Cooperative Resource Management for Parallel and Distributed Systems

Cristian Klein-Halmaghi

N° d'ordre : 773
N° attribué par la bibliothèque : 2012ENSL0773

École Normale Supérieure de Lyon
Laboratoire de l'Informatique du Parallélisme

THÈSE

en vue d'obtenention du grade de

Docteur de l'École Normale Supérieure – Université de Lyon
Discipline : Informatique

au titre de l'École Doctorale Informatique et Mathématiques

présentée et soutenue publiquement le 29 novembre 2012 par

Cristian Klein-Halmaghi

Cooperative Resource Management for Parallel and Distributed Systems

Composition du jury

Directeur de thèse :	Christian Pérez *Directeur de Recherche à LIP/INRIA Rhône-Alpes*
Rapporteurs :	Emmanuel Jeannot *Directeur de Recherche à LaBRI/INRIA Sud-Ouest*
	Ramin Yahyapour *Professeur à Université de Göttingen, Allemagne*
Examinateurs :	Michel Daydé *Professeur à INP/ENSEEIHT, Toulouse*
	Xavier Vigouroux *Résponsable Éducation et Recherche HPC à Bull*

Remerciements

Voici la page des remerciements, la dernière écrite, mais la première lue. Avant de commencer avec les remerciements dits classiques, j'aimerais remercier la France, ce joli pays avec son fromage fort et son bon vin, ses peuples accueillants qui ont su faire de cet endroit un chez moi pendant presque cinq ans. C'est pour leur montrer ma gratitude que cette page est écrite en français.

Tout d'abord, je remercie le jury : Emmanuel Jeannot et Ramin Yahyapour pour avoir accepté d'être rapporteurs, Michel Daydé pour avoir présidé le jury et Xavier Vigouroux pour avoir complété le jury avec son expertise industrielle. J'ai beaucoup apprécié vos questions, qui m'ont permis de montrer l'intérêt que je porte à ce sujet.

Grand merci à Christian pour beaucoup de choses, que mes mots ne suffisent pas à exprimer. Si tu m'excusais pour la simplification, je te remercierais pour ta disponibilité, ta compétence et pour m'avoir poussé toujours plus loin dans ma démarche scientifique.

Parmi d'autres personnes avec qui j'ai collaboré, j'aimerais remercier les partenaires du projet ANR COOP, surtout Aurélie, Fréd, Ronan et Yann. Sans vous ma thèse aurait été amputée d'une bonne partie de sa matière. Merci beaucoup aux gens de l'équipe et du labo, qui, à travers des discussions animées, m'ont donné des conseils importants pour ma recherche. Merci surtout aux "vieux", Ben, Ghislain et Julien, qui m'ont donné de bonnes références au début, ainsi qu'au "jeunes", Noua et Vincent, pour le travail qu'ils ont réalisé sous ma co-tutelle.

Un grand merci à toute la communauté Grid'5000 pour avoir développé cet outil sans lequel ma thèse ne serait restée qu'un exercice théorique. Merci à l'équipe SimGrid, non seulement pour l'outil qu'ils ont développé, mais aussi pour les discussions autour de simulations, qui m'ont permis d'être plus rigoureux dans mes expériences.

Parmi les personnes que j'ai rencontrées avant ma thèse, j'aimerais remercier Florent de Dinechin et Octavian Creț, sans lequels mes études aurait fini plus tôt. Merci à Emil Cebuc et Alin Suciu pour m'avoir donné le goût pour la recherche.

Pour avoir été de mon côté quand les ordinateurs ne l'étaient pas, j'aimerais remercier à mes amis de près et de loin. Grand merci à la « mafia roumaine », parmi d'autres, pour m'avoir aidé ne pas oublier ma langue maternelle, et surtout à mon colocataire pour les soirs de débat et de cuisine, bref, pour m'avoir supporté sous le même toit. Merci à mes amis internationaux pour m'avoir montré que, quelque soit le temps dans les montagnes, on peut toujours faire de la rando. Pour m'avoir intégré dans la société française tout en oxygénant mon cerveau, merci aux gens de Génération Roller et du Club Rock.

Merci à ma famille (y compris la partie allemande avec qui nous avons récemment fusionné) pour leur affection et leur soutien inconditionnel, pour les vacances de Noël et de Pâques. Je profite de cet espace pour m'excuser auprès ma sœur de l'avoir oubliée lors les remerciements oraux, mais bon, tu comprends, mon cerveau était grillé après la soutenance.

Il y a beaucoup d'autres personnes que j'aurais aimé mentionner ici. Je m'arrête avant que mes remerciements ne se transforment en autobiographie.

This manuscript was typeset using XeLaTeX and the memoir class. Figures were drawn with Dia and Gnuplot. During the late nights, when my brain started producing bad English, `netspeak.org` was there to help me. For those who contributed to these tools, thank you!

Throughout the manuscript, generic "she" (instead of universal "he" or singular "they") is being used to designate a person. The choice has been made to reflect current writing trends in scientific journals. The reader should not assume anything about the gender, race, color, age, national origin, religion or disability, citizenship status, Vietnam Era or special disabled veteran status, sexual orientation, gender identity, or gender expression of the person referred to.

Abstract

High-Performance Computing (HPC) resources, such as Supercomputers, Clusters, Grids and HPC Clouds, are managed by Resource Management Systems (RMSs) that multiplex resources among multiple users and decide how computing nodes are allocated to user applications. As more and more petascale computing resources are built and exascale is to be achieved by 2020, optimizing resource allocation to applications is critical to ensure their efficient execution. However, current RMSs, such as batch schedulers, only offer a limited interface. In most cases, the application has to blindly choose resources at submittal without being able to adapt its choice to the state of the target resources, neither before it started nor during execution.

The goal of this Thesis is to improve resource management, so as to allow applications to efficiently allocate resources. We achieve this by proposing software architectures that promote collaboration between the applications and the RMS, thus, allowing applications to negotiate the resources they run on. To this end, we start by analysing the various types of applications and their unique resource requirements, categorizing them into rigid, moldable, malleable and evolving. For each case, we highlight the opportunities they open up for improving resource management.

The first contribution deals with moldable applications, for which resources are only negotiated before they start. We propose CooRMv1, a centralized RMS architecture, which delegates resource selection to the application launchers. Simulations show that the solution is both scalable and fair. The results are validated through a prototype implementation deployed on Grid'5000.

Second, we focus on negotiating allocations on geographically-distributed resources, managed by multiple institutions. We build upon CooRMv1 and propose *dist*CooRM, a distributed RMS architecture, which allows moldable applications to efficiently co-allocate resources managed by multiple independent agents. Simulation results show that *dist*CooRM is well-behaved and scales well for a reasonable number of applications.

Next, attention is shifted to run-time negotiation of resources, so as to improve support for malleable and evolving applications. We propose CooRMv2, a centralized RMS architecture, that enables efficient scheduling of evolving applications, especially non-predictable ones. It allows applications to inform the RMS about their maximum expected resource usage, through pre-allocations. Resources which are pre-allocated but unused can be filled by malleable applications. Simulation results show that considerable gains can be achieved.

Last, production-ready software are used as a starting point, to illustrate the interest as well as the difficulty of improving cooperation between existing systems. GridTLSE is used as an application and DIET as an RMS to study a previously unsupported use-case. We identify the underlying problem of scheduling optional computations and propose an architecture to solve it. Real-life experiments done on the Grid'5000 platform show that several metrics are improved, such as user satisfaction, fairness and the number of completed requests. Moreover, it is shown that the solution is scalable.

Résumé

Les ressources de calcul à haute performance (High-Performance Computing—HPC), telles que les supercalculateurs, les grappes, les grilles de calcul ou les Clouds HPC, sont gérées par des gestionnaires de ressources (Resource Management System—RMS) qui multiplexent les ressources entre plusieurs utilisateurs et décident comment allouer les nœuds de calcul aux applications des utilisateurs. Avec la multiplication de machines péta-flopiques et l'arrivée des machines exa-flopiques attendue en 2020, l'optimisation de l'allocation des ressources aux applications est essentielle pour assurer que leur exécution soit efficace. Cependant, les RMSs existants, tels que les batch schedulers, n'offrent qu'une interface restreinte. Dans la plupart des cas, l'application doit choisir les ressources « aveuglément » lors de la soumission sans pouvoir adapter son choix à l'état des ressources ciblées, ni avant, ni pendant l'exécution.

Le but de cette Thèse est d'améliorer la gestion des ressources, afin de permettre aux applications d'allouer des ressources efficacement. Pour y parvenir, nous proposons des architectures logicielles qui favorisent la collaboration entre les applications et le gestionnaire de ressources, permettant ainsi aux applications de négocier les ressources qu'elles veulent utiliser. À cette fin, nous analysons d'abord les types d'applications et leurs besoins en ressources, et nous les divisons en plusieurs catégories : rigide, modelable, malléable et évolutive. Pour chaque cas, nous soulignons les opportunités d'amélioration de la gestion de ressources.

Une première contribution traite les applications modelables, qui négocient les ressources seulement avant leur démarrage. Nous proposons CooRMv1, une architecture RMS centralisée, qui délègue la sélection des ressources aux lanceurs d'application. Des simulations montrent qu'un tel système se comporte bien en termes d'extensibilité et d'équité. Les résultats ont été validés avec un prototype déployé sur la plate-forme Grid'5000.

Une deuxième contribution se focalise sur la négociation des allocations pour des ressources géographiquement distribuées qui appartiennent à plusieurs institutions. Nous étendons CooRMv1 pour proposer *dist*CooRM, une architecture RMS distribuée, qui permet aux applications modelables de co-allouer efficacement des ressources gérées par plusieurs agents indépendants. Les résultats de simulation montrent que *dist*CooRM se comporte bien et passe à l'échelle pour un nombre raisonnable d'applications.

Ensuite, nous nous concentrons sur la négociation des ressources à l'exécution pour mieux gérer les applications malléables et évolutives. Nous proposons CooRMv2, une architecture RMS centralisée, qui permet l'ordonnancement efficace des applications évolutives, et surtout celles dont l'évolution n'est pas prévisible. Une application peut faire des « pré-allocations » pour exprimer ses pics de besoins en ressources. Cela lui permet de demander dynamiquement des ressources, dont l'allocation est garantie tant que la pré-allocation n'est pas dépassée. Les ressources pré-allouées mais inutilisées sont à la disposition des autres applications. Des gains importants sont ainsi obtenus, comme les simulations que nous avons effectuées le montrent.

Enfin, nous partons de logiciels utilisés en production pour illustrer l'intérêt, mais aussi la difficulté, d'améliorer la collaboration entre deux systèmes existants. Avec GridTLSE comme application et DIET comme RMS, nous avons trouvé un cas d'utilisation mal supporté auparavant. Nous identifions le problème sous-jacent d'ordonnancement des calculs optionnels et nous proposons une architecture pour le résoudre. Des expériences réelles sur la plate-forme Grid'5000 montrent que plusieurs métriques peuvent être améliorées, comme par exemple la satisfaction des utilisateurs, l'équité et le nombre de requêtes traitées. En outre, nous montrons que cette solution présente une bonne extensibilité.

Summary

Table of Contents

List of Algorithms

List of Figures

List of Tables

Part I

Introduction and Context

CHAPTER 1

Introduction

A PhD thesis is not like a thriller. You can give away the ending in the introduction.

C. P.

This chapter sets the mood of the Thesis. It starts by presenting its motivation, which leads to the goal of the Thesis. Finally, the contributions of the Thesis are highlighted.

1.1 Motivation

Today's scientists are increasingly relying on numerical simulations both for fundamental research (often called eScience) as well as for research and development. Indeed, more and more domains are using simulations, such as cosmology, molecular biology, aircraft design, nuclear power plant design validation, car crash simulations, just to name a few. Hence, continuing scientific progress requires fulfilling increasing computing needs.

Answering this demand proves to be challenging in recent years. Since 2005, the processor clock rate in MHz has stopped progressing significantly and previously used methods to speed up execution, such as out-of-order execution, instruction-level parallelism and pipelining, seem to have reached their limit. As a solution, hardware designers recur to parallelism, which can nowadays be found in every device starting from supercomputers to laptops and even smartphones. Even before becoming a necessity, parallelism was routinely used in eScience to enable the resolution of problems which are several orders-of-magnitude larger than what could be solved without parallelism.

However, more parallelism is achieved with increasingly complex hardware. For example, petascale resources present parallelism at several layers. At the top-most layer, users may have access to several parallel machines, each being build of a large number of computing nodes. To interconnect these nodes, one needs to recur to a non-homogeneous network, featuring a torus or a fat-tree topology, since producing a high-radix switch is impractical. At the node level, parallelism is achieved using multi-core CPUs and GPUs featuring non-uniform memory access. Maximizing application performance on these resources is challenging and may become even more difficult on exascale resources expected in 2020.

Due to the large investment in these computing resources, exploiting them at peak capacity is of uttermost importance, which is the core interest of High-Performance Computing (HPC). However, achieving good performance cannot be obtained by the hardware resources alone, but has to be done by orchestrating the whole platform.

One of the most important components of the platform is the Resource Management System (RMS). Indeed, HPC resources, as can be found in Super, Grid and Cloud Computing, are rarely used by a single user. Therefore, it is the responsibility of the RMS to decide how to share the hardware resources among users. Properly doing this is especially important since many applications are highly performance sensitive: Some might run faster on CPUs, others may be able to take advantage of GPUs. Also, most HPC applications are sensitive to the network latency and bandwidth they observe.

Traditionally, resource management has been done as follows: The user (or user code acting on behalf of the user) formulates a request, specifying the resource requirements of the application. Then, the RMS runs a scheduling algorithm, which, taking into account the resource requests of all applications, decides how to multiplex the resources in both space and time. Unfortunately, such an approach forces the application to "blindly" choose resources at submittal, without being able to adapt to the state of the target resources, neither before start, nor during execution. For example, an application cannot express the fact that it may run on 4 nodes, but, if enough resources are available, it would prefer running on 8 nodes, to speed up its execution. Likewise, if resources become free, a currently executing application cannot take advantage of these resources, so as to finish earlier.

Properly solving the above issue would be of benefit both to the user and to the administrator of the platform. On one hand, the user would receive his results faster, on the other hand, the administrator could improve resource usage, thus having a higher scientific return for the investment in the resources.

1.2 Goal of this Thesis

The purpose of this Thesis is to improve resource management, so as to allow applications to make a more efficient use of resources. We start by categorizing applications and showing that application-side workarounds are necessary to circumvent RMS limitations. Therefore, this Thesis proposes improving the cooperation between applications and the RMS. The issue is then to find interfaces and protocols that would allow applications to efficiently express their resource requirements, while allowing the RMS to enforce its own resource management policy.

As opposed to many works, the Thesis does not focus on scheduling algorithms. Nevertheless, in order to prove the usefulness of the proposed concepts and architectures, scheduling algorithms have to be devised on both the application- and the RMS-side. However, instead of reinventing the wheel, we shall attempt to reuse existing algorithms and adapt them as necessary.

Another particularity of this Thesis is that the RMS is assumed to have direct control over the resources, i.e., they are not limited by any lower-level abstractions. In literature one would say that the resources are *dedicated*, i.e., no resource allocation may take place that the RMS does not know about. Our hypothesis is somewhat stronger. We have taken this decision for a very simple reason: Instead of proposing yet another workaround, we have decided to contribute with minimalistic interfaces that would make workarounds unnecessary. Achieving this is best done if the RMS can take all the liberty and has full control over the resources.

Also, allocations are done at node granularity, i.e., at a certain time, each computing node is allocated to at most one application. This decision has been taken because:

- Tightly-coupled HPC applications perform better if they are run exclusively on a computing node. Indeed, even "noise" generated by the operating system can significantly affect their performance [111].
- It simplifies access control and accounting. In fact, this already reflects best-practices at several computing centers [135].
- It somewhat simplifies the devised solutions. In some part of the proposed architectures, we can work with node-counts (i.e., integers), instead of having to enumerate node identifiers (i.e., list of identifiers).

1.3 Contributions of the Thesis

The contribution of the Thesis can be divided among three axis: the types of applications, the scale of resources and the concepts the solution works with. As such, we start by dividing applications by resource requirements and classifying them into moldable and dynamic (malleable and evolving). Moldable applications do not change their resource allocation during execution, therefore, negotiation with the RMS only takes place before they start. Dealing with them first is a prerequisite before dealing with the more challenging case of dynamic applications, which may change their resource allocation at run-time. Depending on who initiates this change, dynamic applications can further be divided into malleable, i.e., the change is initiated by the RMS, and/or evolving, i.e., the change is initiated by the application itself.

Judging by their geographic scale, resources can be either centralized or distributed. Centralized resources are those in which centralized control can be enforced, such as Supercomputers, Clusters and Clouds. In contrast, geographically-distributed resources are owned by multiple institutions, hence, enforcing centralized control is difficult. Examples of such resources include Grids and Multi-Clouds (Sky Computing).

Related to the provided solution, the RMS can either work with low- or high-level concepts.

Type of Application	Type of Resource		Type of Concept
	centralized	distributed	
moldable	CooRMv1	*dist*CooRM	low-level
dynamic	CooRMv2	–	
	–	GridTLSE & DIET	high-level

Table 1.1: Overview of the contributions of the Thesis

Through low-level we understand concepts such as number of nodes and node identifiers, while high-level concepts are tasks and services that are able to execute them.

Table 1.1 gives an overview of our contributions. We start by dealing with moldable applications in the centralized case. As a solution, we propose CooRMv1, a centralized RMS architecture which delegates resource selection to application launchers, thus allowing them to employ their custom resource selection algorithm. We show through simulations that the proposed architecture ensures fairness and scales well. Furthermore, we validate results obtained using simulations by doing real experiments on the Grid'5000 platform. This contribution has been published at an international conference [C3], with a more detailed version being available as a research report [R2].

Our next two contributions extend the CooRMv1 architecture in two directions: distributed resources and dynamic applications. For distributed resources, we propose the *dist*CooRM architecture, which allows moldable application to efficiently co-allocate resources managed by multiple agents. This allows each institution to keep their independence and also improves the fault-tolerance of the system. Simulation results show that *dist*CooRM behaves well and is scalable for a reasonable number of applications.

Next, we extend CooRMv1 to support dynamic applications, i.e., malleable and evolving ones. We first devise a scheduling algorithm for evolving applications and study the gains that can be made by properly supporting them. Next, we use this algorithm and propose the CooRMv2 architecture, which efficiently supports malleable and evolving applications, especially non-predictable ones. An evolving application can make pre-allocations to signal the RMS their maximum expected resource usage, then it can reliably increase/decrease its allocation, as long as the pre-allocation is not exceeded. Resources that are pre-allocated by an evolving application, but not effectively used, can be filled by a malleable application. Simulation results show that resource utilisation can be considerably improved. These contributions have been published as an international conference paper [C3], a workshop paper [C1], with extended versions to be found as research reports [R1, R3].

The previous contributions propose solutions which work with low-level concepts. Our last contribution starts from existing systems which propose high-level resource management abstractions, such as tasks to execute and service which can execute them, to study the benefit as well as the difficulty to make these two software cooperate. To this end, we took GridTLSE as an application and DIET as an RMS and studied a use-case which was previously badly supported. We identify the underlying issue of scheduling optional computations, i.e., computations which are not critical to the user, but whose completion would improve her results. As a solution, we propose DIET-ethic, a generic master-client architecture, which fairly schedules optional computations. For our evaluation we have implemented the architecture within DIET. Real-life experiments show that several metrics are improved, such as user satisfaction, fairness and the number of completed requests. Moreover, the solution is shown to be scalable. These contributions have been published as a research report [R4] and have been submitted to an international

conference [C4].

1.4 Structure of this Document

This manuscript is organized in four parts. Part I, which you are about to read, continues with the context of the Thesis in Chapter 2. It gives the reader the necessary elements to understand the motivation of this Thesis.

Part II presents the first contributions of this Thesis, which deal with moldable applications. Chapter 3 presents CooRMv1, which deals with centralized resources, while Chapter 4 presents *dist*CooRM, an extension to CooRMv1, which deals with geographically-distributed resources.

Part III contains contributions related to malleable and evolving applications. Chapter 5 presents a scheduling algorithm for evolving applications, highlighting the gains that can be made by properly supporting this type of application. Chapter 6 uses this scheduling algorithm and proposes the CooRMv2 architecture, which allows the efficient scheduling of evolving and malleable application. Finally, Chapter 7 starts from production-ready software, GridTLSE and DIET, and studies how optional computations could efficiently be supported using them.

Part IV concludes the Thesis. Chapter ∞ presents concluding remarks and opens up perspectives.

1.5 Publications

The contributions of this Thesis have been published in several international and national conferences with reviewing committees. More detailed versions of these articles have been published as INRIA research reports. Some ideas, especially those related to context and motivation, have been published in the deliverables of the National Research Agency (Agence Nationale de la Recherche – ANR) COSINUS COOP project (ANR-09-COSI-001) [126], which has financed this Thesis.

Articles in International Conferences

[C1] Cristian Klein and Christian Pérez. Towards Scheduling Evolving Applications. In *Euro-Par 2011: Parallel Processing Workshops - CCPI, CGWS, HeteroPar, HiBB, HPCVirt, HPPC, HPSS, MDGS, ProPer, Resilience, UCHPC, VHPC, Bordeaux, France, August 29 - September 2, 2011, Revised Selected Papers, Part I*, volume 7155 of *Lecture Notes in Computer Science*. Springer, August 2011. doi:10.1007/978-3-642-29737-3_15.

[C2] Cristian Klein and Christian Pérez. An RMS Architecture for Efficiently Supporting Complex-Moldable Applications. In *13th IEEE International Conference on High Performance Computing and Communication, HPCC 2011, Banff, Alberta, Canada, September 2-4, 2011*. IEEE, August 2011. doi:10.1007/978-3-642-29737-3_15.

[C3] Cristian Klein and Christian Pérez. An RMS for Non-predictably Evolving Applications. In *2011 IEEE International Conference on Cluster Computing (CLUSTER), Austin, TX, USA, September 26-30, 2011*. IEEE, September 2011. doi:10.1109/CLUSTER.2011.56.

[C4] Frédéric Camillo, Eddy Caron, Ronan Guivarch, Aurélie Hurault, Cristian Klein, and Christian Pérez. Diet-ethic: Fair Scheduling of Optional Computations in GridRPC Mid-

dleware. In *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2013, Delft, Netherlands, May 13–16, 2013*. Submitted.

Articles in National Conferences

[N1] Eddy Caron, Cristian Klein, and Christian Pérez. Efficient Grid Resource Selection for a CEM Application. In *9ème Rencontres francophones du Parallélisme, Renpar'19, Toulouse, France, September 9-11, 2009*, September 2009.

Research Reports

[R1] Cristian Klein and Christian Pérez. Scheduling Rigid, Evolving Applications on Homogeneous Resources. Research Report RR-7205, INRIA, February 2010.

[R2] Cristian Klein and Christian Pérez. Untying RMS from Application Scheduling. Research Report RR-7389, INRIA, September 2010.

[R3] Cristian Klein and Christian Pérez. An RMS for Non-predictably Evolving Applications. Research Report RR-7644, INRIA, June 2011.

[R4] Frédéric Camillo, Eddy Caron, Ronan Guivarch, Aurélie Hurault, Cristian Klein, and Christian Pérez. Diet-ethic: Fair Scheduling of Optional Computations in GridRPC Middleware. Research Report RR-7959, INRIA, May 2012.

Scientific Project Deliverables

[D1] Alexandre Denis, Cristian Klein, Christian Pérez, and Yann Radenac. Resource Management Systems for Distributed High-Performance Computing. Deliverable D2.2, Agence Nationale de la Recherche – ANR COSINUS COOP (ANR-09-COSI-001), February 2011. Available from: http://coop.gforge.inria.fr/lib/exe/fetch.php?media=d2.2.pdf [cited September 26, 2012].

[D2] Frédéric Camillo, Alexandre Denis, Ronan Guivarch, Aurélie Hurault, Cristian Klein, Christian Pérez, Yann Radenac, and André Ribes. Analysis on Elements of Cooperation between Programming Model Frameworks and Resource Management Systems. Deliverable D2.3, Agence Nationale de la Recherche – ANR COSINUS COOP (ANR-09-COSI-001), July 2011. Available from: http://coop.gforge.inria.fr/lib/exe/fetch.php?media=d2.3.pdf [cited September 26, 2012].

CHAPTER 2

Context

The pure and simple truth is
rarely pure and never simple.

Oscar Wilde

This chapter lays out the context of the Thesis by describing the High-Performance Computing (HPC) aspects that are important for understanding the resource management problem. First, the driving force of all complexity is being presented: the HPC resources. Second, the applications that are executed on such resources are classified based on their resource requirements. Finally, the state of the art of resource management is analyzed and improvement opportunities are highlighted.

2.1 High-Performance Computing Resources

This section describes HPC computing resources that are most commonly used today. For each of them their intrinsic properties are analyzed.

2.1.1 Supercomputing

supercomputer

There is no universally accepted definition for a supercomputer [62], however, a **supercomputer** can best be defined as *a computer at the frontline of current processing capacity*. Achieving such performance is mostly done using proprietary (instead of commodity) hardware. Nevertheless, what hardware is to be considered proprietary is not clear [62]. Often, proprietary hardware becomes commoditized, which makes drawing a clear line even more difficult.

To track the fastest public supercomputers in the world, the Top 500 List [146] has been established. It is updated twice a year. Supercomputers are ranked using the High-Performance Linpack benchmark [138], which has a high computation-to-communication ratio, thus exposing the maximum Floating-Point Operations per Second (FLOPS) of the hardware. This has been considered unfair, as it does not highlight the interconnect's performance. Alternative benchmarks have been proposed, such as the HPC Challenge [78] or the Graph 500 List [132], but they are not as widely recognized. In recent years, increasing importance has been given not only to creating powerful supercomputers, but also energy-efficient ones. This is why, the Green 500 List [133] has been established, which tracks the most energy-efficient supercomputers from the Top 500 List.

Let us give some examples of supercomputers. At the time of writing, the IBM Sequoia ranked #1 on the Top 500 List with over 16 PFLOPS. It achieved this performance using 98,304 homogeneous computing nodes, each composed of a 16-core PowerPC A2 processor and 16 GB of memory. These nodes are interconnected using a 5D torus network.

A recent trend is to increase supercomputing performance using accelerators. For example, the Chinese Tianhe-1A held the crown of the Top 500 List from October 2010 to June 2011 with over 2.5 PFLOPS. A particularity of this supercomputer is the usage of GPUs: Besides 2 Intel Xeon 6-core CPUs, each of the 7,168 computing nodes contains 1 Nvidia™ M2050 GPU. Otherwise, the nodes themselves are homogeneous. Interconnection among them is ensured using a proprietary Infiniband-like network organized in a fat-tree topology.

The Blue Waters project aims at creating a supercomputer which delivers 1 PFLOPS for a wide range of applications. According to current plans [11], the supercomputer will be composed of two types of nodes: CPU nodes, featuring the 8-core AMD Operaton™ 6200, and GPU nodes, featuring the NVIDIA® Tesla™. As one can observe, nodes will no longer be homogeneous. These nodes will be interconnected using a 3D torus network.

From the examples above, one can deduce several tendencies in supercomputing. First, the interconnect is not homogeneous, i.e., the observed bandwidth and latency depend on the source and destination node. This is important to take into account when allocating resources to latency sensitive applications [81]. Second, the battle between using few powerful and complex cores (i.e., CPUs) or many specialized cores (i.e., GPUs) has not drawn a clear winner yet. As highlighted by the Blue Waters project, future supercomputing will most likely feature heterogeneous computing nodes, with some of them containing complex cores, accelerating the sequential part of the applications, and others will contain simple cores, accelerating the parallel part of the applications [143]. Both of these tendencies need to be taken into account when designing tomorrow's Resource Management System (RMS).

2.1.2 Cluster Computing

A **computing cluster** is a set of (more or less) loosely connected computers that work together acting as a single system. In a way, clusters can be viewed as the "cheap" version of supercomputers: Instead of using expensive, proprietary technology, clusters use commodity hardware. computing cluster

Due to their good price to performance ratio, clusters can be found in many research institutes and companies. They most commonly feature homogeneous, multi-core nodes connected using a high-bandwidth, low-latency Local-Area Network (LAN), ranging from Gigabit Ethernet to Infiniband. High-end clusters employ a non-blocking switch, thus, for many applications, clusters can be regarded as being perfectly homogeneous both communication- and computation-wise.

However, computing centers generally have multiple clusters. This occurs as a result of upgrading the infrastructure: When a new cluster is bought, instead of decommissioning the old one, the two clusters co-exist. Therefore, the actual hardware that a typical end-user has access to can range from "nearly" homogeneous to heterogeneous. For example, at the time of writing the Computation Center at IN2P3, Lyon, France operated 4 clusters having similar computing power [124]. In contrast, the Texas Advanced Computing Center operates two clusters with quite different capabilities: One has GPUs, while the other one is CPU-only [144].

This makes resource management somewhat more difficult. Specifying a node-count is not enough to properly select resources: One also has to specify on what cluster the allocation should take place.

2.1.3 Grid Computing

In the previous two sections, we have presented resources that are under the administration of a single institution. Pushing this concept further, we can imagine multiple institutions mutualizing their hardware resources, in order to reach a common goal, such as a multi-partner research project.

This is the vision of Grid computing: users having transparent access to computing resources, no matter what their locations, similarly to how the power grid works nowadays. Grids can best be defined using Ian Foster's three-point checklist [49]. Thus, a **Computing Grid** is a system that: computing grid

(1) coordinates resources that are not subject to centralized control

(2) using standard, open, general-purpose protocols and interfaces

(3) to deliver nontrivial qualities of service.

Grid are nowadays widely deployed for doing e-Science. For example, in the United States, the TeraGrid project (now continued with the XSEDE project) aggregated the resources of 10 institutions, serving $4,000$ users at over 200 universities. In Europe, the EGI project provides $13,319$ users with over $300,000$ cores [128]. From a hardware perspective, these Grids are essentially a geographically-dispersed multi-cluster system, in which the clusters are connected through a high-bandwidth Wide-Area Network (WAN).

From a user perspective, **Virtual Organizations** [50] are abstract institutions grouping users and resources belonging to the same research community. As of today, Grids are mostly used for running applications composed of loosely-coupled, sequential tasks [65]. However, they do have the potential to run large, tightly-coupled simulations, which require the simultaneous availability of resources on multiple sites [72, 83]. virtual organizations

Let us highlight the characteristics of Grid resources. First, the computing resources and the interconnecting network are heterogeneous. This means that good resource selection algorithms

have to be employed in order to optimize application performance [30, 44]. Second, there is no centralized control as each institution wants to keep its independence and be able to enforce its own resource allocation policies. Third, Grid platforms are large-scale, distributed and dynamic. Indeed, the availability of the resources changes at a high frequency, however, the installed resources on the Grid platform only change at a low frequency [48].

These characteristics pose additional resource management challenges. A Grid RMS needs to be scalable, distributed and efficiently deal with both static and dynamic resource information.

2.1.4 Desktop Computing

desktop computing

volunteer computing

Desktop Computing or **Volunteer Computing** is a distributed computing paradigm in which resources are donated voluntarily by computer owners. To make it attractive to participate in such projects, impact on the owners' computers is minimized by only exploiting idle resources, such as idle CPU cycles or free disk space. The actual hardware that is donated can range from low-end desktops, having only a CPU, to high-end desktops featuring a GPU. A recent trend is to include gaming consoles as part of the platform, thus taking advantage of the high performance provided by accelerators [95]. This allows desktop computing platforms to reach an aggregated performance of over 6 PFLOPS [123]. Considering that such a platform would rank 4^{th} on the Top 500 list, volunteer computing is an attractive alternative to managing a costly infrastructure.

Desktop computing has the following properties: First, the platform is distributed and highly heterogeneous, as highlighted above. Second, since owners participate voluntarily, nodes can enter and leave the system at a high rate. This high volatility makes it difficult to harness its capacity for large, tightly-coupled simulations [13]. Moreover, even when running bag-of-tasks, guaranteeing nontrivial Quality of Service (QoS) is difficult. This is why we chose to classify Desktop Computing separately from Grid Computing, despite it being often referred to as *Desktop Grids* in literature.

In order to improve resource utilization, the solutions provided in this Thesis require direct control over the resources, which is not feasible in Desktop computing. Therefore, these platforms are out of scope.

2.1.5 Cloud Computing

Let us start presenting Cloud Computing by giving the NIST definition [80]:

cloud computing

> **Cloud computing** is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

Depending on the level of abstraction at which the computing resources are provided, Cloud Computing can be classified as:

Software as a Service (SaaS) allows a Cloud user to provision with ready-to-use software. Examples include popular web-mails, such as Yahoo! Mail, on-line document management software, such as Google Docs, etc.

Platform as a Service (PaaS) gives the Cloud user access to programming languages, libraries and tools, to create and deploy software on the provides premises. For example, in order to run a computation on a PaaS cloud, users might have to use the MapReduce [39] programming model, which on one hand eases the burden of dealing with large data sets, load balancing and fault tolerance, but on the other hand limits the kind of application

that can be executed. Popular examples of PaaS offers include Amazon Web Services [121], Google App Engine [131], etc.

Infrastructure as a Service (IaaS) provisions the user with virtual hardware, such as virtual machines, virtual network, etc. Popular examples include Amazon EC2 [120], Microsoft Azure [149] and Rackspace Cloud [141].

Hardware as a Service (HaaS) provisions the user with bare-metal hardware, such as physical machines. Grid'5000 [12], the French reconfigurable experimental computing platform, is an example of an HaaS. A similar initiative in the USA is called FutureGrid [114].

In this Thesis, we are mostly interested in IaaS. The interfaces offered by PaaS and SaaS are meant to hide resource management details from the users, therefore, the problematics raised in this Thesis are not well highlighted from those perspectives. Note that, resource management issues do appear when *implementing* a PaaS. However, this is no different than considering the PaaS implementation a consumer of an IaaS interface. Regarding HaaS, resource management in this context is no different than managing a cluster, which is why we shall not insist on in this section.

In exchange, IaaS brings new opportunities for a more flexible resource management. Since the allocated resources are virtual, the physical-to-virtual mapping can be transparently changed without the user's knowledge. For example, this could be used to consolidate computing nodes in order to save energy [58].

Nevertheless, there are some intricacies of resource management in IaaS. First, while IaaS aims at abstracting resources, it cannot offer a homogeneous view of them, since the underlying physical resources themselves are not homogeneous. Indeed, Amazon EC2 offers several instance types, ranging from those with little processing power and slow network, to powerful HPC instances with a 10 Gbps full-bisection network. Second, **public Cloud**s, i.e., those operated by a commercial provider, charge for using the resources, usually at a given granularity, such as an hour. Therefore, resource selection not only has to take into account the performance of the selected resources, but also their prices.

public cloud

Sky Computing

Sky Computing is an emerging computing paradigm which consists in on-demand resource acquisition from several Cloud provides [69]. This makes resource management even more difficult, as the user no longer needs to negotiate with a single provider, but needs to coordinate among multiple providers. This is somewhat similar to Grid Computing, except that a new dimension is added to the problem: resource allocation is charged.

sky computing

2.1.6 Hybrid Computing Infrastructures

The above infrastructures can be combined, so as to obtain a system with improved properties. Let us give three examples of such hybrid infrastructures.

Cloud with Desktop Computing As previously highlighted, Desktop computing offers a tremendous amount of computing power at a low price. However, due to the high volatility, efficiently harnessing this computing power can be difficult. As a solution, Desktop computing can be combined with Cloud resources, so as to reduce volatility while at the same time keeping costs down [40].

	Heterogeneity	Scale[1]	Domains[2]	Volatility	Type
Supercomputer	low–medium	building	single	low[3]	physical
Single cluster	low	building	single	low	physical
Multiple clusters	medium	building	single	low	physical
Grid	high	continent	multiple	high	physical
Desktop	very high	world	many	very high	idle physical[4]
Cloud	medium	continents	single	low	virtual
Sky	medium	world	multiple	low	virtual
Hybrid	high	world	many	low	mixed

[1] i.e., geographic scale
[2] number of administrative domains, i.e., number of different organizations controlling the resources
[3] systems comprised of many cores are necessarily volatile [23]
[4] allocations are only made from idle resources

Table 2.1: Intrinsic properties of HPC resources

Cloud with Grid Computing Grid computing requires partner institutions to provision for enough resources, so as to satisfy the resource requirements of all users. However, the load of the platform might greatly vary, having low and high resource utilization periods. For example, the Grid'5000 experimental platform is lightly loaded, except before an important conference deadline [94] when resource utilization is high. Provisioning in order to meet demand during these peak periods would be inefficient, since resources would be idle for most of the time.

In order to reduce the costs of running a Grid, while at the same time improving user-perceived performance, capacity provided by the Grid resources can be augmented with Cloud resources [116]. When the load of the Grid is too high, virtual machines are leased from the Cloud provider. This allows reducing costs and improving user satisfaction at the same time.

Desktop and Grid Computing Augmenting the capacity of a Grid system can also be done by adding Desktop computing resources [113]. For example, user resource requests can be balanced between the two types of resources, so as to obtain good performance and reduced cost.

In this Thesis we shall make a simplification and shall not specifically deal with hybrid infrastructures. Nevertheless, some of our propositions could be extended to them.

2.1.7 Analysis

Let us analyze the previously presented HPC resources as synthesised in Table 2.1. A first observation to make is that resources are becoming increasingly heterogeneous. Computation-wise, it is nowadays common not only to find heterogeneity inside a computing node as CPUs and GPUs co-exist, but also among nodes. Communication-wise, the interconnecting network is also heterogeneous, meaning that the performance observed by an application depends on its mapping on the resources. Moreover, resources are constantly changing. Whether tomorrow's resources will comprise many small cores, few large cores, or a combination thereof is yet to be decided. Soft-

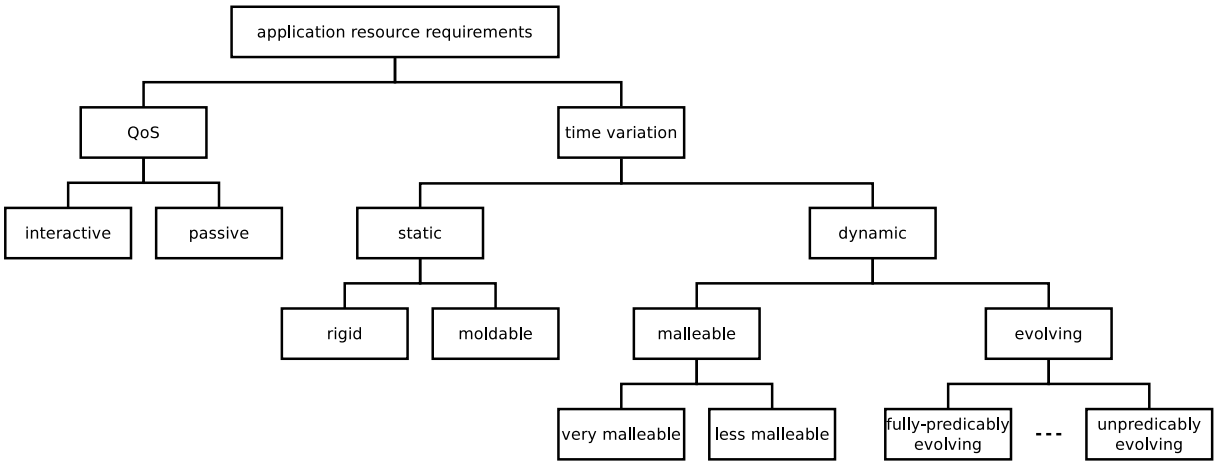


Figure 2.1: A classification of applications by resource requirements

ware platforms will have to be flexible enough, so as to guarantee their independent evolution, without having to upgrade them in lock-step with the resources.

Second, HPC resources are geographically dispersed, belonging to multiple administrative domains. Regarding the former, platforms need to be scalable and fault tolerant, despite the high latencies separating the resources and an increased likelihood of network failures. Regarding the latter, institutions should be allowed to implement their own resource allocation policies, otherwise convincing stakeholders to mutualize resources could prove difficult.

All the above have an impact on the HPC platform. Indeed, HPC resources are simultaneously used by multiple users. To ensure that their applications run at peak performance, one needs to efficiently allocate resources to them. To achieve this, both application and resource characteristics have to be taken into account. An RMS, the part of the platform responsible for multiplexing resources among multiple users, needs to provide good interfaces to allow for this. Before studying what interfaces are currently proposed in the state of the art, let us analyze the resource requirements of applications.

2.2 A Classification of Applications by Resource Requirements

Let us start by giving some definitions aimed a cleaning up confusion. We call an **application** a set of executables and libraries having the goal of processing input data and transforming it into output data, which is useful to a user. Since we are mostly interested in distributed- and parallel-applications, we consider that applications are composed of one-or-more **task**s which do intermediate transformations, producing intermediate output data.

application

task

Distinguishing an application from its constituent tasks is important for our classification. Indeed, in many resource management problems the two concepts may be mixed. Applications may be presented to the system in any of the three following ways:

- An application is submitted "as a whole" to the system: For one application, the system manages one single entity. For example, tightly coupled parallel applications are usually handled this way [47].
- The tasks of an application are each submitted to the system separately: For one application, the system manages several entities. For example, parameter sweep applications are often treated as such [63].

– Several applications are submitted "as a whole" to the system: For several applications, the system manages one single entity. For example, there are some workload management systems that aggregate several applications [31].

Having clarified on what we call an application, in the rest of this section, we analyze the different types of applications depending on their resource requirements. Figure 2.1 offers an overview of the properties we found relevant for our analysis, which we have grouped according to two criteria: QoS requirements and time variation.

2.2.1 QoS Requirements

quality of service

Quality of Service (QoS) is a generic term to describe the fact that services offered to a consumer should have some guarantees. This is in contrast to best-effort delivery, where the service is offered without any guarantees. The most common use of the word QoS is in packet-switched networks, where some packets (e.g., those that are part of a file download) are delivered in best-effort mode, while others (e.g., those that are part of an audio call) are delivered with strict QoS requirements, such as at most 35 ms latency, at most 10% jitter and no-loss delivery.

Similarly, since allocating resources is a service, one can assign to it QoS constraints such as: start-time of the allocation, deadline of the allocation, maximum allowed failure rate of computing nodes, etc. In order to simplify the classification of applications by QoS, we shall group them into two categories: interactive and passive.

interactive

Interactive applications have strict QoS requirements, for example, because they need to cooperate with external entities, such as scientific instruments or human operators. These applications require the allocation to start at a precise moment of time. Once the allocation has started, these applications cannot tolerate a degradation of the allocation, e.g., suspending an application so as to resume it later is not an option. Examples of such applications include web servers, Massive Multi-player Online Role-Playing Game (MMORPG) servers and distributed visualization applications.

passive

Passive applications, in contrast to interactive ones, have more relaxed QoS requirements. They do not interact with external entities, therefore, more liberty can be taken when allocating resources to such applications. For example, the application's start-time may be delayed. Also, during their execution, these applications can be killed and restarted later, suspended and resumed later, or check-pointed and restarted. Nevertheless, the allocation of resources has to be made carefully. At the very least it should be guaranteed that the application will eventually finish. If specified by the user, other constraints might need to be taken into account, such as a deadline (i.e., maximum allowed completion time), a budget (i.e., maximum allowed price), energy consumption, etc. Examples of passive applications include video conversion software, simulation software and build servers.

When allocating resources to interactive applications, one does not have a lot of choice: Either the resources are allocated or the application runs in degraded mode (e.g., web pages are served without pictures or videos). This approach has been extensively studied as part of the SelfXL project [90]. In contrast, this Thesis focuses on efficiently running passive applications, such as large-scale numerical simulations.

2.2.2 Time Variation of Resource Requirements

Let us characterize applications depending on when the resource requirements of the application have been fixed. We distinguish three phases of the application's life-cycle:

– **development**: the application's code is being written by a development team;

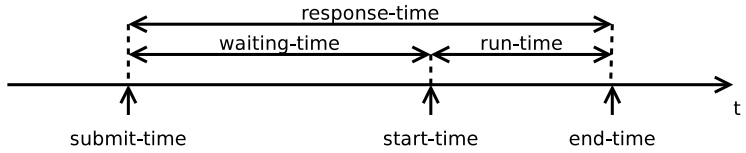


Figure 2.2: Terminology used for expressing times throughout the life-cycle of an application

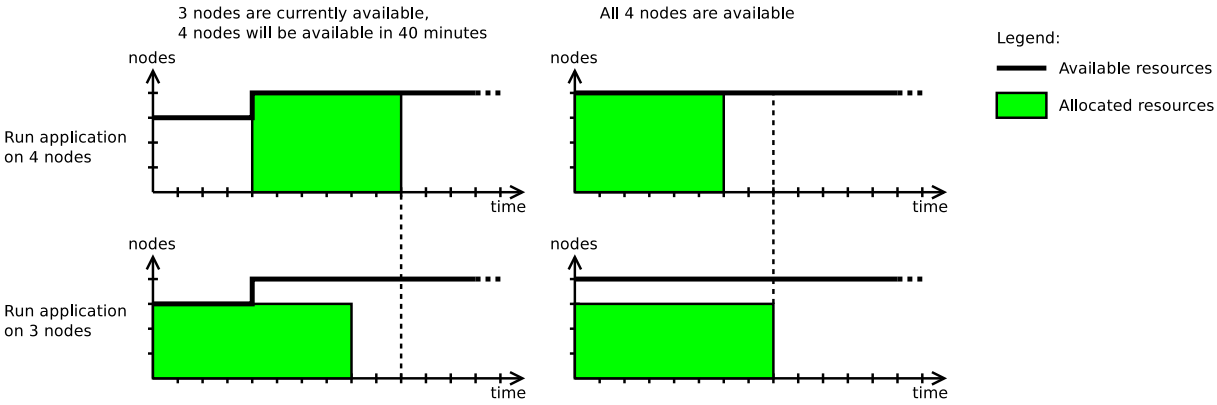


Figure 2.3: Graphical explanation of how to improve response time using moldability

– **submission**: the user has submitted the application for execution to an HPC platform, but no computing resources have been allocated to it yet;
– **execution**: the application is running its code.

In this Thesis, we are mostly interested in the user-perceived performance of an application, after the application has been delivered to the end-user. For this purpose, we use the application's **response-time**, which measures how quickly the user receives the results after the application has been submitted. Figure 2.2 shows that the response time is the sum of the **waiting-time**, the duration the application had to wait for resources to become available, and the **run-time**, the duration the application actually did computations. Depending on how resource requirements are devised, an application may find an optimal compromise between the waiting-time and the run-time, so as to minimize the end-time.

response-time

waiting-time

run-time

In the remaining of this section, we introduce terminology based on Feitelson's taxonomy [46]. Applications whose resource requirements are fixed at development or submission are called static, while the others dynamic.

Static applications

The resource requirements of a **static** application are fixed before execution and cannot change afterwards: The resources used by a static application are constant throughout its execution. The resource requirements can either be fixed at development, in which case the application is called **rigid**, or at submittal, in which case it is called **moldable**. Rigid applications can only run on a hard-coded configuration of resources, whereas moldable applications can run on a wide variety of configurations.

static

rigid

moldable

Moldability allows to improve resource utilization and application performance by allowing more flexibility in how the allocation can be made [59]. Moldability allows to find at submittal the best trade-off between the waiting-time and the run-time, so as to minimize response-time.

Figure 2.3 illustrates this: Let us assume that an application requires either 4 nodes for 1 hour or 3 nodes for 1 hour 20 minutes. If the platform has 4 nodes, one of which is only available in 40 minutes, it is better to run the application on 3 nodes (instead of 4): The run-time will be 20 minutes greater, however since the waiting-time is 0, the final response-time will be reduced. Conversely, if all 4 nodes are available right-away, then the optimum choice is to allocate all nodes to the application.

Many HPC applications are already moldable [105]. For example, many applications coded using Message-Passing Interface (MPI) partition the input data when they start executing. Since data migration is left to be implemented by the programmer [43], these applications generally do not support changing their resource allocation during execution.

Dynamic applications

dynamic

As opposed to static applications, the allocated resources of a **dynamic** application can change throughout its execution. Unfortunately, the word dynamic is very vague, as it does not characterize *who* or *what* is determining the allocation to change. Therefore, dynamic application can further be characterized as being malleable and/or evolving.

malleable

Malleable applications The allocated resources of a **malleable** application can change (grow/shrink) due to resource-related constraints. For example, if additional resources are available, an application may be grown so as to take advantage of them for increased speed-up. Similarly, if resources are becoming scarce, an application may be shrunken, so as to conform to a system policy. When the set of allocated resources changes, a malleable application has to adapt, which can be a more-or-less costly operation. Depending on the required bandwidth, time, CPU power and disk capacity wasted[1] during the adaptation process, we identify two extreme cases:

bag of tasks

parameter-sweep application

- Applications that can quickly grow/shrink as required by the availability of the resources, for example, **Bag of Tasks (BoT)** and **Parameter-Sweep Application (PSA)** can easily be adapted to a varying number of resources during their execution. Growing is handled by launching new tasks on the newly allocated resources, while shrinking is handled either using checkpoint-resume or kill-restart [102]. Since tasks are generally short compared to the duration of the whole application, adaptation can be achieved while wasting only a small percentage of resources.
- Applications that are heavily penalized for each adaptation, for example, applications that need to migrate large amounts of out-of-core data.

To efficiently make use of resources, a malleable application requires that its allocated resources be within its minimum and maximum requirements. Such requirements can usually be determined at the start of the application. Minimum requirements are imposed by the space complexity of the computations, e.g., the amount of memory needed to store the working set. Shrinking the application below this level forces it either to abort computation or to checkpoint and restart later. Regarding maximum requirements, the speed-up curve of every application has a global maximum [42], which limits its parallelism. Therefore, even though a malleable application could be grown beyond its maximum requirements, it would not make effective usage of the extra resources.

[1] We use the word "wasted" as we consider that adaptation is a secondary concern of the application, the primary one being to complete computation and output the results. In no way do we want to suggest that adaptation is not necessary, or that adaptation is hindering the application from executing efficiently.

It has been shown that malleability improves both resource utilization and application response-time [59]. Despite this, in practice, few applications are malleable, due to the fact that commonly used programming models (such as MPI) leave the burden of implementing malleability to the programmer [43]. Moreover, even if applications were malleable, taking advantage of it would be difficult, due to the limited availability of RMS support.

