Advanced Simulation for Resource Management

Adrien Faure

Université Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Adrien Faure

Thèse dirigée par **Denis TRYSTRAM**
et codirigée par **Olivier RICHARD**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Simulation avancée pour la gestion de ressources des superordinateurs

Advanced Simulation for Resource Management

Thèse soutenue publiquement le **2 décembre 2020**,
devant le jury composé de :

Georges DA COSTA
Maître de conférence, IRIT, Université de Toulouse III, France, Rapporteur
Frédéric SUTER
Directeur de Recherche, IN2P3, France, Rapporteur
Yves DENNEULIN
Professeur des universités, LIG, Grenoble INP, France, Examinateur, Président
Adrien LÈBRE
Professeur à IMT Atlantique, France, Examinateur
Pascale ROSSÉ-LAURENT
Architecte Logiciel à Atos, France, Examinateur
Olivier RICHARD
Maître de conférences, LIG, Univ. Grenoble Alpes, France, Co-Directeur de thèse
Denis TRYSTRAM
Professeur des universités, LIG, Grenoble INP, France, Directeur de thèse

Nothing takes the heart out of a man more than the expectation of failure.

— **Robin Hobb**

Remerciements / Acknowledgements

La réalisation de cette thèse s'est déroulée dans le cadre d'une convention CIFRE, guidée par lAssociation Nationale Recherche Technologie (ANRT), entre le Laboratoire d'Informatique de Grenoble (LIG) et la société Atos.

Je voudrais remercier, Frédéric Suter et Georges Da Costa pour leur relecture attentive de mon mémoire. Écrire fut une tâche difficile et chaque remarque fut juste et utile. Merci également à tous les membres du jury pour leur questions, et leur regard critique sur mon travail.

Je souhaite remercier mes encadrants, Olivier Richard, Pascale Rossé-Laurent et Denis Trystram qui, nonobstant un début mouvementé, ont réussi à m'aiguiller grâce à leurs conseils scientifique, mais également par leurs encouragements et leur soutien. Merci à Millian et Michael pour leur patience et leurs conseils scientifique avec qui travailler pendant ces années fut un réel plaisir. Merci à Valentin pour toutes ses idées folles et de m'avoir motivé à rependre le sport. Merci à Clément, Raphaël, Salah, Stéphane, Théophile, Paul, Vincent, Sebastian, Mohammed, Danilo, Pedro, Tristan, Achal, Léo, Saurabh pour toutes les discussions autour d'un café. Merci à toute l'équipe Datamoris (DATAMOve et PolaRIS) où les portes des bureaux restent ouvertes, témoignant de votre acceuil pendant ses trois années. Plus généralement, merci à tous mes collègues du laboratoire : permanents, ingénieurs, doctorants et stagiaires pour tous les moments conviviaux. Merci à Valérie, et Annie pour votre aide et votre support au quotidien. Je tiens à remercier Marc, Guillaume et Guillaume, Emmanuel, Piotr, Florent, Nicolas, Nadya et Pierre, chaque pause café était accompagné de vos blagues et de votre bonne humeur. Merci à toute l'équipe Slurm, et à l'équipe Runtime à Bull, pour tous les bons moments passé au café, et à faire le tour. Finalement, merci à David Glesser, Yiannis Georgiou et Andry Razafinjatovo avec qui tout à commencé.

Merci à Nicolas (aka Jaja) pour son sens de l'humour et les grand moments de Gaming. Merci à Antoine pour son intarissable source de gentillese et de bon-sens. Merci à ma famille, et tout particulièrement à mes parents, mon frère et mes grand-parents de m'avoir soutenu pendant toutes ses années. Un grand merci à Calliane, qui n'a cessé de croire en moi et qui m'a soutenu et supporté, même dans les moments difficiles.

The experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations [1]

The workload log from the CEA Curie system was graciously provided by Joseph Emeras. The workload log from ANL Intrepid was graciously provided by Susan Coghlan (smc@alcf.anl.gov) from ALCF at ANL and Narayan Desai (desai@mcs.anl.gov) from MCS at ANL. It was converted to SWF and made available by Wei Tang (wtang6@iit.edu) from Illinois Institute of Technology The workload log from the RICC cluster was graciously provided by Motoyoshi Kurokawa (motoyosi@riken.jp).

[1] https://www.grid5000.fr.

Abstract / Résumé

Abstract

High-Performance Computing (HPC) provides the computational power dedicated to solving complex problems of our society. HPC computers are large scale and distributed infrastructures composed of several thousands of computing cores. The management of theses systems is left to unique software: the Resource and Job Management System (RJMS). The objective of the RJMS is multiple: Managing the physical infrastructure, and handling the user requests to access to the computing power. The scheduling algorithm is the cornerstone of the RJMS, it decides where and when the user's jobs will be executed. Scheduling is a difficult problem; to manage large scale platforms RJMS needs to dispose of efficient yet scalable scheduling heuristics Evaluating and testing new scheduling algorithms is crucial before releasing it in production. Any failure can have a dramatic impact on the HPC platform leading to wasted time, energy, and resources. The lack of a platform dedicated experiments and tests compels RJMS designers and HPC center's administrators to use different tools and methodologies to evaluate new algorithms.

In the first part of this dissertation, we present and evaluate a new scheduling heuristics with job redirection. The evaluation is done using a large simulation campaign, it results that by redirecting jobs can improve the efficiency of the scheduling. In the second part, we focus on and extend the tools and methodologies available to experiment with RJMS. This part is twofold: Firstly, we propose to extend scheduling simulations with job models to simulate network contention between jobs. Secondly, we propose new tools that enable experiment with production RJMS without the need for an HPC platform. This dissertation aims to broaden the experimental landscape of tools and methodologies to experiment with RJMS and therefore help the release in the production of new scheduling algorithms.

Résumé

Les superordinateurs sont des systèmes mutualisant la puissance de milliers de coeurs de calculs dédiés à la résolution des problèmes compliqués de notre société. Le gestionnaire de ressources est un système distribué et complexe chargé de la gestion de ses ressources de calculs. Son rôle est multiple : Gérer la plateforme physique et traiter les requêtes d'accès des utilisateurs au superordinateur. La pierre angulaire du gestionnaire de ressources est son algorithme d'ordonnancement des requêtes des utilisateurs. L'ordonnancement est un problème difficile ; pour gérer efficacement un superordinateur le gestionnaire de ressources doit disposer d'heuristiques d'ordonnancement efficaces permettant de prendre des décisions pertinentes sur des milliers de ressources de calculs. Évaluer et tester de nouvelles heuristiques est fondamental avant de pouvoir les utiliser dans un système en production. Toute panne induite par une nouvelle politique peut avoir des conséquences importantes sur la qualité de service du superordinateur. Il est ainsi nécessaire de disposer d'outils et méthodes dédiés à l'évaluation des algorithmes d'ordonnancement.

La première partie de ce document présente un nouvel algorithm d'ordonnancement, ainsi que son évaluation par le biais de la simulation. L'algorithme en question repose sur la possibilité de rediriger les programmes des utilisateurs en cours d'exécution. L'évaluation est réalisée par le biais d'une large campagne de simulation, et montre que rediriger des programmes permet d'améliorer les performances de l'ordonnancement. L'objectif principal de la seconde partie de ce document est de proposer et développer de nouveaux outils et méthodes pour l'évaluation des gestionnaires de ressources. Cette seconde partie est elle même divisée en deux arcs : Nous proposons dans un premier temps d'étendre les techniques de simulations d'algorithmes d'ordonnancement avec des modèles dédiés aux programmes permettant ainsi la simulation d'interférences réseaux entre les différents programmes. Dans un second temps, nous proposons deux nouvelles approches pour créer des expériences sur un seul ordinateur, en se basant directement sur de vrais gestionnaires de ressources. L'objectif de ces travaux est d'étendre le paysage expérimental des outils et méthodologies nécessaires à l'évaluation de nouveaux algorithmes d'ordonnancement.

Contents

Abstract / Résumé v

Contents vii

1 Introduction 1
- 1.1 Background . . . 1
- 1.2 Resource and Job Management Systems . . . 2
  - 1.2.1 Scheduling Heuristics for HPC . . . 4
- 1.3 RJMSs Evaluation: Workloads and Metrics . . . 5
  - 1.3.1 Workloads and Metrics . . . 6
  - 1.3.2 Methodology for the Evaluation of RJMSs . . . 6
- 1.4 Contributions . . . 8

2 Experimental Study of RJMS: Methods and State of the Art 11
- 2.1 Introduction . . . 11
- 2.2 Using RJMS Models . . . 14
- 2.3 Experiment with Real RJMSs . . . 17
  - 2.3.1 Real RJMS with a Functional Platform . . . 17
  - 2.3.2 RJMS Hybridization . . . 20
- 2.4 Choosing the Adapted Methodology . . . 22
- 2.5 Conclusion . . . 23

3 On-line Scheduling with Redirection for Independent Jobs 25
- 3.1 Introduction . . . 25
- 3.2 Definition and Notation . . . 26
- 3.3 Related Work . . . 27
- 3.4 Scheduling Parallel Jobs in HPC Environments . . . 28
- 3.5 Scheduling Parallel Jobs with Redirection . . . 29
  - 3.5.1 General Description of the Redirection . . . 29
  - 3.5.2 Dealing with Parallel Jobs . . . 29
  - 3.5.3 Execution . . . 30
- 3.6 Experimental Settings . . . 32
  - 3.6.1 Simulation and Inputs . . . 32
  - 3.6.2 The Workloads . . . 33

3.6.3 Redirection Parameters: α and θ . . . 34
3.6.4 Reproducibility . . . 35
3.7 Experimental Results . . . 35
3.7.1 Parameters Tuning . . . 35
3.7.2 Comparison to EASY Back-filling . . . 37
3.8 Conclusion . . . 40

4 Scheduling Simulation with Job's Models **43**
4.1 Introduction . . . 43
4.2 The Batsim Approach . . . 45
4.2.1 SimGrid . . . 45
4.2.2 SimGrid Provided Models . . . 47
4.2.3 Batsim: Infrastructure Simulator for Resource Management . 50
4.3 Job Execution Profiles . . . 52
4.3.1 Profile Types . . . 52
4.3.2 Profiles Evaluation . . . 54
4.4 Conclusion . . . 60

5 Ptask Model Validation **63**
5.1 Introduction . . . 63
5.2 Experiment Methodology . . . 64
5.3 Parallel Matrix Multiplication (PDGEMM) . . . 64
5.3.1 PDGEMM Algorithm . . . 65
5.3.2 Matrix Block Subdivision . . . 67
5.3.3 PDGEMM Resources Consumption Behavior . . . 67
5.4 Real Experimental Setup . . . 68
5.4.1 Platform and Nodes Configuration . . . 68
5.4.2 PDGEMM and MPI Configuration . . . 68
5.4.3 Controlled Interferences . . . 69
5.4.4 Monitoring . . . 71
5.5 Results and Data Analysis of the Real Executions . . . 71
5.5.1 Results Analysis for *Paravance* . . . 71
5.5.2 Results Analysis for *Grisou* . . . 76
5.5.3 Difference Between *Grisou* and *Paravance* . . . 76
5.6 PDGEMM in Simulation . . . 79
5.6.1 SimGrid Platform, Calibration and Interference . . . 79
5.6.2 *Ptask* Generation . . . 80
5.7 Comparison between the *Ptask* Model and Reality . . . 81
5.8 Interference Analysis . . . 83
5.8.1 Theoretical Interference Model . . . 84
5.8.2 Theoretical Model Calibration and Results . . . 84
5.9 Discussion . . . 86

5.10 Conclusion . . . 88
5.10.1 *Ptask* calibration . . . 88
5.10.2 Scheduling Simulations . . . 89

6 Study RJMS with the Emulation Approach 91
6.1 Introduction . . . 91
6.2 The *Simunix* Approach . . . 93
6.2.1 Project Historic . . . 94
6.2.2 *Simunix* Use Case: Slurm Emulation . . . 95
6.3 Batsky . . . 96
6.4 Discussion . . . 99
6.5 Conclusion . . . 100

7 Tools for Emulation: Interception, Remote SimGrid and *sgwrap* 103
7.1 Introduction . . . 103
7.2 Interception Methods . . . 103
7.3 Remote SimGrid . . . 105
7.3.1 RSG Simulation Concepts . . . 105
7.3.2 Implementation Details . . . 106
7.3.3 Related Work . . . 107
7.4 C Standard Library Interceptions . . . 108
7.4.1 Choosing the Intercepted Functions . . . 109
7.4.2 System Time Interception . . . 109
7.4.3 BSD Socket Interception . . . 109
7.4.4 System Process Interception . . . 112
7.4.5 Threads Interception . . . 114
7.5 Conclusion . . . 115

8 Reproducibility of Experiments with Variations 117
8.1 Software Development Workflow and Reproducibility . . . 118
8.2 Reproducible Software Environments with Nix . . . 120
8.3 Related Work . . . 121
8.4 Discussion . . . 122
8.5 Conclusion . . . 123

9 Conclusion 125
9.1 Contributions and Future Work . . . 125
9.1.1 Scheduling with Job Redirection . . . 125
9.1.2 Simulation Model with Job Resources Consumption . . . 126
9.1.3 Experimenting with Real RJMS: *Simunix* and Batsky . . . 128
9.1.4 Reproducibility of Experiments with Variation . . . 129

A Appendix A3

A.1 PDGEMM A3
A.1.1 Nonblocking Broadcast A3
A.2 Reproduce Experiments A5

Bibliography **A15**

1 Introduction

1.1 Background

The continuous need for computational power has stimulated a global effort to build powerful High-Performance Computing (HPC) platforms, accommodating thousands of cores, and this number still increases. In June 2020, the most powerful super calculator, Fugaku, achieves its peak of performance with $513,854.7TFlop/s$ — $513,854.7 * 10^{16} flops$ with more than 7 millions of cores ($7,299,072$ cores). HPC is a field in constant evolution driven by the continuous need for performances, leading manufacturers to incorporate always more processing cores. Traditionally, the computing cores are distributed among independent computers referred to as the computing nodes, connected with a dedicated high-performance network: the interconnect. With the emergence of new hardware, such as graphics processing unit (GPU) to accelerate the computations and IO accelerators, recent computing platform becomes heterogeneous, in the sense that they incorporate various kinds of resources. In the past decade, tremendous efforts have been made to, both have efficient exploitation of the computing platforms and to cope with the issues induced by the increasing size and complexity of platforms. More specifically these issues are the resiliency, the energy consumption, and the data management [Don+11].

In practice, few applications use all the nodes of a cluster, instead, the resources are shared among multiple applications. To access to the computing platform, the end-users need to submit their applications to a Resource and Job Management System (RJMS), which is in charge of matching the user's requests with the available computing power. To deliver such computing power, RJMSs are complexes and distributed software that need, on the one hand, to schedule user requests and allocate the computing nodes to meet the demand, and on the other hand, to perform all the necessary operations to manage the platform, such as node monitoring and launching the user applications. With the increasing size of the HPC platforms, RJMSs need to be scalable to deliver always more computing power. To cope with this issue, RJMSs leverage the fact that common HPC application requirements can be easily expressed by a simple number of required cores for a fixed amount of time, to use efficient and yet simple scheduling heuristics.

The emerging of a diversity of new usages and practices of the HPC platforms, such as the apparition of large scientific datasets that need to be processed, interactive notebooks or the training of machine learning models, along with the aforementioned heterogeneity of the computing resources, impose RJMSs to deal with more

complex user demands as well as a wider number of applications [Asc+18; Mer19]. Additionally, other issues and challenges have been addressed at the level of the RJMS — the platform management level — such as managing the energy of the computing platform [Poq17; Kas19], allocating different kinds of resources such as Input/Output (IO) accelerators and GPUs [Bea+20], or monitoring applications to detect potential performance issues [Eme13]. To continue to provide a good quality of service in respect of the aforementioned evolutions RJMS source code is enhanced to integrate new resource types, and a wide variety of scheduling algorithms or resources management algorithms to manage next-generation platforms.

Next-generation platforms will be heterogeneous, both in the resources available for the users and in the kind of workloads that will be imposed on it. Due to the price of the different parts of the cluster, such as the interconnect, the number of cores, and the number (and type) of accelerators, building a platform tailored for a specific company is puzzling. The RJMS in charge of resource management, can help to evaluate the appropriate platform sizing for a specific workload. For instance, the cost of the cluster's interconnect could be reduced for facilities that mainly use the computing platform for data-intensive applications, the saving can be used for a more performant data management. However, the platform performance is largely impacted by the ability of the RJMS to efficiently allocate the resources, with poor resources allocation, the applications of the platform can be interfering with each other [Qia+17; Bha+13; Smi+18].

Current RJMSs need to dispose of fast and efficient heuristics to deal with a large number of resources and user requests while dealing with more criteria. The development of new heuristics for resources management is necessary to compel with the production constraints of current and next-generation clusters. Releasing new heuristics in production is complicated as any failure or performance loss may harm the cluster's quality of service. Hence, cluster administrators need convincing arguments about the beneficial impact of new heuristics on their clusters.

During the evaluation of such heuristic, it appeared that proposing a convincing evaluation is a difficult problem, it requires resources and methodologies not always available. A part of this dissertation is dedicated to the resources and methodologies used for the evaluation of new RJMS heuristics for scheduling and resource management.

1.2 Resource and Job Management Systems

The Resource and Job Management System (RJMS) is a middleware responsible to manage the availability and the resource allocations of a supercomputer for the execution of the applications. From a high-level point of view, the role of the RJMS

is to act as an interface between the physical platform and the users to provide the technical requirements to use the computing resources.

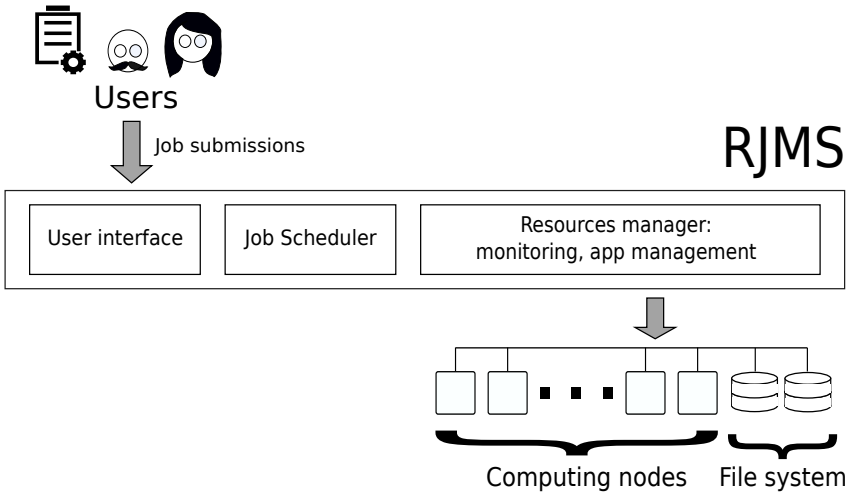


Figure 1.1.: Overview of a Resource and Job Management System. The users submit their applications, called the jobs, to the RJMS. The jobs scheduler finds a number of nodes available according to the job's requirements and attributes a starting date at each one of them. The resources manager manages the platform, monitors the nodes, and controls the job's executions

The RJMS presents the interface to use the computing resources of a cluster: the users need to send a request to the RJMS. The request is referred to as a job and contains information about the application requirements. In HPC, the most basic requirements are the amount of computing power, expressed in cores, and an upper bound on the job's execution time to prevent failing jobs to never release their resources. This upper bound is called the job's *walltime* and is specific to HPC systems — in case of a job's execution exceeds its *walltime*, the job is killed.

The RJMS matches the user demand's requirements to the available computing nodes, it has to create an allocation for each job submitted. An allocation is the attribution of a number of available resources, corresponding to the job requirements, and a time at which the job starts its execution. Finding an allocation for each job is the objective of the scheduling algorithm (or scheduling policy) of the RJMS described in the next section.

The RJMS is also responsible for the successful execution of the decisions taken by the scheduling policy on the physical platform. Thus, the RJMS is in charge of launching the jobs on the selected nodes, killing them when it is necessary, monitoring the computing nodes and tracking their status (availability, failures, on use etc.). Performing these tasks is technical, leading RJMSs to be distributed and complex software composed of thousands of lines of code — for instance, the last Slurm version (20.02), a well established RJMS, has more than four hundred thousands lines of code.

1.2.1 Scheduling Heuristics for HPC

The RJMS of the platform handles the user submissions and is in charge to allocate the computing resources to the user's job. The allocation is created by matching both the user's demand, such as a number of computing nodes and the availability of the platform's resources. If not enough resources are available on the computing platform, the jobs are queued up into a waiting list until enough resources are available. The role of the scheduling algorithm is to find a starting date for each – valid – user's job matching the job's resources requirement.

All job are independents and requests a number of computing resources q_j for a certain duration called the *walltime*. In case the *walltime* is not provided by the user the system uses a default one (usually one hour). Any job exceeding its walltime is killed by the RJMS, and its resources become available for other jobs to execute. The scheduler is only aware of a job at is release time r_j (i.e., its submission date, the time at which the user sends the request). The scheduler is online, it means that it doesn't know the future jobs and takes decisions only based on the jobs already submitted. It is interesting to mention that the RJMS is (most of the time) not aware of the underlying program that is executed by the user, the RJMS only provides the required resources necessary for the job's execution.

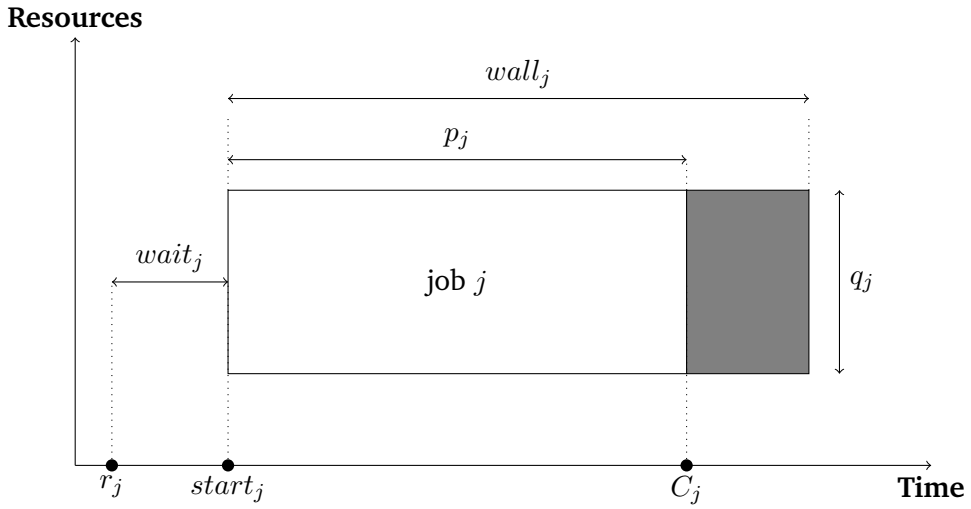


Figure 1.2.: Exhaustive job definition.

Figure 1.2 shows an exhaustive vision of job j. Because the *walltime* is an upper limit on the job execution time, it does not necessarily match its processing time. The scheduler only knows the actual processing time p_j when the job completes (at C_j). The grey part of the job depicts the fact that the job can complete (or crash) before it reaches its *walltime*. The waiting time $wait_j$ is the time between the release time and the beginning of its execution $start_j$. Finally, the flow time F_j of job j is the total time it spends on the system, $F_j = wait_j + p_j$.

For a set of parallel jobs, the goal of the scheduling algorithm is to provide for each submitted job j a starting date r_j with the number of required resources q_j. Different metrics, detailed in the next section, can be used to evaluate the performances of a scheduling policy (1.3). A two-axis chart can represent the result of scheduling policy, figure 1.3 provides such an example.

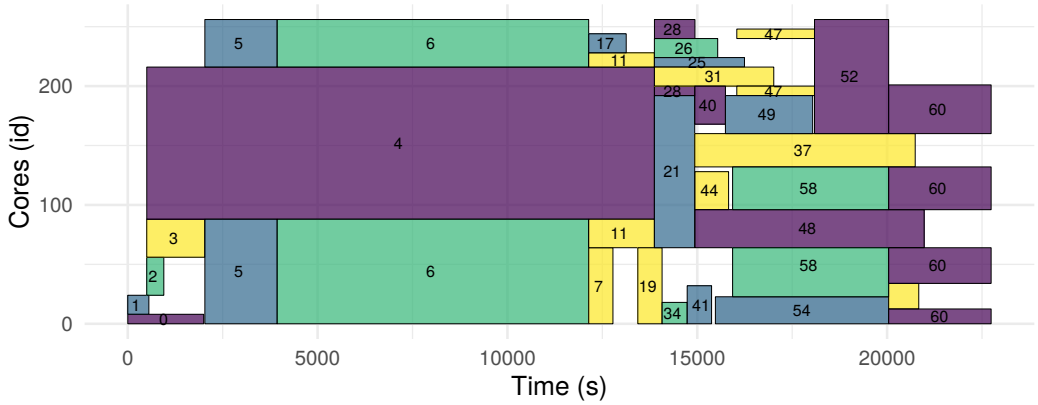


Figure 1.3.: Example of a Gantt chart. It depicts the result a of scheduling policy. The jobs are represented by the rectangles, the *x-axis* shows the time, while the *y-axis* shows the set of allocated resources.

1.3 RJMSs Evaluation: Workloads and Metrics

Integrating recent work in a production RJMS is difficult, any failure or performance loss has a high cost on the platform's quality of service and represents a loss of money and energy. Therefore, it becomes necessary to evaluate a new development or a new algorithm before releasing it in production. The lack of dedicated resources for the experiments makes the evaluation of new algorithms and developments difficult, and the adoption of works from the state of the art remains marginal. Alternatively, creating scheduling algorithms and scalable resources management techniques for next-generation platforms is complicated without the platforms at issue. To cope with this, one needs to dispose of tools and methodologies able to help the evaluations of next-generation RJMSs.

In production mode, the users share the resources. In this situation, the peak performance of the computing cluster is not the only relevant factor to measure the overall platform performance, as it doesn't reflect how the users access the computing platform. Therefore it becomes necessary to evaluate the performances of the RJMS in charge of sharing the resources.

1.3.1 Workloads and Metrics

On a production platform the users submissions create a set of jobs that needs to be executed on the platform, this is the workload. In other words, the workload is the list jobs that needs to be executed on a computing platform.

The efficiency of an RJMS can be evaluated according to its ability to handle the workload in respect of different metrics — i.e., to schedule all the jobs of the workload, and allocate them to the resources of the computing platform. Each metric measures a different aspect of the RJMS. Two categories of metric exist.

The **user level** metrics evaluate the performances of the RJMS from the user perspective:

- The *waiting time* of a job is the time that a job spends in the system, from its submission date to the beginning of its execution.
- The *slowdown* (or *stretch*) of a job is the total time it spends on the system (waiting time plus execution time), divided by its execution time [Fei01a]. This metric is based on the observation that a long waiting time has not the same impact on the user's satisfaction according to the job's execution time. If a job should run for a long period, it is more acceptable to wait longer before its execution.
- The *throughput* of the RJMS is the number of jobs that the RJMS can process in a period (usually one sec). It represents the reactivity of the system during peaks of activity.

The metrics are then aggregated to be representative of the RJMS performance in respect of the entire workload (i.e., all the jobs). For instance, the average waiting time is defined as the sum of all the job's waiting times divided by the number of executed jobs.

The **system-level** metrics evaluate the performance of the RJMS from the platform perspective. The *utilization* defines the RJMS ability to ensure a low idle time of the nodes of the computing platform, and therefore to fill the cluster. One can also compute different metrics, such as the total energy consumption of the platform [Gle16].

1.3.2 Methodology for the Evaluation of RJMSs

Evaluating a new scheduling algorithm is done with experiment campaigns. An experiment features the studied system (the RJMS), or a model of the studied system, and reproduces the production environment in the experimental setup. However, the evaluations of scheduling policies, using a real RJMS or a model of an RJMS,

rely on a common methodology: The *play* — or *replay*, in case of one uses traces of real RJMS — of workloads.

On a production cluster, the workload is directly generated by the user submissions. The RJMS performances can be evaluated using the data available on the cluster [Eme13]. However, during experiments, the standard methodology is to provide a workload as input. The workload is read during the experiment and each job is individually submitted to the studied system at the appropriate time. This methodology has good properties suitable for experiments, it is reproducible, and it enables to compare different scenarios according to a shared input (the workload).

The two principal ways to obtain a workload suitable for experimenting are by either generating a workload thanks to a model [LF03; TEF05; Fei15b] or by extracting workloads from logs of production RJMS. The contributions of this work use methodologies and models from the state of the art. The website Parallel Workload Archive (PWA) [Fei19] regroups a collection of both generative workload models and extracted logs from diverse production clusters. Inquisitive readers are encouraged to read Emeras's work [Eme13] detailing the methodologies and implications of experiments using workload replay.

