. Let f and g be the sampled functions that result from adding functions f_i, $1 \leq i \leq f.v$, and g_j, $1 \leq j \leq g.v$, respectively. Their MSE of parameter p are

$$f.mse_p = \frac{1}{f.v} \sum_{i=1}^{f.v} (f.p - f_i.p)^2, \qquad g.mse_p = \frac{1}{g.v} \sum_{j=1}^{g.v} (g.p - g_j.p)^2.$$

We want to calculate $h = \text{SUM}(f, g)$ only with the information of f and g, without needing to know anything about functions f_i and g_j. We already showed that $h.v = f.v + g.v$ and that $h.p$ can be calculated with a simple operation on $f.p$ and $g.p$ (like the minimum, maximum, average, etc). The attribute $h.mse_p$ is then

$$h.mse_p = \frac{1}{h.v}\left(\sum_{i=1}^{f.v}(h.p - f_i.p)^2 + \sum_{j=1}^{g.v}(h.p - g_j.p)^2\right). \tag{3.3}$$

If we take the term $(h.p - f_1.p)^2$ from (3.3) and add and subtract $f.p$, we obtain

$$\begin{aligned}(h.p - f_1.p)^2 &= (h.p - f.p + f.p - f_1.p)^2 = \\ &= (h.p - f.p)^2 + (f.p - f_1.p)^2 + 2(h.p - f.p)(f.p - f_1.p).\end{aligned} \tag{3.4}$$

Repeating (3.4) for all the terms on f_i we get that

$$\begin{aligned}\sum_{i=1}^{f.v}(h.p - f_i.p)^2 &= f.v(h.p - f.p)^2 + \sum_{i=1}^{f.v}(f.p - f_i.p)^2 + 2(h.p - f.p)\sum_{i=1}^{f.v}(f.p - f_i.p) = \\ &= f.v\Big((h.p - f.p)^2 + f.mse_p\Big) + 2(h.p - f.p)\sum_{i=1}^{f.v}(f.p - f_i.p).\end{aligned} \tag{3.5}$$

The same applies to the terms on g_j. Note that, at this point, we are able to compute $h.p - f.p$ only with the information of f, but we still need $\sum_{i=1}^{f.v}(f.p - f_i.p)$. Let it be called the *linear term* of parameter p in f. We can incorporate it to the sampled function too, in an attribute called lt_p. So far, a sampled function with one scalar parameter p would contain the values (p, mse_p, lt_p, v). Then, from (3.3) and (3.5), the sum of f and g would obtain h where

$$\begin{aligned}h.mse_p = \frac{1}{h.v}\Big[&f.v\Big((h.p - f.p)^2 + f.mse_p\Big) + g.v\Big((h.p - g.p)^2 + g.mse_p\Big) + \\ &+ 2(h.p - f.p)f.lt_p + 2(h.p - g.p)g.lt_p\Big].\end{aligned} \tag{3.6}$$

And the same procedure is used to calculate $h.lt_p$:

$$\begin{aligned}h.lt_p &= \sum_{i=1}^{f.v}(h.p - f_i.p) + \sum_{j=1}^{g.v}(h.p - g_j.p) = \\ &= f.v(h.p - f.p) + \sum_{i=1}^{f.v}(f.p - f_i.p) + g.v(h.p - g.p) + \sum_{j=1}^{g.v}(g.p - g_j.p) = \\ &= f.v(h.p - f.p) + g.v(h.p - g.p) + f.lt_p + g.lt_p.\end{aligned} \tag{3.7}$$

Now, the clustering algorithm is able to take the two sampled functions that produce the minimum MSE when summed up. With one scalar parameter, the distance operation of two sampled functions is straightforward: it just returns the MSE of their sum. But with more than one scalar parameter, we have many MSE values as the result of the sum operation, one for each parameter. As we explained before, the best solution is to calculate the distance as a weighted sum of the normalized MSE for each parameter. That is, for n scalar parameters p_1 through p_n,

$$\text{DIST}(f,g) = \sum_{i=1}^{n} \alpha_i \text{NORM}_i(\text{SUM}(f,g).mse_{p_i}). \tag{3.8}$$

The α_i coefficients represent the weight of each parameter in the distance operation. Usually, they will all have the same value, but in some situations it is interesting to give more importance to some of the parameters over the rest. For the IBP policy, we have decided that both memory and disk space have the same weight. Then, the normalization function NORM_i that appears in (3.8) maps the MSE of parameter p_i to the $[0,1]$ range. Let $\mathbb{B}$ be the set of all the execution nodes of the current branch, whose information is being clustered, then

$$\text{NORM}_i(m) = \frac{m}{\left(\max\limits_{E\in\mathbb{B}}(E.p_i) - \min\limits_{E\in\mathbb{B}}(E.p_i)\right)^2}. \tag{3.9}$$

Values $\max_{E\in\mathbb{B}}(E.p_i)$ and $\min_{E\in\mathbb{B}}(E.p_i)$ must be known at each routing node, so they are also aggregated. An availability summary contains a maximum and a minimum value for each parameter p_i. Execution nodes initialize them with their own value. When two summaries are aggregated, the resulting summary simply gets the maximum of the maximums and the minimum of the minimums. So, the cost in time and space is very little. If the maximum and minimum values for any parameter are equal, the sum operation cannot make an error on that parameter, and it is not taken into account in the linear combination.

3.2.3. Distributing the Availability Information

The aggregation of the availability summaries is done in a hierarchical fashion. Execution nodes send a summary to their father routing node with a single sampled function, representing their own availability. Then, routing nodes aggregate the summaries from both children, and send the result further up the tree. This scheme is reactive: whenever execution nodes change their availability or routing nodes receive new information, the aggregation mechanism is triggered.

Execution node availability changes periodically. For some policies, it may change only every time a task starts or finishes. In other cases, it may be frequently changing, since time can be an important factor of the execution node state. For this reason, the availability information must be kept up to date at the routing nodes. However, if many execution nodes change at the same time, a cascading effect will flood the upper levels with update summaries, rendering the system unusable. To provide scalability and reliability, this problem must be faced.

We propose two solutions. The first one is implicit to the aggregation scheme. Routing nodes periodically receive updated information from their children nodes, aggregate it and store the result as the information of their branch. But when previous information exists, it is compared with the

new one. If they are equal, it is not reported to the father, since it is the same it already has. This may happen when a child node sends a similar summary to the one it sent before, due to the way in which the sum of sampled functions is performed. If it uses operations like the minimum or maximum, as in the IBP example policy, little variations will probably produce the same result. So, a summary going up the tree may stop before reaching the root when the availability changes lightly.

The second measure is to limit the bandwidth used to send summaries. Routing nodes insert a short delay after sending each summary to keep the average used bandwidth under an arbitrary maximum. After the delay expires, only the most recently aggregated summary is sent. This can be done with the availability information because each new summary makes the previous ones obsolete. This bandwidth limit, along with the size of the availability summary, is a tradeoff between the traffic supported by nodes and the time needed by a change in the leaves to reach the root of the tree. The lower the bandwidth limit, the higher the period between two summaries are sent. This results in the availability information being out of date more often. As we will show, this impacts the performance of some policies that are more sensible to availability changes.

3.3. Task Routing

Task routing is the process of forwarding the tasks in an application scheduling request towards the most suitable nodes, given its parameters. We call it "task routing" because it resembles the routing of packets in a computer network. A network router uses the forwarding table to decide in which direction to send a packet. Our forwarding algorithm uses the availability information to decide in which direction a request may reach the best execution node for each of its tasks.

The task routing process starts when a submission node S_u issues a new application scheduling request *req* for *req.n* tasks of application A_i. As we explained in Section 3.1.1, we focus on bag-of-tasks applications. For such an application, a request contains the application properties $\mathbb{PR}_i$, the address of the requester S_u, a request identifier and an interval of task identifiers $[1, req.n]$. All the tasks in a bag-of-tasks application are identical, so we can group them in an interval instead of enumerating each of them. Then, routing nodes invoke the forwarding algorithm on the requests they receive.

3.3.1. Forwarding Algorithm

Using the availability information, the forwarding algorithm decides in which direction it sends each task. So, it ends up sending n_l tasks to the left child, n_r tasks to the right child and n_f tasks to the father, so that $n_l + n_r + n_f = req.n$. With these values, it creates three new requests with the same application properties, request identifier and requester address as the original. The request for the left child will contain the task identifiers in $[1, n_l]$; the one for the right child, task identifiers in $[n_l + 1, n_l + n_r]$; and the one for the father, task identifiers in $[n_l + n_r + 1, req.n]$. The resulting requests are sent in the corresponding direction, as long as they carry at least one task. However, if there is no father because the current routing node is the root, the n_f tasks that were meant for it are discarded. They are treated as a scheduling failure and will be resent by their submission node. The algorithm is stateless with regard to the requests it forwards: it needs to hold no record of them. This improves its scalability and fault tolerance.

Each policy must specialize the forwarding algorithm, as it specializes the availability information. Usually, a policy-dependent objective function sorts sampled functions by the suitability of allocating

Algorithm 3.2 Forwarding algorithm for the IBP policy.

Pre: R_u is this routing node. `request` is the request.
Post: `reqLeft`, `reqRight` and `reqFather` are the resulting requests to be sent to the left child, right child and father nodes, respectively.

```
1:  procedure FORWARD(request)
2:    availableSF ← ∅
3:    if request.srcAddr ≠ R_u.leftAddr then
4:      GETSF(R_u.leftInfo, request.ℙℝ, availableSF)
5:    end if
6:    if request.srcAddr ≠ R_u.rightAddr then
7:      GETSF(R_u.rightInfo, request.ℙℝ, availableSF)
8:    end if
9:    SORT(availableSF)                          ▷ Best nodes are allocated first.
10:   while ¬ISEMPTY(availableSF) ⋀ request.n > 0 do
11:     sf ← POPFRONT(availableSF)
12:     numTasks ← AF(sf, request.ℙℝ)
13:     if ISFROMLEFTCHILD(sf) then
14:       reqLeft ← EXTRACT(request, numTasks)
15:     else
16:       reqRight ← EXTRACT(request, numTasks)
17:     end if
18:   end while
19:   if request.n > 0 then
20:     reqFather ← request
21:   end if
22: end procedure
```

a task to one of their nodes. Then, tasks can be assigned to each child from the most to the least suited sampled function. Besides, in most policies, the forwarding algorithm also updates the availability information of the children branches, trying to estimate how it will change once the sent tasks are allocated. This avoids sending tasks of subsequent requests to the same execution nodes before their availability is updated through the aggregation scheme.

For instance, Algorithm 3.2 shows the forwarding algorithm of the IBP policy. Procedure GETSF fills the list `availableSF` with the sampled functions of those nodes that are able to execute tasks from the request. Then, this list is sorted to allocate first those nodes whose available memory and disk space are closest to the requirements. With this heuristic, nodes with more available resources are saved for future applications with higher requirements. Tasks that cannot be allocated in this branch are sent up to the father routing node.

3.3.2. Routing Patterns

Specializing the forwarding algorithm for each policy may produce several patterns. A common one is the IBP policy pattern, which assigns as many tasks as possible to the current branch and send the

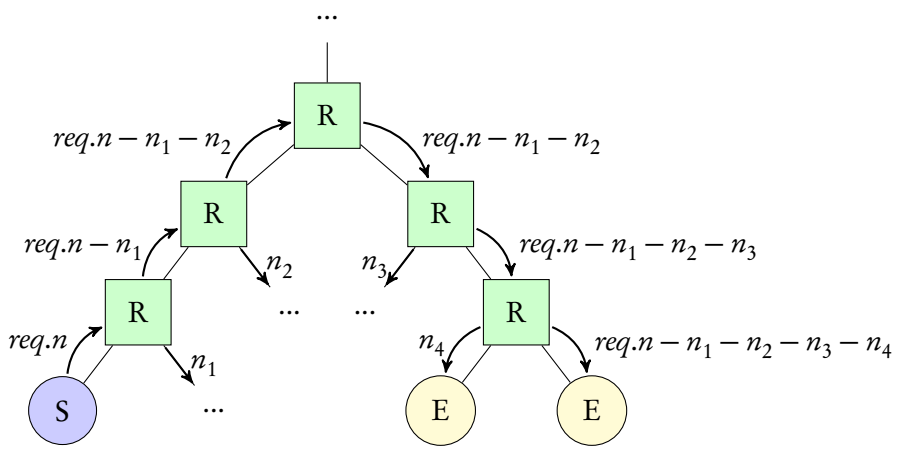

Figure 3.5.: Path followed by a request that is forwarded towards the execution nodes where it is allocated. At each routing node, it can be divided so that independent branches are explored concurrently.

rest to the father. At the next level, the forwarding algorithm assumes that it cannot send any more tasks in the direction the request came from, so it sends some tasks to the other one and the rest again to the father. But if the request comes from the father, all the tasks are sent to the children nodes. Figure 3.5 illustrates this pattern with an example. A request with K tasks is issued by a submission node from a leaf of the tree. At each routing node, the forwarding algorithm separates the n_i tasks that can be allocated to its branch, and the rest is sent up. The process continues until all tasks are allocated or all nodes are reached. A similar pattern consists in sending all the tasks to the father node as long as a certain criteria is not met. Then, send them down again, assigning tasks to both children.

But all the patterns have some advantages in common. The first one is that the task routing process looks for execution nodes in independent branches concurrently, so its cost depends on the number of network jumps of the longest path. In a balanced binary tree, this cost is $O(\log_2 \min(n_i, N))$ jumps, where n_i is the number of tasks to allocate and N is the size of the network. This cost scales very well with both parameters. Another one is that all the submissions originate in the leafs, and only climb up the tree until they discover enough execution nodes. Doing so, maintains most request traffic in the lower levels of the tree, and uses more accurate availability information. Furthermore, if the execution nodes are ordered in the tree by location, as suggested in Section 3.1.3, the discovered ones will be nearby the requester. This might be helpful when the applications have big input or output data.

Chapter 4.

MMP: Makespan Minimization Policy

"It's the job that's never started takes longest to finish."
— J. R. R. Tolkien

4.1. Makespan Scheduling

When the users have no other priority, one of the most common policies consists in trying to finish all the scheduled work in the platform as soon as possible. As shown in Figure 4.1, it is well known that this objective is accomplished when all the nodes finish at the same time. Otherwise, nodes that finish earlier would be able to do part of the work of the nodes that finish later. However, this is only feasible if tasks can be arbitrarily divided. In an heterogeneous environment as we consider, each execution node runs tasks at different speed and with an indivisible amount of computation. So, the objective of such a policy is to minimize the makespan: the maximum time needed by any node to finish its work.

This subject has been widely studied since long ago. The problem is usually divided into offline and online scheduling, and considering machines with both identical and different speeds. The optimal offline solution is NP-hard [46] in any case, but several polynomial time approximation schemes have been proposed. The Graham's well-known list scheduling algorithm [47] is $(2-1/m)$-competitive on m identical machines with linear cost. It sorts the list of jobs by priority and sends each one to the machine that has been assigned the least amount of work so far. Hochbaum and Shmoys [51] propose another solution with arbitrary relative error. On machines with different speed, Lenstra et al. [63] give a 2-competitive solution based on integer programming. On the other hand, online

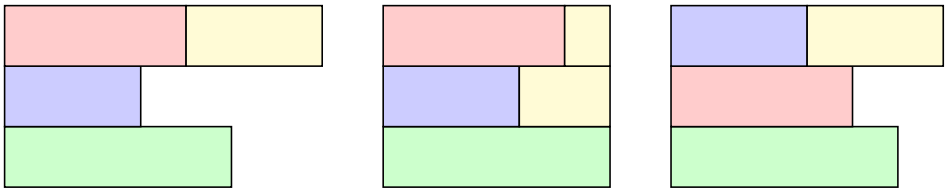

Figure 4.1.: The minimum makespan of a random schedule in (a) with two configurations: with (b) divisible tasks and (c) atomic tasks. The former is always shorter or equal to the later.

solutions deal with the problem of having to decide on each job before knowing about the next one. Fleischer and Wahl present MR [45], an online algorithm for identical machines, that reaches 1.9201-competitiveness, setting the current best upper bound. Englert et al. [42] propose a 2-competitive online algorithm for machines with different speeds, by buffering the incoming jobs and reordering them.

Here, we present the *Makespan Minimization Policy* (MMP) for STaRS. It is an online, decentralized policy to minimize the makespan among all the currently scheduled applications. In this policy, an execution node queue may hold as many tasks as needed, so the makespan is the maximum end time of the longest queue. Then, similarly to the list scheduling algorithm, tasks are routed to those nodes whose queue will remain the shortest after allocating them.

4.2. Local Policy

Unlike the IBP policy, the MMP local schedulers have an unlimited task queue. They accept every incoming task, as long as memory and disk space requirements are met. Then, tasks are inserted at the back of the queue, in FCFS order, because we assume that there is no specific application priority.

Since we want to limit the queue length by allocating a similar workload to every node, we introduce a constraint to the queue length in the availability function: it cannot be longer than a certain time. We define the availability function for this policy as $\mathbb{AF}_u(m_i, d_i, a_i, q_i)$. We recall that m_i, d_i and a_i are the required memory, required disk space and length of a task of application A_i, respectively. Then, the function returns the maximum number of tasks of application A_i that can be added to the queue of execution node E_u so that it finishes no later than time q_i. This function allows the forwarding algorithm to decide how many tasks can be sent to a node in order to increase its queue to a certain length. Let the queue end time of node E_u be Q_u, the availability function is calculated as

$$\mathbb{AF}_u(m_i, d_i, a_i, q_i) = \begin{cases} \left\lfloor \dfrac{(q_i - Q_u)s_u}{a_i} \right\rfloor & \text{if } m_i \leq M_u \bigwedge d_i \leq D_u \\ 0 & \text{otherwise,} \end{cases} \tag{4.1}$$

where we remind that s_u is the computational power of E_u. Note that Q_u is equal to the current time if the queue is empty.

4.2.1. Measuring the Execution Time

This policy is the first to use a key element in scheduling: the execution time of a task in a certain node. We assume that this time can be computed as a_i/s_u. However, measuring these two values raises several problems.

First, what unit should we use with them? We said in Section 3.1.1 that we measure a_i in millions of FLOPs, so s_u is measured in millions of FLOPs per second. While this is valid for a theoretical analysis, in practice it is less useful. The task length and the computing power are not comparable between different architectures because they include instructions of varying complexity. Besides, the execution of a task is also affected by several architecture-dependent delays that are difficult to predict: cache misses, failed branch predictions, data and control dependencies, etc.

The second question is how to estimate a_i and s_u. While the later may be easier to measure with benchmarks, the former has several implications. A few tasks may have a constant execution length, but most depend on the conditional branches and loops that the flow of execution follows. Furthermore, there should be an automatic method of estimating a_i without actually executing a task.

A possible solution would be to adapt the work by Dinda [40], who suggests estimating the execution time of a task in E_u with the nominal execution time in an unloaded, reference node, t_{nom}, and the load of E_u. To provide a good estimation, the load of E_u is constantly monitored and predicted. Then, we could treat a_i as the normalization of t_{nom} with the inverse speed of the reference node, and predict s_u on every node as Dinda predicts the load. Many other works on performance prediction propose methods to estimate the node availability [87, 56, 60]. Task length estimation methods are covered, for instance, by works on worst-case execution-time (WCET) estimation [94]. Unfortunately, these works show that the best results are obtained with sample executions of all or part of the code to measure.

For the sake of simplicity, we have used the millions of FLOPs as the task length unit to carry out the analysis and experiments of this thesis. It is realistic enough to evaluate the STaRS scheduling model and policies. Furthermore, we assume that node E_u has a constant dedicated computing power of s_u for the execution of remote tasks, unless it fails. In a real implementation, the previous considerations would have to be taken into account.

4.3. Global Policy

4.3.1. Availability Information

So, the properties needed to compute the availability function of a node consists of its memory, disk space, computing power and end queue time. Thus, in this policy, the sampled function for a set of nodes is (M, D, s, Q, v), with a sample for each of the previous node parameters plus the number of nodes v. The clustering of such sampled function uses the same scheme for scalar parameters as the IBP policy explained in Section 3.2.2.

We have also decided that the sum operation of f_1 and f_2 should return

$$\begin{aligned}\text{SUM}(f_1, f_2) = \big(&\min(f_1.M, f_2.M),\ \min(f_1.D, f_2.D),\ \min(f_1.s, f_2.s),\\ &\max(f_1.Q, f_2.Q),\ f_1.v + f_2.v\big).\end{aligned} \tag{4.2}$$

It calculates the minimum available memory and disk space that can be found in each node of f_1 and f_2, as usual. But it also gives the minimum speed a task is going to be executed at, and the maximum time a task is going to start at. This may be seen as an excessively conservative option. Unlike the memory and disk constraints, failing to estimate the speed or the queue length is not going to avoid the allocation of a task. In this case, it would make sense to adopt a more optimistic approach with the computing power and queue end time. This can be done calculating their weighted average as

$$\text{avg}_s = \frac{f_1.s f_1.v + f_2.s f_2.v}{f_1.v + f_2.v}, \qquad \text{avg}_Q = \frac{f_1.Q f_1.v + f_2.Q f_2.v}{f_1.v + f_2.v},$$

which would yield

$$\text{SUM}(f_1, f_2) = \big(\min(f_1.M, f_2.M),\ \min(f_1.D, f_2.D),\ \text{avg}_s,\ \text{avg}_Q,\ f_1.v + f_2.v\big).$$

With this approach, a sampled function advertises shorter queues and faster nodes than with the previous one. So, the forwarding algorithm is able to allocate more tasks to the set of nodes of a sampled function. This results in requests needing to climb less tree levels, thus reducing the network traffic. On the other hand, sending too many tasks may also increase the makespan. After simulating both approaches under the same conditions, we have concluded that this optimistic approach always produces a larger makespan without a noticeable decrease of network traffic. So, in the experiments of Section 8 we only use the conservative one.

Time is relevant in this policy, since the availability function depends on Q_u. So, the sampled function reported by each node must be updated when its queue end time changes. If the task queue is not empty, and tasks finish at their expected end time, Q_u does not change until a new task is pushed at the end of the queue. If the task queue is empty, Q_u is equal to the current time, so it changes continuously. However, if it is not updated, a routing node that finds a sampled function with Q_u earlier than current time will automatically deduce that its represented queues have become empty, and update it by itself. This reduces the update frequency.

4.3.2. Forwarding Algorithm

To minimize the global makespan, tasks must be sent to the nodes whose queue will remain shorter after allocating them. However, it is not enough to consider only the current branch. Routing nodes also need information about the queue end times in the rest of the tree. Without it, the routing can be performed by making all requests climb up the tree to the root node, but this solution would quickly flood the root with requests. On the other hand, routing nodes could send availability summaries to the children nodes, too. The summary sent to each child would come from the aggregation of the information obtained from the other child and the father. Then, every node would have information about all the tree. But also, each change in the availability would reach a much larger number of nodes. This solution would flood the network with availability summaries.

As a compromise between these two extremes, routing nodes use the *maximum queue end time* in the rest of the tree in the forwarding algorithm, along with the availability information of their children. They receive it from their fathers. They send to each child the maximum between the values coming from their father's and the other child's subtrees. Being a maximum, its propagation is usually bounded to just a few branches, so the traffic generated is negligible.

Algorithm 4.1 shows how the maximum queue end time is used in the forwarding algorithm. It starts by finding the expected makespan that will be obtained if all the tasks in the request are allocated. The availability function, with the information of the whole branch (R_u`.info`), returns the number of tasks that can be allocated for a certain makespan (`medMakespan`). The minimum expected makespan (`minMakespan`) is found with a binary search, until the availability function returns the same number of tasks the request contains. This new makespan cannot be longer than β times the longest makespan in the tree (R_u`.maxMakespan`), which is the difference between the maximum queue end time and the current time. If it is longer, and the request did not came from the father node, the request is sent to the father to look for a shorter makespan or until the root is reached. Otherwise,

Algorithm 4.1 Forwarding algorithm for the MMP policy.