Evolving applications The allocated resource of an **evolving** application changes due to the fact that the application's resource requirements change during its execution. This can either happen because the computation has grown or shrunken in complexity (e.g., it requires more Random-Access Memory (RAM) or more temporary disk space), or because of some external factors, e.g., a human operator changed some parameters of the computation. While not a passive application, one of the most common evolving applications is a web server, which can exhibit evolving resource requirements, as a result of the varying number of users that it serves.

evolving

Depending on how much time in advance the evolution can be predicted (called the **horizon of prediction**), evolving applications can be divided into three subcategories:

horizon of prediction

- **fully-predictably** evolving applications are those whose evolution can be fully described at the beginning of their execution, i.e., their horizon of prediction is infinite;
- **marginally-predictably** evolving applications have a finite horizon of prediction, e.g., it can predict 5 minutes ahead that more resources will be required;
- **Non-predictably** Evolving Applications (NEA) change their requirements, without making it possible to have any predictions (i.e., the horizon of prediction is zero).

As examples, applications which are described as **workflows** are often evolving. The workflows may have various branching factors [10], which increase/decrease the resource requirements. Regarding predictability, workflows may be fully-predictable if all the tasks are known at start-time (in literature these are commonly called **static workflows** [117]). However, some workflows create tasks depending on the output of previous tasks, in which case, the application is only marginally predictable.

At any rate, no matter what the horizon of prediction, exporting this information outside the application would allow the system, and possibly other applications, to improve resource management decisions. For example, a moldable application could make use of this information to decide on how many resources it should allocate, whether to wait for more nodes to become available or not.

Malleable and evolving applications As it has already been suggested, applications can be both malleable and evolving. For example, either the minimum or the maximum requirements might change as a function of the application's internal state. However, the application can efficiently do computations as long as the allocated resources are within this range.

A good example are Adaptive Mesh Refinement (AMR) simulations, which are gaining increasing momentum [97]. Instead of using a discretization of space computed at start-time (i.e., a static mesh), fine enough to ensure numerical precision throughout the whole domain, AMR dynamically coarsens/refines the mesh on those part of the domain where interesting phenomena happen, as required by numerical precision. This allows using the same computing power to solve problems which are at least an order-of-a-magnitude more complex [17].

Since the simulated phenomena are rather chaotic, AMR applications are necessarily non-predictably evolving. If implemented property, such applications can be made malleable, with minimum requirements as low as to keep the refined mesh in memory, and with maximum requirements as high as to make the application execute efficiently (e.g., at their maximal speedup).

Unfortunately, as of today, most AMR applications are programmed so that the varying resource requirements are not externally exposed. They are written as moldable applications [15], with the number of processors empirically chosen at submittal. Dynamically allocating resources to such an application would allow it to run efficiently throughout its whole execution. Moreover, being able to provide applications with more nodes and memory on the fly is considered necessary for achieving exascale computing [41].

2.2.3 Conclusion

To sum up, based on their resource requirements, applications can either be static or dynamic. Static applications have their resource requirements fixed before execution: They can be either rigid, having their requirements fixed at development, or moldable, having their requirements fixed at submittal.

Dynamic application can have variable resource allocation during execution. For malleable applications, the allocation change is determined by the system, whereas for evolving ones, the allocation change is initiated by the application. An application can be both malleable and evolving at the same time.

At any rate, not dealing properly with each kind of application, for example, not taking advantage of an application's malleability, misses an opportunity at optimizing application performance and resource usage.

2.3 Managing HPC Resources

This section presents the state of the art in HPC resource management. It presents and analyzes resource allocation abstractions and their implementation, which are grouped in five classes by the degree of control they have on taking allocation decisions. In decreasing oder of control, the section discusses Cloud managers, batch schedulers, distributed resource managers, meta-scheduler and application-level schedulers.

2.3.1 Cloud Managers

cloud manager

IaaS **Cloud Manager (CM)** are software solutions which manage bare hardware, such as computing nodes, storage and network, and expose it as a virtual infrastructure: Virtual Machines (VMs), virtual storage, virtual networks. The most popular open-source CMs are Eucalyptus [92], OpenNebula [104], OpenStack [96] and Nimbus [137].

Most of these CMs expose APIs such as the de facto Amazon EC2 [120] standard or the Open Cloud Computing Interface (OCCI) [93] standard. With respect to resource management, they basically offer the same services centered around leasing VMs[2]: The user decides when a virtual machine is instantiated or destroyed, for which she is billed with a certain granularity (e.g., hourly). An exception to this is Amazon's EC2 offer, which proposes **spot instances** [119]: The system decides when to instantiate and release VMs based on a price set by the user and the current market value of resources.

spot instances

These abstractions are simple, yet powerful enough for many applications, especially interactive ones. For example, a web hosting application can instantiate/release VMs to adapt to changing web traffic [28]. This allows both coping with the load, while at the same time only

[2]Cloud managers also have other functions, such as communicating with the computing nodes' hypervisors, managing virtual images and storage. While important, these are not relevant for resource management.

allocating resources that can be effectively used. On the other hand, if the current mapping between VMs and physical machines leaves the latter underutilized, VMs can be migrated so as to shutdown machines and save energy [58]. Spot instances can be used for allocating resources to low-priority malleable applications, with relaxed deadline constrains.

However, this interface is insufficient for dealing with large-scale passive applications. The problem steams from the fact that in the context of IaaS Clouds it is often assumed that the physical platform is either large enough to cover all on-demand needs or that applications can somehow deal with receiving fewer VMs than requested [90]. This is not the case for HPC applications, which may require the whole platform [17]. If two such applications enter the system at the same time, the CM has no choice but to signal an "out-of-capacity" error [125].

In order to make Cloud infrastructures more HPC-friendly, Haizea [103] proposes a custom scheduler on top of OpenNebula. It augments the classic Cloud interface with concepts to be found in batch schedulers, such as submitting rigid jobs and making advance reservations. We shall describe them with more details in the next section, which is dedicated to batch schedulers.

2.3.2 Batch Schedulers

Batch schedulers are resource managers designed for clusters or supercomputers. Inspired by batch processing from the 1950s, the main idea of batch schedulers is that system throughput can be improved by running applications one after another instead of concurrently [76]. Being a key component of an HPC platform, there are quite a few batch scheduler implementations. They range from production-oriented schedulers, such as LoadLeveler [68], Sun Grid Engine [51], SLURM [66], TORQUE [147], to research-oriented ones, such as OAR [22] or ReSHAPE [108]. batch schedulers

Regarding the interface they expose, batch schedulers try to be as generalist as possible, so as to satisfy the needs of many users. Most of them provide two methods: submitting rigid jobs and making advance reservations. Other, more innovative concepts include moldable jobs, low-priority jobs, support for dynamic resizing of homogeneous applications, dynamic jobs. Let us analyze each of these concepts.

Submitting Rigid Jobs

For submitting a **rigid job**, the user specifies a node-count and a maximum execution time. To decide the start-times, the resource manager schedules jobs in space-sharing mode using First-Come First-Serve (FCFS) [99] augmented with aggressive back-filling (EASY) or Conservative Back-Filling (CBF). EASY [76] backfills a job as long as the start-time of the *first* queued job stays the same, while CBF [86] backfills a job as long as the start-time of *all* previously queued jobs are unchanged. Both backfilling strategies improve resource utilization, while at the same time ensuring reasonable fairness among jobs. rigid job

Alternatively, resources could be allocated in time-sharing mode. For example, several applications could be running on the same nodes. In order to ensure reasonable performance one would have to use coarse-grained gang-scheduling, i.e., simultaneously do context switching on all the nodes that an application is running on. Still, due to the relatively high cost of gang-scheduling, time-sharing is rarely employed in practice [47].

At any rate, as has previously been mentioned in Section 2.2, most HPC applications are moldable. Submitting them as rigid jobs reduces the flexibility one has in allocating resources, which leads both to inefficient resource usage and reduced application performance [59]. One could try to emulate moldability by cancelling a job and resubmitting a new one (with a different

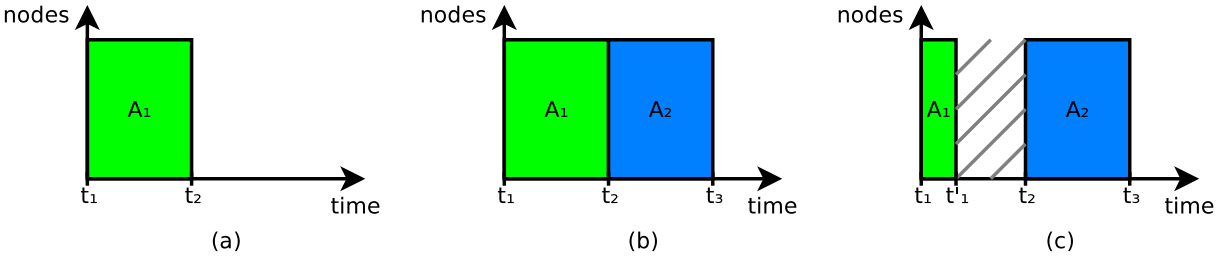


Figure 2.4: Resource waste when using advance reservations: (a) application A_1 enters the system and requests nodes from t_1 to t_2; (b) application A_2 enters the system and requests nodes from t_2 to t_3; (c) application A_1 finishes earlier at $t'_1 < t_2$. The system cannot advance the allocation of A_2, thus, a hole is left in the resource allocation chart.

node-count), however, the job would be resubmitted at the end of the queue, hence, it would lose its implicit priority based on submission time.

Advance Reservations

advance reservations

For **advance reservations**, besides the node-count and maximum execution time, the user additionally specifies a start-time. The resource manager either guarantees that the allocation can start at the given time or the reservation is rejected. Advance reservations are useful when having to synchronize the application with external entities, such as scientific instruments [1] or when needing simultaneous access to computing resources that are managed by different batch schedulers (i.e., **co-allocation**).

co-allocation

On the downside, advance reservation are sometimes inefficient. Since the resource manager is not allowed to choose the start-time, advance reservations waste resources by leaving "holes" in the resource allocation chart [101], as shown in Figure 2.4. Therefore, advance reservations are rarely used in production systems.

Moldable Jobs

A simple support for moldable applications consists in specifying a range of node-counts with a single maximum execution time, as implemented in SLURM. This simpler form of moldability allows to improve application performance by allocating more resources (and reducing execution time) if enough nodes are available. However, since the job specification only includes one single maximum execution time, one cannot express the fact that allocating more nodes to the application reduces its execution time. Therefore, backfilling opportunities might be missed, in which case the application's start-time is delayed.

moldable jobs

moldable configurations

Improved support for moldable applications can be achieved using **moldable jobs**: When submitting a job, a user is allowed to list several **moldable configurations**, which associate a node-count to a maximum execution time. Then, when starting the application, the resource manager chooses the one that minimizes an internal criterion. Moldable jobs are supported both by OAR and TORQUE[3]. OAR choses the configuration which greedily minimizes application finish time, while TORQUE attempts to choose the one which improves resource usage.

[3]These are called "malleable jobs" in the TORQUE User Manual. In this Thesis, we prefer to use Feitelson's terminology [46] and call them "moldable".

While moldable jobs are a considerable step towards supporting moldable applications, there are still some cases where it proves inadequate. For example, let us assume that one wants to describe all configurations that an application may run on, given a cluster of 1024 nodes. One would have to enumerate 1024 configurations: For each node-count from 1 to 1024 one would have to give a separate maximum execution time. If there are many moldable applications that are queued in the system, the resource manager might become overloaded.

This problem is somewhat more aggravated on supercomputers, for which the number of computing nodes is of the order of $100,000$ (see Section 2.1.1). Moreover, some applications are locality-sensitive [89]: For a constant number of nodes, their performance is significantly slowed down when the allocation is non-continuous. Using moldable jobs to express this, so as to optimally select resource for these applications, might significantly increase the number of configurations.

This situation becomes impractical in case of multi-cluster applications. If a platform is composed of c clusters each having n nodes, in order to list all possible configurations, one would need to input $(n+1)^c - 1$ configurations: On each cluster one chooses any node-count from 0 to n, thus giving $(n+1)^c$ combinations, from which the configuration without any nodes is excluded (hence the -1 in the above formula). Clearly, having an exponential number of configurations would not scale. Assuming a modest data-center of 4 clusters with 128 nodes each, the number of configurations to enumerate is of the order of 10^8.

Low-priority Jobs

OAR supports low-priority, preemptible jobs called "best-effort" which are similar to Cloud spot instances. These jobs are only launched if enough resources are left idle by regular jobs. Also, if there are insufficient resources to serve a regular job, best-effort jobs are killed after a grace period. This type of job can be used for building a limited form of malleability, whose benefits and drawbacks are discussed in Section 2.3.5.

Support for Homogeneous Applications

Homogeneous applications are iterative applications which respect two conditions: First, computation and data are relatively uniformly distributed on each node and, second, each iteration does approximately the same amount of work.

homogeneous applications

ReSHAPE [108] is a framework which supports dynamic resizing of homogeneous applications. One of its components is a prototype scheduler which decides how resources are allocated among applications in the system. According to our classification, this scheduler can best be described as a specialized batch scheduler.

ReSHAPE works as follows. Application need to be re-engineered so as to implemented a redistribution function and interact with the ReSHAPE Application Programming Interface (API). At the end of every iteration, they contact the scheduler with some information, and receive grow/shrink commands. The scheduler retains a cache of obtained application performance and attempts to give resources to the application which would most benefit from it. The approach improves resource utilization and application response times.

Despite its elegant solution, the ReSHAPE framework has some disadvantages. First, it offers a very limited type of malleability, mainly targeted at homogeneous applications. Among other, PSAs cannot be efficiently scheduling using the proposed approach. Second, since the performance of the application as a function of the allocated notes is cached, evolving applications, such as

AMR simulations, have no mean to declare their changing resource requirements. Thus, resource allocation to such applications is suboptimal.

Dynamic Jobs

An increasing effort is devoted to enabling batch schedulers to change the resources allocated to a job at run-time, as set forward by the Exascale Roadmap [41]. At least SLURM and TORQUE allow jobs to be dynamically resized (grown or shrunken) during execution. This might be useful to allow evolving applications to acquire resource on-the-fly, instead of having to reserve as many resources as to fulfill their peak demands. This would improve resource utilization, as resources that cannot be efficiently used by the evolving application could be filled by another application that can make a more efficient use of them.

Growing and shrinking is handled slightly differently in SLURM and TORQUE. In SLURM, changing the size of the allocation is directly initiated by the user [142]. For shrinking, the user calls a special update command to release nodes to the system. For growing a job J, the user submits a new job J', then, when job J' has started, she may merge the resources of two jobs J and J'. In TORQUE [147], the RMS regularly queries each application what its current load is, then decides how resources are allocated. The application has to provide a custom script, which is called by TORQUE to enact the growing or shrinking of the application.

Unfortunately, none of the proposed solutions address a fundamental issue: How can the resource manager guarantee the availability of the resources when an evolving application needs to grow? Without this guarantee, evolving applications would have to resort to one of the two following solutions: (i) either wait for more resources to become available, possibly leading to a dead-lock if two evolving applications need to grow at the same time; (ii) checkpoint and resubmit a larger resource request, thus having to wait for queued applications to finish. In both cases, evolving applications would be delayed, i.e., their end-time is increased. Sadly, they would be penalized for attempting to more efficiently allocate resources. Obviously, it would be difficult to convince users to accept this situation, therefore, they would most likely continue to allocate resources for peak demand.

2.3.3 Distributed Resource Managers

distributed resource manager

In this Thesis, we call a **Distributed Resource Manager (DRM)** a set of agent, which have *direct* control over the resources, *collaborating* in order to allocate multiple, independent resources to multiple users. This is in contrast to a batch scheduler or a CM, which are centralized. Also, when using multiple batch schedulers (as is done in many Grid environments), they would generally not cooperate.

The purpose of distributed resource managers is twofold: scalability and fault tolerance. Regarding scalability, since the information related to the state of the resources is distributed on multiple agents, a DRM can handle more resources. As for fault tolerance, if one agent fails, only part of the resources become unaccessible. Also, if the resources get bisected due to a network failure, users can still access resources which are on their side of the bisection.

DRMs aim at hiding the complexity of the platform from the users, therefore, they mostly offer Cloud-like or batch-like interfaces. To better illustrate how DRMs work, let us describe two of them: XtreemOS and DIET.

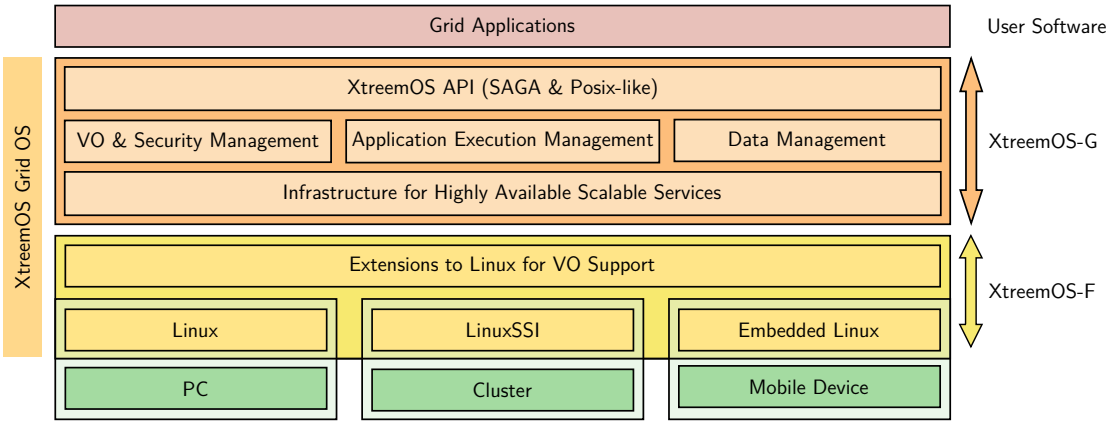


Figure 2.5: Overview of XtreemOS's architecture

XtreemOS

XtreemOS [84] is a grid operating system designed to ease user access to a large-scale platform, managed by multiple, independent institutions. XtreemOS aims at offering for Grids the same ease of use as traditional operating systems do for a single computer.

Figure 2.5 gives an overview of XtreemOS's architecture. On top, applications interact with the system through some APIs: Legacy applications use the POSIX API, while applications designed specifically for the Grid use the SAGA API [54]. This gives them access to the services provided by the following modules:

Virtual Organization (VO) and Security Management This module allows applications access to the whole platform without having to worry about credentials, i.e., it provides single sign-on semantics.

Application Execution Management This module allocates resources, launches processes, etc. We shall return to this module and describe it in more details shortly.

Data management This module allows applications to transparently access files, no matter what their physical location on the platform. This is implemented using XtreemFS [60] a scalable, distributed file system.

These modules form together an XtreemOS agent, which runs on each computing node that is part of an XtreemOS platform[4]. Communication among the modules is done using a highly-available and scalable message-passing infrastructure.

At the lower layers, the GNU/Linux system is extended for native single sign-on and VO support. This allows POSIX user identifiers (IDs) to be transparently mapped to grid user IDs. The Linux kernel itself is specialized depending on the target platform.

Let us focus on how resource management is done in XtreemOS. The main module that takes care of this is the Application Execution Manager (AEM) [91]. Applications formulate resource requests using the Job Submission Description Language (JSDL) [6] specifying constraints such as minimum amount of memory, number of CPUs, etc. Regarding time, resources can be reserved in two ways: either for immediate usage or at a future date (i.e., advance reservation). In both cases allocations are done either exclusively, i.e., no two applications can run on the same nodes, or non-exclusively.

[4]We have taken the liberty to simplify the description. In reality, XtreemOS agents can have several roles and run different modules depending on their roles. For details see the deliverables of the XtreemOS project.

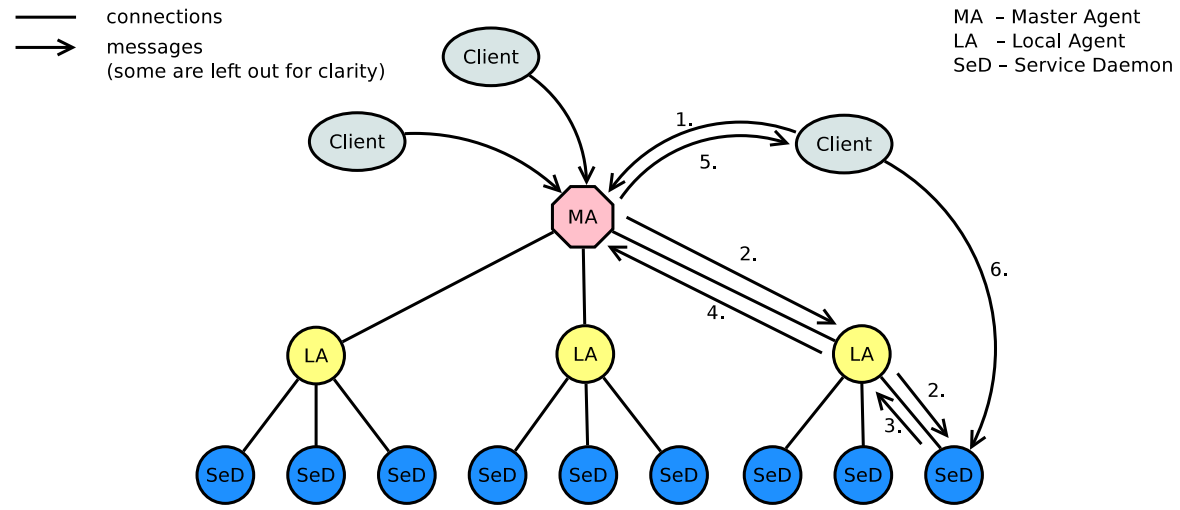


Figure 2.6: Overview of DIET's architecture

To discover fitting resources, AEM first uses a Distributed Hash Table (DHT) [100], which returns a list of candidate resources. Next, AEM contacts each resource, checks its local calendar and reserves a time-slot. If a failure is encountered during the process (e.g., insufficient resources have been discovered, calendars changed just before making the reservation, etc.) AEM retries up to three times, after which an error is reported to the upper layers.

To simplify the management of clusters, XtreemOS uses LinuxSSI, which presents multiple nodes as a single, large SMP machine. This makes resource allocation somewhat easier, since a single XtreemOS agent handles resource allocations on a whole cluster, instead of having to run an XtreemOS agent per node.

Despite the easy-to-use, high-level resource management abstractions that XtreemOS offers, there are three shortcoming. First, non-exclusive reservations are not suitable for HPC applications [45]. Gang-scheduling could be employed as a solution, however, it is difficult to implement it in a distributed system without a central coordinator. Second, XtreemOS does not have a queuing system and only uses advance reservations. As discussed before, this leads to an inefficient resource usage. Third, resource discovery and reservation are not transactional. If several moldable applications entered the system and attempt to reserve the same resources, but in a different order, they may become trapped into a live-lock.

DIET

Distributed Interactive Engineering Toolbox (DIET) [29] is a framework centered around the GridRPC [87] paradigm, designed to allow easy, efficient and scalable access to heterogeneous computing resources. In this section, we describe the general architecture of DIET and show how it can implement a distributed resource manager.

Figure 2.6 gives an overview of the DIET architecture. A **client** is an application that uses the DIET infrastructure to solve computation problems using an RPC approach. A **Service Daemon (SeD)** is a server application exporting computational services through established interfaces. The two are connected through a hierarchy of **agents** which help clients find the best SeD(s) to execute their request(s) on. The SeDs form the leaves of the hierarchy, while the root is a distinguished agent called the Master Agent (MA). The latter acts as an entry-point for the

clients. The other agents, called Local Agents (LAs), help achieve scalability.

A typical DIET interaction goes as follows. (1) First, the client enters the system and sends a "light" version[5] of its request to the MA. (2) Next, the request is propagated from the MA to the SeDs through LAs. The request eventually reaches all SeDs, (3) which reply with an **estimation vector**: a set of values describing how fit they are for solving this particular request. Depending on the implementation of a service, an estimation vector might contain information such as CPU power (in FLOPS), amount of RAM, estimated completion time of the request, number of queued requests, etc. (4) At each level of the agent hierarchy the estimation vectors are aggregated, so that the MA only has to deal with a small amount of them. (5) Finally, one or more estimation vectors are returned to the client, which chooses a suitable SeD. (6) The request with all the data necessary to solve the problem is sent directly from the client to the SeD.

estimation vector

Let us highlight where scheduling decisions are taken. First, SeDs decide what values to put in an estimation vector. This way, a SeD can make itself more or less preferred for solving a problem. Second, agents throughout the hierarchy aggregate and filter SeDs based on these estimation vectors. For example, only the top 10 least loaded SeDs are returned to the parent agent, thus leaving out overloaded SeDs. Third, clients receive estimation vectors from several SeDs, out of which they may choose a SeD.

Using the DIET framework, a distributed resource manager can be implemented as follows. The administrator of the platform installs one SeD per computing node, then uses other nodes (e.g., cluster front-end nodes) to set up an agent hierarchy. For improved performance, this hierarchy should be optimized for the targeted resources [36, 26]. Thus, the SeDs have direct control over the resources: They can take all the liberty in implementing any resource management policy and they can gather all the information they need about the resources they manage. As an example of resource management policy, DIET already has a queuing system implemented which allows an administrator to choose the maximum number of requests that a SeD is serving simultaneously. Related to gathering information, DIET's CoRI [25] module gathers the metrics that are most important to optimize the resource selection for a wide range of applications.

On the positive side, the above architecture allows both the user and the administrator to have control over scheduling. The user may change the client-side scheduling algorithm, while the administrator controls the estimation vectors sent by the SeDs and how aggregation is done by each agent.

On the negative side, the obtained DRM has several limitations, many due to the fact that DIET is designed around the GridRPC paradigm and handles requests individually. First, since SeDs have unsynchronized queues, co-allocating resources managed by different SeDs is difficult and wasteful [83]. Second, clients cannot take resource-dependent decisions, e.g., adapt their requests to the state of the resources, which is required for supporting moldable applications. This is due to the fact that a request needs to be formulated *before* resources are being discovered and the *same* request is sent to be solved by the chosen SeD. For example, a client cannot improve the performance of a matrix multiplication by spreading the matrix across multiple SeDs, since splitting the matrix (and generating requests) would have to be done *after* discovering resources.

XtreemOS vs. DIET

Let us compare and contrast XtreemOS to DIET and highlight the fundamental differences between them. Both DRMs strive to obtain reasonably good scheduling with scalability in mind. XtreemOS uses a DHT to filter candidate resources based on static information, then contacts

[5] "Light" in the sense that only scalar values are sent. For vectors, matrices and strings only their size is sent.

resources directly for dynamic information. In contrast, DIET uses dynamic information from resources which is then filtered at each level of a statically-deployed tree.

Both systems supports non-exclusive and exclusive allocations. In XtreemOS it is the user who chooses what type of allocation is desired, whereas in DIET it is the administrator who sets this parameter. As discussed before, non-exclusive allocations are not suitable for HPC applications.

For exclusive reservations, XtreemOS uses calendars, whereas DIET uses queues. Calendars have the advantage of offering an easy way to do co-allocations at the expense of a less efficient resource usage. In contrast, doing co-allocation over unsynchronized queues is inefficient. Nevertheless, if co-allocation is not required, using queues leads to a more efficient resource usage: Since the start-time of requests is not fixed, requests can be advanced, so as to produce a compact schedule.

Regarding non-rigid application support, as has been previously analyzed, none of the systems support moldable applications efficiently. With respect to dynamic applications, none of the systems have a feedback mechanism which tell applications when new resources are available or when resources are too overloaded and should be freed. Therefore, it is not possible to deal with malleable applications. Also, much like CM and batch schedulers, there is no way for growing evolving applications reliably.

To sum up, both XtreemOS and DIET aim at solving the same issue, large-scale resource management, with slightly different solutions. Unfortunately, none of them provide a solution for efficiently supporting non-rigid applications.

2.3.4 Meta Schedulers

meta-scheduler

In the context of this Thesis, we define a **meta-scheduler** as a set of agents (possibly only one), which manage allocation on multiple independent resources for multiple users. The latter implies that meta-schedulers are deployed and configured by an administrator. In contrast to DRMs, meta-schedulers do not directly control the underlying resources. For this, they have to resort to the interface provided by a lower-level resource managers, called in literature a **Local Resource Management System (LRMS)**, most commonly a batch scheduler.

local resource management system

Meta-schedulers exists in order to extend functionality which is not provided natively by an LRMS, such as:

- load-balancing across multiple cluster: DIET, Legion [88];
- offering co-allocation support: KOALA;
- specialized support for certain types of applications (especially PSAs): Condor [110], DIANE [85], DIRAC, GridWay [56], KOALA, NorduGrid Broker [44].

Let us present the systems that have the most interesting features from a resource management point-of-view.

DIET

In the previous section, we showed that DIET can be used as a distributed resource manager. In this section, we show that the DIET framework is flexible enough to be used to implement a meta-scheduler.

Figure 2.7 shows a slightly different DIET architecture. The client and the agent hierarchy stay the same, instead, specialized SeDs are being deployed: `SeD_batch`, which interfaces batch schedulers, and `SeD_cloud` which interfaces cloud managers [24].

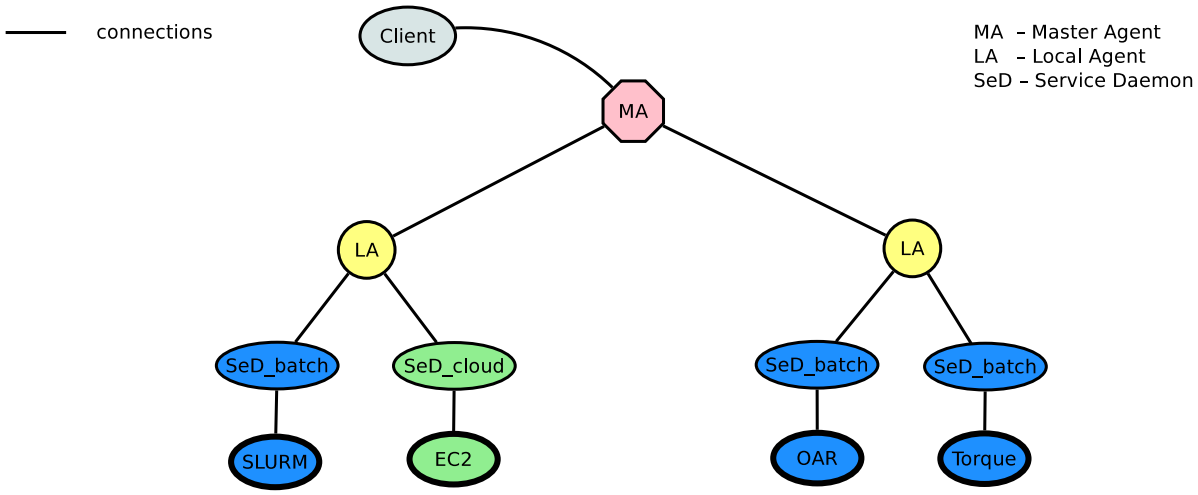


Figure 2.7: Overview of DIET's architecture when used as a meta-scheduler

`SeD_batch` is an extended SeD, which, instead of implementing a computation directly, acts as a wrapper between the DIET API and the batch scheduler. It mainly fulfills two main roles: First, during discovery `SeD_batch` gathers information from the batch scheduler and fills in the estimation vector. For this, it may use the Simbatch [21] framework, to simulate the behaviour of the scheduler's queue and better estimate the completion time of a request. Second, when the client sends a computation request, `SeD_batch` wraps this request in a helper script, pointing to the executable corresponding to the called service, and submits it as a job to the batch scheduler.

`SeD_cloud` works similarly differing in that fact that it interacts with cloud managers [27]. During discovery, estimation vectors are filled with information such as price and availability, retrieved from the cloud manager. When a request is received, `SeD_cloud` instantiates a VM with the virtual image corresponding to the called service, executes the service inside the VM and terminates the VM upon service completion.

This architecture has several advantages. First, it allows the user to easily interact with several resources without changing the underlying platform. Second, features already provided by the LRMS, such as managing job queues, mapping jobs on parallel resources, deploying VMs, etc., do not have to be duplicated in DIET.

On the downside, information related to resources is limited to what the LRMSs provide. Indeed, we have seen above that `SeD_batch` has to duplicate the behaviour of the LRMS to retrieve the information it is interested in. Moreover, LRMSs may accept local resource requests without notifying the meta-scheduler. This changes the state of the resources and might lead to the meta-scheduler taking suboptimal scheduling decisions.

DIRAC

DIRAC [31] is a grid meta-scheduler which is widely used for processing data produced by the Large Hadron Collider. It has been specifically designed for the needs of the LHCb community, which mostly runs PSAs. The problem with traditional LRMSs is that they impose a "push" scheduling, in which applications submit jobs to resources. Due to the large amount of tasks that PSAs contain, sending all tasks at once would overload the system. On the other hand, sending insufficient tasks would underutilize resources. Therefore, "pull" scheduling is desirable, in which resources (once they become available) contact applications to request additional tasks,

similar to the way Voluntary computing middleware is implemented [4].

To solve the above issue, DIRAC proposes a solution based on the pilot job abstraction.

pilot jobs **Pilot jobs** are container jobs that, instead of executing an actual task, launch a daemon. Once the job starts and the daemon is running, the latter connects to a centralized agent and requests tasks to execute.

From the meta-scheduler's point-of-view, pilot jobs improve scheduling decisions due to two reasons. First, access to resources is done the same way, no matter what interface the LRMSs offer.

late binding The meta-scheduler only has to deal with a set of daemons connected to it. Second, **late binding** can be employed, i.e., instead of binding tasks to resources at submittal, the meta-scheduler can defer this decision up to the moment that the allocation actually starts.

Unfortunately, pilot jobs also have drawbacks. First, a fundamental problem is how to choose the number of the pilot jobs to submit. In practice, having insufficient information about the state of the resources, heuristics are used with parameters tuned based on trial-and-error [53]. Second, pilot jobs do not ensure fairness among several meta-schedulers. Since such a middleware generally runs pilot jobs for multiple users under the same local user ID, the LRMSs do not have enough information for improving fairness.

KOALA

KOALA is a grid meta-scheduler that has been developed at the Delft University of Technology. It is extensively used on the Dutch research Grid DAS-3 [145], which is a multi-cluster platform. Each cluster is managed independently by the Sun Grid Engine [51] batch scheduler, thus, KOALA extends scheduler's functionality without requiring a change in the underlying platform. The remaining of this section presents and analyzes two features of KOALA which are most interesting for this Thesis: co-allocation support and malleable applications support.

Co-allocation Support KOALA takes a request from a user, which specifies how many nodes to reserve on each cluster [83]. Then, it is the task of KOALA to allocate resources and launch the application once the nodes on all clusters are simultaneously available. Without support for advance reservations, the process of ensuring this is quite intricate. KOALA submits jobs to each batch scheduler, with the duration of the allocation (i.e., the maximum execution time) artificially increased in order to compensate for the skew between the start-times. To determine by how much the duration of the jobs need to be increased, queue prediction algorithms are employed. Finally, if the queue changed too much and the co-allocation cannot be satisfied with the current jobs in the system, KOALA cancels all job submissions and retries.

While solving the co-allocation problem in a system with unsynchronised queues, this approach has several inconveniences. First, KOALA needs to over-allocate resources in order to compensate for the skew, which wastes resources. Second, there is no guarantee that the process actually succeeds. Third, the behaviour of the LRMS is duplicated through the queue prediction algorithms. If one changes the scheduling algorithm of the LRMS, KOALA also needs updating.

Malleable Applications Support KOALA can also be used to add malleable applications support over an LRMS which lacks such support [16]. On the user side, the application specifies the minimum node-count and receives grow and shrink messages from the scheduler. On the administrator side, several scheduling strategies are proposed. If resources become available, they can either be used to grow malleable applications that are already running (precedence to running applications) or to launch queued applications (precedence to waiting applications). Among

malleable applications, resources can either be used to grow the first malleable application, or can be distributed equally (equi-grow and shrink).

At the time of this writing, there is no LRMS which natively supports malleable applications. To work around this limitation, KOALA submits 1-node jobs and closely monitors the availability of the resources, as given by the LRMS, submitting new jobs or killing existing jobs as needed.

The support for malleable applications can be used to efficiently support PSA. Without such support, PSA can be executed by submitting each task as a separate rigid job. However, due to the relatively short duration of a task and the large amount of tasks, this adds a lot of overhead [102]. Therefore, KOALA proposes scheduling PSAs as malleable application. This is similar to the pilot job concept presented above, except that the resources allocated to an application do not have a fixed size, but increase/decrease as decided by the scheduler. On the application side, growing is handled by spawning new tasks, while shrinking is handled by killing existing tasks. Using equi-grow and shrink ensures a kind of fairness: Each PSAs gets the same proportion of the platform.

There are two issue related to KOALA's malleable application support that should be highlighted. First, workarounds are necessary to simulated malleability on top of a non-supporting LRMS. Having to submit many 1-node jobs is inefficient and might overload the LRMSs. Second, this support cannot be used for evolving applications. Indeed, for evolving applications it is their internal computation that requires the allocation of resources to grow and shrink, whereas in KOALA it is the resource manager that takes this decision.

Redundant Requests and Job Migration

To further highlight the workarounds that meta-schedulers have to implement, let us present two more scheduling schemes which are used to load-balance applications across multiple clusters. They are kept in a separate section, since, to our knowledge, they are not employed in any real meta-scheduler. Nevertheless, the solutions they propose are interesting from a resource management point-of-view.

First, let us reiterate the issue. As discussed before, when submitting jobs to batch schedulers, users have to give an upper bound on the execution time of their applications. If an application finishes earlier, the next application in the queue will be started immediately. This makes predicting the start-time of a job very difficult, as it depends on the correctness of the estimations of all previously queued applications. Therefore, when having access to multiple clusters, managed by independent batch schedulers, it is difficult to choose the cluster to which a job should be submitted: If one cannot predict the application's start-time, the more difficult it is to predict the end-time. Two solutions are often mentioned in literature: redundant requests and job migration.

redundant requests

Redundant requests are jobs which are submitted simultaneously to the K least loaded batch schedulers (possibly all) [107]. When the first job starts on any of these clusters, the jobs on the other $K-1$ clusters are cancelled. By minimizing the start-time of the application, the end-time is also reduced. However, redundant requests are harmful as they worsen even more the estimated start times and create unfairness towards applications which cannot use them, by hindering back-filling opportunities [32]. In theory, redundant requests could also be used to emulate moldable jobs, however their impact has not been studied yet.

job migration

An alternate solution is to **migrate jobs**. First, the meta-scheduler submits jobs to a cluster chosen according to the state of the queues at that time (à la DIET, see beginning of this section). Then, the meta-scheduler regularly wakes up, retrieves the state of the queues and migrates jobs

if it estimates that the end-times can be improved. Migration can be achieved by cancelling the job on one cluster and resubmitting it to another. Job migration has been shown to improve application response-times [19]. Unfortunately, this solution is difficult to apply in practice due to two reasons. First, LRMS administrators are reluctant to giving too much power to a meta-scheduler, as a result, locally queued jobs are not even exposed to the meta-scheduler. Second, assuming administrators would trust meta-schedulers enough, if migration is done while a local job is submitted, then the migrated jobs are placed at the end of the queue. Therefore, their end-time might actually be increased, which would defeat the very purpose of job migration.

Analysis

This section presented and analyzed several meta-schedulers, whose primary purpose is to extend functionality provided by the LRMS and better adapt resource management to a certain type of application. The analysis revealed that achieving this is inefficient: LRMS expose a too simplistic interface, which forces meta-schedulers to resort to workarounds.

When looking from the application's point-of-view, meta-schedulers attempt to optimize the performance across a wide range of applications, but do not take into account the specificities of a particular application. For example, an application cannot use knowledge about its performance model to optimize resource selection.

2.3.5 Application-Level Schedulers

application-level scheduler

An **Application-Level Scheduler (ALS)** is an agent which is responsible for scheduling one application or one type of application. Its purpose is to interact with lower-level resource managers, such as meta-schedulers or batch schedulers, in order to extend their functionality as required by a specific type of application. In contrast to meta-schedulers, an instance of an ALS only handles one single instance of an application, thus, they are deployed and configured by the user. ALS are employed for several reasons:

- improving support for moldable applications: AppLeS;
- simulating malleability: malleability over low-priority jobs;
- improving support for evolving applications: GridARM.

Let us present and analyze each of them separately.

AppLeS

The AppLeS project [9], whose name itself comes from **App**lication-**Le**vel **S**chedulers, delivered a framework which can be used to create specialized ALSs in order to optimize resource selection. The argument is that, since such an ALS has application-specific knowledge, such as the structure of the application, performance-prediction models, etc., application performance can be substantially improved.

Among others, the AppLeS project proposed and implemented a moldable application scheduler called "SA" [37]. It is designed to optimize application performance on batch schedulers which only support rigid jobs. SA works as follows: First, the user launches the application with SA, specifying a list of node-counts and maximum execution times, similarly to moldable configurations supported by TORQUE and OAR. Then, SA retrieves the state of the batch queue and submits a job which minimizes the response time of the application, as computed based on the user estimations.

submit-time moldability

A limitation of SA is that it does **submit-time moldability**, i.e., the choice of the job

size is made at submittal. This has been shown to be inefficient [105, 106] when compared to **schedule-time moldability**, i.e., deferring the choice of the job size until it starts. However, to be usable in practice, this either requires moldable job support or the batch scheduler to actively involve the application in its scheduling decisions. The former is impractical as has been shown in Section 2.3.2, as of the latter, there is currently no resource manager which allows this.

schedule-time moldability

Malleability over Low-priority Jobs

An ALS can be used to support malleability on top of OAR's low-priority, preemptible jobs (see Section 2.3.2) [34]. First, a normal job J_{min} of node-count n_{min} is submitted in order to satisfy the application's minimum resource requirements. Then, when J_{min} starts, the ALS asks OAR for the number of currently free nodes n_{free} and submits a preemptible job J_{extra} having the node-count n_{free}. The application initially runs on $n_{min} + n_{free}$ nodes. When a job is submitted to the system, which requires the preemptible job's resources, OAR sends a custom signal to one of the application processes and kills J_{extra} after a grace-period. This allows the application enough time to transfer data from J_{extra} and continue its computations on J_{min}.

The presented ALS is indeed improving resource usage and application response time, by allocating resources that are otherwise unused. However, the approach only achieves a limited form of malleability and suffers from two limitations. First, the preemptible job J_{extra} is killed even if only part of the resources allocated to it are needed. Ideally, J_{extra} should be resized as required, so that the application can continue running on a maximum number of nodes. Second, when reasoning about multiple malleable applications in the system, since OAR is ignorant to the purpose of the preemptible jobs, only the first application can acquire the free resources on the platform. It is not possible to fairly share resources among multiple malleable applications, for example, as proposed in KOALA.

GridARM

GridARM [35] is a framework for scheduling AMR applications on resources such as Grids. The challenge is to execute the application efficiently, knowing that both the state of application and the state of the resources are dynamic.

GridARM works as follows. The AMR application is split in natural regions (NR), the smallest possible computation unit. Several NRs are mapped to a virtual computation unit (VCU). The active VCUs (also called the working set) are mapped onto Virtual Resource Units (VRUs) using the vGrid middleware [71]. VRUs are an abstraction for individual Grid resources, such as hosts or clusters. To optimize the execution of the application, a complex autonomic control loop is used. It takes into account both application and resource state, so as to dynamically allocated resources and change the NRs to VCUs to VRUs mapping. According to our classification, the resulting application is both evolving and malleable.

Unfortunately, the cited articles aim at offering an efficient application framework and do not address two fundamental resource management issues. First, how resources are co-allocated over several clusters is not discussed. Second, it is not clear how the framework behaves if the platform is highly loaded and the application needs to grow. If the framework aborts, then all computation done so far is lost. If the framework waits for resources to become available, then a dead-lock might occur if two such applications simultaneously run on the platform.

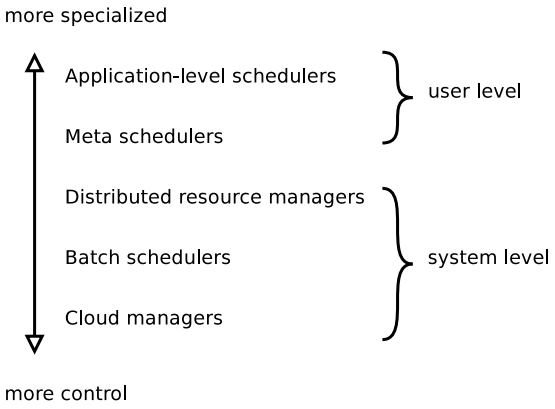


Figure 2.8: A resource management taxonomy by the control one has over scheduling decisions

2.3.6 Analysis

Figure 2.8 classifies the previously presented resource management systems and shows that resource management is done on several levels: system and user. The system level attempts to offer uniform abstractions, so as to simplify access to resources. Software operating at this level has direct control over the resources, thus it has the most flexibility with respect to scheduling decisions. However, the exported interfaces have to be very general, so as to fulfill the needs of a wide range of users. This hinders it from implementing application-specific optimizations.

At the user level, the software aim to offer resource management systems, which are specialized for a particular application or a certain class of applications. Decisions taken at this level are tuned to optimize the performance of the applications. However, since they do not directly control the underlying resources, they have to use the interfaces provided by system-level software. Therefore, the information they can gather to take scheduling decisions and the resulting actions are limited by the abstractions offered by system-level software.

Two things are especially concerning. First, there is a considerable overlap between the decisions taken by user-level and system-level agents. Second, since system-level agents export an insufficient interface, user-level agents need to find workarounds in order to take action for their decisions.

2.4 Conclusion

This chapter presented the context of the Thesis. First, it presented the currently used HPC resources and highlighted that they are becoming increasingly complex. Exploiting them at peak performance requires resource allocation to be carefully done. Next, applications running on these resources have been classified according to their resource usage. It has been highlighted that resource management can be improved if moldable, malleable and evolving applications are properly supported. Finally, the state of the art of resource management has been presented and it has been shown that these types of applications are not efficiently dealt with.

Part II

RMS Support for Moldable Applications

CHAPTER 3

CooRMv1: An RMS for Efficiently Supporting Moldable Applications

> Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.
>
> Antoine de Saint-Exupéry

This chapter addresses the first issue of the Thesis: efficiently scheduling moldable applications. A centralized Resource Management System (RMS) architecture called CooRM*v1 is presented, which actively involves applications in scheduling decisions, thus allowing them to optimize their resource selection. The architecture is evaluated using a simulator which is then validated using a proof-of-concept implementation on Grid'5000. Results show that such a system is feasible and performs well with respect to scalability and fairness. Concepts presented in this chapter can then be extended to efficiently support malleable and evolving applications, as shown in Chapter 6.*

3.1 Why Moldability?

As more and more peta-scale computing resources are built and exa-scale is to be achieved by 2020 [41], optimizing resource allocation to applications is critical to ensure their efficient execution. For example, supercomputers feature non-homogeneous networks, having topologies such as a torus or a fat-tree. Tightly-coupled parallel applications might need their allocated nodes to be grouped in order to execute efficiently [89]. This issue is even more aggravated in multi-cluster systems, as commonly found in modern data centers, Clouds or Grids: the inter-cluster network features high latencies compared to the intra-cluster network, that might significantly slow applications down [30].

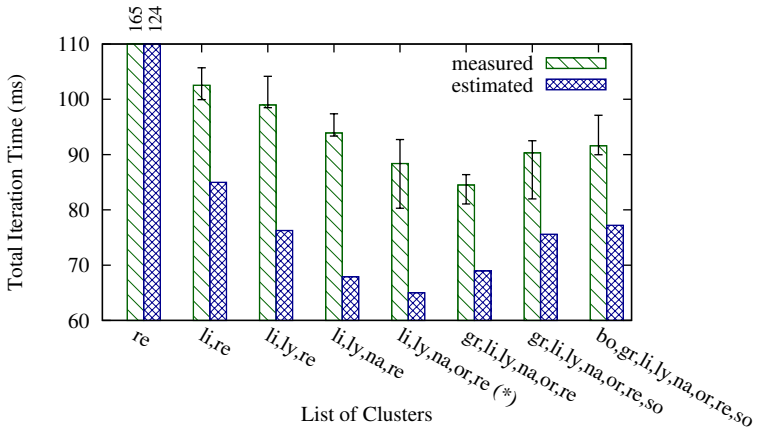
A first step in improving the way resources are allocated to applications is to consider them moldable, so to say, one may chose the resource they execute on, but once started, the allocation cannot be changed. Section 2.2 showed that application response-time can be reduced if application moldability is taken into account. Unfortunately, state of the art RMSs do not efficiently support such applications. Let us briefly remind the proposed solutions and their drawbacks.

Submit-time moldability Selecting a moldable configuration at submittal is inefficient [105], as the state of the system might greatly vary from the time the configuration has been chosen until the application starts. Also, in order to implement submit-time moldability, literature proposes [37] to retrieve the list of jobs from the RMS and simulating its scheduling algorithm. This not only duplicates functionality, but also forces applications to be developed in lock-step with the RMS: every time the scheduling algorithm of the RMS is changed, applications need to be upgraded, too.

Schedule-time moldability Selecting a moldable configuration before the application starts is more efficient than submit-time moldability [105]. The only implementation proposed in literature are **moldable jobs**, which allow an application to enumerate to the RMS several moldable configurations. However, this is only practical if the number of configurations is small. Otherwise, the solution might either overload the RMS or be altogether impractical. As shown in Section 2.3.2, this is problematic both for supercomputers and multi-cluster systems. For the former, the number of configurations is large, while for the latter the number of configurations is exponential.

To solve the problem of efficiently supporting moldable applications, this chapter proposes and evaluates a new RMS architecture called CooRMv1, that implements submit-time moldability by delegating resource selection to applications (more precisely their launchers). The proposed RMS assumes centralized control, targeting any system where such a control can be enforced—such as supercomputing centers, enterprise Grids or High-Performance Computing (HPC) Clouds. Our approach is especially suited for computation centers with multiple clusters or supercomputing centers where the interconnect can be approximated as homogeneous. Large scale platforms with multiple administrative domains such as Grids are outside the scope of this chapter.

The remaining of the chapter is organized as follows: Section 3.2 motivates the work by presenting a moldable application with a specialized resource selection algorithm. Section 3.3 lays out the problem statement, for which a solution is proposed in Section 3.4. Section 3.5 proposes an implementation, which is then evaluated in Section 3.6 both using a simulator and a proof-of-concept implementation. Section 3.7 analyzes some features of the proposition, while Section 3.8 concludes.



Bordeaux, **Gr**enoble, **Li**lle, **Ly**on, **Na**ncy, **Or**say, **Re**nnes, **So**phia

(*) Optimal solution found by our resource selection algorithm.