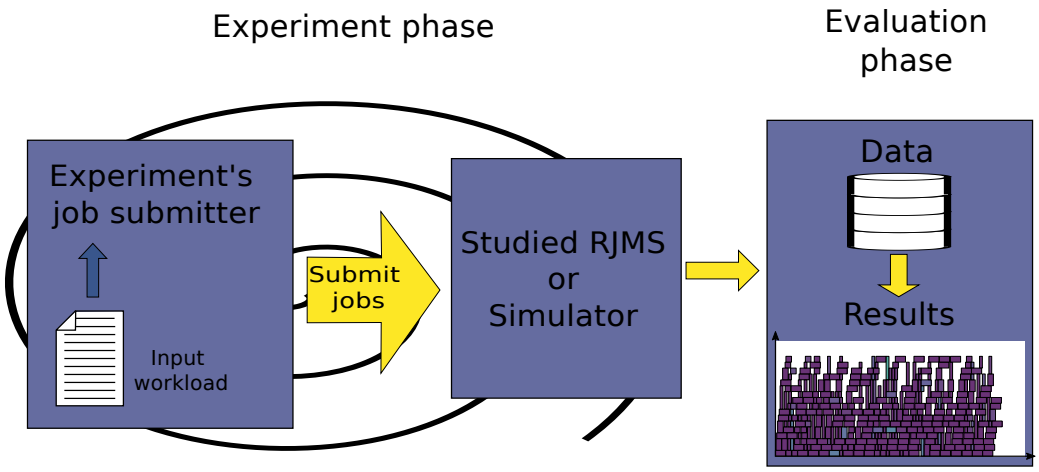


Figure 1.4.: Experiment methodology for RJMS using workload replay. The experiment's setup contains a module (or program) that submits jobs. The studied RJMS (or simulator) receives the jobs as if they were submitted by the users. Once all the jobs are submitted and scheduled, one can analyze the results in respect of the desired metric(s).

Figure 1.4 shows an example of how to experiment with RJMS. In the first place, one needs to have the studied target instantiated, such as an RJMS or a simulator (middle square of the figure). During the experiment, one module is in charge of reading a workload and submitting the jobs to RJMS when the time of the experiment reaches the job's submission date. Depending on the experiment's context, the time may be handled by another module (not depicted in the figure). In the case of the real time is used, the time in the workload must be relative to the beginning of the experiment.

When all the jobs are submitted to the RJMS and have been completed (or evicted), the experiment ends. Finally, depending on the RJMS (or the simulator), the data can be analyzed offline after the end of the experiment.

1.4 Contributions

In a first contribution, chapter 3, we proposes a new scheduling policy to schedule parallel jobs with redirection. The objective of the study is to propose an efficient scheduling algorithm for HPC jobs. Redirecting a job consists of stopping the execution of a running job to execute it into another cluster, or a dedicated part of it. The evaluation of the scheduling policy is conducted with an extensive simulation campaign to assess the performances of the new scheduling policy.

In the second contribution of this work, we are interested to extend scheduling simulations with models for the jobs. As explained in section 1.3.1, a standard methodology is to use a workload composed of jobs that have been submitted to a real platform. However, the workloads are often composed of high-level information about the jobs, such as the submission date, their processing time, and the *walltime*. More detailed information, such as the executed program or the amount of bandwidth used, is difficult to obtain and is not available in the workloads. This level of information available to experimenters has largely impacted the way we simulate scheduling. During the simulation, the job as simulated as a fixed amount of time that the scheduler has to wait before their completion [KSS19; Gal+20]. In production clusters, various effects happen on the computing platform that is not taking into account with this job model. The effects can be performance issues due to bad resource allocation for a job, or performance issues due to different jobs using the network and generating network interferences [Bha+13; Smi+18]. In chapter 4, we propose to extend scheduling simulations with models for the job executions. We present three different models and discuss their trade-offs in terms of performances and simulated effects.

One of the evaluated models is called the *ptask* model, this model has a reasonable trade-off in terms of precision of the simulation and execution time. In chapter 5 we evaluate the interference model of the *ptask* model against an HPC application. The evaluation compares the simulation of the *ptask* with the execution of an HPC application, in both simulation and reality we create synthetic network interferences to evaluate different scenarios.

In chapters 4 and 5 we aims to extend scheduling simulation with job models, however in some scenarios this approach is not always feasible. The major issue with simulation is that by using a model of the RJMS, one loses the details of the implementation of the software, which can lead to unadapted simulations. This is the case for instance, when one need to find the best configuration parameters for a

specific cluster. The third contribution, presented in chapter 6, tackles the problem of experimenting directly with real RJMSs. We propose two new approaches to study distributed systems in controlled environments without requiring a model. Chapter 7 details the technical tools supporting the two aforementioned approaches.

The reproducibility of science is a crucial factor to build reliable knowledge. One factor playing against the reproducibility of computer science work is the software collapse[Hin19]. Software collapse refers to software that stops working if it is not actively maintained or developed, scientific software is also subject to this phenomenon. In chapter 8, we show that Functional Package Managers can be leveraged to achieve the reproducibility of the software stack (not the hardware) on which a program depends. With this methodology we ensure the reproducibility of the execution and build an environment of our scientific software can be preserved. Reproducing the production environment allows us to share and execute scientific programs, and with the build-environment to modify a scientific program (to fix bugs or add new features). This is what we called the variation. The scientific experiments and programs created for this dissertation follow the principle of variation.

2 Experimental Study of RJMS: Methods and State of the Art

2.1 Introduction

HPC platforms are subject to numerous evolutions from diverse sources, the platforms become heterogeneous in resources, but also in the diversity of scientific applications[1] using it. Hence, RJMSs need to evolve to integrate this heterogeneity to efficiently manage the next generations of computing platforms.

The Resource and Job Management and Scheduling Systems (RJMS) are complex software designed to manage HPC platforms where the main objective is to achieve great performance and scalability. This particular context has two implications. First, the RJMS of the platform must provide fast and adapted decisions to match the user's demand in computing resources. Second, it is not possible to modify the RJMS on a production platform without risking to hinder the system's performance or worst generating downtimes of the entire system.

Therefore, it is necessary to have tools and methodologies to support the development of RJMS through experiments and to provide efficient RJMSs matching the evolution of HPC systems.

		Application	
		Real	***Model***
Platform	***Real***	in-vivo *(Testbeds or real platform)*	*benchmark*
	Model	in-vitro *Emulation*	in-silico *Simulation*

Table 2.1. Experiment classes depending on the application and the platform (real or model) [GJQ09].

Traditionally, the study of computer science applications falls into three classes of experiments (the terms are derived from biology): ***In-vivo*** is the study of a real application into its targeted final environment. ***In-vitro*** (or *emulation*) is the study of a real applications into a simulated environment. ***In-silico*** (or *simulation*) consists

[1] This section focuses on computer science *applications* in the broad sens — An RJMS is an application. To remove confusion, the scientific applications used by the end-users of an HPC center are referred to as *jobs* or *job's execution*.

to use a model of the targeted application in a simulated environment. In our specific context, the studied applications are RJMSs.

Table 2.1 illustrates these methodologies adapted to computer science systems [**GUSTEDT_2009**]. The *benchmarking* category is added to the experiment classes and corresponds to using a model of an application on a real platform, in order to evaluate hardware's performances or detect performance issues. However, to the best of our knowledge *benchmarking* is not used in our context due to the complexity of the RJMSs.

The RJMS occupies a central place in the HPC system and has interdependencies between the users, the platform, and the jobs. For instance, the user's behavior is directly impacted by the system's reactivity [ZF14]. In the context of RJMS, experimenters have to compose with all the complexity of the system and need to take into account the users, the physical infrastructure, the jobs and their executions, and the RJMS. Properly assemble all these elements to create, trustworthy, reproducible, controllable, and scalable experiments is challenging.

Experimenters have found diverse ways to conduct experiments with RJMS, from taking advantage of technical tools such as virtual machine to leveraging models from the grid and HPC community. All methodologies broaden the landscape of possible experimental setups to study RJMS. In this work, we propose the notion of **hybridization** lying between real and model, for the platform and the RJMS. This notion comes from the observation that with the methodologies used to study RJMS in the state of the art, the separation between reality and model isn't well defined.

- In [Dut+16], the authors propose a new RJMS simulator: Batsim. Batsim provides an RJMS model, but the scheduling is made by an external program using a generic API. With this design, the authors were able to isolate the scheduler of a real-world RJMS (OAR) and to create an interface between Batsim and the OAR's scheduler. This methodology mixes part of a real RJMS (OAR) with an RJMS model (Batsim), therefore it is nor completely simulation, nor completely emulation.

- In the Slurm simulator [JDC18] the authors altered the Slurm RJMS to remove limiting or irrelevant parts of the RJMS in order to extend its simulation capabilities. More specifically, the modifications enable to accelerate the simulation and increase the reproducibility of the results. In this case, the authors modified a real word RJMS, hence the RJMS is not totally emulated. Even if a large part of the RJMS's source code is used, the modifications made imply that the removed or modified parts are modeled.

This chapter presents the different methodologies from the state of the art to support the idea of hybridization. Figure 2.1 illustrates the *hybridization* and places the different approaches detailed in this chapter.

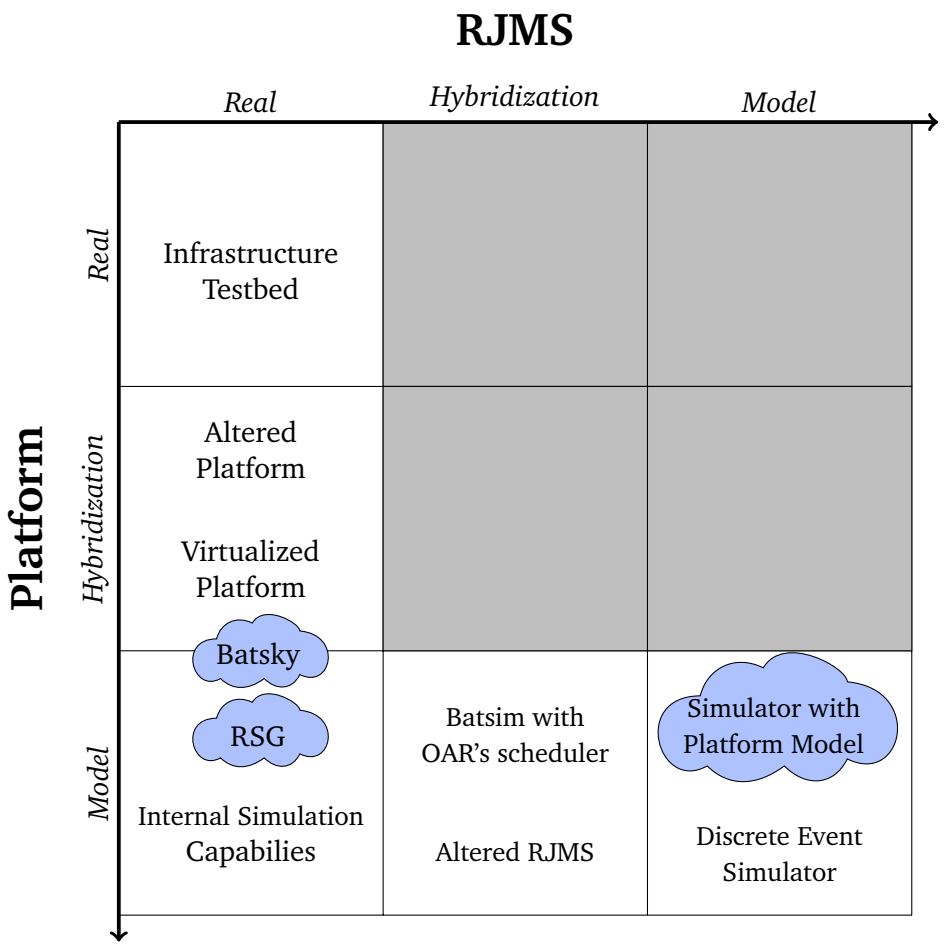


Figure 2.1.: *Hybridization* mixes approaches based on reality and model. The blue clouds locate the contributions of this work according to the *hybridization*. The approaches using a model of RJMS are detailed in section 2.2. The approaches based on real or *hybridized* (left column) RJMS are detailed in section 2.3. Simulators with platform model is the focus of chapters 4 and 5, RSG and Batsky approaches are detailed in 6. Chapter 3 evaluates a new scheduling policy with a *Discrete Event Simulator*.

The remaining of the chapter is organized into two parts corresponding to two main approaches that have been used to conduct RJMS experiments.

- The first approach focuses on testing new ideas with an RJMS model, via *simulation*.
- The second approach focuses on using a real-world RJMS, either to evaluate its performances or as a starting point to evaluate new ideas. This approach is related to *emulation* and *in-vivo*.

2.2 Using RJMS Models

The first approach to study RJMS is to create a model of an RJMS and to run simulations. This approach enables experimenters to run experiment campaigns without requiring a real RJMS or a platform.

Creating an efficient scheduling algorithm for HPC clusters is an active field of research. Evaluating new scheduling policies is a real challenge, and as stated in 2.3.1 using a production cluster is not recommended.

The methodology presented in this section is to develop a simulator for RJMS simulations. Simulation is a way to quickly design, prototype, and evaluate new ideas for HPC clusters, such as new scheduling algorithms. A simulator has good properties, it is fast, reproducible, and increases the number of available scenarios.

One common technique is to use Discrete Event Simulation, which models the workload and the job's execution as a sequence of discrete events. This technique has been used for the last decade to evaluate scheduling algorithm for HPC. However, a new simulator was developed almost at each new idea, and the adoption of tools from one experimenter to another was pretty low. As a consequence, many simulators were not openly released or left unmaintained once the paper has been published, or the original programmer stopped working on it. The author of [Poq17] describes this phenomenon as *Publish and Perish*.

Finally, more stable simulators have started to emerge to propose a common model and methodology for HPC RJMS simulation. The most notable projects are Alea [KSS19], AccaSim [Gal+20], Batsim [Dut+16], and ScSF described in the (more details about ScSF in the next section 2.3.2). Table 2.2 shows the differences between the different simulators, according to four different properties.

From the presented simulator, has the particularity to ScSF rely on a real RJMS. Its design and implementation have been already covered in section 2.3.2. Evaluating new scheduling policies with ScSF is tedious as one needs to integrate it into the Slurm code. The execution of the framework is complicated, as it involved several technical aspects such as creating different virtual machines, with remote access (*ssh*). Therefore, ScSF is not suitable for large experiment campaigns and requires good technical knowledge hindering the ease to use and to reproduce. However, it is an interesting tool for system administrators that wish to find suitable a Slurm configuration for their computing infrastructures.

More recently, the RJMS Flux provides a simulator to test the scheduler without a particular setup. However, currently no documentation is available to run the

	Scheduling	Workload	Platform Model	Job Model
Alea	Internal (Customizable)	• SWF format • Adaptive submissions	GridSim: • Failures model • Platform description	Static time period
AccaSim	Internal (Customizable)	• SWF format • Adaptive submissions	*Ad hoc* : • Extendable with external data	Static time period
Batsim	Generic API	• Custom format • Conversion scripts from and to SWF • Dynamic submission	SimGrid: • Platform description • Network model • Host speed model	• Inter/Intra job interference • Or, Static time period
Flux Simulator	Flux scheduler	• Custom format	*Ad hoc*	Static time period
ScSF	Slurm	• SWF	*Ad hoc*	Syscall *sleep*

Table 2.2. Comparison of five open source simulators for the simulation of RJMS.

scheduler or to change the scheduling policy. The current state of the simulator is awaiting a merge request on a collaborative version control website [2].

The three last simulators (Alea, AccaSim, and Batsim) share the idea to build an efficient, reliable, and easy to use, scheduling simulator for HPC platforms. Using such simulation tools is efficient to quickly prototype and evaluate scheduling policies. Simulators have common properties:

- **Scheduler** (or scheduling algorithms): Determines how the scheduling algorithm is simulated. Most simulators embed the scheduling process in the simulation core, this is the case for AccaSim, Alea, and ScSF. Implementing a new algorithm requires to develop inside the simulator, by extending predefined *scheduler classes*. Batsim's authors made the choice to completely separate the scheduling from the core of the simulator, and instead expose a generic API to separately create new scheduling algorithms. The Flux simulator uses a scheduler compatible with the Flux RJMS. Despite its early stage of development Flux is a complex software, the lack of documentation and tutorials about the simulator lower its usability. Creating a new scheduler in Flux is a complex task.

- **Workload**: As explained in section 1.3.1, the workload is a crucial input for every RJMS experiments. Thus, each simulator has its way to inject a workload during the simulation, one common way is to describe the workload into a static file and provide it as an input of the simulation. The most used file format to describe HPC workloads is Standard Workload Format (SWF) [Cha+99]. Both

[2] https://github.com/flux-framework/flux-core/pull/2561

AccaSim and Alea feature a way to directly read SWF formatted workloads. Furthermore, AccaSim and Alea feature dynamic workloads [ZF14]. Dynamic workloads are based on the observation that static workloads, composed with fixed job arrival dates, do not adapt their behavior depending on the scheduler performances which is the case in production. Dynamic workload adapts the arrival dates of the jobs depending on the scheduler performances. Batsim has its workload model and features different scripts to convert to and from SWF format. Besides, during a Batsim's simulation, the scheduler can submit new jobs not initially present in the input workload. On its side, the Flux simulator has its dedicated workload format, close to SWF, but with information relative to the Flux RJMS.

- **Platform model** is the modelization of the platform for the simulation. Two different categories exist: Simulators with an integrated platform model, this is the case for Accasim and the Flux simulator; and the simulators based on a simulation toolkit, this is the case for Batsim and Alea. Alea is based on GridSim [Sul+08], and Batsim is based on SimGrid [Cas+14]. The description of the simulated computing platform is made using the simulation toolkit (either SimGrid or GridSim). Alea leverages GridSim to simulate failures during the experiment. Although the Accasim's platform model is not based on a simulation toolkit, its platform model is extendable to simulate different scenarios, such as nodes failures or energy consumption.

- **Job model** defines how the jobs are simulated. Flux Simulator, Alea, and AccaSim consider jobs as discrete events, once a job starts its execution the simulator knows the job's ending time — the simulator may hide this information to scheduler because HPC schedulers are online (1.2.1). Once the simulation time has reached the job's completion time, the job naturally ends. This model does not simulate the variability of the job's performances depending on its execution context (the platform capacity and the other jobs running at the same time). Batsim, on the other hand, provides different models for the execution of the jobs. The models are based on SimGrid simulation toolkit. These models enable the jobs to use the computing resources of the simulated platform (such as the CPUs and the network). The execution time of the jobs does not depend on a static time statically provided, but instead is computed during the simulation, and depends on the platform's capacity (node speed, network latency, network bandwidth) and the other jobs executing at the same time.

This thesis focuses on the simulator Batsim, more specifically chapters 4 and 5 target the validation of the aforementioned job model featured with Batsim.

2.3 Experiment with Real RJMSs

The second approach to study RJMS is to directly use real-world RJMS, either to evaluate it or to add new ideas. This approach enables experimenters to be close to a real system and to increase the confidence we have in experiments. Additionally, in some cases, it is necessary to directly use real RJMS. For scalability evaluation or to test new features during development for instance.

In-vivo and *emulation* are good candidates to deal with real RJMSs. Two approaches exist to conduct experiments with a real RJMS. The first possibility is to have a distributed platform to execute the RJMS code (section 2.3.1). One difficulty is to embrace the fact that RJMSs are distributed software and that to execute an RJMS, one needs to have a platform with at least a few functional nodes. A node is functional if it features all the prerequisites to execute the source code of the RJMS, such as an operating system with network access. The second possibility is to work directly with the code of the RJMS to extends its simulation capabilities (section 2.3.2).

The first possibility enables to use a real RJMS without modification at the cost of increasing the experiment complexity because a functional platform is required. The second possibility enables to reduce the complexity of the experiment but at the cost of altering the RJMS to the point that it is not representative of the reality.

2.3.1 Real RJMS with a Functional Platform

Real Platform

Using a production platform is challenging because it is often not possible to modify the RJMS of a production platform. Even in the case of a platform would be available for experimenters the reproducibility of such an experiment is close to zero due to the user submissions.

An alternative is to use a dedicated testbed such as Grid'5000 [Bal+13]. Grid'5000 has several clusters with different configurations. Grid'5000 enables experimenters to access to a computing platform with root access on the nodes. It is therefore possible to install a different RJMS on a platform, or a subset of nodes of the platform to conduct experiments. Additionally, one can install a custom system's image on the nodes, increasing the tuning capabilities and the reproducibility.

Platform testbeds are direct candidates to experiment with RJMS as it is possible to completely customize the software on the platform. This is the approach used in [Mer+17], Mercier *et al.* deployed on a Grid'5000's cluster two different real-world RJMSs on the same platform to measure the feasibility of their approach. One

RJMS comes from the HPC community (OAR), while the second RJMS comes from the big data community (YARN).

Grid'5000 is also used in [**GUSTEDT_2009**] to compare their RJMS simulator with reality. In both aforementioned cases, the authors use a workload composed of application benchmarks ([Bai+91]). The benchmarks are executed on the experiment platform. In the case of [Mer+17] also used benchmarks for big data applications [Wan+14].

In [Ahn+20], they compare the throughput of their RJMS, Flux, to the throughput of the Slurm RJMS. They use a cluster composed of 32 nodes and were able to install both RJMSs to do a performance evaluation. This work focuses on RJMS throughput, therefore each job immediately exists directly after it starts executing.

Using a real platform has good properties because it increases the practical feasibility of new ideas and gives results close to a production system. However, the scope of available studies is limited to platforms at hand. Any study involving an unavailable platform is not directly feasible.

The high variability of real systems and the complexity of the installation hinders the reproducibility of the experiments. Indeed it requires both the platform used to support the experiment and knowledge in system administration and infrastructure management.

Finally, experiments using a real platform are time-consuming because it is in real-time. It is problematic to study RJMS behavior for long periods. To give an example, with the usage policy of Grid'5000 the longest possible jobs are during the weekend (from Friday 5 pm to Monday 9 am).

Hybridization: Mixing Reality and Model

Virtualization and containers enable to execute an operating system on virtual hardware isolated from the actual computer's operating system, they enable to execute several operating systems on top of single computer hardware. These techniques fall into the *hybridization* category as they mix reality — the operating system — and model — the virtualized hardware.

Virtualization or Containers. One solution is to leverage virtualization to create a virtual platform on top of a physical platform. With this technique, one can simulate a different platform than the physical platform used.

Despite the fact that the virtual nodes are less performant due to resource sharing, an operating system can be installed on the virtual nodes. A virtual node shares all the functionalities of a normal one, and are able to act as the computing nodes of a

computing cluster. However, the virtual nodes are necessarily less performant are they share the same hardware. One can install a real-world RJMS on the virtual platform.

Virtualization is useful to increase the number of available nodes and to create a larger platform with few physical resources. However, the virtual nodes are significantly slower than the original nodes, and it is often not possible to execute the real jobs during the experiment.

This methodology has good control over the environment and increases the reproducibility of the experiment; nevertheless reproducing experiments involving thousands of virtual machines requires a setup that can be tedious to reproduce. Using these techniques is also useful for developers because it enables to create a (very) small platform on a single laptop for testing purpose. However, increasing the number of virtual nodes per physical node decreases the performances of each virtual node, and therefore the jobs of the simulation are slow and become unrepresentative of a job's executions. A common technique is to use, instead of a real job, a system-call sleep causing the virtual node to be idle, during the job's execution. In [Geo+15] they developed their scheduling policy directly in Slurm. For the evaluation, they managed to instantiate a platform with 5040 virtual nodes and install Slurm with only 20 physical nodes. They replace all jobs with a *sleep* command.

Altered Platforms. Virtualization (or containerization) is useful to test system scalability and instantiates larger platforms that the available platform. Another approach is platform alteration, it aims to model a different platform on top of a real platform, and increase the number of available scenarios.

In [Sar+13], they propose Distem, a tool for experimenters to model a platform from a set real platform (using Grid'5000 for instance). Leveraging virtualization and containerization they create a modeled platform with reduced capacity. The main difference, with the previous approach (containers and virtualization), is that Distem aims at proposing a realistic platform, but with modified characteristics. Instead of creating a full-scale platform, Distem provides a way to explore different controlled scenarios during experiments, such as changing the network topology of the original platform or injecting nodes failures.

Platform Simulator

The last possibility is to use a platform simulator or a platform simulation toolkit. Several platform simulator have been proposed, and feature platform models that can run on a single node, or a personal laptop, GridSim [Sul+08], SimGrid [Cas+14].

One difficulty is to accurately simulate the network of the computing platform. In [Leg15], the author describes three different approaches to simulate the network at scale. Packet-level simulators, delay-based network model, and flow-level of TCP. While the packet-level simulators are the more realistic, as they simulate each network packet navigating through the network layer, they fail to scale to a large number of nodes because the quantity of packet to simulate drastically increases with huge HPC cluster. The alternative, cost-efficient delay-based network model has good scalability properties, but the proposed models do not include network contention. Finally, the flow-level network simulation proposes good simulation capabilities to simulate network at scale, as it is both scalable and feature a contention model.

Although, even if several platform simulators have been proposed, such as SimGrid or GridSim, using a real RJMS with this kind of simulator is challenging. This kind of simulator is not directly usable with RJMS as it is not possible to install an operating system capable to execute real code on the simulated nodes.

Chapters 6 and 7 of this dissertation focuses on extending the emulation capabilities of the SimGrid simulation framework, in order to use an RJMS on the simulated platform.

2.3.2 RJMS Hybridization

Another possibility to work with a real RJMS is to *modify* its source code to the advantage of the experimenter. For instance, one can remove unnecessary (for the experiment) synchronization threads to increase the execution speed.

The first way is to use the integrated simulation capabilities when they exist, Flux RJMS for instance directly embeds a simulation mode. The second way is to make modifications to the RJMS code to extend its simulation capabilities. Finally, the last solution is to isolate a part of the RJMS to study it in simulation.

Integrated Simulation Capabilities

Slurm and Flux both offer simulation capabilities.

In [Pol+18], they use the flux simulator embedded into the flux source code to evaluate their scheduling policy. However, to the best of our knowledge, the simulation models have not been published yet, so it is difficult to have a precise idea of its simulation models.

In [GH12], Georgiou *et al.*use the Slurm's feature *multiple SlurmD* to evaluate the scalability of the Slurm RJMS by simulating a cluster up to $16,384$ computing nodes on few hundreds of real computing nodes. Slurm architecture is based on a single controller (SlurmCtld) and several Slurm daemons (SlurmD). A Slurm installation

has one controller for the entire cluster, and one SlurmD running on each node. All the computing-intensive operations, such as the scheduling, are executed on the controller. SlurmD daemons are used to control job execution and monitor the computing nodes.

Multiple SlurmD is a Slurm's feature that enables to install multiple SlurmDs on a single computing node. The different nodes are virtualized using processes, from the SlrmCtld point of view one SlurmD equals one virtualized computing node. It is worth mentioning that similarly than from using virtualization, the computing nodes are not able to execute HPC jobs, and thus the jobs are reduced to a call to the system call *sleep*. Additionally, the execution speed of the experiment is based on the real-time, leading to long experiment time. In their publication, the authors measure Slurm's efficiency to deal with both a large number of resources and large numbers of jobs. Therefore, one instance of their experiment lasts around ten minutes, as the workload is representative of jobs burst.

Altered RJMS

The Slurm Simulator. The Slurm simulator has been originally proposed in [Luc11]. The principal idea has been used in [Rod+17] from BSC lab, which has benefited from an upgrade in [JDC18]. The simulator features a modified Slurm's version (17.11 in the latest upgrade), the changes that were made to the Slurm's source code are:

- Original implementation [Luc11].
  - They get rid of all unnecessary threads during the simulation to increase the simulator scalability.
  - They replace SlurmCtld main loop for a simulation loop, and to simulate the SlurmD daemons, which is are no longer needed. Only the SlurmCtld and the database daemon remain active during the simulation, no need to use SlurmD daemons.
  - The simulation is managed by a new component: *sim_mng*. Using *LD_PRELOAD* (detailed in chapter 7), they managed to inject the simulation time to increase the simulation speed.
- In [Rod+17], the authors modified the SlurmD daemon to incorporate it into the simulation. One SlurmD daemon is sufficient for the whole simulation. Additionally, they increase the simulation speed and fix various bugs. The authors also integrate the simulator with various tools to help the workload input generations and the analysis of the results.

- Finally, in [JDC18], they managed to upgrade the Slurm version of the simulator, they also made the simulator deterministic improving at the same time the reproducibility of the results. Additionally, they propose the first comparison to the Slurm Simulator compared to a real-life experiment.

The Slurm simulator is a good example of altered RJMS, as the authors of the simulator managed to increase the simulation capabilities of the Slurm RJMS with few modifications. They remove the unnecessary parts of the RJMS, either to increase the scalability of the simulator or to speed up the simulation. The approach is interesting as it enables to use a functional (yet modified Slurm) for the experiments. In [Álv+17], they implement a workflow aware scheduling policy into Slurm and they use the Slurm simulator to evaluate it.

This approach is useful to experiment with the Slurm RJMS, however, it is limited to a unique Slurm's version. The simulator needs to be adapted and re-evaluated for each new Slurm's release. Thus, this approach is not well suited to support experiments for the evolutions of new Slurm features.

Additionally, the platform of the experiment has very low simulation capabilities. The simulated jobs are reduced to a call to the system function *sleep*. Therefore, it is difficult to use this approach to experiment on a heterogeneous platform, or to evaluate topology-aware job placement algorithms [Geo+17; Cru+19] for instance.

Taking Advantage of Modularity

In [Dut+16], the authors managed to isolate the job scheduling part of the production-ready scheduler OAR [Cap+05]. They plugged the scheduler on a simulator: Batsim. Hence, all the scheduling decisions are left to the OAR scheduler, and Batsim manages the simulated computing platform and the simulation of the workload and the jobs.

This approach is convenient to evaluate the scheduling policies implemented in OAR, and in the case of the original work to validate the simulator Batsim. However, extending this methodology to other RJMS is not directly extendable to other RJMSs, because it is not always possible to isolate the job scheduler of an RJMS.