Pre: R_u is this routing node. `request` is the request.
Post: `reqLeft`, `reqRight` and `reqFather` are the resulting requests to be sent to the left child, right child and father nodes, respectively.

```
1:  procedure FORWARD(request)
2:    minMakespan ← 0
3:    maxMakespan ← BIG_INT                    ▷ Start with a very big value for the maximum.
4:    while minMakespan ≤ maxMakespan + 1 do
5:      medMakespan ← (minMakespan + maxMakespan)/2
6:      numTasks ← AF(R_u.info, request.ℙℝ, medMakespan)
7:      if numTasks ≤ request.n then
8:        minMakespan ← medMakespan
9:      else
10:       maxMakespan ← medMakespan
11:     end if
12:   end while
13:   isTooMuch ← minMakespan > β R_u.maxMakespan
14:   if ¬ISROOT(R_u) ⋀ ¬FROMFATHER(request) ⋀ isTooMuch then
15:     reqFather ← request
16:   else
17:     numLeft ← AF(R_u.leftInfo, request.ℙℝ, minMakespan)
18:     EXTRACT(request, numLeft, reqLeft)
19:     reqRight ← request
20:   end if
21: end procedure
```

the availability function is used with the summary of the left child to calculate how many tasks can be sent to its branch, and the rest are sent to the right child. The β factor is a tradeoff between load balance and makespan minimization. The lower it is, the shorter the makespan will be, but the requests will usually climb more levels of the tree until they find a good set of nodes, so the upper levels will get more loaded. In Chapter 8 we empirically show that 0.5 is the best value.

The routing pattern of this policy is different from the one of the IBP policy. All the tasks are sent upwards until the expected makespan in the current branch gets shorter than a certain threshold. Then, they are distributed among all the nodes of that branch. In this way, the forwarding algorithm is capable of estimating a similar queue end time for all the nodes that will receive one of the tasks. Otherwise, these nodes could end up with very different queue end times, which is contrary to the objective of minimizing the makespan.

Chapter 5.

DP: A Policy for Applications with Deadlines

"Time is money."
— Benjamin Franklin

5.1. Applications with Time Requirements

Most users of a distributed scheduling platform expect their applications to finish as soon as possible. Sometimes, they want – or need – them to finish before a certain deadline. Ramamritham et al. [78] were among the first to propose the use of distributed algorithms to schedule tasks with time and resource restrictions. They give different algorithms for this purpose, and compare their performance. They claim that their solution is effective even in hard real-time environments. However, their approach requires each node to have full knowledge of the rest of the system, which naturally limits its scalability.

Later works propose using soft deadlines as a quality of service guarantee. For instance, meeting production or research deadlines to apply for more budget. Nimrod/G [4] is a scheduling platform on the Grid for parameter sweep applications. It supports additional constraints that can be found in such a heterogeneous environment as the Grid. In particular, it allows to set deadlines on the submitted jobs, and charges users with higher costs for tighter deadlines. In [20], Buyya et. al. optimize the relation between deadline and budget to provide the maximum quality of service at minimum cost. A similar approach is taken by Takefusa et. al. [89], who focus on scheduling jobs with deadlines and minimizing the overall number of missed deadlines.

All these works use several centralized services, like resource allocation managers, to discover and allocate nodes to tasks. Again, these centralized services limit their scalability. We overcome this problem with decentralized algorithms and overlay networks. Similarly, Cao et. al. [25] use a hierarchy of agents to coordinate a set of local grid schedulers. They use performance prediction to minimize the makespan and processor idle time, but also try to meet deadlines. However, they do not encourage an extensive use of the hierarchy to find the best candidate for a request, arguing that grid users prefer a satisfactory resource as fast and local as possible. To our knowledge, other decentralized and P2P-based computing platforms exist [13, 17, 98, 48, 58], but they do not include deadlines. In BOINC [8], deadlines are used only locally [10]. Volunteered nodes use them to decide the order of execution of the tasks coming from different projects. However, the availability of the nodes is not reported to the project servers and so it cannot be used to allocate tasks to nodes.

In this chapter, we present the *Deadline Policy* (DP). With it, we propose an alternative use of deadlines as a quality of service guarantee: to measure the time users are willing to wait for their jobs to finish. In order to find a relation between user behavior and job response time, Shmueli and Feitelson study different traces taken from real deployments [86, 44]. Their conclusion is that user satisfaction is directly related to short job response times, independently of their length. So, they design a scheduling policy whose objective is to minimize the job response time. Our approach consists in allowing users to provide information about their desired job response time to the scheduler in advance.

5.2. Local Policy

The local scheduling policy sorts tasks in EDF [64] order, so that the next task to execute is the one with the earliest deadline. A well-established property of EDF order is that if it cannot ensure that all the deadlines are met, then no other does. So, new tasks are only accepted if, when inserted in the queue in EDF order, all the tasks meet their deadline. However, since we forbid preemption, the running task always remains the first of the queue even if a new task has an earlier deadline.

It is important to state that the acceptance of a task is a contract between submission nodes and execution nodes. On one hand, submission nodes expect their accepted tasks to meet deadlines, unless failures occur. On the other hand, an execution node E_u expects an accepted task from application A_i to have a duration not longer than a_i/s_u. Otherwise, it could make other tasks in the queue miss their deadlines. For that reason, each node checks its task queue periodically, and tasks that run longer than expected are dropped to favor the rest. In this way, a tradeoff is needed to estimate task lengths. The less the estimation overshoots the real length of a set of tasks, the easier it will be to allocate them. But they may be eventually aborted if the estimation was actually lower than the real length.

5.2.1. Availability Function

In the DP policy, the function $\mathbb{AF}_u(m_i, d_i, a_i, \delta_i)$ describes the current availability of a node E_u. It returns the number of tasks of application A_i, with memory and disk space restrictions m_i and d_i, and length a_i, that can be executed by node E_u before deadline δ_i. Let function $l_u(\delta)$ be the amount of FLOPs that node E_u is able to execute before δ. Then, the availability function is calculated as

$$\mathbb{AF}_u(m_i, d_i, a_i, \delta_i) = \begin{cases} \left\lfloor \dfrac{l_u(\delta_i)}{a_i} \right\rfloor & \text{if } m_i \leq M_u \bigwedge d_i \leq D_u \\ 0 & \text{otherwise.} \end{cases} \tag{5.1}$$

Algorithm 5.1 computes $l_u(\delta)$ on node E_u. The local scheduler calculates the position k at which a new task with deadline δ would be placed. The first $k-1$ tasks would be executed in sequence until the time at which the new task would begin; let this time be called b_k. The new task would have to finish before either δ or the time at which task $k+1$ must begin so it does not miss its deadline; let this time be called x_{k+1}. Hence, the available amount of FLOPs in this time interval is

$$l_u(\delta) = (\min(\delta, x_{k+1}) - b_k)s_u.$$

Algorithm 5.1 Computation of $l_u(\delta)$ on node P_u.

Pre: δ is the application deadline. The n tasks are sorted in EDF order.
Post: Return number of instructions available before δ.

```
1:  function l_u(δ)
2:    x_n ← δ_n − a_n/s_u
3:    for j ← n − 1 to 1 do
4:      x_j ← min(δ_j, x_{j+1}) − a_j/s_u
5:    end for
6:    Get k so that δ_{k−1} < δ ≤ δ_{k+1}
7:    b_k ← ν + Σ_{j=1}^{k−1} a_j/s_u                ▷ ν is the current time.
8:    if min(δ, x_{k+1}) > b_k then
9:      return (min(δ, x_{k+1}) − b_k)s_u
10:   else
11:     return 0
12:   end if
13: end function
```

After calculating the position k of the new task, there are two possibilities: either $\delta < x_{k+1}$ or $x_{k+1} \leq \delta$. In the first case, the available amount of FLOPs increases linearly with the deadline, with slope s_u. In the second case, it is constant regardless of how δ varies. These two situations occur for each possible position of a new task in the queue, so we conclude that $l_u(\delta)$ is a piecewise linear function. The endpoints of the intervals that define each piece are, on one hand, those values of δ such that $\delta = x_i \ \forall i$. On the other hand, those at which the position of a new task changes, that is, $\delta = \delta_i \ \forall i$. Moreover, it is trivial to see that the function is continuous also at the interval endpoints.

Figure 5.1 shows an example of $l_u(\delta)$ for a queue with three tasks, τ_1, τ_2 and τ_3. A new task with deadline δ is inserted in the queue between τ_1 and τ_2 if $\delta_1 \leq \delta < \delta_2$, between τ_2 and τ_3 if $\delta_2 \leq \delta < \delta_3$, or behind τ_3 if $\delta_3 \leq \delta$. It would not be accepted if $\delta \leq b_1$, because τ_1 is the running task and it cannot be preempted. The figure shows that, for a task that would be executed between τ_1 and τ_2, $l_u(\delta)$ would evolve in the following way:

- $l_u(\delta) = 0$ if $\delta \leq b_1$, because task τ_1 is still running.
- $l_u(\delta) = (\delta - b_1)s_u$ if $b_1 \leq \delta < x_2$.
- $l_u(\delta) = (x_2 - b_1)s_u$ if $x_2 \leq \delta < \delta_2$, because otherwise τ_2 would not finish before its deadline.

In the moment δ exceeds δ_2, the new task would be scheduled between τ_2 and τ_3. $l_u(\delta)$ continues its evolution in the following way:

- $l_u(\delta) = (x_2 - b_1 + \delta - \delta_2)s_u$ if $\delta_2 \leq \delta < x_3$, due to the time available between task τ_2 and τ_3.
- $l_u(\delta) = (x_2 - b_1 + x_3 - \delta_2)s_u$ if $x_3 \leq \delta < \delta_3$, because τ_3 must meet its deadline.

After δ_3 no other task is scheduled, so $l_u(\delta)$ evolves with unbounded linear growth.

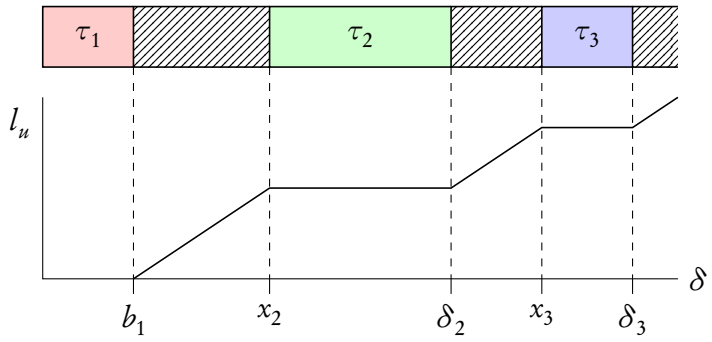

Figure 5.1.: Example of $l_u(\delta)$ for a queue with three tasks. It can be seen that it is a piecewise linear function.

5.3. Global Policy

5.3.1. Availability Information

The sampled function of this policy is a tuple (M, D, L, v). It consists of the number of nodes v, a sample M and D for the available memory and disk space, and a set of samples $L = \{(\delta_j, l(\delta_j)), 1 \leq j\}$. For a sampled function f, let $f.l(\delta)$ be the available amount of FLOPs before δ in each node of f. Then, $f.l(\delta)$ is linearly interpolated from the samples in $f.L$. Being a piecewise linear function, samples are only needed at each interval endpoint of $f.l(\delta)$, in order to know where the slope of this function changes. For instance, in the example of Figure 5.1, there would be a sample for b_1, x_2, δ_2, x_3 and δ_3.

As with the MMP policy, the current time is relevant to calculate the availability function. From the example, it can be seen that the function $l_u(\delta)$ does not change while a task is running, since it cannot be preempted. Thus, the availability information must be updated only when a task finishes.

The sum operation

The sum operation is computed as

$$\text{SUM}(f_1, f_2) = \big(\min(f_1.M, f_2.M), \min(f_1.D, f_2.D), \min(f_1.L, f_2.L), f_1.v + f_2.v\big). \tag{5.2}$$

The deadline is also a restrictive requirement, so the sum operation in (5.2) calculates the minimum available amount of FLOPs before deadline on every node of the sampled function. Then, for $h = \text{SUM}(f, g)$, we have that $h.l = \min(f.l, g.l)$. $h.l$ is again a piecewise linear function, and $h.L$ is the set of samples that interpolate $h.l$. Its slope changes at the same points $f.l$ and $g.l$ do, but also at the points where they cross each other.

Note that, in the worst case, $h.L$ can have up to three times the samples of $f.L$ or $g.L$. We must eliminate some samples to limit the size of all the sampled functions. It would be useless to limit the size of the availability summaries if sampled functions could be arbitrarily large. Eliminating one sample means that the linear interpolation will take place between the adjacent samples. A sample can only be eliminated if the resulting interpolation is lower than the original one, so that it still guarantees a minimum availability. However, this method also introduces an additional approximation error, so the samples that are actually removed are those that introduce the lowest error. The distance operation

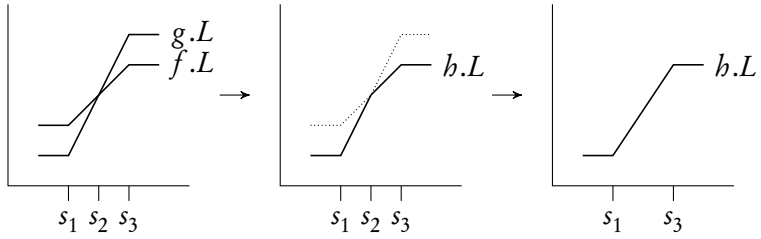

Figure 5.2.: $h.L$ interpolates the minimum of $f.L$ and $g.L$, with sample at s_2 removed.

measures the combination of all these approximation errors committed in the sum operation. An example is shown in Figure 5.2, where functions f and g, with two samples both, are summed up. Both share samples at s_1 and s_3, and cross at s_2. In the result, the minimum is calculated and the sample at s_2 is removed to obtain two samples again.

The distance operation

Again, the MSE is used to compute the distance between two sampled functions. The two scalar parameters M and D are treated as explained in Section 3.2.2. For each of them, we add two additional attributes to accumulate the MSE and the linear term. Also, an availability summary carries the minimum and maximum values for both parameters among all the represented nodes.

For the available amount of FLOPs before deadline, a similar method is used. Being a function instead of a scalar value, the natural way of calculating the MSE is with the definite integral. So, for a sampled function f that represents $f.v$ nodes, we have

$$f.mse_l = \frac{1}{f.v}\sum_{u=1}^{f.v}\int_{\nu}^{h}\left(f.l(\delta) - l_u(\delta)\right)^2 d\delta. \tag{5.3}$$

The endpoints of the domain of integration are the current time ν and an *horizon* h. This horizon must be large enough to compare the $l(\delta)$ functions of all the sampled functions in a summary. So, let t_{last} be the latest sample among the L sets of all the sampled functions, we calculate $h = \nu + 1.2(t_{last} - \nu)$. This adds a 20% margin to take the last piece of every sampled function into account. Since the $l(\delta)$ functions are piecewise linear, there is an analytical form of the integral for each piece. Then, computing the integral in the interval $[\nu, h]$ is $O(n)$, where n is the number of pieces, or the number of samples in L. Remember also that the sum operator limits this value to a maximum.

As we showed for the scalar parameters, the sum operation is also able to calculate the MSE of the resulting sampled function for the l parameter. Let f and g be the sampled functions that result from adding functions f_i, $1 \le i \le f.v$, and g_j, $1 \le j \le g.v$, respectively. Their MSE of parameter l are

$$f.mse_l = \frac{1}{f.v}\sum_{i=1}^{f.v}\int_{\nu}^{h}\left(f.l(\delta) - f_i.l(\delta)\right)^2 d\delta, \qquad g.mse_l = \frac{1}{g.v}\sum_{j=1}^{g.v}\int_{\nu}^{h}\left(g.l(\delta) - g_j.l(\delta)\right)^2 d\delta.$$

The attribute $h.mse_l$ is then

$$h.mse_l = \frac{1}{h.v}\left(\sum_{i=1}^{f.v}\int_{v}^{h}\left(h.l(\delta)-f_i.l(\delta)\right)^2 d\delta + \sum_{j=1}^{g.v}\int_{v}^{h}\left(h.l(\delta)-g_j.l(\delta)\right)^2 d\delta\right). \tag{5.4}$$

Like we did before, if we take the term $\left(h.l(\delta)-f_1.l(\delta)\right)^2$ from (5.4) and add and subtract $f.l(\delta)$, we obtain

$$\begin{aligned}\left(h.l(\delta)-f_1.l(\delta)\right)^2 &= \left(h.l(\delta)-f.l(\delta)+f.l(\delta)-f_1.l(\delta)\right)^2 = \\ &= \left(h.l(\delta)-f.l(\delta)\right)^2 + \left(f.l(\delta)-f_1.l(\delta)\right)^2 + \\ &\quad +2\left(h.l(\delta)-f.l(\delta)\right)\left(f.l(\delta)-f_1.l(\delta)\right).\end{aligned} \tag{5.5}$$

Now, we repeat (5.5) for all the terms on f_i to obtain

$$\begin{aligned}&\sum_{i=1}^{f.v}\int_{v}^{h}\left(h.l(\delta)-f_i.l(\delta)\right)^2 d\delta = \\ &= f.v\int_{v}^{h}\left(h.l(\delta)-f.l(\delta)\right)^2 d\delta + \sum_{i=1}^{f.v}\int_{v}^{h}\left(f.l(\delta)-f_i.l(\delta)\right)^2 d\delta + \\ &\quad +2\int_{v}^{h}\left(h.l(\delta)-f.l(\delta)\right)\sum_{i=1}^{f.v}\left(f.l(\delta)-f_i.l(\delta)\right)d\delta = \\ &= f.v\left(\int_{v}^{h}\left(h.l(\delta)-f.l(\delta)\right)^2 d\delta + f.mse_l\right) + \\ &\quad +2\int_{v}^{h}\left(h.l(\delta)-f.l(\delta)\right)\sum_{i=1}^{f.v}\left(f.l(\delta)-f_i.l(\delta)\right)d\delta.\end{aligned} \tag{5.6}$$

Again, $\sum_{i=1}^{f.v}\left(f.l(\delta)-f_i.l(\delta)\right)$ is the linear term $f.lt_l(\delta)$. In this case, $h.mse_l$ is a scalar value, but $h.lt_l(\delta)$ is also a function. So, from (5.4) and (5.6), the sum operation of f and g would obtain h where

$$\begin{aligned}h.mse_l = \frac{1}{h.v}\Bigg[&f.v\left(\int_{v}^{h}\left(h.l(\delta)-f.l(\delta)\right)^2 d\delta + f.mse_l\right) + \\ &+ g.v\left(\int_{v}^{h}\left(h.l(\delta)-g.l(\delta)\right)^2 d\delta + g.mse_l\right) + \\ &+2\int_{v}^{h}\left(h.l(\delta)-f.l(\delta)\right)f.lt_l(\delta)d\delta + 2\int_{v}^{h}\left(h.l(\delta)-g.l(\delta)\right)g.lt_l(\delta)d\delta\Bigg].\end{aligned} \tag{5.7}$$

and

$$h.lt_l(\delta) = \sum_{i=1}^{f.v} (h.l(\delta) - f_i.l(\delta)) + \sum_{j=1}^{g.v} (h.l(\delta) - g_j.l(\delta)) =$$
$$= f.v(h.l(\delta) - f.l(\delta)) + \sum_{i=1}^{f.v} (f.l(\delta) - f_i.l(\delta)) +$$
$$+ g.v(h.l(\delta) - g.l(\delta)) + \sum_{j=1}^{g.v} (g.l(\delta) - g_j.l(\delta)) =$$
$$= f.v(h.l(\delta) - f.l(\delta)) + g.v(h.l(\delta) - g.l(\delta)) + f.lt_l(\delta) + g.lt_l(\delta). \quad (5.8)$$

Once we calculate the sum of two sampled functions and obtain the MSE for all three parameters, the distance operation is again a weighted sum of their normalized values. The normalization function for the $l(\delta)$ function is similar to the scalar case. The availability summary includes the minimum and maximum $l(\delta)$ functions, let them be called $l_{min}(\delta)$ and $l_{max}(\delta)$. Then, the normalization function is

$$\text{NORM}_l(m) = \frac{m}{\int_v^h (l_{max}(\delta) - l_{min}(\delta))^2 d\delta}. \quad (5.9)$$

5.3.2. Forwarding Algorithm

The forwarding algorithm and routing pattern of the DP policy are similar to the ones in the IBP policy, shown in Algorithm 3.2 and Figure 3.5. The forwarding algorithm obtains the list of sampled functions whose nodes may fulfill the deadline of the new application. The difference is that the DP policy sorts this list having time into account, too. It selects first those sampled functions whose nodes have an available amount of computation before the application deadline closer to its task length. In this way, it tries to minimize the clearance between tasks in each execution node, so that its time is well used. As we said before, minimizing this remainder increases the probabilities of allocating future applications with longer tasks. Finally, if there are any unassigned tasks, they are sent upwards to look for execution nodes in further branches.

If the root node is reached, the unassigned tasks are simply discarded because they cannot be allocated at the moment. This is interpreted by the submission node as a failure in the discovery process. Since deadlines are a quality of service, users must accept that when a task is not allocated, they must look for a longer deadline. The root node does not notify discarding a request, for two reasons. The first one is that it would generate additional traffic. The second one is that with immediate notifications, users would first send a request with a very short deadline, expecting it to fail and slowly increasing the deadline until all tasks are allocated. In this way, they would always obtain the best result for them, with an important increase of load and traffic for every one else. Then, the policy would loose its purpose. The absence of notification is solved by setting a timeout at the submission node of thirty seconds. Having to wait for this time between requests makes it very difficult for users to cheat like that.

Chapter 6.

WDP: DP for Workflow Applications

"All things are difficult before they are easy."
— Thomas Fuller

6.1. Scheduling of Workflow Applications

In this chapter, we explore the possibility of adapting STaRS to a different application type. One of the most common types, specially in scientific domains, are the workflow applications. However, traditional grid platforms, like those based in Condor [90], show poor scaling with big workflows (up to a hundred thousand of tasks) due to scheduler overheads [24]. Scaling is even worse if most tasks in the workflows have a short duration (in the range of minutes). In [57], advance reservations, multi-level scheduling and infrastructure as a service are explored to reduce these overheads.

In [41], the authors propose a distributed double-layer scheduling model for grid workflows. It contains a global scheduler that aggregates information about resource status, to avoid full knowledge. It also considers resource availability fluctuations to allocate the workflows. However, there is no study about the scalability of the system and their experiments, using 3 resource clusters with 10 nodes each, do not allow to evaluate this feature either. In [79], a scheduling algorithm is described based on the cooperation of distributed workflow brokers. A distributed hash table provides a decentralized coordination space that is responsible for the resource discovery and scheduling, and it uses a FCFS allocation strategy.

So, it seems reasonable to apply a decentralized model to the scheduling of workflow applications. We present the *Workflow-with-Deadlines Policy* (WDP), an extension of the DP policy for workflow applications with time constraints. It includes a workflow decomposition process that allows the global scheduler to match sequences of dependent tasks with time constraints. Since this is a first approach, we only consider available computing time as the availability information of execution nodes. Likewise, the information model is simpler than the ones shown in the previous policies.

6.1.1. Workflow Management

A workflow is modeled after a DAG $G(T,D)$, where nodes represent tasks and edges represent dependencies between them. For each $\tau_i, \tau_j \in T$, an edge $(\tau_i, \tau_j) \in D$ exists if task τ_j depends on τ_i. In that case, task τ_i must finish before τ_j may start. An example can be seen in Figure 6.1.

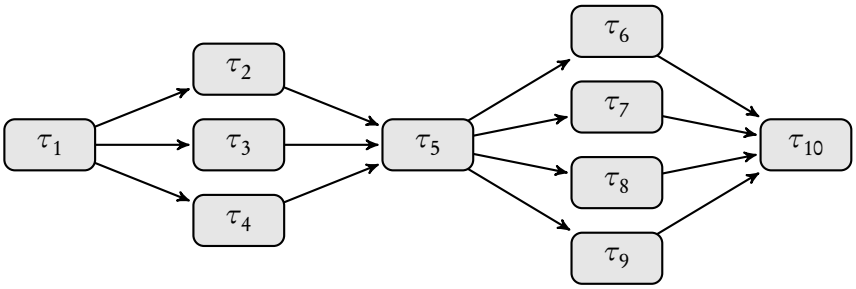

Figure 6.1.: A typical DAG with 10 tasks and several dependencies between them.