Figure 3.1: Performance of a CEM application for various cluster sets of Grid'5000

3.2 A Motivating Example: Optimizing Resource Selection for a Computational Electromagnetics Applications

In this section, we motivate our research by giving an example of an application which requires a specialized resource selection algorithm.

Computational ElectroMagnetics (CEM) is a Finite-Element Method (FEM) which provides solutions to many problems, such as antenna performance, electromagnetic compatibility and radar cross section. FEM works on a discretization of space (i.e., a mesh) over which it applies an iterative algorithm. Increasing the precision of the result is done by using a more refined mesh, which in turn increases the computation-time and the required amount of memory.

In order to take advantage of the latest hardware architectures and improve response-time, the CEM application needs to be parallelized. This is commonly done by partitioning the input mesh into submeshes and mapping them onto target cores. The resulting problem is non-trivially parallelisable as resulting submeshes need to frequently exchange data.

Such applications are usually run on a single cluster. Indeed, since data exchanges happen frequently, a high-speed, low-latency Local-Area Network (LAN) would ensure a good performance of the application. However, the increasing need for more precision, which requires more computing power and more memory, is pushing towards a multi-cluster execution. The application's scalability on such architectures might be problematic, as the higher latencies observed on an inter-cluster Wide-Area Network (WAN) might potentially slow the application down.

In order to study whether a multi-cluster execution of CEM applications is beneficial, the MAXDG1 application has been ported to allow multi-cluster execution [127]. The resulting application uses TCP to efficiently couple MPI codes running on multiple clusters. According to our classification, the application is moldable: one may choose what resources it should execute on, but once started the allocated resources cannot change. Before executing it, one must choose for each cluster the number of nodes the application should run on, thus giving rise to an exponential number of configurations.

Initial experiments (as exemplified in Figure 3.1) showed that multi-cluster execution of the studied application can indeed lead to a smaller execution time, provided the resources are carefully chosen. Therefore, in order to ensure that the application is always executed at peak

performance, an analytical performance model has been devised for it [30]: the function p takes as input cluster metrics, such as node computing power, intra-cluster LAN network latency and bandwidth, as well as the inter-cluster WAN latency and bandwidth, and outputs an execution time.

To optimize resource selection for the CEM application, we need to determine the input that minimizes p. Since the search space is exponential, a heuristic based on a greedy approach and simulated annealing has been devised. The algorithm f takes as input cluster and inter-cluster information (i.e., the set of **available resources**) and estimates the **minimum execution time** and the resources for which is minimum is obtained (i.e., the set of **selected resources**). The performance model and the algorithm manage to reasonably find the minimum execution time of the application (Figure 3.1), requiring up to 1 seconds for 12 clusters and up to 16 seconds for 40 clusters.

Unfortunately, using such a resource selection algorithm in practice is difficult, as has already been highlighted in Section 3.1. Using submit-time moldability and calling f based on information acquired from the RMS is inefficient, first, because the scheduling algorithm of the RMS would have to be duplicated and, second, because the state of the system might change from the moment a configuration is chosen until the application is actually executed. Using schedule-time moldability implemented using moldable jobs, one would have to use p and enumerate all possible moldable configurations, which is impractical.

3.3 Problem Statement

Seeing that application-specific resource selection algorithms can improve performance, this chapter focuses on the following question: **What is the interface that an RMS should provide, to allow each moldable application to employ its specialized resource selection algorithm?**

How such resource selection algorithms should be written is application-specific and outside the scope of this Thesis. However, if a clean and simple RMS interface exists, developers could provide application-specific launchers (which implement a resource selection algorithm) with their applications, so as to improve the response time experienced by the end users. As an example, the above CEM application gives an idea of the effort and benefits of developing such a launcher.

Since these selection algorithms might take some time, we are especially concerned with two issues:

Scalability The selection algorithm should be called only when necessary, e.g., exhaustively iterating over the whole resource space is not practical. Since the result of the selection only depends on the available resources and not on the application's internal state (remember that the applications dealt with are considered moldable), *memoization*[1] can be used. Therefore, the system needs to reduce the number of unique inputs for which the selection algorithm is invoked, a metric that we shall call the number of **computed configurations**.

Fairness In rigid job scheduling, applications are considered to have an implicit priority based on their arrival time. Intuitively, if two application A and B arrive in the system in this order, a fair scheduler should not allow the application B to delay (i.e., increase the end-time) of application A [76, 118].

[1] NIST Definition [122]: save (memoize) a computed answer for possible later reuse, rather than recomputing the answer.

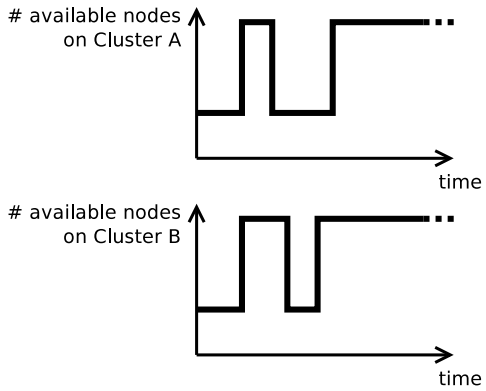


Figure 3.2: Example of a view sent by CooRMv1 to an application

For applications with intelligent resource selection, a new fairness issue arises. An application A with a lengthy selection (e.g., 10 seconds) should not be delayed (i.e., its end-time is increased) by an application B with a quicker selection (e.g., 1 second) submitted just after A. Since A has been submitted before, A should have a higher priority, therefore its resource selection should not be impacted by applications submitted after it.

Thus, the RMS should make sure that applications are not delayed, as long as they have *reasonable* lengthy resource selection algorithms, i.e., the duration of the resource selection algorithm should be negligible compared to the execution time of the applications.

The next section presents an RMS architecture that solves the above issues.

3.4 The CooRMv1 Architecture

This section introduces CooRMv1, an RMS architecture which delegates resource selection to applications. First, the rationale of the architecture is given. Then, interactions between applications and the RMS are detailed.

3.4.1 Principles

CooRMv1 is inspired by a batch-like approach, where the RMS launches one application after the other as resources become available. Batch schedulers work by periodically running an algorithm which loops through the list of applications and computes for each one a start-time based on their resource requests.

Similarly, in CooRMv1 each application i sends one **request**[2] $r^{(i)}$, containing the resources (e.g., the number of nodes) and the maximum duration for which the allocation should take place. The RMS shall eventually allocate resources to the application, guaranteeing exclusive, non-preempted access for the given duration. Requests never fail, their start-time is arbitrarily delayed until their allocation can be fulfilled. Abusing this concept, if an application requests more resources than exist on the platform, the start-time is set to infinity. Regarding the end-time, the application is allowed to terminate earlier, however, the RMS shall kill an application exceeding its allocation.

request

[2]Since the application is moldable and cannot change its resource allocation during execution, one request is sufficient to express a constant resource allocation.

For choosing a request $r^{(i)}$, moldable applications need information regarding the availability of the resources. This allows them, for example, to choose a request $r^{(i)}$ which minimizes their completion time. To this end, CooRMv1 presents each application i a **view** $V^{(i)}$, which stores the estimated availability of resources as a function of time (see example in Figure 3.2). Some resources might not be available during a certain period, either because of the request of another, higher-priority application, or because of policy-specific decisions (e.g., applications may not run overnight). The views represent the currently available information which aids applications in optimizing their requests. Views and requests allow to implement submit-time moldability, without having to duplicate or simulate the behaviour of the RMS.

view

For schedule-time moldability, one more issue remains. When the state of the system changes, the requests $r^{(i)}$ that the applications have previously computed might become sub-optimal. For example, if an application A finished earlier than estimated, another application B, that has not yet started, might want to update its request r^B in order to take advantage of the newly freed resources. Therefore, until its start, each application i receives an up-to-date view $V^{(i)}$, so that it can send an updated request $r^{(i)}$ to the RMS. In contrast to batch schedulers, updating a request does not make the application lose its implicit priority based on its arrival time (see Section 2.3.2).

When a system change impacts multiple application, updating requests can be done in one of two ways: either sequentially or in parallel. For sequential updates, the RMS first deals with the highest-priority application A by sending it an updated view V_u^A and waiting for an updated request r_u^A, before dealing with the next, lower-priority application. For parallel updates, the RMS asynchronously sends updated views to all applications simultaneously and then waits for updated requests. Furthermore, if the probability of an application changing its request is small, the RMS needs not wait for its answer and may simply assume that no answer means no request update. We chose this approach, since it has the potential to reduce the number of messages exchanged between the RMS and the applications.

To sum up, the above approach can be considered a dynamic, distributed scheduling algorithm. The RMS is responsible for applying a **policy**, which decides how to multiplex resources among applications, while the **selection** of resources, which decides how to optimize the application's structure to the available resources, is handled by the applications themselves.

policy

selection

3.4.2 Data Types

In the rest of this chapter, we apply the above principles to a multi-cluster system. We assume that the computing nodes inside a cluster are equivalent, which is true if the network can be assumed to be homogeneous. This allows us to make a few optimization, such as working with number of nodes, instead of having to individually specify node identifiers (IDs).

Before describing the interactions that take place in the system, let us first define some data types:

- `FILTER` is a Job Submission Description Language (JSDL)-like [6] filter to select candidate clusters. It specifies the minimum number of nodes, per-node Random-Access Memory (RAM), total RAM, scratch space, etc.
- `CID` (cluster ID) uniquely identifies a cluster.
- `CINFO` (cluster info) stores the cluster's properties, e.g., the number of nodes, the number of CPU cores per nodes, presence of accelerators, size of RAM, size of scratch space, LAN characteristics, etc. Information that should be contained in this structure has already been discussed in the GLUE schema [5].
- `ICINFO` (inter-cluster info) stores information about the interconnection of one or more

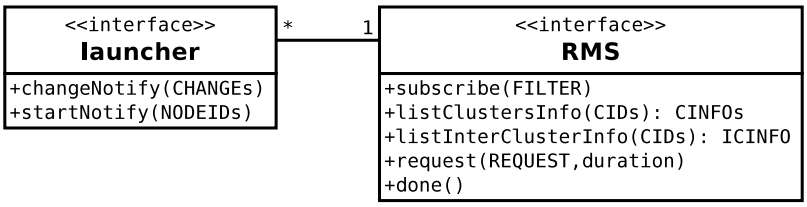


Figure 3.3: Application callbacks and RMS interface in CooRMv1

clusters, e.g., WAN network topology, bandwidth and latency. Information that should be contained in this structure has already been discussed in [74].
- `CAP` (cluster availability profile) is a concretization of the **view** concept for a multi-cluster system. For a given cluster, it represents a step function which stores the number of available computing nodes as a function of time, as exemplified in Figure 3.2.
- `REQUEST` is concretization of the **request** concept for a multi-cluster system. It contains the number of nodes to allocate on each cluster and the duration of the allocation (e.g., 4 nodes on cluster A, 5 nodes on cluster B for 2 hours).
- `NODEID` uniquely identifies a node, e.g., by specifying its fully-qualified domain name.
- `CHANGE` represents a change event for a cluster. It is composed of the tuple

$$\{\texttt{CID}, \texttt{type}, \texttt{CAP}\}$$

  where `CID` is the cluster ID for which some information changed and `type` specifies whether cluster information (`CINFO`), inter-cluster information (`ICINFO`) or the availability (`CAP`) has changed. In the latter case, the new `CAP` is included in the message.
- Plurals are used to denote "set of" (e.g., `CIDs` means "set of `CID`").

3.4.3 Interfaces

In CooRMv1, the system consists of one or more applications, their launchers, which contain the application-specific resource selection algorithm, and the RMS. Since the interactions between the launcher and the application itself are programming-model dependent and do not involve the RMS, we shall focus on the interactions between the launcher and the RMS.

Before starting the actual computations, an application negotiates the resources it will run on through its launcher, which could run, either collocated with the RMS on a front-end, or on a distinct node (e.g., the user's computer). Figure 3.3 presents the interfaces that are exposed by the RMS and the application launcher. The launcher exposes an interface providing two callbacks:
- `changeNotify`, allowing it to receive dynamic information about the resources, e.g., the cluster availability profile (`CAP`);
- `startNotify`, which notifies the application that the resource allocation has started, also specifying what node IDs it can deploy on.

The RMS exposes methods allowing the launcher:
- to subscribe to resource availability (`subscribe`), optionally specifying a filter, so as to reduce the amount of information the launcher receives;
- to retrieve static information about clusters (`listClustersInfo`) and the WAN that interconnects them (`listInterClusterInfo`);
- to update the resource request (`request`);
- to terminate a resource allocation (`done`).

Let us now see how this interface can be used for efficiently negotiating resources.

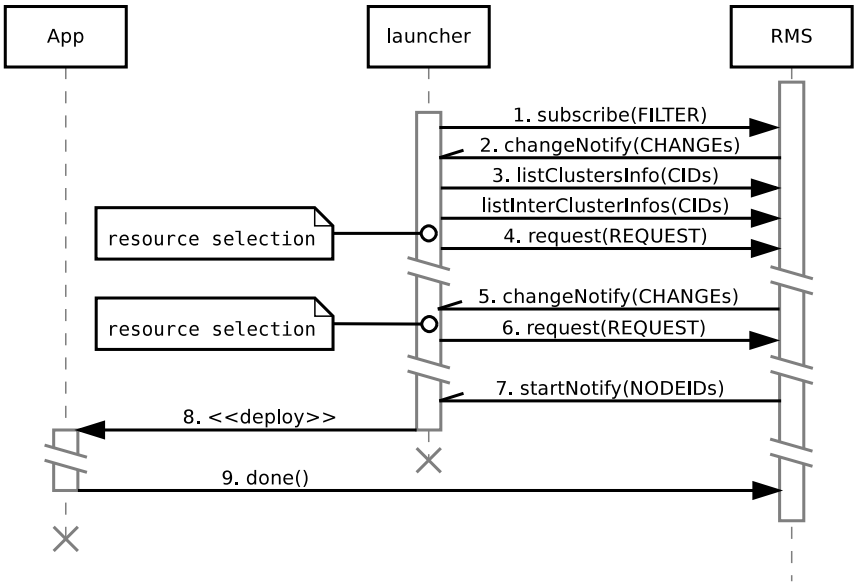


Figure 3.4: Example of interactions between an RMS, an application and its launcher

3.4.4 Protocol

Figure 3.4 presents an example of a typical interaction between a single application (through its launcher) and the RMS:

1. The launcher `subscribe`s to the resources it is interested in. Depending on the input of the application, the launcher might use the `FILTER` to eliminate unfit resources like nodes with too little memory or unsupported architectures. This allows an application to reduce the amount of notifications it receives, also reducing the load on the RMS.
2. The RMS registers the application in its database and sends a `changeNotify` message with the relevant clusters and dynamic information about them, i.e., their `CAPs`. The launcher uses this data to update its local view of the resources.
3. Since the launcher has no previous knowledge about the clusters, it has to pull static information about them, i.e., it has to retrieve the associated `CINFOs` and `ICINFOs` by calling `listClustersInfo` and `listInterClusterInfo`, respectively.
4. The launcher executes the resource selection algorithm, computes a resource request and sends it to the RMS.
5. Until these resources become available and the application can start, the RMS keeps the application informed by sending `changeNotify` messages every time information regarding the resources or their estimated availability changes.
6. The launcher re-runs the selection and updates its request, if necessary.
7. When the requested resources become available, the RMS sends a `startNotify` message, containing the `NODEIDs` that the application may use.
8. The launcher deploys the application.
9. Finally, when the application has finished its computations, it informs the RMS that the resources have been freed by sending a `done` message.

For multiple applications, each launcher creates a separate communication session with the RMS. No communication occurs between the launchers. It is the task of the RMS to compute for each of them a view, so that the goals of the system are met.

3.5 An Example Implementation

The presented architecture gives an answer to the main problem raised in this chapter, that of offering an interface to allow each application to employ its own resource selection algorithm, as posed in Section 3.3. There are still some potential issues such as scalability and fairness. In order to evaluate their impact, we need an implementation for CooRMv1. This section proposes implementations both for application-side resource selections and for an RMS policy.

3.5.1 Application-side Resource Selections

In this section we give examples of several types of application launchers and describe how they interact with a CooRMv1 RMS to negotiate resources so as to minimize the completion-time of the application.

All launchers are structured similarly. They internally store a **local view** of the resources containing the most up-to-date view received from the RMS and the **last request** sent to the RMS. Initially, both the local view and the last request are empty. Then, the `changeNotify` handler reads the information sent by the RMS, updates the local view and uses a specific algorithm to compute a new request. If the new request is different from the last request, it is immediately sent to the RMS and the last request is updated.

Let us present these specific algorithms for three types of applications: rigid, simple-moldable and complex-moldable. For each of them, we give a definition, present a performance model and illustrate through an example the implementation of a selection algorithm. Rigid applications are treated as a particular case of simple-moldable applications, as a result we present the simple-moldable applications first.

Simple-moldable Applications

Definition In what follows, we call **simple-moldable** a moldable application to which all allocated nodes need to belong to the same cluster. This is commonly the case with legacy applications as they have been developed assuming that they will only execute on homogeneous resources. For example, the data is partitioned equally among computing nodes, without taking into account possible differences in computing power. As a result, the application cannot efficiently run on multiple clusters: faster nodes would have to idle, waiting for data from slower nodes to become available.

simple-moldable

An important thing to note is that, for simple-moldable applications, the number of moldable configurations is small. They can still be efficiently dealt with by exhaustively enumerating all configurations to an RMS with support for moldable jobs, such as OAR or TORQUE. As an example, assuming the resources are composed of c clusters, each having n nodes, then a simple-moldable application can run on $n \cdot c$ configurations: for each cluster, one has to enumerate a configuration comprised of 1 to n nodes. As opposed to multi-cluster applications the configuration count is only linear instead of exponential (see Section 2.3.2).

Model Let us consider a simple-moldable application model inspired by [20]. Such an application can be described by the minimum (n_{min}) / maximum (n_{max}) number of nodes, the

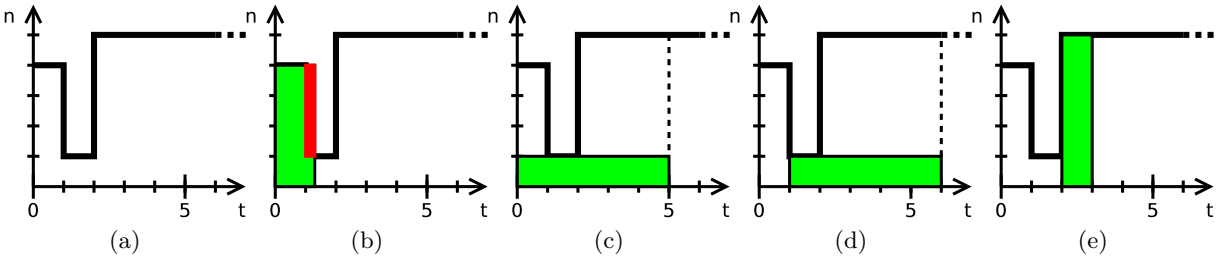


Figure 3.5: Scheduling example for a simple-moldable application

proportion of the program that can be parallelized ($P \in [0, 1]$) and the single-node duration on the i^{th} cluster ($d_1^{(i)}$). Given a cluster i and the number of nodes n, the execution time $d_n^{(i)}$ can be computed according to Amdahl's law:

$$d_n^{(i)} = (1 - P + P/n) \cdot d_1^{(i)} \tag{3.1}$$

Example Let us show by giving an example, inspired by [59], of an algorithm that minimizes the completion time. Assume the locally stored view is composed of a single cluster with 5 nodes having the `CAP` presented in Figure 3.5a and a simple-moldable application with $P = 1$, $n_{min} = 1$, $n_{max} = \infty$ and $d_1^{(1)} = 5$. For each cluster in the view, the launcher iterates through the list of steps (which start at 0, 1 and 2). For the first step at $t = 0$, there are 4 free nodes; however, these 4 nodes will not be available during the whole length of the computed execution time (Figure 3.5b). For the same step at $t = 0$, the launcher retries, with the minimum number of free nodes it has previously found (1 node) and obtains an end-time of 5 (Figure 3.5c). For the step at $t = 1$, it has 1 free node and obtains an end-time of 6. For the step at $t = 2$, there are 5 free nodes and the end-time is 3, which is the best that can be obtained. Thus, given the view in Figure 3.5a, the launcher chooses to request 5 nodes for a duration of 1.

Rigid Applications

Let us consider a single-cluster rigid application, which is characterized by a fixed number of nodes n and a maximum execution time $d_r^{(i)}$ for each cluster i it can run on. Scheduling such an application is done using the same simple-moldable application launcher, using the parameters $P = 1$, $d_1^{(i)} = d_r^{(i)} \cdot n$ and $n_{min} = n_{max} = n_r$.

Complex-moldable Applications

complex-moldable

Definition We define **complex-moldable** a moldable application which can run on multiple clusters simultaneously, such as the CEM application presented in Section 3.2. For such applications, the number of moldable configurations is exponential: if the resources are composed of c clusters, each having n nodes, for each cluster one can select a node-count from 0 to n, combined independently, excluding the configuration with no nodes. Therefore, the number of moldable configurations to enumerate is $(n + 1)^c - 1$. Clearly, it is impractical to schedule such applications using moldable jobs, therefore, specialized resource selection algorithms have to be used.

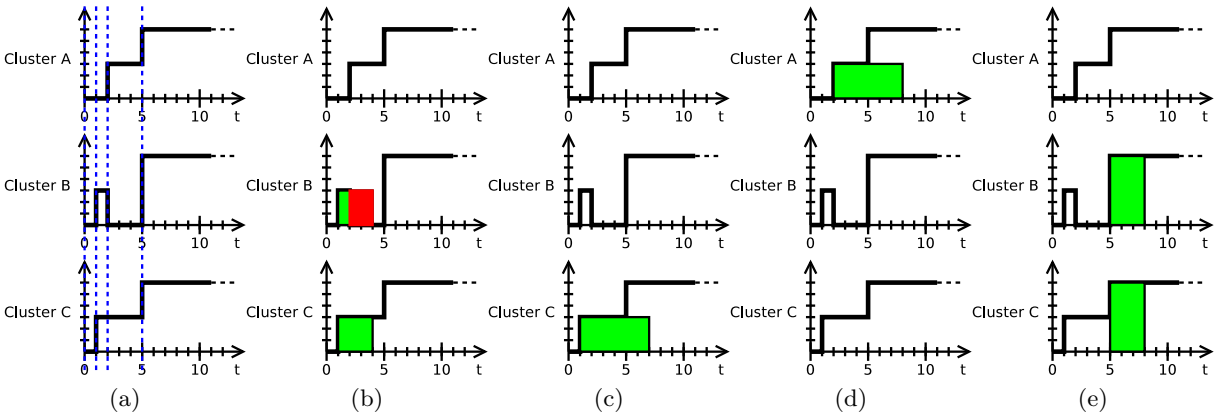


Figure 3.6: Scheduling example for a complex-moldable application

Model Resource selection for such an application can be done similarly to simple-moldable applications, except that the `CAPs` of all the clusters have to be simultaneously considered. Let us illustrate this through an example. Assume that we already have a function

$$f : a^{cid} \mapsto s^{cid}, d$$

which gets as input a^{cid} a mapping from each cluster (identified by its cluster ID) to the number of nodes *available* and outputs s^{cid} a mapping for each cluster to the number of nodes *selected* and d the estimated execution time (for the CEM application in Section 3.2 this function can be found in [30]).

Example Given the view in Figure 3.6a, the resource selection algorithm works as follows: First, the steps of all `CAPs` are identified, as illustrated with dashed lines. For the step at $t = 0$, no resources are available, so it is skipped. For the step at $t = 1$, f is called with the available resources (3 nodes on cluster B and 3 nodes on cluster C) and returns the same resources with a duration of 6 (Figure 3.6b). However, some resources are not available during this time interval. Therefore, this step is retried with the minimum number of free nodes previously found (Figure 3.6c), for which f returns a duration 6, thus giving an end-time of 7.

Next, for the step at $t = 2$, f is called with 3 nodes on cluster A and 3 nodes on cluster C, but for performance reasons it has chosen to select only the 3 nodes on cluster A. This might happen because the WAN latency between cluster A and cluster C is too high and cluster A is faster. Nevertheless, the found end-time is 8 which is worse than what was previously found. Finally, for the step at $t = 5$, f is called with 6 nodes on cluster A, 6 nodes on cluster B and 6 nodes on cluster C. Due to the lower WAN latency, selecting the nodes on cluster B and C lead to the lowest execution time. Nevertheless, the found end-time 8 still does not improve the previously found best.

In the end, the launcher requests the RMS 3 nodes from cluster C for a duration of 6.

3.5.2 A Simple RMS Implementation

This section presents an example of an RMS implementation. First, the behaviour of the RMS is described. Next, the scheduling algorithm that lies at the RMS's core, called policy, is presented. Finally, how the RMS deals with the fairness problem (highlighted in Section 3.3) is explained.

Behaviour When the RMS receives a `subscribe` message, it stores the arrival time $t_a^{(i)}$ of the application i in a database. Next, it sends the application a `changeNotify` message with the **last view**, which was computed by the last invocation of the policy.

When the RMS receives a `request` or a `done` message the database of current requests is updated and the policy is called, similarly to how rigid-job RMSs run their scheduling algorithm when a job is received or a job ends. A `request` message simply replaces the currently stored request of the application with the one to be found in the message, whereas a `done` message sets the end-time of the current request to the current time. After the policy has been executed, `changeNotify` messages are simultaneously[3] sent to the applications whose views have changed.

In order to coalesce messages coming from multiple applications at the same time and reduce system load, the policy is run at most once every **re-policy interval**, a parameter of the system. The choice of this parameter is briefly discussed in Section 3.6.

Policy For each application i, the policy takes as input its arrival-time $t_a^{(i)}$, its status (waiting or running) and its current request $r^{(i)}$. It computes its new view $V^{(i)}$ and its start-time $t_s^{(i)}$. An application is started when the computed start-time $t_s^{(i)}$ is equal to the current time: `NODEIDs` are allocated from the pool of free nodes and `startNotify` is sent to the corresponding application.

The internals of the policy are similar to First-Come First-Serve (FCFS) with repeated Conservative Back-Filling (CBF) [86], as detailed in Algorithm 3.1.

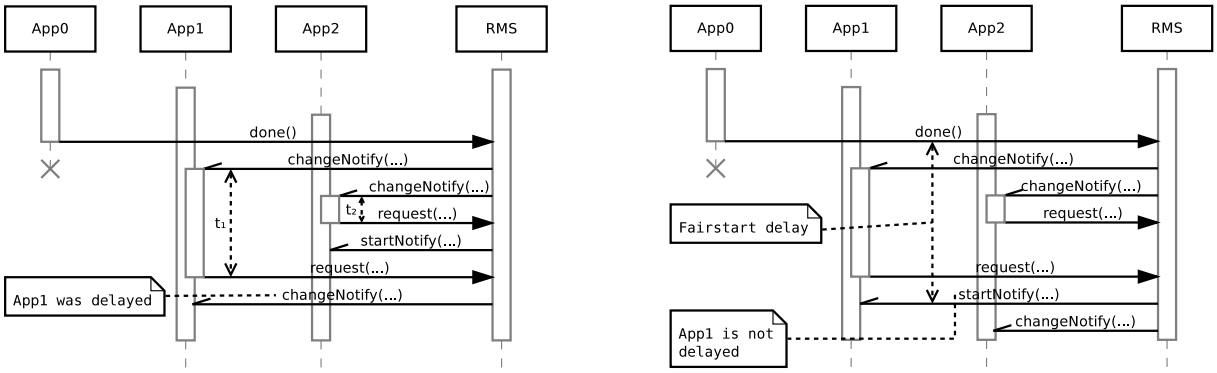
Algorithm 3.1: Example implementation of a CooRMv1 policy

1 For each cluster, a cluster availability profile is maintained, which stores the expected resource usage at future times. Initially, each profile represents that all nodes on the respective cluster are free for an infinite amount of time;
2 Subtract from these profiles the resources used by running applications;
3 **for** each application ordered by their arrival-time (as stored when the `subscribe` message was received) **do**
4 Set the view of the current application to the current profiles;
5 Find the first "hole", where its request fits and store the found start-time;
6 Update the profiles to reflect the allocation of resources to this application;
7 Store the current profiles, which at this step reflect the allocation of resources to all applications, as the last view.

Fairness Let us come back to the fairness issue (see the definition in Section 3.3) and explain how an implementation can easily solve it. Assume that three applications are in the system: `App0` is already running, while `App1` and `App2` have sent requests to the RMS and are waiting for the `startNotify` message. Also, let `App1`'s resource selection algorithm take d_1 seconds, while `App2`'s d_2 seconds, with $d_1 > d_2$. According to our goal of fairness, since `App1` arrived before `App2`, the latter should not be able to delay the former.

However, without any mechanism in place, `App1` could be delayed by `App2`. As shown in Figure 3.7a, when `App0` sends the `done` message, both applications want to take advantage of the newly freed resources. Since `App2` has a quicker resource selection than `App1`, without any mechanism in place, the RMS would launch it immediately. `App1`, which has a slower resource

[3] see Section 3.4.1 for rationale



(a) without fair-start delay

(b) with fair-start delay

Figure 3.7: Fairness issue for adaptable applications

selection, could not take advantage of these resources, despite the fact that it had priority over `App2`.

Therefore, in order to ensure fairness for applications with a lengthy resource selection, resources allocated to an application are not immediately released after a `done` message, but are artificially marked as occupied for a **fair-start** amount of time. Implementation-wise, this is achieved by setting the end-time of the request to the current time plus the fair-start interval. This additional delay allows high-priority applications with intelligent resource selection enough time to adapt and request the resources that will be freed after the fair-start delay expires (see Figure 3.7b). We have decided for the fair-start to be an administrator-tunable parameter, whose choice is discussed in the next section.

3.6 Evaluation

This section evaluates the scalability and fairness of CooRMv1. First, an overview of the experiments is given, then the simulation results are analyzed. Finally, the simulations are validated, by comparing the results with values obtained using a real implementation.

3.6.1 Overview

The ultimate purpose of CooRMv1 is to allow complex-moldable applications to employ their own resource selection algorithms, while ensuring scalability and fairness as defined in Section 3.3. In our experiments, the motivating multi-cluster CEM application is used as a complex-moldable application. However, it is unlikely that a platform will be reserved for running only applications with a complex structure. Therefore, the testing workloads also need to include simple-moldable applications submitted, as it is currently done, either as *rigid* jobs (the node-count is fixed by the user) or using a list of *moldable configurations*.

Comparison to an existing system is difficult, since there are no RMSs that offer exactly the same services. Thus, CooRMv1 is compared to an RMS that gives equivalent schedules, so as to focus on the overhead that CooRMv1 may bring. To this end, we have chosen OAR [22] both because it uses the CBF scheduling algorithm as well as due to its support for moldable jobs. In essence, each job is submitted with a list of node-count, maximum execution time pairs and OAR greedily chooses the configuration that minimizes the job's completion-time.

p	n_{min}	n_{max}	P
0.25	1	32	0.8
0.25	1	96	0.9
0.25	1	256	0.99
0.25	1	650	0.999

Table 3.1: Parameters used to reconstruct moldability from rigid job traces (inspired by [20]). p represents the probability that an application is generated with these parameters. n_{min}, n_{max}, P are the parameters presented in Section 3.5.1.

As previously explained, this is only practical for simple-moldable applications. For complex-moldable applications, the number of configurations to enumerate would be exponential in the number of clusters (see Section 3.5.1). Thus, CooRMv1 is evaluated in two ways: relative to an existing system in scenarios that could previously be supported, then, we use these results to analyze scenarios which could not be previously supported.

Simulator

To study the behavior of CooRMv1, we have written a discrete-event simulator in Python [140], using the greenlet framework [134] for co-routing support. Applications receive their views (Figure 3.2) and "instantly" (i.e., the simulation time is not advanced) send a new request aimed at minimizing their completion-time. The re-policy interval of the RMS has been set to 1 second since we want a very reactive system. The fair-start has been set to 5 seconds, a good starting value in order to allow the applications we target enough time to run their resource selection algorithms. This parameter will be further discussed in Section 3.6.3.

The simulator also includes a "mock" implementation of OAR, that we shall call OARsim. For OARsim, applications enumerate all the possible configurations and let the RMS choose which one to execute.

Resource Model

Resources are made of c clusters, each having $n = 128$ nodes. To add heterogeneity, the i^{th} cluster ($i \in [2, c]$) is considered $1 + 0.1 \times (i - 1)$ times faster than the 1^{st} cluster. The clusters are interconnected using a hierarchical model of a WAN, with latencies typically found over the Internet: two clusters are grouped in a "city" and every two "cities" in a "country". The latency between two clusters is 5 ms if they are in the same city, 10 ms if they are in the same country, and 50 ms otherwise, as can be typically found on the Internet.

Application Model

We generated two types of workloads. First, workloads W_0 include only "legacy" applications to compare CooRMv1 with OARsim. They were generated by taking packs of 200 consecutive jobs from the `LLNL-Atlas-2006-1.1-cln` trace from the parallel workload archive [129]. Since traces do not contain enough information to reconstruct the moldability of applications, we consider that 20% of the applications are simple-moldable, having an Amdahl's-law speed-up (see Table 3.1). The remaining 80% of the applications are considered rigid, with node-counts and execution-times found in the traces, subject to the speed-up of our resource model. The job arrival rate is set to 1 application per second, as the issues we are interested in appear when

Workload	Rigid	Simple	CEM
W_0	80.0%	20.0%	0.0%
W_1	79.6%	19.9%	0.5%
W_2	40.0%	10.0%	50.0%
W_3	0.0%	0.0%	100.0%

Table 3.2: Summary of workloads. Proportion of rigid, simple-moldable and complex-moldable (CEM) applications.

the system is under high load. During the experiments, we have observed that the maximum execution times provided with the original traces are mostly set to the system default. Since in CooRMv1 the launcher is aimed to choose a (more-or-less precise) maximum execution time, we set them to the execution-times multiplied by a uniform random number in $[1.1, 2]$.

Second, we generate workloads which include complex-moldable applications. Since the number of configurations to submit to OARsim is exponential in c (see Section 3.5.1), we only test CooRMv1 with these workloads. We present a subset of setups, so as to focus on the most interesting ones. Workloads W_1 have been generated starting from W_0 in which one instance of the CEM application has been inserted. These workloads act as a reasonable case, with a realistic mix of applications. Workloads W_2 have been generated starting from W_0 by replacing 50% of the jobs with instances of the CEM application. These workloads are the worst-case for CooRMv1 that we have found during all the experiments. Workloads W_3 contain 100% CEM application, to test the scalability in an extreme case, with only complex-moldable applications.

A summary of the workloads can be found in Table 3.2.

3.6.2 Scalability

We are interested in the following metrics: the number of **computed configurations** (defined in Section 3.3) and the **simulation time** (measured on a single-core AMD Opteron™ 250 at 2.4 GHz) which gives us the CPU-time consumed on the front-end to schedule all applications in the workload. For the case where the launchers and the RMS are run on separate hosts, we also need to measure the network traffic. Therefore, we measure the **total size of the messages**, by encoding CooRMv1 messages similarly to CORBA Common Data Representation (CDR) [55]. For each of these metrics, Figure 3.8 plots the minimum, maximum and quartiles.

For simple-moldable applications (W_0), the results show that CooRMv1 outperforms OARsim both regarding the number of computed configurations and the simulation time. We remind the reader that the obtained schedules are equivalent. For CooRMv1, more data needs to be transferred between the RMS and the applications, nevertheless, the total size of the messages is below 35 MB for 200 applications, in average 175 KB per application, which is a relatively low value for today's systems, especially when considering the size of the input / output data of a typical HPC application.

For complex-moldable applications, the results show that introducing one single CEM application does not significantly influence the metrics (W_1). Workloads W_2, which are the worst case, increase the values of the metrics, nevertheless, the scalability of the system holds. The network traffic is below 60 MB, in average 300 KB per application, a value which can be handled by today's systems. The time to schedule all applications is below 400 s, usually below 100 s, which is quite low, considering that it includes the time taken by the CEM resource selection algorithm. For the given resource model, computing a configuration may take up to 1 s.

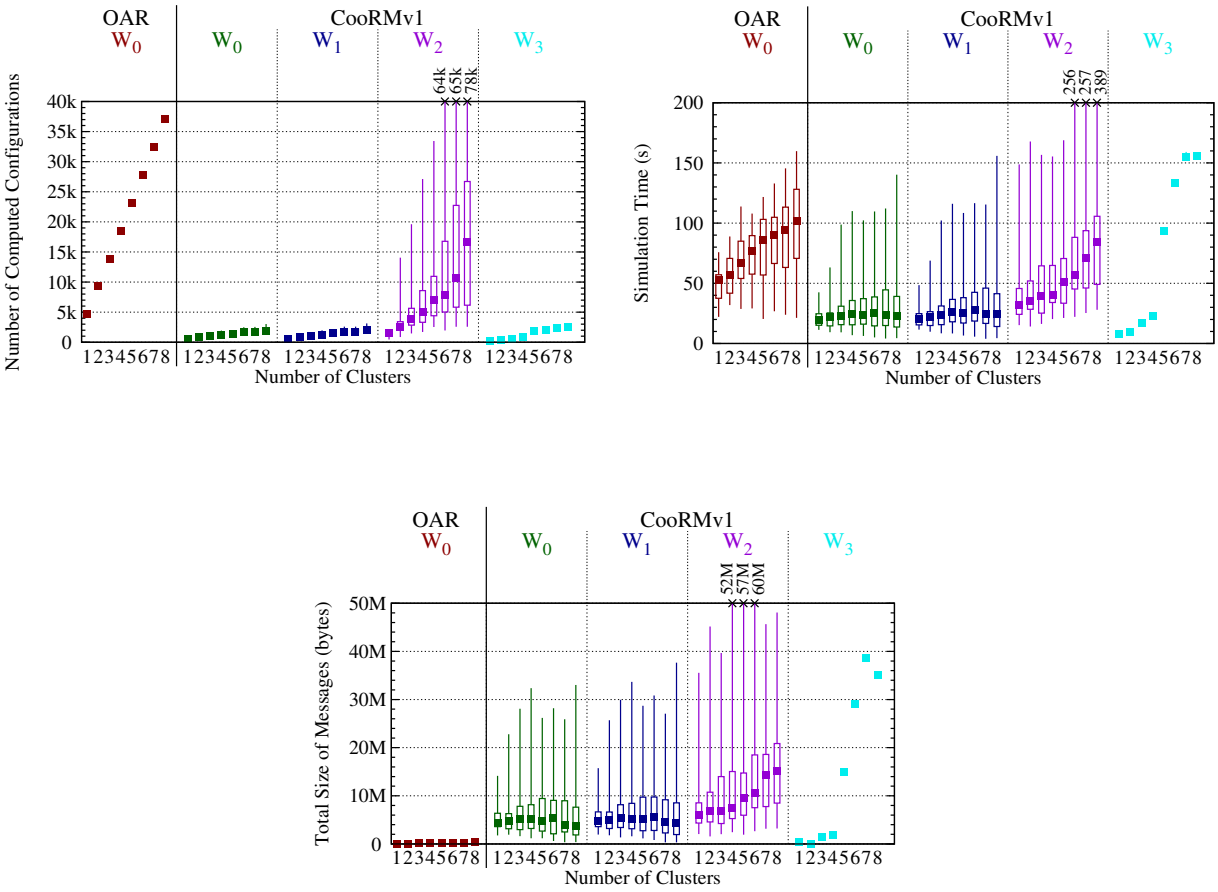


Figure 3.8: Simulation results for OAR (W_0) and CooRM (W_{0-3}) for 1 to 8 clusters.

For W_3, an extreme case where only CEM applications are present, the metrics are of the same order of magnitude as for the previous workloads. The number of computed configurations is smaller, due to the fact that the resource selection algorithm tries to select all the nodes from a cluster, thus applications are better "packed".

To sum up, results show that CooRMv1 scales well both for existing workloads and workloads in which the number of complex-moldable applications is large.

3.6.3 Fairness

Let us now discuss the importance of the fair-start parameter (see Section 3.5.2). To simulate applications with lengthy selections, we took workloads W_1 (see Section 3.6.1) and added to the CEM application an **adaptation delay**: when receiving a new view, the launcher waits for a timeout to expire, before sending a new request. In real-life, this delay represents the execution time of the resource selection algorithm. In the case of the CEM application, this algorithm may take up to 1 second for 12 clusters and up to 16 seconds for 40 clusters. Thus, we are interested in how much the application ends later compared to an "instant" response.

adaptation delay

Figure 3.9 shows the empirical cumulative distribution of the **end-time increase**, which is the end-time for the plotted adaptation delay minus the end-time when the adaptation-delay is zero, for an otherwise identical experiment instance. We note that the higher the adaptation

end-time increase

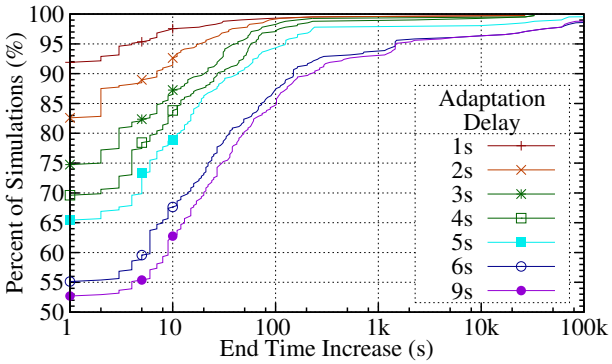
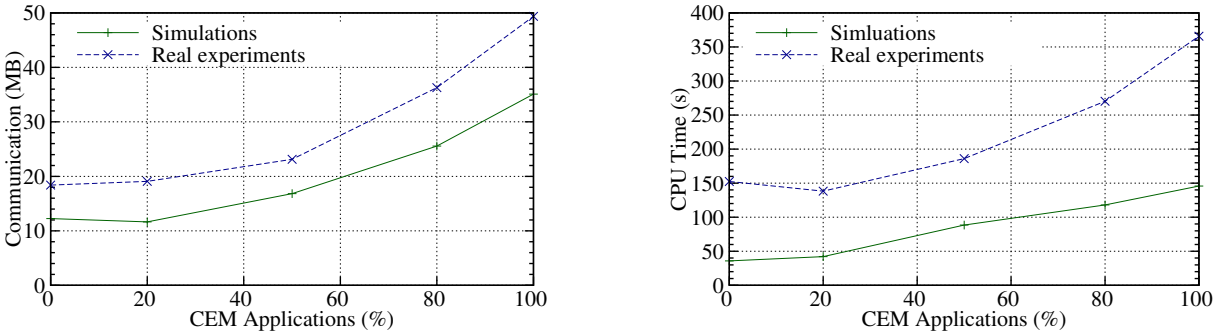


Figure 3.9: Unfairness caused by insufficient fair-start delay.



(a) Network Usage

(b) CPU Usage

Figure 3.10: Comparison between simulations and real experiments.

delay, the more the CEM application is delayed. In particular, when the adaptation-delay is higher than the fair-start delay, the application is significantly impacted. This happens due to the fact that another, lower-priority application, but with 0 adaptation delay, was allocated the resources that the CEM application could have taken advantage of. In up to 5% of the cases the application was delayed more than an hour. Therefore, the fair-start delay should be set to a high-enough value, so that complex-moldable applications have time to compute a new request.

Note that a high fair-start delay makes resources idle longer, thus increasing resource waste. Depending on the average job execution time, a high fair-start delay might not be acceptable. For example, the traces used in the experiments have an average job runtime of 1.5 h. Thus, 0.5% of the resources would be wasted for a fair-start delay of 30 s, which is quite small. In contrast, if the average job runtime were 5 min, 10% of the resources would be wasted. Thus, an administrator has to reach a compromise between resource waste and fairness.

3.6.4 Validation

We have developed a proof-of-concept implementation of CooRMv1 in Python to validate the values obtained by simulations. The communication protocol has been ported to Corba for two reasons:

– Our architecture maps easily to a Remote Procedure Call (RPC)-based implementation, therefore, porting to Corba was straightforward.

– Compared to other commonly used encoding technologies, such as JSON, XML or YAML, CORBA IIOP has the smallest overhead.

Since we focus on the RMS-launchers interactions (which are functionally equivalent to the simulations), launchers are only deploying a sleep payload.

For these experiments, we used an instance of workload W_0 in which we replaced a percentage of applications with instances of the CEM application, similarly to W_2 and W_3 (see Section 3.6.1). We used the same resource model as during simulations, with $c = 8$. The RMS was run on the first processor, while all the launchers were run on the second processor of a system with two single-core AMD Opteron™ 250 processors, running at 2.4 GHz.

Figure 3.10a compares the TCP traffic generated in practice to the simulation results. Values obtained in practice are up to 50% higher than those obtained by simulations. This is caused mainly because of CORBA IIOP's overhead, but also because we neglected some messages in the simulations. However, the generated traffic is of the same order of magnitude, therefore we argue that the scalability from the network perspective is validated, even if the RMS and the launchers run on separate hosts.

Figure 3.10b compares the simulation time to the CPU-time consumed by the whole system (RMS and launchers). Practical values are up to 3.3 times higher than simulations. However, this is to be expected, since data needs to be marshalled/unmarshalled to/from CORBA. Also, the measured CPU-time includes starting up the launchers which, due to the need of loading the Python executable and compiling the byte-code, is non-negligible. Nevertheless, the consumed CPU-time is quite low, which shows that CooRMv1 scales well as the number of applications with intelligent resource selection increases.

To sum up, the differences between theoretical and practical results are implementation specific. A different implementation (e.g., another middleware than CORBA) might show values closer to the simulations. Therefore, we argue that the conclusions drawn in Section 3.6.2 and 3.6.3 are of practical value and that the CooRMv1 architecture is a viable solution for efficiently supporting moldable applications.

3.7 Discussions

Let us highlight some properties of the proposed solution. The RMS policy proposed in Section 3.5.2 does not attempt to do any global, inter-application optimizations. We deliberately chose a simple approach, as, in real-life, access to HPC resources is either paid or limited by a quota. This forces users to choose the right trade-off between efficiency and completion-time. CooRMv1 is flexible enough to allow applications to consider quota-related criteria when selecting resources. Nevertheless, should a better RMS policy be found, it can readily be used by changing only a few implementation details.

Single-cluster moldable applications can be reshaped [106], so as to improve system throughput. Such an approach might be extended and used as an alternative CooRMv1 policy, however, whether this benefits moldable applications with custom resource selection algorithms needs to be studied.

An alternative design could have allowed applications to submit *utility functions* [73], then let the RMS optimize global utility. We have refrained from adopting such a solution, because we do not believe that utility can be translated into comparable values for applications with a complex structure. Also, there is nothing preventing a user from "exaggerating" his utility, thus acquiring more resources than deserved.

3.8 Conclusion

This chapter addressed the first issue of the Thesis, that of efficiently supporting moldable applications. We started by presenting a motivating use-case, the resource selection algorithm of a Computational ElectroMagnetics (CEM) application. We showed that using this algorithm in practice is difficult, thus, we formulated the problem of efficiently scheduling moldable applications. As a solution, a centralized RMS, called the CooRMv1 architecture, has been proposed, which allows application to employ their custom resource selection algorithms for improved performance. Experiments done with a workload mixing rigid and moldable applications showed that the architecture is fair and scalable.

CooRMv1 is applicable to resources in which centralized control can be assumed, such as clusters and supercomputing centers. However, this is difficult to enforce for geographically-distributed, multi-owner resources, as can be found in Grid and Sky computing. Therefore, the next chapter extends CooRMv1 and proposes a distributed version of it.

CHAPTER 4

*dist*CooRM: A Distributed RMS for Moldable Applications

> If you can't write it down in English, you can't code it.
>
> Peter Halpern

In this chapter, we extend the previously introduced problem, that of efficiently scheduling moldable applications, to the case of geographically-distributed resources, as featured in Grid and Sky Computing. We focus on two issues: devising a solution for resources owned by multiple administrative domains and ensuring the fault-tolerance of the system to network bisections.

To this end, we extend the concepts, interfaces and protocols presented in Chapter 3 to a distributed version. Thus, we propose distCooRM*, a distributed resource management architecture, that efficiently supports moldable applications by allowing them to employ their custom resource selection algorithms. Experiments show that the system is well-behaved and scales well for a reasonable number of applications.*

4.1 Introduction

In the previous chapter, we have shown that moldability can be used to improve application performance. We have given an example of an application which uses a custom algorithm to select resources so as to optimize its response time. Next, we have presented a resource management architecture, CooRMv1, in which applications are actively involved in scheduling decisions, enabling them to use their custom resource selection algorithms. CooRMv1 is especially well suited in multi-cluster data centers (e.g., Clouds) or supercomputing centers, in which a centralized Resource Management System (RMS) can be installed. However, a centralized solution has several disadvantages, which makes it impractical for resources which are geographically distributed, as can be found in Grid or Sky Computing:

Multi-owner Since multiple institutions may own the resources, each one would like to keep its independence, so as to control how much and in what conditions their resources are shared with other institutions.

Fault-tolerance A centralized solution has a single point of failure, the RMS. If this component fails, all users are denied access to computational resources. Also, if the network becomes bisected, users which are separated from the RMS cannot use the resources that are on their side of the bisection.

Note that, in this Thesis, since we are interested in resource management, we only deal with fault-tolerance when negotiating resource allocations, i.e., when applications are waiting. In particular, we do not deal with making applications themselves fault-tolerant after they have started executing.

Scalability As of today, the scale of HPC Grids and Clouds is limited judging by the number of distinct parallel machines (clusters or supercomputers) that have to be managed. For example, in Europe, DEISA [52] was composed of 11 supercomputers, while its follow-up, the PRACE project [139], proposed in its last call up to 25 clusters or supercomputers[1]. In North America, the XSEDE project gives access to 9 supercomputers [150]. To our knowledge the largest of these infrastructures is EGI, which contains 109 HPC clusters [128]. Nevertheless, when devising a solution, we have to make sure that scalability is not sacrificed for solving the former two issues.

Due to the above reasons, one needs to split the centralized RMS into several agents. This, however, complicates the problem, as co-allocating resources across multiple agents is difficult, as highlighted in Section 2.3.4. Implementing co-allocation on top of agents with uncoordinated queues, as is commonly the case with batch schedulers, is cumbersome and wasteful [83]. Using advance reservations to coordinate the queues is less cumbersome, but still **wasteful** (see Figure 2.4 in Chapter 2) and, moreover, may lead to distributed deadlocks or livelocks.