2.4 Choosing the Adapted Methodology

All presented methodologies aim to experiment with RJMS and are separated into two main approaches. One can either use and experiment with a real-world RJMS or to build a model of the RJMS. This section aims to clarify the conditions that lead experimenters to use one approach over another.

One thing to consider is the difficulty to implement a new idea or to test and validate new hypothesizes. Using real RJMS is tedious because it requires having a good understanding of every aspect of the RJMS to be able to add new features or new algorithms. Whereas, the simulators presented in section 2.2 aims at reducing the cost to prototype and evaluate new ideas. For example, with Batsim's generic API to create schedulers, a scientist developing a new algorithm doesn't need to understand Batsim's internal functioning but only the exposed API.

From a general perspective, using an RJMS model offers more stability in the results, and are more reproducible. Furthermore, simulations are fast and can simulate months of HPC activity in less than one hour, increasing the number of parameters that can be explored during a single experiment campaign. The experimental setup needed to use real RJMS is often more complex and requires technical manipulations. Furthermore, the experiments are costly in time and one experiment campaign can last several days, experimenters have to drastically limit the number of explored parameters.

Finally, and most importantly, every presented methodology has a different way to control reality, and thus the events that should be happening during an experiment. The experiment class *in-vivo*, for instance, has the advantage to incorporate all the complexity of the platform, from predictable behaviors such as network contention to unpredictable events such as unwanted node failures. This complexity needs to be carefully taken into account by the experimenters, at the risk to miss the target of the study. On the opposite side, *simulation* heavily relies on models. Models are by essence an abstraction of reality. Using too simple models may lead to simulations that are too far from reality, and therefore are not suitable to construct reliable knowledge. When using a model, one needs to consider two points. First, what part of the reality is captured, and if it does apply to the experiment objectives. Secondly, the accuracy and applicability of the model must be cautiously validated to increase the confidence in the results. In case these two points are not carefully verified or are not possible, using an approach based on reality can be a good alternative.

2.5 Conclusion

This chapter depicts the methodologies used for experimenting with RJMS. The original statement based on table 2.1 present four experiment classes, *in-vivo*, *in-vitro*, *benchmark* and *simulation*. This statement assumes that experimenters have the possibility to use either a model of the studied application (in our case the studied application is RJMS) or to use a real application. And, that the studied application needs an environment to carry the experiment that is also either a model or real.

Based on the observations of the methodologies used in the literature we introduced the **hybridization** that mixes real components and modeled ones. Additionally, the axis between reality and model is not discrete, instead, it gradually goes from one reality to model. Figure 2.1 illustrates this idea and places the methodologies described in this section in the two axes. This figure enables to locate the different works done in this thesis.

Next, chapter 3 presents the evaluation of a new scheduling policy for parallel jobs. The scheduling policy has been evaluated using the Discrete Event Simulator approach, with the Batsim simulator.

In chapters 4 and 5, we focus on extending current simulation capabilities, to be able to simulate the jobs, and their induced resources activities, to increase the confidence we have in our simulations and broaden the scope of different scenarios one can experiment with. This is the cloud located on the bottom-left side of the figure, entitled *Simulator with platform model*.

In chapter 6, we present a new approach to experiment with real RJMS on a platform simulator, without prior modification on the RJMS's source code or extensive use of virtual machines.

3 On-line Scheduling with Redirection for Independent Jobs

3.1 Introduction

The need for efficient automatic tools for managing the resources in large scale modern parallel and distributed platforms become more important as their complexity increases [Don+11; Ahn+20]. On the first hand, we need simple enough mechanisms able to deliver an allocation of jobs to the processors at scale, but on the other hand, such mechanisms should include all features needed to deal with specific situations (like delay in delivering data associated with a job, big differences in job size or the consequences of disturbances). The situation of actual resource management systems is paradoxical in the sense that the processor allocation policies remain very simple with a lot of additional specialized or generalized plugins which make the whole system considerably hard to maintain [Gle16; Geo10].

The problem considered in this chapter is to determine an allocation of the jobs submitted to the platform to the available distributed resources (1.2.1). This problem has been considered from two different perspectives. First, from the viewpoint of the middleware community, many solutions have been provided which consist of rather simple heuristics developed in simulations or actual systems. There is not always an explicit objective to optimize and the main challenge is to design robust strategies that are implemented in existing management systems like SLURM or Torque. Second, from the viewpoint of combinatorial optimization, many idealized problems have been solved for adequate and simplified cases. Most of the existing studies in this context consider restricted hypotheses (like sequential jobs, precisely known processing times, no congestion in the interconnection network, etc.). The challenge is to reconciliate both viewpoints and design algorithms with good performances on realistic models of the platforms. This can be achieved by theoretical analysis involving approximation or competitive algorithms and to assess the proposed methods on well-targeted experiments.

We consider the problem of scheduling parallel jobs without preemption in multiprocessor clusters. More specifically, we consider the concept of job redirection, where a job can be killed and restarted into another cluster — or another dedicated set of processors. The idea of redirection comes from *resource augmentation* [KP00], a technique for analyzing the competitivity ratio of on-line algorithms under the assumption that we compare the on-line algorithm to a weaker version of the

corresponding off-line optimal. Scheduling with redirection has been previously studied [LMT17] for sequential independent jobs where promising results have been presented. As far as we know, this idea has never been presented in HPC.

However, actual management systems are built mainly to cope with parallel jobs, which differs from existing redirection algorithms that consider sequential jobs. In this work, we introduce a new algorithm to schedule parallel and independent jobs with redirection. It is a step further to integrate the redirection into production management systems. The redirection operates by detecting *heavy* jobs and redirecting them into a dedicated pool of processors, where no redirection is further possible. Detecting such heavy jobs is done by keeping track of the set of jobs submitted during the execution of a job. When a fixed threshold of arrivals is reached for a particular running job, the job is characterized as heavy and it is redirected, leaving room for queued jobs.

In this work we propose an algorithm to schedule parallel independent jobs with redirection. We validate our approach through an extensive simulation campaign based on the analysis of logs extracted from three production management system. We compare this algorithm to a well-known and widely used scheduling policy, FCFS with EASY backfilling, and we show that scheduling parallel jobs with redirection improves the average bounded slowdown objective. The slowdown is a popular metric that targets the time a job stays in the system from its release time to its completion (normalized by its size). It reflects the user satisfaction [Fei01b].

The work of this chapter has led to one publication [Fau+20]. It has been made in collaboration with Geogio LUCARELLI, Olivier RICHARD and Denis TRYSTRAM.

3.2 Definition and Notation

As defined in in 1.2.1, we consider in this work the problem of scheduling a set of jobs into an HPC platform. That is to say, finding an allocation of each job to a set of free — or available — resources as well as assigning it a starting time. Note that preempting the execution of a job is not allowed. Moreover, the parallel execution of two or more jobs on the same resource is not permitted, and hence the assigned resources to a job should be available during the whole interval of its execution.

To evaluate our scheduling algorithm, we use the bounded stretch also known as the bounded slowdown (BSLD) metric [Fei01a], which is defined for a job j as follows:

$$BSLD_j = \max\left(\frac{F_j}{\max\left(p_j, \tau\right)}, 1\right) \tag{3.1}$$

where τ is a constant that prevents small jobs to have a high impact on the overall performances. In this work, we set τ to 60 seconds. Accordingly, we define the average and max slowdown as follows:

$$\begin{aligned} BSLD_{avg} &= \frac{1}{n} \sum_{j \in [n]} BSLD_j \ , \\ BSLD_{max} &= \max_{j \in [n]} (BSLD_j) \ . \end{aligned} \tag{3.2}$$

3.3 Related Work

There exists a huge literature dealing with job scheduling and resource allocation. We present below the most relevant studies related to this problem. Resource allocation and job scheduling is a basic problem, which was studied at several levels. In this work, we are mainly interested in the problem of scheduling at a wide level such as HPC computing and cloud computing where the scheduler has to handle a set of resources and responds to many requests submitted by the users of the system. This problem is well known to be NP-complete for decades, even in several restricted cases [Ull75]. The underlying complexity of the scheduling problem makes it well studied and often simple solutions are in use in practical environments. Most existing production schedulers use the FCFS policy in conjunction with a technique called back-filling to increase the utilization of the cluster [MF01]. In the past years, the scheduling problem has become even more complicated with the increasing complexity of the computation platforms. Job scheduling for this kind of platform remains challenging. Scheduling with preemption has been shown useful, especially in environments such as big data and cloud where jobs can be more flexible. In cloud system, jobs can be migrated to another place thanks to machine virtualization. In big data frameworks, the jobs are fault-tolerant and the scheduler can dynamically adapt their number of resources[Mer19].

In this work, we are interested in scheduling parallel jobs, with the ability to redirect a running job, but without allowing the preemption of its execution. That is to say, a job can be killed and restarted (from the beginning) later. Scheduling with redirection with preemption has been studied, to increase the resource usage[Ana+12], in the cloud and big data environments where the jobs are more flexible. Studies show that preemption can be leveraged to optimize the waiting time[BSS13], or to free resources needed for a high priority job[Cho+13].

Our contribution differs in the way that we propose redirection for HPC jobs, where the constraints imposed on the system do not allow preemption. However, we allow the system to kill jobs to redirect them, thus approaching the idea of preemption.

3.4 Scheduling Parallel Jobs in HPC Environments

Given a set of jobs, a scheduling algorithm defines for every jobs a starting date, and an allocation. In HPC environments, each job is known by the scheduler only at its arrival time, and hence the scheduler has to take on-line decisions. At its arrival time, a job may have to wait in the queue if there are not enough available resources on the system. Depending on a priority function, the scheduling algorithm determines the order to run the jobs in the waiting queue; this order defines the primary queue policy. In this work we focus on the First Come First Served policy (FCFS) according to which the jobs are sorted in non-decreasing order of their release time: a job j precedes a job j' if $r_j < r_{j'}$. In case of a tie ($r_j = r_{j'}$), an arbitrary order is chosen.

When the first job in the queue is delayed due to the unavailability of a sufficient number of resources, processors can remain idle while some jobs are waiting to be executed. To increase resource utilization of the platform, the actual job and resource management systems use the mechanism of *back-filling* in conjunction with the primary queue policy. Back-filling takes benefit from the idle waiting resources and the *walltime*, by allowing the execution of one or several smaller jobs during the period when bigger jobs are waiting for a sufficient number of resources. Several back-filling mechanisms have been proposed, such as conservative back-filling and EASY back-filling [FW98]. In this work, we consider the EASY back-filling mechanism introduced by Feitelson et al. [MF01], which is one of the most widely used among HPC resources managers, such as SLURM [Gau+18]. It owes its popularity to its ability to achieve high resource allocation thanks to a simple and scalable scheduling policy.

Back-filling a job means that a job with a lower priority is allowed to overtake a job with a higher priority. The condition under which a job can be back-filled is that it should not delay the provisional execution of its preceding jobs that are not started yet. Several variants of back-filling exist, like conservative back-filling and EASY back-filling. The conservative back-filling mechanism satisfies this condition for all jobs in the primary queue. EASY back-filling satisfies this condition only for the first job in the primary queue. In practice, EASY back-filling is categorized as aggressive because it causes most of the small jobs to be executed before the big jobs.

In this work, we focus on the EASY back-filling mechanism, because of its small complexity which makes it suitable for HPC clusters that have a high constraint on response times.

3.5 Scheduling Parallel Jobs with Redirection

3.5.1 General Description of the Redirection

The redirection is a generic mechanism, it can be used in conjunction with any other HPC scheduling algorithm. This is mainly because the redirection does not directly impact the scheduling decisions, instead it can independently choose to redirect a job to improve later scheduling decisions. The redirection mechanism identifies jobs that are worth to be redirected and move them into a dedicated part of the cluster — or another independent cluster. To set up the redirection, we propose to split the resources of the platform into two independent groups as depicted in Fig. 3.1: the *principal group* and the *redirection group*. The size of the redirection group is determined by a parameter α, the *percentage of allocation*. Depending on the total number of available resources and on the properties of the jobs targeting the HPC platform, the size of the two groups needs to be adapted. The *principal group* contains $(1-\alpha)m$ processors while the redirection group contains the remaining αm processors of the platform, where m is the total number of processors. Both groups can be scheduled independently without interfering.

Identifying jobs that harm the overall cluster's performance is difficult, as it is not easy to evaluate. The redirection mechanism identifies such jobs by counting the number of submitted jobs during the execution of another job. This method enables the estimation of the induced pressure by a job. If a job has a counter higher than a fixed threshold θ, the system can choose to trigger a redirection. That is to say, each job running into the *principal group* holds the number of jobs submitted during its execution. When a job has been chosen to be redirected, it is killed by the system and moved to the redirection group which is exclusively dedicated to these jobs in order to not further delay the execution of redirected jobs. A job cannot be redirected multiple times. Besides, as stated before, we focus on the case where no preemption is allowed, as a result, a redirected job is terminated and restarted from the beginning into the *redirection group*.

3.5.2 Dealing with Parallel Jobs

The idea of redirection comes from [LST16], where the authors introduced the rejection as a resource augmentation technique: the scheduler is allowed to reject a fraction of the jobs submitted to the system. Redirection is a practical adaptation of the redirection where the system is not allowed to reject jobs. The idea of the redirection has been studied for sequential and independent jobs in [LMT17]. It also introduces the resource partitioning which is discussed in 3.8. However, recent management systems are built to cope with parallel jobs, which is not possible with

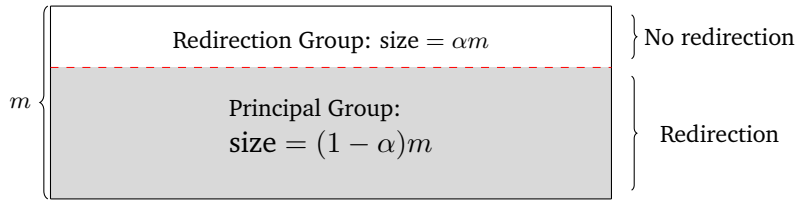


Figure 3.1.: Division of the nodes of a cluster of m processors in respect of the input parameter α, with $0 \leqslant \alpha \leqslant 1$. Both group are managed by an independent scheduler.

the existing redirection algorithms. Extending the redirection to parallel jobs is necessary to use it in a production management systems, but it is not an easy task.

The idea of counting jobs has been introduced by Lucarelli *et al.* [Luc+16]. One challenge induced by extending the redirection to parallel jobs comes from the resource allocation. The sequential algorithm allocates a processor to a job j at the time r_j. Reproducing this allocation is not feasible for the parallel algorithm — or too complicated. The direct cause of this difference is that the early allocation gives an estimation on a load for each processor. More specifically it is possible to estimate the impact of the execution of a specific job onto another set of jobs, *i.e.* the jobs waiting for the same processor. In the case of parallel jobs, estimating the impact of a specific job execution is harder since allocating a job to a set of processors is often done at the last time.

This cannot be applied in the case of parallel jobs in HPC systems using FCFS with back-filling since the assignment of a job to a set of processors is in general decided at the beginning of the execution of the job, while this assignment may implicate several processors and hence several running jobs. For this reason, the counters of all jobs running in the *principal group* are increased at the arrival of a new job.

The second challenge of extending the redirection to parallel jobs is the fact that the redirection relies on a static separation of the resources of the cluster into two static groups. The size of the *redirection partition* limits the size of the jobs that can be redirected, and thus limits the capacity for the redirection to take the best decisions. Parallel redirection needs to cope with this issue and determine the best partition sizes that will enable to redirect heavy jobs, without allocating an unreasonable number of resources for the *redirection group*.

3.5.3 Execution

At the initialization, the redirection needs to define the scheduling policy for both *principal* and *redirection* groups. We set FCFS with EASY back-filling for both principal and redirection groups, and we activate the redirection only for the *principal group*.

Algorithm 1 Scheduler using the redirection.
The scheduler manages two distinct groups of processors, with their corresponding job queue.

procedure SCHEDULER(SP_1,SP_2, j) ▷ SP_1 and SP_2 two scheduling policies and j a newly submitted job
  if $NbAvail(Resources) \leq q_j$ and j 1st in queue **then**
    Start j
  else
    $T \leftarrow$ Redirection(j, θ) ▷ c.f. in Algorithm 2
    if T is not $None$ **then**
      Kill T
      add T to the queue of the *redirection group*
    end if
  end if
  Schedule *principal group* with SP.
  Schedule *redirection group* with SP.
end procedure

Algorithm 2 Algorithm for the redirection mechanism.

procedure REDIRECTION(j, θ, $m' = \alpha m$) ▷ j is a newly submitted job, θ is the threshold value, and m' the size of the redirection group.
  for $k \in Runningjobs$ **do**
    if $w_j >= w_k$ **then** ▷ restricts impact of huge jobs
      $Counter_k \leftarrow Counter_k + 1$
    end if
  end for
  $T \leftarrow \{l \in Runningjobs, Counter_l > \theta, r_j \leq m'\}$ ▷ Create the set of jobs exceeding θ and fitting in the redirection group size
  if $T \neq \emptyset$ **then**
    $r \leftarrow conflictRedirectionPolicy(T)$ ▷ $conflictRedirectionPolicy$ tunes the selection of jobs in case of conflict.
    Reset every $Counter$
    return job r
  end if
  return $None$
end procedure

The algorithm 1 describes how the redirection is integrated into a scheduling algorithm, while the redirection mechanism itself is detailed in Algorithm 2. The parameters of the latter one are the set of m processors, the percentage of allocation α, and the threshold θ used for tuning the redirection.

When a job j enters the system for the first time (r_j), it is assigned to the queue of the *principal group*. If the queue is empty and there are enough available resources, the job is directly started on the *principal group*. However, if the queue is not empty or there are not enough resource available to start the job immediately, the job will wait and will be assigned to the scheduling queue of the *principal group*. In addition, the redirection increment the counters of the jobs running onto the platform. Once all submitted jobs have proceeded, the redirection mechanism can decide to redirect a job (Algorithm 2). The decision to redirected is taken when one of the job's counter exceeds the input parameter θ — the redirection threshold. Whenever a job is redirected it is killed and submitted from the beginning to the scheduler of the redirected group — in which no further redirection is possible. One important restriction induced by parallel jobs is that a job can be redirected only if its number of requested resources (q_j) is less or equal to the size of the *redirected group*. That is to say, for a job j to be illegible for redirection, it needs to satisfy $q_j \leq \alpha m$. After one redirection occurred, the counters of each running job are reinitialized to zero.

In some cases, several jobs can exceed the threshold at the same time, the redirection mechanism determines which job will be redirected according to a *conflict resolution policy* (e.g., select the job with the greatest *walltime*, or the max q_j).

Another effect that should be controlled is the impact of huge jobs on small jobs. Our algorithm focuses on identifying *big* jobs hurting the cluster performance, if job triggers the redirection of a small job then the redirection can negatively impact smaller jobs. To avoid this issue, we set a filter preventing jobs to trigger redirection of smaller jobs. The filter is configured such that if a job j is submitted, only the counter of each job k satisfying $w_k \geq w_j$ is increased.

3.6 Experimental Settings

3.6.1 Simulation and Inputs

The redirection algorithm has been integrated into the Batsim simulator [Poq17] which is available as an open-source software. Section 4.2 provides a more detailed description of Batsim. In our case, the scheduler is an external program implementing the EASY-Backfilling policy with the redirection mechanisms. The model of job used during the simulation is the delay profile (section 4.3), therefore the job's executions are not impacted by their placement on the cluster.

The performance of our algorithm depends on a set of input parameters, namely, the size of the platform, the workload and the inputs related to the redirection mechanism itself. The latter is the threshold of redirection (θ), the parameter α specifying the proportion of processors used for redirected jobs. To understand the behavior of the redirection, we ran thousands of simulations with several sets of inputs for the redirection in order to determine the best possible combination of parameters. The following of this section explains how each parameter impacts the scheduling performance and presents the parameters used in the simulation campaign.

3.6.2 The Workloads

The workload is a static set of jobs that need to be scheduled during a simulation, and it is given as a simulation input. Batsim uses a *JSON* definition of the workload, where each job is defined by a *json* object containing all the parameters introduced in Section 3.2. As in a production cluster, the scheduler only knows the *walltime* of the jobs. During the simulation, at the release time of a job, Batsim sends to the scheduler the given *walltime* of the job and the number of requested resources. The actual processing time remains known only by Batsim, allowing Batsim to kill the jobs exceeding their walltime (w_j). Since the users overestimate the execution times of their jobs, we choose to give to the scheduler the exact job processing time. This can be justified by considering better estimations for the exact processing times of jobs obtained for example by a learning algorithm [Gau+15] (this study is out of the scope of this work).

The workload is a sensitive input of the simulation, as its impact on the scheduling policy is unpredictable[LTZ19]. Choosing a workload for the simulations is complicated. One common practice is to directly use logs from production clusters, where we can get information about jobs — like processing time, time of arrival, etc. The web site *parallel workload archive* (PWA) [Fei19] hosts a consequent number of workloads dedicated to this purpose. To evaluate our algorithm, we run an extensive campaign of simulation on workloads extracted from logs of production clusters — namely, Curie, Intrepid [Tan+11] and Ricc, all provided by PWA. For each of the production cluster logs, we extract 20 weeks of jobs. The extraction routine extract weeks (168 hours) with a mean utilization of at least 70%, ensuring that the traces are loaded enough to benefits from the redirection — indeed under-loaded traces do not represent any challenge and can be handled by any other traditional schedulers. TABLE 3.1 summarizes the characteristics of the original traces used for the extraction. Inquisitive readers are invited to read the section dedicated to each log provided by PWA for an extensive description of the log used.

One limitation induced by the replay of static workloads is that the number of processors of the cluster needs to be the same as for the original platform. If the

Cluster	Total processors	Total jobs
Intrepid	163,840	68,936
Curie	93,312	773,138
RICC	8,192	47,794

Table 3.1. List of the clusters used for replay and their important characteristics.

simulation is configured to use fewer processors in the *principal group* than the number of processors of the cluster the workload stem from, some jobs may be unable to execute during the simulation. Enforcing the scheduler to forget such jobs is not something that is allowed. In the case of the redirection, we split the cluster resources into two independent groups — the greater the parameter α, the more resources we need to reserve for the redirection. To cope with the above problem, and still execute every job of the original trace, we instead increase the number of resources based on the parameter α. Once the resource groups are created the *principal group* holds the number of resources of the original cluster. Note that to ensure the fairness of our study, we also give the extra resources to the algorithms without redirection. One aspect of this choice is that a too large value of α decreases the load of the schedule, as there is room for more jobs. To limit this effect, we are interested in determine small α values for which the redirection improves the performances.

3.6.3 Redirection Parameters: α and θ

As stated before, θ is the value of the redirection threshold and α determines the size of the *redirection group*. Both parameters need to be wisely tuned to find the best possible redirection performance. The threshold of redirection will impact the sensitivity of the redirection. A high value will rarely be reached leading to idle time for the *redirection group*, while setting θ to a low value will trigger a large number of redirections, leading to a *redirection group* overloaded. A large value of α allocates a lot of processors for the redirected jobs, leaving room for a lot of redirected jobs at the cost of reducing the size of the *principal group*. On the other hand, reducing the α value increases the number of jobs that are not eligible for the redirection.

Both parameters need to be carefully configured, in order to obtain the best performance for the redirection mechanism. To determine the best configuration, we ran simulations using all the combinations for $\theta \in \{1, 2, 5, 10, 15, 25, 50, 100, 125\}$ and $\alpha \in \{0.1, 0.15, 0.20, 0.25\}$, for each of the 20 extracted weeks. The result and the selection for the best parameters are described in the following section.

It is worth mentioning that the best parameter configuration also depends on other factors, such as the distribution of the size of the jobs of a particular workload. For Instance, the minimum job size allowed for the Intrepid cluster is 256, meaning that

a *redirection group* of size smaller than 256 leads to a loss of resources without any positive affect on the performance.

3.6.4 Reproducibility

The redirection mechanism implementation is available on-line and open-source project, along with the data, the analysis source code and the visualisation tools[1].

3.7 Experimental Results

In this section, we first present the impact of the parameters in the performance of the proposed mechanism. Then, we compare the results of our algorithm to FCFS with Back-filling. In the following figures, as we extracted a set of workloads from the original logs, the results are presented as box plots.

3.7.1 Parameters Tuning

In Fig. 3.2 the y-axis corresponds to the considered objective. More specifically, we study (a) $BSLD_{avg}$, (b) $BSLD_{max}$ and (c) the average waiting time. For figures at the left, each column grid represents a different value of parameter α, the x-axis represents the different threshold values θ we used. Respectively, in the right figures each column grid represents a different value for the threshold θ, so the x-axis represents the different values for parameter α.

For θ (looking at the right figures), note first that small threshold values redirect a lot of jobs, and as a result, increases the load of the set of processors allocated for the redirected jobs. On the opposite side, high threshold values will never trigger any redirection, letting the processors in the redirection group idle. Based on these observations, we deduce that the *average waiting time* decreases as the threshold is increasing, for all workloads (Curie, Ricc and Intrepid). However for Ricc and Intrepid, we see that the *average waiting time* increases if the redirection threshold is too high ($\theta => 26$). This effect does not happens for Curie. On another hand, redirection leads to an important improvement for $BSLD_{avg}$ when using small and moderate values of θ, which leads to a larger number of redirected jobs. Specifically, we observe that there is an optimal threshold value for $BSLD_{avg}$ which is between 10 and 75 for the Curie instances, between 5 and 15 for Intrepid and between 5 and 25 for RICC.

For the percentage of allocation α (looking at the right sided figures), we first note that a small value may cause congestion in the redirection group, especially if there are a lot of redirected jobs. On the second hand, a very big value would improve the performance in the redirection group, but it is not realistic as we want to keep a

[1] https://gitlab.inria.fr/adfaure/evipar

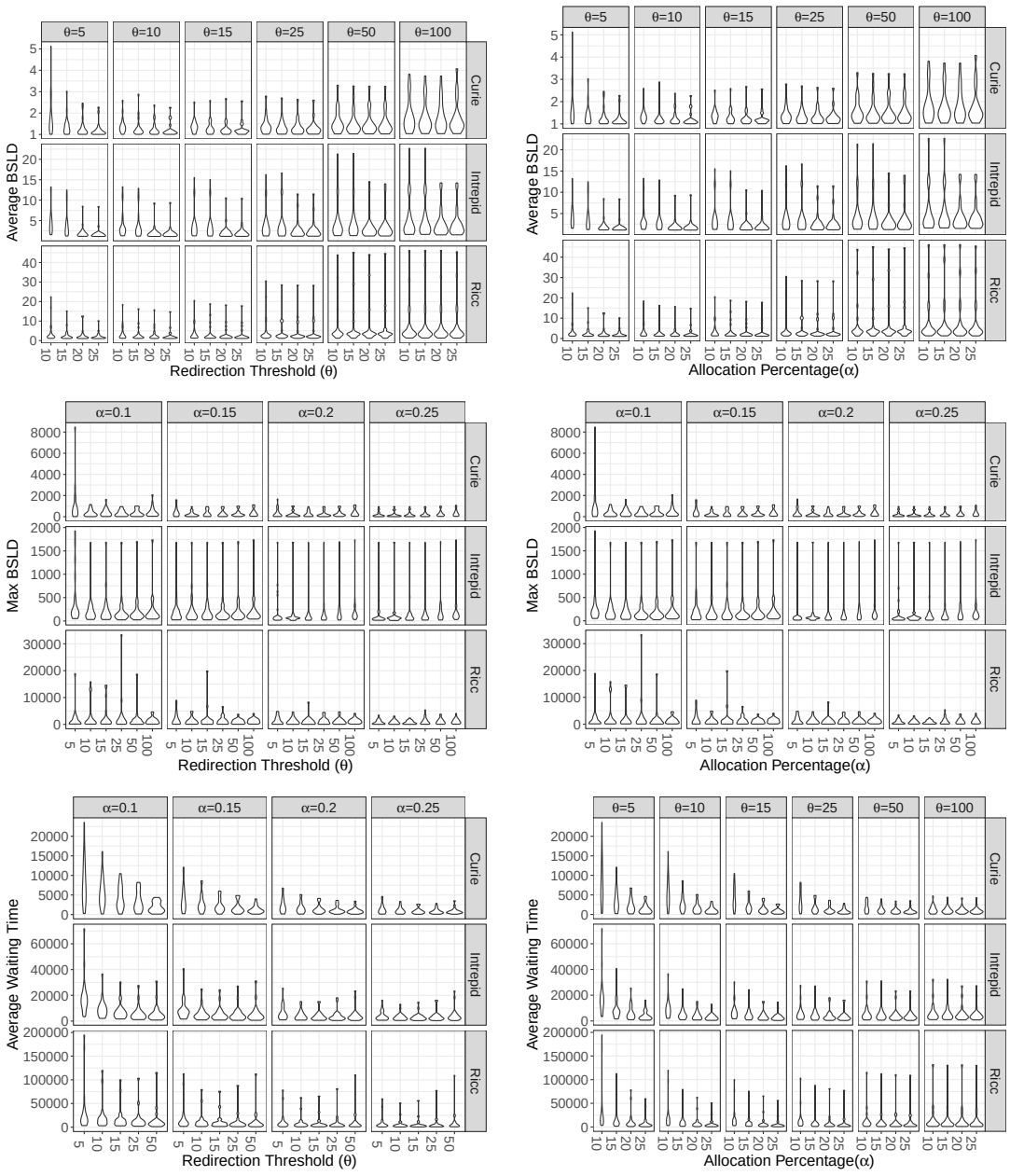


Figure 3.2.: **Lower is better** — For each of the 20 workloads extracted from Curie, we computed $BSLD_{avg}$ on the upper side and $BSLD_{max}$ on the midel line, the average waiting time ($WAIT_{avg}$) on the lower line. Each violin is created by 20 such workloads under a certain set of parameters. Each plot under the same line shows the same objectives, with different perspectives. On the left plots, the redirection threshold is in the x-axis and the y-axis represents the observed metric. Each column of the grid represents a different allocation percentage α. While on the right plots, the allocation percentage is in the x-axis, each column grid represents a different threshold.

reasonable total number of processors. We observe that the impact of the parameter α to the *average waiting time* objective is very important due to the increased number of processors used in the redirection group as we increase α. However, this effect stops for large values of θ ($>= 100$) where the α parameter does not impact the performances anymore.