We divide workflows into several parts that can be submitted concurrently, to exploit the inherent parallelism of the graph structure. Classic workflow scheduling algorithms[16] usually divide them by DAG levels. Two tasks τ_i and τ_j are in the same level when an edge (τ_j, τ_i) or (τ_i, τ_j) does not exist. This decomposition is suitable when the objective of the scheduler is to optimize makespan. A set of resources is reserved in advance, and levels are allocated one after another minimizing the queue length of these resources.

However, we use a different approach, since we allocate resources as they are discovered. We divide the DAG into *sequences* of dependent tasks. A sequence is an ordered set of tasks $S = \{\tau_i | 1 \leq i \leq n\}$ where $\forall i, 1 \leq i < n, (\tau_i, \tau_{i+1}) \in D$. The longest sequence gets its deadline directly from the DAG. Shorter ones get their time constraints from the sequences they depend on, as they are allocated.

The decomposition is depicted in Algorithm 6.1. It starts by extracting the sequence that contains the longest path of the DAG, in terms of task length. Then, at each iteration, it extracts the longest sequence S so that edges may only go:

1. From an already extracted task to the first task of S.

2. From the last task of S to an already extracted task.

Note that any DAG may be decomposed this way, as at each iteration at least a sequence of one task is eligible. Thus, all tasks are eventually assigned to a sequence.

Once the DAG is decomposed into sequences, they are assigned not only a deadline, but also a *startline*. Every sequence must start after its startline and end before its deadline. The startline of a sequence is calculated after the task it depends on is allocated, because we know when it is supposed to finish. Likewise, its deadline is calculated after the tasks that depend on the sequence are allocated, because we know when they must start to meet their own deadline. The longest sequence is allocated first, because its deadline is provided by the user and the startline is the current time. After it is allocated, all the sequences that get their constraints from it get prepared. At each step, all the prepared sequences may be sent concurrently, as they do not depend on each other. Thus, the submission process consists of a set of stages, in which all the prepared sequences are sent. We define the *width* of a workflow as the number of stages needed to submit the complete workflow. Likewise, we define the *minimum length* of a workflow as the sum of the task lengths in the critical path, and the *total length* of a workflow as the sum of all the task lengths. As we show in the experimental results of Section 8.6, these properties have relevant impact in the system performance.

Algorithm 6.1 Extract the sequences of dependent tasks from G

Pre: G is a DAG with tasks in $G.T$ and dependencies in $G.D$.
Post: $\mathbb{S}$ contains the sequences extracted from G.

```
1: function DECOMPOSITIONINTOSEQUENCES(G)
2:   Extract longest path from G to sequence S.
3:   𝕊 ← 𝕊 ⋃ S
4:   while G.T ≠ ∅ do
5:     Extract the tasks from G.T that form the longest path S so that:
         • ∀s_i ∈ S | s_i ≠ s_1, ∄τ_i ∈ 𝕊 | (τ_i, s_i) ∈ G.D
         • ∀s_i ∈ S | s_i ≠ s_n, ∄τ_i ∈ 𝕊 | (s_i, τ_i) ∈ G.D
6:     𝕊 ← 𝕊 ⋃ S
7:   end while
8:   return 𝕊
9: end function
```

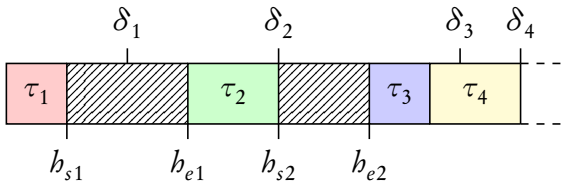

Figure 6.2.: Task queue with four tasks, and the holes available between them.

6.2. Local Policy

This policy, like the DP policy, uses an EDF-based local scheduler. For a sequence of tasks, the deadline of the sequence is applied to the last task. Then, the deadline and estimated execution time of each task is used to calculate the deadline of its predecessor.

Every time the task queue changes, the availability of an execution node is recalculated, to report when and of what length new sequences could be accepted. To obtain this information, all tasks in the queue are pushed to their deadline, except the currently running task, because task preemption is not allowed. Then, a list is built up from the "holes" of availability that exist between tasks. These holes represent a simplified view of the maximum length a new sequence may have to avoid making another task miss its deadline, if it is accepted.

Figure 6.2 shows an example of a task queue with four tasks. A hole that starts at h_{si} and ends at h_{ei}, represents the maximum length a sequence may have, with a deadline in $[\delta_i, \delta_{i+1})$, so that in EDF order it would execute before τ_{i+1}. Actually, if τ_2 is executed just after τ_1, the hole between τ_2 and τ_3 would be longer, but it is not considered for sake of simplicity. Note that there is no hole between τ_3 and τ_4, as τ_4 should have already started by δ_3.

The computational power of the node is also reported, so that it can be used to estimate how much work is performed by the node in an arbitrary period of time; for instance, when a node is completely idle.

Algorithm 6.2 Generate a set of n interval endpoints from the current time v.

Pre: n is the number of endpoints to create, d_{min} is the minimum interval duration.
Post: L contains n interval lower endpoints.

```
1: procedure CREATEENDPOINTS(n)
2:   L ← ∅
3:   t_0 ← v − BEGINNINGOFDAY(v)
4:   d_1 ← d_min
5:   for i ← 1 to n do
6:     L ← L ⋃ {⌈t_0/d_i⌉ + 1) × d_i}
7:     d_{i+1} ← 2d_i
8:   end for
9: end procedure
```

6.3. Global Policy

6.3.1. Availability Information

Once the set of holes is created, it is sent to the father routing node so that it will be aggregated with the sets of the other execution nodes in that branch. However, aggregating sets of arbitrary holes would be impossible. Every pair of holes start and end at different times. For this reason, we have developed a method to approximate them to a set of fixed values.

Since this is a first approach, we have simplified the availability information model. An availability summary only contains one sampled function, with the number v of nodes it represents and an ordered set of *time intervals*. Intervals are consecutive, so interval I_i ends at the time interval I_{i+1} starts. Each one contains the list of holes that finish within its endpoints. Within each list, holes are further classified with two criteria. First, holes are grouped by their *span*. A hole that starts in interval I_i and finishes in interval I_{i+k} has a span of $k+1$ intervals. So, a hole that starts and finishes in the same interval has a span of one interval. However, holes with the same span may not have the same availability, as this depends on the computational power of each execution node. So, they are also classified by availability levels.

Instead of using the clustering algorithm presented in Section 3.2.2, we select a suitable set of availability levels and interval endpoints that allow an easy aggregation of summaries. They must have similar values in all the availability summaries that are to be aggregated together. First, the availability of every hole is approximated to the immediately lower power of two. Like with the DP policy, this conservative method enables finding nodes where tasks meet deadlines. Also, by using powers of two, the error is always lower than 50%.

The interval endpoints are generated with Algorithm 6.2. It only generates the lower endpoints; the upper endpoint of an interval is equal to the lower endpoint of its successor. Its objective is that summaries created at moments near in time have most endpoints in common. To accomplish this, it generates a set of lower endpoints whose difference with the beginning of the day is a multiple of certain durations. In the algorithm, the creation time t_0 is the current time v, relative to the beginning of the day. Then, d_{min} is the minimum interval duration. The first lower endpoint is $t_0 + d_{min}$, rounded up to the next multiple of d_{min}. Then, the duration is doubled for every successive

Table 6.1.: Interval endpoints generated from three example creation times t_0. The difference between each endpoint and t_0 is always between one and two times the intended interval duration.

	Intended interval duration									
t_0	5'	10'	15'	30'	1h	2h	4h	8h	16h	1 day
4:33	4:40	4:50	5:00	5:30	6:00	8:00	12:00	16:00	1d:00:00	2d:00:00
5:48	5:55	6:00	6:15	6:30	7:00	8:00	12:00	16:00	1d:00:00	2d:00:00
17:17	17:25	17:30	17:45	18:00	19:00	20:00	1d:00:00	1d:08:00	1d:16:00	2d:00:00

interval, again to maintain the error of the approximation under 50%. So, for interval i, its duration d_i is $d_{min} \times 2^{i-1}$ and it starts at $t_0 + d_i$ rounded up to the next multiple of d_i. The result is that the difference between the lower endpoint of interval i and t_0 is between one and two times d_i. Table 6.1 shows three examples of the output of this algorithm. There, d_i is slightly modified to provide more "human-readable" results. Instead of being doubled at every step, it jumps from 10 minutes to 15 minutes, and from 16 hours to 1 day.

6.3.2. Forwarding Algorithm

The forwarding algorithm decides how and where to route task sequences. It tries to find holes with enough availability to execute all the tasks in the sequence. Like in the DP policy, it looks for holes that better fit the sequence requirements. That is, one that starts short before the startline and ends short after the deadline of the sequence, and with an availability similar to the sequence length. In this way, it leaves bigger holes in case a longer sequence is received later, trying to improve resource usage.

The forwarding algorithm of this policy is shown in Algorithm 6.3. First, GETPARTITIONS calculates all the possible partitions of the sequence into multiple consecutive subsequences. The algorithm iterates them, in ascending order of number of subsequences. At each iteration, GETHOLES looks for a set of holes that can accept each of the subsequences of the partition. They cannot overlap, to respect the dependencies between consecutive subsequences. When a hole is found for each subsequence, a new request is constructed for each one and sent to the corresponding subbranch. For each subsequence, the startline and deadline are taken from its respective hole endpoints. Finally, if the forwarding algorithm finds no hole for any of the subsequences, the original request is routed to the next level of the tree to try further.

Algorithm 6.3 Forwarding algorithm for the WDP policy.

Pre: R_u is this routing node. `request` is the request.
Post: `reqLeft`, `reqRight` and `reqFather` are the resulting requests to be sent to the left child, right child and father nodes, respectively.

```
1:  procedure FORWARD(request)
2:    P ← GETPARTITIONS(request.sequence)
3:    for numParts ← 1 to request.sequence.length do
4:      for all S ∈ {s ∈ P | s.length = numParts} do
5:        holes ← GETHOLES(R_u.info, S)
6:        if |holes| ≥ numParts then                ▷ There is a hole for every sequence.
7:          reqLeft ← GETLEFTPARTS(holes)
8:          reqRight ← GETRIGHTPARTS(holes)
9:          return
10:       end if
11:     end for
12:   end for
13:   reqFather ← request                           ▷ If we get here, there was no suitable partition.
14: end procedure
```

Chapter 7.

FSP: Fair Share Policy

"These men ask for just the same thing, fairness, and fairness only. This, so far as in my power, they, and all others, shall have."
— Abraham Lincoln

7.1. Fairness as the Scheduling Objective

The *Fair Share Policy* (FSP) tries to provide a similar share of the platform among the scheduled applications. The fair sharing of resources has been deeply studied before in other areas, like networking [49, 68], which consider the amount of data to be transferred by each user. When scheduling multiple applications, we consider the amount of computation that each user wants to get done. In this case, the most suited metric seems to be the maximum *stretch*, or slowdown [69, 62]. The stretch of an application is defined as the ratio of its response time under the concurrent scheduling of applications to its response time when it is the only application executed on the platform. Let r_i be the release time of an application, e_i its finish time and t_R its response time in a dedicated platform, its stretch S_i is calculated as

$$S_i = \frac{e_i - r_i}{t_R}. \tag{7.1}$$

Ideally, a fair share of the platform is obtained scheduling applications so that all of them obtain the same stretch. Like it happened with the MMP policy, this is only possible in practice with offline scheduling and divisible load. We consider a more constrained environment, with online scheduling and atomic tasks, so the best tradeoff is obtained by minimizing the maximum stretch among all applications. Previously, Benoit et al. [14] have studied the minimization of maximum stretch for concurrent applications in a centralized setting. In particular, their study shows that interleaving tasks of several concurrent bag-of-tasks applications performs better than scheduling each application after the other.

The main problem we face is that computing the response time of an application in a dedicated platform requires full knowledge of its characteristics. While easy to perform in a centralized context, it is unthinkable in a decentralized one. However, with some reasonable assumptions we can find a good approximation.

1. Applications have much less tasks than nodes in the platform.

2. The distribution of computing power changes very little.

We already presented a first design of this policy in [37]. In this chapter, we refine its methods to obtain better results.

7.1.1. Slowness

The first assumption is that *applications have much less tasks than nodes in the platform*. From the traces used in the experiments of Section 8, the number of tasks per application in a common cluster computing environment is distributed as:

- The application with more tasks has 1120 tasks.
- 95% of applications have less than 128 tasks.
- 75% of applications have less than 16 tasks.
- 50% of applications have only one task.

Since we aim for platforms with a hundred thousand nodes or more, to minimize t_R of an application with n_i tasks, we have to allocate a task to each of the n_i fastest nodes. Then, the response time would be the time needed by the slowest of these nodes to execute its task.

The second assumption is that *the distribution of computing power changes very little*. Even more, we assume that the fastest nodes will have a similar computing power. This is more complicated, since the computing power distribution is usually skewed towards the lower values, but with the relation of a thousand tasks to a hundred thousand nodes, we think it is reasonable. In that case, we can approximate t_R as a_i/s_{max}, where s_{max} is the computing power of the fastest node. In practice, this value is still unknown. But since we assume that it changes very seldom, we turn the problem of minimizing the maximum stretch into minimizing the maximum ratio between the stretch and s_{max}. This ratio z_i is

$$z_i = \frac{S_i}{s_{max}} = \frac{e_i - r_i}{a_i}. \tag{7.2}$$

Note that z_i is the inverse of the effective speed at which a task has been executed, so we call it *slowness*.

7.2. Local Policy

7.2.1. Minimizing the Local Maximum Slowness

The local scheduler only knows about the applications which have at least one task allocated to its execution node. So, its objective is to minimize the maximum slowness among these applications. Moreover, in order to estimate the eventual slowness of an application, it must calculate its end time e_i based only on the tasks that are contained in the queue. Therefore, two local schedulers may compute a different slowness for the same application. The global scheduling policy is in charge of minimizing this unbalance.

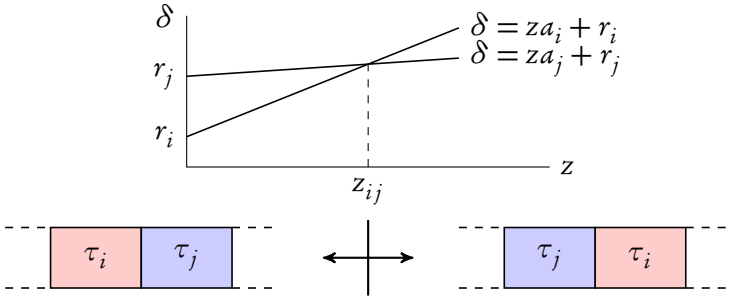

Figure 7.1.: Value z_{ij}, where tasks τ_i and τ_j switch positions because their deadline functions cross.

Given an ordering of the task queue, each application has a certain slowness. To reduce the current maximum slowness, the local scheduler must find out if there is a better ordering. Our strategy is to set a target maximum slowness and calculate the queue ordering that provides it, if it exists. From equation 7.2, to obtain a certain maximum slowness, each application must finish before an end time e_i so that its slowness is equal or lower to the maximum. So, the end time of each application can be interpreted as a deadline δ_i:

$$z_i = \frac{\delta_i - r_i}{a_i} \implies \delta_i = z_i a_i + r_i \tag{7.3}$$

Given the deadline of their application, the tasks in the queue are sorted in EDF order as we did in the DP policy. If all the tasks meet their deadline, the current queue ordering provides a maximum slowness below or equal to the target. Otherwise, the target maximum slowness is not feasible. So, we can use a binary search to find the ordering that minimizes the maximum slowness.

Since the ordering of the queue depends on the ordering of the deadlines, the same ordering is obtained by a range of slowness values. As we cross the boundary between two ranges, the deadline of a task surpasses the deadline of another and they switch positions (Figure 7.1). These boundary values z_{ij} are

$$\delta_i = \delta_j \qquad \implies \qquad z_{ij}a_i + r_i = z_{ij}a_j + r_j \qquad \implies \qquad z_{ij} = \frac{r_j - r_i}{a_i - a_j}. \tag{7.4}$$

In Algorithm 7.1, the set of boundary values is computed from every pair of tasks of the queue, discarding those pairs where $a_i = a_j$ and the negative boundary values. Then, the set is sorted. The queue maintains the same ordering between any two consecutive boundary values, so the binary search is performed on the set of boundary values. This is shown in Algorithm 7.2, that sorts the queue to minimize the maximum slowness. It uses Algorithm 7.3 to sort the queue with any target slowness that lies between two consecutive boundary values. Then, Algorithm 7.4 checks whether all the tasks meet their deadline, with that order, at the lower boundary. If they do, we must continue the binary search under that value. Otherwise, we must continue over it.

Algorithm 7.1 Find the set of boundary values for the tasks in the queue Q.

```
Pre:  Q is the task queue.
Post: B is the sorted set of boundary values for the tasks in Q.
  function GETBOUNDARIES(Q)
    B ← ∅
    for all τ_i ∈ Q do
      for all τ_j ∈ Q do
        if a_i ≠ a_j then
          z_ij ← (r_j − r_i)/(a_i − a_j)
          if z_ij ≥ 0 then
            B ← B ⋃ {z_ij}
          end if
        end if
      end for
    end for
    SORT(B)
    return B
  end function
```

7.2.2. Availability Function

The availability of an execution node is described by function $\mathbb{AF}_u(m_i, d_i, a_i, r_i, z_{max})$. It returns the number of tasks of a new application A_i, with memory and disk constraints m_i and d_i, task length a_i and release time r_i that can be executed by node E_u so that the maximum slowness among all its applications is not higher than z_{max}. Like in previous policies, to compute it we must take the amount of FLOPs of application A_i available at E_u, divide it by a_i and round down. In the DP policy, the available amount of FLOPs only depended on the application's deadline δ_i. In this policy, it depends on a_i, r_i and z_{max}. Representing and clustering a function of three parameters is much more complex. So, we simplify the problem to reduce the number of independent variables. We do this in two ways:

1. We eliminate r_i, so that $\mathbb{AF}_u(m_i, d_i, a_i, z_{max})$ returns the number of tasks of application A_i that node E_u is able to execute if A_i is released at the current time.

2. Like in the MMP policy, the best way of reducing the slowness is to allocate each task of A_i to a different node. So, we calculate the maximum slowness of adding just one task to each node.

Let function $z_u(a)$ be the maximum slowness among all the applications in E_u if we add a new application with one task of length a that is released at the current moment. Then, the result of $\mathbb{AF}_u(m_i, d_i, a_i, r_i, z_{max})$ will be 1 if memory and disk space requirements are met, and $z_u(a_i) \leq z_{max}$. $z_u(a)$ is a one parameter function, like $l_u(\delta)$, so it can be represented and clusterized in a similar way. Of course, two problems arise: Are we able to estimate $z_u(a)$ if the release time of the application is later than the one we used to compute it? Are we able to estimate $z_u(a)$ if we want to allocate more than one task of the same application? We show later how to solve these problems.

Algorithm 7.2 Sort a task queue to minimize its maximum slowness.

Pre: Q is the task queue.
Post: Q is sorted so that the maximum slowness among its applications is minimized.

```
procedure SORTMINSLOWNESS(Q)
  B ← GETBOUNDARIES(Q)
  I_min ← 1, I_max ← |B|
  while I_min + 1 < I_max do
    I_med ← ⌊(I_min + I_max)/2⌋
    medianSlowness ← (B[I_med] + B[I_med + 1])/2        ▷ B[i] is element i of sorted set B.
    SORTBYSLOWNESS(Q, medianSlowness)
    if MEETDEADLINES(Q, B[I_med]) then
      I_max ← I_med
    else
      I_min ← I_med
    end if
  end while
  medianSlowness ← (B[I_min + B[I_min + 1])/2
  SORTBYSLOWNESS(Q, medianSlowness)
end procedure
```

Algorithm 7.3 Recalculate deadlines for a maximum slowness and sort tasks.

Pre: Q is a task queue, z is the target maximum slowness.
Post: Tasks in Q are sorted by non-decreasing deadlines.

```
procedure SORTBYSLOWNESS(Q, z)
  for all τ ∈ Q do
    τ.δ ← τ.r + zτ.a
  end for
  SORTEDF(Q)          ▷ The first task always remains first, because it is not preemptible.
end procedure
```

Algorithm 7.4 Check if all tasks in a queue meet their deadlines for a certain slowness.

Pre: Q is a task queue.
Post: Whether all tasks in Q, with their current order, meet their deadlines for a certain slowness.

```
function MEETDEADLINES(Q, z)
  b ← ν                                   ▷ ν is the current time.
  for all τ ∈ Q do
    b ← b + τ.a/s_u
    if b > τ.r + zτ.a then return false
    end if
  end for
  return true
end function
```

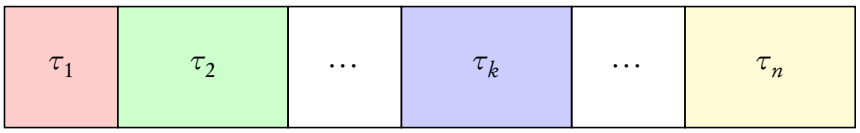

Figure 7.2.: A possible task queue with n tasks.

Characterization of $z_u(a)$

From Equation 7.2, the slowness of an application depends on its finish time e_i, so it depends on the position of its tasks in the queue. If we want to assign a similar slowness to every applications, we are forcing a relation between the position of a task and its length. As a rule of thumb, shorter tasks will usually finish sooner than longer ones. So, to characterize $z_u(a)$ we must take into account that the queue ordering changes for different values of a.

Consider the task queue in Figure 7.2. It contains n tasks τ_i, $1 \leq i \leq n$. For simplicity in the notation, assume that the subindex i identifies the position of the task in the queue. Also, consider that a_i is the length of τ_i, δ_i is its deadline, and so on. During the explanation, task τ_k is the new task we want to introduce in the queue. It occupies position k, has length a_k and is released at r_k. So, we want to characterize $z_u(a_k)$, taking into account that the current time is r_k.

For any value of a_k, each task has the following slowness:

- The slowness of tasks before τ_k does not depend on a_k:

$$z_{i<k} = \frac{e_i - r_i}{a_i} = \frac{r_k + \sum_{j=1}^{i}(a_j/s_u) - r_i}{a_i} = \frac{\sum_{j=1}^{i} a_j + (r_k - r_i)s_u}{a_i s_u} \tag{7.5}$$

- The slowness of τ_k depends inversely on a_k:

$$z_k = \frac{e_k - r_k}{a_k} = \frac{a_k + \sum_{j=1}^{k-1} a_j + (r_k - r_k)s_u}{a_k s_u} = \frac{\sum_{j=1}^{k-1} a_j}{a_k s_u} + \frac{1}{s_u} \tag{7.6}$$

- The slowness of tasks after τ_k depends linearly on a_k:

$$z_{i>k} = \frac{e_i - r_i}{a_i} = \frac{a_k}{a_i s_u} + \frac{\sum_{j=1, j\neq k}^{i} a_j + (r_k - r_i)s_u}{a_i s_u} \tag{7.7}$$

The result of $z_u(a_k)$ is the maximum of all of these values. The function maintains the same tendency (constant, inverse or linear) until one of the following conditions is true:

- Another task becomes the task with maximum slowness. This happens at the value of a_k for which the current maximum slowness is equal to the slowness of another task.

- The queue changes its order to provide a lower maximum slowness. This happens at the value of a_k for which the deadline of two tasks is equal.

Algorithm 7.5 Computation of $z_u(a)$ pieces.