Let us illustrate the latter two with an example. Figure 4.1 presents the Gantt charts of two initially empty resources R_1 and R_2. Two applications A_1 and A_2 enter the system at the same time. They first gather information about the availability of the resources and compute a resource selection. For example, both might want to reserve both resources (Figure 4.1a). Next, each application reserves their computed selection. However, since we are in a distributed system, with no global coordinator and heterogeneous network latencies, it might happen that A_1 first reserves R_1 and A_2 first reserves R_2 (Figure 4.1b). Then, both applications attempt

[1]Strictly speaking, DEISA is not operated as a Grid, as it does not respect condition 1 and 2 of the Grid definition given in Section 2.1.3. Indeed, in their propositions, users explicitly state the machine they are interested in. We are mentioning them here just to give an idea of the scale of such systems.

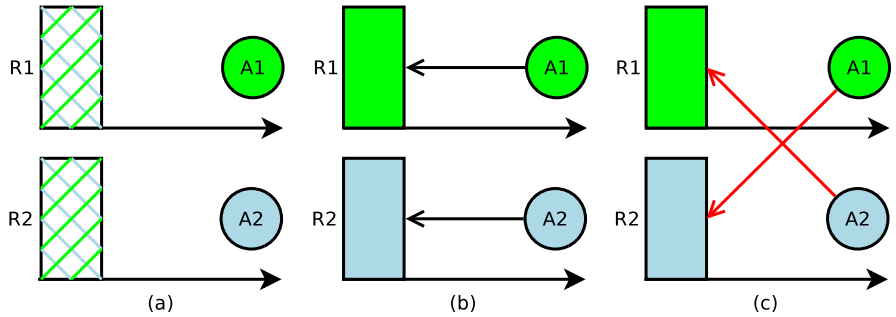


Figure 4.1: Deadlock with Advance Reservations

to reserve the other resource, however, since the other application already took that slot, their second reservations fail (Figure 4.1c).

There are several strategies that the applications may adopt. They may either *wait* for the other slot to become free, in which case a **deadlock** occurs. They may cancel and *retry* the whole process, in which case a **livelock** may occur. Effectively, since there is no tie-breaking between the two applications, they cannot reliably select resources.

This chapter takes a first step towards solving the above issues. We propose *dist*CooRM, a distributed resource management architecture, which builds upon CooRMv1 and advance reservations. It guarantees that applications can reliably and efficiently select resources using **loose reservations** and resource state change **notifications**. Deadlocks are eliminated by imposing a **global order** on applications in the system.

The remaining of this chapter is structured as follows. Section 4.2 presents *dist*CooRM the novel resource management architecture. Section 4.3 proposes an implementation, which is used for the evaluation of the architecture in Section 4.4. Finally, Section 4.5 concludes the chapter.

4.2 The *dist*CooRM Architecture

This section presents *dist*CooRM. It starts by describing the principles of the system. Then, it enumerates the agents that are part of the system and details the interfaces they export. Finally, the interactions between the agents in usual and exceptional cases are detailed.

4.2.1 Principles

*dist*CooRM builds upon CooRMv1 (see Section 3.4). At the core of the application-RMS interactions are the same two concepts: requests and views. **Request**s allow applications to express what resources they want to have allocated. Applications compute requests based on their current **view**, which contains the most up-to-date information about the resources that are available to an application, as computed by the RMS.

request

view

A fundamental difference between *dist*CooRM and CooRMv1 is that in *dist*CooRM the RMS is distributed. There are multiple agents that manage resources and collectively deal with the requests and views of applications. Each agent operates independently on **partial** information, i.e., a set of **partial request**s and **partial view**s for all applications, but restricted to the resource it manages. As a consequence, a resource request, that is computed by the application-side resource selection algorithm, will eventually have to be split into several partial requests before reaching each of the agents that manage the resources. Likewise, the partial views coming

partial request

partial view

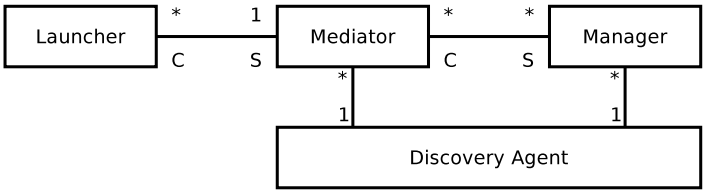


Figure 4.2: Overview of *dist*CooRM agents. Roles: C = Client, S = Server.

from different agents will eventually have to be merged into a single view before inputing them to the application-side selection algorithm.

Regarding the start-time, in CooRMv1 the start-time of each allocation is computed by the rms. This approach cannot be used in *dist*CooRM any more. Due to the distributed nature of the solution, each partial request is scheduled independently by a different agent. In order to allow applications to coordinate partial allocations, requests are extended to contain a **start-time**. This is similar to advance reservations, however, in *dist*CooRM the start-time is not strict, but rather it is interpreted as not-earlier-than. The allocation is allowed to start later than the start-time given in the request, but not earlier. As such, we define a **loose reservations** as a resource request containing a cluster, a node-count, a minimum start-time and an end-time. For example, an application might request 4 nodes on cluster C_1 from time 100 to time 120 and 5 nodes on cluster C_2 from time 100 to time 120. Note that, since each agent might take scheduling decisions at a different time, the actual start-time of the partial allocations might skew. For the previous example, the application's allocation might start at time 101 on cluster C_1 and time 102 on cluster C_2. Since we deal with moldable applications, these partial allocations have to be merge in a single allocation before the application may start executing on them. We shall discuss this more when presenting the interactions in *dist*CooRM.

loose reservations

Using loose reservations instead of advance reservations helps make the system converge faster. For example, if two applications A and B reserve the exactly same slot on the same resources, the system may delay the reservation of B. The application can deduce if this happens by monitoring its view. It can then either decide to accept this delay or adapt and send a new request.

One more issue remains. Since scheduling decisions are taken for each resource independently, there is a risk of deadlocks occurring (as exemplified in Figure 4.1). In order to break deadlocks when allocating resources, a **global order** is imposed on applications. When an application initially enters the system its submit-time (with a certain precision, for example, microsecond) and a randomly generated session ID is recorded. This data is then transmitted throughout the whole distributed system and must be used to order applications in the scheduling algorithms. This is further illustrated by giving an example implementation in Section 4.3.

global order

4.2.2 Agents

launcher

Figure 4.2 shows the main types of agents that are part of the system. A **launcher** is a user-code, part of an application, that is responsible for implementing the application-specific resource selection algorithm and negotiating the resources on which the application is going to execute.

mediator

An **mediator** is the user entry-point to access the resources. Its main purpose is to hide the complexity of how resources are reached, thus, providing the user with a more general interface. It also deals with various issues, such as authenticating the user, traversing firewalls, etc.

manager

A **manager** is responsible for multiplexing access to a computational resource, such as a su-

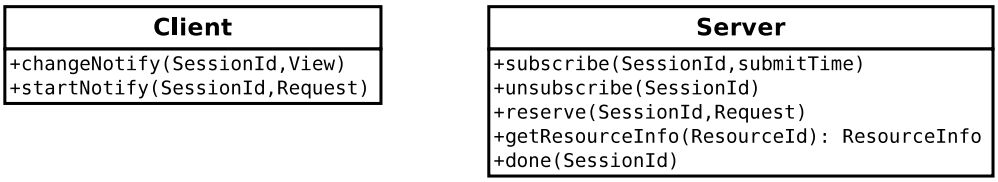


Figure 4.3: Interfaces corresponding to each role

percomputer or a cluster, and enforcing exclusive access. More precisely, this agent is responsible for tracking the requests and allocations for the resource it manages.

Each of these agents takes the role of a server, a client or both. A manager acts as a server and exposes an interface to give access to resources, while a launcher acts as a client and uses the server interface to gain access to resources. A mediator is a proxy which acts as a server to launchers and as a client to managers. More details about the interface of each role is given in the next section.

discovery agent

Additionally, we assume that the mediators and managers are connected to a **discovery agent**. Its purpose is to provide a publish/subscribe catalog allowing mediators to discover managers. The discovery agent could either be centralized or could itself be distributed. For example, it could be implemented using a Distributed Hash Table (DHT). There are many works dealing with implementing a scalable discovery service (see, for example, Section 2.3.3). Therefore, in this chapter we shall focus on the interactions which take place between the launchers, the mediators and the managers.

4.2.3 Interfaces

In the previous section we have defined two roles: server and client. In this section, after defining a few data types, we shall describe the interface of each role. The datatypes used in *dist*CooRM are the following:

- `SessionId` is a unique identifier of an *instance* of a launcher used throughout the whole system, for example a UUID;
- `ResourceId` is a unique identifier of a resource used throughout the whole system, for example the fully-qualified domain name of the node on which the manager is running;
- `Request` describes a loose reservation, by specifying the `ResourceId` on which the allocation should take place, the earliest **start-time**, the **duration** and the **number** of computing nodes to reserve;
- `ResourceInfo` stores static information about a resource, allowing the application to estimate its computing power and network connectivity. For example, it may store the number of CPUs and GPUs, the computing power of a node in FLOPS, the latency and bandwidth of the Local-Area Network (LAN), etc. For the Wide-Area Network (WAN), the Vivaldi coordinates [38] of the resource allow applications to estimate network latencies, while the inbound and outbound bandwidth allow the estimation of the bandwidth using the last-mile model [8];
- `View` (same as in CooRMv1, see Figure 3.2) stores availability information about resources. It is a data structure that associates to each `ResourceId` a step function representing the number of available nodes as a function of time.

Let us now describe the interface of each role (Figure 4.3). A server exposes an interface which allows a client to subscribe/unsubscribe to resource state notifications, manipulate reservations

(`reserve`), get static information about resources (`getResourceInfo`) and signal completion of the execution (`done`). Note that, the subscribe message contains a time-stamp (`submitTime`) which is generated by the mediator (the time-stamp given by the launcher is ignored, for security reasons). In order to enforce a global order among launchers and solve the deadlock issue presented in Figure 4.1, resource agents *must* use this time-stamp when taking scheduling decision, as illustrated in Section 4.3. This is in constrast to other resource management systems, such as batch schedulers, which take scheduling decisions based on the time at which a reservation has been requested.

A client exposes an interface which allows it to be notified when dynamic information about the resources have changed (`changeNotify`) or when its reservation has started (`startNotify`). We have chosen for the `startNotify` message to have the same signature as the `reserve` message, for the following two reasons:

- Spatial disambiguation: a launcher needs to determine on what resource its requested allocation has started. Indeed, since the start-time of the partial allocations may skew, as managers may take scheduling decisions at different moments of time, the launcher may receive multiple, partial `startNotify` messages from different resources. Hence, by looking at the `ResourceId` of the `Request` with which `startNotify` was called, a launcher can find out the resource on which its allocation has started;
- Temporal disambiguation: a launcher needs to make sure that the `startNotify` it has received corresponds to the last request it had sent. As a launcher adapts to new views it receives, it may send new requests. Since we are in a distributed system, it may happen that a `startNotify` message is send by a resource, before it received the last `reserve` message from the launcher. To deal with this situation, a launcher should compare the fields of the `Request` it has last send with the information contained in the `startNotify` message.

Of course, the above disambiguations can also be achieved by giving each request a unique identifier. However, we refrained from adopting this approach, as we wanted to send the minimum amount of information.

4.2.4 Interactions

The interactions between launchers, mediators and managers occur similarly to CooRMv1 (see Section 3.4.4). Mediators only act as relays, splitting resource requests coming from launchers into separate, partial requests for each manager and forwarding views from managers to launchers. More advanced mediator implementations may cache static resource information, merge partial views and merge partial `startNotify` messages into a single one. In our case, we chose to leave merging of partial information coming from the managers to the launcher.

From the *dist*CooRM launcher's and manager's point of view, there are two differences compared to CooRMv1:

- Launchers may receive partial `startNotify` messages, which indicate that some managers have started their allocations, but allocations on other managers have yet to start;
- Launchers need to send an `unsubscribe` message to inform managers to no longer send dynamic availability information (views). In CooRMv1, this was implicitly done when the application started. We shall illustrate through an example, why in *dist*CooRM unsubscribing needs to be explicit.

For better illustration, let us give examples of interactions between a launcher, a mediator and two managers in three scenarios which we call the trivial scenario, the typical scenario and the start-abort scenario. In the **trivial scenario** (Figure 4.4), we assume that the state

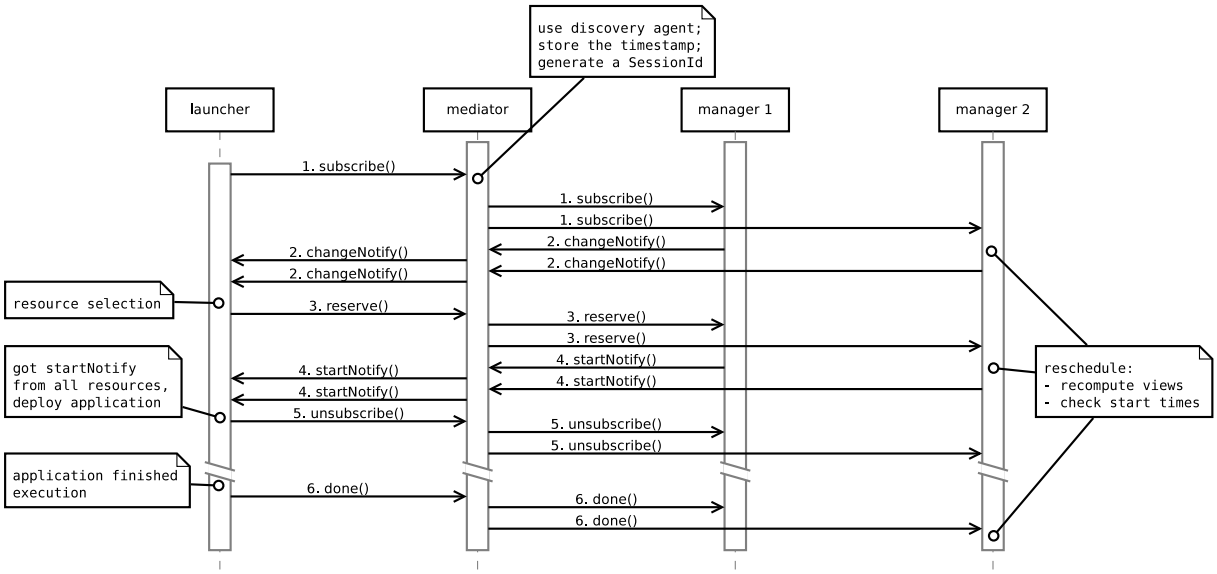


Figure 4.4: Trivial Scenario

of the resources does not change between the arrival of the application and its start, therefore, negotiation is straight-forward. Upon entering the system, the launcher subscribes to receive resource information (Step 1). It receives views from the two managers (Step 2) and, if necessary, retrieves static information about the resources (not illustrated for briefness). Next, it uses this information to run its resource selection algorithm and sends resource requests (Step 3). Some time later, the managers start the requested allocations and send `startNotify` messages to the launcher (Step 4). When all allocations have started, the launcher unsubscribes (Step 5) and deploys the application on the node IDs it has received. Finally, when the application has finished executing a `done` message is sent to the managers (Step 6).

In the **typical scenario** (Figure 4.5) the state of the resources changes before the application starts, for example, because a higher priority application changed its reservation. In this case, the launcher receives a `changeNotify` message (Step 4), re-runs in resource selection algorithm and changes its reservations (Step 5). Otherwise, the interactions are identical to the trivial scenario (steps 6 to 8).

Finally, in the **start-abort scenario** (Figure 4.6) the state of the resources managed by one manager changes, while the other manager already started the resource allocation. In such a case, the application receives a `startNotify` from one manager (Step 6) and a `changeNotify` from the other (Step 7). The application re-runs its resource selection algorithm and submits a new reservation (Step 8). Effectively, it has aborted the start on one resource, hence the name of this scenario. In order to continue negotiating, the launcher needs to continue receiving resource state change notifications until *all* its allocations have started. This is why we had to make unsubscription explicit in *dist*CooRM.

Note that, the both `startNotify` skew as well as the start-abort scenario may waste resources. Initial experiments revealed that this waste is negligible compared to the execution-time of the targeted applications. Therefore, we have not looked further into this issue.

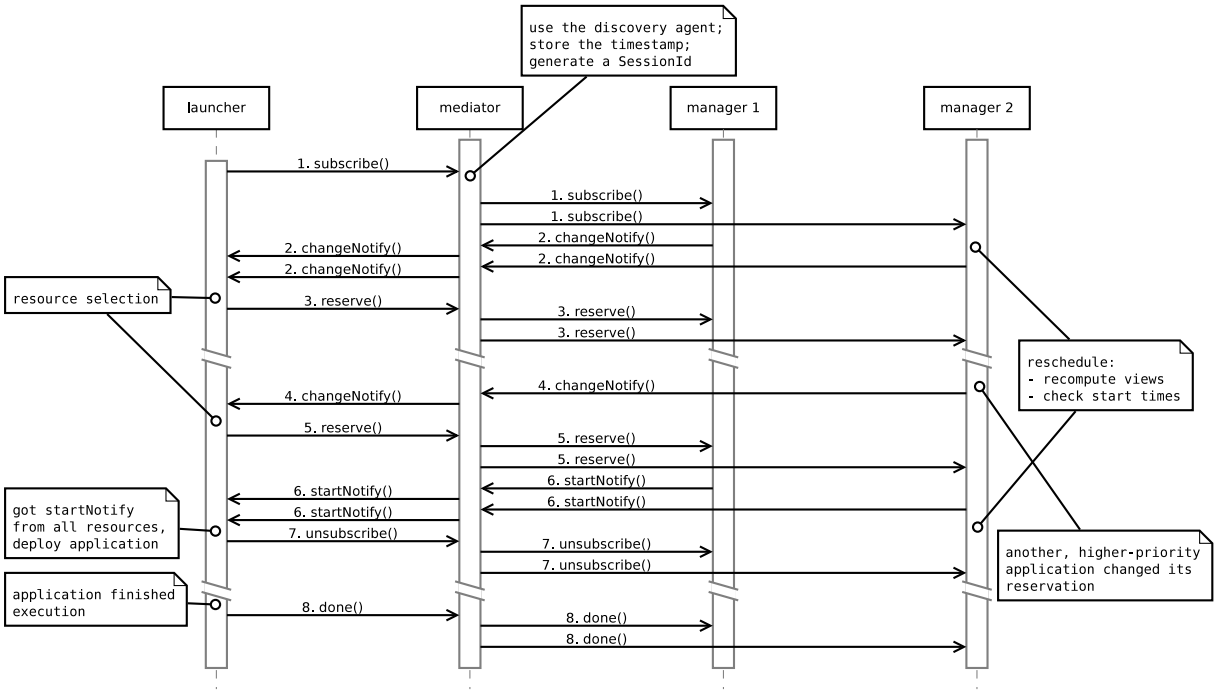


Figure 4.5: Typical Scenario

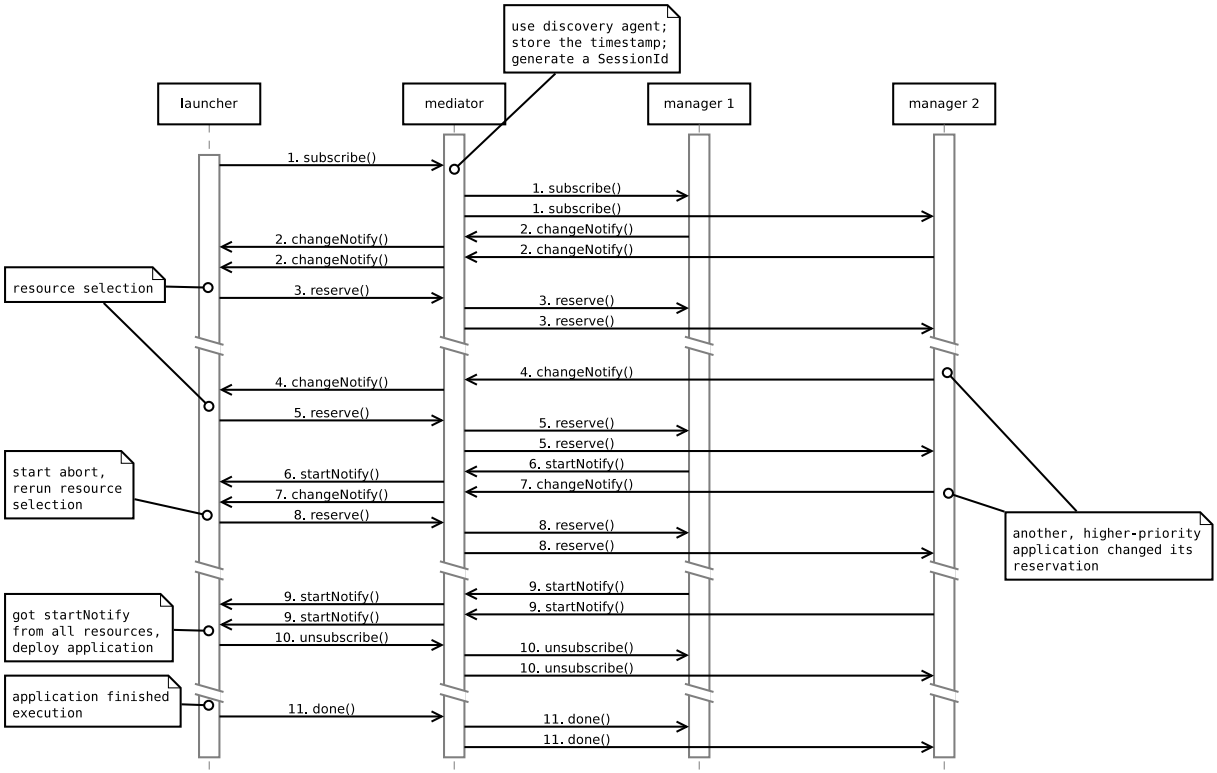


Figure 4.6: Start-abort Scenario

4.3 An Example Implementation

This section describes one possible implementation for the *dist*CooRM architecture, which is used in the evaluation section. To test the prototype, we have implemented the *dist*CooRM agents as part of a simulator, which uses the SimGrid framework [33].

Manager A *dist*CooRM manager is responsible for managing one resource (in our case a cluster). It has to decide how to allocate nodes, it has to compute views, send them to applications and process resource requests from applications. Our manager implementation works similarly to the CooRMv1 RMS implementation given in Section 3.5.2: It waits for requests from applications, runs a scheduling algorithm which computes views and application start-times, sends `changeNotify` messages for applications whose views have changed and `startNotify` messages for applications whose allocations have started. There are three implementation differences which deserve to be highlighted:

1. To ensure that applications are treated consistently by all managers (otherwise, deadlocks or livelocks may occur), the list scheduling algorithm (Algorithm 3.1 in Chapter 3) iterates through the applications as given by their **global order**: Applications are first sorted by the time-stamp received in the `subscribe` message, then by session ID. This is similar to the behavior of the CooRMv1 RMS, except that the time-stamp is given by a mediator and not by the manager itself.
2. When computing the start-time of a request, the first "hole" in the availability of the resources is searched starting from the minimum start-time given in the request. This is in contrast to CooRMv1, whose requests do not contain a start-time, therefore, the "hole" is searched starting from the current time.
3. The application is not explicitly unsubscribed when it starts, as needed to deal with the start-abort scenario described in the previous section. CooRMv1 automatically stops sending `changeNotify` messages once the application has started.

As in the case of CooRMv1, the scheduling algorithm is triggered every time a `subscribe`, `reserve` or `done` message arrives. However, to lower the CPU usage on the nodes the mediators are running, the scheduling algorithm is run at most once per **coalesce interval**, an administrator-chosen parameter. How to chose a value for this parameter is discussed in the evaluation section.

coalesce interval

Mediator A *dist*CooRM mediator acts as a proxy between launchers and managers. We have devised a naïve implementation which assumes that the system is static, i.e., managers do not change during the live-time of a mediator. During initialization, the mediator first uses the discovery agent to discover managers, then retrieves static resource information by calling their `getResourceInfo` method. This information is cached for serving it to the launchers when they request it.

Next, the mediator waits for launchers to connect to it. When a `subscribe` messages is received the current time is recorded and used as a time-stamp in the `subscribe` messages which are sent to all discovered managers. `changeNotify`, `startNotify` and `done` messages are relayed unchanged. `reserve` messages are split by the resource ID they target and the resulting sub-requests are sent to the corresponding manager.

Launchers We have implemented three launchers: rigid, single-cluster moldable and multi-cluster moldable for the CEM application. Their resource selection algorithms are identical to those of the applications used to evaluate CooRMv1, which are described in Section 3.5.1.

A *dist*CooRM launcher works as follows. At initialization, it connects to the mediator running on the same node as itself, using a well-known method (e.g., a pre-define TCP port on localhost). It sends a `subscribe` message and enters the monitoring phase. During this phase, it listens for `changeNotify` messages and updates its local view of the resources with the information contained therein. When its view changes, it runs the resource selection algorithm corresponding to the application type it handles and sends a `reserve` message. As in the case of managers, to reduce the CPU usage, the resource selection algorithm is called at most once per coalesce interval. For simplicity, we assume that the whole system is characterized by a single coalesce interval and that the same value is being used by all managers and launchers.

The launcher also listens for `startNotify` messages. When all requested allocations have started, it sends an unsubscribe message and deploys the application on the node IDs that have been allocated to it. When the application finished executing, a `done` message is sent.

The above agents have been implemented in a simulator which uses the SimGrid framework [33]. Our contribution consists of 3600 SLOC[2] of C++ code.

4.4 Evaluation

In this section, we evaluate whether the devised solution solves the issues raised in Section 4.1. First, we give the experimental setup common to all experiments, next we present experiments which test whether the system behaves well in a multi-owner environment. Finally, the scalability of the system is evaluated.

4.4.1 Experimental Setup

To evaluate *dist*CooRM, we are using a resource and an application model similar to the ones used to evaluate CooRMv1 in Section 3.6.1. Let us resume the main elements here.

Application Model The workload is composed of 200 applications: 100 CEM applications, 80 single-cluster moldable applications and 20 rigid applications, that are implemented as presented above. The single-cluster moldable applications and rigid applications have been generated by choosing among 200 consecutive jobs from the `LLNL-Atlas-2006-1.1-cln` trace from the parallel workload archive [129]. For rigid applications, the requested node-count and execution-time are taken unmodified from the traces. For single-cluster moldable applications, the node-count and the execution-time found in the traces are used to reconstruct the sequential execution-time, as given my Amdahl's law (see Equation 3.1). The amount of parallelism and maximum parallelism is chosen randomly as presented in Table 3.1.

To simulate the inherent unreliability of estimating the execution time, each application sends requests with a duration that is s times larger than its execution time, where s is a uniform random variable in $[1.1, 2]$. In order to test the system's properties in an extreme case, when it is highly loaded, the inter-arrival time is set to 1 second. In comparison, in the original traces approximately 7 applications arrive every hour.

[2] as measured by David A. Wheeler's SLOCCount utility [148]

Cluster ID	x	y	z
1	−3.72915245662	24.7160772325	3.62602427974
2	−4.25755167998	24.9087688069	−0.587859438454
3	5.57375542796	24.4550412426	3.05256792234
4	5.0461271909	24.7167805855	−1.15395583134
5	3.5885666087	−24.6894143829	−3.86609732834
6	0.208382141164	−24.4527765207	−6.40596892806
7	−1.52894680558	−24.9203236456	3.93929406457
8	−4.90118042655	−24.7341533185	1.39599525955

Table 4.1: Vivaldi coordinates, in milliseconds, which approximate the latencies targeted in the resource model in Section 4.4.1. The square root of the sum of the residuals is 3.61.

Resource Model The resource model consists of 8 clusters each having one frontend and 128 computing nodes. We consider that the LAN latency is negligible, e.g., the latency between the frontend and the nodes is zero. To simulate heterogeneity, cluster $i \in [2, 8]$ is considered $1 + 0.1 \cdot (i - 1)$ times faster than cluster 1.

To model the WAN, every 2 clusters are grouped in a city, every 2 cities are grouped into a country. Next, 3D euclidean Vivaldi Coordinates are assigned to each cluster, so that the resulting latencies be 5 ms between clusters belonging two the same city, 10 ms between clusters belonging to the same country and 50 ms otherwise (Table 4.1).

During initial experiments, we measured the size of any message to be less than 100 bytes. Therefore, for simplicity, we neglect transmission delays due to limited bandwidth, i.e., our network model behaves as if bandwidth was infinite.

Deployment Model A mediator and a manager are deployed on the frontend node of each cluster. Application launchers are also deployed on frontend nodes (either all on the same frontend or on different ones, depending on the experiment) and connect to the mediator which runs on the same node. Initially, we set the coalesce interval to 1 second. The importance of this parameter is studied in the second part of the evaluation.

Metrics We are interested in the following two metrics: the average completion time and the maximum traffic per frontend. The **average completion time** measures the average of the completion times of the applications. Its purpose is to quantify the quality of the schedule that the system devised.

average completion time

The **maximum traffic per frontend** measures the maximum of the number of bytes that both enter and exit a frontend node. Note that, since the size of the messages is small, it would make more sense to measure the number of messages. However, we preferred measuring bytes, so as to be able to compare *dist*CooRM to CooRMv1. This metric serves to judge the scalability of the system.

maximum traffic per frontend

4.4.2 Multi-owner Feasibility

In this section, we evaluate *dist*CooRM's behavior when resources are managed by multiple owners and also compare its performance to a centralized solution. To this end, we first deploy all launchers on the same frontend. Effectively, the other mediators are not taking part in the interactions, in other words, they are inactive. Next, we randomly "spread" the launchers on an

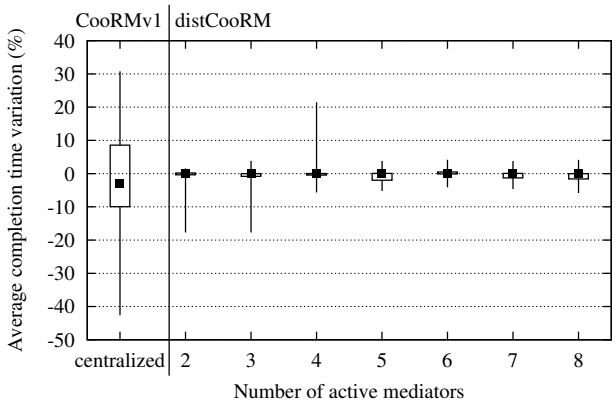


Figure 4.7: Variation of average completion time

increasing number of frontends, thus increasing the number of active mediators, and study how this affects the devised schedule. Ideally, the system should behave exactly the same: applications should be allocated the same resources during the same time-slots.

To verify the behavior, for each workload, we measure the Average Completion Time (ACT) of the applications for a number of mediators between 1 and 8. Then, for each workload, we compute the relative difference between the ACT when the number of mediators is greater than 2, and the ACT when the number of mediators is 1. For comparison with a fully-centralized solution, we also compute the relative difference between the ACT obtained with CooRMv1 and *dist*CooRM with a single mediator.

Figure 4.7 plots the difference of the average completion time as a function of the number of active mediators, relative to the case in which a single mediator is active. The plot shows the minimum, the first quartile, the median, the third quartile and the maximum of the measured metric. Two observations can be made: First, when looking at the minimum and maximum, we observe that the resulting ACT with CooRMv1 and *dist*CooRM may be quite different. This is due to the fundamental different ways that the two systems work. Nevertheless, when looking at the median, the ACT obtained with *dist*CooRM is only 3% higher than CooRMv1, from which we conclude that *dist*CooRM is competitive compared to a centralized solution.

Second, the experiments show that, for *dist*CooRM, varying the number of active mediators has a limited impact on the average completion time. In the studied experiments, the difference is always less than 21% and is mostly less than 2%. In fact, the quartiles can barely be distinguished. While the deviations are small, the observed behavior is not ideal (i.e., zero deviations), which is why we wanted to have more refined explanations. Therefore, we have studied in detail a few traces of the simulations. We identified three causes for the differences in schedules:

C1. Due to different observed latencies, the `reserve` messages arrive at the managers at moments which can slightly differ from one experiment to another. This in turn influences the time at which the scheduling algorithms of the managers are triggered, which causes changes in the moments the `changeNotify` messages are sent and arrive at launchers. These small differences propagate from higher-priority applications to lower-priority ones. Thus, the launchers may receive views which are slight delayed and, in the end, requested start-times of the allocations may drift.

C2. Due to the previous reason, the resources that are available when a moldable application runs its scheduling algorithm can differ from one experiment to another. For example, let us

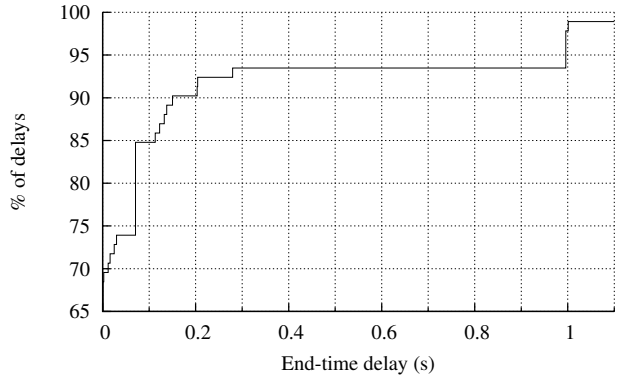


Figure 4.8: Empirical distribution function of end-time delays

assume we have two experiments α and β. In both experiments, an application A reserved resources until t_1, but finishes a lot earlier at time $t_0 \ll t_1$. Let us further assume that in experiments α, due to the way previous `changeNotify` messages arrived, a lower-priority application B runs its scheduling algorithm at $t_0 - \epsilon$. According to its local view, the resources that A is currently running on are occupied until t_1, so it decides to immediately reserve the available resources and start on them, expecting to finish at $t_\alpha < t_1$. Since the applications are only moldable (and not malleable), at moment t_0, when A finishes, B is unable to take advantage of the newly freed resources.

In exchange, in experiment β, the application B runs its scheduling algorithm at $t_0 + \epsilon$. Its local view contains the resources that have just been freed by application A. Application B reserves the resources freed by A, which are somewhat faster, and starts on them, thus finishing at $t_\beta < t_\alpha$. In the end, the end-time of B is smaller in experiment β than in experiment α.

C3. When from one experiment to another an application selects different resources, which reduce its end-time (such as application B in the previous paragraph), it is highly likely that a lower-priority application C is forced to choose slower resources, thus increasing its end-time.

In our opinion, end-time increases (i.e., delays) caused by C3 are not really an issue. As in the case of CooRMv1, *dist*CooRM should favor higher-priority applications and allow them to adapt to the available resources as they choose fit.

To have more insight into the contribution of the above causes to the end-time variations shown in Figure 4.7, let us study the distribution of the end-times. For each application, we compute the end-time delay, i.e., the difference between its end-time when multiple mediators are active in the system and its end-time when a single mediator is active. We ignore delays caused by C3 (as argued above) as follows: When an application A reduces its end-time, all the applications that have arrived later than A are ignored. Since the number of delays that occur is small, instead of having separate plots for each mediator-count from 2 to 8, we chose to aggregate the results of all these experiments into a single plot.

Figure 4.8 plots the empirical distribution function of the end-time delays, which shows two distinct features: First, the results show that more than 93% of all delays are significantly less than the coalesce interval (more precisely, less than 0.3 seconds), while more than 98.5% of all delays are less than 1.1 seconds, i.e., a little above the coalesce interval. We believe that these delays can be attributed to cause C1. Second, we had one instance when the delay was 13 seconds.

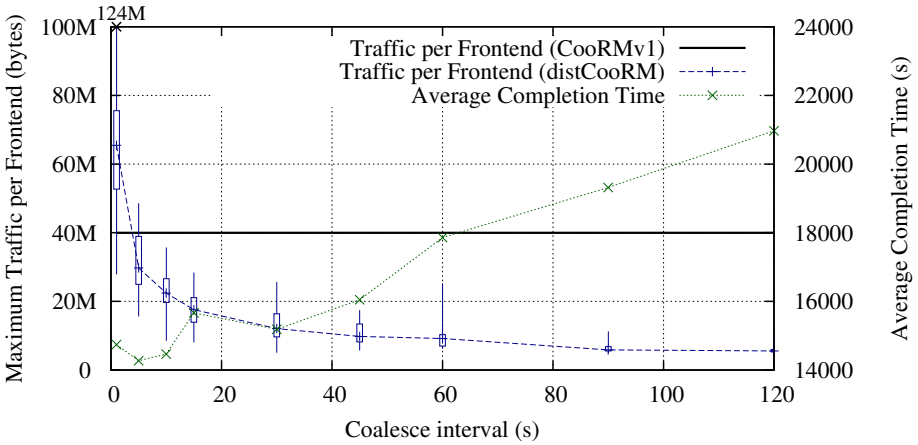


Figure 4.9: Results of the scalability experiment, compared to a centralized system

By carefully studying the traces, we were able to identify that these delays are due to cause C2: By incrementing the number of active mediators, the time when the scheduling algorithm of a launcher was triggered was advanced. Another application, which estimated a large maximum execution time, finished around the same time. Thus, the former application was delayed, because it missed an opportunity to execute on the resources that the latter application freed.

To sum up, we have tested *dist*CooRM and studied how the number of deployed mediators affects its behavior. We have shown through simulations that its behavior remains essentially the same. The small variations are mainly caused by the inherent uncertainty in the system, due to the inability to accurately predict execution times. This allows us to conclude that the system can be effectively used in a multi-owner environment. Each owner can deploy his own manager and mediator and, by doing so, not only do the owners keep their independence, but they also improve the fault-tolerance of the system: If a network bisection occurs, the waiting applications can continue requesting the resources that are on their side of the bisection and eventually start on them.

For the rest of the evaluation, we shall always deploy one manager and one mediator per cluster.

4.4.3 Scalability: Comparison to a Centralized rms

Let us now focus on the scalability of *dist*CooRM. Figure 4.9 shows both the maximum traffic per frontend and the Average Completion Time (act) as a function of the coalesce interval. As a reference, the traffic of the CooRMv1 rms node (with coalesce interval 1 second) is included in the graph. For each value of the coalesce interval, at least 10 experiments have been done. For the act we have observed large deviations, which, unfortunately, highlight artifacts and hide the trends we are interested in. Therefore, inspired by input shaking [112], we reduce the evaluation sensitivity and render graphs more readable by only plotting the medians.

One observes that, for the same coalesce interval (1 second), the amount of communication is more than 1.5 times higher for *dist*CooRM when compared to CooRMv1. This is due to the fact that in CooRMv1 it is up to the rms to choose a start-time for the requests of the applications. Therefore, if an application terminates earlier, the next applications in the queue are automatically advanced and might not need to send a new resource request. Hence, the negotiation between the applications and the rms is less costly in terms of communication. In

contrast, *dist*CooRM resource requests have a minimum start-time which is needed to allow an application to coordinate allocations on different managers. If an application terminates earlier and a "hole" is created in the schedule, managers are not allowed to advance the allocation of any application at their own initiative. Instead, it is up to the applications to detect (through their views) that a hole has been created in the schedule and potentially take advantage of it by sending new resource requests. While this allows both coordinating managers and filling in holes (otherwise, as in the case of advance reservations, resources would be left idle—see Figure 2.4), the negotiation protocol is more communication intensive.

The results also show that reducing the amount of communication, which translates to improving the scalability of *dist*CooRM, can be done by increasing the coalesce interval. In particular, when the coalesce interval is 10 seconds, the *dist*CooRM maximum traffic per frontend is lower than the traffic on the RMS node for CooRMv1 with a 1 second coalesce interval. On the downside, increasing the coalesce interval reduces the reactivity of the system and leaves resources idle. As the graph shows, this has a significant impact on the ACT, which nearly monotonically increasing. The deviations from a monotonic increase are due to a phenomenon similar to cause C2 described in Section 4.4.2: As the coalesce interval is changed, some moldable application may take better decisions and may thus contribute to reducing the ACT. Nevertheless, these spurious gains are eventually eclipsed by the increasing idleness of the resources.

In the end, choosing a coalesce interval of 10 seconds seems to be a good compromise between scalability and ACT, which is why we shall use this value in the following experiments.

4.4.4 Scalability: Strong and Weak Scaling with the Size of the Platform

Let us now test the limits of *dist*CooRM and see how it behaves on large platforms. For this set of experiments, we use the following resource and application models to test the scalability in the both the **strong** sense, i.e., the number of resources is increased, while the number of applications is kept constant, and the **weak** sense, i.e., the number of resources and the number of applications are increased at the same rate.

Resource Model We generate platforms with increasing number of clusters from 8 to 384, having the following characteristics: The computation power of cluster i is $\min(1+0.1\cdot(i-1), 2)$, i.e., the computation power of each cluster is 10% higher than the previous cluster, but no higher that twice the computation power of the first cluster[3]. As for the Vivaldi coordinates of the clusters, they have been taken from real traces measured "in the wild" on the Internet [136, 75].

Application Model As in the previous experiments, the workload consists of 50% multi-cluster moldable applications (in the previous experiments we used the CEM application), 40% single-cluster moldable and 10% rigid. For strong scalability, we fix the number of application at 200. For weak scalability, we fix the application-count to cluster-count ratio at 25, so as to obtain 200 applications for 8 clusters.

Unfortunately, the resource selection algorithm of the CEM application was not designed to scale to the number of clusters that we target in this set of experiments. It has a cubic complexity in the number of clusters and using it for all multi-cluster moldable applications would significantly slow down simulations. Therefore, to reduce the duration of the simulations, we have replaced the CEM applications with a simpler application, that we call **trivial multi-cluster moldable**

trivial multi-cluster moldable

[3]This reduces considerably the heterogeneity of the resources, as most of them will have twice the computation power of the first cluster, however, this does not influence the scalability results.

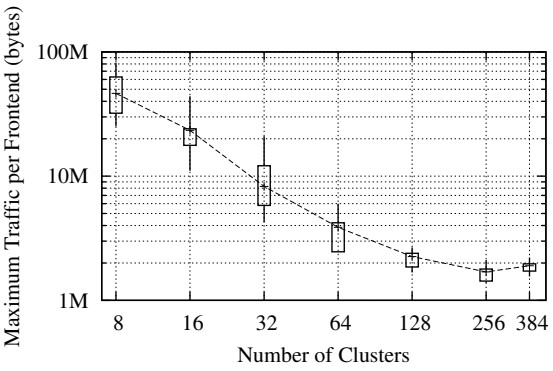


Figure 4.10: Strong scaling results

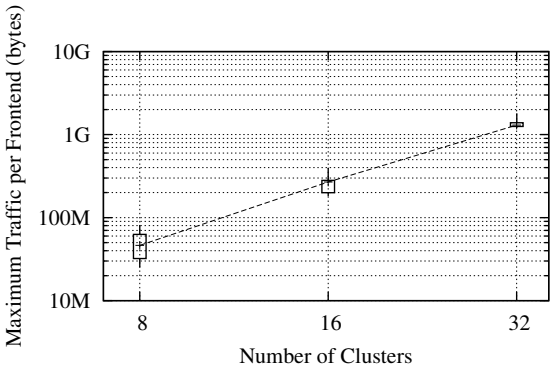


Figure 4.11: Weak scaling results

moldable applications. In essence, this application works similarly to the simple-cluster moldable application: It takes its sequential execution time from the `LLNL` traces and the maximum node-count randomly as presented in Table 3.1. However, the trivial multi-cluster moldable application assumes a parallelism of 1 and can execute on nodes spanning multiple clusters. Its execution time is computes by dividing the sequential execution time to the sum of the computation powers contributed by all nodes allocated to it. While we admit that this application model is somewhat simplistic, it is good enough for testing the scalability of *dist*CooRM. For example, the measured amount of communication for 8 clusters is similar to the workload with the original CEM application.

Strong Scaling Results Figure 4.10 plots the minimum, first quartile, median, third quartile and maximum traffic per frontend as a function of the number of clusters in the system for the strong scalability experiments. One can observe that for the left part of the plot, i.e., when the cluster-count is smaller than 256, the traffic per frontend decreases: As more resources are available to the applications, their average waiting time decreases and less negotiation takes place before they start. For the right part of the plot, i.e., cluster-count larger than 256, the amount of communication increases. This happens due to the fact that more static and more dynamic information is transmitted from the managers to the applications, which eventually overtakes the previously described phenomenon. Nevertheless, we observe that the increase is relatively small, therefore, we conclude that *dist*CooRM presents good strong scalability.

Weak Scaling Results Figure 4.11 plots the results of the weak scaling experiment. Due to the length of the simulations (approximately 4 hours for 32 clusters), we were only able to obtain a limited number of results. Nevertheless, the graph clearly shows that *dist*CooRM does not scale well in the weak sense: The maximum traffic per frontend shows a large increase as clusters are added to the system, instead of remaining constant, as would be ideal. This is due to the fact that, *dist*CooRM follows a $M \times N$ (quadratic) communication pattern, i.e., all M applications may potentially communicate with all N resources: On one hand, each manager sends to each application a `changeNotify` message, on the other hand, each application generates requests that (after being split by the mediators) may potentially arrive at all managers.

An improvement to the system would consist in adding a mechanism to limit communication complexity to at most linear. For example, a pre-selection phase might be added, so that each launcher only negotiates with a subset of managers based on network proximity. There is ongoing

work with the Myriads team at IRISA/INRIA in Rennes, France in this direction.

4.5 Conclusion

In this chapter, we have continued studying the problem of efficiently scheduling moldable applications. In contrast to the previous chapter, which solved the problem for resources in which centralized control can be assumed, such as clusters and supercomputing centers, this chapter targeted geographically-distributed, multi-owner resources, as can be found in Grid and Sky computing. As a solution, an RMS architecture called *dist*CooRM has been proposed, which combines the previously proposed centralized RMS (CooRMv1) with advance reservations. The evaluation has shown that the system is feasible, behaves well in multi-owner systems and can be deployed so as to improve the fault-tolerance of the system. *dist*CooRM showed to scale well for a reasonable number of applications, however, improvements have to be devised to reach large-scale.

Up to this chapter, we have been interested in scheduling moldable applications, which can only select resources before they start. As has been shown in literature and in Section 3.2, this considerably improves application response time. However, applications may take non-optimal decisions, due to the inherent uncertainty in the system: Accurately estimating execution times is difficult. Indeed, we have observed during our experiments that moldable applications would start just before additional resources would be freed. Therefore, resource management can further be improved by supporting dynamic, run-time negotiation of resources. This is the direction that we are heading for in the next part of the Thesis.

Part III

RMS Support for Malleable and Evolving Applications

CHAPTER 5

Towards Scheduling Evolving Applications

This chapter takes the first step towards supporting evolving applications in Resource Management Systems (RMSs). It focuses on the case where full information is available to the system and quantifies the gains that can be expected. More precisely, it deals with scheduling fully-predictably evolving applications, i.e., the complete evolution of the resource requirements is known to the system at submittal. Off-line scheduling algorithms are proposed, which show that several user- and system-centric metrics can be improved. Thus, this chapter paves the way to dealing with more challenging cases, where only partial information is available. Indeed, the proposed concepts and algorithms are used in the next chapter to design an RMS which supports non-predictably evolving applications.

5.1 Introduction

High-Performance Computing (HPC) resources, such as clusters and supercomputers, are managed by an RMS which is responsible for multiplexing computing nodes among multiple users. Commonly, users get an exclusive access to nodes by requesting a static allocation of resources, i.e., a *rigid job*, characterized by a node-count and a duration. Some resource managers support "dynamic jobs" which allow resource allocations to be grown or shrunken while the application is running. However, none of the existing RMSs are able to do this reliably, so as to guarantee that the application will actually receive additional resources (see Section 2.3.1 and 2.3.2).

This is required for supporting applications that exhibit **evolving** resource requirements, i.e., their resource requirements change during execution. The most commonly studied evolving applications are workflows. There are two main approaches for running them: submitting tasks as individual jobs [2] or creating a big pilot job, inside which tasks are scheduled [31]. However, none of these resource allocation abstractions are usable for running tightly-coupled, evolving applications, such as Adaptive Mesh Refinement (AMR) [97] simulations or component-based applications with spatial and temporal relationships [14, 98]. Indeed, the former change the working set size as the mesh is refined/coarsened, while the latter may have non-constant resource requirements as components are activated/deactivated during certain phases of the computation. Unfortunately, without being able to grow their resource allocation reliably, such applications can only use static allocations and allocate resources based on their maximum requirements. Needless to say, this leads to an inefficient resource usage.

This chapter quantifies the gains that can be made by supporting evolving applications in RMSs. It focuses on **fully-predictably evolving** applications (see Section 2.2) for which the system has full information about their evolution at submittal. Studying this case is interesting for two reasons. First, it paves the way to supporting marginally-predictably evolving applications. If little gain can be made with fully-predictably evolving applications, where the system has complete information, it is clear that it makes little sense to support marginally-predictable ones. Second, the developed algorithms can be extended to the marginally- and non-predictable case. Each time an application submits a change to the RMS, the scheduling algorithm for fully-predictable applications could be re-run with updated information.

The contribution of this chapter is threefold. First, it formalizes the novel problem of scheduling fully-predictably evolving applications. Second, it proposes a solution based on a list scheduling algorithm. Third, it evaluates the algorithm and shows that significant gains can be made. Therefore, we argue that RMSs should be extended to take into account evolving resource requirements.

The remaining of this chapter is structured as follows. Section 5.2 gives a few definitions and notations used throughout the chapter and formally introduces the problem. Section 5.3 proposes algorithms to solve the stated problem, which are evaluated using simulations in Section 5.4. Finally, Section 5.5 concludes.

5.2 Problem Statement

To accurately define the problem studied in this chapter, let us first introduce some mathematical definitions and notations.

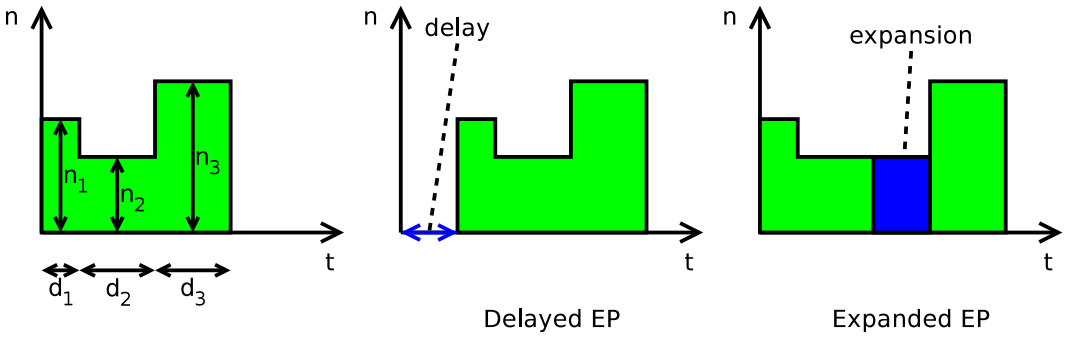


Figure 5.1: Example of an evolution profile, a delayed and an expanded version of it

5.2.1 Definitions and Notations

Let an **Evolution Profile (EP)** be a sequence of **steps**, each step being characterized by a *duration* and a *node-count*. Formally, $ep = \{(d_1, n_1), (d_2, n_2), \ldots, (d_N, n_N)\}$, where N is the number of steps, d_i is the duration and n_i is the node-count during Step i (see Figure 5.1). evolution profile

An evolution profile can be used to represent three distinct concepts. First, a **resource EP** represents the resource occupation of a system. For example, if 10 nodes are busy for 1200 s, afterwards 20 nodes are busy for 3600 s, then: resource EP

$$ep_{res} = \{(1200, 10), (3600, 20)\}$$

Second, a **requested EP** represents application resource requests. For example, requested EP

$$ep_{req} = \{(500, 5), (3600, 10)\}$$

models a two-step application with the first step having a duration of 500 s and requiring 5 nodes and the second step having a duration of 3600 s and requiring 10 nodes. Non-evolving, rigid applications can be represented by an EP with a single step.