We can retain two messages from the results of Fig. 3.2. First, the choice of the parameters is very important for the performance of our mechanism while the two parameters are strongly related: in case of a small redirection threshold which implies a lot of redirected jobs, the size of the redirection group should be larger in order to execute them without important delays. Second, the different parameters work well for different objectives. Thus, it is up to the system administrator to choose how to configure the system with respect to the type of quality of service that she/he wants to provide to the users.

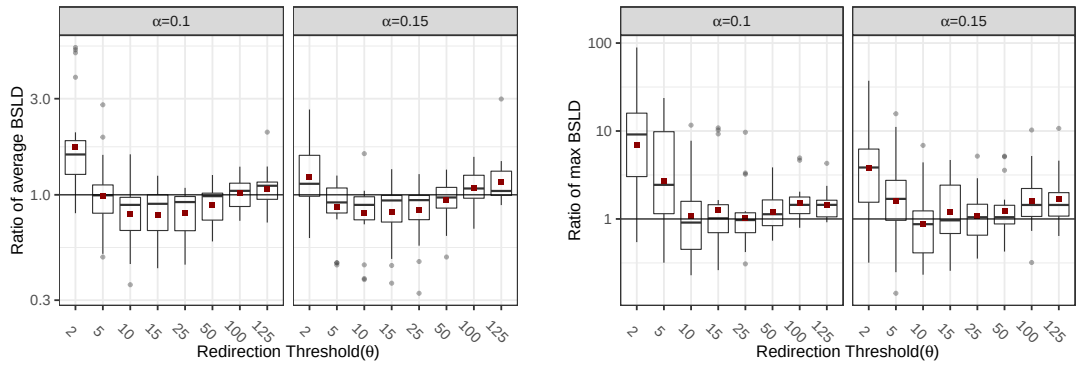
3.7.2 Comparison to EASY Back-filling

In this section, we perform a comparison of the performance of FCFS/EASY back-filling when using or not the proposed redirection mechanism. We focus only on $BSLD_{avg}$ and $BSLD_{max}$ objectives as well as on the values of the parameters θ and α which show good performances for these objectives. The y-axis of Fig. 3.3 shows the ratio of the corresponding objective function of FCFS with EASY back-filling policy (without redirection) over FCFS/EASY back-filling and redirection. If this ratio is smaller than 1, the redirection improves the performance.

Recall that the redirected jobs are restarted from the beginning to satisfy the non-preemptive constraint, and hence the workload executed in the presence of the redirection mechanism is larger. However, we observe in Fig. 3.3 that by appropriately choosing the parameters θ and α, the performance of FCFS/EASY back-filling can be improved by a factor of 10% for Curie, 20% for Intrepid and 40% for Ricc when considering the $BSLD_{avg}$ objective. On another hand, the impact of redirection is not so beneficial for $BSLD_{max}$ since the redirected jobs are restarted and since the job with the maximum $BSLD$ value tends to appear in the redirection group. However, there is always a couple of parameters for which both objectives are improved, e.g., $\alpha = 0.15$ and $\theta = 10$ for the Curie trace where the average improvement for both $BSLD_{avg}$ and $BSLD_{max}$ is around 10%.

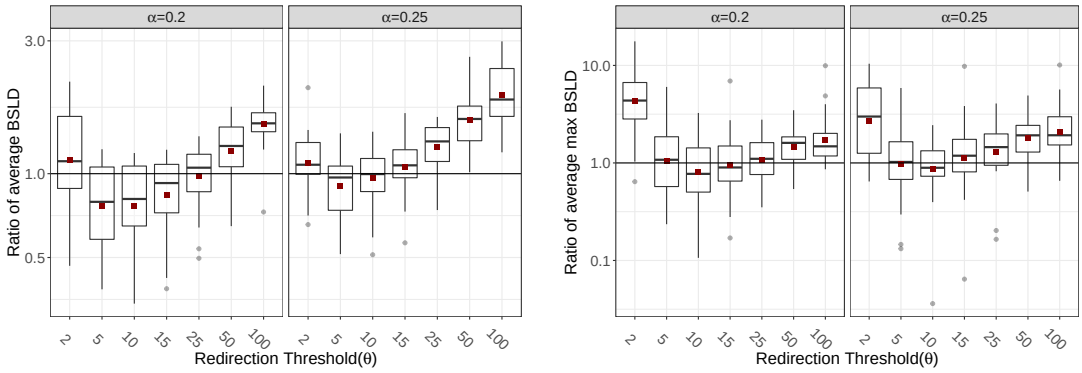
The last figures 3.4 show the impact of the performances of the redirections in respect of the average waiting time objective. The figure shows both the previously studied objective $BSLD_{avg}$ along with the average waiting time. We observe that the redirection can improve the average waiting objectives for two of the three used workloads. For the Curie, cluster the redirection has a negative impact of the average waiting time, while still improving the $BSLD$ objective. Intrepid on the

Comparison of EASY back-filling with redirection and without redirection for the Average and Max $BSLD$ objectives for Curie, Intrepid and Ricc.



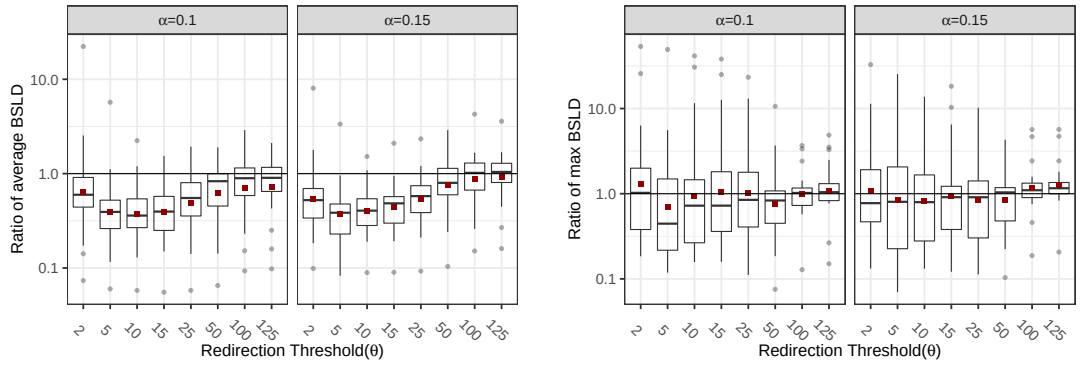
(a) **Curie — $BSLD_{avg}$**

(b) **Curie — $BSLD_{max}$**



(c) **Intrepid — Ratio of $BSLD_{avg}$**

(d) **Intrepid — Ratio of $BSLD_{max}$**

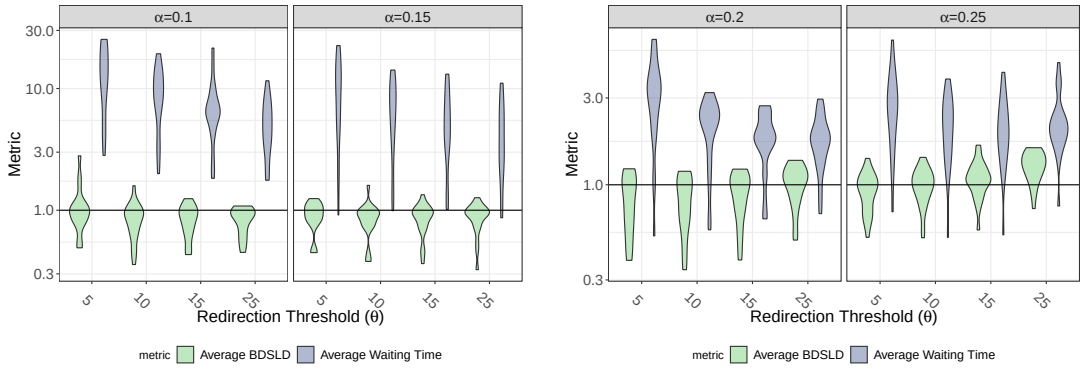


(e) **Ricc — Ratio of $BSLD_{avg}$**

(f) **Ricc — Ratio of $BSLD_{avg}$**

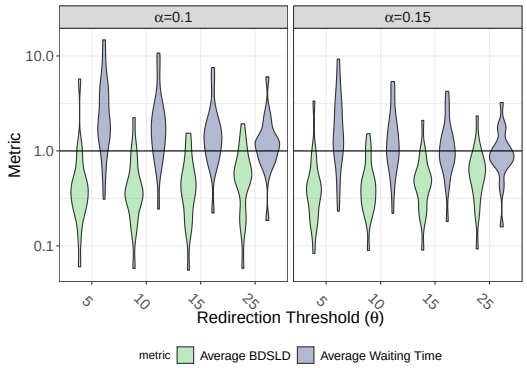
Figure 3.3.: Lower is better, under horizontal line means that the redirection is more effective — For each of the 20 workloads extracted from a cluster, we computed the ratio of $BSLD_{avg}$ without redirection over $BSLD_{avg}$ with redirection, and we did the same for the $BSLD_{max}$. Each box is induced by 20 such ratios. The black line is the median ratio and the red square the average ratio. The horizontal lines represent the quartiles. The figure presents the results for each cluster, the two upper figures deal with Curie, the middle one corresponds to Intrepid while the bottom one concerns Ricc.

Comparison of EASY back-filling with redirection and without redirection for the waiting time objective
Curie, Intrepid and Ricc.



(a) **Curie**

(b) **Intrepid**



(c) **Ricc**

Figure 3.4.: Lower is better, under horizontal line means that the redirection is more effective — For each of the 20 workloads extracted from a cluster, we computed the ratio of $BSLD_{avg}$ and we did the same for the average waiting time. Each violin is induced by 20 such ratios. These plots have the same structure than Figure 3.3, but focus on the *waiting time* objective. Ratio of the average waiting time in blue (dark gray) $BSLD_{avg}$ in green (light gray)

other hand, has the same behavior as for Curie, but the redirection does not arm the average waiting time as much, and some workloads can benefit from the redirection in respect of the average waiting time. Interestingly, for the Ricc cluster we show that certain set of parameters can improve both the average waiting time and the $BSLD_{avg}$, especially for the parameters $\alpha = 15$ and $\theta = 25$.

3.8 Conclusion

We proposed a mechanism based on redirection of parallel jobs that can be used on the top of other queuing scheduling and allocation policies. The selection of the jobs that should be redirected is done with respect to their impact on the performance of the other jobs in the queue. Although the redirection mechanism causes an additional load due to the kill-restart policy applied to the redirected jobs, it can improve the performance of the system especially for average objectives as shown in the extensive simulation campaign. Interestingly, the experimental results show that the redirection can exploit the benefits of preemption, even in cases where this is not explicitly allowed. Redirection presents interesting results, especially for $BSLD_{avg}$, which is a metric based on the satisfaction of the user. We showed that some workloads are more responsive than other to the redirection; further investigations are needed to understand these variations.

A direct limitation of the redirection is the creation of two distinct groups of processors. Splitting the resources can have a strong impact on the utilization. In case of a cluster filled with jobs that cannot be redirected (because of a too small partition size for instance), the resources of the *redirection group* are lost. Moreover, new requests need to constantly arrive into the cluster to trigger redirections. If no redirection is triggered, the resource of the redirection group are also lost. As a future work, we will consider using a dynamic partitioning of the resources, or no partitioning at all. The second issue is that one job cannot benefit from the whole cluster at the time, leading to a loss in computing capacity. This can be solved buy adapting the algorithm to allow huge jobs to use resources from both groups.

Tuning the parameters α and θ is an important step to setup the redirection. It challenges the ease to use the redirection in a production center without prior configuration. Simulating the redirection with past logs of the system can be used. Another solution is to use online-learning, such as multi armed bandit, to find the best parameters during production.

One important metric for system administrators is the resource utilization of the cluster. Considering the *principal group*, the redirection can use any scheduling algorithm. For instance, in our experiment we use FCFS-Easy backfilling, which is known to generate high cluster utilization. The *redirection group* also uses a basic scheduling algorithm (FCFS-easy in our case); the same reasoning can be applied.

Another impact of the redirection on the utilization is the lost computations induced by killing jobs. This lost computation aims at improving the user satisfaction as the cluster will use the extra place for more user jobs. As explained in the following paragraph, mechanism such as checkpoint restart could be leverage to lower lost computations.

Another future work is to evaluate how the redirection can settle into a more realistic environment. Redirecting a job brings new challenges to the job and resource management system. Some jobs can have side effects such as appending data to a dataset or holding active connections. Killing and restarting a job might require extra work from the users to be effective. In addition, the link with mechanisms such as checkpoint restart (C/R) should be studied, as it enables partial preemption of a job.

This work shows that it is possible to adapt scheduling policies from theoretical work to a more practical use case using simulation. However, the current simulations are not fully representative of a real use case, as many effects happening on the platform are not taken into account during the simulation. The following of this dissertation proposes to extend the simulation capabilities of Batsim by incorporating models for the job executions.

4 Scheduling Simulation with Job's Models

4.1 Introduction

From the RJMS perspective the jobs are black boxes because the RJMS has only a partial view of what is happening on the platform. For a job submission, the user must provide the number of resources required (in recent system one may also need to specify the kind of resources), a walltime (detailed in section 1.2.1), and a script to start the job. The RJMS does no (or few) monitoring on the computing platform during the execution of the jobs. Alone, these information lead to a partial view of the computing cluster. However, to build a more accurate vision of the computing platform it is necessary to obtain more information from the jobs, such as how they use the resources of the computing platform. In [Eme13], the author shows that by using more information about the computing platform, one can detect bottlenecks and therefore improves the overall system performance.

In a production system the job executions are affected by various factors:

- Physical capacity of the computing platform, the speed of the nodes, the interconnect capacity and the storage performance.
- The placement of the applications on the computing platform.
- The application itself, how it uses its allocated resources (network, IO and CPU).

In this environment, the scheduling and placement decisions impact the performances of the applications, and therefore impact the performance of the whole cluster. For instance, the locality of MPI applications has an impact on their performances and its optimization has received attention in the literature, as TreeMatch[JMT14] or EagerMap[Cru+19] aim at improving locality of MPI application to reduce their inter node communications.

Traditionally, HPC applications are parallel applications performing homogeneous phases of communications and computations on an homogeneous computing platform. However, recent platforms have become more and more heterogeneous, and the applications evolved in such a way that the RJMS has to handle more complex user demands — such as allocating diverse accelerators, BurstBuffers or General Purpose Graphical Processing Units (GPGPU). To handle these requests, the RJMS

has to increase its resource description definition to correctly manage the computing platform and answers the user demand. On the other hands, from the RJMS perspective the jobs are black boxes that need to be executed onto the computing platform. The lack of information about the applications and their usage of the computing platforms can lead the RJMS to take decisions with undesirable side effects. Indeed, the scheduler can make harmful decisions such as launching two applications that will intensively use the network or the file system leading to performance reduction [Bha+13; Bro+18], or reach a power state too high for the cluster.

In this context, evaluating the scheduler performance is a real challenge. It becomes crucial to integrate into the experiments the behavior of the jobs concerning their usage of the computing resources.

One approach is to use simulation to study new scheduling algorithms. The simulation instantiates a model of a computing platform managed by the studied scheduling algorithm. A technique of workload replay can be used to evaluate the performance of the algorithm (see section 1.3.1). However, these simulations often use simple models for the jobs and the platform. The jobs can be modeled as fixed amount of time to spend on a platform, while the platform itself is modeled as an array of heterogeneous computing nodes, this is the approach made by the simulator Accasim [Gal+20]. This model of simulations is used for its simplicity and its scalability. One can simulate months of a computing platform in a few hours. Nevertheless, this model fails to capture many effects occurring when a job is executed in a production environment. For instance, jobs are not impacted by their placements on the platform or by the other jobs running at the same time. These limitations limit the scope of the feasible studies. For instance, evaluating the impact of different allocation policies on heterogeneous cluster is not easily feasible.

To cope with these limitations, we need a job model that is able to capture the behavior of real HPC jobs. In particular, we want to integrate the resources consumption behavior of the applications in simulations, such as network usage or computation. Moreover, looking at resources consumption enables to model applications independently of their underlying technologies as we can focus on their activity on the computing platform, and not their implementation.

One requirement is that simulations need to be scalable in time, as we want to simulate different scheduling policies or platform configurations to compare them. The simulation time of a single job should not drastically increase as one simulation could embed thousand of different jobs representing days, months of a computing platform usage. However, increasing the complexity of a simulation model necessarily increases the computing time of the simulation. Depending on the requirement of the simulation, one can trade performance for precision.

In this chapter, we present how we leverage the Batsim [Poq17] simulator to create a first step towards scheduling simulations with job models. The job models enable to simulate slowdown effects based on the platform capacity and inter-job interferences. Batsim is a batch scheduler simulator which internally uses SimGrid [Cas+14], a generic distributed platform simulator, to simulate the applications running on a computing platform (including network interference, and platform topology).

In section 4.2, we present Batsim and detail how it works, we then discuss why it is a good candidate to support our work — in this first part we also presents SimGrid (4.2.1) and its simulation models. In section 4.3, we present the different job models available with Batsim (some are provided by SimGrid), and we compare their overheads and the different phenomena that they are able to simulate.

The work presented in this chapter has been done in collaboration with Millian POQUET and Olivier RICHARD, and has led to one publication [FPR18] (in french).

4.2 The Batsim Approach

Batsim [Poq17] is a scientific simulator to analyze batch schedulers and scheduling policies. It leverages the simulator SimGrid and uses its simulation models. The section first introduces SimGrid, and the remaining of the section presents Batsim and its architecture.

4.2.1 SimGrid

Why SimGrid?

Three main classes of simulator exist for the simulation of large scale distributed infrastructures. The classes are based on how the network is simulated: The packet-level simulators, the delay simulators and the flow level simulators [Leg15].

- **Packet-level** simulators are discrete event simulators where the events are all the network packet needed for the communications. This approach has a high level of realism, as every byte that is transiting through the network is simulated. However, in the case of large scale computing systems, simulating every packets is resources demanding and significantly increases the time of the simulations. The simulators NS2 [IH09] and CODES [Mub+17], for instance, use this model. CODES accelerates the simulation using parallel branching predictions, which is resources demanding.
- **Delay-based** simulators simulate the network traffic as fixed amount of time between the communications. This model is very effective and scalable as no extra computation is required. However, it fails at simulating potential network contentions.

- **Flow-level network model** is a more scalable alternative of the packet-level network model enabling to study network contentions. Instead of simulating all the network packets, the communications are modelled as unique entity: *flows*. This level of abstraction enables to increase the scalability of the model, while being able simulate network contentions for large scale systems. The simulators SimGrid [Cam11] and GridSim [Sul+08] are such of these simulators.

SimGrid is a framework to design simulators of distributed applications [Cas+14]. It is a versatile simulator created not to be specific to one computing domain. Instead, SimGrid provides network and computation models to be used across different distributed computing domains, such as HPC, Grid, peer computing, cloud computing and volunteer computing.

The SimGrid's network model is based on the flow model, instead of simulating all the packets of the network traffic (packet-level simulator), it relies on a purely analytical model [Cam11]. This level of abstraction on the network model makes it possible for Batsim to simulate large scale platform along with the execution of the applications. The soundness and the validity of the SimGrid's network model has already been evaluated [FC07; VL09]. Besides the network model, the SimGrid project actively developed, and the community is active and has been very helpful regarding their users.

How does it works?

SimGrid provides models to simulate a distributed computing platform, and provide ways to model the applications using it. The platform is defined by hosts, each having a computing speed, and links connecting the hosts. Each link is associated with a transfer speed, and a latency.

To model an application, one creates activities on the platform, each activity induces an amount of work to the simulated platform. One activity can be an amount of computations, or an amount of data to transfer between hosts, the activity is finished when this amount of work reaches zero. SimGrid transforms the different activities in a set of linear constraints, in which each activity is represented by as many variables as the number of resources it uses. Each used resources is represented by a constraint in the system, and the capacity of the resource bounds the constraint.

Solving an optimization problem associated to the linear model enables to create a resources attributions for each activities of the simulation. A more detailed and precise version of the core of the simulation is described in [Leg15].

4.2.2 SimGrid Provided Models

Additionally, SimGrid features different platform models to tune how SimGrid solves the activities. The different models are described in the following subsections.

SimGrid MPI (SMPI), Online and Offline

SMPI is a SimGrid implementation of MPI enabling to emulate an MPI application directly with SimGrid, on a single computer [Deg+17]. Interestingly, SimGrid leverages its MPI implementation by integrating trace replay mechanism. The trace replay enables to simulate an MPI application execution using a trace of a previous execution [Des+11]. Instead of executing the real application's code, the trace replay executes the actions contained by the trace. The traces are called Time-Independent Traces(TiT) because it contains only the action performed by the traced application without the time at which the action occurred. More specifically, the TiT trace contains the actions done by each process of the application during its execution sorted par order at which they occurred. Relevant information are logged, if the action is a communication the amount of data is logged as well as the processes involved. SimGrid simulates the trace by unfolding the actions done by the processes of the application.

Parallel Task (*ptask*) Model

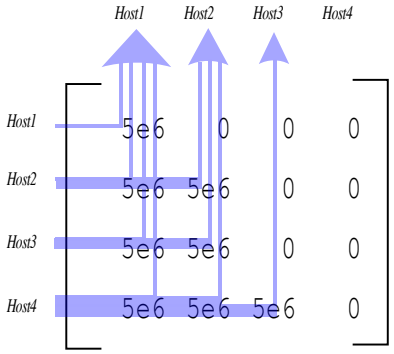


Figure 4.1.: *Ptask* communication matrix. The arrows show the direction of the communications.

The parallel task(*Ptask*) model defines a way to describe a parallel application by specifying two information concerning its usage of the computing platform. The objective of a *Ptask* is to simulate a parallel application running on a platform without simulating its detailed behavior, it aims at simulating a parallel task that will be executed on several hosts (or cores) of the computing platform. *Ptask* features a way to bundle the usage of an application on a platform to create an homogeneous

progress for the whole application. The model's running time — the time predicted by SimGrid— depends both on the platform (its performance and topology) and on the other activities that execute concurrently on the platform.

To compute this model SimGrid requires two information. The first information is an amount of computation to perform on each host allocated fort the model. The second information is the amount of communication from and to each host, it is represented as a matrix. Figure 4.1 illustrates how the communications are represented.

SimGrid computes the *ptask* progress by first finding the bottleneck of the task, this is the limiting factor of the *ptask*. From this limiting factor, the completion time of the *ptask* (in the current state) can be computed. As the *ptask* (may) bundles different activities on other resources and homogenizes the progress, SimGrid adapts the load on the other resources to reduce the speed of the *ptask*'s activities.

Figure 4.2 unfolds an example with three *ptasks* on a platform with two hosts (*node-1* and *node-2*) connected by one link. The speed of the hosts, is $100\ flops/sec$, and the link capacity is set to $100\ b/s$ — the values are simple to facilitate the example:

1. First, at $t = 0$ the platform has only one *ptask*, the purple present(plain borders), using all the resources. This *ptask* is configured to perform $1000\ flops$ on each hosts, and a data transfer of $600\ b$ between *node-1* to *node-2* . The bottlenecks are the CPUs, which are $100\ \%$ loaded (using the $100\ flops/sec$), and therefore the link is only loaded at $60\ \%$ (using $60\ b/s$) to finish at the same time.

2. The blue *ptask* (dashed borders) arrives on second late, at $t = 1$, and only uses the link, for a total amount of $1000\ b/s$ to perform. After one second, the purple *ptask* still has $900\ flops$ on each hosts to compute, and $540\ b$ to transfer. SimGrid shares the link to both tasks to use 50% of the link's capacity each, therefore the pink task has to decrease its load (to $83.3\ \%$) on the two hosts to adapt the pace as the Link becomes the bottleneck.

3. Finally, 5 seconds later at $t = 6$, the yellow task (dotted borders) arrives and has to perform $300\ flops$ on *node-1* and $1000\ flops$ on *node-2*. *Node-2* is the bottleneck of the yellow task, and uses half of its total capacity, the second half being allocated to the purple task. Then purple task adapts its load. *Node-2* becomes the new bottleneck and it uses $50\ \%$ of its total capacity, the load of the purple task on *node-1* is also adapted to $50\ \%$.

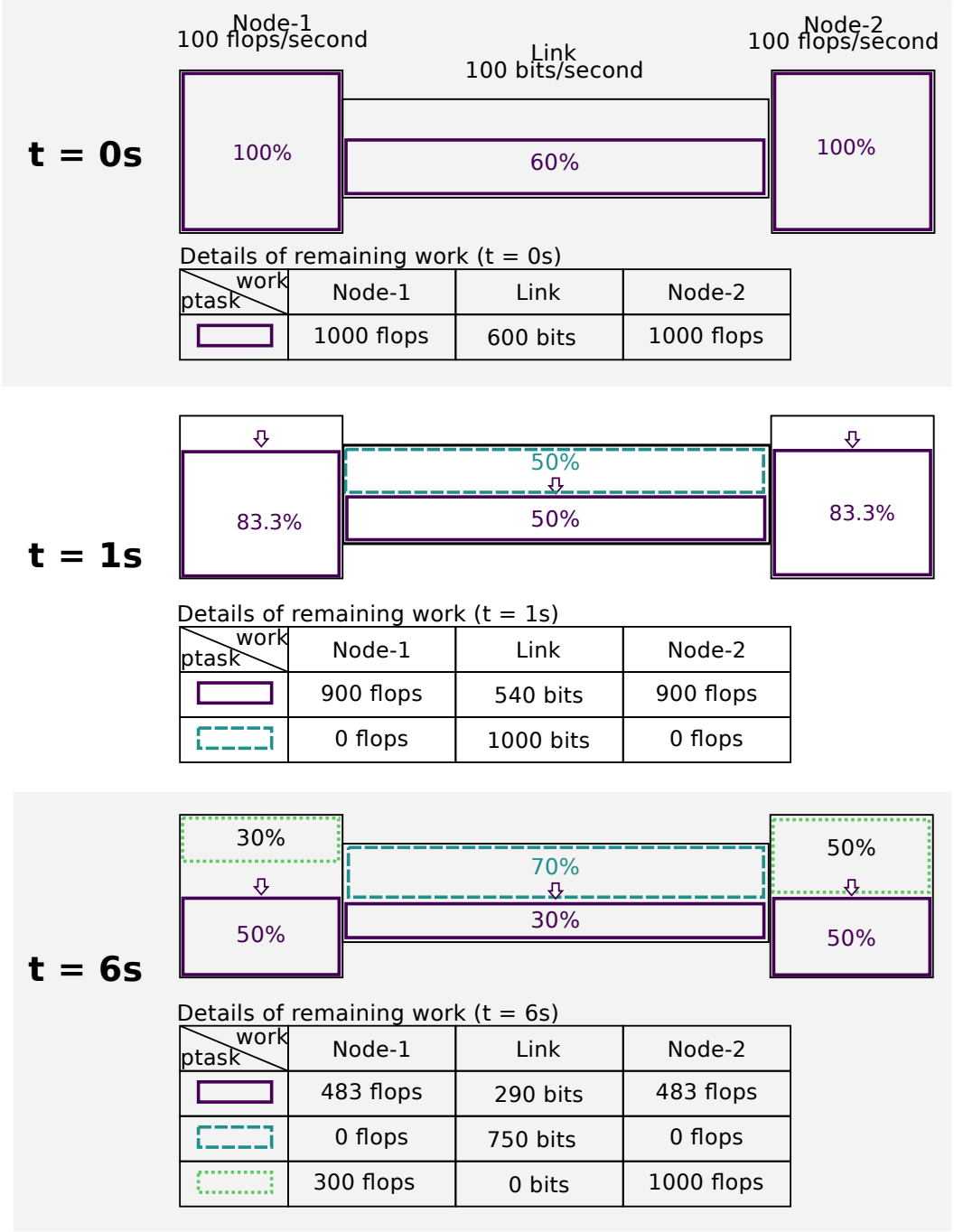


Figure 4.2.: Example of how the application progress is computed using the *ptask* model. At $t = 0$, the purple task (plain borders) uses the two nodes at their full capacity, while the link is used at $60\ \%$. Then, at $t = 1$, a new task arrives (dashed borders) performing a data transfer on the link, taking half the of the link's bandwidth. Hence, the link becomes the bottleneck of the purple task which has to adapt its load on the two hosts. Finally, at $t = 6$, the green task (dotted borders) arrives with some work on the two nodes, taking $50\ \%$ of the host's speed.

4.2.3 Batsim: Infrastructure Simulator for Resource Management

What is Batsim?

Batsim [Poq17] is a simulator to analyze batch schedulers, its is an open source software and is actively developed. Batsim uses SimGrid to simulate the computing platform and the jobs running on it, while the scheduling decision are left to an external program called the scheduler.