Pre: Q is a task queue.
Post: S is a list of pieces of function $z_u(a)$ for queue Q.

```
1:  function GETPIECES(Q)
2:    S ← ∅
3:    A ← {a_min}
4:    while A ≠ ∅ do
5:      a_k ← min(a ∈ A)
6:      Q' ← Q ⋃ {τ_k}                          ▷ Insert new dummy task of length a_k.
7:      SORTMINSLOWNESS(Q')
8:      Get i so that τ_i is the task that sets the maximum slowness in Q'.
9:      if i < k then
10:       p ← New piece is constant, left endpoint is a_k.
11:     else if i = k then
12:       p ← New piece is inverse, left endpoint is a_k.
13:     else if i > k then
14:       p ← New piece is linear, left endpoint is a_k.
15:     end if
16:     S ← S ⋃ {p}
17:     A ← {a | τ_{i<k} sets the maximum (Equation 7.5)}
18:     A ← A ⋃ {a | τ_k sets the maximum (Equation 7.6)}
19:     A ← A ⋃ {a | τ_{i>k} sets the maximum (Equation 7.7)}
20:     A ← A ⋃ {a | z_u(a) ∈ GETBOUNDARIES(Q')}   ▷ Queue changes order.
21:   end while
22:   return S
23: end function
```

The conclusion is that $z_u(a)$ is a piecewise function again. Each piece can be a constant, an inverse function or a linear function of a. We can characterize each piece $p_j(a)$ with parameters u_j, v_j and w_j and endpoints α_j and ω_j as

$$p_j(a) = \frac{u_j}{a} + v_j a + w_j, \text{ when } a \in [\alpha_j, \omega_j). \tag{7.8}$$

Algorithm 7.5 calculates the parameters and endpoints of each piece of $z_u(a)$ from the queue of E_u. At each iteration, set A contains all the potential left endpoints of the next piece, so we select a_k as their minimum. Then, the algorithm creates a copy of the queue Q that includes a dummy task τ_k of length a_k. a_k starts with a minimum task length $a_{min} > 0$, to avoid dividing by zero when calculating the slowness. The queue is sorted and the algorithm calculates which task is setting the maximum slowness. If it is previous to τ_k, the new piece is constant ($u = 0$, $v = 0$ and $w \neq 0$). If it is τ_k, the new piece is inverse ($u \neq 0$, $v = 0$ and $w \neq 0$). If it is after τ_k, the new piece is linear ($u = 0$, $v \neq 0$ and $w \neq 0$). In any case, the left endpoint is a_k and the new piece is added to the result set. Then, the algorithm calculates the potential left endpoints of the next piece, as expected for the

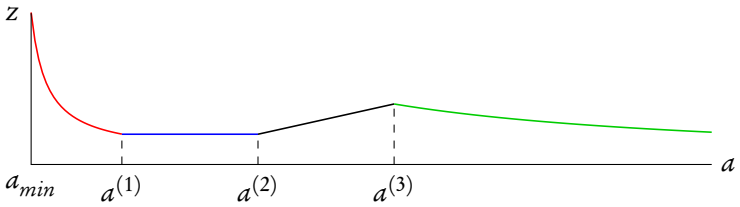

Figure 7.3.: Example of a $z_u(a)$ function with four pieces. a_{min} is the shortest length a task may have.

next iteration. Set A is populated with the a values at which another task sets the maximum slowness – using Equations 7.5, 7.6 and 7.7 – and those at which the queue changes order – using the set of boundaries of Algorithm 7.1. When this set is empty, the algorithm ends, and the last piece has no right endpoint. We have chosen $a_{min} = 1$ because, although it is an unprovable task length, it covers all the possible situations.

Figure 7.3 shows an example of an $z_u(a)$ function with four pieces. The first piece is an inverse function, because the new task is setting the maximum slowness. This is a very common situation, because for a small value of a the slowness of the new task is very big. When $a = a^{(1)}$, a task whose position in the queue is before the new one becomes the one that sets the maximum, so the second piece is constant. At $a = a^{(2)}$, the new task has pushed a later one so much that now it becomes the task that sets the maximum, and the third piece is a linear function. Finally, at $a = a^{(3)}$, the new task is setting the maximum slowness again.

Estimating $z_u(a)$ with different r_i

The problem of simplifying variables is that, once we have calculated the piece endpoints and parameters, we do not have them anymore. Not using r_i as a variable of $z_u(a)$ means that we have to assume that the release time is always the same. It is value r_k in Equation 7.6. The problem is that $z_u(a)$ is used on applications that will be released in the future. We need to estimate the result of $z_u(a)$ for applications with $r_i = r_k + t$, where t is an arbitrary amount of time. Equation 7.6 would look like

$$z_k = \frac{e_k - r_k}{a_k} = \frac{a_k + \sum_{j=1}^{k-1} a_j + (r_k - r_k - t)s_u}{a_k s_u} = \frac{\sum_{j=1}^{k-1} a_j}{a_k s_u} + \frac{1}{s_u} - \frac{t}{a_k},$$

showing that the slowness of the new task decreases as its release time advances. That was expected from Equation 7.2. Note that $z_u(a)$ only changes at those pieces where τ_k sets the maximum slowness ($u_j \neq 0$). So, we have two options to estimate $z_u(a)$ for $r_i = r_k + t$:

1. Calculate $z_u(a)$ as if $t = 0$. At those pieces where τ_k sets the maximum slowness, the estimation will be higher than the real value because we miss the $-t/a_k$ term.

2. Calculate $z_u(a)$ with $t = r_i - r_k$ to obtain a better estimation. In this case, for every $z_u(a)$ function we also need the r_k value that was used to calculate its parameters. Then, subtract t from u_j in every piece where $u_j \neq 0$.

In practice, we have observed that ignoring t yields better results. Since we are minimizing the maximum slowness, we do not want to get a higher real value than the one we estimated. Besides, only with the piece endpoints and parameters, we have a very limited information about E_u queue. First, in those pieces where we subtract t from u_j, we do not know whether τ_k is still setting the maximum slowness. Second, the endpoints of those pieces may change in an unknown way, yielding a wrong estimation in the adjacent ones.

Nevertheless, note that the previous considerations assume that the queue does not change. So, in any case, $z_u(a)$ pieces must be recalculated every time a new task is added to the queue or an old one finishes.

Estimating $z_u(a)$ for more than one task

This is how Equations 7.6 and 7.7 look like if we allocate n tasks of size a_k:

$$z_k = \frac{na_k + \sum_{j=1}^{k-1} a_j + (r_k - r_k)s_u}{a_k s_u} = \frac{\sum_{j=1}^{k-1} a_j}{a_k s_u} + \frac{n}{s_u}$$

$$z_{i>k} = \frac{na_k}{a_i s_u} + \frac{\sum_{j=1, j\neq k}^{i} a_j + (r_k - r_i)s_u}{a_i s_u}$$

Once when we have the piece endpoints and parameters of $z_u(a)$ for one task, we can safely say that, multiplying the v_j and w_j parameters of each piece by n, we obtain a good estimation of $z_u(a)$ for n tasks. Two problems arise, similarly to the estimation with different r_i. First, in pieces where τ_k was setting the maximum slowness, a task later in the queue may take over, or vice versa. Second, the piece endpoints may also change, like in the previous case. Nevertheless, the experiments have shown that, in this case, the estimation is better than not taking action.

7.3. Global Policy

7.3.1. Availability Information Management

Similarly to the DP policy, the sampled function of the FSP policy consists of the parameters (M, D, Z, v). It contains the number of nodes v, a sample M and D for the available memory and disk space, and a set of samples $Z = \{(\alpha_j, u_j, v_j, w_j), 1 \leq j\}$. These samples define the pieces of the $z(a)$ function of the nodes represented by this sampled function, with their endpoints and parameters. Only the left endpoint α_j is needed because pieces are adjacent, the right endpoint is the next piece's left endpoint. The last piece has no right endpoint. The first two parameters of a sampled function are scalar values, as usual. Z is a functional parameter, and all the considerations that applied to L in Section 5.3.1 apply to Z now. In particular, for notation, the function represented by $f.Z$ is written as $f.z(a)$. As we said in the previous section, $z_u(a)$ must be recomputed in E_u every time its queue changes, so an availability summary is sent to the father routing node with this frequency.

Figure 7.4.: Two examples of how to join two pieces (black) into one (red).

Sum operation

The sum operation is computed as

$$\text{SUM}(f_1, f_2) = \big(\min(f_1.M, f_2.M), \min(f_1.D, f_2.D), \max(f_1.Z, f_2.Z), f_1.v + f_2.v\big). \tag{7.9}$$

As we said before, since we are trying to minimize the maximum slowness, we prefer that the estimation overshoots the real value. So, the sum operation calculates the maximum between $f_1.Z$ and $f_2.Z$. Like with the L parameter of the DP policy, the result is a set of pieces whose endpoints come from the endpoints of $f_1.z(a)$ and $f_2.z(a)$, and from the points where they cross each other. So, the number of pieces in $\text{SUM}(f_1, f_2).Z$ may be up to three times the number of pieces of $f_1.Z$ or $f_2.Z$. In this case, we reduce its number by joining two consecutive pieces into a new one. It covers the same range as the pieces it replaces, and its parameters are calculated so that it maintains the same value at its endpoints, and provides a higher or equal estimation of the slowness between them. Figure 7.4 show two examples of this process. Joining two pieces carries an error in the estimation, so the sum operation iteratively joins those pieces that minimize the error, until a fixed number of pieces is reached again. Then, these estimation errors are also taken into account in the distance operation.

Distance operation

In the distance operation of the FSP policy, the parameters M and D are treated as scalar values, as explained in Section 3.2.2, while parameter Z is treated like the parameter L of the DP policy in Section 5.3.1. Let $f.z(a)$ be the function described by the pieces in $f.Z$. Then, the sum operation of f and g would obtain h with the MSE of parameter Z,

$$\begin{aligned} h.mse_z = \frac{1}{h.v} \Bigg[& f.v \left(\int_v^h \big(h.z(a) - f.z(a)\big)^2 da + f.mse_z \right) + \\ & + g.v \left(\int_v^h \big(h.z(a) - g.z(a)\big)^2 da + g.mse_z \right) + \\ & + 2 \int_v^h \big(h.z(a) - f.z(a)\big) f.lt_z(a) da + \\ & + 2 \int_v^h \big(h.z(a) - g.z(a)\big) g.lt_z(a) da \Bigg], \end{aligned} \tag{7.10}$$

and the linear term of parameter Z,

$$h.lt_z(a) = f.v\big(h.z(a) - f.z(a)\big) + g.v\big(h.z(a) - g.z(a)\big) + f.lt_z(a) + g.lt_z(a). \tag{7.11}$$

For the normalization function of the Z parameter, the availability summary includes the minimum and maximum $z(a)$ functions as usual. Let them be called $z_{min}(a)$ and $z_{max}(a)$, the normalization function is

$$\text{NORM}_z(m) = \frac{m}{\int_\nu^h \left(z_{max}(a) - z_{min}(a)\right)^2 da}. \tag{7.12}$$

7.3.2. Forwarding Algorithm

The FSP policy tries to minimize a global property, the maximum slowness. So, its forwarding algorithm and routing pattern are closer to those of the MMP policy. Tasks must be sent to those nodes whose maximum slowness will remain lower after accepting them. To have an idea about the global maximum slowness, besides the availability information that comes from each child, routing nodes receive the maximum slowness of the rest of the tree from their father. Like in the MMP policy, being a maximum, it seldom changes and its impact in the traffic is negligible. Requests climb up the tree until a routing node decides that, with the nodes of its branch, the maximum slowness will be no greater than β times the maximum slowness of the rest of the tree. Then, it divides the set of tasks of the request between its children. In Chapter 8 we also show that, in this case, the value of β that obtains the best results is 0.04.

The forwarding algorithm of the FSP policy appears in Algorithm 7.6. First, it calculates the minimum slowness that can be reached allocating the tasks of the request to the nodes in the current branch. This is done by function GETMINSLOWNESS, in Algorithm 7.7. It creates a list `candidates` with all the sampled functions and estimates the slowness obtained by assigning one task to the nodes of each candidate. Procedure PURGE eliminates from the list the worst candidates so that there remains just enough nodes to allocate `request.`n tasks. At this point, the partial result is the maximum slowness among them, in `candidates.last.slowness`. However, if we assign more tasks to the nodes with lowest slowness, they could still get a lower slowness than the other nodes. To find this out, the algorithm iteratively checks whether it obtains a better result by assigning one task more to some sampled function. At iteration i, the sampled functions that obtained $i-1$ tasks in the previous one are tested with one task more. This is done with function ESTIMATEZ, that estimates the value of function $z(a)$ with i tasks. If they still get a lower slowness than the partial result, they get that new one task, and the list of candidates is purged again. The process stops when it cannot obtain a lower slowness. The forwarding algorithm then compares the obtained slowness with β times the maximum slowness in the rest of the tree. If it is lower, the availability function decides how many tasks to send to each child. It is easy to see that the routing pattern generated by this forwarding algorithm is the same as that of the MMP policy.

Algorithm 7.6 Forwarding algorithm for the FSP policy.

Pre: R_u is this routing node. `request` is the request.
Post: `reqLeft`, `reqRight` and `reqFather` are the resulting requests to be sent to the left child, right child and father nodes, respectively.

```
1: procedure FORWARD(request)
2:   minSlowness ← GETMINSLOWNESS(R_u.info, request)
3:   isTooMuch ← minSlowness > βR_u.maxSlowness
4:   if ¬ISROOT(R_u) ⋀ ¬FROMFATHER(request) ⋀ isTooMuch then
5:     reqFather ← request
6:   else
7:     numLeft ← AF(R_u.leftInfo, request.ℙℝ, minSlowness)
8:     EXTRACT(request, numLeft, reqLeft)
9:     reqRight ← request
10:  end if
11: end procedure
```

Algorithm 7.7 Get the minimum slowness that can be reached when assigning the tasks in `request` to the nodes described by `info`.

Pre: `info` is an availability summary. `request` is the request.
Post: `candidates.last.slowness` is the minimum slowness that can be reached when assigning the tasks in `request` to the nodes described by `info`.

```
1: function GETMINSLOWNESS(info, request)
2:   candidates ← ∅
3:   for all sf ∈ info do
4:     candidates ← candidates ⋃ {(sf, sf.z(info.a), 1)}
5:   end for
6:   PURGE(candidates, request.n)
7:   oneMoreTask ← true
8:   i ← 1
9:   while oneMoreTask do
10:    i ← i + 1
11:    oneMoreTask ← false                ▷ If nothing happens, this is the last iteration.
12:    for all (sf, slowness, n) ∈ candidates | n = i − 1 do
13:      if ESTIMATEZ(info.a, i) < candidates.last.slowness then
14:        candidates ← candidates \ {(sf, slowness, n)}
15:        candidates ← candidates ⋃ {(sf, ESTIMATEZ(info.a, i), i)}
16:        oneMoreTask ← true
17:      end if
18:    end for
19:    PURGE(candidates, request.n)
20:  end while
21:  return candidates.last.slowness
22: end function
```

Chapter 8.

Experimentation

"It doesn't matter how beautiful your theory is, it doesn't matter how smart you are.
If it doesn't agree with experiment, it's wrong."
— Richard P. Feynman

We have measured the scalability, fault-tolerance and performance of our proposal through a set of tests and simulations. We have first developed tests to evaluate the accuracy of the aggregation scheme. They are run by a specific evaluation program that aggregates the information of a set of nodes in the same way that would be done in the tree. Then, we have also developed an ad-hoc discrete event simulator (DES) to observe our model under more realistic conditions. This DES is written in C++, focused on minimizing the memory footprint to simulate as many nodes as possible. Its details can be found in Appendix B. The five presented policies have been tested, but only the IBP, MMP, DP and FSP policies are compared with each other. Due to its characteristics and it being a first attempt to schedule a different kind of application, the WDP policy results are presented in Section 8.6.

8.1. Aggregation Tests

We have studied the *accuracy* of the aggregation scheme for the different kind of parameters that have appeared in this thesis. To evaluate its scalability, we have varied the system size and the number of sampled functions per summary. We understand the accuracy of the aggregation scheme, applied to a set of nodes, as the fraction of their actual availability that is represented in the resulting summary. With 100% accuracy, the resulting summary perfectly represents the actual availability. The 0% accuracy corresponds to the minimum value that the scheme could potentially calculate. For instance, to aggregate the available memory of two nodes, we compute their minimum. So, the minimum of all the nodes would be an accuracy of 0%. Note that this is more restrictive than assigning a 0% accuracy to no available memory.

We have developed a specific evaluation program that calculates this fraction. The parameters of the program are the size of the network (N), the policy being tested and the maximum number of sampled functions per summary (SF_{max}). First, it generates the availability information of a set of N nodes, setting their properties and queue states at random with a uniform distribution. This distribution of values is the worst case for a clustering algorithm. Then, it aggregates this information recursively, imitating the organization of the nodes in a balanced binary tree. Finally, it compares the result with the actual availability of all the nodes and calculates the accuracy of the aggregation.

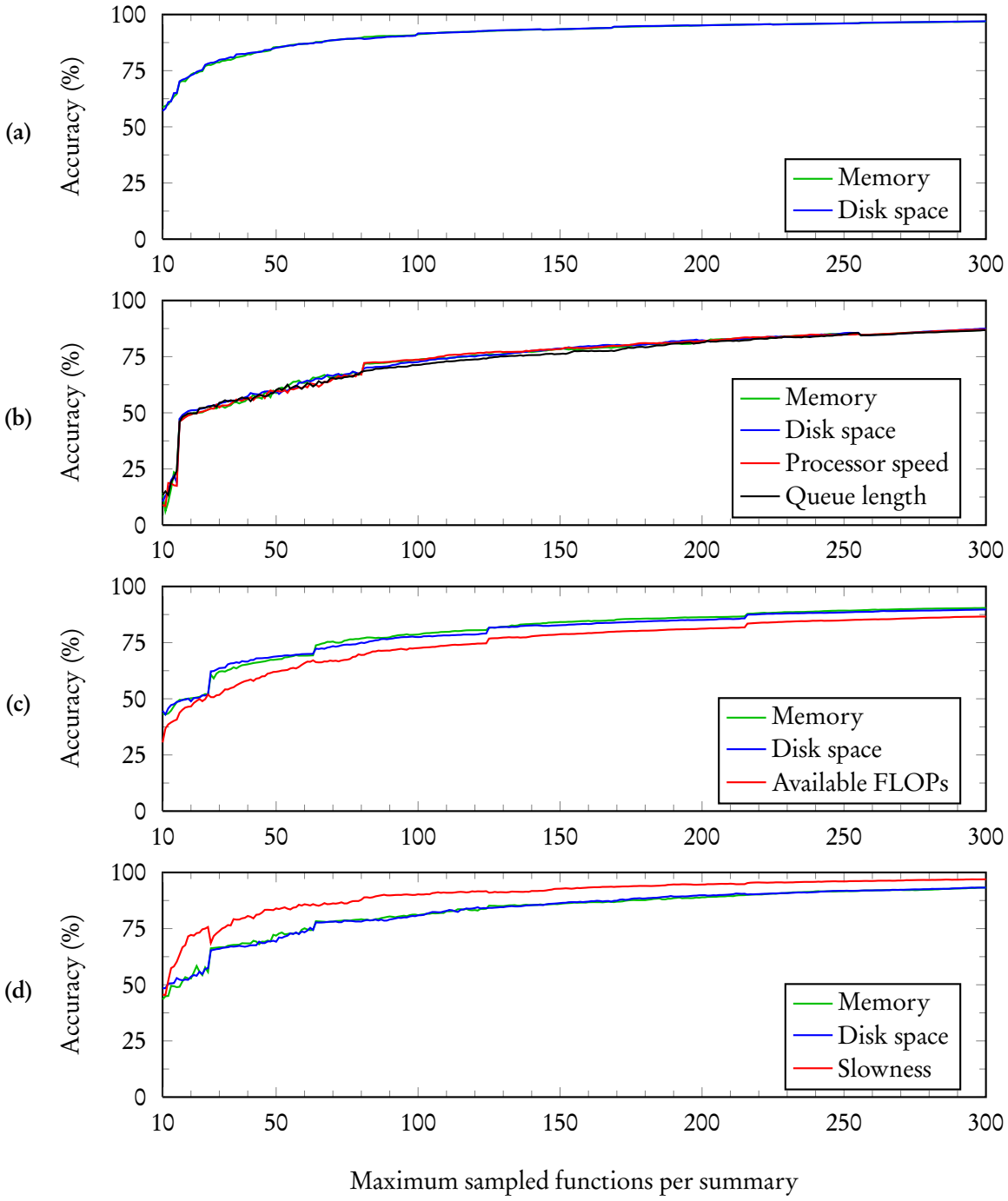

Figure 8.1.: Aggregation accuracy of a set of 1024 nodes for an increasing SF_{max}, for the (a) IBP, (b) MMP, (c) DP and (d) FSP policy parameters. The accuracy represents the fraction of the actual availability of a set of nodes that is represented in the aggregated summary. It is very similar for the scalar parameters.

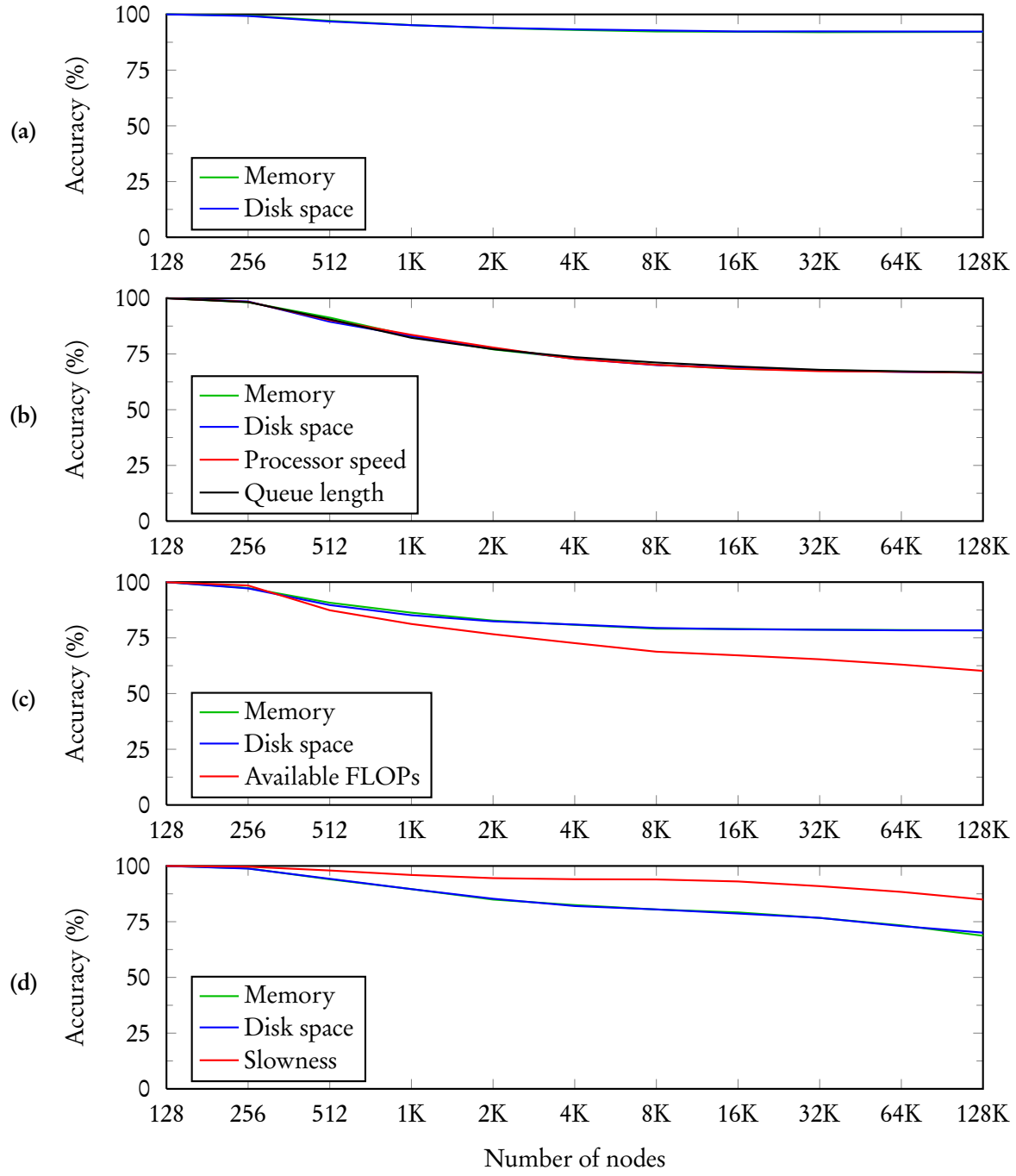

Figure 8.2.: Aggregation accuracy with 200 sampled functions per summary for an increasing number of nodes, for the (a) IBP, (b) MMP, (c) DP and (d) FSP policy parameters. The accuracy represents the fraction of the actual availability of a set of nodes that is represented in the aggregated summary. It is very similar for the scalar parameters.