Third, a **scheduled EP** represents the number of nodes actually allocated to an application. For example, an allocation of nodes to the previous two-step application might be: scheduled EP

$$ep_s = \{(2000, 0), (515, 5), (3600, 10)\}$$

For the first 2000 s, no nodes are allocated to the application, therefore, it has to wait before starting execution. Then, it would have 5 nodes allocated for 500 seconds, so as to execute its first step. Before being able to start its second step, which requires 10 nodes, it would have to wait another 15 s (= 515 s − 500 s). During this time, the application would still have 5 nodes allocated: it retains all its data, but it cannot do any useful computations. Finally, it would have 10 nodes allocated for 3600 seconds, during which it can execute its second step.

We define the expanded and delayed EPs of $ep = \{(d_1, n_1), \ldots, (d_N, n_N)\}$ as follows:

$$ep' = \{(d'_1, n_1), \ldots, (d'_N, n_N)\}$$

is an **expanded EP** of ep, if $\forall i \in \{1, \ldots, N\}, d'_i \geq d_i$ and expanded EP

$$ep'' = \{(d_0, 0), (d_1, n_1), \ldots, (d_N, n_N)\}$$

is a **delayed EP** of ep, if $d_0 > 0$. For a graphical representation of these two concepts, see Figure 5.1. delayed EP

For manipulating EPs, we use the following helper functions:

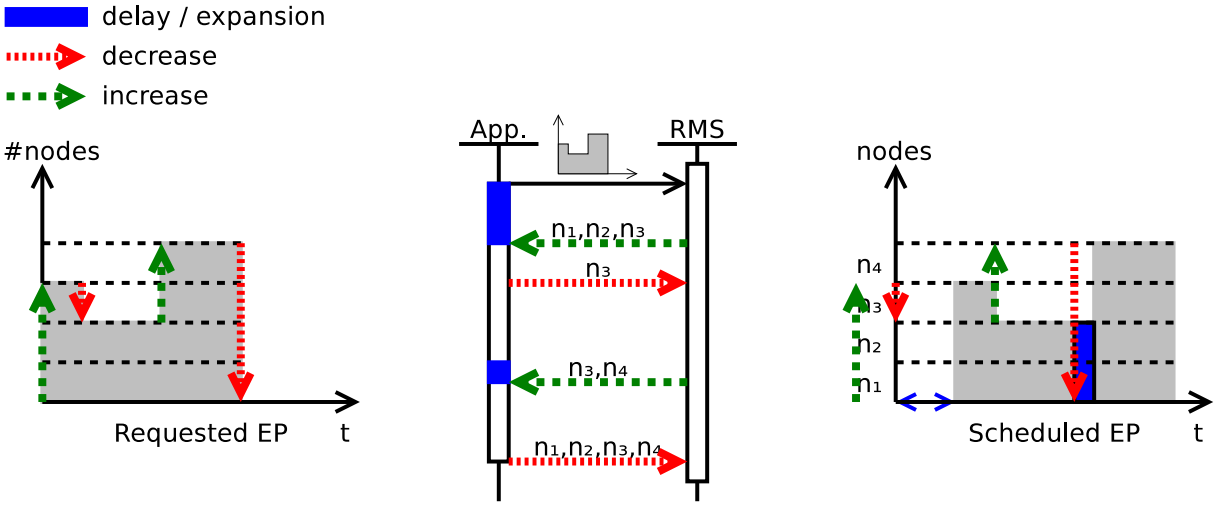


Figure 5.2: Example of interaction between a fully-predictably evolving application and the RMS

- $ep(t)$ returns the number of nodes at time coordinate t,
  i.e., $ep(t) = n_1$ for $t \in [0, d_1)$, $ep(t) = n_2$ for $t \in [d_1, d_1 + d_2)$, etc.
- $\max(ep, t_0, t_1)$ returns the maximum number of nodes between t_0 and t_1,
  i.e., $\max(ep, t_0, t_1) = \max_{t \in [t_0, t_1)} ep(t)$, and 0 if $t_0 = t_1$.
- $\text{loc}(ep, t_0, t_1)$ returns the end-time of the last step containing the maximum, restricted to $[t_0, t_1]$,
  i.e., $\text{loc}(ep, t_0, t_1) = t \Rightarrow \max(ep, t_0, t) = \max(ep, t_0, t_1) > \max(ep, t, t_1)$.
- $\text{delay}(ep, t_0)$ returns an evolution profile that is delayed by t_0.
- $ep_1 + ep_2$ is the sum of the two EPs, i.e., $\forall t, (ep_1 + ep_2)(t) = ep_1(t) + ep_2(t)$.

5.2.2 Towards an RMS for Fully-Predictably Evolving Applications

To give a better understanding on the core problem we are interested in, this section briefly describes the interaction between a fully-predictably evolving applications and a supportive RMS.

Let us consider that the platform consists of a homogeneous cluster of n_{nodes} computing nodes, managed by a centralized RMS. Fully-predictably evolving applications are submitted to the system. Each application i expresses its resource requirements by submitting a requested EP[1] $ep^{(i)}$ ($ep^{(i)}(t) \leq n_{nodes}, \forall t$). The RMS is responsible for deciding when and which nodes are allocated to applications, so that their evolving resource requirements be met.

During run-time, each application maintains a session with the RMS (Figure 5.2). If from one step to another the application increases its resource requirements, as shown by the coarse-dashed arrows, it keeps the currently allocated nodes and has to *wait* for the RMS to allocate additional nodes to it. Note that, the RMS can *delay* the allocation of additional nodes, i.e., it is allowed to expand a step of an application. However, we assume that during the wait period the application cannot make any useful computations: the resources currently allocated to the application are **wasted**. Therefore, **the scheduled EP** (the EP representing the resources

[1]Note that this is in contrast to traditional parallel job scheduling, where resource requests only consist of a node-count and a maximum execution time.

effectively allocated to the application) must be equal to the requested EP, optionally **expanded** and/or **delayed**.

If from one step to another the node-count decreases, as shown by the fine-dashed arrows, the application has to release some nodes to the system (the application may choose which ones). The application is assumed fully-predictable, therefore, it is not allowed to contract nor expand any of its steps at its own initiative.

To completely devise an RMS several issues would have to be dealt with. An RMS interface would have to be defined and the RMS-application protocol would have to be developed. Protocol violations should be detected and handled, e.g., an application which does not release nodes when it is required to should be killed. These issues are dealt with in Chapter 6.

In this chapter, we focus on quantifying the gains that can be made by developing such a system. The first step consists in finding an offline scheduling algorithm that operates on the queued applications and decides how nodes are allocated to them. It can easily be shown that such an algorithm does not need to operate on node identifiers (IDs): if for each application, a scheduled EP is found, such that the sum of all scheduled EPs never exceeds available resources, a valid mapping can be computed at run-time. The next section formally defines the problem.

5.2.3 Formal Problem Statement

Based on the previous definitions and notations, the problem can be stated as follows. Let n_{nodes} be the number of nodes in a homogeneous cluster. n_{apps} applications having their requested EPs $ep^{(i)}$ $(i = 1 \dots n_{apps})$ are queued in the system $(\forall i, \forall t, ep^{(i)}(t) \leq n_{nodes})$. The problem is to compute for each application i a scheduled EP $ep_s^{(i)}$, such that the following conditions are simultaneously met:

C1 $ep_s^{(i)}$ is equal to $ep^{(i)}$ or a delayed/expanded version of $ep^{(i)}$;

C2 resources are not overflown $(\forall t, \sum_{i=1}^{n_{apps}} ep_s^{(i)}(t) \leq n_{nodes})$.

Application completion time and resource usage should be optimized.

5.3 Scheduling Fully-Predictably Evolving Applications

This section aims at solving the above problem in two stages. First, a list-scheduling algorithm is presented, which transforms requested EPs into scheduled EPs. It takes as input a helper function which operates on two EPs at a time, a resource EP and a requested EP, and outputs a scheduled EP. Second, several algorithms for implementing this helper function are described.

5.3.1 An Algorithm for Offline Scheduling of Evolving Applications

Let us present an offline scheduling algorithm that solves the stated problem (Algorithm 5.1). The algorithm takes as input the requested EP $ep^{(i)}$ of each application i and the number of computing nodes in the system n_{nodes}. It outputs for each application i a scheduled EP $ep_s^{(i)}$, so that resources are not overflown.

The algorithm works as follows. Throughout the algorithm, the resource EP ep_r is maintained, which stores the future resource occupation, as computed after a certain step of the algorithm. This variable is initialized to the empty EP. Then, the algorithm considers each requested EP, potentially expanding and delaying it using a helper `fit` function. The resulting scheduled EP $ep_s^{(i)}$ is added to ep_r, effectively updating the resource occupation.

Algorithm 5.1: Offline scheduling algorithm for evolving applications.

Input: $ep^{(i)}, i = 1 \dots n_{apps}$, requested EP of the application i,
n_{nodes}, number of nodes in the system,
`fit`$(n_{nodes}, ep_{req}, ep_{res}) \rightarrow (t_s, ep_s)$, a `fit` function

Output: $ep_s^{(i)}$, scheduled EP of application i

```
1 ep_r ← empty EP ;
2 for i = 1 to n_apps do
3 |   t_s^(i), ep_x^(i) ← fit(ep^(i), ep_r, n_nodes) ;
4 |   ep_s^(i) ← delay(ep_x^(i), t_s^(i)) ;
5 |   ep_r ← ep_r + ep_s^(i) ;
```

The `fit` function takes as input the number of nodes in the system n_{nodes}, a requested EP ep_{req} and a resource EP ep_{res} and returns a time coordinate t_s and ep_x an expanded version of ep_{req}, such that $\forall t, ep_{res}(t) + \text{delay}(ep_x, t_s)(t) \leq n_{nodes}$. A very simple `fit` implementation consists in delaying ep_{req} such that it starts after ep_{res}.

Throughout the whole algorithm, the condition

$$\forall t, ep_r(t) \leq n_{nodes}$$

is guaranteed by the post-conditions of the `fit` function. Since at the end of the algorithm

$$ep_r = \sum_{i=1}^{n_{apps}} ep_s^{(i)}$$

resources will not be overflown.

5.3.2 The `fit` Function

The core of the scheduling algorithm is the `fit` function, which delays and/or expands a requested EP over a resource EP. It returns a scheduled EP, so that the sum of the resource EP and scheduled EP does not exceed available resources.

Because it can expand an EP, the `fit` function is an element of the efficiency of a schedule. On one hand, a step can be expanded so as to interleave applications, potentially reducing their response time. On the other hand, when a step is expanded, the application cannot perform useful computations, thus resources are wasted. Hence, there is a trade-off between the resource usage, the application's start time and its completion time.

In order to evaluate the impact of expansion, the proposed `fit` algorithm takes an additional parameter, the **expand limit**. This parameter expresses how many times the duration of a step may be increased. For example, if the expand limit is 2, a step may not be expanded to more than twice its original duration. Having an expand limit of 1 means applications will not be expanded, while an infinite expand limit does not impose any limit on expansion.

Base `fit` Algorithm

Algorithm 5.2 aims at efficiently computing the `fit` function, while allowing to choose different expand limits. It operates recursively for each step in ep_{req} as follows:

Algorithm 5.2: Base `fit` Algorithm

Input: n_{nodes} : number of nodes in the system,
$ep_{req} = \left\{\left(d_{req}^{(1)}, n_{req}^{(1)}\right), \ldots, \left(d_{req}^{(N_{req})}, n_{req}^{(N_{req})}\right)\right\}$, EP to expand,
$ep_{res} = \left\{\left(d_{res}^{(1)}, n_{res}^{(1)}\right), \ldots, \left(d_{res}^{(N_{res})}, n_{res}^{(N_{res})}\right)\right\}$, destination EP,
l : maximum allowed expansion ($l \geq 1$),
i : index of step from ep_{req} to start with (required for recursion, initially 1),
t_0 : first moment of time where ep_{req} is allowed to start (required for recursion, initially 0)

Output: ep_x : expanded ep_{req},
t_s : time when ep_x starts **or** time when expansion failed

`/* Base case (for terminating recursion), remaining` ep_{src} `is empty */`
if $i > N_{req}$ **then**
  $t_s \leftarrow t_0$; $ep_x \leftarrow$ empty EP ;
  return

$d \leftarrow d_{req}^{(i)}$; $n \leftarrow n_{req}^{(i)}$; `/* duration and node-count of current step */`
$t_s \leftarrow t_0$;
while *True* **do**
  if $n_{nodes} - \max(ep_{res}, t_s, t_s + d) < n$ **then**
    $t_s \leftarrow \mathrm{loc}(ep_{res}, t_s, t_s + d)$; **continue**
  if $i > 1$ **then**
    $t_{eas} \leftarrow t_s - l \cdot d_{req}^{(i-1)}$ `/* earliest allowed start of previous step */`
    if $t_{eas} > t_0 - d_{req}^{(i-1)}$ **then**
      $t_s \leftarrow t_{eas}$; $ep_x \leftarrow \varnothing$; **return**
    else if $n_{nodes} - \max(ep_{res}, t_0, t_s) < n_{req}^{(i-1)}$ **then**
      $t_s \leftarrow \mathrm{loc}(ep_{res}, t_0, t_s)$; $ep_x \leftarrow \varnothing$; **return**
  $t_s^{tail}, ep_x \leftarrow$ `fit`$(ep_{req}, ep_{res}, n_{nodes}, i + 1, t_s + d)$;
  if $ep_x = \varnothing$ **then**
    $t_s \leftarrow t_s^{tail}$; **continue**
  if $i > 0$ **then** prepend $(t_s^{tail} - t_s, n)$ to ep_x ;
  else
    prepend (d, n) to ep_x ;
    $t_s \leftarrow t_s^{tail} - d$;
  return

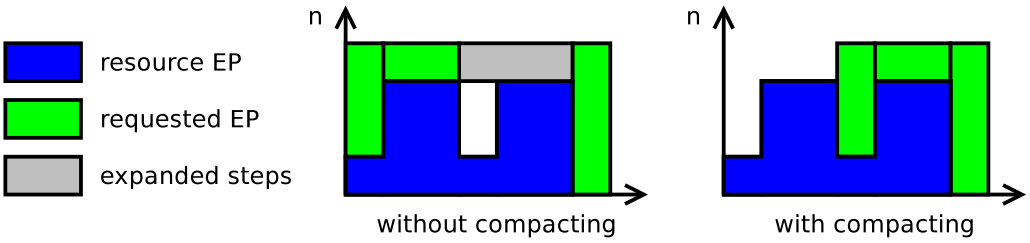


Figure 5.3: Example of post-processing optimization (compacting)

1. find t_s, the earliest time coordinate when the current step can be placed, so that n_{nodes} is not exceeded (lines 5 – 8);
2. test if this placement forces an expansion on the previous step, which exceeds the expand limit (lines 9 – 12) or exceeds n_{nodes} (lines 13 – 14);
3. recursively try to place the next step in ep_{req}, starting at the completion time of the current step (line 15);
4. prepend the expanded version of the current step in ep_x (line 18). The first step is delayed (i.e., t_s is increased) instead of being expanded (line 21).

The recursion ends when all steps have been successfully placed (lines 1–3).

Placement of a step is first attempted at time coordinate t_0, which is 0 for the first step, or the value computed on line 15 for the other steps. After every failed operation (placement or expansion) the time coordinate t_s is increased so that the same failure does not repeat:

- if placement failed, jump to the time after the encountered maximum (line 8);
- if expansion failed due to the expand limit, jump to the first time which avoids excessive expansion (computed on line 12, used on line 17).
- if expansion failed due to insufficient resources, jump to the time after the encountered maximum (computed on line 14, used on line 17);

Since each step, except the first, is individually placed at the earliest possible time coordinate and the first step is placed so that the other steps are not delayed, the algorithm guarantees that the application has the earliest possible completion time. However, resource usage is not guaranteed to be optimal.

Post-processing Optimization (Compacting)

In order to reduce resource waste, while maintaining the guarantee that the application completes as early as possible, a **compacting** post-processing phase can be applied. After a first solution is found by the base `fit` algorithm, the expanded EP goes through a compacting phase: the last step of the applications is placed so that it ends at the completion time found by the base algorithm. Then, the other steps are placed from right (last) to left (first), similarly to the base algorithm (Figure 5.3). In the worst case, no compacting occurs and the same EP is returned after the compacting phase.

The base `fit` algorithm with compacting first optimizes completion time then start time (it is optimal from expansion point-of-view), but because it acts in a greedy way, it might expand steps with high node-count, so it is not always optimal for resource waste.

5.3.3 Discussions

This section has presented a solution to the problem stated in Section 5.2. The presented strategies attempt to minimize both completion time and resource waste. However, these strategies treat applications in a pre-determined order and do not attempt to do a global optimization. This allows the algorithm to be easier to adapt to an online context for two reasons. First, list scheduling algorithms are known to be fast, which is required in a scalable RMS implementation. Second, since the algorithms treat application in-order, starvation cannot occur, i.e., an application is guaranteed to eventually be allocated the resources it requested.

One can observe that Algorithm 5.1 has some similarities to Algorithm 3.1 which implements a CooRMv1 policy. Both algorithms get as input the resource requests of applications, and output application start-times: the former returns this value as a delay in scheduled EP, while the latter returns it explicitly. Both algorithms maintain a profile of the future availability of resources and both of them iterate through the requests of the applications, given a certain order, and update the resource availability at each iteration step.

However, the two algorithms differ in several aspects. First, for Algorithm 5.1 resource requests are expressed as evolution profiles, i.e., as a variation of node-count in time, whereas for Algorithm 3.1 resource requests express the number of nodes to allocate on each cluster and the maximum duration. Second, Algorithm 3.1 also computes views (presented in Section 3.4.1) to send to the applications. Algorithm 5.1 does not compute them, as it assumes a static scenario, in which applications do not adapt to the state of the resources.

Nevertheless, we observe that the two algorithms were designed with the same purpose: to be readily usable in an RMS. This feature is exploited in the next chapter to combine the two, the CooRMv1 architecture and the offline scheduling algorithm, to devise an RMS for evolving applications.

Let us now resume to quantifying the gains that can be made when evolving applications are properly supported.

5.4 Evaluation

This section evaluates the benefits and drawbacks of taking into account evolving resource requirements of applications. It is based on a series of experiments done with a home made simulator developed in Python. The experiments are first described, then the results are analyzed.

5.4.1 Description of Experiments

The experiments compare two kinds of scheduling algorithms: `rigid`, which does not take into account evolution, and variations of Algorithm 5.1. Applications are seen by the `rigid` algorithm as non-evolving: the requested node-cound is the maximum node-count of all steps and the duration is the sum of the durations of all steps (Figure 5.4). Then, `rigid` schedules the resulting jobs in a Conservative Back-Filling (CBF)-like manner.

Five versions of Algorithm 5.1 are considered to evaluate the impact of its options: base fit with no expansion (`noX`), base fit with expand limit of 2 without compacting (`2X`) and with compacting (`2X+c`), base fit with infinite expansion without compacting (`infX`) and with compacting (`infX+c`).

Two kinds of metrics are measured: system-centric and user-centric. The five system-centric metrics considered are:

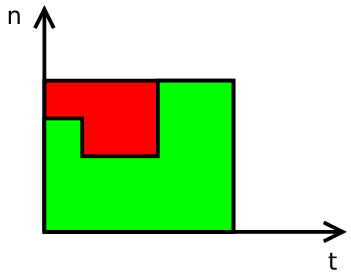


Figure 5.4: Example of how the `rigid` algorithm works. The resource area marked in green (light grey) is what the application requested. The resource area marked in red (dark grey) is wasted: the application cannot make use of it.

1. **Resource utilisation** is the resource area that has been allocated to applications, whether the application could make use of it or not. Effectively, from the schedulers point of view, these resources are busy.
2. **Effective resource utilisation** is the resource area (expressed as percent of total resources) that has been effectively used for computations by the applications.
3. **Resource waste** is the resource area (nodes×duration, expressed as percent of total resources), which has been allocated to applications, but has not been used to make computations. It can be computed as the difference between the resource utilisation and the effective resource utilization. For `rigid`, resources are wasted because they are uselessly allocated to the application, due to the algorithm not taking into account evolving resource requirements. For the other algorithms, resources are wasted due to expansion, during which, as previously said, the application cannot make use of resources.
4. **Makespan** is the completion times of the last application. This is a classical offline scheduling metric to quantify performance.
5. **Schedule time** is the computation time taken by a scheduling algorithm to schedule one test on a computer with an Intel®Core™2 Duo processor running at 2.53 GHz. This metrics is important, as it allows us to judge whether the algorithms are fast enough to be adapted to an on-line context and integrated in a real RMS implementation.

The five user-centric metrics considered are:

1. Average application completion time (`Avg. ACT`), computed for each test separately: this is a commonly scheduling metric to highlight the scheduler's performance as perceived by the user.
2. Average application waiting time (`Avg. AWT`), computed for each test separately: this allows us to have an insight of the behaviour of the system as discussed in the Analysis section.
3. The number of expanded applications (`num. expanded`), i.e., the number of applications for which at least a step was expanded, as a percentage of the total number of applications in a test: this allows us to evaluate how many users are affected by expansion.
4. By how much was an application expanded (`App Expansion`) as a percentage of its initial total duration: this allows us to observe whether few applications suffer a large expansion or many applications suffer a small expansion.
5. Per-application `waste` as a percentage of resources allocated to the application: this is similar to the system-centric resource waste, but allows us to measure waste as perceived by the user.

As we are not aware of any public archive of evolving application workloads, we created synthetic test-cases. A test case is made of a uniform random choice of the number of applications, their number of steps, as well as the duration and requested node-count of each step. We tried various combinations that gave similar results. Table 5.1 and 5.2 respectively present the results for the system- and user-centric metrics of an experiment made of 1000 tests. The number of applications per test is within $[15, 20]$, the number of steps within $[1, 10]$, a step duration within $[500, 3600]$ and the node-count per step within $[1, 75]$.

5.4.2 Analysis

Administrator's Perspective `rigid` is outperformed by all other strategies. They improve effective resource utilisation, reduce makespan and drastically reduce resource waste within reasonable scheduling time. Compared to `rigid`, all algorithms reduce resource utilization. We consider this to be a desired effect, as it means that, instead of allocating computing nodes to applications which do not effectively use them, these nodes are release to the system. The RMS could, for example, shut these nodes down to save energy.

There is a trade-off between resource waste and makespan (especially when looking at maximum values). However makespan differs less between algorithms than waste. If maintaining resources is expensive, an administrator may choose the `noX` algorithm (which does no expansion), whereas to favour throughput, she would choose `2X+c`.

User's Perspective When compared to `rigid`, the proposed algorithms always improve both per-application resource waste and average completion time. When looking at maximum values, the trade-off between expansion / waste vs. completion time is again highlighted. Algorithms which favor expansion (`infX`, `infX+c`) reduce average waiting time, but not necessarily average completion time. In other words, application are started earlier, but due to the fact that the duration of a step is increased, the completion time is not always reduced.

The results show that waste is not equally split among applications, instead, few applications are expanded a lot. Since in Cloud computing resources are payed for and, similarly, most cluster / Grid systems are subject to accounting, using the `infX` and `infX+c` algorithm, which do not guarantee an upper bound on the waste, should be avoided. Regarding algorithms which limit expansion, the benefits of using `2X+c` instead of `noX` are small, at the expense of significant per-application resource waste. Therefore, users might prefer not to expand their applications at all.

Global Perspective From both perspectives, expanding applications has limited benefit. Therefore, the `noX` algorithm seems to be the best choice. Taking into account evolving requirements of applications enables improvement of all metrics compared to an algorithm that does not take evolvement into consideration.

5.5 Conclusions

Some applications, such as Adaptive Mesh Refinement simulations, can exhibit evolving resource requirements. As it may be difficult to obtain accurate evolvement information, this chapter studied whether this effort would be worthwhile in terms of system and user perspectives. The chapter has presented the problem of scheduling fully-predictable evolving applications, for which it has proposed an offline scheduling algorithm, with various options. Experiments show that

Name	Waste (%)			Utilisation (relative)			Eff. Util. (%)			Makespan (relative)			Sch. Time (ms)		
	min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max
`rigid`	43	70	116	1	1	1	30	40	51	1	1	1	4.64	6.2	9.41
`noX`	0	0	0	.46	.58	.69	49	61	73	.49	.65	.82	11.4	24.7	55.8
`2X`	0	2	11	.47	.60	.71	50	63	75	.48	.64	.82	11.4	24.4	45.4
`2X+c`	0	ϵ	4	.46	.59	.70	53	63	75	.48	.63	.82	17.1	36.7	88.6
`infX`	0	7	22	.49	.63	.78	52	64	73	.49	.63	.78	11.4	23.4	49.2
`infX+c`	0	1	11	.46	.59	.71	55	64	74	.47	.62	.78	17.6	36	124

Table 5.1: Comparison of Scheduling Algorithms (System-centric Metrics)

Name	Avg. ACT (relative)			Avg. AWT (relative)			Num. expanded (%)			App expansion (%)			Per-app. waste (%)		
	min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max
`rigid`	1	1	1	1	1	1	0	0	0	0	0	0	0	67	681
`noX`	.42	.61	.84	.36	.55	.81	0	0	0	0	0	0	0	0	0
`2X`	.45	.61	.84	.36	.54	.80	0	22	56	0	4	76	0	2	75
`2X+c`	.44	.60	.84	.37	.54	.81	0	7	40	0	ϵ	60	0	ϵ	41
`infX`	.43	.62	.81	.27	.53	.76	0	26	62	0	19	884	0	6	360
`infX+c`	.44	.60	.81	.35	.53	.76	0	13	47	0	5	1354	0	1	119

Table 5.2: Comparison of Scheduling Algorithms (User-centric Metrics)

taking into account resource requirement evolvement leads to improvements in all measured metrics, such as resource utilization and completion time. However, the considered expansion strategies do not appear valuable.

Thanks to this work, we can now devise an RMS architecture which deals with non-predictably evolving applications. Concepts and algorithms proposed in this chapter can easily be extended for this, as highlighted in the next chapter.

CHAPTER 6

CooRMv2: An RMS for Non-predictably Evolving Applications

We can predict everything,
except the future.

John Galsworthy

This chapter presents CooRM*v2, an extension to the* CooRM*v1 architecture proposed in Chapter 3. Besides support for rigid and moldable application which it inherits from* CooRM*v1,* CooRM*v2 adds extensions which greatly widen the resource requirements that applications may express. For example, a non-predictably evolving application can make "pre-allocations" to specify its peak resource usage, then reliably allocate resources during execution as long as the pre-allocation is not outgrown. Resources which are pre-allocated but not used, can be filled by other applications, such as a malleable one. Results show that the approach is feasible and leads to a more efficient resource usage.*

6.1 Introduction

Let us start this chapter by resuming the elements described so far. Chapter 3 presented CooRMv1, a Resource Management System (RMS) to support moldable applications. Instead of submitting a static allocation of resources, as commonly done in today's RMSs, application launchers are actively involved in the scheduling decisions taken by the RMS, which allow them to optimize their resource selection. Negotiating resources only takes place before the applications start, i.e., resource allocation is not re-negotiated during their execution.

In Section 2.2 we highlighted that applications may have evolving resource requirements. For example, Adaptive Mesh Refinement (AMR) simulations may need to acquire/release new computing nodes as the mesh they operate on needs to be refined/coarsened. In Section 2.3 we have shown that such applications are currently not efficiently dealt with in the state of the art. On one hand, if static allocations are used, such as common in batch schedulers, one needs to allocate resources to meat the peak demands of the application, thus resources are wasted. On the other hand, if one uses dynamic allocations, such as common in Cloud managers, the application cannot be guaranteed that it can grow. Indeed, large-scale resource requests may be refused with an "out-of-capacity" error, thus, allocating resources on the fly may lead to the application reaching an out of memory condition. To properly support such applications, we have made a first step in Chapter 5, in which we have studied the case of fully-predictable evolving applications and determined the gains that can be made by efficiently supporting them.

In this chapter we reuse and extend previously introduced concepts, such as requests, views, evolution profiles and scheduling algorithms, and propose CooRMv2, an extension to the CooRMv1 architecture, that significantly improves the expressiveness of the RMS-application protocol, so as to support applications with dynamic resource requirements. For example, it enables efficient scheduling of evolving applications, especially non-predictable ones. It allows applications to inform the RMS about their maximum expected resource usage, using **pre-allocations**. Resources which are pre-allocated but unused can be filled by malleable applications.

The remaining of this chapter is organized as follows. Section 6.2 motivates the work and refines the problem statement by presenting a model for Non-predictably Evolving Applications (NEAs) based on an AMR application. Section 6.3 introduces CooRMv2, a novel RMS which efficiently supports various types of applications as evaluated in Section 6.4. Among them, we focus on malleable and evolving applications: Section 6.5 proposes an RMS implementation, which is used in Section 6.6 to highlight the gains that can be made by properly supporting them. Finally, Section 6.7 concludes the chapter.

6.2 A Model for Non-predictably Evolving Applications

This section motivates the present work and presents a new model of a non-predictably evolving application based on observations of AMR applications. It is used both for defining the problem statement and evaluating the solution. First, we model the evolution of the working set size for each step of the application and the duration of the steps. Second, the model is analyzed and the problem statement is refined.

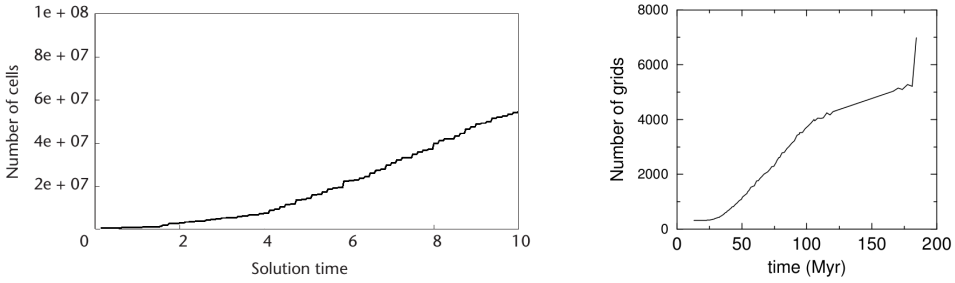


Figure 6.1: Evolution of the mesh inside AMR simulations, as shown in [79, 15]. x-axis represents the simulation (or solution) time, roughly equivalent to the step number. y-axis represents the number of cells, to which the working set size is proportional.

6.2.1 Working Set Evolution Model

Our goal is to devise a model for simulating how the data size evolves during an AMR computation. In order to have a model which is parametrizable, we first devise a "normalized" evolution profile, which is independent of the maximum size of the data or the duration of the computations. The model has to be simple enough to be easy to analyze and use, but at the same time it should resemble observed behaviors. To this end, we consider that the application is composed of 1000 steps. During step i the size of the data s_i is considered constant ($s_i \in [0, 1000]$).

Unfortunately, we have been able to find only a few papers on the evolution of the refined mesh. Most of the papers focus either on providing scalable implementations [17] or showing that applying AMR to a certain problem does not reduce its precision as compared to a uniform mesh [97]. From the few papers we found [79, 15] (see reproduction in Figure 6.1), we extracted the main features of the evolution:

– it is mostly increasing;
– it features regions of sudden increase and regions of constancy;
– it features some noise.

Using the above constraints, we derive the following model, that we call the **acceleration-deceleration model**. First, let the evolution of the size of the mesh be characterized by a velocity v_i, so that $s_i = s_{i-1} + v_i$. The application is assumed to be composed of multiple *phases*. Each phase consists of a number of steps uniform randomly distributed in $[1, 200]$. Each step i of an even phase increases v_i ($v_i = v_{i-1} + 0.01$), while each step i of an odd phase decreases v_i ($v_i = v_{i-1} \times 0.95$). Next, a Gaussian noise of $\mu = 0$ and $\sigma = 2$ is added. Finally, the profile is normalized, so that the maximum of the series s_i is 1000. The above values have been chosen so that the obtained profiles (Figure 6.2) resemble the ones found in the cited articles (Figure 6.1).

To obtain the actual, non-normalized data size profile S_i at step i, given the maximum data size $S_{\max}$, one can use the following formula:

$$S_i = s_i \times S_{\max} \tag{6.1}$$

6.2.2 A Speed-up Model

The next issue is to find a function $t(n, S)$ that returns the duration of a step, as a function of the number of allocated nodes n and the size of the data S. Both n and S are assumed constant during a step.

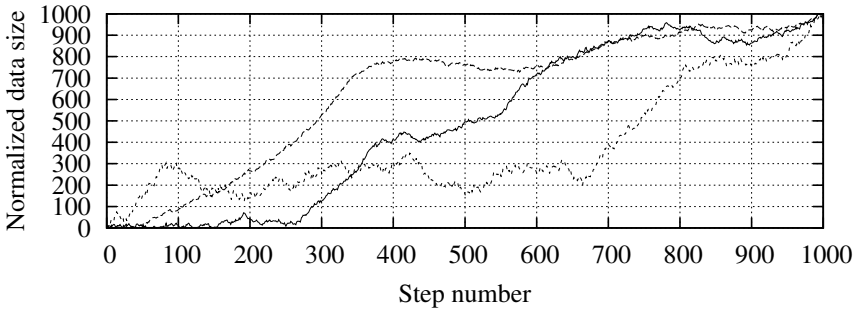


Figure 6.2: Examples of obtained AMR working set evolutions

$$\begin{aligned} A &= 7.26 \times 10^{-3}\,\mathrm{s} \times \mathrm{node} \times \mathrm{MiB}^{-1} \\ B &= 1.23 \times 10^{-4}\,\mathrm{s} \times \mathrm{node}^{-1} \\ C &= 1.13 \times 10^{-6}\,\mathrm{s} \times \mathrm{MiB}^{-1} \\ D &= 1.38\,\mathrm{s} \\ S_{max} &= 3.16\,\mathrm{TiB} \end{aligned}$$

Table 6.1: Input values to the NEA model used throughout this chapter

Instead of having a precise model (such as in [70]) which would be applicable to only one AMR application, we aimed at finding a simple formula, which could be fit against existing data. To this end, we chose to use the data presented in [77] as it shows both strong and weak scalability of an AMR application for data sizes which stay constant during one experiment. After trying several functions and verifying how well they could be fit, we found a good one to be:

$$t(n, S) = A \times \frac{S}{n} + B \times n + C \times S + D \tag{6.2}$$

The terms of this formula can be intuitively explained. Term A expresses the part of the computation which is perfectly parallelisable. Term B is the overhead of the parallelization, such as the extra effort to keep track of all the nodes in the system or the extra cost of collective communication. Term C is the time payed by each nodes per unit of data, the limiting factor for a good weak scalability. Finally, D is a constant term.

Figure 6.3 presents the result of logarithmically fitting this formula against actual data. The model fits within an error of less than 15% for any data point. The values of the parameters used in the rest of the chapter are presented in Table 6.1.

6.2.3 Analysis of the Model

Let us analyze the model and refine the problem statement. Users want to adapt their resource allocation to a target criterion. For example, depending on the computational budget of a scientific project, an application has to run at a given **target efficiency**. This allows scientists to receive the results in a timely fashion, but at the same time control the amount of used resources.

To run the above modeled application at a target efficiency e_t throughout its whole execution, given the evolution of the data size S_i, one can compute the number of nodes n_i that have to

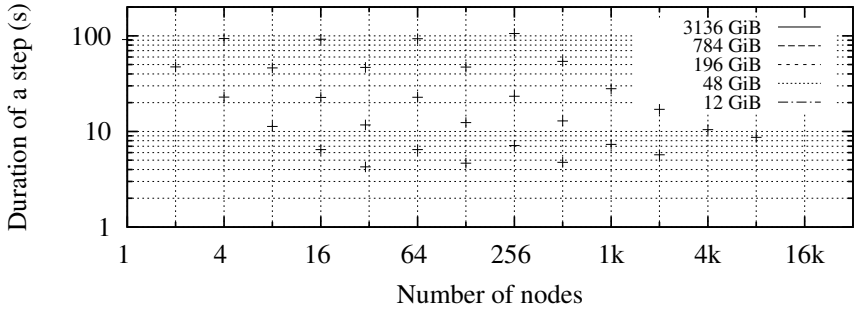


Figure 6.3: AMR speed-up model: crosses represent measured values (taken from [77]), while lines represent the speed-up model given by Equation 6.2 with the parameters in Table 6.1

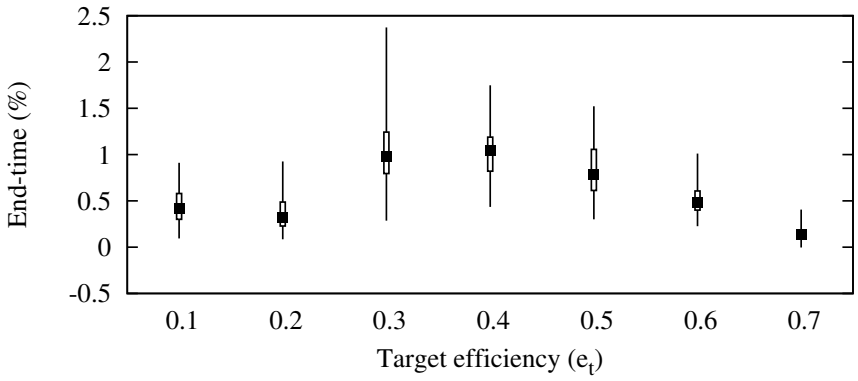


Figure 6.4: End-time increase when an equivalent static allocation is used instead of a dynamic allocation

be allocated during each step i of the computation. Let us denote the consumed resource area (node-count×duration) in this case $A(e_t)$. To achieve this in practice, one would need dynamic allocations: at the start of a step i one must allocate n_i nodes, which is computed using the value of the size of the date S_i. Thus, one does not need any a priori knowledge of the evolution of the size of the data, i.e., the values of S_i do not need to be known in advance.

However, most High-Performance Computing (HPC) resource managers do not support dynamic resource allocations, which forces a user to request a static number of nodes. Let us define the **equivalent static allocation** a number of nodes n_{eq} which, if assigned to the application during each step, leads to the consumed resource area $A(e_t)$. Computing n_{eq} requires to know all S_i a priori. Using simulations, we found that n_{eq} exists for $e_t < 0.8$. Of course, with a static allocation, some steps of the application run more efficiently, while others run less efficiently than e_t. However, interestingly, the end-time of the application increases with at most 2.5% (Figure 6.4). We deduce that, if users had an *a priori* knowledge of the evolution of the data size, they could allocate resources using a *rigid* job and their needs would be fairly well satisfied.

Unfortunately, the evolution of an AMR application is generally not known a priori, which makes choosing a static allocation difficult. Let us assume that a scientist wants to run her application efficiently. For example, she wants her application not to run out of memory, but at the same time, she does not want to use 10% more resources than $A(75\%)$. Figure 6.5 shows the range of nodes the scientist should request, depending on the maximum data size S_{max}. We observe that taking such a decision without knowing the evolution of the application in advance

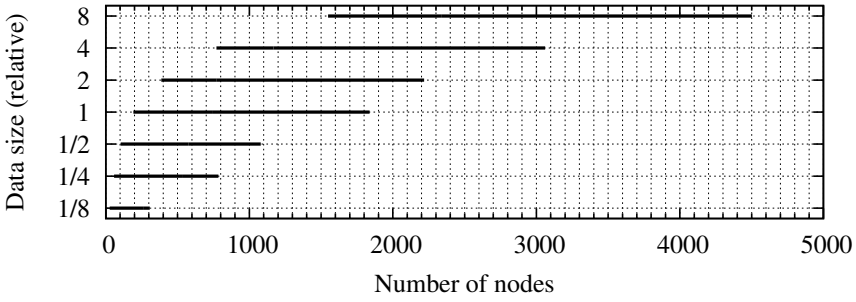


Figure 6.5: Static allocation choices for a target efficiency of 75%.

is difficult, in particular, if the behavior is highly unpredictable.

Ideally, a distinction should be made between the resources that an evolving application expects to use in the worst case and the resources that it actually uses. The unused resources should be filled by malleable applications, that can free these resources if the evolving application requests them. However, since applications might belong to different users, this cannot be done without RMS support. The next section proposes such an RMS.

6.3 The CooRMv2 Architecture

This section introduces CooRMv2, an extension to the CooRMv1 architecture presented in Chapter 3, which allows to efficiently schedule malleable and evolving applications, whether fully-, marginally- or non-predictable. First, it explains the concepts that are used in the system. Second, it gives an example of interaction between the RMS, a malleable application and an evolving application.

6.3.1 Principles

At the core of the RMS-application interaction are four concepts: requests, request constraints, high-level operations and views. Let us detail each one of them.

Requests

request

A **request** is a description, sent by applications to the RMS, of the resources an application wants to have allocated. As in usual parallel batch schedulers, a request consists of the number of nodes (node-count), the duration of the allocation and the cluster on which the allocation should take place[1]. It is the RMS that computes a start-time for each request. When the start-time of a request is the current time, node identifiers (IDs) are allocated to the request and the application is notified.

non-preemptible

preemptible

CooRMv2 supports the following three request types. **Non-preemptible** requests (also called run-to-completion, default in most RMSs [47]) ask for an allocation that once started cannot be interrupted by the RMS. **Preemptible** requests ask for an allocation that can be interrupted by the RMS, whenever it decides to, similarly to OAR's best-effort jobs [22] or to

[1] In practice, separate batch queues are used for each cluster [32].

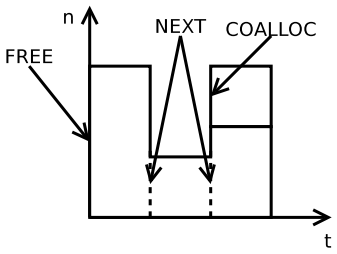


Figure 6.6: Visual description of request constraints (Section 6.3.1). The x-axis represents time and the y-axis the number of nodes.

Amazon's EC2 spot instances [120]. Preemptible requests are not automatically re-scheduled by the RMS, as this decision is left to the application[2].

In order to allow a non-predictably evolving application to inform the RMS about its maximum expected resource usage, CooRMv2 supports a third request type which we shall call **pre-allocation**. No node IDs are actually allocated as a result of submitting such a request. The goal of a pre-allocation is to allow the RMS to mark resources for possible future usage. In order to have node IDs allocated, an application has to submit non-preemptible requests inside the pre-allocation. Pre-allocated resources cannot be allocated non-preemptibly to another application, but they can be allocated preemptibly.

pre-allocation

Request Constraints

Since CooRMv2 deals with applications whose resource allocation can be dynamic, each application is allowed to submit several requests, so as to allow it to express varying resource requirements. However, the application needs to be able to specify some constraints regarding the start-times of the requests relative to each other. To this purpose, each request r has two more attributes: `relatedTo`—points to an existing request (noted r_p)—and `relatedHow`—specifies how r is constrained with respect to r_p.

CooRMv2 defines three possible values for `relatedHow` (Figure 6.6): `FREE`, `COALLOC`, and `NEXT`. `FREE` means that r is not constrained and `relatedTo` is ignored. `COALLOC` specifies that r has to start at the same time as r_p. `NEXT` specifies that r has to start immediately after r_p, with r and r_p sharing *common* resources. If r requests more nodes than r_p, then the RMS will allow the application to keep the resources allocated as part of r_p and will send additional node IDs when r is started. Conversely, if r requests fewer nodes than r_p, then the application has to decide which node IDs to release at the end of r_p.

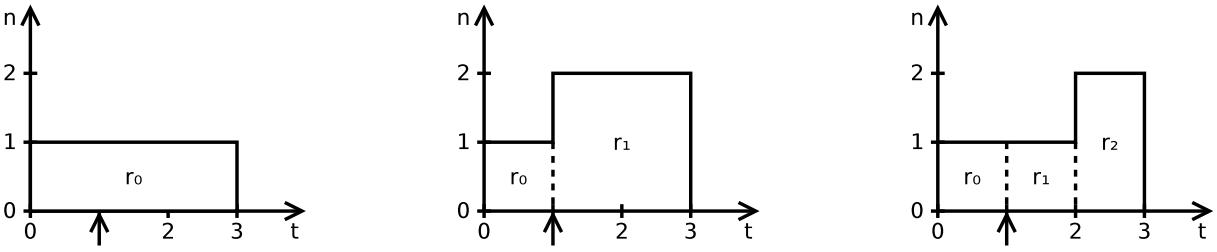
High-Level Operations

Let us now examine how applications can use the request types and constraints to fulfill their need for dynamic resource allocation.

Having defined the above request types and request constraints, the targeted applications only need two low-level operations:

– **`request()`** adds a new request into the system;

[2]According to the Oxford Dictionary, "to preempt" may mean "to interrupt or replace", e.g., "The special newscast preempted the usual television program." In contrast to its usage in OS scheduling, "to preempt" does not imply resuming at a later time.



(a) Initial state (b) Spontaneous (c) Announced

Figure 6.7: Performing an update. The x-axis and y-axis represent time and node-count, respectively. The arrow on the x-axis represents the current time, i.e., the time at which an update is performed. The area before that moment represents resources already allocated to the application, while the area after represents resources that the RMS will allocate to the application, based on the most up-to-date information it has. Dotted lines between requests represent the `NEXT` constraint.

– `done()` immediately terminates a request, i.e., its duration is set to the current time minus the request's start-time. For requests constrained with `NEXT`, the application also has to specify which node IDs it has released.

Using these two operations, an application can perform high-level operations. We shall present two of which are the most relevant: spontaneous update and announced update.

spontaneous update

A **spontaneous update** is the operation through which an application immediately requests the allocation of additional resources. It can be performed by first calling `request()` with a different node-count and a `NEXT` constraint to the current request, then calling `done()` on the current request. For example, let us assume that an application currently has 1 node allocated for another 2 hours (through request r_0), as illustrated in Figure 6.7a. If the application immediately desires to increase its allocation to 2 nodes, it can submit a request r_1 asking for 2 nodes for 2 hours, then calling `done()` on request r_0, thus obtaining the allocation in Figure 6.7b.

announced update

An **announced update** is the operation through which an application announces that it will require additional resources at a future moment of time (i.e., after an **announcement interval** has passed), thus allowing the system (and possible other applications) some time to prepare for the changes of resource allocation. It can be performed by first calling `request()` with a node-count equal to currently allocated number of nodes and a duration equal to the announcement interval, then calling `request()` with a new node-count and, finally, calling `done()` on the current request. For example, let us assume that an application currently has 1 node for another 2 hours (through request r_0 as in Figure 6.7a) and desires to increase its allocation to 2 nodes, with an announce interval of 1 hour. To this end, it shall submit a new request r_1 for 1 node for 1 hour, then a request r_2 for 2 nodes for 1 hour, finally calling `done()` on request r_0. Thus, it obtains the allocation illustrated in Figure 6.7c.

Note that, in order to guarantee that at update is successful (i.e., the RMS can actually allocate additionally requested resources), updating non-preemptible requests can only be guaranteed if they can be served from a pre-allocation.

Views

In order to allow moldable and malleable applications to adapt their resource request to the state of the resources, views, i.e., information about the future availability of the resources,

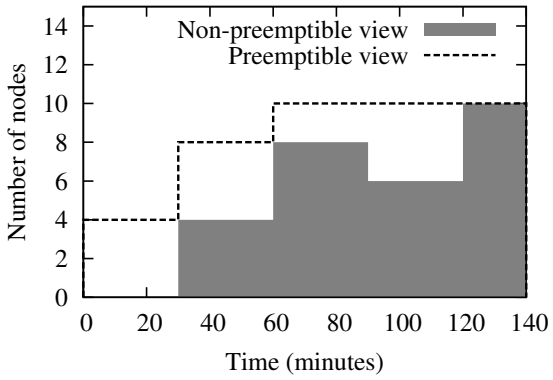


Figure 6.8: Example of views for one cluster

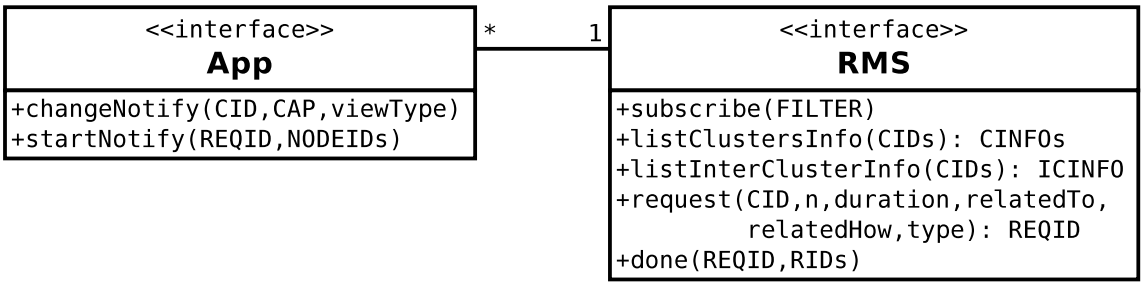


Figure 6.9: Application callbacks and RMS interface in CooRMv2

are transmitted to applications. In CooRMv2, two types of views are sent to each application i (Figure 6.8): **non-preemptive view** ($V_{NP}^{(i)}$) and **preemptive view** ($V_{P}^{(i)}$). The former, identical in purpose to CooRMv1's views, allows applications to estimate when pre-allocations and non-preemptive requests will be served. The latter allows applications to estimate when preemptive requests will be served.

non-preemptive view

preemptive view

The preemptive view is also used to signal an application, that it has to release some preemptively allocated resources, either immediately or at a future time. In CooRMv2, each application is supposed to cooperate and immediately release node IDs when it is asked to, using the above presented update operation. Otherwise, if a protocol violation is detected, the RMS kills the application's processes and terminates the session with it.