Figure 4.3 depicts the architecture of Batsim. The scheduler communicates with Batsim through the Batsim protocol, which is a network text-based protocol. All the scheduling decisions, such as allocating a set of resources and deciding when a job starts are left to the external scheduler. The jobs are known by Batsim and are submitted to the scheduler using the protocol. This architecture following the separation of concerns enables scientists or system administrators to focus on their scheduling algorithm and their evaluations, while the simulation capabilities are provided by Batsim. Figure 4.4 depicts a simplified exchange between Batsim and the scheduler. Batsim submits events to the scheduler, and waits for its response before continuing the simulation. The basic events are the job submissions and job ending. The scheduler controls the jobs of the simulation, it asks for their execution.

Batsim is built upon SimGrid: It enables to use the SimGrid simulation models, such as the platform models and description to simulate the execution of the jobs (described in chapter 4). Each input job of the simulation has an execution profile attached to it, describing how the job should be simulated.

The separation of concerns between the scheduler and the simulation brings great potential to quickly prototype and evaluate new scheduling algorithms. The internal usage of SimGrid enables to benefit from the SimGrid models for HPC applications and infrastructures.

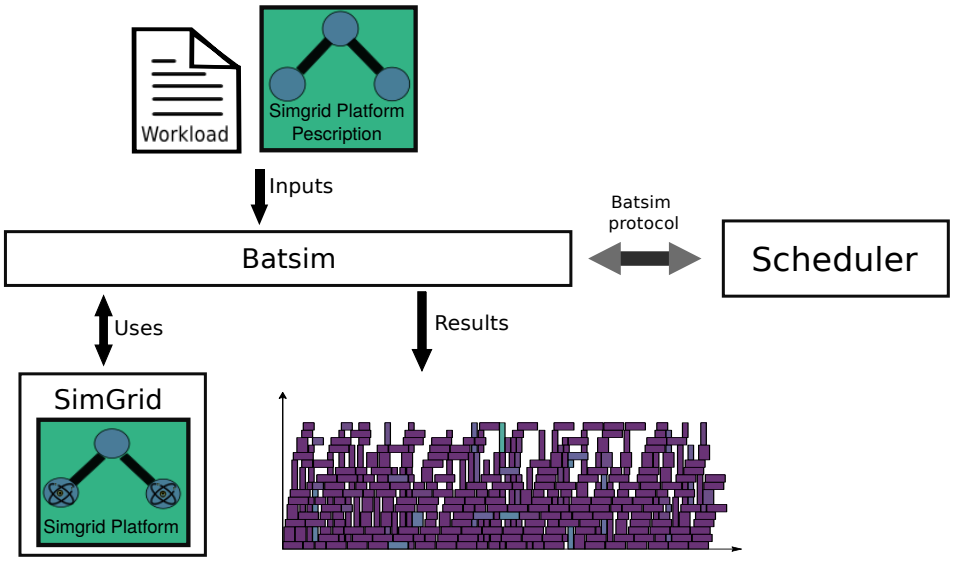


Figure 4.3.: Batsim overview, the scheduler takes decisions from the network protocol. The simulation inputs are a platform description and a workload of jobs handled by Batsim. The simulated platform is handled by SimGrid. The results of a simulation comprises data about the simulation, such as jobs resources allocations and execution times.

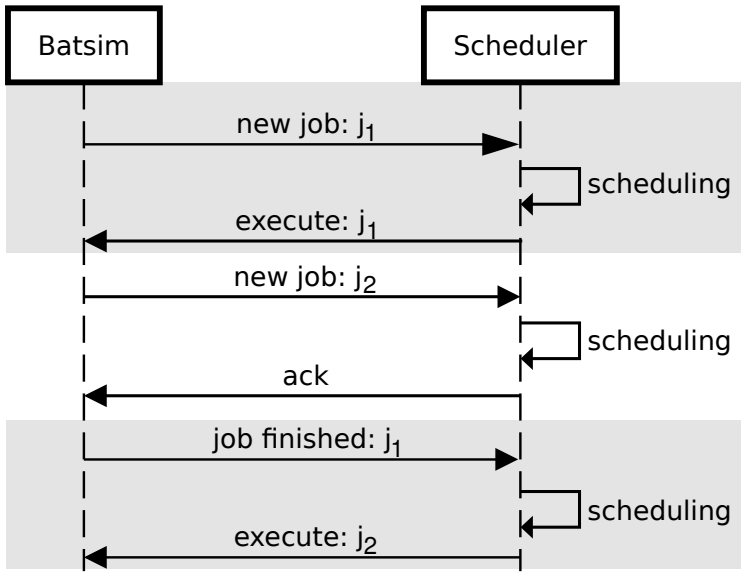


Figure 4.4.: Example of a simplified exchange between Batsim and the scheduler taking the scheduling decisions. The protocol is based on request-answer, Batsim waits for a response for each request (the grey, and white layers represent a request-answer phase). Batsim sends events to the scheduler, such as new job submissions, or job completions. The scheduler takes all scheduling decisions.

4.3 Job Execution Profiles

A job execution profile is an additional information specifying the model used for the job simulation. Depending on the model, the jobs can adopt different behavior during the simulation. Batsim provides different profiles based on SimGrid application models.

This section presents the profiles available in Batsim and proposes a preliminary evaluation of the different job models.

4.3.1 Profile Types

The considered profile types are:

- Delay profile.
- Parallel Task Profile (***Ptask***).
- Time Independent Trace (**TiT**) profile.
- Sequence profile.

Delay Profile

The profile delay is the simplest one. Executing the job consists of considering the execution of the job as a predefined amount of time during which the nodes of the job will be occupied. This profile is commonly used in scheduling simulations for its simplicity and its accessibility. It requires no information about the job resource usage, only the time the job spent in the system. For instance, RJMS logs provided by Parallel Workload Archive directly feature the running time of the job which can be directly adapted into a profile delay. Job delay is also the most scalable Batsim profile as it requires no extra computations.

The jobs simulated with this profile have a processing time independent of their execution context. If multiple jobs share the same network, their running time will remains the same as defined in the workload. This profile can be used in scenario were the context of execution of the jobs does not impact their execution times, or if the scalability of the simulation is a key objective. This is the case, for instance, for homogeneous cluster with no network interference, or for workloads composed of only sequential jobs (jobs using a single node) without co-location of jobs In a broaden way, this is the case when jobs don't have access to shared resources, which remains marginal as most cluster have a shared Parallel File System.

Parallel Task Profile

This execution profile provide a way to use the parallel task model (*ptask*) featured by SimGrid for the jobs. The profile requires two inputs. A vector specifying the amount of computation to perform on the allocated hosts. The second input is a matrix of communication, the matrix declare the data quantity that needs to transit between one host to another. The number situated at line i and the column j gives the quantity of data to send from host i to host j, as shown in figure 4.1.

This model is detailed in section 4.2.2 and gives a way to simulate parallel jobs. However it relies on the hypothesis that all jobs behave the same way and that the behavior of a job does not evolve during its execution. For instance, for an application that have an iterative pattern, such as consecutively performing the different phases of communications and then computations, this profile may smooth the effect merging the different phases into a long phase. Hence, the iterative behavior of this job is not captured with this model.

This profile bring interesting perspective for the simulation of scheduling algorithm. The jobs of the simulation are responsive to their environment, and their simulation times depends on the description of the platform and on the other jobs using the platform at the same time. Using this model for scheduling simulation enables to observe effects such as network interference.

Time Independent Trace (TiT) Replay with SMPI

Batsim provides an execution profile enabling to use TiT replay as job profile during the simulation, this model is detailed in section 4.2.2. The trace must be provided with the profile.

Sequence Profile

Lastly, the *sequence model* (or *composed*) enables to chain profiles to execute them in sequence. A number of repetition can also be specified. This profile inherits the properties of the profiles it is composed of. The main purpose of this model is to specify different job behaviors. For instance, as stated in section 4.3.1, the *ptask* model assumes that the application has the same behavior throughout its execution and therefore can fail to capture bursty effects. Sequence profiles have the ability to render more complex application behavior by composing different models. However, its computation time can increase as the number of profiles of a profile increases (especially with *ptask* and the TiT).

4.3.2 Profiles Evaluation

The multiple profiles featured in Batsim enable to set up different experiments depending on the effect(s) that the simulation should capture.

On the one hand, the delay is the simplest and fastest profile, however the number of effects that we can observe with this profile is limited, observing network interference for instance is not possible. On the other hand, the time independent trace replay is an heavy models that enable to observe many effects on the executions of the jobs. This model is precise and has been evaluated by the SimGrid community, however it has been principally designed for study focusing on one application, and may not be scalable to study thousands of applications as it is necessary for RJMS studies. Alternatively, the *Ptask* model proposes an higher abstraction that bundles all the activities of an application (computations and data transfers) into an homogeneous model that is more scalable.

To illustrate the different models and their trade-offs between performances and the captured effects we set up an experimental setting with Batsim. We study two scheduling algorithms using the same input workloads — same number of jobs, same arrival date, same number of requested resources — with different execution profiles.

Experimental Setup

For the schedulers we use a simple scheduling algorithm, First Come First Serve (FCFS) under two declinations. Note that keep the experiment simple, we don't use backfilling, which is an improvement commonly used in production RJMS — backfilling is detailed in chapter 3. The first declination restricts the resources allocated for a jobs to a contiguous set of machines (forced contiguity) [Luc+15]. The contiguity of an allocation tends to decrease the number of switches used by the job if the machines are numbered in increasing order by switch, at the cost to not use all the free available resources if the allocation conditions are not fulfilled. The second declination has no restriction on the allocated resources (free allocation). FCFS is a simple scheduling algorithm that prioritizes all waiting jobs by order of submission (more recent jobs have a lower priority). A job can be executed on the allocated resources only if it is the first in the waiting queue and if enough resources are available. Both algorithm are available in the batsched[1] project, which is a tunable scheduler made for Batsim.

The method used to generate the workloads is described in the 9th chapter of the book [Fei15b]. The arrival times of the jobs follows the Weibull distribution. We generate two workloads, the first is composed of 512 jobs and the second is composed

[1] https://gitlab.inria.fr/batsim/batsched

of 1024 jobs. Each workload contains three different kinds of jobs using 8, 16 and 32 resources representing the *ft* application available in the NAS parallel benchmark suits (NPB) [Bai+91]. The *ft* application is available for different numbers of resources, the jobs using 8 resources use *ft.C.8*, *ft.C.16* is used for the 16 resources jobs and *ft.C.32* for the 32 resources jobs.

This objective of this experiment is to illustrate the different profiles with a concrete example. Each workload contains the same number of identical jobs (same arrival times, same number of requested resources), only the job execution profiles are different. The first version uses the TiT replay profile, the second workload version uses the *ptask* profile while the last version uses delay profiles.

SimGrid can directly generate a TiT from a SMPI simulation, therefore the generation of the Tit profiles is made by executing the *ft* application with SMPI.

To generate the *ptask* profiles we need two information. The first information is the total amount of computations (in $flops$) executed on each resources of the jobs. This information can be extracted directly from the TiT(s). The second mandatory information is the communication matrix (depicted in 4.1). The TiTs contain details about the communications made by the application, however the precise communication matrix cannot be directly extracted from the TiT(s). This is mainly due to the fact that the TiT contains the high level operations made by the traced application such as send and receive, or the collective name (broadcast, reduce etc). In the case of the broadcast collective for instance, the precise communication matrix depends on the underlying algorithm used to achieve these high level communication collectives. To extract the communication matrix we replayed the TiT application using SMPI (in SimGrid) with the tracing activated. The SimGrid tracing is able to generates a trace of the execution of the TiT replay into a different format (paje [KOB00]) containing the detailed point-to-point communications.

Finally, the only information needed to generate the delay profiles is an execution time. Similarly to the *ptask* profile generation, we execute the TiT application with SimGrid and we use the time predicted by TiT the simulation as execution time for the profile.

The platform used for the simulation is modeled from the Graphene cluster, which was part of the Grid'5000 test-bed[2]. The calibration values are obtained from the SimGrid project[3]. Figure 4.5 depicts the network topology of the platform. On this platform we observe different network contention points. The first contention point is at the machine level, if two nodes under the same cabinet sends data to each other, max bandwidth capacity (of $1\ G$) can be reached. The second possible

[2]Unfortunately, the platform has been removed from Grid'5000

[3]More details can be found about the graphene calibration at https://simgrid.org/contrib/smpi-saturation-doc.html

contention point is inside a cabinet, if we group all machine in pair and each pair communicates with each other, the switch of the cabinet can be overloaded. The last contention point is located between cabinets, if all the machines from the cabinet 1 (for instance) communicate with a different machine of the 2nd cabinet, the capacity of the top link of $10\ G$ bit can be reached.

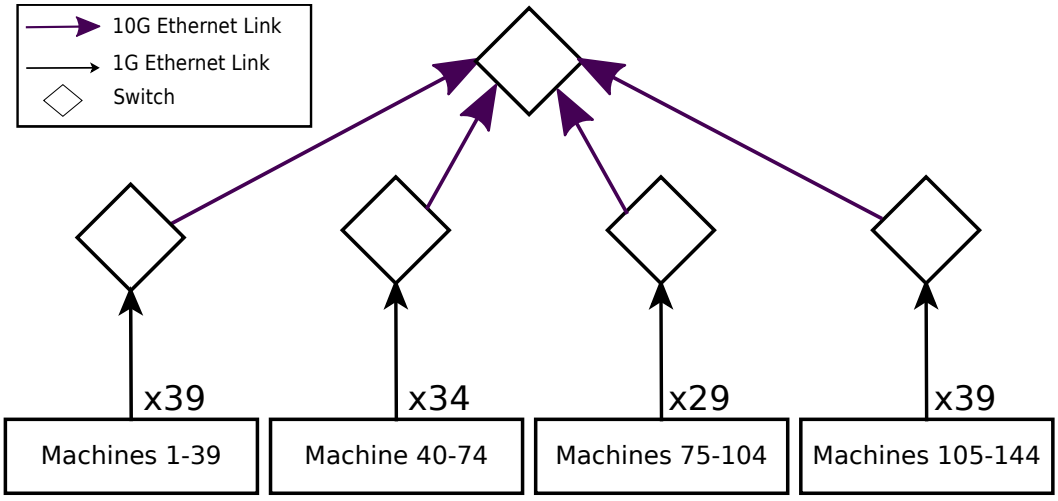


Figure 4.5.: Network topology of the Graphene platform. The platform is organized in 4 irregular cabinets, connected to a top switch with a $10\ G$ Ethernet link.

Results

In the first place, we take a look at the simulation's performance. Figure 4.6 compares the time took to run the full workloads on a personal laptop with a processor *Intel(R) Core(TM) i7-7820HQ CPU @ 2.90GHz*. The first thing to notice, is that the SMPI model using the TiT traces is the slowest model (more than $50\ s$ for the 512 jobs workloads and between 125 to 175 for the workload containing 1024 jobs). The delay and *ptask* profiles are faster to simulate, for the 512 jobs workloads the simulation took less than 1 second, and less than 2 seconds for the 1024 jobs workloads. Additionally, we observe that the simulation time also depends on the scheduling algorithm used within the simulation. This is especially the case when considering the TiT execution profiles, the *ptask* are also impacted by the scheduling algorithm used, whereas the profiles delay are not impacted by this factor.

Secondly, we take a look at the results of the simulations. Figure 4.7 shows the Gantt charts of the jobs scheduled during the simulations. Figure 4.8 shows the distribution of the running times and waiting time of the jobs for each of the different execution profiles and scheduling and scheduling algorithms. From the Gantt charts figures we can see that both the *ptask* and the TiT profiles we see that changing the allocation policy has a visible impact on the resulting schedule. In particular, the allocation policy seems to have a negative impact of the waiting time of the job

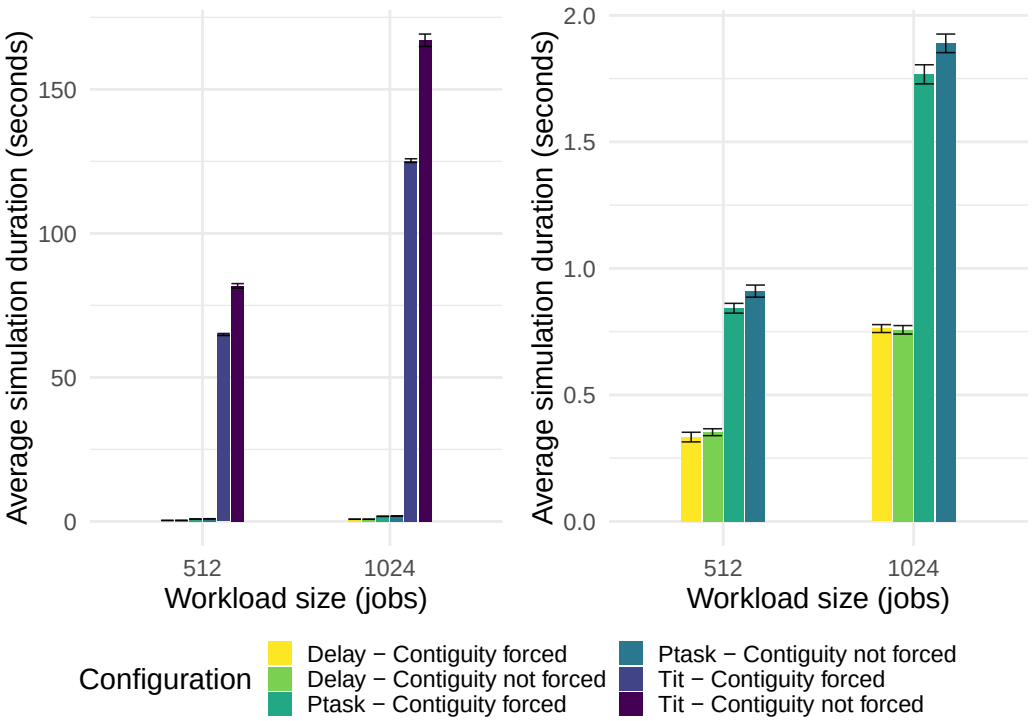


Figure 4.6.: Mean (the error bar are the standard deviation) of the (real world) duration for the simulation of the entire workload over 10 different executions. The figure on the right zooms over the delay and the *ptask* profiles.

when the contiguity is not forced. This can be explain by the fact that the forcing the contiguity provides a better jobs placement leading to lower execution time of the jobs, this effect is visible when comparing the histogram of the execution time in figure 4.8.

On the other hand, the delay profiles have similar results when comparing the two scheduling algorithm. This is expected as the execution time of a job does not depends on its placement on the platform nor on the other jobs executing at the same time.

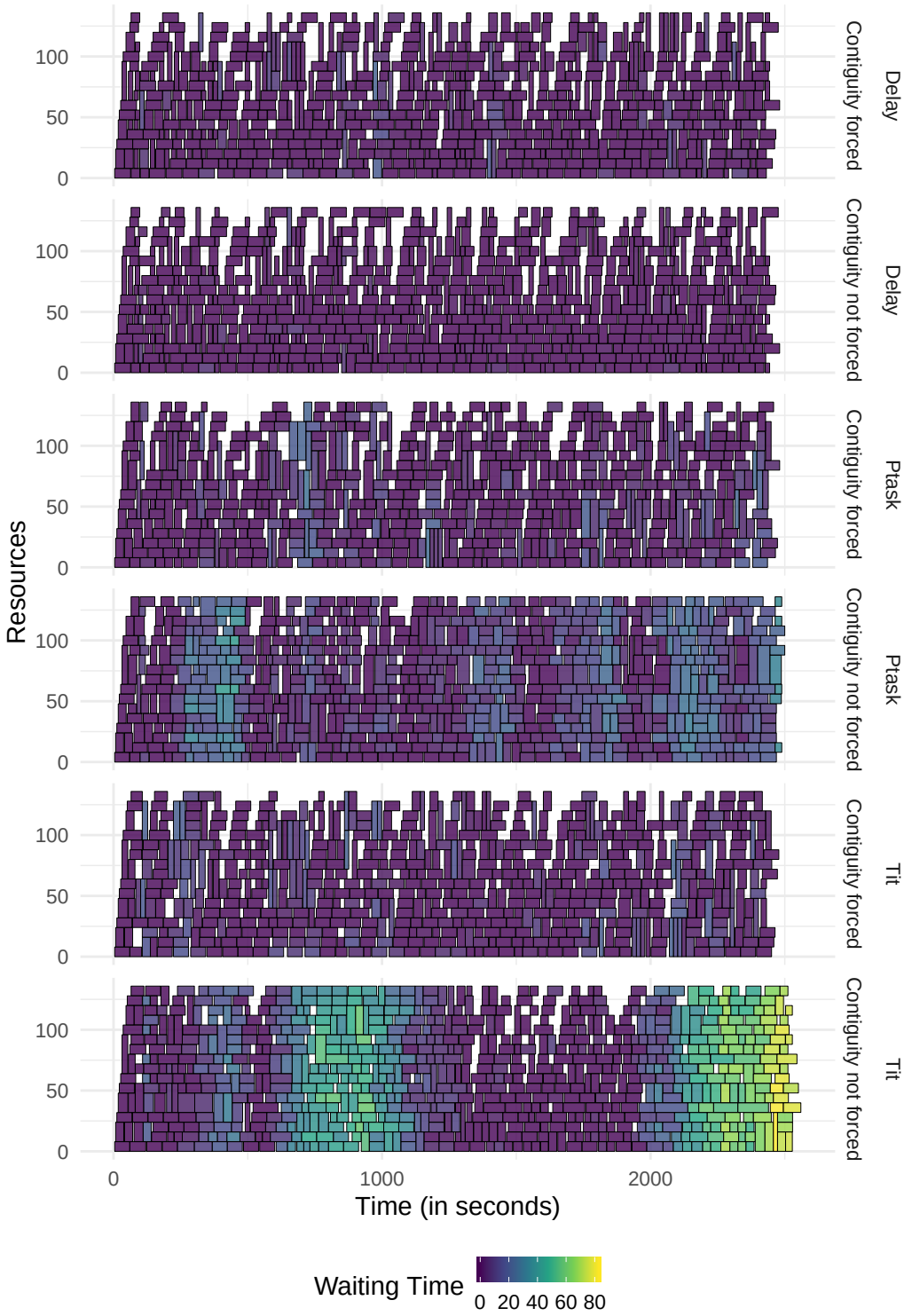


Figure 4.7.: The figure presents only the schedules for the workload containing 512 jobs. Each Gantt chart depicts the scheduling decisions taken by the scheduler process, the first two upper charts correspond to the profile of execution delay, followed by the *ptask* execution profile. The last two charts on the bottom are the Gantt charts obtained with the TiT profiles.

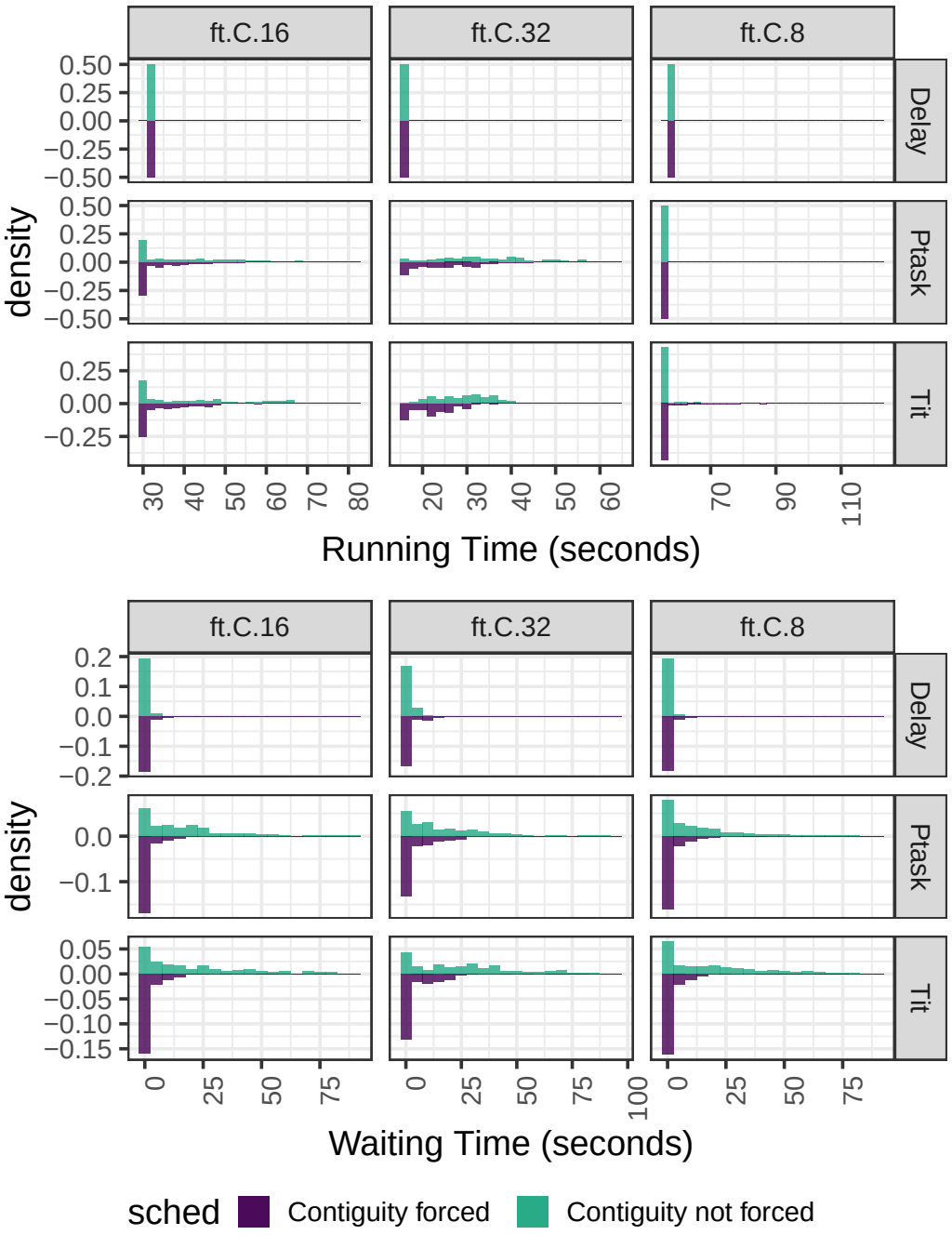


Figure 4.8.: Histogram of the waiting times and execution times for the different profiles. *ft.C.8* takes 8 machines for its execution, *ft.C.16* takes 16 machines, and *ft.C.32* takes 32 machines. The histogram on the left shows the distribution of the running time per profile, the x-axis is the running time in seconds. The histogram on the right shows the distribution of the waiting times. The color shows the different scheduling algorithm used, forced contiguity in purple (bottom) , not forced contiguity in blue (upper histogram).

4.4 Conclusion

In this chapter, we state that scheduling simulation should be enhanced to capture effects occurring in real platforms. For example, simulation taking into account the platform topology in our simulation by simulating the network interference of the jobs, the different host computation speed, the energy consumed by a platform, or the IO contention.

We present Batsim and develop why it is a good candidate to create scheduling simulations that can capture a wide range of effects depending on the jobs, or the platform itself thanks to the SimGrid simulation capabilities. Batsim provides a way to customize the models used to simulate the jobs of the simulation thanks to the profiles implemented in Batsim and leveraging SimGrid. The different profiles are, the profile delay, TiT, and *ptask*, and composed. Each profile has its own set of advantages and limitations, for instance, the profile delay — which is the more simple — can be used to achieve fast simulation, but lacks to capture effects such as network interference. On the other hand, the TiT profile can render interference effects at the price of higher simulation runtimes. Ptask is a good compromise, it shows better computation times while capturing different effects such as interferences.

To compare each model, we created a synthetic workload and create a variant of this workload for each of the three profiles, delay, TiT, and *ptask*. Next, we use two scheduling algorithms to compare the behavior of each model, one of the scheduling algorithm used can provide better job allocation. We observe that with the *ptask* and TiT models we observe a higher running time of the jobs — due to a bad allocation on the cluster, the job durations increase — generating higher waiting times.

TiT profiles present good simulation capabilities and has been the object of many scientific publications from the community to validate it. However, it is a heavy model and has high computation time hindering the scalability of scheduling simulations — thousand of jobs in less than one hour if possible. One alternative is to use the *ptask* model, which can capture different effects of a real platform (network interference, CPU speed) but that presents lower computation times. Unlike TiT, the *ptask* model has been featured by SimGrid without — to the best of our knowledge — validation.

While the delay execution profile is fast to compute, the main limitation of this profile is that it cannot render effects that appear on the computing platform, such as network interferences. From this experiment, we can see that the *ptask* model is at least 100 times faster than the TiT profiles while we can observe different effects such as network interferences.

While the model *ptask* is interesting as it proposes a way to simulates different applications it a high level of abstraction with reasonable computation time. However,

the *ptask* model the ability of the model to predict the behavior of an HPC application has not been evaluated yet. This lack of evaluation hinders the trust we should put in the results of the Batsim simulations using this model. Furthermore, the following elements should be evaluated with the *ptask* model.

- Can the *ptask* model accurately predicts the running time of the applications on a platform, under perfect conditions (application alone on a cluster), and with several applications on the platform (with resources conflicts)?
- It exists a wide range of different HPC applications. Which applications are best represented with a *ptask*?

5 Ptask Model Validation

5.1 Introduction

The previous chapter introduced the different job profiles available in Batsim. As seen in section 4.3, the *ptask* model is able to simulate different effects such as network interferences, while providing reasonable simulation times. Therefore the *ptask* model is a good candidate to be used as a job model for scheduling simulations. This chapter focuses on the validation of the *ptask* model.