We execute the program for each policy and several network sizes and SF_{max} values. At each step, we double the number of nodes to increment the tree height one more level.

Figure 8.1 shows the aggregation accuracy with different SF_{max}, for the four policies and a set of 1024 nodes, which would be found at level 10 of the tree. It can be seen that the accuracy quickly improves when we increase SF_{max} in the low value range. It hardly improves anymore when we increase SF_{max} over 200 sampled functions per summary. We show later how SF_{max} affects the network traffic and each policy performance, in order to choose an appropriate value for each situation. Then, Figure 8.2 presents the aggregation accuracy with an increasing number of nodes, for an SF_{max} of 200 and each policy. The decrease of accuracy is very low compared to the increase in the number of nodes. This figure also highlights that it is difficult to normalize the MSE of the functional parameters. The available amount of FLOPs before deadline of the DP policy gets noticeably less accuracy than the memory and disk space parameters, while the slowness per task length of the FSP policy gets more, because they are not correctly weighted in the distance operator. But in general, the clustering algorithm provides a well balanced accuracy among the different resource types, even when their values lay in very different intervals.

8.2. Simulation Setup

Besides the aggregation tests, we have also developed a DES that executes our scheduling model in a network of nodes for a certain amount of time. The network size is configurable, and there is direct communication between every pair of nodes. Nodes can send messages to each other, or messages can be injected to emulate the user behavior. To check the scalability of our model, we have simulated networks from five thousand up to a hundred thousand nodes. With the IBP policy, whose complexity is lower, we are able to simulate networks of up to a million nodes.

For every message, the DES takes into account both the transmission and computational times. The transmission time of a message is calculated by modeling the end-to-end link. We test our proposal with the end-to-end link of a typical volunteer computing platform, with 10 Mbps bandwidth and a delay between 50 ms and 300 ms, and of a fast cluster interconnection network, with 1 Gbps bandwidth and a delay between 0.1 ms and 1 ms. The delay follows a Pareto distribution, as suggested in [97]. The processing time of a message is the time needed by the simulation machine to process it. It could be scaled to the computing power of each node, but we decided to not do it. In this way, we are able to compare the processing time of a request in our model with the processing time in a single centralized machine. Only task execution time is calculated with the computing power of the execution nodes.

Simulations consist in having a user at each node, continuously submitting new jobs. For the generation of this workload, we have implemented the *site-level simulation model* proposed by Shmueli and Feitelson [86, 44]. It simulates the interaction between users and the system to calculate the timing and parameters of each submitted application. It was designed analyzing the influence of the scheduling performance on user decisions in several production system traces. These traces contain many years of activity of about 2000 users, with information of nearly half a million jobs. The DES also uses them to extract the distribution of application parameters and node properties (power, memory and disk), so we consider it realistic enough.

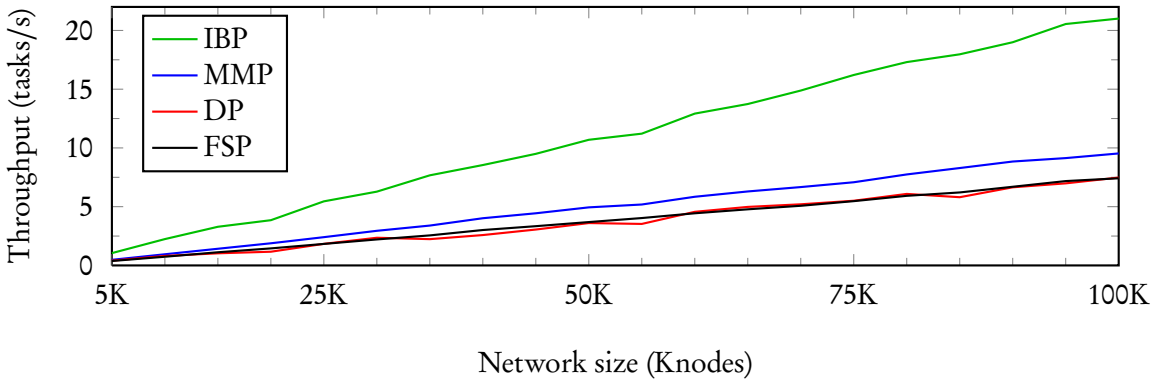

Figure 8.3.: Average throughput of each policy by network size. It depends on the workload characteristics, like the distribution of the number of tasks and release time.

A problem we faced with Shmueli and Feitelson's model is that the probability of a user having a break increases with the job turnaround time. Poorer performance usually yields longer job turnaround time. So, for certain performance metrics, like number of successfully finished tasks, the difference between two simulations' results may be larger than it should just because users are breaking more often and sending less jobs in the least performing one. For this reason, instead of applying the site-level model to every simulation as is, we apply it just once, generate our own trace of workload and replay it in the simulations we want to compare together. Then, users always submit the same amount of jobs and differences in the results are only due to our design.

Finally, we have developed a centralized version of each policy but the WDP. It is a centralized online scheduler that uses the same heuristics as its decentralized counterpart, but with full knowledge of the system state. In this way, it provides a reference to evaluate the advantages of a decentralized design against a centralized one. Their implementation and details can be found in Appendix C.

8.3. Scalability Results

We first show how our model preserves its scalable properties regardless of the policy being used through three metrics: the throughput, the allocation time and the bandwidth usage.

Figure 8.3 presents the average throughput of each policy for increasing number of nodes. Parameters other than the network size are kept constant. The absolute values are not relevant, because they depend on the workload characteristics of the site-level simulation model, and no policy tries to maximize the throughput. Instead, it is interesting to see that, for every policy, the throughput linearly increases with the system size.

We define the allocation time as the time elapsed between a request is submitted and all its tasks get accepted. It measures the cost of the task allocation algorithm as perceived by the user. Figures 8.4 and 8.5 compare the average allocation time of a request with a thousand tasks between the decentralized and centralized versions of all four policies, respectively. Figures 8.4a and 8.5a show the results of the decentralized and centralized versions of each policy in the slow network link model, while Figures 8.4b and 8.5b are plotted with the results of the fast one. As predicted in Section 3.3, in the decentralized case, the increase of allocation time with network size is near logarithmic, which

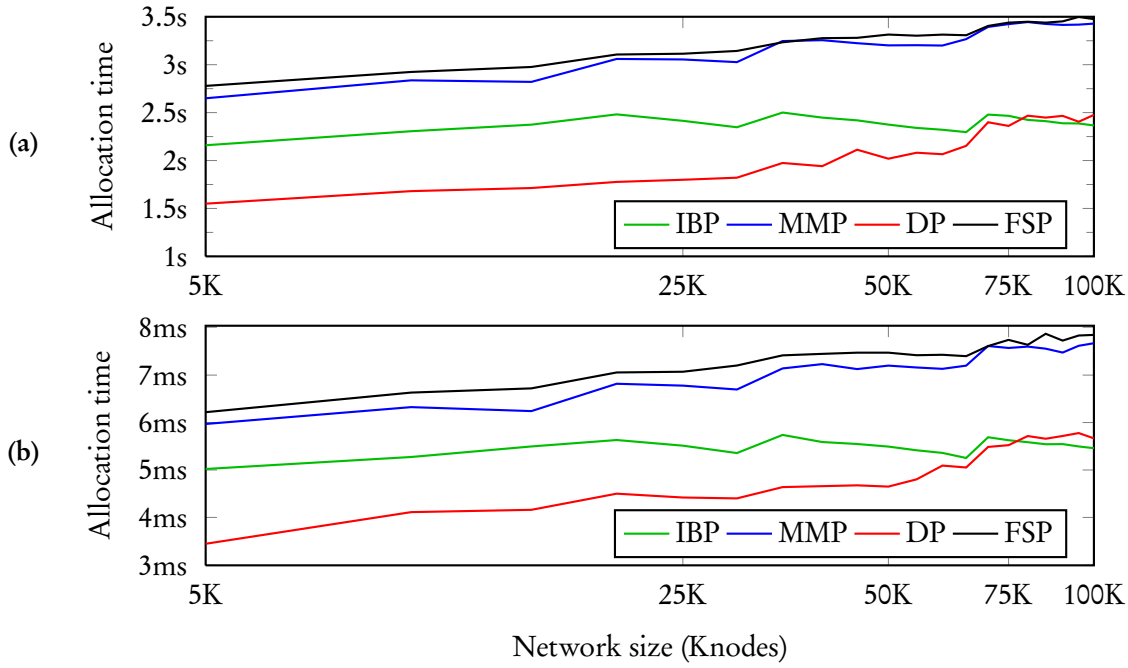

Figure 8.4.: Average allocation time against network size and policy, of a 1000 task request, with the (a) slow and (b) fast network link, by the decentralized versions of each policy.

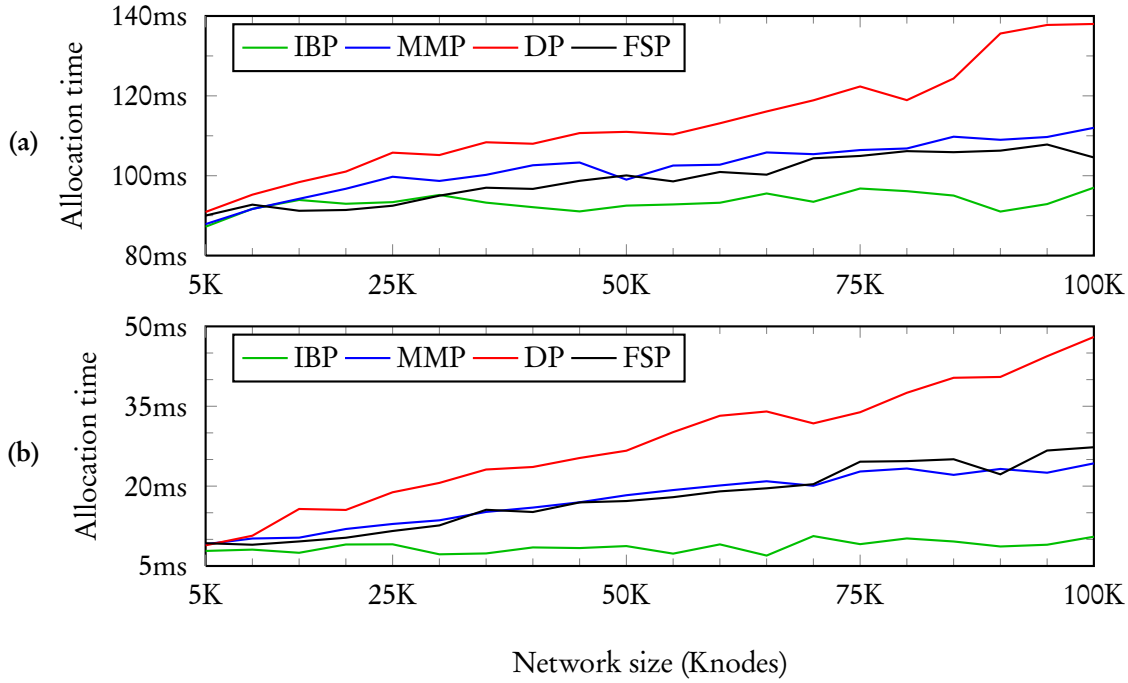

Figure 8.5.: Average allocation time against network size and policy, of a 1000 task request, with the (a) slow and (b) fast network link, by the centralized versions of each policy.

Table 8.1.: 99th percentile and maximum percentage of link bandwidth used by link model, policy, sampling interval and SF_{max}, on simulations with an update bandwidth limit of 100 KBps.

		Slow model			Fast model		
	SF_{max}	Average	10 sec.	1 sec.	Average	10 sec.	1 sec.
IBP	20 s.f.	.002(0.35)	.111(1.93)	.720(10.3)	<.001(.003)	.001(.010)	.003(.102)
	50 s.f.	.002(0.51)	.109(2.68)	.747(9.84)	<.001(.004)	.001(.015)	.004(.098)
	200 s.f.	.002(1.39)	.126(8.11)	.914(12.5)	<.001(.007)	.001(.019)	.005(.102)
MMP	20 s.f.	.002(0.70)	.615(13.4)	1.47(16.4)	<.001(.004)	.001(.011)	.007(.030)
	50 s.f.	.003(1.25)	.710(14.2)	1.60(17.0)	<.001(.008)	.003(.019)	.018(.058)
	200 s.f.	.008(3.62)	.862(15.6)	1.66(18.4)	<.001(.022)	.002(.042)	.016(.154)
DP	20 s.f.	.008(5.34)	.865(15.1)	7.04(17.4)	<.001(.026)	.001(.041)	.012(.065)
	50 s.f.	.014(10.0)	1.37(16.6)	10.9(19.4)	<.001(.054)	.002(.079)	.018(.126)
	200 s.f.	.017(15.8)	1.74(17.2)	12.3(26.3)	<.001(.145)	.003(.170)	.025(.238)
FSP	20 s.f.	.011(15.9)	.264(17.6)	1.64(30.8)	<.001(.159)	.002(.179)	.010(.314)
	50 s.f.	.019(15.9)	.410(17.9)	2.52(28.1)	<.001(.159)	.003(.185)	.019(.319)
	200 s.f.	.024(15.9)	.608(18.5)	3.80(36.7)	<.001(.160)	.005(.194)	.030(.338)

supports the scalability of our model. To show it, Figures 8.4a and 8.4b are plotted with a logarithmic x axis. The MMP and FSP policies are almost perfectly logarithmic, the IBP policy is even better than logarithmic, while the DP policy is a bit over the logarithmic behavior. They also show the difference between the routing patterns of each policy. The decentralized versions of the MMP and FSP policies are the slowest ones because they look for as much nodes as possible in order to fulfill their objective, while the DP and IBP policies can use the first nodes they find. On the other hand, in the centralized case, the allocation time is an almost linear function of the network size. For this reason, our decentralized model is much faster when used in large, fast networks, like a data center. This behavior would also bound the maximum size of a real centralized implementation, that would vary depending on the policy complexity. Among our proposed policies, it seems the DP policy would saturate the centralized scheduler at a lower system size.

To study the impact of our algorithms on the network traffic, we have measured the link bandwidth used on both link models. We have recorded the average bandwidth used throughout the simulation, and the peaks of usage, sampling at intervals of 10 and 1 second. Table 8.1 shows this values as a percentage of the link bandwidth, for both network models, the four policies and three SF_{max} values. Only the incoming traffic appears in the table, because it is always higher than the outgoing. The distribution of link usage among nodes is very skewed towards the low values. To illustrate this, for each configuration we present the 99th percentile and the maximum in parentheses. These simulations were performed with an update bandwidth limit of 100 KBps, thus giving a reference for the *very worst case*.

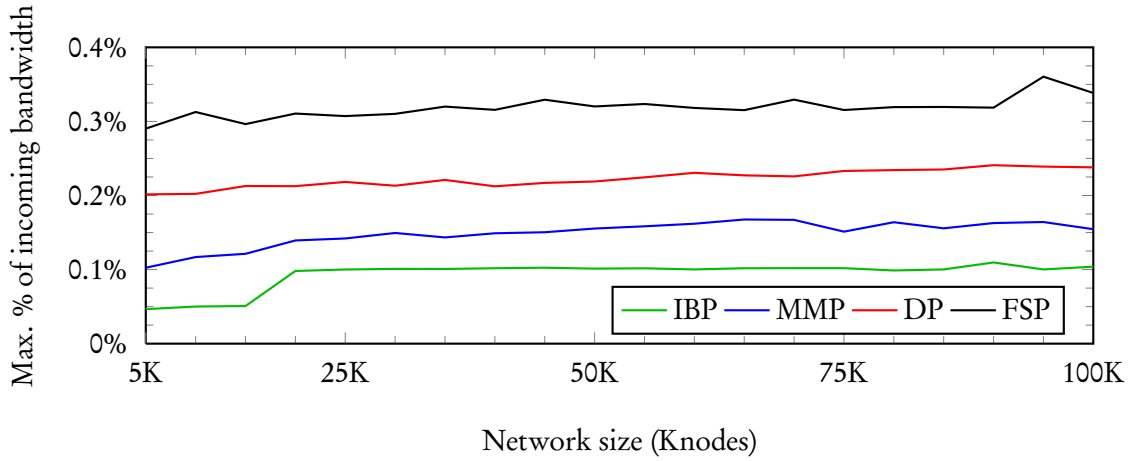

Figure 8.6.: Maximum percentage of link bandwidth used by policy and network size, with the fast link model, a sampling interval of 1 second and an SF_{max} of 200.

As expected, the traffic seems to be affected by the availability summary size and update frequency. It grows with the number of sampled functions per summary, but also with the size of each sampled function. The IBP and MMP policy, with just scalar parameters, use a small fraction of the bandwidth in average. Meanwhile, the DP and FSP policies have a higher average traffic due to their functional parameters. It can also be seen the great difference between the 99th percentile and the maximum in all the values of the table. This means that the peaks (of up to 20%-30% in the slow network model) are very rare. They are dominated by the update bandwidth limit of 100 KBps. In the slow network model, it is 10% of the link bandwidth, and in the fast model, it is 0.1%. So, if a node receives an update from both children simultaneously, if will suffer a burst of 20% or 0.2% of its incoming bandwidth, respectively. The possibility of receiving two updates at the same time depends on the frequency of the updates. It is higher in the DP and FSP policies, so they reach higher maximum peaks of traffic.

Figure 8.6 illustrates the evolution of the maximum percentage of link bandwidth for increasing number of nodes. These tests were performed with the fast link model, a sampling interval of 1 second and 200 sampled functions per summary. Besides its low values, it tends to stabilize instead of linearly growing, which is an indicator of scalability.

8.4. Policy Performance Results

The definition of performance depends on each policy objective, so we evaluate it separately for each one. We study how each parameter affects performance separately, except for failures, which are covered in the next section. First we analyze the impact of the network configuration, testing each policy for 60 hours with the maximum network size, a SF_{max} value of 200, both link models and an update bandwidth limit of 100, 1000 and 10000 bytes per second (Bps). These tests are repeated 5 times with the same trace but different random seed. We have seen that they only yield noticeable differences for the MMP and FSP policies. For the IBP and DP policies, we stick to using the fast

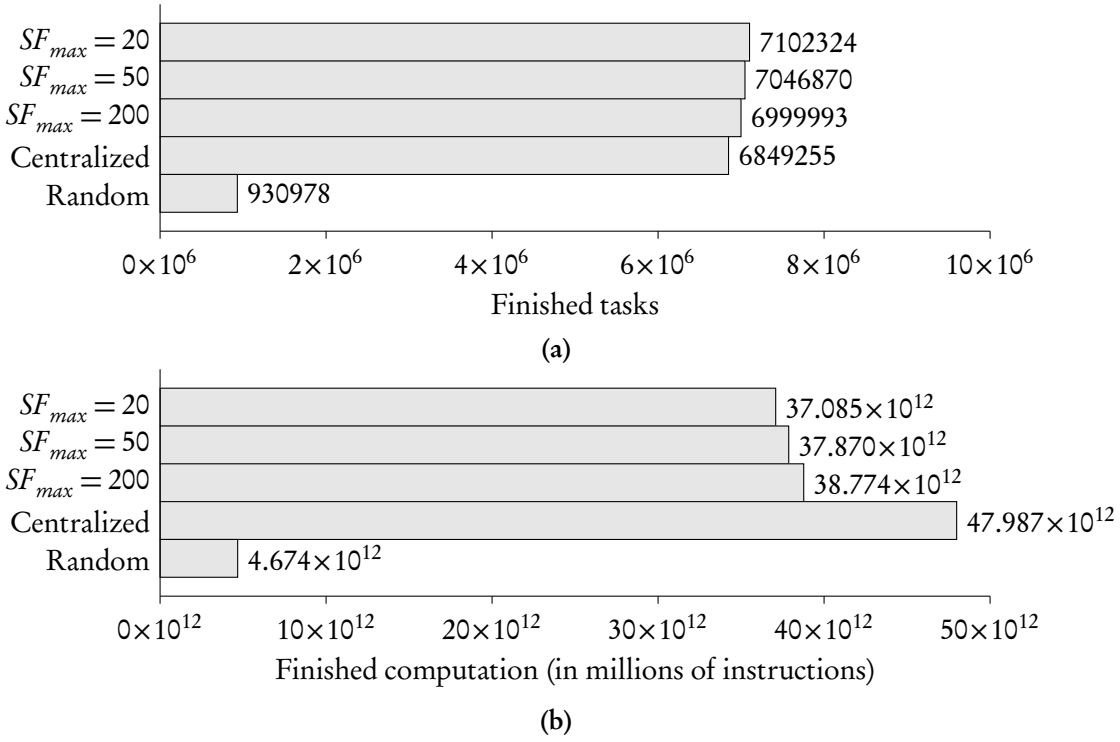

Figure 8.7.: (a) Finished tasks and (b) finished computation by the IBP policy, on a million nodes, for different values of SF_{max}, compared to its centralized version and a random allocation.

link model and an update bandwidth limit of 1 KBps. Once the network parameters are fixed, we simulate the decentralized version of each policy with different values of SF_{max}. To provide a context, we also simulate the centralized version and a trivial allocation of tasks to nodes at random. Then, we compare the performance results of all these simulations to see how far our model stays from the random allocation, and how near it gets to a centralized implementation.

8.4.1. IBP Policy

We measure the performance of the IBP policy with the amount of computation and the number of tasks successfully finished. We test a SF_{max} value of 20, 50 and 200 sampled functions per summary on a million node network. Figure 8.7a shows the number of tasks successfully finished when the simulation ends. The random allocation is far behind the others. However, it looks like the decentralized version finishes more tasks with less availability information, even more than the centralized version. We explain this surprising behavior with the distribution of the task length in the workload. The model we use generates a lot of short tasks and a few long ones. With better information, long tasks have more probabilities of being allocated, but then nodes get occupied for longer and many short tasks are not executed. So, the total amount of computation finished, in Figure 8.7b, behaves as expected. It shows little improvement in using 200 sampled functions over 50 or even 20, as expected from the accuracy tests, but it is still only around 20% behind the centralized version. Our conclusion is that a low summary size is enough for this policy.

8.4.2. MMP Policy

The performance of the MMP policy is measured as the maximum makespan obtained at each moment. As said before, it seems to be affected by the network parameters. The network configuration tests for this policy show important variations when the random seed changes, illustrated in Figure 8.8. Shaded regions represent the difference between minimum and maximum values obtained in the tests for each configuration. These differences narrow with a faster link and a higher update bandwidth limit. Test with the fast link model and an update bandwidth limit of 10 KBps show the most consistent results, colored in red in Figure 8.8b. So, the rest of the tests are performed with this network configuration.

Another important parameter for this policy is the β factor of the forwarding algorithm. It decides if the makespan obtained in a branch is short enough (see Section 4.3.2). This factor affects how high in the tree a request goes. The higher, the better the performance, but the higher the traffic. We want to obtain a good tradeoff between the performance of the policy and the traffic generated in the top levels of the tree. We have seen that for $\beta < 0.5$, the number of requests that reach the upper levels and the performance decrease very little. But at $\beta = 0.5$, the number of requests drops abruptly by around 200 times. Then, for $\beta > 0.5$, this value continues decreasing very slowly again, while performance drops significantly. So, $\beta = 0.5$ is the best value of this parameter.