6.3.2 Interfaces

Figure 6.9 shows the interface of the RMS and the callbacks that the application must implement in CooRMv2. One can observe that they are similar to those of CooRMv1 presented in Section 3.4.3. Let us highlight the differences:

- CooRMv2 allows applications to have multiple requests in the system. The CooRMv2 `request` method adds a new request in the system, whereas in CooRMv1 it updates the application's request (if it had already submitted one).
- CooRMv2 supports multiple types of requests, while CooRMv1 only supports non-preemptible ones.
- CooRMv2 requests are somewhat simpler in that they are made of a single cluster and a single node-count, instead of a list. The application may express co-allocation over multiple

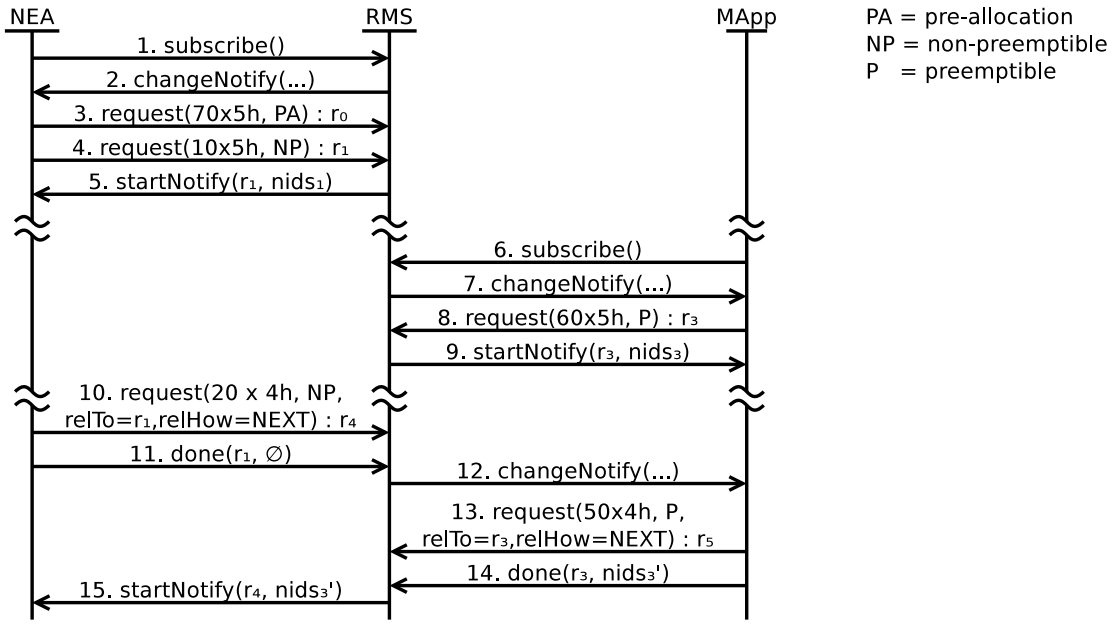


Figure 6.10: Example of an interaction between the Resource Management System (RMS), a Non-predictably Evolving Application (NEA) and a Malleable Application (MApp)

cluster using the `COALLOC` constraint.
- Since CooRMv2 allows an application to submit multiple requests, the `done` method also specifies which request to terminate. In addition, for requests constraint with `NEXT`, `done` is also used to declare which nodes to release.
- Last, CooRMv2 transmits two types of views to applications, through the `viewType` parameter in the `changeNotify` callback, whereas CooRMv1 only transmits one type.

6.3.3 Example Interaction

Let us describe the RMS-application protocol using an example. Assume there is one Non-predictably Evolving Application (NEA) and one malleable application in the system (see Figure 6.10). First, the NEA connects to the RMS (Step 1). In response, the RMS sends the corresponding views to the application (Step 2). The NEA sends a pre-allocation (Step 3) and a first non-preemptible request (Step 4), to which the RMS will immediate allocate resources (Step 5). Similar communication happens between the RMS and the malleable application, except that a preemptible request is sent (Step 6–9). As the computation of the NEA progresses, it requires more resources, therefore, it performs a spontaneous update (Step 10–11). As a result, the RMS updates the view of the malleable application (Step 12), which immediately frees some nodes (Step 13–14). Then, the RMS allocates these nodes to the NEA (Step 15).

To sum up, CooRMv2 builds upon concepts introduced in CooRMv1 and adds extensions, which, despite being relatively small, greatly increase the expressiveness of the RMS-application negotiation, as required to support malleable and evolving applications.

6.4 Application Support

This section evaluates how the CooRMv2 architecture supports all the types of application presented in Section 2.2.

Rigid Applications A rigid application sends a single non-preemptible request of the user-submitted node-count and duration. Since the application does not adapt, it ignores its views.

Moldable Applications A moldable application waits for the RMS to send a non-preemptive view, then runs a resource selection algorithm, which chooses a non-preemptible request. Should the state of the system change before the application starts, the RMS shall keep it informed by sending new views. This allows the application to re-run its selection algorithm and update its request, similarly to CooRMv1 (see Section 3.4.4).

Malleable Applications A malleable application first sends a non-preemptible request r_{min} with its minimum requirements. Next, for the extra resources (i.e., the malleable part), the application scans its preemptive view V_P and sends a preemptible request r_{extra}, which is `COALLOC`ated with r_{min}. r_{extra} can either request a node-count equal to the node-count in V_P, or it can request fewer nodes. This allows an application to select only the resources it can actually take advantage of. For example, if the malleable application requires a power-of-two node-count, but 36 nodes are available in its preemptive view, it can request 32 nodes, leaving the other 4 to be filled by another application. During execution, the application monitors V_P and updates r_{extra} if necessary.

Evolving Applications A **fully-predictably** evolving application sends several non-preemptible requests linked using the `NEXT` constraint. During its execution, if from one request to another the node-count decreases, it has to call `done` with the node IDs it chooses to free. Otherwise, if the node-count increases, the RMS sends it the new node IDs.

A **Non-predictably Evolving Application (NEA)** first sends a pre-allocation request (whose node-count is discussed below), then sends an initial non-preemptible request. During execution, the application updates the non-preemptible request as the resource requirements change. Since such updates can only be guaranteed as long as they happen inside a pre-allocation, such an application may adopt two strategies: sure execution or probable execution.

The application can opt for a **sure execution**, i.e., uninterrupted run-to-completion, if it knows the maximum node-count (n_{max}) it may require. To adopt this strategy, the size of the pre-allocation must be equal to n_{max}. In the worst case, n_{max} is the whole machine.

Otherwise, if the application only wants **probable execution**, e.g., because pre-allocating the maximum amount of required resources is impractical, then the application sends a "good-enough" pre-allocation and optimistically assumes never to outgrow it. If at some point the pre-allocation is insufficient, the RMS cannot guarantee that updates will succeed. Therefore, the application has to be able to checkpoint. It can later resume its computations by submitting a new, larger pre-allocation. Note that, in this case, the application might further be delayed as the RMS might have placed it at the end of the waiting queue.

If multiple NEAs enter the system, one of the two following cases may occur. Either their pre-allocations are small enough to fit inside the system simultaneously, in this case the NEAs are launched at the same time, or their pre-allocations are too large to fit simultaneously, in which case the one that arrived later will be queued after the other. In both cases, the RMS is

able to guarantee that whenever one of the NEAs requests an update inside its pre-allocation, it can actually be served.

Note that, resources that are pre-allocated to a NEA, but not effectively used, cannot be non-preemptively allocated to another application. For example, these resources cannot be allocated to a rigid or a moldable application. However, a malleable application can preemptively allocate such resources and use them for its malleable part. This approach is used in Section 6.6, to highlight the gains one may obtain by properly supporting evolving and malleable applications.

6.5 An Example RMS Implementation

In order to focus our evaluation on evolving and malleable applications, we need an RMS implementation. This section presents an example implementation that we have used for our prototype. A CooRMv2 RMS has three tasks:

(i) for each application i compute a non-preemptible ($V_{NP}^{(i)}$) and a preemptible view ($V_{P}^{(i)}$),
(ii) compute the start-time of each request and
(iii) start requests and allocate node IDs.

In this section, we shall only give a high-level description of the implementation. The full specification is quite intricate, therefore, the reader is asked to find the details, such as used data structures and algorithms, in Appendix A.1.

The RMS runs a scheduling algorithm whenever it receives a `request` or `done` message. In order to coalesce messages coming from multiple applications at the same time and reduce system load, the algorithm is run at most once every **re-scheduling interval**, which is an administrator-chosen parameter. Initial testing showed that a value of 1 second gives a system that is very reactive, yet scales well.

re-scheduling interval

The scheduling algorithm goes as follows. Applications are sorted in a list based on the time the applications send the `subscribe` message. First, pre-allocation requests are scheduled using Conservative Back-Filling (CBF) [86]. Next, inside these pre-allocations, non-preemptible requests are scheduled. The remaining resources are used to schedule preemptible requests.

Regarding preemptive views and the start-time of preemptive requests, they are computed so as to achieve equi-partitioning. However, should an application not use its partition, other applications are allowed to fill it in. We shall call this **"smart" equi-partitioning**.

"smart" equi-partitioning

The proposed scheduling algorithm has a linear complexity with respect to the number of requests in the system. We have developed a Python implementation (with more compute-intensive parts in C++) which is able to handle approximately 500 requests/second on a single core of an Intel® Core™2 Duo CPU @ 2.53 GHz. The prototype is composed of approximately 1600 SLOC[3] of Python code and 1000 SLOC of C++ code.

The above implementation is just one possible implementation and can easily be adapted to other needs. For example, the amount of resources that an application can pre-allocate can be limited, by clipping its non-preemptible view.

6.6 Evaluation with Evolving and Malleable Applications

This section evaluates the gains that can be made with the CooRMv2 architecture if evolving and malleable applications are properly supported. It uses the previously presented RMS

[3] as measured by David A. Wheeler's SLOCCount utility [148]

implementation. First, the application and resource models are presented. Next, the impact on evolving and malleable applications is analyzed.

The evaluation is based on a discrete-event simulator, which simulates both the CooRMv2 RMS and the applications. To write the simulator, we have first written a real-life prototype RMS and synthetic applications. Then, we have replaced remote calls with direct function calls and calls to `sleep()` with simulator events.

Regarding the figures in this section, we use candlesticks to represent minimum, lower quartile, median, upper quartile and maximum.

6.6.1 Application and Resource Model

As other works which propose scheduling new types of applications [108], we shall only focus on the malleable and evolving applications, which can fully take advantage of CooRMv2's features. Therefore, we shall not evaluate our system against a trace of rigid jobs [129] as we have done in Chapter 3 and as is commonly done in the community. Nevertheless, as shown in Section 6.4, the proposed system does support such a usage.

For the evaluation, two applications are used: a non-predictably evolving AMR and a malleable Parameter-Sweep Application (PSA). Let us present how the two applications behave with respect to CooRMv2.

AMR Application

A synthetic AMR application is considered, which behaves as a non-predictably evolving application with sure execution (see Section 6.4). The application knows its speed-up model, but cannot predict how the working set will evolve (see Section 6.2). Knowing only the current working set size, the application tries to target an efficiency of 75% using spontaneous updates.

The user submits the application by trying to "guess" the node-count n_{eq} of the equivalent static allocation (defined in Section 6.2). The application uses this value for its pre-allocation. Inside this pre-allocation, non-preemptible allocations are sent/updated, so as to keep the application running at the target efficiency.

Since the user cannot determine n_{eq} a priori, her guesses will be different from this optimum. Therefore, we introduce as a simulation parameter the **overcommit factor**, which is defined as the ratio between the node-count that the user *has chosen* and the node-count that the user *should have chosen*, i.e., the best node-count assuming a posteriori knowledge of the application's behavior.

overcommit factor

Parameter-Sweep Application (PSA)

Inspired by [102], we consider that this malleable applications is composed of an infinite number of single-node tasks, each of duration d_{task}. As described in Section 6.4, the PSA monitors its preemptive view. If more resources are available to it than it has currently allocated, it updates its preemptible request and spawn new processes. If the RMS requires it to release resources immediately, it kills a few tasks then updates its request. The computations done so far are lost. We measure the **PSA waste** which is the number of node-seconds wasted in such a case.

PSA waste

If the RMS is able to inform the PSA some time in advance that resources will become unavailable, then the PSA waits for some tasks to complete, afterwards it updates its request to release the resources on which the completed tasks ran. No waste occurs in this situation.

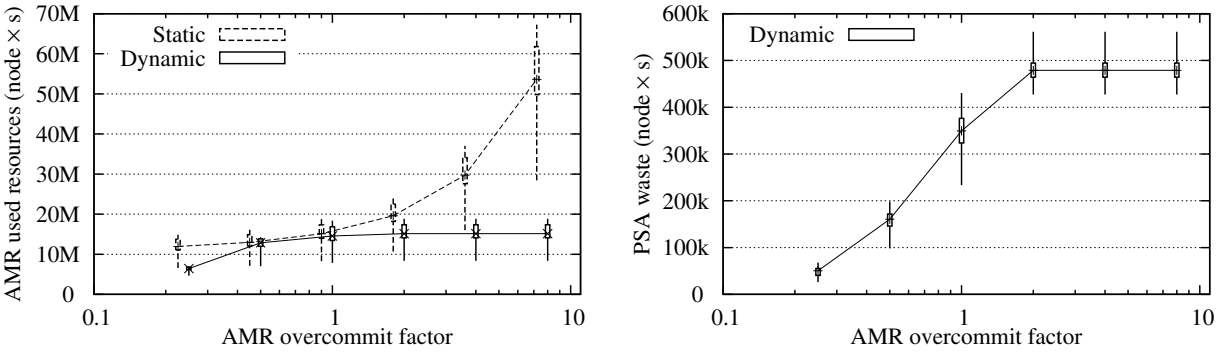


Figure 6.11: Simulation results with spontaneous updates

Resource Model

We assume that resources consist of one single, large homogeneous cluster of n nodes. The re-scheduling interval of the RMS is set to 1 second, to obtain a very reactive system.

6.6.2 Scheduling with Spontaneous Updates

In this set of experiments, we evaluate whether CooRMv2 solves the initial problem of efficiently scheduling NEAs.

Two applications enter the system: one AMR and one PSA (PSA_1 with $d_{task} = 600\,\text{s}$). Since we want to highlight the advantages that supporting evolving applications in the RMS brings, we shall schedule the AMR application in two ways: **dynamic**, in which the application behaves as described above, and **static**, in which the application is forced to use all the resources it has pre-allocated.

Regarding the resources, the number of nodes n is chosen so that the pre-allocation of the AMR application can succeed. By observing the behaviour of the application during some preliminary simulations, we concluded that, for an overcommit factor $\phi \in [1/8, 8]$, the AMR application needs $n = 1400 \times \phi$ nodes.

Figure 6.11 shows the results of the simulations. Regarding the amount of resources effectively allocated to the AMR (AMR used resources), the figure shows that as the overcommit factor grows, a static allocation forces the application to consume more resources. This happens because, as the user chooses more nodes for the application than optimal, the application runs less efficiently. Moreover, the application is unable to release the resources it uses inefficiently to another application.

In contrast, a dynamic allocation allows the AMR application to cease resources it cannot use efficiently. Therefore, as the overcommit factor grows, a dynamic allocation allows the application to maintain an efficient execution. Note however, that thanks to the pre-allocation, if the resource requirements of the application increase, the application can request new resources and the RMS guarantees their timely allocation. Since, CooRMv2 mandates that preemptible resources be freed immediately, the impact on the AMR application is negligible.

In this experiment, the RMS allocates the resources that are not used by the AMR application to the malleable PSA. However, since the AMR application does spontaneous updates and the RMS cannot inform the PSA in a timely manner that it has to release resources, the PSA has to kill some tasks, thus, computations are wasted (Figure 6.11). Nevertheless, we observe that the amount of resources wasted is smaller that the resources which would be used by an inefficiently

running AMR application.

The PSA waste first increases as the overcommit factor increases, then stays constant after the overcommit factor is greater than 1. This happens because once the AMR application is provided with a big enough pre-allocation, it does not change the way it allocates resources inside it.

To sum up, this experiment shows that, if there are non-predictably evolving applications in the system, properly supporting them can lead to a more efficient resource usage. Nevertheless, PSA waste is non-negligible, therefore, the next section evaluates whether using announced updates makes sense to reduce this waste.

6.6.3 Scheduling with Announced Updates

This section evaluates whether using announced updates can improve resource utilization. To this end, the behavior of the AMR application has been modified as follows. A new parameter has been added, the **announce interval**: instead of sending spontaneous updates, the application shall send announced updates (see Section 6.3.1). The node-count in the update is equal to the node-count required at the moment the update was initiated. This means that the AMR receives new nodes later than it would require to maintain its target efficiency. If the announce interval is zero, the AMR behaves as with spontaneous updates. Otherwise, the application behaves like a marginally-predictably evolving application.

announce interval

For this set of experiments, we have set the overcommit factor to 1. The influence of this parameter has already been studied in the previous section.

Using announced updates instead of spontaneous ones has several consequences. On the negative side, the AMR application is allocated fewer resources than required, thus, its end-time increases[4] (see Figure 6.12). On the positive side, this informs the system some time ahead that new resources will need to be allocated, thus, it allows other applications more time to adapt.

Figure 6.12 shows that the PSA waste is decreasing as the announce interval increases. This happens because, in order to free resources, the PSA can gracefully *stop* tasks when they complete, instead of having to *kill* them. Once the announce interval is greater than the task duration d_{task}, no PSA waste occurs.

Let us discuss the **percent of used resources**, which we define as the resources allocated to applications minus the PSA waste as a percentage of the total resources. Figure 6.12 shows that announced updates do not generally improve resource utilization. This happens because the PSA cannot make use of resources if the duration of the task is greater than the time the resources are available to it. In other words, the PSA cannot fill resources unused by the AMR if the "holes" are not big enough: these "holes" are either wasted on killed tasks (Figure 6.13a) or released to the system (Figure 6.13b). However, the higher the announce interval, the more resources the PSA frees, so that the RMS can either put them in an energy saving mode, or allocate them to another application, as shown in the next section.

percent of used resources

We observe that the percent of used resources has two peaks, when the announce interval is 300 s and 600 s. In these two cases, a "resonance" occurs between the AMR application and the PSA. The "holes" left by the AMR application have exactly the size required to fit PSA tasks. On the contrary, when the announce interval is just below the duration of the task (i.e., 550 s) the percent of used resources is the lowest, as the PSA has to kill tasks just before they end.

[4]Instead of sending an update with the currently required node-count, the application could use predictions and send a larger announced update, at the expense of an increased resource usage. Devising such prediction algorithms and studying the compromise between end-time and resource usage is outside the scope of this Thesis.

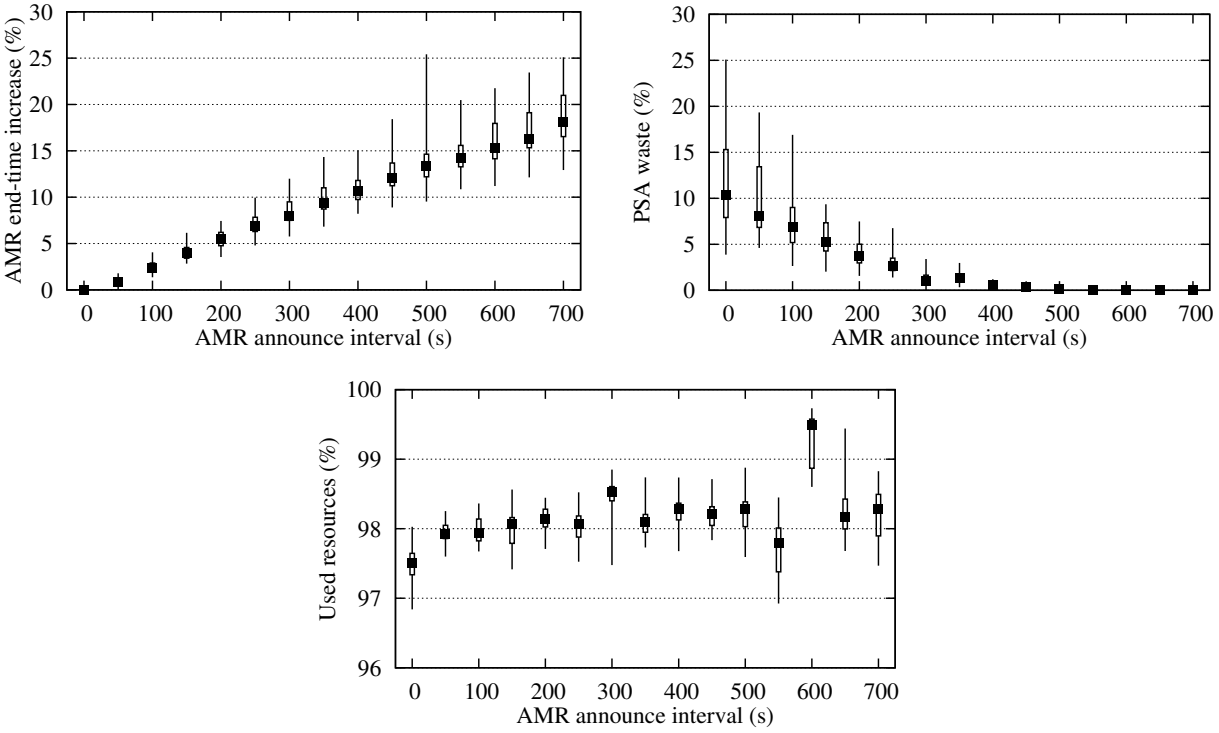


Figure 6.12: Simulation results with announced updates

The resource utilisations shown in the graphs are higher than typically found in today's computing centers, however, it has been shown that these values can easily be obtained using a mixture of moldable and malleable applications [59], but without NEAs.

6.6.4 Efficient Resource Filling

In the previous section, we have observed that while announced updates reduce PSA waste, they do not improve resource utilization. This happens because the PSA's task duration is greater than the "holes" left by the AMR application (Figure 6.13b). In this section, we evaluate whether CooRMv2 is able to allocate such resources to another PSA, with a smaller task duration, in order to improve resource utilization.

To this end, we add another PSA in the system with a smaller task duration (PSA_2 with $d_{task} = 60\,\text{s}$). By default, the RMS does equi-partitioning between the two PSA applications, however, when one of them signals that it cannot use some resources, the other PSA can request them (Figure 6.13d).

"strict" equi-partitioning

To highlight the gain, we compare this experimental setup with an RMS which implements **"strict" equi-partitioning** (Figure 6.13c): instead of allowing one PSA to fill the resources that are not requested by the other PSA, the RMS presents both PSAs a preemptible view, which makes them request the same node-count.

Figure 6.14 shows the percent of used resources. For readability, only medians are plotted. Experiments show that CooRMv2 is able to signal PSA_2 that some resources are left unused by PSA_1. Resource utilization is improved, because PSA_2 is able to request these resources and do useful computations on them.

To sum up, CooRMv2 is able to efficiently deal with the applications it has been designed

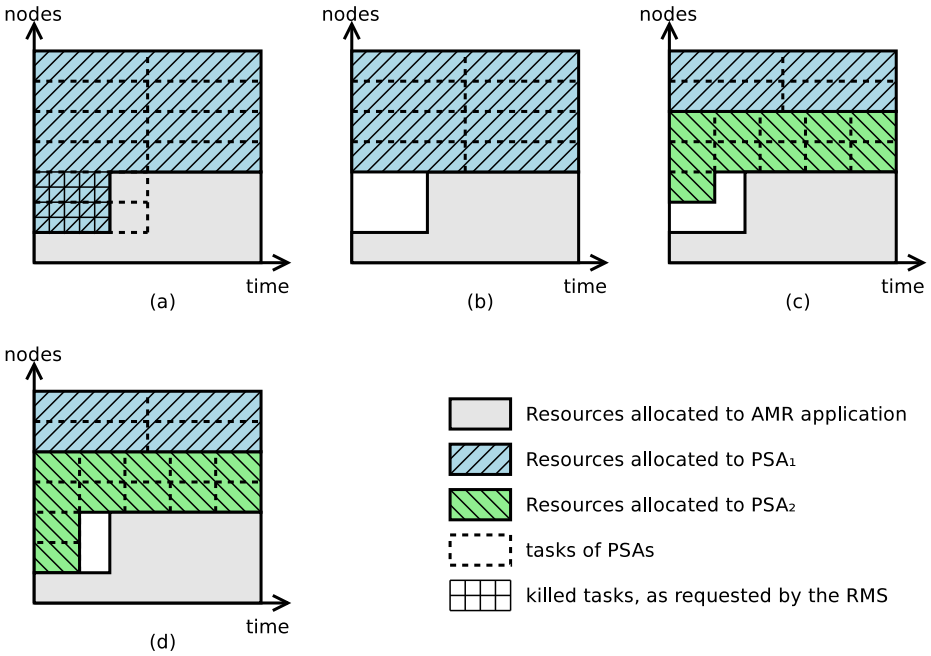


Figure 6.13: Graphical illustration of resource filling. The figures incrementally represent: (a) one AMR application using spontaneous updates and one PSA, (b) using announced updates, (c) two PSAs with "strict" equi-partitioning, (d) two PSAs with "smart" equi-partitioning.

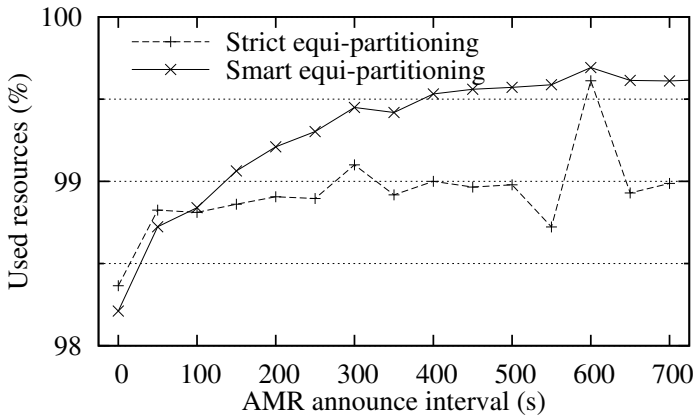


Figure 6.14: Simulation results for two Parameter-Sweep Applications (PSAs)

for: evolving and malleable. On one hand, evolving applications can efficiently deal with their unpredictability by separately specifying what their maximum and what their current resource requirements are. Should evolving applications be able to predict some of their evolution, they can easily export this information into the system. On the other hand, malleable applications are provided with this information and are able to improve resource usage.

6.7 Conclusion

This chapter presented CooRMv2, a centralized RMS architecture, which extends CooRMv1 presented in Chapter 3. In addition to rigid and moldable application support which it inherited, CooRMv2 supports malleable and evolving applications, ranging from fully-predictable to non-predictable. Thus, we reached the goal of this Thesis and devised an RMS architecture which supports all the types of applications that we have presented in Section 2.2.

The new resource management features enabled by CooRMv2 were evaluated using two types of synthetic applications: non-predictably evolving, based on the model of an AMR application, and malleable, based on the model of a parameter-sweep application. The results showed that resources can be more efficiently allocated with proper support for these types of applications.

Up to this chapter, we have devised resource management abstractions which operated at a low level: the applications negotiated with the RMS in terms of computing nodes and duration. In the next chapter, we study how the collaboration between two systems, working with higher-level resource management abstractions, such as tasks to be executed, can be bettered for improved resource management.

CHAPTER 7

Fair Scheduling of Optional Computations in GridRPC Middleware

Și numai cu cei mari egalitate
vrem. [We want equality only
with the greater ones.]

Grigore Alexandrescu

In this last contribution chapter, we continue our quest to improve resource management by promoting collaboration between applications and Resource Management Systems (RMSs). In previous chapters, we proposed lower-level abstractions which were mostly inspired by batch schedulers. In this chapter, we start our research from existing systems which propose higher-level resource management abstractions, then study how cooperation could be improved.

To this end, we took GridTLSE as an application and DIET as an RMS. We present a use-case which is badly supported by these systems and we identify an underlying research problem: scheduling optional computations, i.e., computations which are not critical to the user, but their completion would improve results.

The main contribution of this chapter is to propose a generic, distributed master-client architecture, which fairly schedules optional computations. The architecture has been implemented in the DIET GridRPC middleware. Real-life experiments show that several metrics are improved, such as user satisfaction, fairness and the number of completed requests. Moreover, the solution is shown to be scalable.

7.1 Introduction

In previous chapters, we have devised systems which improve resource management, by promoting the cooperation between applications and RMSs. The proposed systems were designed with the purpose of allowing for a great flexibility in expressing resource requirements. The resulting interfaces are mostly inspired by batch scheduler and are relatively low-level. Thus, with this great flexibility comes a greater burden for the application developer. For example, in order to run a Parameter-Sweep Application (PSA) in CooRMv2, one must not only write code to interact with the RMS, but also code to schedule tasks inside the allocation.

In this chapter, we study how resource management could be improved, by promoting cooperation at a higher level: the applications not longer negotiate with the RMS in terms of computing nodes and duration, but in terms of tasks to be executed. To this purpose, we start from two existing systems, which previously did not cooperate: GridTLSE is used as a motivating application, while DIET is used as a distributed RMS. We are given a use-case which is badly supported by the two systems and aim at extending them so as to find a satisfactory solution. In doing so, we identify an underlying research problem: **scheduling optional computations**, i.e., computations which are not critical to the user, but their completion would improve results. The challenge is to find a collaborating, distributed resource management architecture, which fairly schedules optional computations.

The use-case has been chosen due to its generality. Indeed, a similar issue arises in *sampling-based uncertainty analysis* [57], such as *Monte Carlo* experiments, which is a widely used method for testing the numerical stability of complex simulations. Applications range from aerospace engineering to validating nuclear power plant design. At its core, the method consists in varying input parameters and studying the changes in the output parameters. The larger the number of tested input parameters, the better the quality of the results are. Hence, obtaining good results in a timely manner would consist in testing as many parameters as possible before a given deadline.

Unfortunately, many system which aim at easing the submission of PSA to distributed computing infrastructures, such as Condor [110], DIANE [85] and DIRAC [31], propose interfaces which force a user to submit a pre-determined number of computation requests. This is cumbersome to do in advance for applications which have optional computations. If too many requests are submitted, then the resources are over-utilized, preventing *other* scientists from completing their simulations in due time. If too few requests are submitted, resources might be left idle, thus, the user lost an opportunity to improve her results. Hence, the user faces the difficulty of choosing the number of requests to submit.

In remaining of this chapter we present a motivating use-case in Section 7.2, which is then formalized into a problem statement in Section 7.3. Section 7.4 proposes a generic solution, the DIET-ethic master-client architecture, which is then implemented in the DIET GridRPC middleware. Section 7.5 presents the evaluation using real-life experiments and shows that several metrics can be improved, such as user happiness and fairness. Also, the architecture is shown to be scalable. Finally, Section 7.6 concludes the chapter.

7.2 A Motivating Use-case

This section briefly presents the GridTLSE project and a motivating use-case called multiple threshold pivoting.

GridTLSE [3] is a joint project initiated by several research laboratories (CERFACS, IRIT,

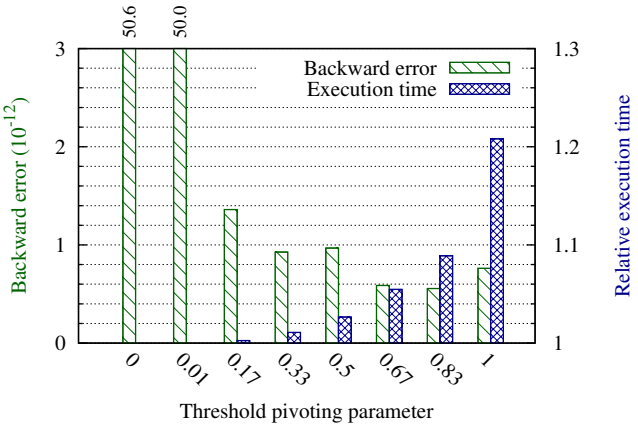


Figure 7.1: Example of the results output by a direct solver

LaBRI, LIP) and industrial partners (CNES, CEA, EADS, EDF, IFP) in France. Its main goal is to design an expert site that provides easy access to many direct solvers for sparse linear systems, allowing their comparative analysis. The site assists users in choosing the right solver for their problems, appropriate values for the control parameters of the selected solver and the best computer architecture to run the solver on. GridTLSE also serves as a testbed for experts in sparse linear algebra. A computational Grid is used to execute all the runs arising from user requests, which is accessed using GridRPC. In production deployments, DIET is used as a GridRPC middleware. The GridTLSE project has started in 2002 years and is currently being used by 157 users.

Let us detail how direct solvers work. Solving a linear system of the form $Ax = b$ usually consists of three steps: analysis, factorisation and solving. During the factorisation step of an asymmetric matrix, two matrices are computed, a lower triangular matrix L and an upper triangular matrix U, such that $LU = A$. In order to preserve the numerical stability (e.g., avoiding the division by a small number) pivoting can take place. One way to select the pivot is to choose a diagonal entry according to a given threshold $thr \in [0, 1]$. The selection of the threshold is important and for some values, the result of the solution of the linear system can be very bad.

Figure 7.1 gives an example of running a direct solvers with the same input matrices A and B, but with different thresholds. The **backward error**, an indication of the accuracy of the solution (smaller is better), is computed after having determined x as $||b - Ax||/||b||$. One may observe that for the given example, a threshold $thr = 0.83$ gives the smallest backward error. In contrast, the default threshold for the selected solver, $thr = 0.1$, gives a very inaccurate result. Unfortunately, the relationship between the backward error and the threshold is highly dependent on the input matrix A and cannot be determined in advance. Therefore, there is no analytic method to determine the optimal threshold.

backward error

Returning to GridTLSE, an often requested feature is to allow a user to run as many factorizations as possible, with different thresholds, until a given deadline is reached. If the deadline is too tight or resources are highly loaded, the system should test at least a predefined number of thresholds (for example, three). This would allow the user to find the solution with the smallest backward error, given the time constraints. In other words, the user needs to submit some **mandatory requests**, which need to be solved whether the deadline is due or not, and a (potentially large) number of **optional requests**, which the user would like to have computed,

mandatory requests

optional requests

but are not as useful as to wait for their completion past the deadline.

It is up to the users to voluntarily choose how many of their requests are mandatory and how many are optional. In small data centers with few users, optional requests might be used as a way to be fair towards workmates. In large data centers, users may be charged differently for mandatory and optional requests, thus, declaring their requests as optional may allow them to more efficiently use their quota. Nevertheless, devising a cost model is outside the scope of this paper.

Regarding the classification in Section 2.2, this application is malleable. However, the problematic is different from what has been done in literature. Related work [16, 34, 43, 59, 102, 108] aims at improving some kind of metric, but, in the end, the computations that are executed are always the same, no matter how resources are allocated to the application. In contrast, to our knowledge, we are the first to propose for a malleable application to adapt to the state of the resources by also changing the computations it does. If many resources are available, the application produces more accurate results, whereas if few resources are available, the application guarantees at least a minimum accuracy.

Using classical resource management abstractions, such as GridRPC, it is difficult to choose the number of optional requests to submit to the platform. Obviously, if too few requests are submitted, the number of tested thresholds is suboptimal. If too many are submitted, the optional requests of other users might not have a chance at getting executing, which would be **unfair**. Even worse, the platform might be so overloaded that the other users might need to wait past their deadlines for the completion of the mandatory requests, which makes them **unhappy**.

Now that we presented a motivating use-case, let us give a more formal definition of the targeted problem.

7.3 Problem Statement

This section formalizes the problem. First, the resource and user models are described. Second, the metrics that the system has to optimize are defined.

7.3.1 Resource Model

Let the platform be composed of n_R resources, which are *homogeneous* (a computation request has the same execution time on any resource), *static* (resources are neither added nor removed during execution) and *reliable* (resources do not fail).

This model is somewhat simple but still applicable in many cases. For example, it fairly well approximates production-level multi-cluster systems such as the Decrypthon grid [7]. Nevertheless, these assumptions will be relaxed in future work.

7.3.2 User/Application Model

Let the platform be used by n_U users. A user i enters the system at time $t_0^{(i)}$ (which is not known in advance) and needs to solve at least $n_{min}^{(i)}$ (also called **mandatory** requests) and at most $n_{max}^{(i)}$ requests (including mandatory and **optional** requests).

deadline

The user sets a "tentative" **deadline** $d^{(i)}$ which acts as follows. If at time $d^{(i)}$ all mandatory requests are completed, the remaining optional requests are cancelled and the user exits the system. Otherwise, the user waits until all mandatory requests are completed, even if this means waiting past the deadline. In the latter case, optional computations can still be executed until

the last mandatory request finishes. In other words, the hard deadline is equal to the maximum between the user-provided deadline and the last completion time of the mandatory requests.

To completely characterize the workload, the execution times need to be modeled. We consider that the requests of user i are homogeneous, having the same **execution time** $T^{(i)}$. This is a reasonable approximation for the targeted use-cases (see Figure 7.1), as well as many parameter-sweep applications [102]. However, execution times are not known in advance.

7.3.3 Metrics

To evaluate how well a system deals with a workload, the following metrics are of interest: the number of unhappy users, unfairness and the number of completed requests.

The **number of unhappy users** is the number of users who did not complete their mandatory requests before their deadline $d^{(i)}$. These users had to wait additionally, after the tentative, user-provided deadline. Ideally, the number of unhappy users should be 0, provided the workload permits such a solution. number of unhappy users

Before defining fairness, let us introduce some helper notations. For each user i, the amount of *deserved* resources $r^{(i)}_{deserved}$ (i.e., the amount the system *should* allocate the user) is computed as follows. The set of users in the system as a function of time is piece-wise continuous. Let $U^{(j)}$ be the set of users in the system during the time-slot $\left[S^{(j)}, S^{(j+1)}\right)$. Theses values can be computed as follows:

$$S^{(j)} = \text{j}^{th} \text{ element of } \left\{t_0^{(i)}, \forall i\right\} \cup \left\{d^{(i)}, \forall i\right\} \text{ ordered ascending} \tag{7.1}$$

$$U^{(j)} = \left\{i \in \text{set of users} \,\middle|\, \left(t_0^{(i)} \leq S^{(j)}\right) \wedge \left(S^{(j+1)} \leq d^{(i)}\right)\right\} \tag{7.2}$$

The resource area (number of resources times duration) available during that time-slot is divided equally among the users in $U^{(j)}$:

$$r^{(i)}_{deserved} = \sum_{j:i \in U^{(j)}} \frac{n_R \cdot \left(S^{(j+1)} - S^{(j)}\right)}{\#U^{(j)}} \tag{7.3}$$

Next, for each user i, the satisfaction $s^{(i)}$ is defined as the amount of resources the system allocated her $r^{(i)}_{allocated}$ over the amount of resources she deserved $r^{(i)}_{deserved}$. A satisfaction $0 \leq s^{(i)} < 1$ means that the user i was allocated fewer resources than deserved, while $s^{(i)} > 1$ means that the user i was allocated more resources than deserved. Ideally, the satisfaction of all users should be 1, i.e., they are allocated as many resources as deserved.

Having all prerequisites, let **unfairness** be defined as the difference between the maximum and the minimum among the user satisfactions: unfairness

$$\text{unfairness} = \max_i s^{(i)} - \min_i s^{(i)} \tag{7.4}$$

We refrained from defining unfairness as an average, for the same reasons as stated in [118]: users tend to be more sensitive to fairness than to performance, thus, the former should be more like a law or a guarantee that the system is heavily penalized for breaking. Ideally, unfairness should equal 0.

Finally, the **number of completed requests** is a performance-oriented metric, computed as the sum of all the requests (mandatory and optional) belonging to any user that have completed. number of completed requests

To sum up, we aim at finding a system, which minimizes the number of unhappy users, minimizes unfairness and maximizes the number of completed requests, in this order. Note that, the three presented metrics can only be computed a posteriori, after all users exited the system.

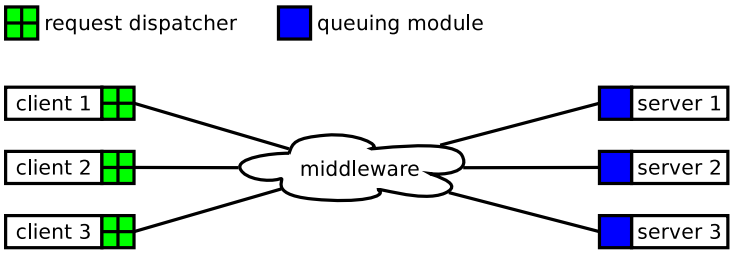


Figure 7.2: DIET-ethic architecture

7.4 DIET-ethic

In this section, the DIET-ethic platform for fair scheduling of optional computations is presented. First, the architecture is described in an abstract, implementation-independent manner. Second, our implementation of DIET-ethic in the production-level DIET GridRPC middleware is detailed.

7.4.1 DIET-ethic Extension

DIET-ethic is an extension over a client-server architecture. The clients (representing the users of the system) are resource consumers that generate computational requests, while the servers are resource providers, doing computations on behalf of the clients. Clients and servers are connected through a middleware, that implements a discovery mechanism.

Before describing DIET-ethic, let us highlight some design choices. First, we chose to keep the server-side scheduling algorithm simple and make servers unaware of the user deadlines. This choice has been taken based on a similar rationale as in the case of CooRMv1 (see Chapter 3) or CooRMv2 (see Chapter 6): clients, which are under the control of the user, are able to evolve their scheduling algorithms separately from the functionality offered by the server. For example, a future extension of DIET-ethic might schedule workflows containing optional computations, without requiring administrators to upgrade the servers.

Second, the number of requests stored in the platform have to be minimized. This is important, since, for the targeted use-cases, any single user could fill the whole platform alone with her optional requests. Therefore, having a system in which all users submit all their requests to the system would clearly not scale.

Let us now describe the DIET-ethic architecture consisting of a server-side queuing module and a client-side request dispatcher (Figure 7.2), that are described in the next sections.

Server-side Queuing Module

When a request arrives at a server, it is not immediately executed, but is added to a local queue. The queue is regularly checked to determine which requests should be started, cancelled (erased from the queue) or killed (prematurely terminated after having been started). The **scheduling algorithm** orders requests according to the following five rules (rules presented first take precedence):

1. mandatory requests are ordered before optional ones; this ensures that the number of unhappy users is minimized;
2. started requests are ordered before waiting ones; otherwise resources might be wasted as requests are killed and computations completed so-far are lost; this improves the number

of completed requests, paying a small price on fairness;

3. mandatory requests are ordered by submit time, i.e., the First-Come First-Serve (FCFS) scheduling strategy is used; this allows users who arrived first in the system to get a better chance at completing their mandatory requests before the deadline, thus decreasing the number of unhappy users;
4. requests are ordered by the amount of resources allocated to their user so far; this ensures server-local fairness;
5. for users having the same amount of allocated resources (such is the case when users enter the system at the same time) a request is chosen randomly; this ensures global fairness, as each server most likely chooses to execute the request of a different client.

For example, a waiting mandatory request is ordered before a started optional request, a waiting optional requests is ordered after a started optional request.

When a request gets to the front of the queue it is started. Otherwise, when sorting moves a request from the front of the queue, it is killed. For example, if a user submits a mandatory request to a server which is currently executing an optional request, the latter is killed. Requests can also be cancelled or killed on the client's demand.

Client-side Request Dispatcher

On the client-side, a custom request dispatcher is required. It works in three phases: setup, monitoring and cleanup.

The **setup phase** starts with a resource discovery, asking the middleware to return at most $n_{max}^{(i)}$ servers. Next, mandatory requests are dispatched to discovered servers, for example, in round-robin. Finally, one optional request is submitted to each discovered server.

In the **monitoring phase**, the client enters an event-loop. It stays in this phase until all mandatory requests are completed and the deadline has not expired. When an optional request is completed, the client submits a new optional request to the server which executed the former request. As a result, as long as the user is in the system, provided $n_{max}^{(i)}$ is large enough, each server has at least one optional request in its queue, ready to be executed.

Finally, in the **cleanup phase**, the client cancels all requests that have been submitted but not yet completed. Note that, this phase is entered when all mandatory requests have been completed, thus, only optional requests need to be cancelled.

Remarks

DIET-ethic is **generic** and can be applied to any client-server architecture. For example, it can be applied to a video conversion platform, accessed through REST or RPC-XML, in which users tag certain videos as mandatory and others as optional. In this paper, we are interested in improving resource management in HPC data centers, therefore, we chose to implement it on top of an existing GridRPC middleware.

7.4.2 Implementation on Top of DIET

The above concepts have been implemented in the DIET GridRPC middleware, which has been described in detail in Section 2.3.3. Besides the addition of a request dispatcher and a queuing module, two types of changes had to be made: GridRPC API extensions and client-server protocol extensions.

The GridRPC API has been extended with three functions:

1. The `grpc_discover` function allows clients to discover servers in the system. The function accepts two parameters: the name of the service and the maximum number of servers to return.
2. The `grpc_function_handle_set_optional` function allows clients to declare a request as optional.
3. The `grpc_wait_any_until` function has been implemented, which is an extension to `grpc_wait_any` allowing a client to specify a timeout. The function block until either a GridRPC request completes or the timeout expires, similarly to the POSIX™ `select` or `poll` system calls [61].

Regarding the client-server communication protocol, it has been extended so that each request is tagged with the user it belongs to. This is necessary information allowing servers to ensure fairness among different users. The current implementation assumes that users are honest and do not attempt to falsify their identity. A future implementation will include authentication, thus making sure that requests can be securely associated with a user.

Thus, we obtained a production-ready implementation which we shall use to evaluate the DIET-ethic architecture in the following section. The contribution consist of approximately 1000 SLOC of C++ code.

7.5 Evaluation

This section evaluates the proposed architecture. First, we show the gains that can be made with DIET-ethic by comparing it to a standard system which has not been designed to support optional computations. Second, we show that the architecture is scalable. We would like to highlight that all experiments have been done on a **real platform**. Finally, we discuss the benefits that can be observed by an end-user.

7.5.1 Gains of Supporting Optional Computations

Let us consider increasingly complex scenarios and make a comparative analysis between DIET-ethic and a system without optional computation support. For the latter, we used the DIET middleware as it was before our contribution: the Service Daemons (SeDs) serve incoming requests using the FCFS policy, without distinguishing mandatory from optional requests. On the client-side, we implemented the following behaviour. When a client i enters the system it has to blindly choose $n^{(i)}_{submit}$, a number between $n^{(i)}_{min}$ and $n^{(i)}_{max}$, representing the number of requests to submit. First, it submits $n^{(i)}_{min}$ mandatory and $n^{(i)}_{submit} - n^{(i)}_{min}$ optional requests (in this order). Then, it waits for the mandatory requests to finish. If the deadline has not expired, it sleeps until the deadline is reached. Finally, it gathers the results of all completed requests and cancels the remaining optional requests submitted to the system. To simplify the analysis of the results, all clients "guess" the same value n_{submit}. Let us call this system the **legacy system**.

Before detailing the scenarios, let us present the common methodology. The platform consists of 1 Master Agent (MA) and $n_R = 10$ Service Daemon (SeD). The SeDs only implement a sleep service, i.e., the service itself consumes no CPU nor network bandwidth. The platform is used by $n_U = 10$ identical clients with their parameters chosen as follows: to make experiments as useful as possible, but at the same time reduce the time it takes to complete them, we have chosen to "compress" the time: 8 hours are normalized to 100 s. Therefore, we set the execution

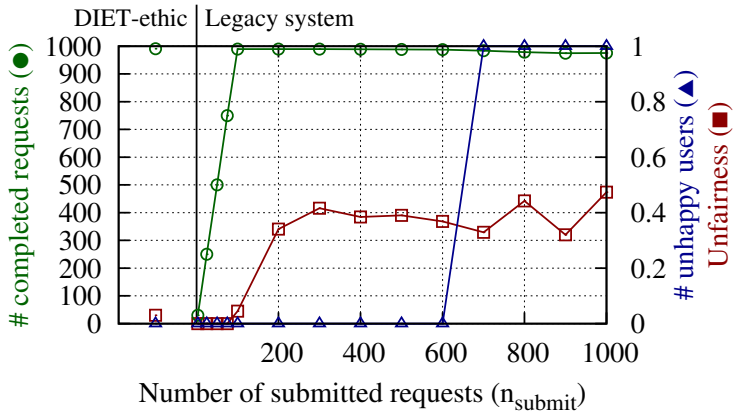


Figure 7.3: Results: night-time simultaneous submissions scenario

time $T^{(i)} = 1\,\text{s}$ and the deadlines $d^{(i)} = 100\,\text{s}$. These values are large enough compared to the time scheduling decisions take, yet small enough so that the time of experiments be reasonable. Next, we set the number of mandatory requests $n_{min}^{(i)} = 3$ and the number of total requests $n_{max}^{(i)} = 1000$. These parameters have been chosen so that the following conditions be met:

- there is a solution which makes all users happy;
- each user can generate enough optional computations to fill all resources.

The above conditions are the ones in which our system is the most interesting to be studied. Otherwise, if the number of mandatory computations is too high, the platform has no choice but to schedule them in a FCFS fashion, being forced to make some users unhappy. Also, if the number of optional requests is too low, its fairness properties cannot be highlighted.

The metrics we are interested in are those presented in Section 7.3.3. All measurements have been done at least 10 times and, since we found deviations to be small, we only plot the median to make graphs more readable.

The figures in this section are structured as follows. The x-axis is divided in two. On the left, the metrics obtained with DIET-ethic, which fairly schedules optional requests, are plotted. There is a single data point, since DIET-ethic requires no manual tuning and automatically choses the number of requests to submit. On the right, the metrics are plotted for the legacy system, which is does not fairly schedule optional requests. There are several data points on the x-axis, since it requires manually choosing the number of submitted requests n_{submit}.

The systems are compared in 5 different, increasingly complex scenarios, which are described in detail in the next sections.

Night-time Simultaneous Submissions

Let us start with a simple scenario. Users want to do computations during the night, so that their results would be ready in the morning and could be analyzed during the workday. Effectively, users enter the computation platform in the evening, just before leaving work and have a tentative deadline for the next morning, when they arrive at work. In our experiments, we can model them by setting the same arrival-time $t_0^{(i)} = 0$ and deadline $d^{(i)} = 100$ for all users.