The *ptask* model has been used to simulate workload composed of data-intensive applications, and simulate the inputs and the outputs of the cluster [Mer19]. However, to the best of our knowledge, this model has not been tested against real HPC applications, and its soundness is yet to prove. In this work, we are interested in evaluating the capacity of the *ptask* model to capture the behavior of HPC applications. This validation is important to increase the trust of our scheduling simulations. We compare the *ptask* model to an HPC application performing a matrix product in parallel on a real cluster, implemented with MPI.

This model simulates a parallel task (such as an HPC application) at a high level. It considers only the total of computations and communications made by the applications. A *ptask* performs work on a set of allocated resources of the simulated platform. The work can be either an amount of data that needs to transit between each host of the parallel task, or an amount of computation to perform on each host. Computations and communications are combined by the *ptask* model to be computed together.

Ptask supposes that the advancement of the task is homogeneous between all computations and communications and that the overall advancement of the task is limited by the bottleneck of any of the resources used by the *ptask*. The advancement of a *ptask* depends on three different parameters. The input given to the model is a number of computations and communications to execute on the platform. For instance, if another activity (a *ptask* for instance) uses the same link, it may impact the speed of the *ptask* and slows its progress forward. A detailed explanation of the *ptask* model is provided in chapter 4.

5.2 Experiment Methodology

Our methodology aims at validating if the behavior of a *ptask* under a congested network (on simulation) corresponds to the behavior of an HPC application with the same congested network (on a real platform). In other words, we aim to validate the interference model of the *ptask* model. To do this analysis, we execute an HPC application on a real platform onto which we generate controlled network traffic to create network interference. Then the same conditions are reproduced in simulation with a *ptask*.

In a production system, the application executions might be impacted by the network interferences induced by the other jobs, however, the jobs do not start (or end) necessarily at the same moment, and two different applications can have two different network usage patterns. To reproduce these effects, during the experiments, we generate different interference patterns by alternating between idle phases and creating network interferences (interference phases). However, unlike a production system, where the network interferences can be sporadic and unpredictable we choose to use a regular pattern to increase the reproducibility of the experiments.

As explained in section 4.2.2, the *ptask* model bundles the computation and the communication and smoothes the advancement of the task. Thus, we compare a *ptask* to an HPC application that shows a close behavior. We choose the application PDGEMM [AGZ94; FOH87] because it has a regular pattern for its whole execution. PDGEMM performs successive phases of communications (broadcast) and computations until it finishes. Additionally, PDGEMM is a real HPC application which is simple enough to be modeled properly with the *ptask*.

The next section details the PDGEMM application. Section 5.4 details the experimental setup to create controlled interferences and execute PDGEMM on two real clusters. Section 5.5 details the results of the real experiments on two real clusters. Section 5.6 describes how we simulate PDGEMM with the *ptask* model. Finally, sections 5.7 and 5.8 evaluate the *ptask* model by comparing the real executions with simulations. The remaining chapter opens a discussion (section 5.9) about the *ptask* model for scheduling simulation before concluding in section 5.10.

5.3 Parallel Matrix Multiplication (PDGEMM)

PDGEMM computes a product of two matrices (A and B of size $n \times n$) in parallel (on different processes) and stores the result into a third matrix (C). PDGEMM has two modes, blocking broadcast and nonblocking broadcasts:

- The first mode, detailed in section 5.3.1, performs sequentially: 1) blocking communications, and 2) then blocking computations. These two actions are repeated until the application completes.
- The second mode (detailed in appendix A.1.1), performs the communications (i.e., data transfers) in background of the computations. All the data transfers needed for the next computation phase occur during the current computation phase — except for the first data transfer required for the first computation which is necessarily blocking.

The last parameter is the number of subdivisions, it adds the possibility to create subdivisions of the matrices block managed by each process. This parameter is detailed in section 5.3.2.

5.3.1 PDGEMM Algorithm

The PDGEMM algorithm is depicted in algorithm 3. Each process holds a different block of the matrices A and B (none of the processes disposes of the whole matrix). The resulting matrix C also is distributed among the different processes. For P processes, the matrices are divided into $\sqrt{P}$ blocks of size n^2/P. The process p holds the blocks situated on the block line $p/\sqrt{P}$ and the block column $p \mod \sqrt{P}$ of the matrix A, B, and C.

Each process belongs to three groups. The first group corresponds to the id of the line of its block in the matrix A, and the second group corresponds to the id of the column of its block in matrix B. Separating the processes into groups enables to use of MPI's collectives on the member of the groups, for instance, one can perform a broadcast only between the group members. Inside each group, the processes are identified by a unique id. Process number p (the world group) belongs to the groups $p/\sqrt{P}$ for the lines and to the group $p \mod \sqrt{P}$ for the columns.

The algorithm performs $\sqrt{P}$ iterations, in which each process p computes a partial sum of its result $block_C^p$. At the iteration k, each process with the id k in its group line broadcasts its $block_A$, the process with the id k in its group column broadcasts its matrix $block_B$. Figure 5.1 depicts the communications made at iteration k. Each process then computes the partial sum of the block matrix C with the broadcasted blocks. At the end of all iterations, the matrix C blocks correspond to the product between A and B.

Algorithm 3 Parallel Matrix Multiplication (PDGEMM). Each process of the application executes this function.

procedure PDGEMM(id_ranks,$world\_size$,$matrix\_block_A$, $matrix\_block_B$)
  $q \leftarrow \sqrt{world\_size}$
  $row\_group \leftarrow id\_rank/q$ ▷ Row group of the process
  $col\_group \leftarrow id\_rank \mod q$ ▷ Column group of the process
  $group\_row\_id \leftarrow$ get_id(row_group) ▷ Process unique id in its row group
  $group\_col\_id \leftarrow$ get_id(col_group) ▷ Process unique id in its col group
  $res_C \leftarrow [block\_length \times block\_length]$
  $buffer_A \leftarrow [block\_length \times block\_length]$
  $buffer_B \leftarrow [block\_length \times block\_length]$
  for $i \leftarrow 0, i < \sqrt{world\_size}, i++$ **do**
    $root\_col \leftarrow k \mod q$; ▷ Determines the broadcast source for $block_A$
    $root\_row \leftarrow k \mod q$; ▷ Determines the broadcast source for $block_B$
    if $group\_row\_id == root\_row$ **then**
      broadcast_on_lines($matrix\_block_A$, $source = True$)
    else
      broadcast_on_lines($buffer_A$, $source = False$)
    end if
    if $group\_col\_id == root\_col$ **then**
      broadcast_on_cols($matrix\_block_B$, $source = True$)
    else
      broadcast_on_cols($buffer_B$, $source = False$)
    end if
    if $group\_row\_id == root\_col \ \& \ group\_col\_id = root\_row$ **then**
      $res_C \leftarrow res_C + matrix\_block_A * matrix\_block_B$
    else if $group\_col\_id == root\_col$ **then**
      $res_C \leftarrow res_C + buffer_A * matrix\_block_B$
    else if $group\_row\_id == root\_row$ **then**
      $res_C \leftarrow res_C + matrix\_block_A * buffer_B$
    else
      $res_C \leftarrow res_C + buffer_A * buffer_B$
    end if
  end for
end procedure

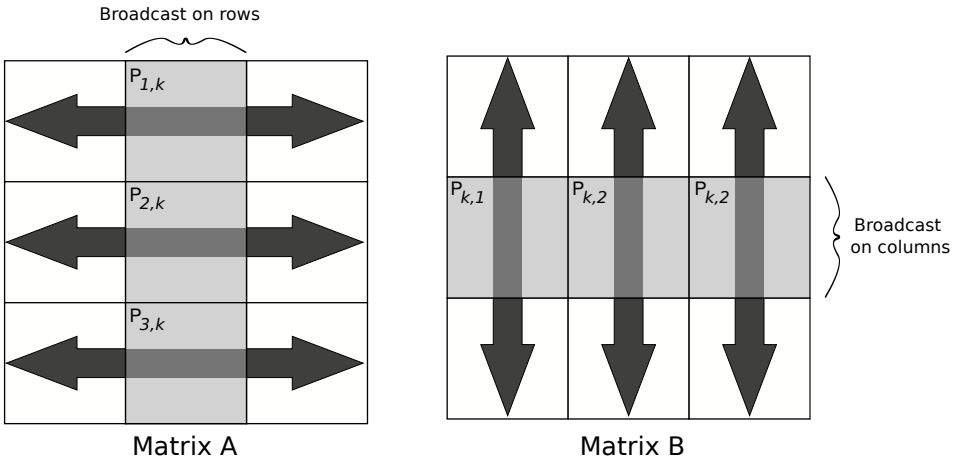


Figure 5.1.: At iteration k, the processes on the k^{th} line broadcast their block to the other processes of their line for the matrix A. Once the fist broadcast is done, the processes on the k^{th} column broadcast their block to the other process of their column.

5.3.2 Matrix Block Subdivision

The last modification of the PDGEMM algorithm is to add the possibility to divide the block of the matrix managed by each process into several sub-blocks.

Creating sub-blocks has the advantage of reducing the memory amount consumed by each process because the buffers used for receiving the matrices A and B only need to store the sub-block instead of the whole block. Increasing the number of matrix blocks sub-division increases the total number of iterations, but decreases the number of size of the computation and communications done at each iteration.

However, the application will perform more — but smaller — iterations. The number of iterations is $\sqrt{P} * S$, with P the number of the process involved and S the number of subdivisions of a block. At each iteration, the processes send (or receive) a subpart of the whole block and perform the multiplication on this subpart.

5.3.3 PDGEMM Resources Consumption Behavior

In summary, PDGEMM is a regular application executing the same actions until completion. Two modes are available for the communications: Blocking broadcasts and nonblocking broadcasts. The last parameter enables to change the number of subdivisions of the block managed by each process.

All these parameters affect the resource consumption of PDGEMM during its execution. For instance, with the number of sub-divisions set to 1 with nonblocking broadcast PDGEMM application have a chaotic network consumption because of the large data sizes send over the network. Thus, such behavior is difficult to predict with a *ptask* which models a homogeneous behavior

5.4 Real Experimental Setup

This section presents the experimental setup necessary to study the PDGEMM on two real clusters. First we present the experimental setup created for the experiments. This setup enables to control the network, generate network interferences during the execution of PDGEMM and to gather data about the execution.

5.4.1 Platform and Nodes Configuration

To run the application, we have access to the Grid'5000 infrastructure. However, Grid'5000 does not provide a platform with a network that can easily be saturated, which is required if we want to observe network interferences. To cope with this problem, we create an experimental setup on two available clusters from Grid'5000: *Paravance*, and *Grisou*. The setup is common to the two platforms. In this setup, we force communications between two sets of nodes to pass through a unique node. This setup enables to study our application under a congested network. The different processes of the MPI application are distributed among the two sets of nodes. During the execution, we create network traffic from one node set to the other to saturate the routing node.

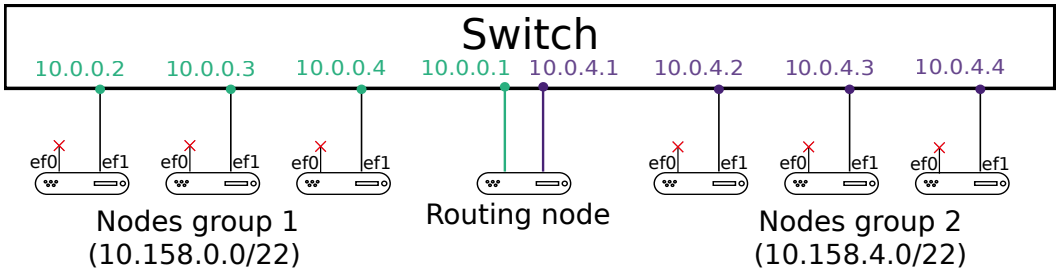
This setup requires that all the nodes are located under the same network switch. The nodes are divided into three groups of machines, the first group is composed of a single node that will act as a router. The remaining nodes are allocated in one of the two other groups, that will be used as computing nodes for the MPI application. We create one subnetwork per computing group, and we attribute an address to every node in the group, the routing node has two network interfaces, on per group. Figure 5.3 shows an example of this configuration with 6 nodes and two subnetworks.

The last configuration is to force the communications from one subnetwork to another to pass through the routing node. This is done by configuring the routing table of each node of the two groups.

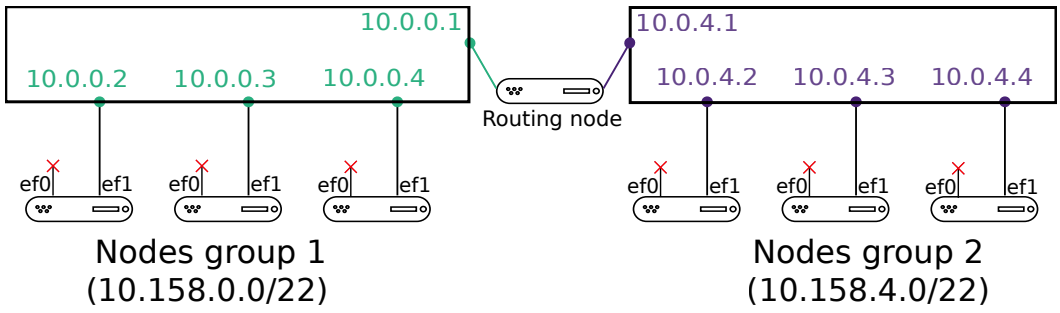
5.4.2 PDGEMM and MPI Configuration

One switch has 35 connected nodes. Each node of *Paravance* is equipped with two Intel Xeon E5-2630 v3 (Haswell, 2.40GHz, 8 cores).

256 cores configuration The PDGEMM application is executed on the set of computing nodes (split into the two sub-networks as described in the previous section). With 35 nodes, we use 8 nodes per subnetwork for a total of 16 nodes for the PDGEMM application. On each node, we run 16 processes, 1 process per physical core for a total of 256 cores for the whole application.



(a) Physical view of the installation. All 6 nodes are connected to the same switch. Nodes are distributed across two subnetworks: `10.0.0.0/22` and `10.0.4.0/22`. The `ef0` interfaces of the computing nodes are disabled. Each link from a node to the switch has a bandwidth of 1.25 Gigabytes/s



(b) Logical view of the installation. The nodes in the same subnetworks communicates using the switch, while the communication between the subnetwork are forced to pass through the routing node. The routing node has two addresses, each into a different subnetwork so it can join nodes from both subnetwork. Activating ip forwarding enable the routing nodes to transfer packets from one subnetwork to the other.

Figure 5.2.: Final configuration for the experiment. Figure 5.2a depicts the physical setup of the nodes, while the fig. 5.2b shows the logical view we achieve.

The matrix size is $6.4 * 10^8$ (with $n = 8 * 10^4$). Every process of the application has in its memory 5 matrix blocks ($block_b$, $block_A$, $block_C$, and two buffers for the communications). Each process uses 1GB of memory, for a total of 16GB per computing nodes.

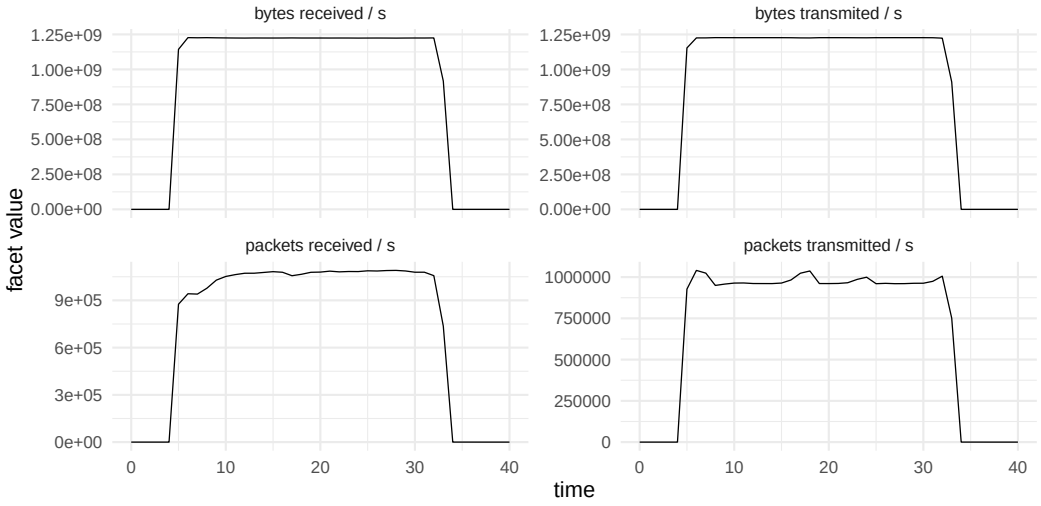
The last parameter we need to tune is the placement of the matrix blocks over the cluster. For our use case, we want to measure the impact of network interference on our application, so we choose to spread the matrix blocks over the nodes with a random policy, in this way we create a placement that should be bad enough to saturate the routing node.

5.4.3 Controlled Interferences

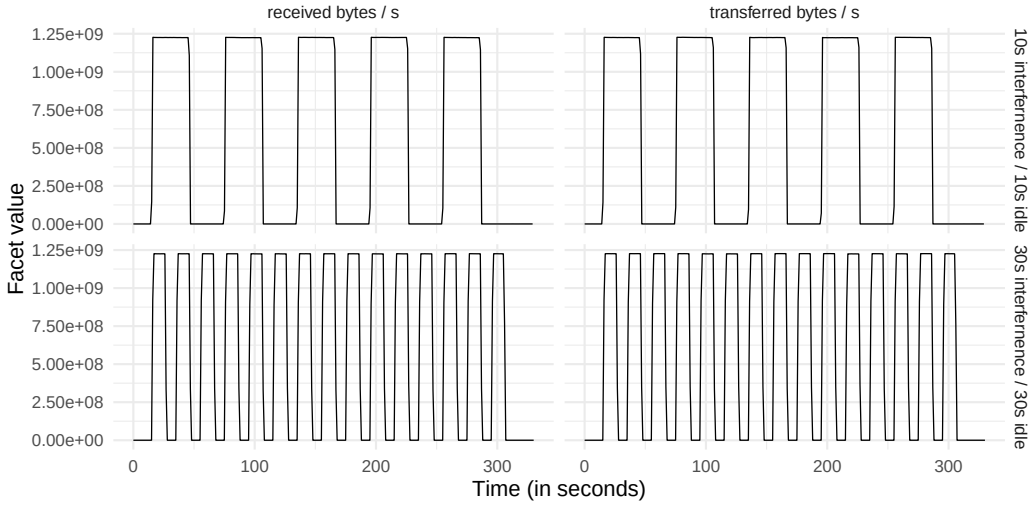
To create interferences, we use a load generator software named *Tcpkali*[1]. *Tcpkali* is an application that implements a client and a server that is able to generate a load on a network.

[1] https://github.com/satori-com/tcpkali

The load is generated by the client by repeatedly sending the same message during the activity. The server can be configured to have the same behavior. When a client is connected to a server, it can be configured to create a number of TCP connections and will create as many connections between the server and the client. *Tcpkali* is multi-threaded and uses all the threads of the nodes to increase the generated load. With a single client and server, *Tcpkali* is able to reach the max bandwidth of the network link (fig. 5.3a).



(a) Traffic monitored on the routing node with one client and one server *Tcpkali* (100 tcp connections) for 30 seconds. The client is executed on a node of the first subnetwork, and the client on a node of the other subnetwork. All the traffic passes through the routing node. *Tcpkali* is able to reach max link capacity ($1.25\ Gigabytes/s$) during the whole execution.



(b) Monitored network activity on the routing node with two different interference patterns. The upper line of the grid shows a pattern alternating between 30s of idle times and 30s of interference. The lower line shows the same pattern, but with 10 seconds.

Figure 5.3.: *Tcpkali* evaluation (upper figure). Interference pattern examples (lower figure).

During the experiment, we create different interference patterns by generating periods in which we alternate between idle time and network interference. Figure 5.3b shows the network traffic on the routing nodes under two interference patterns. One server is started on a node of the group 1 and remains active the whole experiment, when there is not client the server waits. The pattern is created by starting and stopping repeatedly a client on a node of group 2. The client first connects to the server and generates load for a period of time, when the time is over the client is then stopped leaving the server is a state where it is waiting for new connections. One script, executed on group 2 is in charge of starting, and stopping the client, and during idle phases, sleeping.

5.4.4 Monitoring

On each node of the experiment we use MojitO/S[2], an open-source monitoring software which can poll a wide range of metrics from different sources of the node it is executed on. In our case, we use MojitO/S to poll information about the network interfaces on the experiment's nodes.

5.5 Results and Data Analysis of the Real Executions

This section details the results of the experiment on *Paravance* and *Grisou*. In the first place, we present a detailed view of the *Paravance* experiments to explicit the resource consumption of PDGEMM. Secondly, we show the results obtained for the *Grisou* platform. Finally, we illustrate the difference between the two clusters.

5.5.1 Results Analysis for *Paravance*

Figure 5.4 plots the running times for each configuration on *Paravance*, note that each configuration is executed 5 times on G5K to cope with system variability. In the first place, we notice the running time of the application is sensitive to network interference, and all configurations have the same behavior — more interference means higher running time. On the other hand, we observe that jumping from no subdivisions to 50 block subdivisions (block subdivision is detailed in section 5.3.2), the nonblocking broadcasts become less efficient and the blocking broadcasts bring better running time.

Table 5.1 presents the mean runtime for each configuration (along with the standard deviation). Both the figure and the table show that the application execution is slowed with interferences. From the results table, we can observe that constant

[2] https://sourcesup.renater.fr/www/mojitos

interference slows the execution in the same way for every broadcast and subdivision configuration — around 15 %, the max is 16 % and the min 14 %.

Broadcast	Subdivision	Interference	Mean runtime(s)	Standard dev	Increase %
Blocking	No subdivision	No interference	450.42	0.76	
		15s interference / 45s idle	469.19	0.41	4.17
		15s interference / 15s idle	484.57	1.37	7.58
		30s interference / 30s idle	480.31	1.05	6.64
		45s interference / 15s idle	492.76	2.35	9.40
		Constant interference	513.93	0.93	14.10
	50 subdivision	No interference	346.08	1.33	
		15s interference / 45s idle	360.37	0.64	4.13
		15s interference / 15s idle	373.85	0.98	8.02
		30s interference / 30s idle	374.13	1.43	8.11
		45s interference / 15s idle	388.59	1.23	12.28
		Constant interference	402.69	1.20	16.36
Non blocking	No subdivision	No interference	384.06	4.11	
		15s interference / 45s idle	395.39	3.04	2.95
		15s interference / 15s idle	409.19	5.16	6.54
		30s interference / 30s idle	407.61	2.37	6.13
		45s interference / 15s idle	421.66	4.12	9.79
		Constant interference	434.65	2.10	13.17
	50 subdivision	No interference	374.29	1.59	
		15s interference / 45s idle	389.57	0.84	4.08
		15s interference / 15s idle	403.58	1.63	7.83
		30s interference / 30s idle	403.35	0.97	7.76
		45s interference / 15s idle	416.79	1.90	11.35
		Constant interference	430.49	1.10	15.01

Table 5.1. Mean runtime of all PDGEMM executions for each configuration (number of subdivisions, interference patterns and broadcast types) on ***Paravance***. The last column is the percentage of increase compared to the execution of the same category without interference.

Between blocking and nonblocking broadcasts, we observe that the subdivisions impact more the blocking broadcast (up to a difference of 2 %), more subdivisions increase the slowdown. Whereas for nonblocking broadcast the number of divisions has a lower impact. Moreover, the configuration with blocking broadcast and no subdivision is globally less impacted by the interferences than the other configurations. This is because no subdivisions imply longer phases of calculation and communication, and during a computation phase, no transfer is performed. Figure 5.5 shows the network traffic recorded for the blocking broadcast, as each configuration has been run 5 times, the figure only shows one of the five runs for each configuration (interference pattern and subdivisions). The upper figure shows that periodically, the network is not used by the application, the periods of inactivity correspond to the computation phases. On the other hand, the lower figure shows that when subdividing the matrix the application performs more (but smaller) phases, that homogenize the network usage. Hence, with 50 subdivisions the application is sensitive to the network interference during its whole execution.

As figure 5.5 for blocking broadcast, figure 5.6 shows the network traffic for nonblocking broadcasts. One can observe that, for both blocking and nonblocking broadcasts, using a number of 50 subdivisions increases the homogeneity of the net-

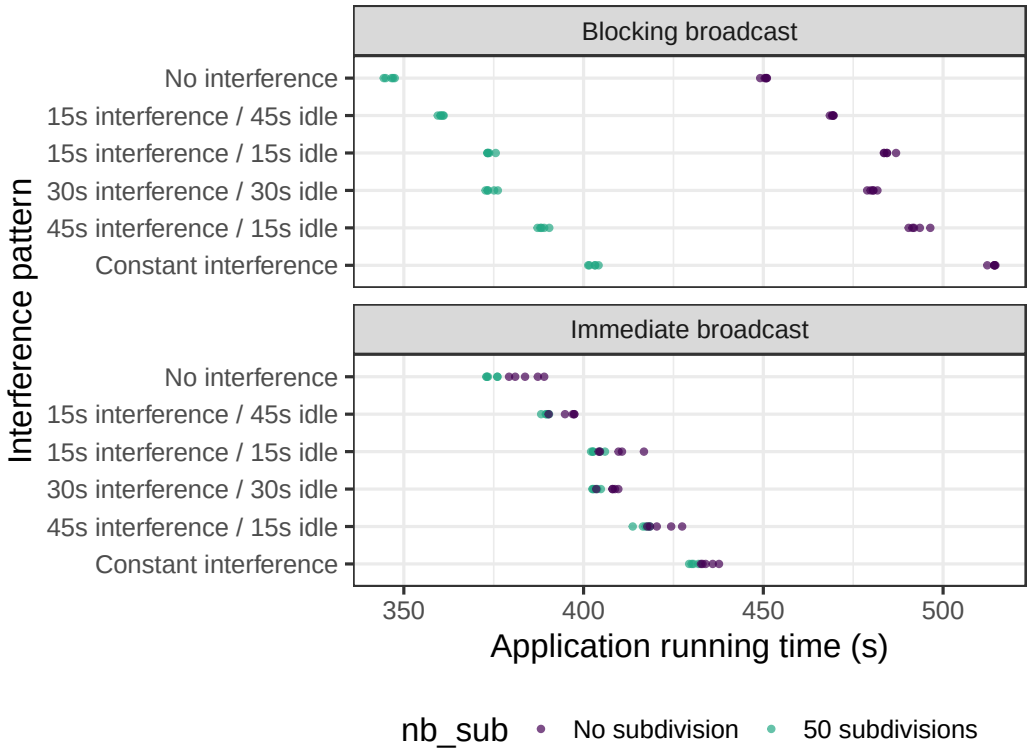
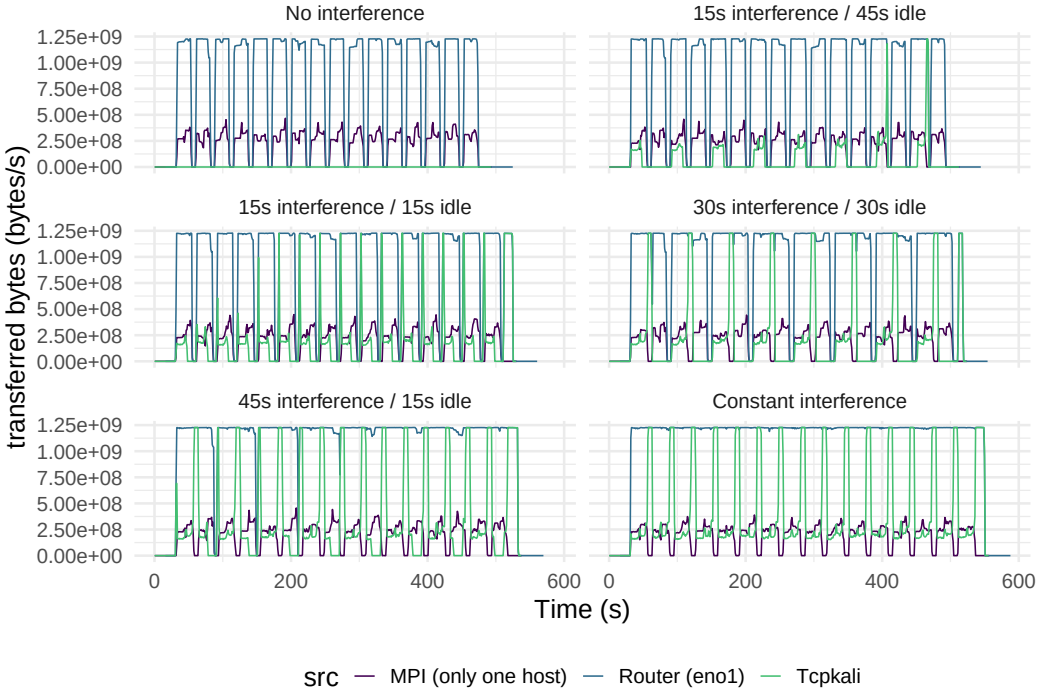


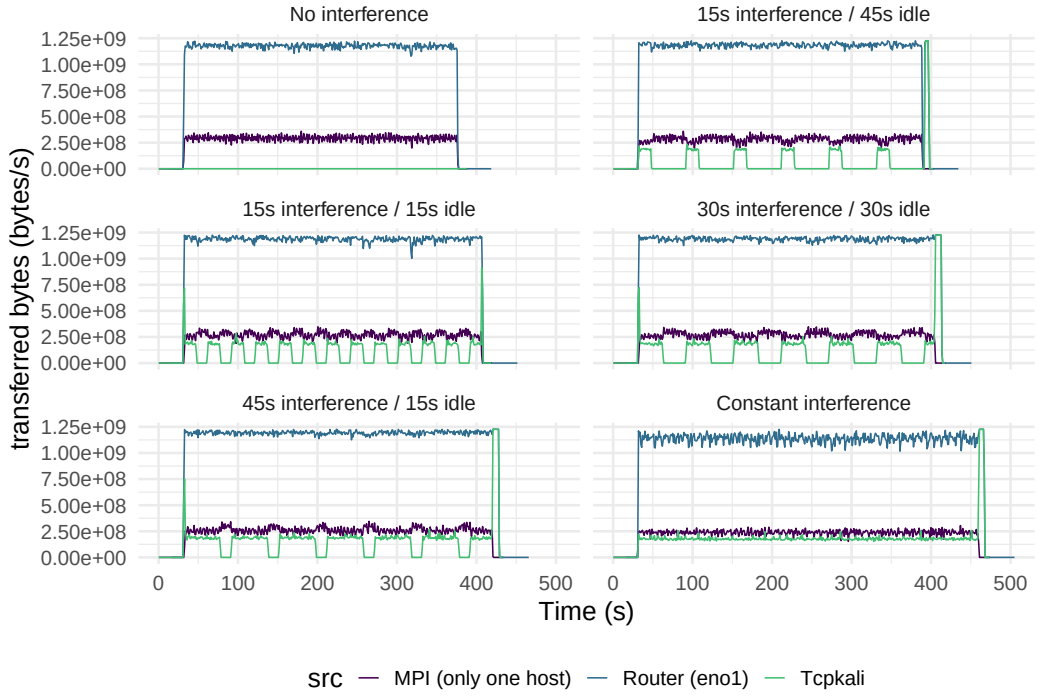
Figure 5.4.: Mean runtime of PDGEMM under different interference patterns and broadcasts (blocking or nonblocking).

work traffic, leading to more stable executions. In the case of nonblocking broadcast the network is always used but chaotically (this can also explain the higher standard deviation visible in table 5.1).