With the network and β parameters set, we tested a SF_{max} value of 20, 50 and 200 sampled functions per summary on a hundred thousand nodes. Figure 8.9 shows the evolution of the maximum makespan throughout the simulation. The decentralized version with 50 sampled functions per summary performs like with 200, but it makes some wrong decisions that should be considered when selecting the SF_{max} value. However, both are very close to the centralized version. The decentralized version with 20 sampled functions per summary is omitted because of its bad results. So, for this policy, a summary size of at least 50 sampled functions should be used.

8.4.3. DP Policy

To evaluate the DP policy, we faced a problem with the site-level simulation model. It does not generate deadline information, because it did not appear in the original traces. So, we generate the deadline for an application that is going to be submitted, assuming that the user would like to have it finished before having a break or going to sleep. The deadline should make the user decide to continue working. As Shmueli and Feitelson calculated, the probability of continuing is $0.8/(0.05t_r + 1)$, where t_r is the response time of the last application in minutes. Then, a suitable deadline distribution would be $\Delta = (0.8/U - 1)/0.05$, where U is the standard uniform distribution. If the result would make an application finish while the owner is sleeping, then its deadline is set to the wake time, to provide weaker time restrictions. Additionally, we enforce a maximum and minimum values, δ_{max} and δ_{min}, for each application A_i. $\delta_{max} = a_i n_i / s_u$ is the time it would take in the submitting user's computer. $\delta_{min} = a_i n_i / S_{plat}$ is the time the user thinks it will take in the platform. To calculate δ_{min}, S_{plat} is the platform speed as perceived by the user. We compute it from the response time of the previously submitted applications, giving more weight to the most recent ones.

We evaluate the performance of the DP policy with the amount of computation and the number of tasks successfully finished, as for the IBP policy. The results are presented in Figure 8.10. Again, we test a SF_{max} value of 20, 50 and 200 sampled functions per summary on a hundred thousand nodes. The decentralized version with 20 sampled functions per summary performs quite poorly, but with

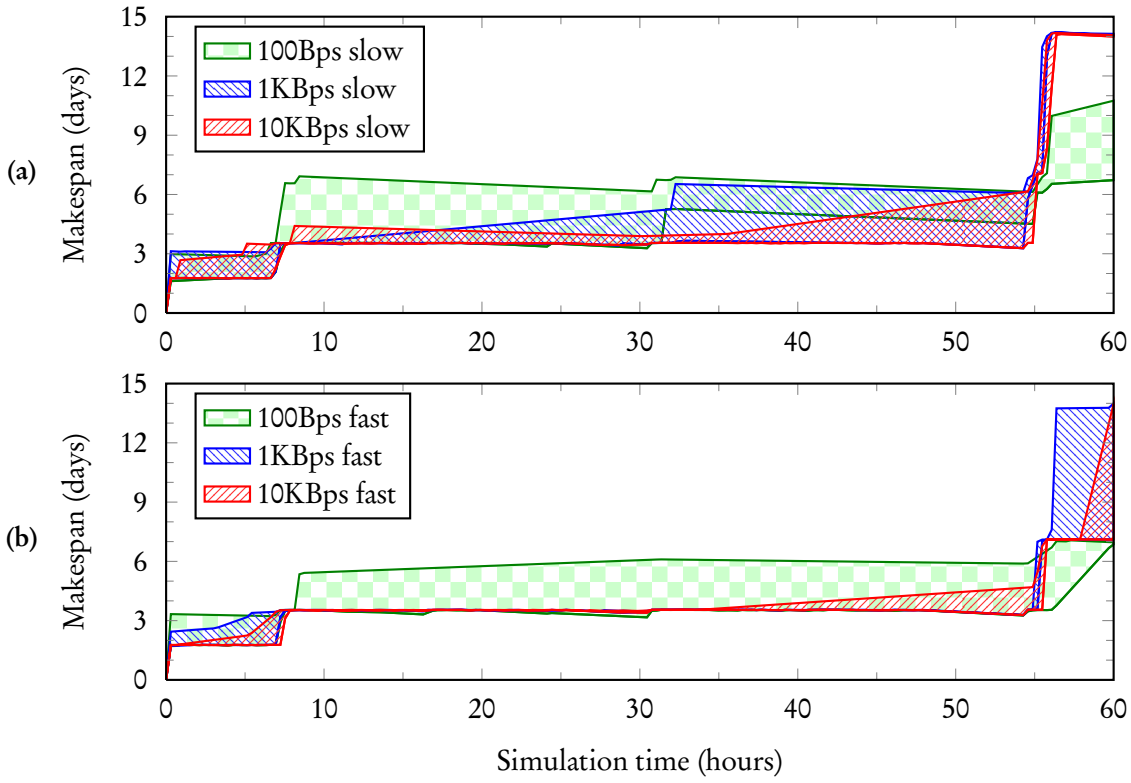

Figure 8.8.: Performance variation in 5 simulations of the MMP policy for different update bandwidth limit with (a) the slow link model and (b) the fast link model.

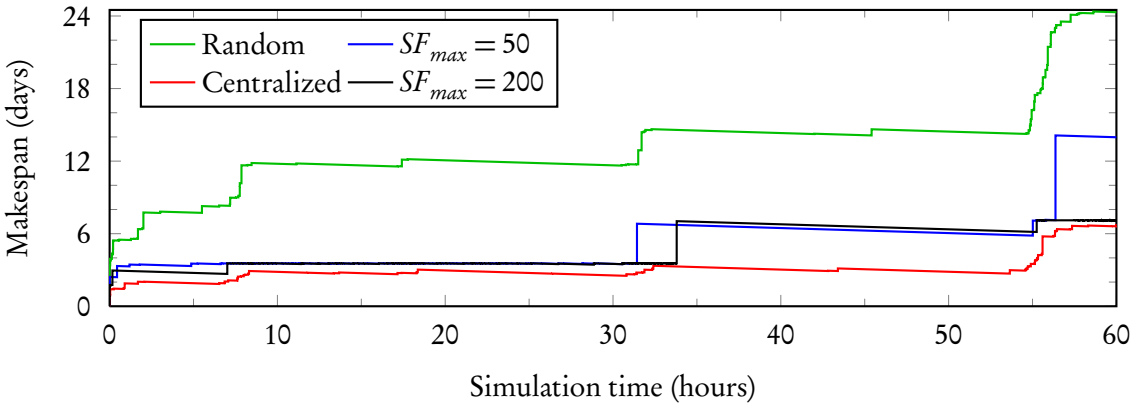

Figure 8.9.: Makespan by the MMP policy, on a hundred thousand nodes, for different values of SF_{max}, compared to its centralized version and a random allocation.

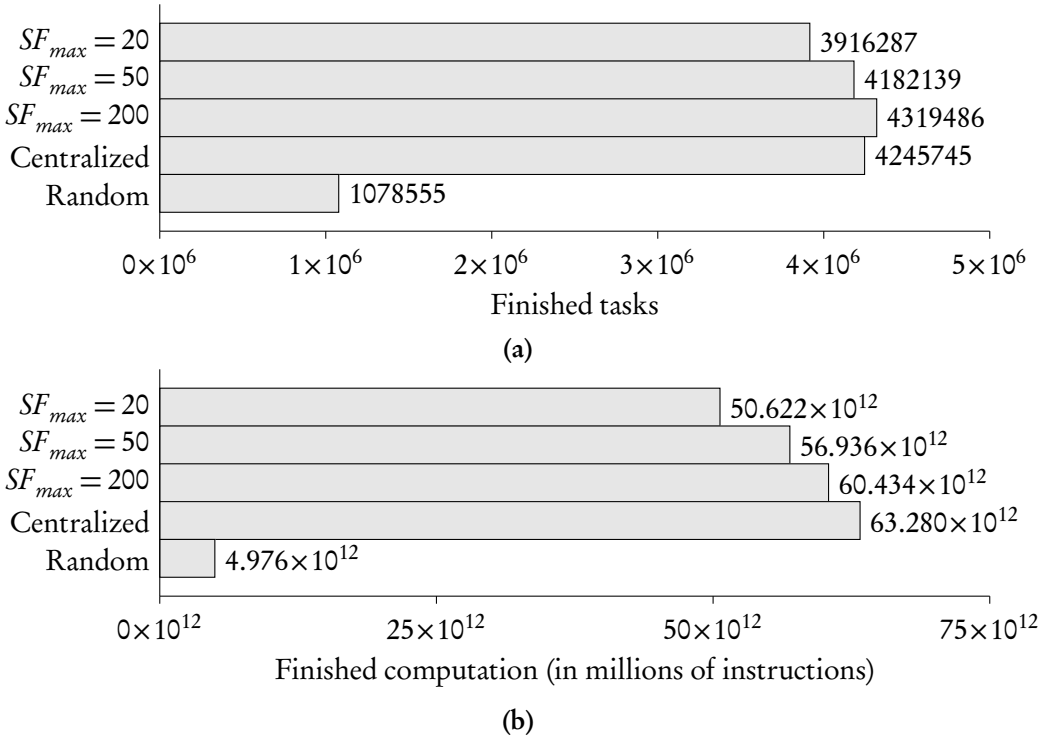

Figure 8.10.: (a) Finished tasks and (b) finished computation by the DP policy, on a hundred thousand nodes, for different values of SF_{max}, compared to its centralized version and a random allocation.

50 and 200 sampled functions per summary, it is only 10% and 4.5% behind the centralized version. With these values, 50 sampled functions per summary seem accurate enough and save bandwidth at the same time.

8.4.4. FSP Policy

Before presenting the FSP policy results, we must check that the site-level simulation model generates a set of applications and nodes that fulfill our assumptions. In Section 7.1.1 we stated that applications should have less tasks than nodes in the platform, that the distribution of computing power should change very little and that the fastest nodes should have a similar computing power. In fact, we already noted that the generated applications have no more than 1120 tasks. This is far less than the 100,000 nodes we are testing. Also, we never change the computing power distribution during the simulation. We think it is a reasonable scenario in most situations, even with nodes entering and leaving the system frequently. Finally, from the set of nodes generated, 1008 nodes have the maximum speed and 2000 have at least 93% of the maximum speed. So, we can safely say that the stretch of each application is proportional to its slowness, and that the maximum slowness is a good measure of the fairness.

Now we study the behavior of the FSP policy against different network configurations. There seems to be little difference between the slow and the fast network model, but the update bandwidth limit plays an important role. With 10 KBps, the maximum slowness was 1.75, with 1 KBps it was

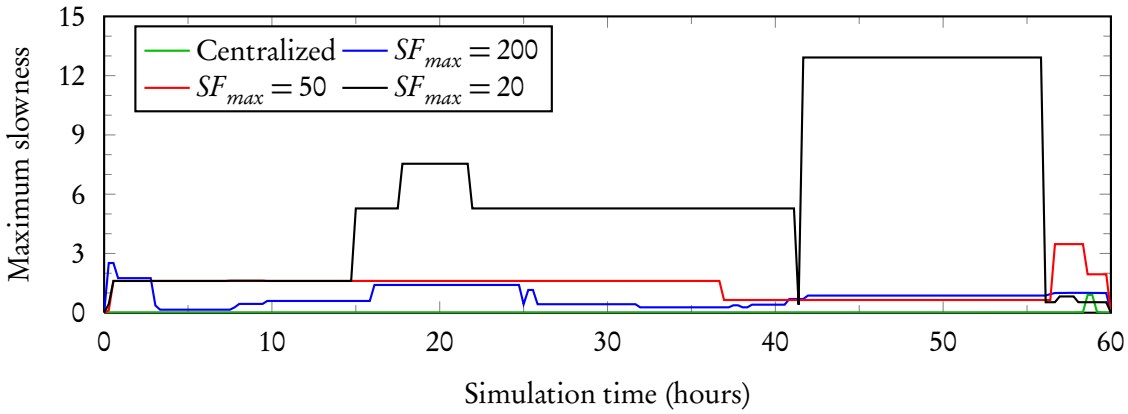

Figure 8.11.: Maximum slowness among coexisting applications during the simulation, by the FSP policy, on a hundred thousand nodes, for different values of SF_{max}, compared to its centralized version.

32.1 and with 100 Bps it was 58.4. We conclude that maintaining the availability information up to date is determinant for this policy. With these differences, we have decided to do the rest of the simulations with the fast link model and 10 KBps of update bandwidth limit.

Like in the MMP policy, we also have to set the β parameter to find a tradeoff between performance and traffic. In this case, the β parameter decides if the slowness obtained by the forwarding algorithm in a branch is short enough (see Section 7.3.2). We have tested the performance of the FSP policy for several values of β between 0.01 and 2. The maximum slowness is regular with $\beta < 1$, quickly increasing after this value. However, the number of requests that reach the root start growing significantly after $\beta = 0.04$. So, we think that $\beta = 0.04$ is the correct value for the following performance tests.

Finally, we measure the performance of the FSP policy by the maximum slowness among the applications that coexist in the system. Figure 8.11 shows how it evolves during the simulation on a hundred thousand nodes. We have plotted the results for a SF_{max} value of 20, 50 and 200 sampled functions per summary and for the centralized version. The random allocation, on the other hand, is off the charts. Its maximum slowness quickly grows to 138, remaining there the rest of the simulation. For the decentralized version, it can be seen that it presents a noticeable variation of performance among different values of SF_{max}. The performance with $SF_{max} = 200$ is up to 20 times that with $SF_{max} = 20$. However, we are still far away from the performance of the centralized version. After 5 hours of simulation, when it was nearest to the performance of the decentralized version with $SF_{max} = 200$, it was still about 40 times better.

8.5. Fault-tolerance Results

To test the fault-tolerance of our model, we also perform simulations in which nodes fail. As explained in Section 3.1.4, the consequences of a node failing are that all the tasks in its queue are aborted and the availability information of its branch is lost. The information about issued application requests and tasks waiting to be finished is saved to a database. We model failures as in [83]: every time a node leaves another one replaces it, so that the system maintains its size. We assume that the tree overlay

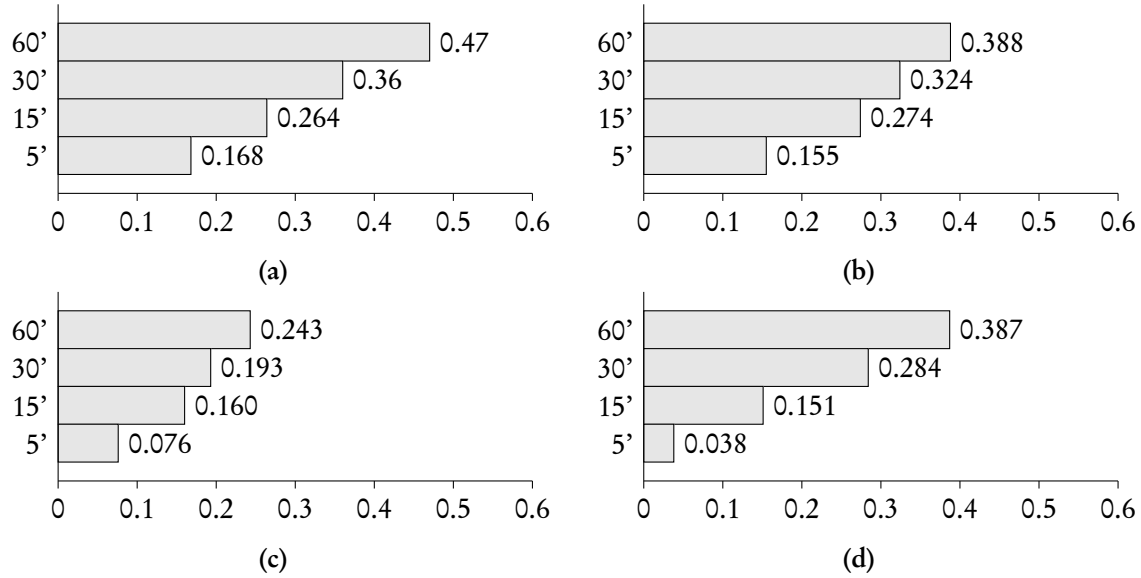

Figure 8.12.: Finished computation for simulations with churn, with median session times of 60, 30, 15 and 5 minutes, for the (a) IBP, (b) MMP, (c) DP and (d) FSP policies. It is normalized to the results without churn.

is one of the works cited in Section 3.1.4 and that it recovers by itself, but we do not consider its recovery overhead. That would depend on the actual implementation used, so we focus on the cost of recovering the components of our model. Then, the system automatically recovers by redistributing the availability information and resubmitting aborted tasks. We perform these failure tests with the same parameters as the performance tests, and an SF_{max} value of 200.

We simulate churn and catastrophic failures. Churn is the continuous process of node arrival and departure, very common in desktop grids and P2P computing platforms. In managed environments, like clusters and data centers, churn is very light. Rhea et al. [83] characterize churn by the median session time of a node. From the observed session times in various P2P systems, they suggest median session times from 5 to 60 minutes. We show how the computation finished by each policy degrades with churn in Figure 8.12, for median session times of 60, 30, 15 and 5 minutes. Values are normalized to the finished computation by each policy without churn.

We understand a catastrophic failure as the simultaneous failure of a large set of nodes. It is rare, but it is most significant in managed environments; e.g. due to a power loss in a data center, a cutting in an interconnection network or a misconfiguration propagated to several virtual instances. We tested our model with all four policies when a catastrophic failure occurs after 1 day of simulation, with 5%, 10%, 20% and 40% of the nodes failing. As soon as the underlying tree structure recovers from the failure, our model is able to quickly recreate the lost availability information and resubmit the aborted tasks, so the impact in global performance of each policy is minimal. The only noticeable effect is that the recovered applications obtain a longer response time, and in the DP policy some may not meet their deadline.

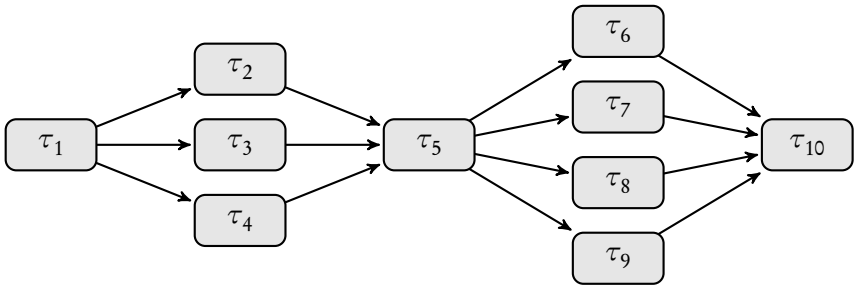

Figure 8.13.: A Fork-Join graph model with 10 tasks.

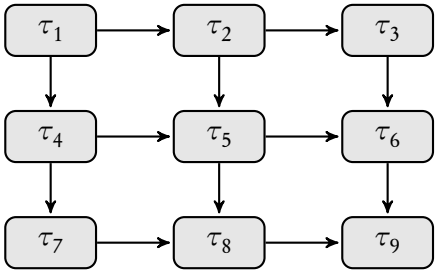

Figure 8.14.: A Laplace equation solver graph model with 9 tasks.

8.6. WDP Policy Tests

The WDP policy is different to the other policies in many aspects. It is a first approach to schedule a different kind of application. So, the site-level simulation model cannot be applied to this policy, because it is not designed to generate workflow applications. Also, the availability model has been simplified, and there is nothing like a sampled function in this policy. For these reasons, and to assess whether it is worth further development of this policy, we only present results on allocation time, speed-up and computational and network costs of our decentralized proposal. Additional experiments will be carried out in the future.

8.6.1. Simulation Setup

We have simulated this policy on the slow network model with one million nodes. As workflows, two different kinds of DAGs has been selected to get results on different sizes and workflow widths. The selected models are a Fork-Join and a Laplace equation solver-like graph, shown in Figures 8.13 and 8.14. The arrival of new workflows follows a Poisson process, with a mean interarrival time of 3 milliseconds. The mean arrival rate is twice the aggregated speed of all the platform, so it maintains every node busy at almost any time. Finally, the deadline of each workflow is calculated through the *workflow relative priority* W_j. It is the ratio between the time the submission node would need to execute the workflow and the time remaining until deadline. A value of 1 or less would mean that the sender could execute that workflow by itself, so higher values are tested.

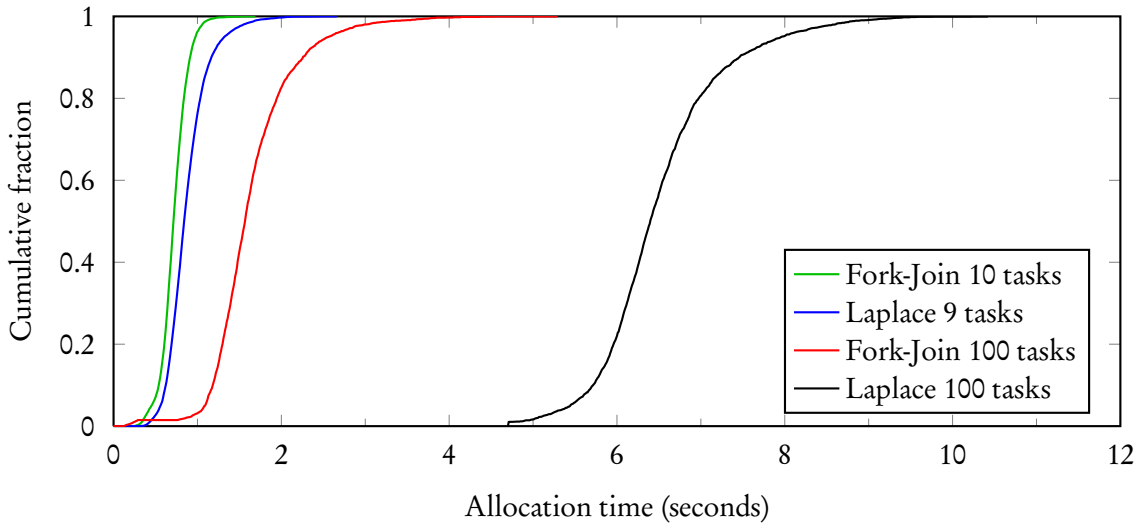

Figure 8.15.: Allocation time for different workflow widths, in a network of 1 million nodes.

8.6.2. Results

Probably, the system property that users firstly notice is response time. It depends on two factors: workflow allocation time and workflow execution time. The time a workflow needs to be allocated is directly related to its width. Figure 8.15 shows the cumulative distribution of the allocation time for both models and different sizes. As expected, the allocation time of fork-join workflows varies very little with their size, while it grows significantly for Laplace-like models, whose concurrency is lower. As with the other policies, the allocation is done in just a few seconds.

The workflow execution time is the time lapse between a request is sent and the last task is finished. We define the *speed-up* as the ratio between the time the submission node would need to execute the workflow by itself and the actual execution time. The ratio between workflow total length and minimum length provides the maximum speed-up that can be expected in execution nodes with similar computing power as the submission node. Table 8.2 presents average speed-up values registered for different workflow relative priorities. As it can be seen, the fork-join model has an expected maximum speed-up of 3.3 while the Laplace model could get a speed-up of just 1.8. Results show that for the Laplace model, the system performs better than expected, as tasks may run in faster machines than the client's one. However, in the fork-join model, the WDP policy is unable to reach the maximum expected speed-up. In both cases, the speed-up increases with the workflow priority as deadlines become tighter.

Finally, we present the network traffic of the WDP policy. It has been tested with links of 1Mbps of bandwidth and an update bandwidth limit of 10 KBps. The average traffic along the simulation is very low, so we have studied the peaks of traffic. Like with the other policies, we measure the traffic at intervals of one second, and register the maximum of all intervals. The average peak of outgoing traffic among all the nodes was 216 Bps, 75% of the nodes only generated less than 2500 Bps of peak traffic, and the maximum peak was 8200 Bps. Likewise, the average peak of incoming traffic was 190 Bps, about 75% of the nodes received less than 1000 Bps of peak traffic, just 1% of the nodes received

Table 8.2.: Average speedup by workflow model and priority.