Figure 7.3 shows that DIET-ethic managed to find a solution with no unhappy users, with good (almost ideal) fairness, while maximizing the number of completed requests. Regarding the legacy system, one observes that, if n_{submit} is small, the resources are not filled with com-

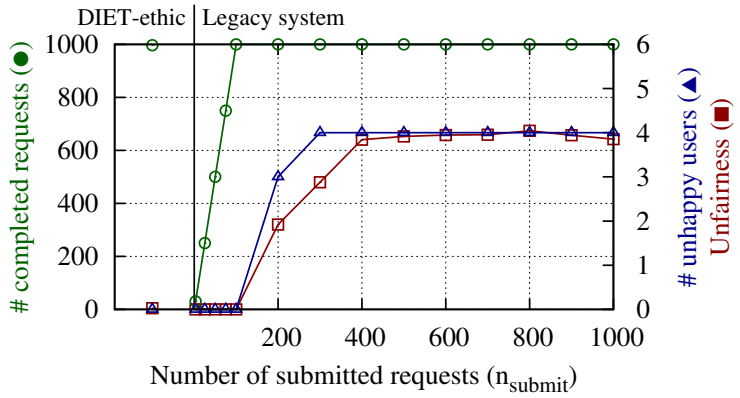


Figure 7.4: Results: night-time consecutive submissions scenario

putation requests, thus, the number of completed requests is suboptimal. However, if n_{submit} is high enough, on average, the legacy system behaves fairly well. This is due to an experiment artifact that, since all users enter the system at precisely the same moment of time, their requests favorably interleave, therefore, the FCFS policy is mostly finding the optimal solution: all mandatory requests are started first, followed by the optional requests.

Night-time Consecutive Submissions

However, in production systems, users never enter the platform at exactly the same time. In fact, the time the users enter the system might be quite different: some people leave work earlier, others later. Their deadlines are about the same, since those who leave work earlier, often come earlier the next day. To model this scenario, we insert a very small inter-arrival gap $t_0^{(i)} = i \cdot 0.1\,\mathrm{s}$ and keep $d^{(i)} = 100\,\mathrm{s}$.

Figure 7.4 shows that, unless all users guess the ideal solution (that of each user submitting exactly $n_{ideal} = 100$ requests), the legacy system either does not manage to optimize the number of completed requests (if $n_{submit} < n_{ideal}$) or makes users unhappy (if $n_{submit} > n_{ideal}$). The latter happens because the FCFS policy fills resources with optional requests of users who arrived early in the system. Therefore, the mandatory requests of users who arrive later start later and can be delayed past the tentative deadline. A similar observation applies to fairness: the FCFS policy favors users who enter the system early, instead of trying to balance requests equally among them.

In contrast, since DIET-ethic distinguishes mandatory and optional requests, it makes sure that mandatory requests have priority over optional ones. Also, instead of favoring users who arrive early, resources are allocated equally among the optional requests of the users. In the end, DIET-ethic improves fairness up to 150 times and behaves as if all users chose the ideal number of requests to submit, but without having to guess it.

Day-time Submissions with Regular Arrivals

Let us pass on to a different scenario: day-time submissions. During the day, the users to not enter the system during a short time interval, but are separated by significant inter-arrival times. Let us start with a simple scenario, in which the inter-arrival time between consecutive clients is constant.

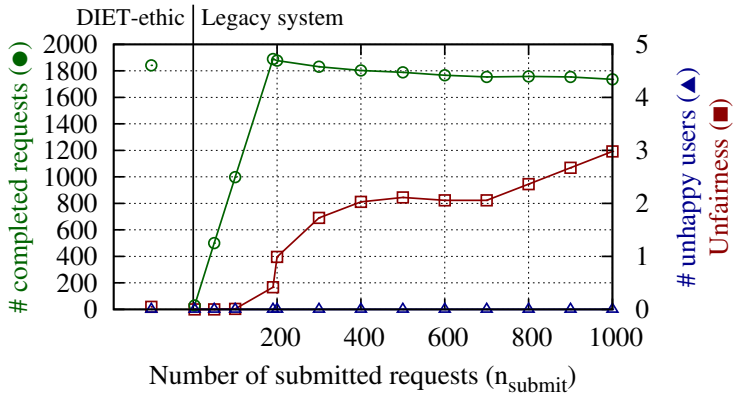


Figure 7.5: Results: day-time scenario with regular arrivals

To model this, we took the night-time consecutive scenario and set $t_0^{(i)} = i \cdot 10\,\text{s}$. Except arrival times, all other experimental parameters are kept the same. When contrasting the two scenarios, the main difference is that, in the previous one all clients have entered the system before the mandatory requests of the first client are completed. In contrast, in the current scenario, mandatory requests of a client are already completed by the time the next client arrives in the system. One could say that the previous scenario resembles an off-line scheduling problem (i.e., all requests are known in advance, decisions can be taken in advance), whereas the current scenario resembles an on-line scheduling problem (i.e., the system needs to adapt to arriving requests).

Figure 7.5 shows the results for this scenario. One can observe that the legacy system behaves best for $n_{submit} = 190$. No users are unhappy, unfairness is low and the number of completed requests is the highest, even when compared to DIET-ethic. The latter happens because, when a new client enters the system, DIET-ethic immediately starts its mandatory requests, killing optional request if necessary. Therefore, some started computations are interrupted, thus reducing the number of completed requests. Nevertheless, one observes that DIET-ethic's solution has a lower (near-ideal) unfairness and the number of completed requests stays competitive to the legacy system (1842 vs. 1890, i.e., $\approx -2.5\%$).

However, on a real platform guessing the best number of requests $n_{submit}^{(i)}$ each client i should submit is difficult, as it depends on a number of factors, such as the number of resources, arrivals and requirements of other users. Some of this information is unknown at the time a client enters the system. When looking at the results for $n_{submit} \neq 190$, one observes that the legacy system is outperformed by DIET-ethic. As in previous scenarios, if fewer requests are submitted, then the number of completed requests is suboptimal. Deviating in the other direction, if too many requests are submitted then unfairness increases.

Interestingly, as the number of submitted requests increases, the number of completed requests slightly decreases. This happens because, increasing n_{submit} also increases the probability that clients launched earlier still have executing requests in the system by the time their deadline is reached. These requests are killed, thus are not completed.

In contrast, DIET-ethic auto-tunes itself and finds a good solution, without requiring the user to guess a good number of requests to submit.

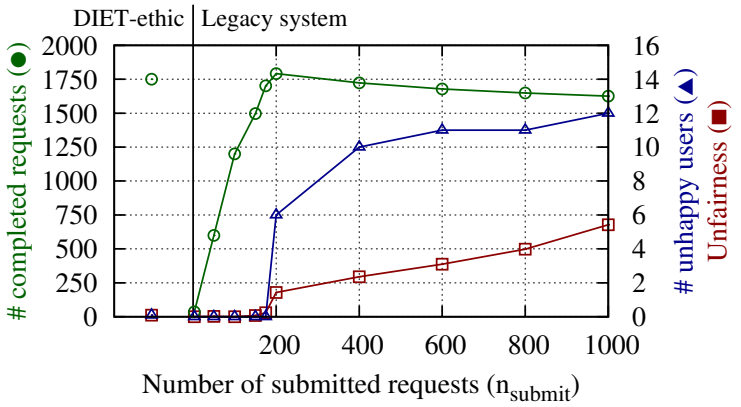


Figure 7.6: Results: day-time scenario with irregular arrivals

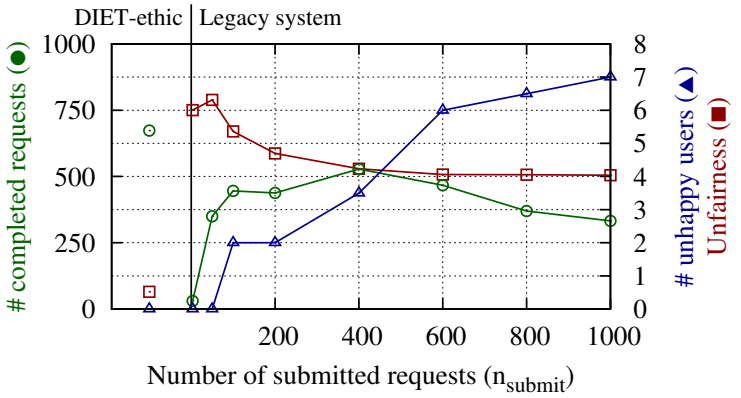


Figure 7.7: Results: irregular arrivals and random execution times

Other Day-time Scenarios

In order to make sure that the proposed system is well-behaved in more realistic cases, let us present two more scenarios.

In the first scenario, we drop the assumption that the inter-arrival time between consecutive clients is constant. Instead, the arrivals are assumed to obey the well-known and widely recognized diurnal cycle [130]. To model this, we took the polynomial P proposed in [18], scaled its input to the $[0, 100]$ seconds interval and used the output value as follows: at each second t, the probability of launching a new client is $P(t)/40$. The division by 40 was made so as to keep the average number of clients in the system equal to 10, similarly to the previous scenarios. Figure 7.6 presents the experimental results of this scenario. The same observations can be made as in previous sections. There is a value of n_{submit} in which the legacy system behaves well, however, this value is difficult to compute a priori in a real system. In contrast, DIET-ethic obtains almost the same values for the targeted metrics without requiring to manually choose this parameter.

In the second scenario, we give each client a different execution time by choosing $T^{(i)}$ uniform randomly in $[0.125, 8]$. Requests generated by the same client are still homogeneous and the arrivals are considered to obey the diurnal cycle as in the paragraph above. The results of this scenario, presented in Figure 7.7, show that no matter how n_{submit} is chosen, the legacy system

Cluster	# nodes	Configuration
capricorne	2	2×AMD Opteron 246 @ 2.0 GHz
sagittaire	69	2×AMD Opteron 250 @ 2.4 GHz

Table 7.1: Grid'5000 deployment for scalability experiment

Unhappy users	0	ideally 0
Unfairness	0.70	ideally 0
Number of completed requests	13605	out of 13800

Table 7.2: Results of scalability experiment

cannot optimize all targeted metrics. Indeed, due to the large variation in execution times, each user i should choose a different number of requests to submit $n^{(i)}_{submit}$. In practice, when a user i enters the system, optimizing $n^{(i)}_{submit}$ would require complete information about future arriving users (or at least accurate estimations) which is unlikely to be available. In contrast, DIET-ethic auto-tunes itself and manages to minimize the number of unhappy users, minimize unfairness and maximize the number of completed requests.

7.5.2 Scalability

In order to assess the scalability of our solution and measure the overhead, we have designed the following experiment. We reserved the **whole** Lyon site on the Grid'5000 experimental platform [12] (Table 7.1). The set of nodes has been divided into three: 1 client node, 1 MA node and 69 SeD nodes.

Experiments has been done as follows: first, 1 MA has been deployed on the MA node. Second, SeDs have been deployed on each core of the bi-processor SeD nodes, totalling $n_R = 2 \times 69 = 138$ SeDs. Finally, on the client node, $n_U = 100$ client have been launched simultaneously with the parameters: $n^{(i)}_{min} = 3$, $n^{(i)}_{max} = 10000$, $d^{(i)} = 100$, $T^{(i)} = 1$. As a reference, traces from the Grid Workload Archive [64] contains less that 100 users per day.

Besides the metrics presented in Section 7.3.3, we measure the CPU utilization on the client node, the MA node and one of the SeD nodes. On the client node, we have been careful to filter out CPU usage due to process creation and destruction. The SeDs implement a simple "sleep" service, which does not do any computations. Therefore, the measured CPU usage represents the **overhead** our system incurs for managing computational requests.

Table 7.2 reports the metrics of Section 7.3.3. There are zero unhappy users and unfairness is low. Also, the number of completed requests is 13605, which, compared to the maximum of 13800 requests that could have been completed by 138 SeDs in the 100 s deadline, represents 98.6%. Hence, we conclude that the system managed to optimize the targeted metrics, even under stress conditions.

Let us now take a deeper look at the behaviour of the system and study the CPU usage overhead during the experiments. Figure 7.8 presents the CPU usage as a function of time on the client node, the MA node and one of the SeD nodes. First, let us observe that the measured CPU usage before launching the clients and after the clients finished is below 1%, thus indicating a low measurement noise. Next, one can clearly distinguish the three phases of the clients (Section 7.4.1): setup, monitoring and cleanup.

The *setup* phase takes about 13 s from $t = 0$ to $t = 13$. During this phase the CPU usage

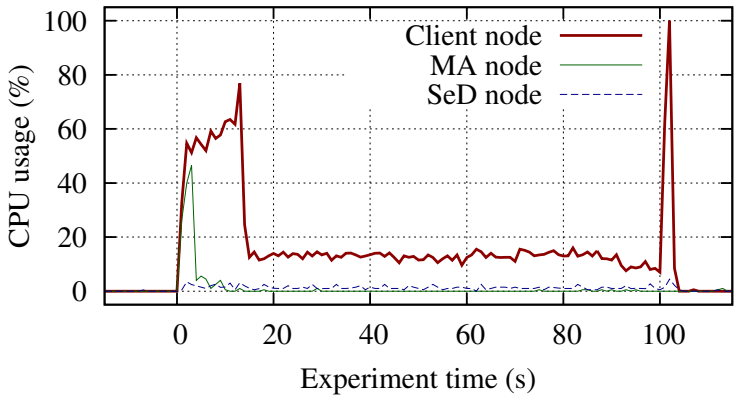


Figure 7.8: System overhead: CPU usage for sleep requests

on the client node is somewhat high, due to all clients simultaneously discovering SeDs, then submitting requests to all of them. Discovery is handled by the MA, on which one observes a peak in CPU usage. On the SeD node, the CPU usage is negligible, being less than 3%.

During the *monitoring* phase, which lasts from $t = 13$ to $t = 100$, one observes that the CPU usage stays relatively low on all nodes. The most stressed is the client node, as the client applications have to submit 138 requests/s to keep SeDs busy. The MA is idling, since it is not participating in this phase, while the CPU usage on the SeD node is negligible.

Finally, during the *cleanup* phase, the CPU usage on the client node tops at 100%, as cancellation requests are sent by all clients, simultaneously to all SeDs. Again, the MA is idling, while the CPU usage on the SeD node is still fairly low ($< 5\%$).

Note that the scalability experiment is extreme. The clients arrive simultaneously, which is unlikely in real systems and the execution time is short (1 s). For comparison, the average job inter-arrival time on the LHC Computing Grid is 5 s, whereas the average run-time is 2.5 h [64].

To sum up, the CPU usage on the SeD nodes is negligible when a sleep service is used. This means that our system involves **low overhead** and that SeDs can perform useful computations. The CPU usage on the client node is high during the setup and cleanup phase, nevertheless it managed to generate enough requests, so as to keep SeDs busy. The MA did not prove to be a bottleneck in these experiments.

7.5.3 End-User Perspective: Integration with GridTLSE

Now that we evaluated DIET-ethic with a synthetic "sleep" service, we return to the motivating application (Section 7.2) and discuss the advantages that a GridTLSE user observes. In close collaboration with the GridTLSE team at IRIT, Toulouse, France, we have implemented a working prototype in order to test an initial version of DIET-ethic against GridTLSE. In this section, we first give an overview of GridTLSE from the end-user's perspective, then describe the changes that we have made.

scenario

As presented in Section 7.2, the purpose of GridTLSE is to allow the comparative analysis of direct solvers. To this end, GridTLSE proposes the user a list of **scenario**s, which are in fact abstract workflows. A typical user would select a scenario and input parameter values, such as providing a list of matrices and solvers, thus instantiating a concrete workflow called an **expertise**. The workflow operates with **experiment**s, which are characterized by a *matrix* (the A term in a linear system $Ax = B$), the name of a *solver*, *control parameters* for the solver

expertise

experiment

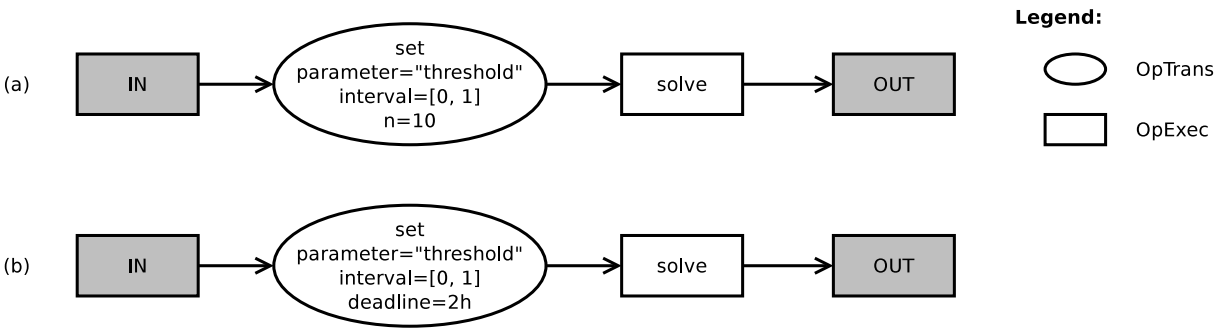


Figure 7.9: GridTLSE scenario for multiple threshold pivoting:
(a) with manually-chosen thresholds, (b) with automatically-chosen thresholds

and measured *metrics*. Two types of operations may be applied on experiments: `OpTrans` and `OpExec` (see example in Figure 7.9).

`OpTrans` represent computationally *inexpensive* tasks, such as modifying control parameters. They are executed on a single node, usually the one which hosts the GridTLSE website. An `OpTrans` receives as input a list of experiments and outputs a potentially different list of experiments. It may change some parameters of the experiments, it may create new experiments or it may delete experiments.

In contrast, `OpExec`s represent computationally *expensive* tasks, which need to run on a Grid. For example, an `OpExec` may initiate the transfer of a matrix onto Grid resources, launch a solver and store some metrics, such as the execution time, the number of FLOPS, the backward error, etc. Initially, `OpExec`s were designed to receive a list of input experiments and output the same list of experiments with some parameters and metrics potentially updated.

Let us now describe the changes that we have made. GridTLSE is composed of an upper layer, ALTO, responsible for interpreting expertises and issuing computation requests generated by `OpExec`s. The resulting requests are executed by the lower layer, BASSO, responsible for communicating with a GridRPC middleware and performing computations. Before our contribution, ALTO expected that each computation request returned one result. This is no longer a valid assumption if the number of computations to be executed depends on the status of the platform. Therefore, ALTO has been extended to accept a variable number of results for every request sent to BASSO. The code dealing with `OpExec`s has been changed so that new experiments are generated to store the metrics of the additional results.

Next, a new `OpTrans` has been implemented. It sets the necessary control parameters to signal that a certain parameter should be modulated depending on the resources (Figure 7.9). To keep a clear separation of concerns, ALTO is unaware of the state of the platform, how the number of requests is chosen or how the requests are dispatched to available resources. This is in contrast to "classical" `OpTrans` which are entirely implemented in ALTO itself. Instead, the semantics of the resource-dependent `OpTrans` are implemented in the lower layer, BASSO, which is responsible for dealing with such low-level, resource-related issues.

As to BASSO, the following behavior has been implemented. When it receives a computation request, it first checks whether the request contains a resource-dependent parameter. If so, the computation request is cloned, assigning each time a different value to the resource-dependent parameter. Next, a client-side dispatcher and a server-side queue module is used, as described in Section 7.4.1. If no resource-dependent parameter is found in the request, BASSO will use the legacy behavior, i.e., it will use the GridRPC API to invoke the request and return a single

result.

In the end, the user can more easily run experiments because the platform auto-tunes parameters for her. Previously, when a user wanted to solve a linear system, she had to choose the number of thresholds to test. This was done using a combination of trial-and-error and guesswork. The user had to know approximately the number of available resources and the time a solve takes. If the user overestimated the number of thresholds, she would manually cancel the computations. Conversely, the user underestimated, she would have to manually relaunch the computations with a new set of thresholds.

In contrast, thanks to our contribution, a working prototype showed that the user only has to specify the minimum and the maximum number of thresholds to test, and a deadline. The system automatically tests as many thresholds as to respect the constraints imposed by the users.

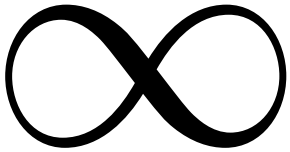
7.6 Conclusion

This chapter continued the goal of the Thesis, to promote cooperation between applications and resource management systems for a more efficient resource usage. In contrast to previous chapters, we started by studying existing systems, which work with higher-level resource management abstractions, such as tasks to execute and services which are able to execute them. We took a use-case which was previously inefficiently supported, fairly scheduling optional computations.

As a solution, we proposed a generic architecture, called DIET-ethic, that is applicable to any client-server architecture. It allows clients to tag requests as either mandatory or optional and submit them for execution to servers in a scalable manner. DIET-ethic is especially well suited to be implemented in GridRPC middleware, such as GridSolve [115] or Ninf [109]. Evaluation has been done using an implementation on top of the production-level DIET middleware. Real-life experiments showed that several metrics can be improved, for example, user unhappiness can be reduced to 0 and unfairness can be decreased up to 150 times. Additionally, the architecture was shown to be scalable. Finally, DIET-ethic has been integrated into a real application, the GridTLSE project, which has been modified to fully support multiple threshold pivoting.

Part IV

To Conclude

CHAPTER 

Conclusions and Perspectives

… to boldly go where no man has gone before.

Star Trek: The Next Generation

This chapter concludes the Thesis, reminding the addressed problem, highlighting the contributions and opening up perspectives.

∞.1 Conclusions

Scientific research is increasingly relying on numerical simulations. Indeed, it has found applications in cosmology, molecular biology, electromagnetic compatibility, particle physics and many other disciplines. This in turn generates more demands for computing power.

Supplying this demand has become increasingly difficult. Since 2005, due to technological limitations related, amongst others, to energy usage and heat dissipation, increasing the clock rate of a computing processor seems to have reached a cap. Moreover, previously-used speed-up methods, such as out-of-order execution, instruction-level parallelism, branch prediction, etc., seems to give diminishing returns. Therefore, parallel processing needs to be used to increase the raw computing power of a system.

As more and more petascale machines are being build and exascale is expected to be achieved in 2020, computing hardware is becoming increasingly complex. It presents parallelism at several levels, each level being composed of heterogeneous computing units and a heterogeneous interconnect. Therefore, harvesting the hardware's computing power at peak efficiency is very challenging.

The aim of the Thesis was to make a more efficient usage of High-Performance Computing (HPC) platforms by improving resource management at the system level. Indeed, whether we are talking about Super, Cluster, Cloud, Grid or Sky Computing, computing resources are rarely used by a single user. Therefore, a component of the platform, called the Resource Management System (RMS) is responsible for multiplexing resources, in space and time, among multiple users. Doing so properly is of uttermost importance to ensure that the user applications are running efficiently.

The context of the Thesis was presented in Chapter 2. First, we have described the current trends of HPC resources and presented their intrinsic properties to highlight the difficulty of exploiting them properly. Second, we presented a taxonomy of applications by resource requirements. Applications can either be *static*, i.e., their resource allocation does not change during execution, or *dynamic*. Static applications can further be divided into *rigid*, i.e., their resource requirements are fixed at compilation, or *moldable*, i.e., their resource requirements are chosen before they start. Dynamic applications can be *malleable*, i.e., their resource allocation may change as initiated by the system, and/or *evolving*, i.e., they require a change in resource allocation due to some internal constraints, such as an increase in the size of the simulation. Evolving applications can further be divided into fully predictable, marginally predictable and non-predictable, depending on how much time in advance they are able to predict their evolution. Finally, the chapter presented the state of the art in resource management, showed that some types of applications are not efficiently dealt with and that workarounds are necessary.

The contributions of the Thesis proposed resource management architectures, which support all types of applications without needing to recur to workarounds. This is achieved by promoting collaboration between applications and the RMS in order to efficiently negotiate resources. Each contribution consists both in an architecture, described in an abstract, implementation-independent manner, and a prototype implementation.

The first contribution, presented in Chapter 3, dealt with improving resource management for running moldable applications on centralized resources. This is applicable to Super, Cluster and Cloud computing, which are owned by a single institution and for which centralized control can be enforced. As a motivating example, we have presented how a multi-cluster computational electromagnetics application could improve its response time if it were able to use a custom

resource selection algorithm. We have highlighted that taking advantage of such an algorithm is difficult in practice. As a solution, we proposed CooRMv1, a centralized resource management architecture, which delegates resource selection to applications, while at the same time enforcing its own policy. Evaluation was done using simulations, which showed that the system is feasible, ensures fairness and scales well. Furthermore, simulation results were validated with real-life experiments on the Grid'5000 platform.

The second contribution of the Thesis dealt with moldable applications on distributed resources in Chapter 4. Such resources are owned by multiple parties, therefore, it is impractical to impose centralized control on them. They are to be found in Grid computing and when using multiple Cloud providers, also called Sky Computing. We proposed *dist*CooRM, a distributed version of CooRMv1, which allows moldable applications to efficiently use their resource selection algorithm to co-allocate resources managed by multiple agents. This allows participating institutions to keep their independence when managing resources and also improves the fault-tolerance of the whole system: For example, if a network bisection occurs, users still have access to the resources which are on their side of the bisection. Simulation results showed that the system is well-behaved and scales well for a reasonable number of applications. In collaboration with the Myriads team at IRISA/INRIA, Rennes, France, we are working on pushing scalability even further.

Next, we return to centralized resources and shift focus to dynamic applications. We start by studying theoretical aspects of how to efficiently support fully-predictably evolving applications in Chapter 5, a prerequisite to dealing with non-predictable ones. We formulated a problem statement and proposed a scheduling algorithm which considerably improves effective resource utilization, as compared to an algorithm which does not take application evolution into consideration. The proposed algorithm is then used as a basis for the CooRMv2 architecture in Chapter 6, which is an extension to CooRMv1. It supports all types of applications and, in particular, it allows non-predictable applications to make an efficient use of resources. An application may declare its peak resource requirements using pre-allocations. Resources that are pre-allocated but not effectively used, may be filled by other applications such as malleable ones. For the evaluation, an adaptive mesh refinement application was used as a non-predictable application, which we have modelled based on data that can be found in literature. Simulation results showed that the approach is feasible and that considerable gains can be made.

The previous contributions proposed low-level interfaces, working with number of nodes and node identifiers. In our last contribution, presented in Chapter 7, we study how collaboration could be promoted when negotiation is done with high-level concepts, such as tasks to execute and services which can execute them. To this end, we took two existing, production-level software, GridTLSE as an application and DIET as an RMS, and studied a use-case which was previously badly supported. We thus identified the underlying problem of scheduling optional computations, for which we proposed a generic master-client architecture called DIET-ethic. For the evaluation, a prototype implementation of DIET-ethic within DIET was used. Evaluation was done using real-life experiments on the Grid'5000 platform and a synthetic workload. Results showed that many metrics can be improved, such as user happiness, fairness and the number of completed requests. Furthermore, the architecture has been tested and shown to be scalable. Collaboration with IRIT, Toulouse, France allowed to make changes in GridTLSE to improve support for the motivating use-case.

∞.2 Perspectives

The contributions of this Thesis can be extended in several directions. We have divided them into short-term, medium-term and long-term.

∞.2.1 Short-term Perspectives

Short-term perspectives include implementing the proposed concepts in production-ready software, supporting dynamic applications on distributed resources and improving the scalability of *dist*CooRM.

Implementation The proposed architectures should be implemented within a production-ready RMS. We have already had initial discussions with the Mescal team at IMAG in Grenoble, France, who are working on the OAR batch scheduler. We concluded that implementing CooRMv1 should be fairly straight-forward, only requiring to change the default scheduler. On the other hand, implementing CooRMv2 is somewhat more challenging: Significant changes would have to be made to implement the semantics of the `NEXT` request relationship. More precisely, OAR would need modifications to make sure that the nodes which are shared between two consecutive requests (linked with a `NEXT` relationship) would not be cleaned.

*dist*CooRM would best be implemented in XtreemOS, since the latter also targets Grids. The two systems share a number of concepts, such as the presence of multiple agents, each managing a different resource and the usage of reservations to coordinate allocations on different agents. Works in this direction have already been started in collaboration with the Myriads team at IRISA/INRIA in Rennes, France.

Dynamic Applications Support for dynamic applications on distributed resources should be improved. This would allows non-predictably evolving applications, such as adaptive mesh refinement simulations, to make an efficient use of resources owned by multiple institutions, such as Grids or multiple Clouds. *dist*CooRM has already provided valuable insight into how to devise a distributed version of a collaborative RMS, while CooRMv2 proposed concepts to efficiently deal with dynamic applications. As such, combining the two systems, so as to solve the issue presented at the beginning of the paragraph, should not be too difficult.

Improved Scalability We have observed that *dist*CooRM does not present good weak-scaling capabilities: When the application to resource ratio is kept constant, the traffic per node increases quickly, due to the fact that an application negotiates with all managers of the platform. For targeting large-scale distributed platforms, an additional negotiation mechanism would be needed. For example, a pre-selection phase could be established, which would limit the number of managers an application negotiates with. This pre-selection phase would have to work with data structures which are constant in the size of the platform. Works in this direction have already been started in collaboration with the Myriads team.

∞.2.2 Medium-term Perspectives

Medium-term perspectives include improved support for exascale machines, formally verifying the devised architectures, proposing more differentiated allocations and an economic model for the them.

Exascale Support for petascale and exascale machines needs to be improved. A fundamental assumption of all architectures working with low-level concepts (CooRMv1, *dist*CooRM and CooRMv2) is that all computing nodes of a cluster or supercomputer are equivalent. This allowed us to do several simplifications, such as working with node-counts, instead of list of nodes, in some places. However, petascale and exascale machines feature non-homogeneous networks, mostly a torus or a fat-tree topology. Effectively, the observed latency and bandwidth between nodes may be different, which may considerably affect the performance of HPC applications. Therefore, optimizing resource selection needs to take this into consideration too. It is possible that applying a simple Hilbert-curve scheduling, as currently done in SLURM, to regroup close nodes, might prove sufficient for most use-cases. Otherwise, delegating resource selection to applications might need to be done at a finer granularity.

Formal Verification The fact that *dist*CooRM is well behaved has not been verified exhaustively. It has only been checked on paper on a certain number of cases and has been confirmed by doing a large amount of experiments. However, it would be desirable to have stronger proofs. This could be achieved using model checking, which, to state it simply, simulates all unique, reachable states of a distributed system to verify whether certain properties are held. A limited version of model checking has been implemented in SimGrid [82], the simulation framework that we used for *dist*CooRM. However, since it can only verify safety properties, it is insufficient for our needs. A PhD in the AlGorille team at LORIA/Université de Lorraine in Nancy, France is in progress aiming to extend SimGrid, so as to enable the verification of liveness properties.

Differentiated Allocations To further improve resource management, new differentiated allocations can be proposed. CooRMv2 currently proposes two levels of allocations. Non-preemptible allocations are guaranteed for their whole duration, while preemptible ones provide no guarantees: The RMS may terminate such an allocation whenever it decides to. Therefore, when using preemptible allocations, a malleable application would have to save its current state (i.e., checkpoint) often enough to make sure that it minimizes the amount of computations it loses. Even so, it does lose the computations it has done since the last checkpoint, since preemptible resources have to be release immediately upon the RMS's request. To reduce the amount of computations lost, it would be interesting to extend CooRMv2 and add a mechanism which would allow negotiating a **grace period**: The RMS can only terminate an allocation after having announced it some time ahead. An evolving application which needs more nodes could request the resources that are currently allocated to the malleable application, but would only receive them after the grace period expires. This is in contrast to the mechanism we proposed in Chapter 6, announced updates, in which the evolving application chooses an announce interval by itself, without taking into account the constraints of other applications in the system.

Economic Models Related to the above differentiated allocations, a new economical model needs to be devised. Indeed, each platform proposes some kind of economic model, which give users incentives to more efficiently use resources. In private infrastructures, such as Supercomputers or Grids, users are allocated a certain quota, from which their resource usage is deduced. In public infrastructures, such as Cloud, users (assumed to have a limited budged) pay for the resources they allocate. While economic models have already been used in many contexts, how to charge users for *different* types of allocations in the context of computing resource management needs to be studied. To our knowledge, Amazon is the only Cloud provider which proposes a pricing mechanism for non-guaranteed resources, called spot instance. As example of pricing

for CooRMv2 and the grace period mechanism described above, preemptible allocations could be charged less than non-preemptible ones. Likewise, the higher the grace period, the more the user pays. On the contrary, a bonus is awarded for announcing resource demand increases in advance.

∞.2.3 Long-term Perspectives

As long-term perspectives, we should return to the original problem of exploiting the full computing power of increasingly complex hardware. As has been said, this can only be done by orchestrating all layers of the platform, ranging from the application to the hardware itself. In this Thesis, we have focused on the RMS, however, it is unlikely that the proposed concepts can easily be used by programmers, provided they are not given higher level abstractions.

Increasing the abstraction level is usually the task of the runtime system and the programming model. Indeed, several high-level programming models have been developed, which attempt to simplify parallel programming. For example, Charm++ [67] is an object-oriented programming model, which allows the programmer to over-decompose an application into concurrent
chares object called **chares**, that communicate using message-passing. The programmer is unaware of the way her code is going to be mapped onto resources. Instead, it is the task of the runtime to dynamically load-balance computations on target resources.

Taking this concept one step further, we could imagine Charm++ cooperating with the RMS and handling resource allocations on behalf of the end-user. Instead of asking the user to choose a node-count at submittal, a Charm++ application could dynamically adapt its allocations to the load the application is currently experiencing and the state of the platform, thus obtaining a malleable and evolving application without additional programming effort. For example, if there are many resources available and many chares are active, the runtime could ask the RMS to allocate more resources. On the contrary, if few chares are active, the runtime could release some resources, so as to allow other users to speed-up their computation. Last, if many users are queued on the platform, waiting for resources to become available, the runtime could shrink the allocation of the application to allow fair sharing of the platform.

A similar approach could be used for OpenMP applications. When the runtime encounters a parallel-for which operates on a large array, it could request additional resources from the RMS. Conversely, if a sequential region is encountered, resources could be release back to the system to allow other users of the platform to do useful computations.

Taking this research direction would raise several related issues, such as choosing an algorithm, a granularity and extending the programming model. First, an algorithm would have to be devised that, based on the load of the application and the status of the platform, decides what request to send to the RMS.

Second, a level of granularity would have to be chosen. Join-fork application, as promoted by OpenMP) usually feature very short transitions between a parallel and a sequential section. Acquiring resources before the parallel section and releasing them at the end would not only slow down the application, waiting for RMS to allocate resources, but would also overwhelm the RMS. Therefore, a negotiation granularity should be chosen, which determines how often to negotiate with the RMS. It may either be a system parameter or a value negotiated with the RMS. As an example for the later approach, if the load of the RMS is low, a small (e.g., millisecond) granularity might be chosen. Conversely, if the load of the RMS is high, a large (e.g., minutes) granularity might be imposed.

Finally, as with any solution that tries to minimize the burden on the programmer, one size does not fit all. Primitives would need to be provided to the programmer to override the default

behavior of the runtime related to negotiating with the RMS. Also, new "hints" might be added to the programming model, to allow the programmer to highlight certain opportunities of optimizing resource management. For example, she might tag a sequential section as "long", which would indicate the runtime that it should definitely release resources before entering it. Similarly, the programmer might hint at the fact that parallelism is going to increase substantially and that the runtime should acquire resources some time before entering the parallel section. More interestingly, properly supporting some of these hints might not only need to evolve the runtime, but also the negotiating protocol with the RMS, thus having an impact on all layers of the platform.

Appendices

APPENDIX A

Supplementary Material

A.1 CooRMv2 RMS Implementation

This section gives details about the implementation of the RMS, which has been briefly presented in Section 6.5. First, the manipulated data structures are described in detail, next the pseudo-code of the implementation is given.

A.1.1 Requests

Requests (defined in Section 6.3.1) stored inside the RMS have two types of attributes: those sent by the application and those set by the RMS. The attributes which are sent by the application are:

- `cid` – the ID of the cluster;
- `n` – the number of nodes;
- `duration` – the duration of the allocation;
- `type` – the type of the request: pre-allocation (PA), non-preemptible ($\neg P$) or preemptible (P);
- `relatedHow` – the type of constraint: `FREE`, `NEXT` or `COALLOC`;
- `relatedTo` – the request to which the start-time is constrained.

While computing a schedule, the RMS sets the following additional request attributes:

- $\texttt{n}_{\texttt{alloc}}$ – the number of nodes that will be effectively allocated (see details below);
- `scheduledAt` – the time at which the allocation of this request should start;
- `fixed` – the request is fixed, see Section A.1.4;
- `earliestScheduleAt` – earliest time when the request can be scheduled, see Section A.1.4.

To store information about requests after they started, the RMS sets the following attributes:

- `startedAt` – time when the request has started; if the request has not started yet, e.g., it has just been sent by the application, this attribute is set to `NaN`; `started`(r) returns true if the request has been started.
- `nodeIDs` – the node IDs that have been allocated to this request.

Let us explain the purpose of the $\texttt{n}_{\texttt{alloc}}$ attribute. Assume there are two applications in the system: a malleable application A and an evolving application B. Let us assume the following scenario. A scans its preemptive view, finds out that additional resources are available for it and updates its preemptible request. At the same time, B needs more nodes, so it spontaneously

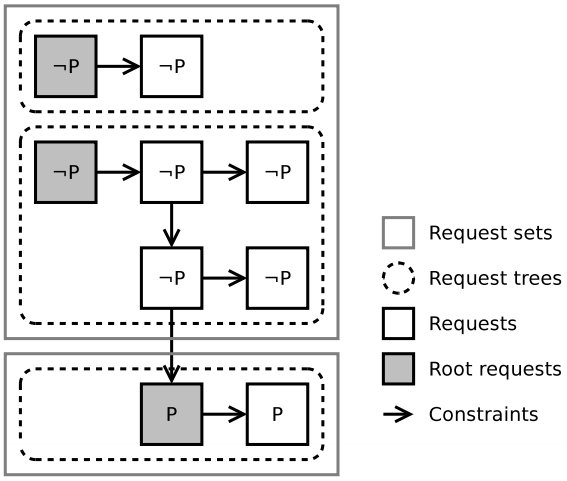


Figure A.1: Example of request trees

updates its non-preemptible request. The changes induced by the two applications trigger the scheduling algorithm of the RMS. Due to the race between A and B, if insufficient resources are available for both applications, then the RMS cannot allocate the requested node-count to A. Therefore, in order to correctly compute new views and request start-times, `nAlloc` stores the number of nodes which A can be allocated according to the current resource state. When a preemptible request is started, `nAlloc` is used to decide how many node IDs to allocate. $\mathtt{n}_{\mathtt{alloc}}$ might be smaller than `n`, which, since preemptible requests are not guaranteed, is allowed by the CooRMv2 specifications.

A.1.2 Request Constraints

In CooRMv2 each application is allowed to send multiple requests, thus the RMS needs to store for each application a **request set**. However, the RMS needs to treat requests of each type differently. Therefore, for each application i, the RMS stores three separate request sets: the set of pre-allocation requests $R^{(i)}_{PA}$, the set of non-preemptible requests $R^{(i)}_{\neg P}$ and the set of preemptible requests $R^{(i)}_{P}$.

Inside a request set, the requests and their constraints form multiple trees. Indeed, requests which are unconstrained or whose constraints are outside the request set are the roots of distinct trees, while `COALLOC` or `NEXT` constraints determine parent-child relations (see Figure A.1). Some of the algorithms that we shall present require navigating the request sets as a tree. Therefore, we define the following functions:

- $\mathtt{roots} : R \mapsto R_{\text{roots}}$, returns the set of root requests in R
  i.e., $\mathtt{roots}(R) = \{r \in R, \text{ such that } r.\mathtt{relatedHow} = \mathtt{FREE} \vee r.\mathtt{relatedTo} \notin R\}$
- $\mathtt{children} : r, R \mapsto R_{\text{children}}$, returns the set of child requests of r which belong to R
  i.e., $\mathtt{children}(r, R) = \{r_c \in R, \text{ such that } r_c.\mathtt{relatedTo} = r\}$

A.1.3 Views

Views (as defined in Section 6.3.1) map a cluster ID `cid` to a `CAP`, which is a step function: the x-axis represents the absolute time, while the y-axis represents the number of available nodes.

The `CAPs` are stores as a list of duration, node-count pairs, similarly to Evolution Profiles (EPs) (see Section 5.2). For example,

$$V = \{a : [(3600, 4), (3600, 3)], b : [(\infty, 6)]\}$$

represents that on cluster a, 4 nodes are available for time $t \in [0, 3600)$, 3 nodes are available for time $t \in [3600, 7200)$ seconds and 0 nodes are available for time $t \in [7200, \infty)$. On cluster b, 6 nodes are always available.

Let us define a few operations on views:

- we note $V[\texttt{cid}]$ the step-function which corresponds to `cid` in the view V;
  e.g., for the view above, $V[a] = [(3600, 4), (3600, 3)]$;
- $\texttt{CAP} : t \mapsto n$, returns the number of nodes n available at time t;
  e.g., $V[a](1800) = 4$, $V[a](3600) = 3$, $V[a](7200) = 0$;
- $\cup : V_1, V_2 \mapsto V_R$, returns the **union** view V of the views V_1 and V_2;
  i.e., $V_1 \cup V_2 = V \Leftrightarrow V[\texttt{cid}](t) = max(V_1[\texttt{cid}](t), V_2[\texttt{cid}](t)), \forall \texttt{cid}, t$
- $+ : V_1, V_2 \mapsto V_R$, returns the **sum** view V of the views V_1 and V_2;
  i.e., $V_1 + V_2 = V \Leftrightarrow V[\texttt{cid}](t) = V_1[\texttt{cid}](t) + V_2[\texttt{cid}](t), \forall \texttt{cid}, t$
- $- : V_1, V_2 \mapsto V_R$, returns the **difference** view V of the views V_1 and V_2;
  i.e., $V_1 - V_2 = V \Leftrightarrow V[\texttt{cid}](t) = V_1[\texttt{cid}](t) - V_2[\texttt{cid}](t), \forall \texttt{cid}, t$
- $\texttt{alloc} : V, r \mapsto n$, returns the node-count that should be allocated to r, without changing its start-time, limited by the resources available in V. This function is used to compute $\texttt{n}_{\texttt{alloc}}$.
  i.e., $\texttt{alloc}(V, r) = n \Leftrightarrow n = \min(\min_{t \in [r.\texttt{scheduledAt}, r.\texttt{scheduledAt} + r.\texttt{duration})} V[r.\texttt{cid}](t), r.\texttt{n})$
- $\texttt{findHole} : V, r, t_0 \mapsto t_s$, returns the first time after $\max(t_0, r.\texttt{earliestScheduleAt})$ when r could be started, so that all requested resources can be allocated;
  i.e., find $t_s \geq \max(t_0, r.\texttt{earliestScheduleAt})$, such that
  $\min_{t \in [t_s, t_s + r.\texttt{duration})} V[r.\texttt{cid}](t) \geq r.\texttt{n}$

All above operations can be easily implemented by iterating through the list of duration, node-count pairs of the relevant `CAPs`. Due to their simplicity, the pseudo-code of the above operations is not given.

A.1.4 Helper Functions

In order to ease the description of the main scheduling algorithm, three helper functions are defined. Their purpose is to transform a set of requests into a view (called *the generated view*), representing how resources are being occupied by the input requests. Once a set of requests has been transformed into generated views, the above defined operations on views can be used to compute the resources that are left unused after a set of requests is served. The helper functions are: `toView`, `fit` and `eqSchedule`.

`toView`()

`toView`() (see Algorithm A.1) generates a view representing resources occupied by **fixed requests**, defined as requests which are either started or are constrained to a fixed request. These requests are treated specially, because the RMS cannot choose a start-time for them

fixed requests

Algorithm A.1: Implementation of the `toView()` function

Input: R, set of requests
V_i, view of available resources (optional)
Output: V_o, view generated by requests
The `scheduledAt`, $\mathtt{n}_{\mathtt{alloc}}$ and `fixed` attributes of requests are updated

```
   /* Initialization: start with an empty output view, clear fixed flag of
      requests and create an empty queue                                    */
1  V_o ← ∅ ;
2  forall the r ∈ R do r.fixed ← false ;
3  Q ← ∅ /* Queue of requests which are to be processed                    */
   /* First, add started requests to queue                                  */
4  forall the r ∈ R, such that started(r) do
5      enqueue(Q, r) ;
   /* Next, process requests in the queue                                   */
6  while Q ≠ ∅ do
7      r = dequeue(Q) ;
8      switch r.relatedHow do
9          case FREE
10             r.scheduledAt ← r.startedAt ;
11         case NEXT
12             r.scheduledAt ← r.relatedTo.scheduledAt + r.relatedTo.duration ;
13         case COALLOC
14             r.scheduledAt ← r.relatedTo.scheduledAt ;
15         otherwise
16             continue /* Constraint not implemented                       */
17     if V_i = ∅ then
18         r.n_alloc = r.n ;
19     else
20         r.n_alloc = alloc(V_i, r) ;
21     r.fixed ← True ;
22     V_o ← V_o + {r.cid : [(r.scheduledAt, 0), (r.duration, r.n_alloc)]} ;
       /* Enqueue children of this request                                  */
23     forall the r_c ∈ children(r) do
24         enqueue(Q, r) ;
```

anymore. Indeed, in order not to violate the CooRMv2 protocol and allocate resources to applications according to their requests, the start-times are fixed. As a side effect, `toView`() sets the `scheduledAt`, `fixed` and $\mathtt{n}_{\mathtt{alloc}}$ attributes of the input requests as needed.

For generating the view of preemptible requests, `toView`() accepts an optional parameter V_i representing the available resources. If this parameter is given, the generated view uses at most as many resources as available in V_i. The attribute $\mathtt{n}_{\mathtt{alloc}}$ of preemptible requests is set according to this limit.

`fit`()

`fit`() (see Algorithm A.2) generates a view representing resources that are occupied by non-fixed requests. The RMS has the liberty to choose the start-time for these requests, subject to the application-provided constraints. Note that its purpose is similar to the `fit`() function in Chapter 5, which is why we have chosen to keep the same name.

The main idea of the algorithm is as follows. Each request has an `earliestScheduleAt` attribute, which specifies which is the time that the request can be scheduled the earliest. Initially, this attribute is set to t_0, which is input to the algorithm (line 3). Then, the algorithm uses a queue of requests it has to process, which is initially filled with all root requests (line 5). Each request is then dequeue (line 7) and, depending on the request's constraint, the `scheduledAt` attribute is computed (lines 14–33). If this attribute is changed, all child requests are queued, so as to recompute their `scheduledAt` attribute (lines 34–35).

In certain cases, the current schedule of the parent request makes it impossible for a child request to be scheduled so as to respect its constraint (lines 22, 31). If this happens, the `earliestScheduleAt` attribute of the parent is updated and the parent is added anew to the queue. This attribute is then used by `findHole` to find a later schedule-time for the parent, which might allow the child to find a valid schedule (lines 15, 21, 30). For preemptible requests, instead of delaying their parent, these requests are shrunken and $\mathtt{n}_{\mathtt{alloc}}$ is set accordingly (lines 19, 28). This behavior is allowed by the CooRMv2 specifications.

Eventually, the algorithm converges and the `scheduledAt` attribute of all requests is stable (in worst case, all requests are scheduled at infinity). The `scheduledAt` and $\mathtt{n}_{\mathtt{alloc}}$ attributes of the requests are used to computed the generated view (lines 36–38).

`eqSchedule`()

The purpose of this function is to do the equi-partitioning of resources available for preemptible requests. It gets as input the preemptible request set of each application, the view representing the available resources for equi-partitioning and the time after which non-started requests have to be scheduled. It outputs for each application a preemptible view. As a side effect, the `scheduledAt` and $\mathtt{n}_{\mathtt{alloc}}$ attributes of requests are updated.

The pseudo-code is shown in Algorithm A.3. It consists of three main parts. First, preliminary occupation views are computed (lines 1–3). These are used to determine time intervals, during which the requested node-count of all applications are constant. Second, for each piece-wise constant resource state (i.e., application resource request and available resources), the number of nodes that can be assigned to each application is determined (lines 4–27). Third, the computed views are used to reschedule the application requests and correctly set the `scheduledAt` and $\mathtt{n}_{\mathtt{alloc}}$ attributes of requests (lines 28–30).