Blocking and nonblocking broadcast using the configuration with 50 subdivisions presents a better fit for the *ptask* model. With these configurations, the application network usage is homogeneous, which corresponds to the *ptask* model. Indeed, the *ptask* model assumes that the application's progress during the execution is homogeneous.

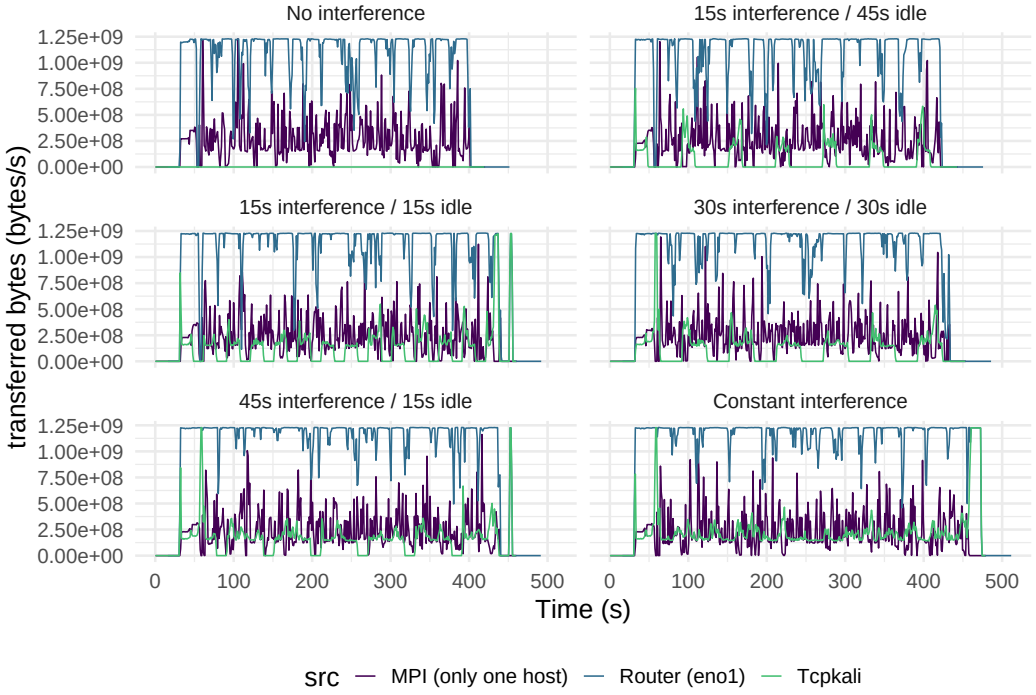


(a) Network traffic monitored with **blocking broadcast** and **no subdivisions**.

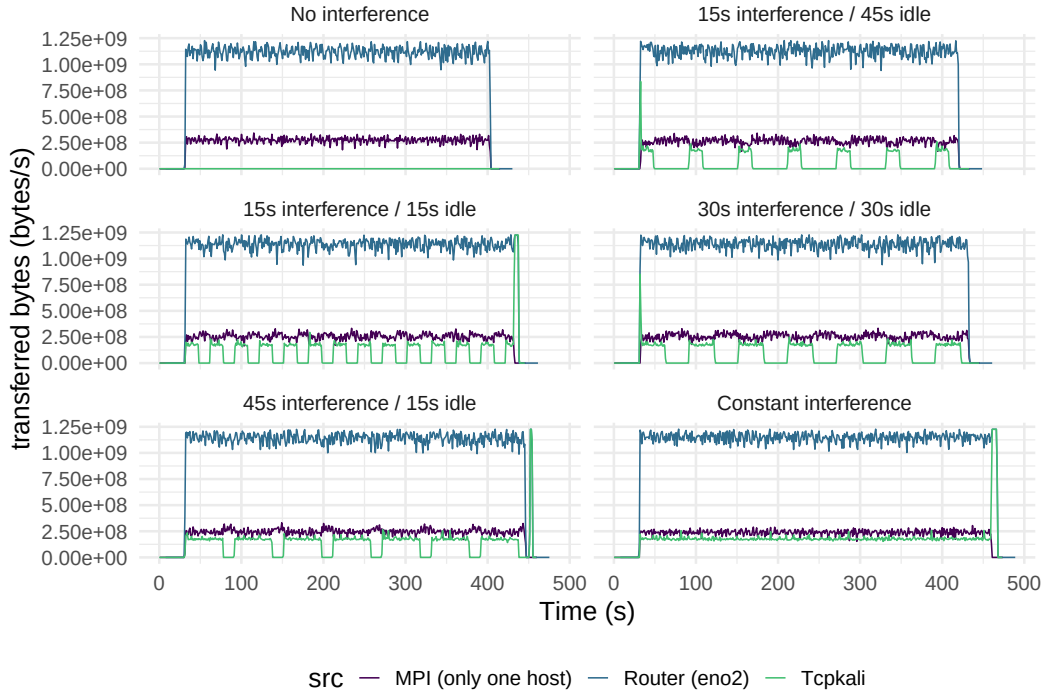


(b) Network traffic monitored with **blocking broadcast** with **50 subdivisions**.

Figure 5.5.: ***Paravance.*** Network traffic monitored on one MPI host, and on *tcpkali* server's host for **blocking broadcast**. The upper figure plots one instance for each interference pattern without subdivision, whereas the lower figure plots it with 50 subdivisions.



(a) Network traffic monitored with **nonblocking broadcast** with **no subdivisions**.



(b) Network traffic monitored with **nonblocking broadcast** with **50 subdivisions**.

Figure 5.6.: ***Paravance.*** Network traffic monitored on one MPI host, and on *tcpkali* server's host for **nonblocking broadcast**. The upper figure plots one instance for each interference pattern without subdivision, whereas the lower figure plots it with 50 subdivisions.

5.5.2 Results Analysis for *Grisou*

Broadcast	Subdivision	Interference	Mean runtime(s)	Standard dev	Increase %
Blocking	No subdivision	No interference	452.28	0.99	
		15s interference / 45s idle	583.11	0.49	28.93
		15s interference / 15s idle	806.33	1.72	78.28
		30s interference / 30s idle	820.04	2.23	81.31
		45s interference / 15s idle	1243.07	18.45	174.85
		Constant interference	Reached Timeout		
	50 subdivision	No interference	349.30	0.86	
		15s interference / 45s idle	442.71	1.00	26.74
		15s interference / 15s idle	568.50	0.60	62.76
		30s interference / 30s idle	574.34	1.19	64.43
		45s interference / 15s idle	796.79	3.76	128.11
		Constant interference	1415.59	28.52	305.27
Non blocking	No subdivision	No interference	372.83	2.34	
		15s interference / 45s idle	475.88	4.67	27.64
		15s interference / 15s idle	624.58	1.96	67.52
		30s interference / 30s idle	599.13	1.97	60.70
		45s interference / 15s idle	860.33	25.32	130.75
		Constant interference	1490.47	21.72	299.77
	50 subdivision	No interference	379.14	1.29	
		15s interference / 45s idle	469.59	0.92	23.86
		15s interference / 15s idle	599.06	0.59	58.01
		30s interference / 30s idle	595.54	0.35	57.08
		45s interference / 15s idle	832.91	1.80	119.69
		Constant interference	1455.86	17.55	284.00

Table 5.2. Mean runtime of all PDGEMM executions for each configurations (number of subdivisions, interference patterns and broadcast types) on ***Grisou***. The last column is the percentage of increase compared to the execution of the same category without interference.

The experiment on *Grisou* shows the same behavior as the experiment on *Paravance*. However, in *Grisou* the PDGEMM application is more impacted by the interferences.

Table 5.1 presents the mean runtime for each configurations on *Grisou* (along with the standard deviation). As for the experiment on *Paravance*, the table shows that the application execution is slowed when there is interferences. The major difference is that the application is slowed at maximum by $305\ \%$ with Blocking broadcasts and 50 subdivisions (all instance of the blocking configurations with no subdivisions have been killed because they reached the timeout, so the maximum is probably more), and the minimum is $283\ \%$ for non-blocking broadcasts with 50 subdivisions.

5.5.3 Difference Between *Grisou* and *Paravance*

Figure 5.7 shows the monitoring traces of *Grisou* and *Paravance* side by side for three configurations. Looking at the configuration without interferences (upper plots), we notice that both executions are similar: The execution took in average $349.30\ s$ (table 5.2) on *Grisou* and $346.08\ s$ (table 5.1) on *Paravance*. It also appears that the network activity generated by MPI is similar.

The main difference can be observed when looking at the phase with interferences. The lower plots show the monitoring traces with the configuration doing constant interferences. The green line depicts the *Tcpkali* network activity, while the (dark-)purple shows the MPI activity. In *Paravance*, the PDGEMM application is less impacted by the interferences on *Grisou*, up to $16\ \%$ for *Paravance* against $300\ \%$ on *Grisou*. During interferences phases the MPI application doesn't use the same bandwidth on *Grisou* and *Paravance*: *Tcpkali* manages to get more bandwidth on *Grisou*. This can explain why the application on *Paravance* is less sensitive to network interferences generated with *Tcpkali*.

Paravance and *Grisou* are similar clusters, they both features *Dell PowerEdge R630* with two *Intel Xeon E5-2630 v3 (Haswell, 2.40GHz, 8 cores)* per nodes. The main difference is the network switch connecting the nodes of each cluster, the *Paravance* nodes are connected with a *Cisco Nexus 9508* and the *Grisou* nodes are connected with a *Nexus 56128P*. Both switch might have a different behavior or they might have different configurations.

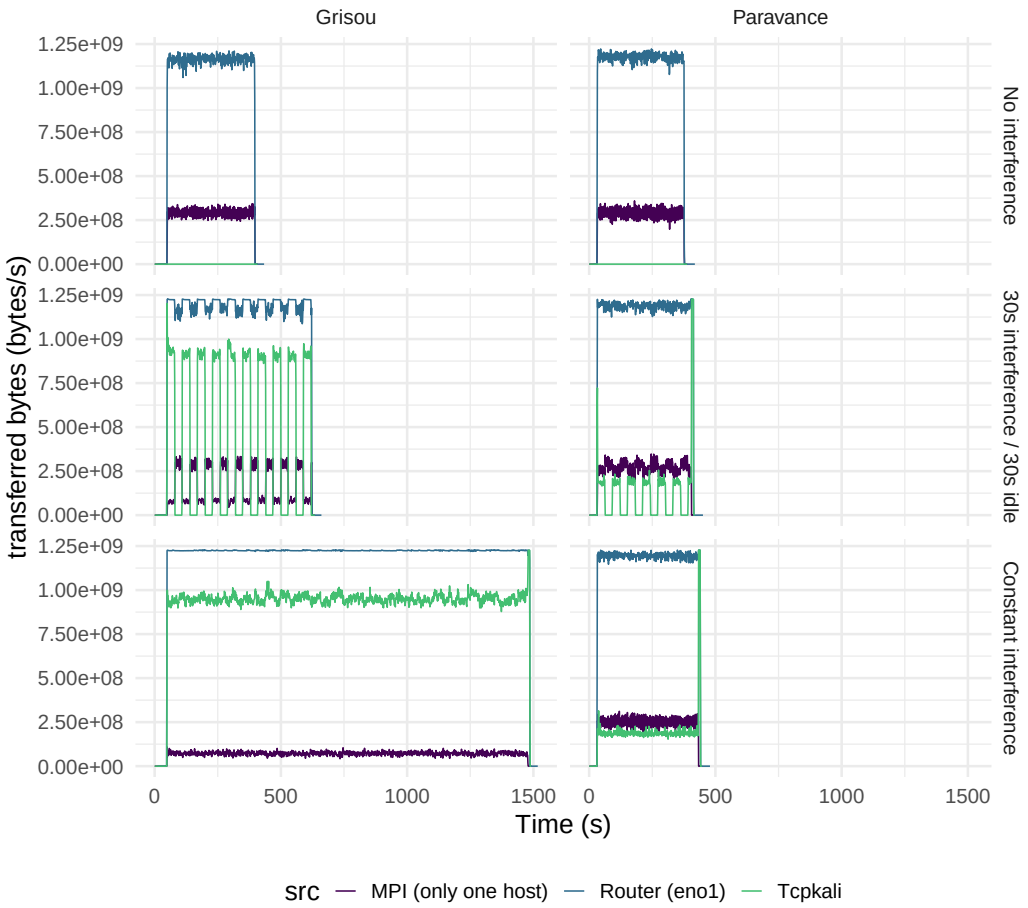


Figure 5.7.: Comparison of monitoring traces of *Grisou* and *Paravance*. The plots show the configuration of PDGEMM with blocking broadcasts and 50 subdivisions. It appears that without interferences, *Paravance* and *Grisou* have similar executions, the network activities are alike and have the same execution times. However, under constant interferences (bottom line), the bandwidth of PDGEMM is significantly lower on *Paravance* than on *Grisou* and have a different execution time.

5.6 PDGEMM in Simulation

The second part of the experiment consists of creating a *ptask* corresponding to PDGEMM. To evaluate the model's capacity to predict the application running time under interferences the *ptask* is executed in the same simulated conditions: the same platform and the same controlled network interferences. That is to say, we need to create the SimGrid platform corresponding to the setup depicted in figure 5.2b, to simulate the same interferences, and a *ptask* representation of PDGEMM.

5.6.1 SimGrid Platform, Calibration and Interference

The created platform uses the SimGrid *xml* API. The platform needs several attributes, the hosts performing the calculation, and how they are connected (SimGrid is detailed in section 4.2.1). In the simulation, we create the platform depicted in 5.2b, which consists of two distinct homogeneous clusters connected to two switches (one per cluster) and one link connecting the two clusters (our routing node).

One important part of SimGrid simulations is the platform calibration, the speeds of the hosts and links need to be carefully studied to create a SimGrid platform accurately representing the reality. To calibrate the network links, we use the SimGrid default TCP policy, which has already been evaluated several times[Vel+13; Béd+13]. However, to accurately represent the computation speed of the hosts we use the calibration made by Tom Cornebize to accurately predict HPL running time at large scale [CLH19]. This is possible because, both the application HPL and PDGEMM use the *Dgemm* function from the *openblas*[3] library. The *Dgemm* function perform the following operation:

$$C := \alpha * A * B + \beta * C$$

For matrices C, A and B, and the two scalars α and β. *Dgemm* is the function used to do the matrix product in the actual implementation of algorithm 3.

The whole experiment is created with a unique SimGrid program, that both execute a *ptask* and generates the same interference used in the real platform. To simulate the interferences we use two methods, the first method is to create a *ptask* with no computation that will transfer data between one node of the first group and another node in the second group. The second method is to use the SimGrid API: *s4u* enables to program actors to take part in the simulation. To mimic the behavior of *Tcpkali*, we create two SimGrid actors that will periodically initiate data transfer to match with the different configurations used in the real platform.

[3] https://www.op10nblas.net/

5.6.2 *Ptask* Generation

One difficulty of the *ptask* model is to be able to generate a *ptask* corresponding to the real application. As a reminder, to compute a *ptask* SimGrid needs two information: the communication matrix and a vector of computation amount to complete.

The number of elements in the vector of computation corresponds to the number of SimGrid actors in the simulation. In our case, we use one actor per MPI rank. The computation amount, in the case of PDGEMM, is easy to obtain as we consider that one matrix product generates approximatively $m * n * k$ number of operation, for matrix $U_{m,k}$ and matrix $V_{k,n}$, m, n, and k are the dimension of the matrices U and V, the first letter being the number of lines, and the second the number of columns. On each MPI rank, we have one matrix block of size $n * n$ (with in our case $n = 8 * 10^4$ as described in 5.4.2). Thus each rank has locally a block of size $b * b$ with $b = n/\sqrt(P)$ (P is the number of ranks). Thus each rank will execute b^3 operations $\sqrt(P)$ times (loop at line 10 of algorithm 3).

The second parameter is the number of block subdivisions, in our experiment we use a subdivision of 50 sub-block. However, the total amount of computation made by each rank is not impacted by the number of divisions of the blocks. The number of iteration is consequently increased.

Obtaining the communication matrix of the application is more tedious for two main reasons. The first reason is that the exact communication depends on the underlying broadcast algorithm used during the application. The second reason is that we lack monitoring tools permitting us to know the point to point communication destinations. MojitO/S only provides the data that a node receives and sends. To cope with this issue, as in section 4.3.2, we use SMPI with the tracing options enabled in order to get the point-to-point communication. This enables to compute the exact communication matrix. Next, we force the broadcast algorithm to the *binary tree* algorithm in simulation, and in reality, both MPI and SMPI enable to tune this parameter at execution time.

It is worth mentioning that the *ptask* model has not the possibility to model the different parameters of PDGEMM. Thus, the different combinations of PDGEMM, number of subdivisions, and broadcast types are represented with the same *ptasks*. The only parameters impacting the *ptask* generation are the number of processes used and the size of the input matrices.

Ptask	Interference	runtime	increase %
Tcpkali is *s4u::Actor*	No interference	312.00	
	15s interference / 45s idle	382.00	22.44
	15s interference / 15s idle	472.00	51.28
	30s interference / 30s idle	472.00	51.28
	45s interference / 15s idle	635.00	103.53
	Constant interference	935.00	199.68
Tcpkali is *ptask*	No interference	312.00	
	15s interference / 45s idle	382.00	22.44
	15s interference / 15s idle	472.00	51.28
	30s interference / 30s idle	472.00	51.28
	45s interference / 15s idle	635.00	103.53
	Constant interference	935.00	199.68

Table 5.3. Predicted runtime for the *ptask* model under each interference pattern, and two methods for simulating *tcpkali*. The obtained results are identical for the two methods: Using a *ptask* or the SimGrid's *s4u* API doesn't impact the simulation's results. The last column is the percentage of increase compared to the execution of the same category without interference.

5.7 Comparison between the *Ptask* Model and Reality

In this section, we compare the results obtained with the *ptask* model in simulation to the real execution of PDGEMM. Table 5.3 shows the prediction obtained with the *ptask* model. The first thing to notice is that without interference the model predicts a runtime of 312 seconds, as a reminder the mean runtime of PDGEMM application' execution without interference was, for 50 subdivisions, $346\ s$ (error of $10\ \%$) for blocking broadcasts, and $373\ s$ (error of $18\ \%$) for nonblocking broadcasts on *Paravance*. On *Grisou* the mean runtime is $349\ s$ and $379\ s$ respectively.

Figure 5.8 compares the *ptask* prediction with the executions on *Paravance* and *Grisou*. As already stated, the runtime prediction without interferences remains close, however, *ptask* fails to accurately predict the runtime with interferences. The predictions are less accurate when the number of interferences increases, to the final point with constant interferences. However, it appears that the model is a better fit for *Grisou* than *Paravance*. For PDGEMM, the running time is increased at max $16\ \%$ for *Paravance*, and $300\ \%$ for *Grisou*. The *ptask* model predicted time increases to $199.68\ \%$.

We observe that the *ptask* model reacts to interferences in the same way as PDGEMM does, the application execution is slowed during interference phases, and longer interference time (in totality) means slower executions, therefore *10s interference / 10s idle* and *30s interference / 30s idle* have the same running time for *ptask*.

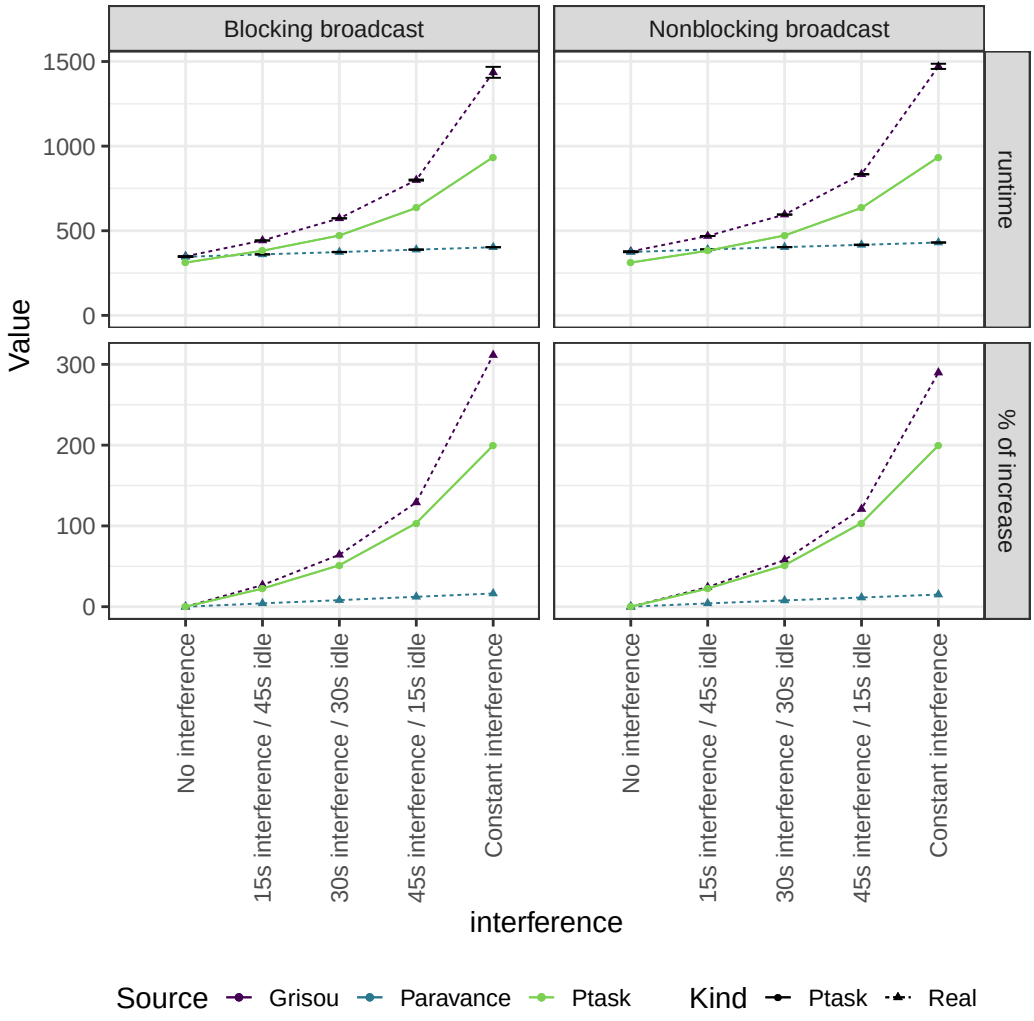


Figure 5.8.: Comparison of the *ptask* with *Paravance* and *Grisou* for configurations with 50 subdivisions. The error bar are only present for *Paravance* and *Grisou*, they show the confidence intervals (95 %) of the mean runtime. The *15s interference / 15s idle* pattern has been removed to increase readability.

5.8 Interference Analysis

The current experiment's results show that the *ptask* model keeps the tendencies of PDGEMM while failing to accurately predict the running time with interferences. More specifically, the *ptask* model tends to either under predict the runtime for *Grisou*, or over predict the runtime for *Paravance*. However, the *ptask* seems to adopt the same behavior as the PDGEMM application when we simulate a *ptask* with interferences.

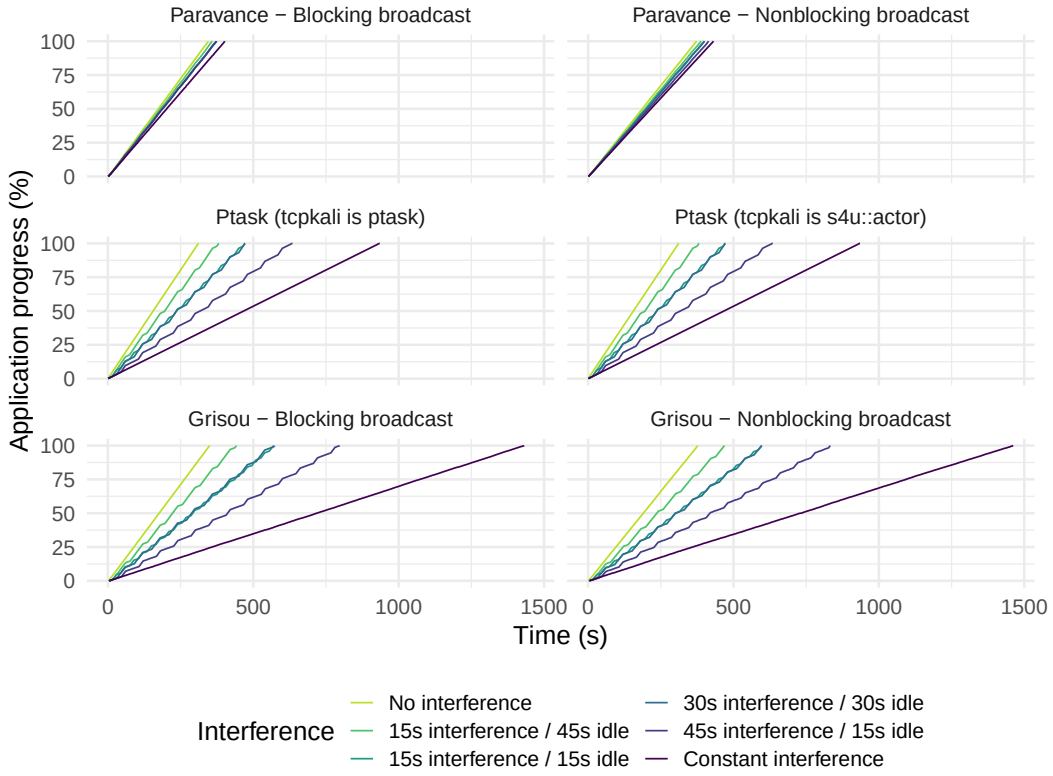


Figure 5.9.: Progress of PDGEMM and *ptask* over the simulated time until its completion ($progress = 100$).

Figure 5.9 depicts the progression through time of PDGEMM and *ptasks*. For PDGEMM, the progress lines are obtained by logging during PDGEMM execution the time at which the application enters a loop and the loop's index for each process. The provided information is aggregated, and the percentage corresponds to the current number of loop done divided by the total of loops required. For *ptask*, the progression lines are obtained by periodically logging during the simulation the remaining of work to perform before the *ptask*'s completion. This figure gives a vision of the impact of the interference on the progress of PDGEMM and *ptasks*. We can note behaviors of PDGEMM common to *Grisou*, *Paravance*, and the *ptask* model:

1. The application has two progress rates: The progress rates during interference phases, and the progress rate corresponding to the phases without interferences. This phenomenon is more visible, for instance, on *Grisou*, and for the *ptask* model.

2. The two progress rates remain identical regardless of the time of the interference made. This suggests that PDGEMM is sensitive to the total quantity of interference (in time) independently of when the interferences occur. This is illustrated with the configurations *15s interference / 15s idle* and *30s interference / 30s idle* that have a close running times (table 5.1 and table 5.2).

From the different observations of the experiments, we deduce a theoretical intermediate model to predict the runtime of both the *ptasks* and the PDGEMM application with the different interference patterns.

5.8.1 Theoretical Interference Model

The model works as follows. For a time period T (in our experiments $T = 60\ s$) the application progress for this time period can be calculated by the formula:

$$T(\alpha * c_1 + (1 - \alpha) * c_2) \tag{5.1}$$

The parameters c_1 and c_2 correspond to the progress rate of the application during the different phases: c_1 corresponds to the progress rate with interferences (the slow rate) and c_2 correspond to the progress rate without interference (the fast rate). The parameter α corresponds to the percentage of the time period with interferences (the slow rate) and $1 - \alpha$ corresponds to the percentage of the time period (T) that doesn't have interference (the fast rate).

The model predictions can be computed with the procedure 4. The procedures compute the different periods until the application completes (i.e., the $remaining$ variable reaches 1). During a period, the procedure first computes the progress for the slow rate and the progress of the fast rate (the same order as in the experiments). In case of the application remaining exceeds the completion value (line 7 and line 11), the procedure rollbacks the predicted times. The procedure returns the predicted time.