Model	Maximum Speed-up	Workflow relative priority W_j			
		1.1	1.2	1.3	1.4
Laplace	1.8	2.05	2.09	2.18	2.22
Fork-Join	3.3	2.12	2.85	2.90	2.92

more than 8000 Bps, and the maximum value registered during the tests was 29000 Bps. Again, the impact of the communications in the user experience is very limited.

8.7. Comparison with Other Works

In this chapter we have quantitatively compared the scalability and performance of our decentralized model with a centralized one, which we have also implemented. But it would also be interesting to compare ourselves with some of the work presented in Chapter 2. However, this is often a difficult task, because each one uses different metrics and parameters. Many decisions made by the authors affect the results, like the policies being used, the generation of workload or the network model. So, we have made a more qualitative comparison. After studying the characteristics that the experiments of the related work have in common with us, we have selected four of them as indicators of scalability, fault-tolerance and versatility: The number of simulated nodes, to provide an idea of how far the authors pushed their implementations; the existence of a failure scenario; the implemented policies; and the workload generation, to evaluate if the simulations were realistic enough. Table 8.3 shows the results.

Decentralized resource discovery platforms offer the best numbers, but they obviously implement no scheduling policy. NodeWiz, SWORD and the work by Cardosa and Chandra show simulations of 10,000 nodes. NodeWiz authors even test a prototype implementation on 1,000 emulated nodes and 100 real nodes, and take failures into account. No other work shows results of tests with failing nodes. All of them use real data collected from platforms like PlanetLab [72]. The exception is Cohesion, whose authors only test platforms of 64 nodes with synthetic data.

The works on grid schedulers with aggregated information present the tests with the least number of nodes. Their scalability is limited because they usually only consider one broker per grid domain, and they use a centralized scheduler (Kokkinos and Varvarigos) or expect every domain to know each other (Brunner et al. and Rodero et al.). Surprisingly, Rahman et al. claim to decentralize their scheduling by using a DHT to index domain capabilities, but they only test it on 100 domains. On the other hand, they test several scheduling policies, including scientific workflows. Rodero et al. and Kokkinos and Varvarigos also use traces from the Grid Workloads Archive [91].

Finally, we consider that we have improved the state of the art in relation to other decentralized scheduling platforms. They experiment with less than 10,000 nodes, no node failures, only one policy and synthetic workloads. WaveGrid does not take node failures into account, but it migrates tasks when users reclaim their nodes. Kwan and Mupala mention the resilience of their unstructured network, but it is not tested. Kim et al. compare the performance of its model with an idealistic

Table 8.3.: Comparison of several distributed scheduling projects.

Project	N. nodes	Failure	Policies	Workload
Our model	1,000,000	Yes	Various	Traces
NodeWiz [12]	10,000	Yes	N.A.	Mixed
SWORD [7]	10,000	No	N.A.	Mixed
Cardosa and Chandra [26]	10,000	No	N.A.	Traces
Cai and Wang [23]	8,192	No	N.A.	Traces
WaveGrid [98]	5,000	No	IBP	Synthetic
Kwan and Mupala [61]	4,000	No	IBP	Poisson p.
Kim et al. [58]	1,000	No	MMP	Poisson p.
Kokkinos and Varvarigos [59]	1,000	No	Various	Traces
Brunner et al. [18]	1,000 dom.	No	Workflows	Synthetic
Rahman et al. [75]	100	No	Workflows	Poisson p.
Cohesion [85]	64	No	N.A.	Synthetic
Diet [27]	50	No	IBP	Synthetic
Rodero et al. [84]	18	No	various	Traces

centralized scheduler, like we do, with a MMP-like policy. Yet, we consider that 1,000 tested nodes are too few nowadays. Diet authors propose an extensible architecture, with plug-in policies, but only one is tested. Although only 50 nodes are tested, they perform tests on a full implementation, not on simulation. All these works generate their own synthetic workloads, usually with a Poisson process. They could present more realistic results with workloads coming from real system traces. In contrast, we test up to one million nodes, millions of tasks generated from real traces, failures of varying size and frequency, and five different policies.

8.8. Discussion

Returning to the properties we considered in Section 1.2 a distributed scheduling model should have, we can evaluate now if we accomplished our goals.

The model should be scalable, in the sense that it should be able to deal with an increase in the number of nodes without a noticeable impact in its performance. Our scalability results in Figures 8.4 and 8.5 show that the allocation cost has a logarithmic-like behavior with the system size, due to the concurrent allocation in different branches. We emphasize that our model can be faster than a centralized implementation in low delay interconnection networks, like those found in cloud computing facilities and data centers. Yet, it still manages to allocate a thousand tasks among a hundred thousand nodes with high link delay in less than 3.5 seconds. With its logarithmic behavior, we can extrapolate an average allocation time of less than 5 seconds with ten million nodes. Besides,

the communication overhead seems to be bounded even with an increase of the number of nodes. The bandwidth usage with the fast link model is negligible, and even the peaks with the slow link model can be considered low. This is due to the aggregation scheme, that effectively limits the traffic in the top levels of the tree. Finally, the throughput increases linearly with the number of nodes, without notice of deceleration. We consider reasonable to extrapolate these results and think that our model would behave similarly in higher scales than tested.

The model should also be fault-tolerant, degrading its performance but continuing its work in the case of failure. The tests on churn and catastrophic failures confirm that our model supports high rates of failures without loosing functionality, just degrading the performance accordingly. However, churn rises some special considerations. It can be seen that churn quickly affects the performance of the DP policy. This is because deadlines are a heavy requirement and resubmitted tasks cannot meet them. On the other hand, short session times badly affect the FSP policy, while it should be expected to behave like the MMP policy. Since the FSP policy has been recently developed, we have to further investigate the reason of this problem. In general, complex policies or policies with strong requirements should get worst results under churn. Additionally, most long tasks cannot finish with the shortest session times. To overcome this problem, users should also implement checkpointing or replication into their applications, to avoid resending failed tasks too often.

Finally, the model should be extensible to several different policies. By design, our generic availability information representation, tunable aggregation scheme and task routing approach make it feasible. With them, we have implemented five different policies of increasing complexity. We want to highlight that the most common IBP, MMP and DP policies obtain very good performance results compared to their centralized versions, while using availability information of very different level of detail. The WDP and FSP policies, however, still have room for improvement. The novelty of the former and the simplifications we made in the later yield moderate results. It is also worth bearing in mind that the MMP and FSP policies require a higher update bandwidth. The most probable cause is their routing pattern, but we must study this problem further.

Chapter 9.

Conclusions

"Yet in all those cases I finally steeled myself to seize the opportunity, and find a way to muddle through and eventually conclude that I had, in fact, chosen the right path, as risky as it seemed at the time."
— Vinton Cerf

In this thesis we presented a distributed scheduling model for large-scale platforms. It aggregates availability information about execution nodes on a hierarchical overlay. Then, using that information, it forwards tasks towards the most suitable execution nodes. We claim that it reaches scales of millions of nodes, tolerates high rates of failures and supports policies with very different objectives. We provide results from trace-driven simulation tests on a network of up to a million nodes.

The scalability is achieved through two main mechanisms. First, we propose an aggregation scheme that provides enough availability information to the top levels of the hierarchy without flooding them. An agglomerative clustering algorithm summarizes the availability information to avoid it growing without limit. Then, the update bandwidth is also bounded to reduce the network traffic. This scheme finds a good tradeoff between resource usage and accuracy. Second, we take a task-routing approach to scheduling. The nodes of the hierarchy use a forwarding algorithm that looks for execution nodes in several branches of the hierarchy concurrently, because they are independent. In our tests, the communication overhead is bounded and the allocation cost shows an almost logarithmic behavior with the system size. In networks of 100,000 nodes, link bandwidth of 1Gbps and link delays of under 1 millisecond, our decentralized scheduler can allocate tasks up to 10 times faster than its centralized equivalent. For instance, the decentralized version of the DP policy allocates a thousand tasks in 5ms, against 50ms of the centralized version. Meanwhile, the maximum bandwidth usage peak was under 0.25% of the link bandwidth. With delays of up to 300 milliseconds, the slowest policy still needs less than 3.5 seconds. The result is that our model is very well suited even for applications with thousands of short tasks, like many-task computing and map-reduce applications.

Faults are managed with a best-effort strategy. When a node failure is detected, the tasks it was executing are resubmitted by their owners and the availability information it managed is rebuilt by its neighbors. In this way, our model is able to degrade its performance accordingly and recover its functionality. To show it, we have performed tests of churn and catastrophic failures. Even with nodes failing with a median period of only 5 minutes, our scheduler is able to continue giving a degraded service. Meanwhile, it is able to recover from the failure of an important fraction of the nodes, as long as the underlying hierarchical overlay supports it.

Finally, we have designed our distributed scheduling model with extensibility in mind. The availability information model supports a generic set of operations to aggregate different properties of the execution nodes. Besides, the forwarding algorithm provides a common prototype to different realizations. So, we have implemented five policies, by specializing their own availability representation and forwarding algorithm. The IBP policy allocates bag-of-tasks applications to idle nodes as they become ready, if they meet the memory and disk space requirements. The MMP policy also considers queue length to minimize the global makespan. The DP policy allows the use of time constraints to schedule first the applications with a shorter deadline. The FSP policy introduces the concept of slowness to provide a fair share of the platform to every application. And the WDP policy explores the extension of the DP policy to a different application type, the workflow, that includes additional dependencies between tasks. We propose a common method to represent and clusterize scalar parameters of the execution node availability, like available memory and disk space. It is applied in the four policies for bag-of-tasks applications. Besides, for the DP and FSP policies, we also present a method to represent and clusterize functional parameters, like the available amount of FLOPs before a certain deadline.

After checking the feasibility of implementing policies with very different objectives on our scheduling model, we have also tested their performance. We have compared them with a scheduler that allocates tasks to nodes at random, and with a centralized version of each policy that has full knowledge of every execution node state. The simpler IBP, MMP and DP policies perform very close to a centralized implementation. On the other hand, the FSP and WDP policies, due to their novelty and the simplifications we have made, present more moderate results.

These results open the door to many possible improvements. The immediate one would be to improve the FSP and WDP policies, so that they yield results comparable to the other policies. Then, we could complete the missing parts and create a fully-fledged distributed computing platform, or integrate our code in an existing one. In this way we could test our model on a real scenario. Meanwhile, we want to study the generation and simulation of specific workloads, like many-task, map-reduce and data-intensive applications. Since many policies depend on the task length, we also plan to study different methods of task length estimation. Likewise, availability prediction models could provide better information about execution nodes to those policies where the future state is important, like the MMP and DP policies.

Bibliography

[1] The INET Framework Website. http://inet.omnetpp.org/, May 2013.

[2] ABERER, K., CUDRÉ-MAUROUX, P., DATTA, A., DESPOTOVIC, Z., HAUSWIRTH, M., PUNCEVA, M., AND SCHMIDT, R. P-Grid: A Self-organizing Structured P2P System. *ACM SIGMOD Record 32*, 3 (2003), 29–33.

[3] ABRAMSON, D., BETHWAITE, B., ENTICOTT, C., GARIC, S., AND PEACHEY, T. Parameter Exploration in Science and Engineering Using Many-Task Computing. *IEEE Transactions on Parallel and Distributed Systems 22*, 6 (2011), 960–973.

[4] ABRAMSON, D., GIDDY, J., AND KOTLER, L. High Performance Parametric Modelling with Nimrod-G. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing* (2000), IEEE, pp. 520–528.

[5] ABRAMSON, D., LEWIS, A., PEACHEY, T., AND FLETCHER, C. An Automatic Design Optimization Tool and its Application to Computational Fluid Dynamics. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing* (2001), ACM, p. 47.

[6] ABRAMSON, D., SOSIC, R., GIDDY, J., AND HALL, B. Nimrod: a Tool for Performing Parameterised Simulations Using Distributed Workstations. In *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing* (Aug. 1995), IEEE, pp. 112–121.

[7] ALBRECHT, J., OPPENHEIMER, D., VAHDAT, A., AND PATTERSON, D. A. Design and implementation trade-offs for wide-area resource discovery. *ACM Transactions on Internet Technology 8*, 4 (Oct. 2008), 18:1–18:44.

[8] ANDERSON, D. BOINC: A System for Public-Resource Computing and Storage. In *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing* (2004), IEEE, pp. 4–10.

[9] ANDERSON, D. P., COBB, J., KORPELA, E., LEBOFSKY, M., AND WERTHIMER, D. SETI@home: An Experiment in Public-Resource Computing. *Communications of the ACM 45*, 11 (2002), 56–61.

[10] ANDERSON, D. P., AND VII, J. M. Local Scheduling for Volunteer Computing. In *Proceedings of the 21th International Parallel and Distributed Processing Symposium (IPDPS)* (2007), IEEE, pp. 1–8.

[11] ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., STOICA, I., AND ZAHARIA, M. A View of Cloud Computing. *Communications of the ACM 53*, 4 (Apr. 2010), 50–58.

[12] BASU, S., COSTA, L., BRASILEIRO, F., BANERJEE, S., SHARMA, P., AND LEE, S.-J. NodeWiz: Fault-tolerant grid information service. *Peer-to-Peer Networking and Applications 2* (2009), 348–366. 10.1007/s12083-009-0030-1.

[13] BEAUMONT, O., CARTER, L., FERRANTE, J., LEGRAND, A., MARCHAL, L., AND ROBERT, Y. Centralized versus Distributed Schedulers for Bag-of-Tasks Applications. *IEEE Transactions on Parallel and Distributed Systems 19*, 5 (2008), 698–709.

[14] BENOIT, A., MARCHAL, L., PINEAU, J.-F., ROBERT, Y., AND VIVIEN, F. Scheduling concurrent bag-of-tasks applications on heterogeneous platforms. *IEEE Transactions of Computers 59*, 2 (2009), 202–217.

[15] BLANCO, H., AND CELAYA, J. Diseño de herramientas para la gestión de simulaciones distribuidas de modelos de red de gran escala. Master's thesis, Universidad de Zaragoza, Sept. 2007.

[16] BLYTHE, J., JAIN, S., DEELMAN, E., GIL, Y., VAHI, K., MANDAL, A., AND KENNEDY, K. Task Scheduling Strategies for Workflow-based Applications in Grids. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid* (2005), pp. 759–767.

[17] BRASILEIRO, F., ARAUJO, E., VOORSLUYS, W., OLIVEIRA, M., AND FIGUEIREDO, F. Bridging the High Performance Computing Gap: the OurGrid Experience. In *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid (CCGRID)* (2007), IEEE, pp. 817–822.

[18] BRUNNER, R., CAMINERO, A. C., RANA, O. F., FREITAG, F., AND NAVARRO, L. Network-aware summarisation for resource discovery in P2P-content networks. *Future Generation Computer Systems 28*, 3 (2012), 563–572.

[19] BUYYA, R., AND ABRAMSON, D. *The Nimrod/G Grid Resource Broker for Economic-based Scheduling*, vol. 75. Wiley Press, 2009, pp. 371–402.

[20] BUYYA, R., MURSHED, M., ABRAMSON, D., AND VENUGOPAL, S. Scheduling parameter sweep applications on global Grids: a deadline and budget constrained cost–time optimization algorithm. *Software: Practice and Experience 35*, 5 (2005), 491–512.

[21] BUYYA, R., AND VENUGOPAL, S. The Gridbus Toolkit for Service Oriented Grid and Utility Computing: an Overview and Status Report. In *Proceedings of the 1st IEEE International Workshop on Grid Economics and Business Models, GECON* (2004), IEEE, pp. 19–66.

[22] BUYYA, R., YEO, C. S., VENUGOPAL, S., BROBERG, J., AND BRANDIC, I. Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. *Future Generation Computer Systems 25*, 6 (2009), 599–616.

[23] CAI, M., AND HWANG, K. Distributed Aggregation Algorithms with Load-Balancing for Scalable Grid Resource Monitoring. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (march 2007), IEEE, pp. 1–10.

[24] CALLAGHAN, S., MAECHLING, P., DEELMAN, E., VAHI, K., MEHTA, G., JUVE, G., MILNER, K., GRAVES, R., FIELD, E., OKAYA, D. G. D., BEATTIE, K., AND JORDAN, T. Reducing Time-to-solution Using Distributed High-throughput Mega-workflows - Experiences from SCEC Cybershake. In *Proceedings of the Fourth IEEE International Conference on e-Science* (2008), IEEE, pp. 151–158.

[25] CAO, J., SPOONER, D., JARVIS, S., SAINI, S., AND NUDD, G. Agent-based Grid Load Balancing Using Performance-driven Task Scheduling. In *Proceedings of the International Parallel and Distributed Processing Symposium* (Apr. 2003), IEEE, p. 10.

[26] CARDOSA, M., AND CHANDRA, A. Resource Bundles: Using Aggregation for Statistical Large-Scale Resource Discovery and Management. *IEEE Transactions on Parallel and Distributed Systems 21*, 8 (2010), 1089–1102.

[27] CARON, E., DEPARDON, B., AND DESPREZ, F. Modelization and Performance Evaluation of the DIET Middleware. In *Proceedings of the 39th International Conference on Parallel Processing* (2010), IEEE, pp. 375–384.

[28] CARON, E., AND DESPREZ, F. Diet: A Scalable Toolbox to Build Network Enabled Servers on the Grid. *International Journal of High Performance Computing Applications 20*, 3 (2006), 335–352.

[29] CARON, E., DESPREZ, F., AND TEDESCHI, C. Dynamic Prefix Tree for Service Discovery within Large Scale Grids. In *Proceedings of the Sixth IEEE International Conference on Peer-to-Peer Computing* (2006), IEEE, pp. 106–116.

[30] CASANOVA, H., LEGRAND, A., AND QUINSON, M. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In *Proceedings of the Tenth International Conference on Computer Modeling and Simulation* (2008), IEEE, pp. 126–131.

[31] CATALÁN-SÁNCHEZ, V., AND CELAYA, J. Diseño e implementación de una capa de red P2P jerárquica, escalable y tolerante a fallos. Master's thesis, Universidad de Zaragoza, sept. 2012.

[32] CELAYA, J., AND ARRONATEGUI, U. Scalable Architecture for Allocation of Idle CPUs in a P2P Network. In *High Performance Computing and Communications*, M. Gerndt and D. Kranzlmüller, Eds., vol. 4208 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 240–249.

[33] CELAYA, J., AND ARRONATEGUI, U. YA: Fast and Scalable Discovery of Idle CPUs in a P2P network. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing (GRID)* (2006), IEEE, pp. 49–55.

[34] CELAYA, J., AND ARRONATEGUI, U. Distributed Scheduler of Workflows with Deadlines in a P2P Desktop Grid. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing* (2010), IEEE, pp. 69–73.

[35] CELAYA, J., AND ARRONATEGUI, U. A Highly Scalable Decentralized Scheduler of Tasks with Deadlines. In *Proceedings of the 12th IEEE/ACM International Conference on Grid Computing (GRID)* (2011), IEEE, pp. 58–65.

[36] CELAYA, J., AND ARRONATEGUI, U. A Task Routing Approach to Large-scale Scheduling. *Future Generation Computer Systems 29*, 5 (2013), 1097–1111.

[37] CELAYA, J., AND MARCHAL, L. A Fair Decentralized Scheduler for Bag-of-tasks Applications on Desktop Grids. In *Proceedings of the 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)* (may 2010), IEEE, pp. 538–541.

[38] DAM, M., AND STADLER, R. A generic protocol for network state aggregation. In *In Proceedings of Radiovetenskap och Kommunikation (RVK)* (june 2005), pp. 14–16.

[39] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM 51*, 1 (January 2008), 107–113.

[40] DINDA, P. A. Online Prediction of the Running Time of Tasks. *Cluster Computing 5*, 3 (2002), 225–236.

[41] DONG, F., AND AKL, S. G. Distributed Double-level Workflow Scheduling Algorithms for Grid Computing. *Journal of Information Technology and Applications 1*, 4 (2007), 261–273.

[42] ENGLERT, M., OZMEN, D., AND WESTERMANN, M. The Power of Reordering for Online Minimum Makespan Scheduling. In *Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)* (Oct.), IEEE, pp. 603–612.

[43] EPEMA, D., LIVNY, M., VAN DANTZIG, R., EVERS, X., AND PRUYNE, J. A Worldwide Flock of Condors: Load Sharing among Workstation Clusters. *Future Generation Computer Systems 12*, 1 (1996), 53–65.

[44] FEITELSON, D., AND SHMUELI, E. A Case for Conservative Workload Modeling: Parallel Job Scheduling with Daily Cycles of Activity. In *Proceedings of the IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems* (2009), IEEE, pp. 1–8.

[45] FLEISCHER, R., AND WAHL, M. Online Scheduling Revisited. In *Algorithms - ESA 2000*, M. Paterson, Ed., vol. 1879 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2000, pp. 202–210.

[46] GAREY, M. R., JOHNSON, D. S., AND SETHI, R. The Complexity of Flowshop and Jobshop Scheduling . *Mathematics of Operations Research 1*, 2 (1976), 117–129.

[47] GRAHAM, R. L. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal 45*, 9 (1966), 1563–1581.

[48] GUPTA, R., SEKHRI, V., AND SOMANI, A. K. CompuP2P: An Architecture for Internet Computing Using Peer-to-Peer Networks. *IEEE Transactions on Parallel and Distributed Systems 17*, 11 (2006), 1306–1320.

[49] HAHNE, E. Round-robin scheduling for max-min fairness in data networks. *IEEE Journal on Selected Areas in Communications 9*, 7 (1991), 1024–1039.

[50] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation* (2011), USENIX Association, pp. 22–22.

[51] HOCHBAUM, D. S., AND SHMOYS, D. B. Using dual approximation algorithms for scheduling problems theoretical and practical results. *Journal of the ACM (JACM) 34*, 1 (Jan. 1987), 144–162.

[52] INTANAGONWIWAT, C., GOVINDAN, R., ESTRIN, D., HEIDEMANN, J., AND SILVA, F. Directed diffusion for wireless sensor networking. *IEEE/ACM Transactions on Networking 11*, 1 (feb 2003), 2–16.

[53] IOSUP, A., OSTERMANN, S., YIGITBASI, M., PRODAN, R., FAHRINGER, T., AND EPEMA, D. H. J. Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. *IEEE Transactions on Parallel and Distributed Systems 22*, 6 (2011), 931–945.

[54] JAGADISH, H., OOI, B. C., VU, Q. H., ZHANG, R., AND ZHOU, A. VBI-Tree: A Peer-to-Peer Framework for Supporting Multi-Dimensional Indexing Schemes. In *Proceedings of the 22nd International Conference on Data Engineering* (2006), IEEE, p. 34.

[55] JAIN, A. K., MURTY, M. N., AND FLYNN, P. J. Data Clustering: A Review. *ACM Computing Surveys 31*, 3 (1999), 264–323.

[56] JARVIS, S. A., SPOONER, D. P., KEUNG, H. N. L. C., CAO, J., SAINI, S., AND NUDD, G. R. Performance prediction and its use in parallel and distributed computing systems. *Future Generation Computer Systems 22*, 7 (2006), 745–754.

[57] JUVE, G., AND DEELMAN, E. Resource Provisioning Options for Large-scale Scientific Workflows. In *Proceedings of the Third International Workshop on Scientific Workflows and Business Workflow Standards in e-Science (SWBES)* (2008), IEEE, pp. 608–613.

[58] KIM, J.-S., NAM, B., KELEHER, P., MARSH, M., BHATTACHARJEE, B., AND SUSSMAN, A. Trade-offs in Matching Jobs and Balancing Load for Distributed Desktop Grids. *Future Generation Computer Systems 24*, 5 (2008), 415–424.

[59] KOKKINOS, P., AND VARVARIGOS, E. Scheduling efficiency of resource information aggregation in grid networks. *Future Generation Computer Systems 28*, 1 (2012), 9–23.

[60] KONDO, D., ANDRZEJAK, A., AND ANDERSON, D. P. On Correlated Availability in Internet-Distributed Systems. In *Proceedings of the 9th IEEE/ACM International Conference on Grid Computing (GRID)* (2008), IEEE, pp. 276–283.