The second part of the algorithm, which operates on a piece-wise constant resource state, works as follows. The system is either in a congested state, i.e., more resources are requested

Algorithm A.2: Implementation of the `fit` function

Input: R, set of requests (`scheduledAt` and `fixed` must have been set)
V_i, view of available resources
t_0, requests have to be scheduled after this time
Output: V_o, view generated by requests
The `scheduledAt` and $\texttt{n}_{\texttt{alloc}}$ attributes of non-fixed requests are updated

```
1  Q ← ∅ /* Queue of requests which are to be processed                          */
2  forall the r ∈ R, such that ¬r.fixed do
3      r.earliestScheduleAt ← t_0 ;      /* No request can be scheduled earlier than t_0 */
4      r.scheduledAt ← ∞ ;               /* In case of error, the request never starts */
5  forall the r ∈ roots(R) do enqueue(Q, r) ;
6  while Q ≠ ∅ do
7      r ← dequeue(Q) ;
8      if r.fixed then      /* If this is a fixed request, just add children to queue */
9          forall the r_c ∈ children(r) do enqueue(Q, r_c) ;
10         continue ;
       /* Take a decision for the current request, depending on its constraint     */
11     r_p ← r.relatedTo ;          /* Store parent request to make pseudo-code briefer */
12     r.n_alloc ← r.n ;    /* Set a default value for n_alloc (will be overwritten later) */
13     t_before ← r.scheduledAt ;  /* Store the previous value, used to detect changes */
14     switch r.relatedHow do
15         case FREE  r.scheduledAt = findHole(V_i, r, 0) ;
16         case COALLOC
17             if r.type = P ∧ r_p.type ∈ {PA, ¬P} then
18                 r.scheduledAt ← r_p.scheduledAt ;
19                 r.n_alloc = alloc(V_i, r) ;
20             else
21                 r.scheduledAt = findHole(V_i, r, r_p.scheduledAt) ;
22                 if r.scheduledAt ≠ r_p.scheduledAt then
23                     r_p.earliestScheduleAt ← r.scheduledAt ;
24                     enqueue(Q, r_p) ;
25         case NEXT
26             if r.type = P then
27                 r.scheduledAt ← r_p.scheduledAt + r_p.duration ;
28                 r.n_alloc = alloc(V_i, r) ;
29             else
30                 r.scheduledAt = findHole(V_i, r, r_p.scheduledAt + r_p.duration) ;
31                 if r.scheduledAt ≠ r_p.scheduledAt + r_p.duration then
32                     r_p.earliestScheduleAt ← r.scheduledAt − r_p.duration ;
33                     enqueue(Q, r_p) ;
       /* If scheduledAt has changed, reschedule children                          */
34     if t_before ≠ r.scheduledAt then
35         forall the r_c ∈ children(r) do enqueue(Q, r_c)
   /* Schedule converged, compute generated view                                  */
36 V_o = ∅ ;
37 forall the r ∈ R, such that ¬r.fixed do
38     V_o ← V_o + {r.cid : [(r.scheduledAt, 0), (r.duration, r.n_alloc)]} ;
```

Algorithm A.3: Implementation of the `eqSchedule` function

Input: $R_P^{(i)}$, for each application i ($i = 1 \dots n_{app}$), the set of preemptible requests
V_{in}, view of available resources for equi-partitioning
t_0, non-started requests have to be scheduled after this time
Output: $V_P^{(i)}$, preemptive view of application i
The `scheduledAt` and $\texttt{n}_{\texttt{alloc}}$ attributes of the requests are updated

1 **for** $i \leftarrow 1$ **to** n_{app} **do**
2 $\quad V_{occ}^{(i)} \leftarrow \texttt{toView}(R_P^{(i)}, V_{in})$;
3 $\quad V_{occ}^{(i)} \leftarrow V_{occ}^{(i)} - \texttt{fit}(R_P^{(i)}, V_{in} - V_{occ}^{(i)}, t_0)$;
4 **forall the** *cid* $\in$ *clusters* **do**
5 $\quad$ **forall the** $t, d \in$ *time interval start-times and durations* **do**
$\quad\quad$ `/* Store requested and available node-counts during this interval */`
6 $\quad\quad$ **for** $i \leftarrow 1$ **to** n_{app} **do** $r^{(i)} \leftarrow V_{occ}^{(i)}(t)$;
7 $\quad\quad$ $v_{in} \leftarrow V_{in}(t)$;
$\quad\quad$ `/* Is the system congested during this time interval? */`
8 $\quad\quad$ **if** $\sum_{i=1}^{n_{app}} r^{(i)} > v_{in}$ **then**
$\quad\quad\quad$ `/* Initialize views for this interval */`
9 $\quad\quad\quad$ **for** $i \leftarrow 1$ **to** n_{app} **do** $v^{(i)} \leftarrow 0$;
$\quad\quad\quad$ `/* Distribute resources equally until none are left free */`
10 $\quad\quad\quad$ **while** $v_{in} \neq \varnothing \wedge \exists i,$ *such that* $r^{(i)} \geq 0$ **do**
11 $\quad\quad\quad\quad$ $n_{partitions} \leftarrow \#\{r^{(i)} > 0\}$; `/* Compute number of equi-partitions */`
12 $\quad\quad\quad\quad$ **if** $\exists i,$ *such that* $r^{(i)} = 0$ **then** $n_{partitions} \leftarrow n_{partitions} + 1$;
13 $\quad\quad\quad\quad$ $v_{eq} \leftarrow \max(v_{in} \div n_{partitions}, 1)$; `/* Compute size of equi-partitions */`
14 $\quad\quad\quad\quad$ **forall the** $i,$ *such that* $r^{(i)} \geq 0$ **do**
15 $\quad\quad\quad\quad\quad$ $v_{in} \leftarrow v_{in} - \min(r^{(i)}, v_{eq})$;
16 $\quad\quad\quad\quad\quad$ $r^{(i)} \leftarrow r^{(i)} - v_{eq}$;
17 $\quad\quad\quad\quad\quad$ $v^{(i)} \leftarrow v^{(i)} + v_{eq}$;
18 $\quad\quad\quad\quad\quad$ **if** $v_{in} = 0$ **then** break ;
19 $\quad\quad$ **else**
$\quad\quad\quad$ `/* Give each application the amount of resources left free by other applications */`
20 $\quad\quad\quad$ **forall the** $i \leftarrow 1$ **to** n_{app} **do**
21 $\quad\quad\quad\quad$ $v^{(i)} \leftarrow v_{in} - \sum_{i \neq j} r^{(j)}$;
$\quad\quad\quad\quad$ `/* But not less than the equi-partition */`
22 $\quad\quad\quad\quad$ $n_{partitions} \leftarrow \#\{r^{(j)} > 0\}$;
23 $\quad\quad\quad\quad$ **if** $r^{(i)} = 0$ **then** $n_{partitions} \leftarrow n_{partitions} + 1$;
24 $\quad\quad\quad\quad$ $v_{eq} \leftarrow v_{in} \div n_{partitions}$;
25 $\quad\quad\quad\quad$ $v^{(i)} \leftarrow \max(v_{in}, v_{eq})$;
$\quad\quad$ `/* Append values computed for this time interval to application views */`
26 $\quad\quad$ **for** $i \leftarrow 1$ **to** n_{app} **do**
27 $\quad\quad\quad$ append step $\left(d, v^{(i)}\right)$ to $V_P^{(i)}[cid]$;
28 **for** $i \leftarrow 1$ **to** n_{app} **do**
29 $\quad V_{occ}^{(i)} \leftarrow \texttt{toView}(R_P^{(i)}, V_P^{(i)})$;
30 $\quad V_{occ}^{(i)} \leftarrow V_{occ}^{(i)} - \texttt{fit}(R_P^{(i)}, V_P^{(i)} - V_{occ}^{(i)}, t_0)$;

than available for equi-partitioning, or the system is uncongested. The former situation might occur, due to the following reasons:

- since the last scheduling iteration, the number of resources for equi-partitioning has been reduced,
- a new application has chosen to participate in equi-partitioning,
- an application has decided to increase the requested node-count and use more of its partition.

If any of these situations occur, it is the duty of the RMS to inform malleable applications (by sending them new views), that they have to release some resources. The available resources are distributed equally, until the requests of applications are satisfied (lines 8–18).

In case the system is not congested (during a certain time interval), each application is assigned a view based on the number of resources that other applications leave unused (lines 19–25). However, this should not be smaller that the equi-partition of the application.

The number of equi-partitions is computed as follows (lines 22–23). Applications are either considered active, i.e., they currently request preemptible resources, or inactive. When computing the view of active applications, the number of partitions is equal to the number of active applications in the system. For inactive applications, the number of partitions is equal to the number of active applications in the system plus 1, i.e., the number of partitions if this application were to become active.

A.1.5 Main Scheduling Algorithm

Finally, let us describe the main scheduling algorithm. As stated in Section 6.5, this algorithm is triggered whenever a `request` or `done` message is received from an application. The purpose of this algorithm is to compute views for applications and start-times for requests, given as input the current requests of the applications, the number of nodes on each cluster and the current time.

The proposed pseudo-code is presented in Algorithm A.4. Two scratch variables are used, $V_{\neg P}$ and V_P, which store two views, representing non-preemptible / preemptible resources that are still available for scheduling. The algorithm works as follows:

1. The scratch variables are initialized to represent all resources (line 1–2);
2. Resources that are already allocated and cannot be preempted, i.e., resources assigned to started preallocations and non-preemptible requests, are subtracted (line 3–5);
3. Preallocations and non-preemptible requests that are not started are scheduled (line 6–11). Non-preemptive views are computes as part of this step;
4. Preemptive views are computed and preemptible requests are scheduled using the previously described `eqSchedule` function (line 12);
5. If any of the computed start-times is the current time, the corresponding requests are started (line 13–14).

For simplicity, the presented scheduling algorithm only allocates non-preemptible requests inside preallocations. The newly computes views are sent immediately to the application.

Let us focus on the `nodeIDs` attribute of requests. This attribute is initialized to an empty set when the request is first submitted to the RMS. Next, if the main scheduling algorithm decided that a request should start, two situations may occur:

1. either enough nodes are free, the `nodeIDs` attribute is immediately assigned $\mathtt{n}_{\mathtt{alloc}}$ node IDs and the application is sent a `startNotify` message;

Algorithm A.4: CooRMv2 main scheduling algorithm

Input: $n^{(cid)}$, number of nodes on cluster cid
$R^{(i)}_{PA}$, $R^{(i)}_{\neg P}$, $R^{(i)}_{P}$, set of PA, $\neg P$ and P requests of application i ($i = 1 \dots n_{app}$)
`now`, the current time

Output: $V^{(i)}_{\neg P}$, non-preemptive view presented to application i
$V^{(i)}_{P}$, preemptive view presented to application i
updates the `scheduledAt`, `startedAt` and $\texttt{n}_{\texttt{alloc}}$ attributes of requests

`/* Initialize temporary views with all resources */`

1 $V_{\neg P} \leftarrow \{\texttt{cid} : [\infty, n^{(cid)}], \forall \texttt{cid}\}$; `/* non-preemptible resources */`

2 $V_{P} \leftarrow \{\texttt{cid} : [\infty, n^{(cid)}], \forall \texttt{cid}\}$; `/* preemptible resources */`

`/* Subtract resources allocated to started requests */`

3 **for** $i \leftarrow 1$ **to** n_{app} **do**

4 $\quad V_{\neg P} \leftarrow V_{\neg P} - \texttt{toView}(R^{(i)}_{PA})$; `/* Subtract pre-allocated resources */`

5 $\quad V_{P} \leftarrow V_{P} - \texttt{toView}(R^{(i)}_{\neg P})$; `/* Subtract non-preemptibly allocated resources */`

`/* Compute non-preemptive views and start-times of non-preemptible requests */`

6 **for** $i \leftarrow 1$ **to** n_{app} **do**

7 $\quad V^{(i)}_{\neg P} \leftarrow \texttt{toView}(R^{(i)}_{PA}) + V_{\neg P}$

$\quad$ `/* Compute what this application occupies */`

8 $\quad V^{occ}_{PA} = \texttt{fit}(R^{(i)}_{PA} V^{(i)}_{\neg P} \texttt{now})$;

9 $\quad V^{occ}_{\neg P} = \texttt{fit}(R^{(i)}_{\neg P} \texttt{toView}(R^{(i)}_{PA} + V^{occ}_{PA} - \texttt{toView}(R^{(i)}_{\neg P}, \texttt{now})$;

$\quad$ `/* Update views for next application */`

10 $\quad V_{\neg P} = V_{\neg P} - V^{occ}_{PA}$;

11 $\quad V_{P} \leftarrow V_{P} - V^{occ}_{\neg P}$;

`/* Compute preemptive views and start-times of preemptible requests */`

12 $V^{(1 \dots n_{app})}_{P} = \texttt{eqSchedule}(R^{(1 \dots n_{app})}_{P}, V_P, \texttt{now})$;

`/* Start requests, whose start-time is now */`

13 **forall the** $\neg\texttt{started}(r)$ *and* $r.\texttt{scheduledAt} \leq \texttt{now}$ **do**

14 $\quad r.\texttt{startedAt} \leftarrow \texttt{now}$;

2. or insufficient nodes are currently free, the RMS waits for an application to release resources. When the application sends the `done` message (as part of an update), the scheduling algorithm is re-run and the `nodeIDs` attribute is reconsidered.

A.1.6 Limitations

The implementation presented in this section has the following limitations:

- Non-preemptible allocations can only be done inside preallocations.
- The scheduling algorithm does not handle overlapping requests, i.e., requests of the same type, on the same cluster that temporarily overlap.
- Invalid request updates are not handle gracefully. For example, if an application updates a started pre-allocation, so as to request more resources than are available, the RMS is unable to allocate resources to it. The above implementation does not handle these cases well.
- Applications which "steal" resources, i.e., do not release preemptively allocated resources when they are asked to, are not killed.

The above implementation provides a solid foundation and it can easily be extended to address these issues.

APPENDIX B

Acronyms

ACT	Average Completion Time
AEM	Application Execution Manager (a module of XtreemOS)
AMR	Adaptive Mesh Refinement
API	Application Programming Interface
ALS	Application-Level Scheduler
BoT	Bag of Tasks
CBF	Conservative Back-Filling
CDR	Common Data Representation
CEM	Computational ElectroMagnetics
CM	Cloud Manager
CPU	Central Processing Unit
DEISA	Distributed European Infrastructure for Supercomputing Applications
DIET	Distributed Interactive Engineering Toolbox
DHT	Distributed Hash Table
DRM	Distributed Resource Manager
EASY	aggressive back-filling (Extensible Argonne Scheduling sYstem)
EGI	European Grid Infrastructure
EP	Evolution Profile
FCFS	First-Come First-Serve
FEM	Finite-Element Method
FLOPS	Floating-Point Operations per Second
GPU	Graphics Processing Unit
HaaS	Hardware as a Service
HPC	High-Performance Computing
IaaS	Infrastructure as a Service

ID	identifier
IIOP	Internet Inter-ORB Protocol (part of the CORBA specification)
JSDL	Job Submission Description Language
JSON	JavaScript Object Notation
LA	Local Agent
LAN	Local-Area Network
LHC	Large Hadron Collider
LRMS	Local Resource Management System
MA	Master Agent
MMORPG	Massive Multi-player Online Role-Playing Game
MPI	Message-Passing Interface
NEA	Non-predictably Evolving Application
OCCI	Open Cloud Computing Interface
PaaS	Platform as a Service
POSIX	Portable Operating System Interface
PRACE	Partnership for Advanced Computing in Europe
PSA	Parameter-Sweep Application
QoS	Quality of Service
OpenMP	Open MultiProcessing
OS	Operating System
RAM	Random-Access Memory
REST	Representational state transfer
RMS	Resource Management System
RPC	Remote Procedure Call
SaaS	Software as a Service
SAGA	Simple API for Grid Applications
SeD	Service Daemon
SLOC	Source Lines of Code
SMP	Symmetric MultiProcessing
TCP	Transmission Control Protocol
UUID	Universally Unique IDentifier
VM	Virtual Machine
VO	Virtual Organization
WAN	Wide-Area Network
XML	eXtensible Markup Language
XSEDE	Extreme Science and Engineering Discovery Environment
YAML	Yet Another Markup Language

APPENDIX C

Bibliography

[1] Rashid J. Al-Ali, Kaizar Amin, Gregor von Laszewski, Omer F. Rana, David W. Walker, Mihael Hategan, and Nestor J. Zaluzec. Analysis and Provision of QoS for Distributed Grid Applications. *Journal of Grid Computing*, 2(2):163–182, 2004. doi:10.1007/s10723-004-6743-8.

[2] A. Amar, Raphael Bolze, Aurelien Bouteiller, Andréea Chis, Yves Caniou, Eddy Caron, Pushpinder-Kaur Chouhan, Gaël Le Mahec, Holly Dail, Benjamin Depardon, Frédéric Desprez, Jean-Sébastien Gay, and Alan Su. Diet: New Developments and Recent Results. In *Euro-Par 2006 Workshops: Parallel Processing, CoreGRID 2006, UNICORE Summit 2006, Petascale Computational Biology and Bioinformatics, Dresden, Germany, August 29-September 1, 2006, Revised Selected Papers*, volume 4375 of *Lecture Notes in Computer Science*, pages 150–170. Springer, 2007. doi:10.1007/978-3-540-72337-0_15.

[3] Patrick R. Amestoy, Iain S. Duff, Luc Giraud, Jean-Yves L'Excellent, and Chiara Puglisi. Grid-TLSE: A Web Site for Experimenting with Sparse Direct Solvers on a Computational Grid. In *Proceedings of the SIAM conference on Parallel Processing for Scientific Computing, San Francisco, USA, 25–27 February 2004*, 2004.

[4] David P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04, pages 4–10. IEEE Computer Society, 2004. doi:10.1109/GRID.2004.14.

[5] Sergio Andreozzi, Stephen Burke, Felix Ehm, Laurence Field, Gerson Galang, Balazs Konya, Maarten Litmaath, Paul Millar, and JP Navarro. GLUE Specification v. 2.0. Recommendation GFD-R-P.147, Open Grid Forum (OGF), 2009.

[6] Ali Anjomshoaa, Fred Brisard, et al. Job Submission Description Language (JSDL) Specification, Version 1.0. Full Recommendation GDF-R.136, Open Grid Forum (OGF), 2008.

[7] N. Bard, R. Bolze, E. Caron, F. Desprez, M. Heymann, A. Friedrich, L. Moulinier, N. H. Nguyen, O. Poch, and T. Toursel. Décrypthon grid - grid resources dedicated to neuromuscular disorders. *Studies in Health Technology and Informatics*, 159:124–133, 2010.

[8] Olivier Beaumont, Lionel Eyraud-Dubois, and Young J. Won. Using the last-mile model as a distributed scheme for available bandwidth prediction. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par'11, pages 103–116. Springer-Verlag, 2011.

[9] Francine Berman, Richard Wolski, Henri Casanova, Walfredo Cirne, Holly Dail, Marcio Faerman, Silvia Figueira, Jim Hayes, Graziano Obertelli, Jennifer Schopf, Gary Shao, Shava Smallen, Neil Spring, Alan Su, and Dmitrii Zagorodnov. Adaptive Computing on the Grid using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):369–382, 2003. doi:10.1109/TPDS.2003.1195409.

[10] Shishir Bharathi, Ann Chervenak, Ewa Deelman, Gaurang Mehta, Mei-Hui Su, and Karan Vahi. Characterization of scientific workflows. In *Third Workshop on Workflows in Support of Large-Scale Science*, pages 1–10, 2008. doi:10.1109/works.2008.4723958.

[11] Blue Waters Project Team. The Cray Blue Waters system. *Blue Waters Project Newsletter*, 3(1), January 2012.

[12] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Iréa Touche. Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, November 2006. doi:10.1177/1094342006070078.

[13] Aurelien Bouteiller, Thomas Hérault, Géraud Krawezik, Pierre Lemarinier, and Franck Cappello. MPICH-V Project: A Multiprotocol Automatic Fault-Tolerant MPI. *International Journal of High Performance Computing Applications*, 20(3):319–333, 2006. doi:10.1177/1094342006067469.

[14] Hinde-Lilia Bouziane, Christian Pérez, and Thierry Priol. A Software Component Model with Spatial and Temporal Compositions for Grid Infrastructures. In *Euro-Par 2008 - Parallel Processing, 14th International Euro-Par Conference, Las Palmas de Gran Canaria, Spain, August 26-29, 2008, Proceedings*, volume 5168 of *Lecture Notes in Computer Science*, pages 698–708. Springer, 2008. doi:10.1007/978-3-540-85451-7_75.

[15] Greg L. Bryan, Tom Abel, and Michael L. Norman. Achieving extreme resolution in numerical cosmology using adaptive mesh refinement: resolving primordial star formation. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, Supercomputing '01. ACM, 2001. doi:10.1145/582034.582047.

[16] Jeremy Buisson, Omer Ozan Sonmez, Hashim H. Mohamed, Wouter Lammers, and Dick H. J. Epema. Scheduling malleable applications in multicluster systems. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, pages 372–381. IEEE, 2007. doi:10.1109/CLUSTR.2007.4629252.

[17] Carsten Burstedde, Omar Ghattas, Michael Gurnis, Tobin Isaac, Georg Stadler, Tim Warburton, and Lucas Wilcox. Extreme-Scale AMR. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–12, Washington, DC, USA, 2010. IEEE Computer Society. doi:10.1109/SC.2010.25.

[18] Maria Calzarossa and Giuseppe Serazzi. A Characterization of the Variation in Time of Workload Arrival Patterns. *IEEE Transactions on Computers*, C-34(2):156–162, 1985.

[19] Yves Caniou, Ghislain Charrier, and Frederic Desprez. Analysis of Tasks Reallocation in a Dedicated Grid Environment. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing*, CLUSTER '10, pages 284–291. IEEE Computer Society, 2010. doi:10.1109/CLUSTER.2010.39.

[20] Yves Caniou, Ghislain Charrier, and Frédéric Desprez. Evaluation of Reallocation Heuristics for Moldable Tasks in Computational Grids. In *9th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2011), Perth, Australia, 17-20 January, 2011*. ACM, 2011.

[21] Yves Caniou and Jean-Sébastien Gay. Simbatch: An API for Simulating and Predicting the Performance of Parallel Resources Managed by Batch Systems. In *Euro-Par 2008 Workshops - Parallel Processing, VHPC 2008, UNICORE 2008, HPPC 2008, SGS 2008, PROPER 2008, ROIA 2008, and DPA 2008, Las Palmas de Gran Canaria, Spain, August 25-26, 2008, Revised Selected Papers*, volume 5415 of *Lecture Notes in Computer Science*, pages 223–234. Springer, 2009. doi:10.1007/978-3-642-00955-6_27.

[22] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron, and Olivier Richard. A batch scheduler with high level components. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid*, volume 2 of *CCGRID '05*, pages 776–783. IEEE Computer Society, 2005. doi:10.1109/CCGRID.2005.1558641.

[23] Franck Cappello, Al Geist, Bill Gropp, Sankay Kale, Bill Kramer, and Marc Snir. Toward Exascale Resilience. Technical Report TR-JLPC-09-01, INRIA-Illinois Joint Laboratory on PetaScale Computing, July 2009.

[24] Eddy Caron. *Contribution to the management of large scale platforms: the Diet experience.* Habilitation thesis, Ecole Normale Supérieure de Lyon, October 2010.

[25] Eddy Caron, Andréea Chis, Frédéric Desprez, and Alan Su. Design of plug-in schedulers for a GridRPC environment. *Future Generation Computer Systems*, 24(1):46–57, 2008. doi:10.1016/j.future.2007.02.005.

[26] Eddy Caron, Benjamin Depardon, and Frédéric Desprez. Multiple Services Throughput Optimization in a Hierarchical Middleware. In *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2011, Newport Beach, CA, USA, May 23-26, 2011*, pages 94–103. IEEE, 2011. doi:10.1109/CCGrid.2011.20.

[27] Eddy Caron, Frederic Desprez, David Loureiro, and Adrian Muresan. Cloud Computing Resource Management through a Grid Middleware: A Case Study with DIET and Eucalyptus. In *Proceedings of the 2009 IEEE International Conference on Cloud Computing*, CLOUD '09, pages 151–154. IEEE Computer Society, 2009. doi:10.1109/CLOUD.2009.70.

[28] Eddy Caron, Frédéric Desprez, and Adrian Muresan. Pattern Matching Based Forecast of Non-periodic Repetitive Behavior for Cloud Clients. *Journal of Grid Computing*, 9(1):49–64, 2011. doi:10.1007/s10723-010-9178-4.

[29] Eddy Caron and Frédéric Desprez. DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006. doi:10.1177/1094342006067472.

[30] Eddy Caron, Cristian Klein, and Christian Pérez. Efficient Grid Resource Selection for a CEM Application. In *9ème Rencontres francophones du Parallélisme, Renpar'19, Toulouse, France, September 9-11, 2009*, 2009.

[31] Adrian Casajus, Ricardo Graciani, Stuart Paterson, Andrei Tsaregorodtsev, and the LHCb DIRAC Team. DIRAC pilot framework and the DIRAC Workload Management System. *Journal of Physics: Conference Series*, 219(6), 2010. doi:10.1088/1742-6596/219/6/062049.

[32] Henri Casanova. Benefits and Drawbacks of Redundant Batch Requests. *Journal of Grid Computing*, 5(2):235–250, 2007. doi:10.1007/s10723-007-9068-6.

[33] Henri Casanova, Arnaud Legrand, and Martin Quinson. SimGrid: A Generic Framework for Large-Scale Distributed Experiments. In *Proceedings of the Tenth International Conference on Computer Modeling and Simulation*, UKSIM '08, pages 126–131. IEEE Computer Society, 2008. doi:10.1109/UKSIM.2008.28.

[34] Márcia C. Cera, Yiannis Georgiou, Olivier Richard, Nicolas Maillard, and Philippe O. A. Navaux. Supporting malleability in parallel architectures with dynamic CPUSETs mapping and dynamic MPI. In *Proceedings of the 11th International Conference on Distributed Computing and Networking*, ICDCN'10, pages 242–257. Springer-Verlag, 2010. doi:10.1007/978-3-642-11322-2_26.

[35] Sumir Chandra, Manish Parashar, and Salim Hariri. GridARM: An Autonomic Runtime Management Framework for SAMR Applications In Grid Environments. In *New Frontiers in High-Performance Computing, Proceedings of the Autonomic Applications Workshop 10th International Conference on High Performance Computing, HiPC 2003, Hyderabad, India. December 2003*, pages 286–295. Elite Publishing, 2003.

[36] Pushpinder-Kaur Chouhan, Eddy Caron, and Frédéric Desprez. Automatic middleware deployment planning on heterogeneous platforms. In *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, Miami, Florida USA, April 14-18, 2008*. IEEE, 2008. doi:10.1109/IPDPS.2008.4536171.

[37] Walfredo Cirne and Francine Berman. Using Moldability to Improve the Performance of Supercomputer Jobs. *Journal of Parallel and Distributed Computing*, 62(10):1571–1601, 2002. doi:10.1006/jpdc.2002.1869.

[38] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: a decentralized network coordinate system. In *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '04, pages 15–26. ACM, 2004. doi:10.1145/1015467.1015471.

[39] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150. USENIX Association, 2004.

[40] Simon Delamare, Gilles Fedak, Derrick Kondo, and Oleg Lodygensky. SpeQuloS: a QoS service for BoT applications using best effort distributed computing infrastructures. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 173–186. ACM, 2012. doi:10.1145/2287076.2287106.

[41] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, Franck Cappello, Barbara Chapman, Xuebin Chi, Alok Choudhary, Sudip Dosanjh, Thom Dunning, Sandro Fiore, Al Geist, Bill Gropp, Robert Harrison, Mark Hereld, Michael Heroux, Adolfy Hoisie, Koh Hotta, Zhong Jin, Yutaka Ishikawa, Fred Johnson, Sanjay Kale, Richard Kenway, David Keyes, Bill Kramer, Jesús Labarta, Alain Lichnewsky, Thomas Lippert, Bob Lucas, Barney Maccabe, Satoshi Matsuoka, Paul Messina, Peter Michielse, Bernd Mohr, Matthias S. Mueller, Wolfgang E. Nagel, Hiroshi Nakashima, Michael E Papka, Dan Reed, Mitsuhisa Sato, Ed Seidel, John Shalf, David Skinner, Marc Snir, Thomas Sterling, Rick Stevens, Fred Streitz, Bob Sugar, Shinji Sumimoto, William Tang, John Taylor, Rajeev Thakur, Anne Trefethen, Mateo Valero, Aad van der Steen, Jeffrey Vetter, Peg Williams, Robert Wisniewski, and Kathy Yelick. The International Exascale Software Project roadmap. *International Journal of High Performance Computing Applications (IJHPCA)*, 25(1):3–60, 2011. doi:10.1177/1094342010391989.

[42] Allen B. Downey. A Model For Speedup of Parallel Programs. Technical Report UCB/CSD-97-933, University of California, 1997.

[43] Kaoutar El Maghraoui, Travis J. Desell, Boleslaw K. Szymanski, and Carlos A. Varela. Dynamic Malleability in Iterative MPI Applications. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '07, pages 591–598. IEEE Computer Society, 2007. doi:10.1109/CCGRID.2007.45.

[44] Erik Elmroth and Johan Tordsson. A grid resource broker supporting advance reservations and benchmark-based resource selection. In *Proceedings of the 7th international conference on Applied Parallel Computing: state of the Art in Scientific Computing*, PARA'04, pages 1061–1070. Springer-Verlag, 2006. doi:10.1007/11558958_128.

[45] Dror G. Feitelson and Larry Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *Journal of Parallel and Distributed Computing*, 16:306–318, 1992. doi:10.1016/0743-7315(92)90014-E.

[46] Dror G. Feitelson and Larry Rudolph. Toward Convergence in Job Schedulers for Parallel Supercomputers. In *Job Scheduling Strategies for Parallel Processing*, pages 1–26. Springer-Verlag, 1996.

[47] Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel Job Scheduling - A Status Report. In *Job Scheduling Strategies for Parallel Processing, 10th International Workshop, JSSPP 2004, New York, NY, USA, June 13, 2004, Revised Selected Papers*, volume 3277 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2005. doi:10.1007/11407522_1.

[48] Laurence Field and Rizos Sakellariou. How dynamic is the Grid? Towards a quality metric for Grid information systems. In *Proceedings of the 2010 11th IEEE/ACM International*

Conference on Grid Computing, Brussels, Belgium, October 25-29, 2010, pages 113–120. IEEE, 2010. doi:10.1109/GRID.2010.5697957.

[49] Ian Foster. What is the Grid? - a three point checklist. *GRIDtoday*, 1(6), July 2002.

[50] Ian T. Foster. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. In *Euro-Par 2001: Parallel Processing, 7th International Euro-Par Conference Manchester, UK August 28-31, 2001, Proceedings*, volume 2150 of *Lecture Notes in Computer Science*, pages 1–4. Springer, 2001. doi:10.1007/3-540-44681-8_1.

[51] W. Gentzsch. Sun Grid Engine: towards creating a compute power grid. In *Proceedings of the first IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 35–36, 2001.

[52] Wolfgang Gentzsch, Denis Girou, Alison Kennedy, Hermann Lederer, Johannes Reetz, Morris Riedel, Andreas Schott, Andrea Vanni, Mariano Vazquez, and Jules Wolfrat. DEISA—Distributed European Infrastructure for Supercomputing Applications. *J. of Grid Computing*, 9(2):259–277, June 2011. doi:10.1007/s10723-011-9183-2.

[53] Tristan Glatard and Sorina Camarasu-Pop. A model of pilot-job resource provisioning on production grids. *Parallel Computing*, 37(10-11):684–692, 2011. doi:10.1016/j.parco.2011.04.001.

[54] Tom Goodale et al. A Simple API for Grid Applications (SAGA). Full Recommendation GFD-R-P.90, Open Grid Forum (OGF), 2008.

[55] Object Management Group. Common Object Request Broker Architecture (CORBA) Specification, Version 3.2, Part 2: CORBA Interoperability. Specification formal/2011-11-02, Object Management Group, 2011. Available from: http://www.omg.org/spec/CORBA/3.2/Interoperability/PDF [cited June 1, 2012].

[56] Janko Heilgeist, Thomas Soddemann, and Harald Richter. Design and Implementation of a Distributed Metascheduler. In *Proceedings of the 2009 Third International Conference on Advanced Engineering Computing and Applications in Sciences*, ADVCOMP '09, pages 63–72. IEEE Computer Society, 2009. doi:10.1109/ADVCOMP.2009.17.

[57] J. Helton, J. Johnson, C. Sallaberry, and C. Storlie. Survey of sampling-based methods for uncertainty and sensitivity analysis. *Reliability Engineering & System Safety*, 91(10-11):1175–1209, October 2006. doi:10.1016/j.ress.2005.11.017.

[58] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall. Entropy: a consolidation manager for clusters. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 41–50, New York, NY, USA, 2009. ACM. doi:10.1145/1508293.1508300.

[59] Jan Hungershofer. On the Combined Scheduling of Malleable and Rigid Jobs. In *SBAC-PAD '04: Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing*, pages 206–213, Washington, DC, USA, 2004. IEEE Computer Society. doi:10.1109/sbac-pad.2004.27.

[60] Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Jonathan Marti, and Eugenio Cesario. The XtreemFS architecture — a case for object-based file systems in Grids. *Concurrency and Computing : Practice and Experience*, 20(17):2049–2060, 2008. doi:10.1002/cpe.v20:17.

[61] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) System Interfaces, Issue 6*. IEEE, 2001. Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992, Open Group Technical Standard Base Specifications, Issue 6.

[62] Interconnects: Is Exascale End-of-the-line for Commodity Networks? Panel at the 2011 IEEE International Conference on Cluster Computing, Austin, Texas, USA, September 27, 2011.

[63] Alexandru Iosup, Mathieu Jan, Omer Ozan Sonmez, and Dick H. J. Epema. The Characteristics and Performance of Groups of Jobs in Grids. In *Euro-Par 2007, Parallel Processing, 13th International Euro-Par Conference, Rennes, France, August 28-31, 2007, Proceedings*, volume 4641 of *Lecture Notes in Computer Science*, pages 382–393. Springer, 2007. doi:10.1007/978-3-540-74466-5_42.

[64] Alexandru Iosup, Hui Li, Mathieu Jan, Shanny Anoep, Catalin Dumitrescu, Lex Wolters, and Dick H. J. Epema. The Grid Workloads Archive. *Future Generation Computer Systems*, 24(7):672–686, 2008. doi:10.1016/j.future.2008.02.003.

[65] Alexandru Iosup, Omer Ozan Sonmez, Shanny Anoep, and Dick H. J. Epema. The performance of bags-of-tasks in large-scale distributed systems. In *Proceedings of the 17th International Symposium on High-Performance Distributed Computing (HPDC-17 2008), 23-27 June 2008, Boston, MA, USA*, pages 97–108. ACM, 2008. doi:10.1145/1383422.1383435.

[66] Morris A. Jette, Andy B. Yoo, and Mark Grondona. SLURM: Simple Linux Utility for Resource Management. In *Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*, Lecture Notes in Computer Science, pages 44–60. Springer-Verlag, 2002. doi:10.1007/10968987_3.

[67] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[68] S. Kannan, M. Roberts, P. Mayes, D. Brelsford, and J. Skovira. *Workload Management with LoadLeveler*. IBM Press, 2001.

[69] Katarzyna Keahey, Maurício O. Tsugawa, Andréa M. Matsunaga, and José A. B. Fortes. Sky Computing. *IEEE Internet Computing*, 13(5):43–51, 2009. doi:10.1109/MIC.2009.94.

[70] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, Supercomputing '01, pages 37–37, New York, NY, USA, 2001. ACM. doi:10.1145/582034.582071.

[71] Bithika Khargharia, Salim Hariri, Manish Parashar, Lewis Ntaimo, and Byoung uk Kim. vGrid: A Framework For Building Autonomic Applications. In *Proceedings of the 1st International Workshop on Challenges of Large Applications in Distributed Environments*, CLADE '03, pages 19–26, Washington, DC, USA, 2003. IEEE Computer Society. doi:10.1109/CLADE.2003.1209995.

[72] Soon-Heum Ko, Nayong Kim, Joohyun Kim, Abhinav Thota, and Shantenu Jha. Efficient Runtime Environment for Coupled Multi-physics Simulations: Dynamic Resource Allocation and Load-Balancing. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 349–358, Washington, DC, USA, 2010. IEEE Computer Society. doi:10.1109/CCGRID.2010.107.

[73] Krzysztof Kurowski, Jarek Nabrzyski, Ariel Oleksiak, and Jan Węglarz. A multicriteria approach to two-level hierarchy scheduling in grids. *Journal of Scheduling*, 11(5):371–379, October 2008. doi:10.1007/s10951-008-0058-8.

[74] Sébastien Lacour, Christian Pérez, and Thierry Priol. A Network Topology Description Model for Grid Application Deployment. In *5th International Workshop on Grid Computing (GRID 2004), 8 November 2004, Pittsburgh, PA, USA, Proceedings*, pages 61–68. IEEE Computer Society, 2004. doi:10.1109/GRID.2004.2.

[75] Jonathan Ledlie, Paul Gardner, and Margo Seltzer. Network coordinates in the wild. In *Proceedings of the 4th USENIX conference on Networked systems design and implementation*, NSDI '07, pages 22–22. USENIX Association, 2007.

[76] David A. Lifka. The ANL/IBM SP Scheduling System. In *Proceedings of the Job Scheduling Strategies for Parallel Processing (JSSPP) Workshop 1995*, volume 949 of *Lecture Notes in Computer Science*, pages 295–303. Springer, 1995. doi:10.1007/3-540-60153-8_35.

[77] Justin Luitjens and Martin Berzins. Improving the performance of Uintah: A large-scale adaptive meshing computational framework. In *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*, pages 1–10. IEEE, 2010. doi:10.1109/IPDPS.2010.5470437.

[78] Piotr Luszczek, Jack Dongarra, and Jeremy Kepner. Design and Implementation of the HPCC Benchmark Suite. *CT Watch Quarterly*, 2(4A), November 2006.

[79] Daniel F. Martin, Phillip Colella, Marian Anghel, and Francis J. Alexander. Adaptive Mesh Refinement for Multiscale Nonequilibrium Physics. *Computing in Science and Eng.*, 7:24–31, 2005. doi:10.1109/mcse.2005.45.

[80] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing. Technical report, National Institute of Standards and Technology, Information Technology Laboratory, July 2009. Available from: http://www.csrc.nist.gov/groups/SNS/cloud-computing/ [cited May 18, 2012].

[81] Guillaume Mercier and Emmanuel Jeannot. Improving MPI Applications Performance on Multicore Clusters with Rank Reordering. In *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI 2011, Santorini, Greece, September 18-21, 2011. Proceedings*, volume 6960 of *Lecture Notes in Computer Science*, pages 39–49. Springer, 2011. doi:10.1007/978-3-642-24449-0_7.

[82] Stephan Merz, Martin Quinson, and Cristian Rosa. SimGrid MC: Verification Support for a Multi-API Simulation Platform. In Roberto Bruni and Juergen Dingel, editors, *31st IFIP International Conference on Formal Techniques for Networked and Distributed Systems*, volume 6722 of *Lecture Notes in Computer Science*, pages 274–288, Reykjavik, Islande, June 2011. Springer. doi:10.1007/978-3-642-21461-5_18.

[83] Hashim Mohamed and Dick Epema. Experiences with the KOALA Co-Allocating Scheduler in Multiclusters. In *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05)*, volume 2 of *CCGRID '05*, pages 784–791. IEEE Computer Society Press, 2005. doi:10.1002/cpe.1268.

[84] Christine Morin. XtreemOS: a Grid Operating System Making your Computer Ready for Participating in Virtual Organizations. In *IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC)*, 2007. doi:10.1109/ISORC.2007.62.

[85] J. T. Mościcki. Distributed analysis environment for HEP and interdisciplinary applications. *Nuclear Instruments & Methods in Physics Research*, 502:426–429, 2003. doi:10.1016/S0168-9002(03)00459-5.

[86] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, 2001. doi:10.1109/71.932708.

[87] H. Nakada, S. Matsuoka, K. Seymour, J. Dongorra, C. Lee, and H. Casanova. A GridRPC Model and API for End-User Applications. Full Recommendation GFD-R.052, Open Grid Forum (OGF), 2007.

[88] Anand Natrajan, Marty Humphrey, and Andrew S. Grimshaw. Capacity and Capability Computing Using Legion. In *International Conference on Computational Science*, volume 2073 of *Lecture Notes in Computer Science*, pages 273–283. Springer, 2001. doi:10.1007/3-540-45545-0_36.

[89] Javier Navaridas, Jose Antonio Pascual, and José Miguel-Alonso. Effects of Job and Task Placement on Parallel Scientific Applications Performance. In *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2009, Weimar, Germany, 18-20 Febuary 2009*, pages 55–61. IEEE Computer Society, 2009. doi:10.1109/PDP.2009.53.

[90] Hien Nguyen Van, Frederic Dang Tran, and Jean-Marc Menaud. Autonomic virtual resource management for service hosting platforms. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, CLOUD '09, pages 1–8. IEEE Computer Society, 2009. doi:10.1109/CLOUD.2009.5071526.

[91] Ramon Nou, Jacobo Giralt, Julita Corbalán, Enric Tejedor, Josep Oriol Fitó, Josep M. Pérez, and Toni Cortes. XtreemOS Application Execution Management: A scalable approach. In *Proceedings of the 2010 11th IEEE/ACM International Conference on Grid Computing*, pages 49–56. IEEE, 2010. doi:10.1109/GRID.2010.5697954.

[92] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The Eucalyptus Open-source Cloud-computing System. In *Proceedings of Cloud Computing and Its Applications*, 2008.

[93] Ralf Nyrén, Andy Edmonds, Alexander Papaspyrou, and This Metsch. Open Cloud Computing Interface - Core. Proposed Recommendation GFD-P-R.183, Open Grid Forum (OGF), 2011.

[94] Anne-Cécile Orgerie, Laurent Lefèvre, and Jean-Patrick Gelas. How an experimental Grid is used: The Grid'5000 case and its impact on energy usage. In *Poster at the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid'2008)*, 2008. Available from: http://www.ens-lyon.fr/LIP/RESO/energy_grid/ [cited June 28, 2012].

[95] Alfred Park and Richard Fujimoto. A scalable framework for parallel discrete event simulations on desktop grids. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, GRID '07, pages 185–192, Washington, DC, USA, 2007. IEEE Computer Society. doi:10.1109/GRID.2007.4354132.

[96] Ken Pepple. *Deploying OpenStack*. O'Reilly Media, 2011.

[97] Tomasz Plewa, Timur Linde, and V. Gregory Weirs. *Adaptive Mesh Refinement – Theory and Applications*, volume 41 of *Lecture notes in computational science and engineering*. Springer, 2005.

[98] André Ribes and Christian Caremoli. Salome platform component model for numerical simulation. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007), 24-27 July 2007, Beijing, China*, pages 553–564. IEEE Computer Society, 2007. doi:10.1109/COMPSAC.2007.185.

[99] Uwe Schwiegelshohn and Ramin Yahyapour. Analysis of first-come-first-serve parallel job scheduling. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, SODA '98, pages 629–638. Society for Industrial and Applied Mathematics, 1998.

[100] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. A Structured Overlay for Multi-dimensional Range Queries. In *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, chapter 54, pages 503–513. Springer Berlin Heidelberg, 2007. doi:10.1007/978-3-540-74466-5_54.

[101] Warren Smith, Ian T. Foster, and Valerie E. Taylor. Scheduling with Advanced Reservations. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS '00)*, pages 127–132. IEEE Computer Society, 2000.

[102] Omer Ozan Sonmez, Bart Grundeken, Hashim H. Mohamed, Alexandru Iosup, and Dick H. J. Epema. Scheduling Strategies for Cycle Scavenging in Multicluster Grid Systems. In *9th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGrid 2009, Shanghai, China, 18-21 May 2009*, pages 12–19. IEEE Computer Society, 2009. doi:10.1109/CCGRID.2009.46.

[103] Borja Sotomayor, Kate Keahey, and Ian Foster. Combining batch execution and leasing using virtual machines. In *Proceedings of the 17th international symposium on High performance distributed computing*, HPDC '08, pages 87–96, New York, NY, USA, 2008. ACM. doi:10.1145/1383422.1383434.

[104] Borja Sotomayor, Ruben S. Montero, Ignacio M. Llorente, and Ian Foster. Virtual Infrastructure Management in Private and Hybrid Clouds. *IEEE Internet Computing*, 13:14–22, 2009. doi:10.1109/MIC.2009.119.

[105] Srividya Srinivasan, Vijay Subramani, Rajkumar Kettimuthu, Praveen Holenarsipur, and P. Sadayappan. Effective Selection of Partition Sizes for Moldable Scheduling of Parallel Jobs. In *Proceedings of the 9th International Conference on High Performance Computing*, HiPC '02, pages 174–183. Springer-Verlag, 2002. doi:10.1007/3-540-36265-7_17.

[106] Sudha Srinivasan, Savitha Krishnamoorthy, and P. Sadayappan. Robust scheduling of moldable parallel jobs. *International Journal of High Performance Computing and Networking*, 2(2-4):120–132, 2004. doi:10.1504/IJHPCN.2004.008913.

[107] Vijay Subramani, Rajkumar Kettimuthu, Srividya Srinivasan, and P. Sadayappan. Distributed Job Scheduling on Computational Grids Using Multiple Simultaneous Requests. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, HPDC '02, pages 359–366. IEEE Computer Society, 2002. doi:10.1109/HPDC.2002.1029936.

[108] Rajesh Sudarsan and Calvin J. Ribbens. ReSHAPE: A Framework for Dynamic Resizing and Scheduling of Homogeneous Applications in a Parallel Environment. In *Proceedings of the 2007 International Conference on Parallel Processing*, ICPP '07. IEEE Computer Society, 2007. doi:10.1109/ICPP.2007.73.

[109] Yoshio Tanaka, Hidemoto Nakada, Satoshi Sekiguchi, Toyotaro Suzumura, and Satoshi Matsuoka. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Journal of Grid Computing*, 1(1):41–51, 2003. doi:10.1023/A:1024083511032.

[110] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the Condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005. doi:10.1002/cpe.v17:2/4.

[111] Dan Tsafrir, Yoav Etsion, Dror G. Feitelson, and Scott Kirkpatrick. System noise, OS clock ticks, and fine-grained parallel applications. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 303–312, New York, NY, USA, 2005. ACM. doi:10.1145/1088149.1088190.

[112] Dan Tsafrir, Keren Ouaknine, and Dror G. Feitelson. Reducing Performance Evaluation Sensitivity and Variability by Input Shaking. In *15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2007), October 24-26, 2007, Istanbul, Turkey*, pages 231–237. IEEE Computer Society, 2007. doi:10.1109/MASCOTS.2007.58.

[113] Etienne Urbah, Péter Kacsuk, Zoltan Farkas, Gilles Fedak, Gabor Kecskemeti, Oleg Lodygensky, Csaba Attila Marosi, Zoltán Balaton, Gabriel Caillat, Gabor Gombás, Adam Kornafeld, József Kovács, Haiwu He, and Róbert Lovas. EDGeS: Bridging EGEE to BOINC and XtremWeb. *Journal of Grid Computing*, 7(3):335–354, 2009. doi:10.1007/s10723-009-9137-0.

[114] Gregor von Laszewski, Geoffrey C. Fox, Fugang Wang, Andrew J. Younge, Archit Kulshrestha, and Greg Pike. Design of the FutureGrid Experiment Management Framework. In *Proceedings of Gateway Computing Environments 2010 at Supercomputing 2010*, New Orleans, LA, Nov 2010. IEEE.

[115] Asim YarKhan, Keith Seymour, Kiran Sagi, Zhiao Shi, and Jack Dongarra. Recent Developments in GridSolve. *International Journal of High Performance Computing Applications*, 20(1):131–141, 2006. doi:10.1177/1094342006061893.

[116] Nezih Yigitbasi and Dick H. J. Epema. Overdimensioning for Consistent Performance in Grids. In *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, CCGrid 2010, 17-20 May 2010, Melbourne, Victoria, Australia*, pages 526–529. IEEE, 2010. doi:10.1109/CCGRID.2010.44.

[117] Jia Yu and Rajkumar Buyya. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Record*, 34(3):44–49, 2005. doi:10.1145/1084805.1084814.

[118] Yulai Yuan, Guangwen Yang, Yongwei Wu, and Weimin Zheng. PV-EASY: a strict fairness guaranteed and prediction enabled scheduler in parallel job scheduling. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 240–251, New York, NY, USA, 2010. ACM. doi:10.1145/1851476.1851505.

APPENDIX D

Webography

[119] Amazon EC2 Spot Instances. Available from: http://aws.amazon.com/ec2/spot-instances/ [cited July 31, 2012].

[120] Amazon Elastic Compute Cloud (EC2) Documentation. Available from: http://aws.amazon.com/documentation/ec2/ [cited March 19, 2012].

[121] Amazon Web Services. Available from: http://aws.amazon.com [cited June 26, 2012].

[122] Paul E. Black. Dictionary of Algorithms and Data Structures. Available from: http://xlinux.nist.gov/dads/ [cited July 10, 2012].

[123] BOINC Stats: Detailed stats. Available from: http://boincstats.com/en/stats/-1/project/detail/overview [cited June 26, 2012].

[124] Computation Center of IN2P3: Computing farms. Available from: http://cc.in2p3.fr/Fermes-de-calcul [cited May 16, 2012].

[125] Cycle Computing. Lessons learned building a 4096-core Cloud HPC Supercomputer. Available from: http://blog.cyclecomputing.com/2011/03/cyclecloud-4096-core-cluster.html [cited March 19, 2012].

[126] ANR COOP Project. Available from: http://coop.gforge.inria.fr/ [cited September 26, 2012].

[127] French ANR DiscoGrid project, 2005–2009. Available from: http://www-sop.inria.fr/nachos/team_members/Stephane.Lanteri/DiscoGrid/.

[128] EGI: Figures and utilisation. Available from: http://www.egi.eu/infrastructure/operations/figures_and_utilisation/ [cited June 26, 2012].

[129] Dror G. Feitelson. The Parallel Workloads Archive. Available from: http://www.cs.huji.ac.il/labs/parallel/workload [cited June 1, 2012].

[130] Dror G. Feitelson. *Workload Modeling for Computer Systems Performance Evaluation, version 0.34*. 2011. Available from: http://www.cs.huji.ac.il/~feit/wlmod/ [cited July 31, 2012].

[131] Google App Engine. Available from: https://developers.google.com/appengine [cited June 26, 2012].

[132] The Graph500 List. Available from: http://www.graph500.org [cited June 26, 2012].

[133] The Green500 List. Available from: http://www.green500.org [cited June 26, 2012].

[134] greenlet: Lightweight concurrent programming. Available from: http://greenlet.readthedocs.org/en/latest/index.html [cited July 12, 2012].

[135] National Institute for Computational Sciences: Running Jobs on Kraken. Available from: http://www.nics.tennessee.edu/computing-resources/kraken/running-jobs [cited June 29, 2012].

[136] Network Coordinates Research Group. Available from: http://www.eecs.harvard.edu/~syrah/nc/ [cited September 19th, 2012].

[137] Nimbus Project. Available from: http://www.nimbusproject.org/ [cited March 19, 2012].

[138] Antoine Petitet, Clint Whaley, Jack Dongarra, and Andy Cleary. HPL – A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. Available from: http://www.netlib.org/benchmark/hpl/ [cited May 16, 2012].

[139] Partnership for Advanced Computing in Europe. Available from: http://www.prace-project.eu/ [cited September 20th, 2012].

[140] Python Programming Language. Available from: http://www.python.org [cited July 12, 2012].

[141] RackSpace Cloud. Available from: http://www.rackspace.com/cloud/ [cited June 26, 2012].

[142] Rod Schultz. SLURM: Advanced Usage. Presentations from SLURM User Group Meeting, September 2011. Available from: http://www.schedmd.com/slurmdocs/slurm_ug_2011/Advanced_Usage_Tutorial.pdf [cited May 25, 2012].

[143] Herb Sutter. Welcome to the Jungle, 2011. Available from: http://herbsutter.com/welcome-to-the-jungle/.

[144] Texas Advanced Computing Center: High Performance Computing (HPC) Systems. Available from: http://www.tacc.utexas.edu/resources/hpc/ [cited June 26, 2012].

[145] The Distributed ASCI Supercomputer 3. Available from: http://www.cs.vu.nl/das3/ [cited March 21, 2012].

[146] TOP500 Supercomputing Sites. Available from: http://www.top500.org [cited June 26, 2012].

[147] TORQUE Resource Manager. Available from: http://www.adaptivecomputing.com/products/open-source/torque/ [cited March 19, 2012].

[148] David A. Wheeler. SLOCCount. Available from: http://www.dwheeler.com/sloccount/ [cited August 1st, 2012].

[149] Windows Azure: Microsoft's Cloud Platform. Available from: http://www.windowsazure.com [cited June 26, 2012].

[150] XSEDE: High Performance Computing. Available from: https://www.xsede.org/high-performance-computing [cited September 22, 2012].