[61] KWAN, S. K., AND MUPPALA, J. Bag-of-Tasks applications scheduling on volunteer desktop grids with adaptive information dissemination. In *Proceedings of the 35th IEEE Conference on Local Computer Networks (LCN)* (oct. 2010), IEEE, pp. 544–551.

[62] LEGRAND, A., SU, A., AND VIVIEN, F. Minimizing the stretch when scheduling flows of biological requests. In *Proceedings of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2006), ACM, pp. 103–112.

[63] LENSTRA, J., SHMOYS, D., AND TARDOS, É. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming 46*, 1-3 (1990), 259–271.

[64] LIU, C. L., AND LAYLAND, J. W. Scheduling Algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM) 20*, 1 (1973), 46–61.

[65] LOGOTHETIS, D., AND YOCUM, K. Wide-scale data stream management. In *Proceedings of the USENIX 2008 Annual Technical Conference* (2008), USENIX Association, pp. 405–418.

[66] MAKHLOUFI, R., BONNET, G., DOYEN, G., AND GAÏTI, D. Decentralized Aggregation Protocols in Peer-to-Peer Networks: A Survey. In *Modelling Autonomic Communications Environments*, J. Strassner and Y. Ghamri-Doudane, Eds., vol. 5844 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009, pp. 111–116.

[67] MEDEL, V., AND CELAYA, J. Diseño de un Overlay Jerárquico y Tolerante a Fallos. Master's thesis, Universidad de Zaragoza, apr 2013.

[68] MO, J., AND WALRAND, J. Fair end-to-end window-based congestion control. *IEEE/ACM Transactions on Networking 8*, 5 (Oct. 2000), 556–567.

[69] MUTHUKRISHNAN, S., RAJARAMAN, R., SHAHEEN, A., AND GEHRKE, J. Online scheduling to minimize average stretch. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science* (1999), IEEE, pp. 433–443.

[70] The Omnet++ Website. http://www.omnetpp.org, May 2013.

[71] OZDEMIR, S., AND XIAO, Y. Secure Data Aggregation in Wireless Sensor Networks: A Comprehensive Overview. *Computer Networks 53*, 12 (2009), 2022–2037.

[72] PlanetLab Webpage. http://planet-lab.org, nov 2012.

[73] PMC, A. H. Apache Hadoop website. http://hadoop.apache.org/, January 2011.

[74] PRIETO, A., AND STADLER, R. A-GAP: An Adaptive Protocol for Continuous Network Monitoring with Accuracy Objectives. *IEEE Transactions on Network and Service Management 4*, 1 (june 2007), 2–12.

[75] RAHMAN, M., RANJAN, R., AND BUYYA, R. Cooperative and decentralized workflow scheduling in global grids. *Future Generation Computer Systems 26*, 5 (2010), 753–768.

[76] RAICU, I., FOSTER, I., AND ZHAO, Y. Many-Task Computing for Grids and Supercomputers. In *Proccedings of the Workshop on Many-Task Computing on Grids and Supercomputers* (Nov. 2008), IEEE, pp. 1–11.

[77] RAICU, I., ZHAO, Y., DUMITRESCU, C., FOSTER, I., AND WILDE, M. Falkon: a Fast and Light-weight tasK executiON framework. In *Proceedings of the ACM/IEEE Conference on Supercomputing* (Nov. 2007), ACM, pp. 1–12.

[78] RAMAMRITHAM, K., STANKOVIC, J., AND ZHAO, W. Distributed Scheduling of Tasks with Deadlines and Resource Requirements. *IEEE Transactions on Computers 38*, 8 (Aug. 1989), 1110–1123.

[79] RANJAN, R., RAHMAN, M., AND BUYYA, R. A Decentralized and Cooperative Workflow Scheduling Algorithm. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid)* (2008), IEEE, pp. 1–8.

[80] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R., AND SCHENKER, S. A Scalable Content-Addressable Network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '01)* (2001), ACM, pp. 161–172.

[81] RATNASAMY, S., HANDLEY, M., KARP, R., AND SHENKER, S. Topologically-aware Overlay Construction and Server Selection. In *Proceedings of the Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002)* (2002), vol. 3, IEEE, pp. 1190–1199.

[82] RENESSE, R. V., BIRMAN, K. P., AND VOGELS, W. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems 21*, 2 (2003), 164–206.

[83] RHEA, S., GEELS, D., ROSCOE, T., AND KUBIATOWICZ, J. Handling Churn in a DHT. In *Proceedings of the USENIX Annual Technical Conference* (2004), USENIX Association.

[84] RODERO, I., GUIM, F., CORBALAN, J., FONG, L., AND SADJADI, S. M. Grid broker selection strategies using aggregated resource information. *Future Generation Computer Systems 26*, 1 (2010), 72–86.

[85] SCHULZ, S., BLOCHINGER, W., AND HANNAK, H. Capability-Aware Information Aggregation in Peer-to-Peer Grids. *Journal of Grid Computing 7*, 2 (2009), 135–167. 10.1007/s10723-008-9114-z.

[86] SHMUELI, E., AND FEITELSON, D. G. On Simulation and Design of Parallel-Systems Schedulers: Are We Doing the Right Thing? *IEEE Transactions on Parallel and Distributed Systems 20*, 7 (2009), 983–996.

[87] SPOONER, D., JARVIS, S., CAO, J., SAINI, S., AND NUDD, G. Local grid scheduling techniques using performance prediction. *IEE Proceedings on Computers and Digital Techniques 150*, 2 (2003), 87–96.

[88] STOICA, I. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking 11*, 1 (2003), 17.

[89] TAKEFUSA, A., CASANOVA, H., MATSUOKA, S., AND BERMAN, F. A Study of Deadline Scheduling for Client Server Systems on Computational Grid. In *Proceedings of 10th IEEE International Symposium on High Performance Distributed Computing* (2001), IEEE, pp. 406–415.

[90] TANNENBAUM, T., WRIGHT, D., MILLER, K., AND LIVNY, M. Condor: a distributed job scheduler. In *Beowulf cluster computing with Linux*. MIT Press, Cambridge, MA, USA, 2002, ch. Condor: a distributed job scheduler, pp. 307–350.

[91] The Grid Workloads Archive. http://gwa.ewi.tudelft.nl/pmwiki, nov 2012.

[92] VARGA, A. OMNeT++. In *Modeling and Tools for Network Simulation*. Springer, 2010, pp. 35–59.

[93] VISHNUMURTHY, V., CHANDRAKUMAR, S., AND SIRER, E. G. KARMA: A Secure Economic Framework for P2P Resource Sharing. In *Proceedings of the Workshop on the Economics of Peer-to-Peer Systems* (2003).

[94] WILHELM, R., ENGBLOM, J., ERMEDAHL, A., HOLSTI, N., THESING, S., WHALLEY, D., BERNAT, G., FERDINAND, C., HECKMANN, R., MITRA, T., MUELLER, F., PUAUT, I., PUSCHNER, P., STASCHULAT, J., AND STENSTRÖM, P. The worst-case execution-time problem–overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems 7*, 3 (May 2008), 36:1–36:53.

[95] XIAO, L., ZHU, Y., NI, L. M., AND XU, Z. Incentive-Based Scheduling for Market-Like Computational Grids. *IEEE Transactions on Parallel and Distributed Systems 19*, 7 (2008), 903–913.

[96] YALAGANDULA, P., AND DAHLIN, M. A Scalable Distributed Information Management System. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '04)* (2004), ACM, pp. 379–390.

[97] ZHANG, W., AND HE, J. Modeling End-to-End Delay Using Pareto Distribution. In *Preceedings of the Second International Conference on Internet Monitoring and Protection (ICIMP)* (July 2007), IEEE, p. 21.

[98] ZHOU, D., AND LO, V. WaveGrid: a Scalable Fast-turnaround Heterogeneous Peer-based Desktop Grid System. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium* (2006), IEEE, pp. 28–37.

Appendix A.
Notation

Table A.1.: Notation in text and algorithms.

Notation	Description
A_i	Application.
$\mathbb{PR}_i$	Properties of application A_i.
r_i	Release time of A_i.
e_i	End time of the last task of A_i.
n_i	Number of tasks of A_i.
a_i	Length of a task of A_i, in millions of FLOPs.
m_i	Required memory to execute a task of A_i, in megabytes.
d_i	Required disk space to execute a task of A_i, in megabytes.
q_i	Desired makespan for A_i.
δ_i	Deadline of A_i.
z_i	Slowness of A_i.
P_u	Physical node.
$R_u/E_u/S_u$	Routing/Execution/Submission node role.
$\mathbb{AF}_u$	Availability function of node E_u.
s_u	Computing power of node E_u, in millions of FLOPs per second.
M_u	Memory available at node E_u, in megabytes.
D_u	Disk available at node E_u, in megabytes.
Q_u	Queue end time of node E_u.
$l_u(\delta)$	Amount of FLOPs available before δ at E_u.
$z_u(a)$	Maximum slowness at E_u adding one task of length a.
x, y, z	Availability summaries.
f, g, h	Sampled functions.
$f.p$	Parameter of a sampled function.

Continues on next page.

Table A.1.: Continued from previous page.

Notation	Description
$\text{SUM}(f, g)$	Sum operation of two sampled functions.
$\text{DIST}(f, g)$	Distance operation of two sampled functions.
τ_k	Task in position k of the queue.
β	Factor to adjust MMP and FSP forwarding algorithm.
ν	Current time.
SF_{max}	Maximum summary size.
`request`	An application scheduling request.
`request.srcAddr`	Request source address.
`request.`$\mathbb{PR}$	Application properties of a request.
`request.`p	One of the properties in `request.`$\mathbb{PR}$.
`request.`n	Number of tasks in a request.
R_u`.fatherAddr`	Address of a routing node father.
R_u`.leftAddr`	Address of a routing node left child.
R_u`.rightAddr`	Address of a routing node right child.
R_u`.leftInfo`	Availability information of the left child.
R_u`.rightInfo`	Availability information of the right child.
R_u`.info`	The aggregation of R_u`.leftInfo` and R_u`.rightInfo`.
$E_u.p$	A parameter of execution node E_u.

Appendix B.

The STaRS Simulator

To evaluate the scalability, fault-tolerance and performance of our model, we have developed a simulator in C++. The simulator code is available at http://webdiis.unizar.es/~jcelaya/stars. Our main reason to develop it instead of using an existing alternative was to reduce its memory footprint. In this way, we are able to simulate up to a million nodes with a simple policy, like the IBP.

B.1. History

The development of a simulator for the STaRS model started in 2005, to obtain the results of [32]. We built it with Omnet++ [92, 70], a DES for computer networks. It includes the INET framework [1], a good TCP stack and topology model, but we decided to use a simpler network model. The INET framework introduces too much overhead, so we simulated our scheduling model over a star network with fixed link bandwidth and delay. With 2 gigabytes of memory we were able to simulate 50,000 nodes.

One of the reasons for using Omnet++ was the possibility of distributing the simulation among several computers, to increase its speed and/or memory usage. However, at that time, it was an experimental feature, it only used a very conservative synchronization protocol and it had no way of correctly stopping the simulation. So, a Master Thesis [15] was carried out to fix these problems.

Meanwhile, we started the implementation of a new DES engine. Its objective was to provide only the features of Omnet++ that we needed, stripping everything else out, to reduce the memory footprint as much as possible. Now we are able to simulate one million nodes of the IBP policy in under 10 gigabytes of memory.

B.2. Design

Our DES engine consists of three main classes:

- **Simulator:** An object of this class drives the simulation. It contains a set of nodes, a network model and an event queue. It iteratively extracts the next event from the queue and sends it to its destination node. The node processes the event, and may generate new events that are inserted in the queue. The network model calculates when the events arrive at its destination.

- **StarsNode:** Each object of this class is a node in the network. It implements the distributed scheduling model presented in this thesis, with the different components of each node role. The model code is decoupled from the simulation engine, so it can be used in a future real

implementation. The simulator maintains a table with as many instances of this class as nodes in the network.

- **SimulationCase:** An object of this class prepares the simulation and decides how it evolves. For instance, the site-level simulation model presented in Section 8 is implemented in a class derived from SimulationCase. At the beginning of the simulation, a SimulationCase instance is created and configured with the contents of a configuration file.

In each simulation, a single instance of the Simulator class is created. It then creates a SimulationCase instance and configures it with the contents of a file, that is provided through the command line. This file is just a set of arbitrary key-value pairs. We have a helper tool that generates several configuration files for parameter-sweep simulations. Then, the SimulationCase instance generates the initial events, and the Simulator starts processing the event queue. For the transmission events, the simulator uses a simple network model. It simulates end-to-end links between every pair of nodes, with fixed bandwidth and a random delay that follows a Pareto distribution, as explained in Section 8. It also simulates the transmission and reception queue of every node. The simulation ends when the queue is empty, or certain configurable conditions are met, like reaching a maximum elapsed time or number of submitted requests.

Besides these three classes, there are other ones that provide extra functionality. There is an implementation of the different policies in a centralized scheduler, an in-memory implementation of the submission node application database, and several classes that gather data during the simulation, like throughput, traffic and performance statistics.

B.3. Future Work

Having reduced the memory footprint of the simulator, we are able to simulate huge networks in a reduced amount of memory. However, we have lost important functionality that other simulators provide. The two most important features we miss is a more realistic network model and the distributed simulation. For this reason, we are evaluating the reimplementation of core elements of our DES engine with either Omnet++ or SimGrid [30]. They provide these features to some extent, and are worth bearing in mind.

Appendix C.

Centralized Version of each Policy

C.1. IBP Policy

The centralized version of the IBP policy is shown in Algorithm C.1. It knows exactly how much memory and disk space is available at every node, and whether they are idle or busy. So, it creates a list of the nodes that fulfill the application requirements and sorts it as in the decentralized version. Then, a task is sent to each of the best nodes as long as there are enough. It is trivial to see that both loops have a time complexity of $O(n)$, where n is the number of nodes, and the sort procedure can be performed with complexity $O(n \log n)$ with a heap structure. Meanwhile, the space complexity is also $O(n)$.

C.2. MMP Policy

The centralized version of the MMP policy appears in Algorithm C.2. Again, it uses the same heuristic as the decentralized one, allocating tasks to those nodes whose queue will remain shortest afterward. Besides their available memory and disk space, it also records information about the queue end time of every node. The algorithm starts creating a list `candidates` of nodes that fulfill the application requirements, and their queue end time if they accept a task of the new application. This list is sorted by the queue end time. Then, each task is sent to the node that will finish it earlier. The queue end time of that node is updated, it is inserted in the list and the list is sorted again.

The first loop has a time complexity of $O(n)$, and the sort procedure can be performed with complexity $O(n \log n)$ with a heap structure, like in the IBP policy. The second loop has n repetitions at most, but it includes a sorting operation. Since it just inserts a new element in an already sorted heap, its complexity is just $O(\log n)$. In the end, the whole algorithm can be performed with complexity $O(n \log n)$. The space complexity is still $O(n)$.

C.3. DP Policy

The centralized version of this policy can be seen in Algorithm C.3. Its workings are similar to the previous two policies. First, it fills the list `holes` with the nodes that can execute at least one of the new tasks before its deadline. For each node, it calculates the amount of FLOPs that remain free after allocating as many new tasks as possible. Then, the list is sorted so that the nodes with less remaining FLOPs come first, and tasks are allocated to them.

The first loop is repeated for every node in the system, but its body contains a call to the function $l_u(\texttt{request}.\delta_i)$, that returns the amount of FLOPs that can be executed by node E_u before $\texttt{request}.\delta_i$.

Algorithm C.1 Centralized version of the IBP policy forwarding algorithm.

Pre: `request` is the request, $\mathbb{E}$ is the set of execution nodes.
Post: All the tasks in `request` are allocated to nodes in $\mathbb{E}$.

```
 1: procedure FORWARDCENTRALIZED(request)
 2:   availableNodes ← ∅
 3:   for all E_u ∈ 𝔼 do
 4:     if E_u fulfills request.ℙℝ then
 5:       availableNodes ← availableNodes ⋃ E_u
 6:     end if
 7:   end for
 8:   SORT(availableNodes)                          ▷ Best nodes are allocated first.
 9:   while ¬ISEMPTY(availableNodes) ⋀ request.n > 0 do
10:     E_u ← POPFRONT(availableNodes)
11:     task ← EXTRACTTASK(request)
12:     SEND(task, E_u)
13:   end while
14: end procedure
```

Algorithm C.2 Centralized version of the MMP policy forwarding algorithm.

Pre: `request` is the request, $\mathbb{E}$ is the set of execution nodes.
Post: All the tasks in `request` are allocated to nodes in $\mathbb{E}$.

```
 1: procedure FORWARDCENTRALIZED(request)
 2:   candidates ← ∅
 3:   for all E_u ∈ 𝔼 do
 4:     if E_u fulfills request.ℙℝ then
 5:       candidates ← candidates ⋃ {(E_u, E_u.Q + request.a_i/s_u)}
 6:     end if
 7:   end for
 8:   SORT(candidates)                              ▷ Nodes are sorted by their queue end time.
 9:   while request.n > 0 do
10:     (E_u, t_e) ← POPFRONT(candidates)
11:     task ← EXTRACTTASK(request)
12:     SEND(task, E_u)
13:     PUSHBACK((E_u, t_e + s_u request.a_i))
14:     SORT(candidates)
15:   end while
16: end procedure
```

Algorithm C.3 Centralized version of the DP policy forwarding algorithm.

```
Pre:  request is the request, 𝔼 is the set of execution nodes.
Post: All the tasks in request are allocated to nodes in 𝔼.
 1: procedure FORWARDCENTRALIZED(request)
 2:   holes ← ∅
 3:   for all E_u ∈ 𝔼 do
 4:     if E_u fulfills request.ℙℝ then
 5:       h ← l_u(request.δ_i)
 6:       if h ≥ request.a_i then
 7:         holes ← holes ⋃ {(E_u, h, h mod request.a_i)}
 8:       end if
 9:     end if
10:   end for
11:   SORT(holes)                    ▷ Nodes are sorted by the remaining amount of FLOPs.
12:   while request.n > 0 do
13:     (E_u, t_h) ← POPFRONT(holes)
14:     numTasks ← ⌊t_h/request.a_i/s_u⌋        ▷ Allocate as many tasks as possible
15:     tasks ← EXTRACTTASKS(request, numTasks)
16:     for all τ ∈ tasks do
17:       SEND(τ, E_u)
18:     end for
19:   end while
20: end procedure
```

It can be seen in Algorithm 5.1 that its cost is $O(m)$, where m is the number of tasks in E_u queue. So, the cost of the first loop is $O(T)$, where T is the total number of tasks currently allocated in the system. In practice, this means that the first loop takes much longer to execute than in the two previous policies. Again, sorting the list `holes` has a time complexity of $O(n \log n)$, and the second loop has a complexity of $O(n)$. Finally, in this case, the space complexity is $O(T)$.

C.4. FSP Policy

Finally, the centralized version of the FSP policy appears in Algorithm C.4. Like the decentralized version, it calculates how many tasks to send to each node to minimize the maximum slowness. However, it works with full knowledge about all the execution nodes. It creates a list `candidates` with all the nodes that meet the memory and disk requirements, and calculates the slowness obtained by assigning one task to each of them. In this case, the function GETSLOWNESS(E_u, a, n) uses the algorithms of Section 7.2.1 to calculate the exact value of the maximum slowness reached when adding n tasks of length a to the queue of E_u. Then, it purges the list to keep the `request.n` best nodes, and iterates on the number of tasks to see if some nodes provide a better slowness with more tasks than others. When no better result is obtained, it sends its assigned number of tasks to each node in the `candidates` list.

Algorithm C.4 Centralized version of the FSP policy forwarding algorithm.

```
Pre:  request is the request, 𝔼 is the set of execution nodes.
Post: All the tasks in request are allocated to nodes in 𝔼.
 1: procedure FORWARDCENTRALIZED(request)
 2:   candidates ← ∅
 3:   for all E_u ∈ 𝔼 do
 4:     if E_u fulfills request.ℙℝ then
 5:       l ← GETSLOWNESS(E_u, request.a, 1)
 6:       candidates ← candidates ⋃ {(E_u, l, 1)}
 7:     end if
 8:   end for
 9:   PURGE(candidates, request.n)
10:   oneMoreTask ← true
11:   i ← 1
12:   while oneMoreTask do
13:     i ← i + 1
14:     oneMoreTask ← false                    ▷ If nothing happens, this is the last iteration.
15:     for all (E_u, l_old, n) ∈ candidates | n = i − 1 do
16:       l_new ← GETSLOWNESS(E_u, request.a, i)
17:       if l_new < candidates.last.slowness then
18:         candidates ← candidates \ {(E_u, l_old, n)}
19:         candidates ← candidates ⋃ {(E_u, l_new, i)}
20:         oneMoreTask ← true
21:       end if
22:     end for
23:     PURGE(candidates, request.n)
24:   end while
25:   return candidates.last.slowness
26: end procedure
```

Both loops that iterate on the candidate nodes have a time complexity of $O(n)$. The while loop can be repeated up to `request.`n times, in the case that all the tasks end up in the same node. However, this is extremely rare, and it is repeated no more than four times in most cases. Besides, the `candidates` list must be kept sorted with a heap structure, to be able to purge and find the worst candidate in $O(1)$. So, like in previous policies, the whole algorithm can be performed with complexity $O(n \log n)$. The space complexity is still $O(n)$.

Glossary

clustering algorithm An algorithm that groups a set of data objects in such a way that objects in the same group (called cluster) are more similar, in some sense or another, to each other than to those in other groups. 8, 9, 19, 20, 23, 44, 59, 62, 79

deadline The time by which all the tasks of an application must be finished. 3–5, 13, 33–37, 39, 41–45, 49–52, 62, 69, 72–74, 77, 80, 89, 93

directed acyclic graph (DAG) A graph with directed edges and no cycle. That is, it is not possible to start at some vertex v and follow a sequence of edges that eventually loops back to v again. 41–43, 98

discrete event simulator (DES) A simulator that models the operation of a system as a discrete sequence of events in time. Each event occurs at a particular instant in time and marks a change of state in the system. 59, 62, 91, 92

Earliest Deadline First (EDF) A local scheduling policy that sorts tasks so that those with the earliest deadline come first. 13, 34, 35, 43, 49

First Come First Served (FCFS) A local scheduling policy that sorts tasks in the same order that they arrived at the execution node. 13, 28, 41

floating point operation (FLOP) An atomic operation of the microprocessor involving floating point arithmetic. 13, 14, 28, 29, 34–37, 50, 60–62, 80, 89, 93, 95

forwarding algorithm In networking, an algorithm that decides in which direction a router should send a packet to reach its destination. In the context of this thesis, it decides in which direction to send one or more tasks. 4, 9, 11, 13, 16–18, 24–26, 28, 30, 31, 39, 45, 57, 69, 71, 79, 80, 90, 94–96

makespan In the scheduling of tasks, the makespan is the total duration of the schedule, from when the first task starts until the last one ends. 4, 5, 27, 28, 30, 31, 33, 42, 67–69, 80, 89

mean square error (MSE) A measure that quantifies the difference between values implied by an estimator and the true values of the quantity being estimated, corresponding to the expected value of the squared error loss. That is, it measures the average of the squares of these differences. 21, 23, 37, 39, 56, 62

overlay network A computer network that is built on top of another one. Nodes are connected through logical links that correspond to paths of one or more links in the underlying network. 4, 9, 11, 12, 14–16, 33, 71, 79

slowness Term used in this thesis to denote the ratio of the amount of time an application spends in the system to its task length. Under certain conditions, it is proportional to the stretch. 48–58, 60–62, 70, 71, 80, 89, 95

startline Term used in this thesis to denote the time after which the tasks of an application are allowed to start. 42, 45, *see* deadline

stretch Ratio of the amount of time an application spends in the system to the amount of time it would spend if it was the only scheduled application. 4, 47, 48, 70, 98

workflow A set of tasks and a set of dependencies between some of them. When a task depends on another one, the later must finish before the former may start its execution. It is usually represented with a DAG, where nodes are tasks and edge are dependencies. 1, 3–5, 13, 41, 42, 73–76, 80