5.8.2 Theoretical Model Calibration and Results

To use the model on the different experiment one need to find the proper values of T, c_1, c_2 and α . From the experiment configurations used, the time period T equals to $60\ s$, and α depends on the interference pattern. Table 5.10a shows the calibrated values for α.

Algorithm 4 Theoretical model prediction procedure

```
1:  procedure PREDICT(α, c1, c2, T)
2:      remaining ← 0                          ▷ Completed when remaining is at 1
3:      time ← 0                               ▷ Initializes the prediction to 0
4:      while remaining < 1 do
5:          remaining ← remaining + T * α * c1 ▷ Updates progress of slow rate
6:          time ← time + T * α                ▷ Increases time
7:          if remaining > 1 then              ▷ Rollbacks if progress exceed 1
8:              time ← time − (remaining − 1) ÷ c1
9:              Break                          ▷ Exits while loop
10:         end if
11:         remaining ← remaining + T * (1 − α) * c2 ▷ Updates progress of fast rate
12:         time ← time + T * (1 − α)          ▷ Increases time
13:         if remaining > 1 then
14:             time ← time − (remaining − 1) ÷ c2
15:         end if
16:     end while
17:     Return time
18: end procedure
```

The values for c_1 and c_2 are directly calculated from the experiment results. The slow rate c_1 corresponds to the slope of the progress line of PDGEMM with constant interferences, while the value for the fast rate (c_2) corresponds to the slope of the progress line without interferences. The easiest way to take these values is to use the values of the run with constant interferences and without interference. For the *Paravance* and *Grisou* clusters, we take the mean runtime of the corresponding configuration, for the *ptask* we have only one value so we directly take the prediction. For more readability, we choose one configuration for the real executions, the configuration with *Nonblocking broadcasts*, and 50 subdivisions. The obtained progress rates are presented in table 5.10b.

Interference pattern	α
No interference	0
15s interference / 45s idle	0.25
30s interference / 30s idle	0.5
45s interference / 15s idle	0.75
Constant interference	1

(a) Values for α corresponding to the different interference pattern used in the experiments.

Experiment	c_1	c_2
Grisou	$1 \div 1455.86$	$1 \div 379.14$
Paravance	$1 \div 430.49$	$1 \div 374.29$
Ptask	$1 \div 935.00$	$1 \div 312.00$

(b) Values of c_1 and c_2 calculated from the experiment's result (*Grisou*, *Paravance* and *ptask*).

Figure 5.10.: Model parameters.

Figure 5.11 shows the obtained results for each different calibration. We observe that with the proper calibration of the theoretical model is able to predict, for each interference pattern, the runtime of both *Grisou* and *Paravance*, and for the

Ptask. This result suggests that the *ptask* has the potential to accurately simulate PDGEMM application provided that one modifies or calibrates the model to increase its accuracy.

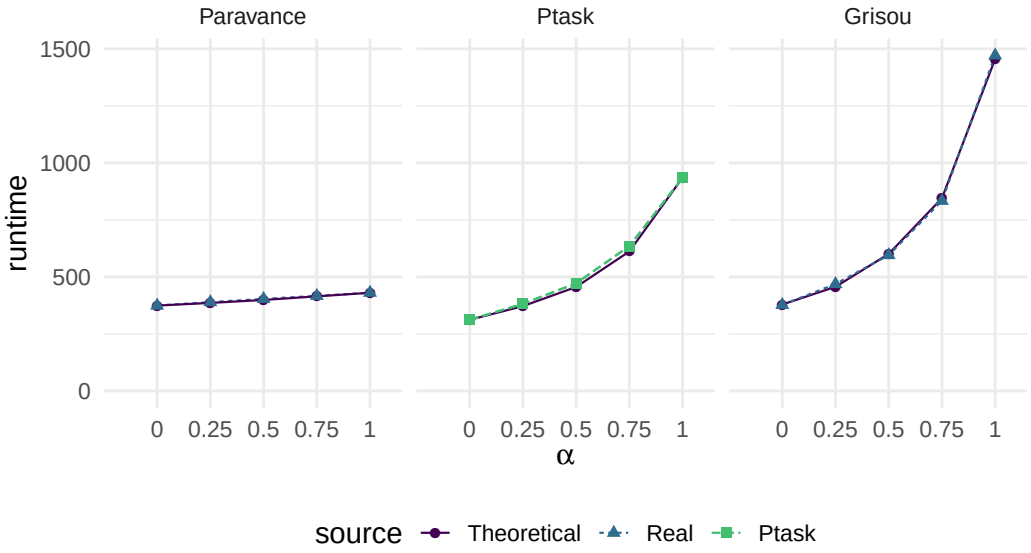


Figure 5.11.: Theoretical model's results for the different calibrations. The x-axis shows the different values of α, corresponding to the different interference patterns.

5.9 Discussion

In Figure 5.5a, we observe that PDGEMM alternates between phases in which the network is used (around 27s) and phases in which it is not used (around 5s). However, the *ptask* aggregates the whole network activity into homogeneous network activity.

In the case of periodic network activity or bursty network activity, the *ptask* loses information about these behaviors. Figure 5.12 illustrates this effect: the *ptask* loses the periodic effects of the real activity, and the network is always solicited but with a lower intensity. The *ptask* model is not able to simulate this effect with only one *ptask*, and therefore, in this case, the *ptask* might fail to soundly simulate the interferences. For instance, if another application uses the network (only) during the computation phases, the observed application should not be impacted. In contrast, if the application has only one network burst during its execution, the *ptask* might generate a low network activity that doesn't represent the burst.

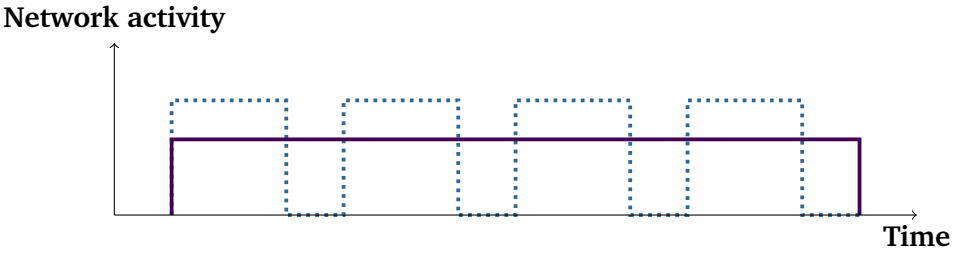


Figure 5.12.: Illustration of how the *ptask* simulates a periodic application. The dotted blue line represent an example of a real activity. The solid purple line shows the activity of *ptask*'s simulating the blue line.

One solution is to use different *ptasks* to simulate a single application, each *ptask* simulates different phases. For instance, for a periodic application, one can alternate between a *ptask* for the communications and a *ptask* for the computations.

A typical scheduling simulation features a lot of (different) jobs. Jobs that last days can be simulated along with jobs that only last hours or a few minutes. In this context, for one application alternating between phases of 27 seconds of communications and 5 seconds of computations during one day, it would require more than two thousand of *ptask* executions. The same reasoning applies to short applications: An application of ten minutes alternating between two phases of 1 second requires 600 *ptasks*.

Although this solution of using several *ptasks* to simulate the temporality of an application is interesting, it raises a more fundamental important question: **What temporality one has to consider to create application models suitable for scheduling simulations?** In other words, what is the time scale that we need to consider for creating application models?

- A small time scale increases the level of details of the simulation and potentially the total time of the simulation. It simulates short jobs with several *ptasks* extending the number of *ptasks* required for very long jobs.

- A large time scale exacerbates the smoothing effect and decreases the level details of the simulation. It simulates both small and long jobs with a single *ptask*.

Finding the proper time scale should be a trade-off between the level of details required for the study, and the simulation time. Another possible solution is to run preliminary simulations with a large time scale to detect situations that are worth to study with a small time scale.

5.10 Conclusion

The chapter evaluates the *ptask* model by comparing it to a real MPI application (PDGEMM) under different network interference patterns. The experimental setup benefits from the Grid'5000 infrastructure to execute a real application and to create controlled network interference with a real HPC cluster — on a TCP network. The evaluation uses two different clusters: *Paravance* and *Grisou*.

5.10.1 *Ptask* calibration

The results obtained show that the *ptask* model provides acceptable predictions for PDGEMM applications when there are no network interferences (error around $10\ \%$), however, the accuracy of the model decreases as the quantity of interferences increases.

Despite the accuracy issue under interferences, we show that the *ptask* has the potential to soundly simulate PDGEMM application. Indeed, the *ptask* model shows the same behavior under interferences, both PDGEMM, and the *ptask* has two modes: a slow mode during interferences, and a fast mode when there are no interferences. This behavior has been verified with a theoretical interference model that accurately predicts the running time of the *ptask* model and the PDGEMM application, provided that the theoretical model is calibrated. This suggests that the *ptask* model can accurately predict the running time of PDGEMM with proper calibration.

Calibrating the *ptask* model is not yet possible as it doesn't currently expose parameters to be calibrated. Modifying the model might be necessary to incorporate different parameters for the calibration. First, one needs to identify the parts of the model that need to be calibrated. The *ptask* model is implemented into the simulator SimGrid, which is a complex piece of software. Identifying the part of the model that needs calibration is not immediate and requires further investigations.

SimGrid tracing is equivalent to monitoring in reality but in the simulation. With this feature, it is possible to trace the activity of *ptask* during the simulation: The quantity and time of the bandwidth used on the different links or the computation activity of the hosts, etc. Using tracing is a good way to compare the real network activity with the network activity in simulation, and to identify differences that could lead to a calibration point. However, the tracing of SimGrid is currently broken[4]: a first step toward *ptask* calibration is to fix the tracing.

[4] https://framagit.org/simgrid/simgrid/-/issues/40

5.10.2 Scheduling Simulations

As stated in the previous chapter, the *ptask* has reasonable simulation time for scheduling simulation, while being able to simulate the resource consumption of an application. The results obtained in this chapter show that the *ptask* model has the potential to simulate an HPC application with interferences. Particularly, the model has been evaluated for an application with a homogeneous activity for the network and the computation, for which the *ptask* is well suited.

Nevertheless, our final objective is to create a model of application suitable for the simulation of next generation platforms and applications. Currently, the *ptask* has been tested only for one application with synthetic interferences. Further investigations are necessary to understand the capability of the model. For instance, what is the accuracy of the *ptask* in a scenario where the interferences are replaced by another application?

Additionally, if PDGEMM is a real application, it is not a production application and it has been developed for this evaluation. Comparing the *ptask* model to real production applications is needed to increase our confidence in the model. Real HPC application can be complex and requires experts to understand their resources consumption. However, leveraging monitoring techniques used in this chapter one can pinpoint phases in the application that show homogeneous patterns. Identifying these phases is the entry point to simulate complex production applications.

6 Study RJMS with the Emulation Approach

6.1 Introduction

There are two families of methods to study a distributed application: the ones that work on the implementation of the target application, and the ones that work on a model of the target application (using simulation). Resource and Job Management Systems (RJMS) are distributed software in charge to manage the job and the resources of a computing platform. In chapter 3 we evaluate a new scheduling policy in simulation, and in chapters 4 and 5 we propose to extend scheduling simulations with a dedicated model for the simulated jobs. In this chapter, we are interested in techniques and methodologies that enable to study real RJMS, instead of using a model of it.

RJMSs are complex and configurable software, the scheduling policy of real RJMSs often incorporates a large part of real cluster constraints, such as different queues depending on the project to handle different priorities, or to configure the queue of the backfilling, etc. One common methodology to study RJMS is to rely on Simulation. Simulation, by essence, uses a model of the targeted systems. Creating a model able to capture the whole complexity of a system is a complicated issue, and needs to be validated (or invalidated) before using it (chapters 4 and 5). Furthermore, RJMSs are subject to evolutions, and therefore one RJMS model needs to be updated (and validated) with the evolution of the RJMS. These aforementioned approaches use a model of an RJMS, and therefore these studies do not directly target a specific RJMS implementation.

Some scenarios impose to use a specific RJMS implementation. For instance, experimenting with real RJMS is a good option to conduct preliminary experiments, to help system administrators to tune the RJMS configuration for their cluster. Or to enable developers to test new features during the development phase.

As detailed in chapter 2, the methodologies used to experiment with real RJMS either rely on *emulation* or *in-vivo*. *in-vivo* consists of executing a real RJMS on a real platform. *in-vivo* is close to a production system, but this approach can be time-consuming, complex, and difficult to reproduce. *In-vivo* experiments also have the drawback of being limited to the platforms at hand. *Emulation*, on the other hand, consists of studying a real RJMS on a platform model. This approach is a good alternative to *in-vivo*, as it enables to study an RJMS in a controlled environment

and permits the study of *what-if* scenarios, such as « What if my platform has more computing nodes? » or « What if my platform has a smaller network? ».

State of the art *emulation* techniques (detailed in chapter 2) mostly relies on containers and virtual machines to set up a platform during the experiments. This methodology is useful to create larger platforms than the platforms at-hand. However, we refer to this approach as *hybridization* because the platform is partially modeled: the physical resources are virtual, and enable to execute a real operating system (such as Linux). One drawback of the virtualization (or using containers) is that the modeled platform is limited to the capacity and the architecture of the platform at hand. Distem [Sar+13] enables a different platform on top of a real platform, with the limitation that the modeled platform is less performant than the real platform.

These tools and methodologies are a good start to study RJMS in a controlled environment, and study scenarios involving platforms that are not available to experimenters: larger platform using a large number of containers on a single node, or platform with degraded network performances for instance with Distem. However, even if it enables to broaden the scope of study than one can do with RJMS, these methodologies still require a computing platform to carry them.

The Slurm simulator [Luc11; Rod+17; JDC18] and the Flux simulator [Pol+18] on the other hand, enables to study and execute a complete RJMS on a single computer. This feature is convenient for testing new innovations or evaluate new scheduling algorithms. However, the Slurm simulator is tightly coupled to its implementation (and to a particular Slurm version), at each new version of Slurm the simulator needs to be updated and evaluated. Additionally, the simulator implementations are dependant on a particular RJMS. Reproducing the approach for other RJMSs needs to deal which each RJMS's idiosyncrasy (programming language, architecture, etc). For the Slurm simulator, and the Flux simulator the description of the simulated platform is limited and is not able to simulate a realistic platform with different network topologies for instance.

Current approaches designed to experiment with real RJMS, are limited either in the number of scenarios one can study, or because they are tightly coupled to a particular RJMS implementation, or worse a particular RJMS version. Our motivation is to propose new tools and methodologies to experiment with RJMS that, first don't depend on a particular RJMS implementation, and second that can bring the full potential of a platform simulator such as SimGrid or Batsim.

The contribution of this chapter is the presentation of two new approaches to execute a real RJMS on a simulated platform, both enable to execute an RJMS on a single computer. The two approaches are based on a (different) mechanism that enables to use of a platform simulator, the former uses SimGrid while the later uses Batsim.

This chapter details the presented approaches from a high-level point of view and doesn't detail the different tools and pieces of software used to carry out *Simunix* and Batsky. This separation enables us to focus on the approaches and their utility, and facilitate their comparison. The focus of the chapter 7, on the other hand, fills the gap and details the different tools and software used and developed to implement these approaches.

6.2 The *Simunix* Approach

Simunix initial objective is to execute, on a single computer, the RJMS Slurm on a simulated platform. The simulated platform is managed by the simulation toolkit SimGrid. Despite the main objective of *Simunix* which is to emulate Slurm, the approach doesn't depend on the Slurm source code. *Simunix* can be seen as a sandbox which executes a program, and changes (some of) the functions called by the program to modify or replace their original behavior. In other words, *Simunix* is able to change the function called by a program to replace them with a simulated version of the function. *Simunix* is not coupled with Slurm (or another RJMS) because the replaced functions are from the C library (*libc*), which makes this approach usable with all programs that rely on the *libc*.

An RJMS is by essence a distributed software that runs programs (daemons) on the computing nodes of a cluster and uses a program on another node to manage the cluster. To execute an RJMS on a single computer, *Simunix* creates a SimGrid platform that simulates a cluster and executes all the RJMS program on a sandbox. The programs are executed on isolated sandboxes and only a subset of the *libc* functions are intercepted. This subset comprises all the communication functions (the BSD socket API), all the process management functions (such as fork, or thread), and the functions to manage and request the time.

SimGrid features a programming interface to create actors that interact with the simulated platform. This model enables *Simunix* to simulate the program running in the sandbox with actors; an actor is executing on a simulated node of the SimGrid platform. *Simunix* project implements the *libc* functions with the Remote SimGrid API, which creates a wrapper around SimGrid to isolate the different actors of the simulation into a single process. With *Simunix* an actor can either be a process (in case of a fork) or a thread. For instance, when the process calls the function *gettimeofday*, *Simunix* intercepts this call and use the actor corresponding to the current process to get the simulation time from SimGrid. The RSG API enables actors to communicates with each other. The RSG's actor model is detailed in section 7.3. Figure 6.1 illustrates how *Simunix* works for a single execution with two different processes from a high-level perspective.

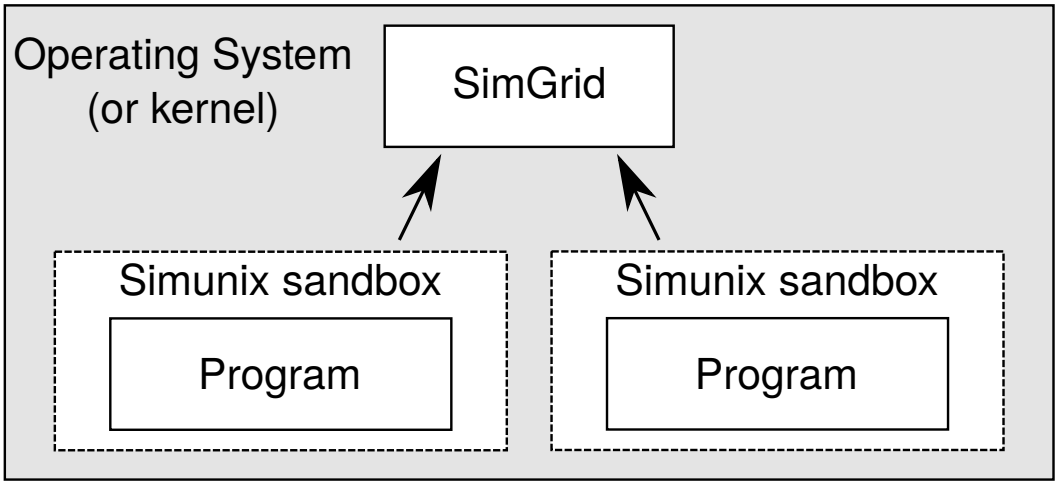


Figure 6.1.: High-level illustration of a *Simunix* simulation involving two programs. The execution of the two programs are on the same computer, and therefore the SimGrid execution and the two programs share the same operating system. Each process is executed into a different sandbox, and the black arrows depict the function replacing the original *libc* function and interacting with SimGrid. The sandbox is permeable: the function that is not intercepted by the sandbox are executed normally by the OS, on the other hand, the intercepted functions use the SimGrid simulation. Note that to provide process isolation for each separated actors, the sandbox accesses the SimGrid simulation using the RSG project.

6.2.1 Project Historic

Simunix's first idea has been originally created by David GLESSER with SimGrid, which developed the first *Simunix* version. The first version presented several limitations: the global variables were shared among the different processes breaking the consistency of each individual process. To code this architectural issue, the second version uses RSG. The development of the second version is a contribution of this dissertation [Gle16; GF15]. The original version of RSG has also been developed, and many features have been incorporated into the project to support the development of *Simunix*.

The second *Simunix* version uses the *ELF poisoning* (detailed in section 7.2) approach to intercept the relevant functions of the *libc*. The first proof of concept was able to simulate a cluster up to 10 nodes [GF15], and worked on different Slurm version without requiring any modifications. Unfortunately, recent changes in the *libc* made the use of *ELF poisoning* failing to intercept the *fork* function. The issue comes from changes in the *libc* impacting the ELF structure to the point where the *fork* function doesn't appear in the *ELF* file. More investigations are necessary to pinpoint the origin of this behavior.

Simunix project is not maintained anymore. However, the *sgwrap* project can be seen as the third version of *Simunix*. It aims to generalize the *Simunix* approach and extend the idea for other programs. *sgwrap* is similar to *Simunix*: It models

the *libc* using the RSG project. The current implementation is at an early stage of development and doesn't support yet the execution of Slurm.

The tools and mechanisms used to build the different versions of *Simunix* are explained in section chapter 7.

- The first tool is the interception mechanism that enables to replace the function called by the program and therefore creates the sandbox.
- The second tool is called Remote SimGrid, it creates a wrapper around SimGrid to isolate the different program of Slurm to system processes (which is not possible with SimGrid only).
- Finally, *sgwrap* models the functions of the *libc* using the RSG API. Although *sgwrap* is more recent than *Simunix*, both *sgwrap* and *Simunix* use RSG to model the *libc*. Therefore the *libc* modelization stands for both projects.

6.2.2 *Simunix* Use Case: Slurm Emulation

The initial objective of *Simunix* is to be able to execute the Slurm RJMS on a single computer, using the SimGrid simulation toolkit. A typical Slurm installation features a Slurm controller (*SlurmCtld*) and Slurm daemon (*SlurmD*) per node. The *SlurmCtld* is responsible for the scheduling and handling the job submissions. The *SlurmD* are responsible for the node it is running to, including the management of the job's execution. The controller controls the *SlurmD* processes with a remote procedure call protocol. To submit a new job to the Slurm installation, the *srun* program can be used.

The figure 6.2 depicts how to execute Slurm with *Simunix* on a simulated platform. The different programs necessary for the Slurm execution are executed on their dedicated sandboxes. As explained in the previous section, each sandbox handles the set of actors necessary for the execution of the program. An actor can be either a process or a thread. Slurm uses the *libc* socket API to communicate between the different processes of the installation. The functions are intercepted (with the sandbox) to use the SimGrid simulation instead.

As explain in chapter 1, to experiment with RJMS, on common input is the workload. The workload is the list of jobs that needs to be scheduled during the experiment. To inject the different jobs, during the execution of Slurm with *Simunix*, another program is executing in *Simunix*. Its sole objective is to execute the *srun* program (inside a sandbox) to submit new jobs.

The jobs of the experiment are a simple call program calling the system call *sleep*. When the *SlurmD* daemons execute jobs, it uses the *fork* function to create a new

process. *Simunix* intercept the call to the *fork*, and create a dedicated actor for the jobs, therefore the *sleep* function called is also intercepted.

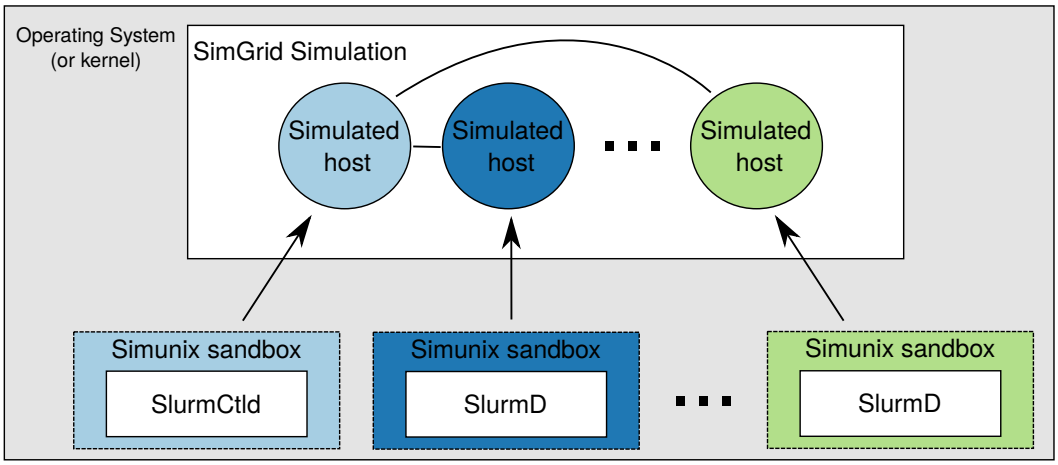


Figure 6.2.: Example of usage of *Simunix* with Slurm.

6.3 Batsky

Batsky is an approach to plug a real RJMS on a simulator such as Batsim. Batsky leverages the fact that RJMSs, at some point, need to launch an external job — a user's program. Using a special job, Batsky can inject a program to take the control from the RJMS and gather information about the RJMS decisions. The current state of the project is a proof a concept that aims at executing Slurm.

The main principle is based on two main components:

- A simulation controller, that handles the time of the simulation, the job executions, and the job submissions. All requests concerning the time emitted by the RJMS are intercepted and transmitted to the controller. These requests involve, among other, the call to functions such as *sleep* or *gettimeofday*.
- The second component is *BatJob*: A program that is started when the RJMS launches a job. *BatJob* connects to the simulation controller and sends information about its allocation. The controller notifies the *BatJob* when it shall exit.

With this design, Batsky is able to launch the RJMS in a controlled environment, controlling the time and jobs. Batsky aims at using Batsim as a simulation controller, giving a way to plug any RJMS on the Batsim project to benefits form the job models(chapters 4 and 5). Figure 6.3 depicts an overview of Batsky.

Batsky works with an RJMS setup, in our case we target the RJMS Slurm. More details about Slurm is provided in previous section (6.2.2). Batsky starts a Slurm

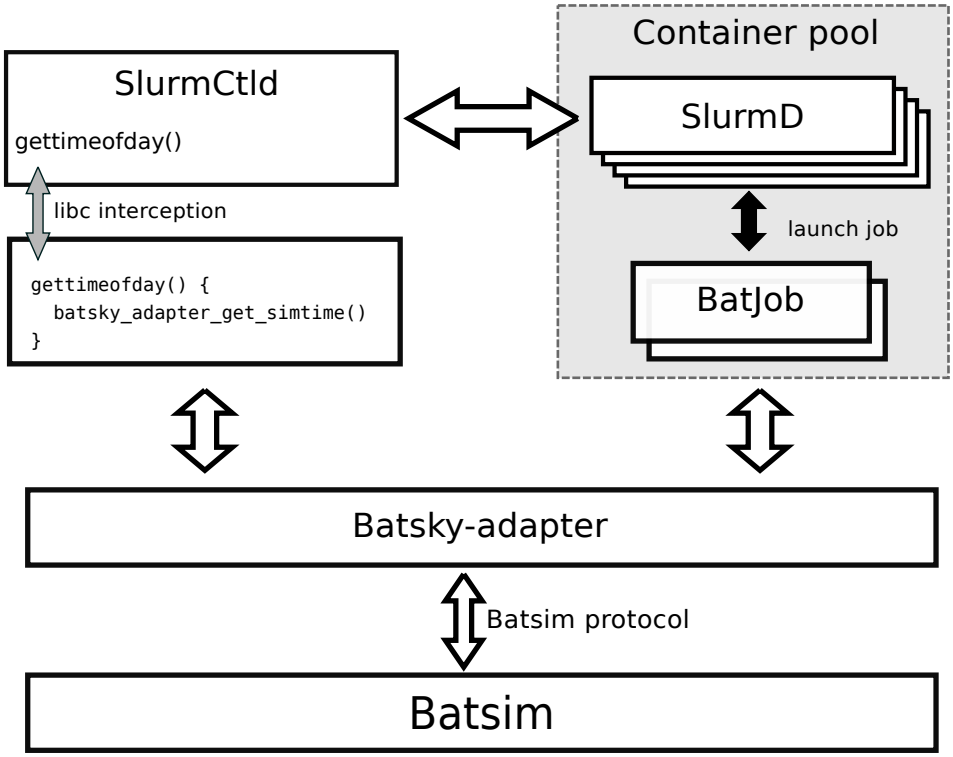


Figure 6.3.: Batsky overview. The *Batsky-Apdater* is the bridge between the simulation and the reality, it controls the time provided to *SlurmCtld*, and gathers information from the BatJob. The white arrows represent network traffic between the different components.

controller daemon (*SlurmCtld*) with a time interception mechanism. The interception mechanism is based on linking a modified C library to the target application (Slurm). Unlike *Simunix*, the network communications are not intercepted by Batsky, hence the communications between *SlurmCtld* and the *SlurmD* daemons are not simulated.

Figure 6.4 shows the sequence diagram of the execution of one job with Batsky (The circled numbers identify a phase in the sequence diagram):

1. A **Batsim** simulation is executed. Batsim is in charge of providing the simulation time, handling the workload, and the job executions (1).

2. ***Batsky-Adatper*** is the scheduler of the Batsim simulation. Its role is to be the intermediary between the Batsim simulation and the Slurm installation. The adapter receives the job submission from Batsim, and create the adapted Slurm request (using the *srun* utility) (2). The intercepted time requests of *SlurmCtld* are redirected to the adapter, which interacts with Batsim to provide the current simulation time.

3. A **Slurm installation**. A *SlurmCtld* is executed with a time interception mechanism, at each time-related functions, *SlurmCtld* calls a modified time function. A number of *SlurmD* are executed in different containers and are in charge of

executing the *BatJob* on the containers. When *SlurmCtld* takes the decision to launch a job (after a scheduling phase for instance), it notifies the *SlurmD* in charge of the nodes allocated for the new jobs and sends the job execution requests (3).

4. ***BatJob***. *BatJob* is the program executed for all the jobs of the simulation. When *SlurmD* starts a job, it starts a *BatJob* and sets an environment variable specifying the node allocated for this job (4). *BatJob* sends the content of the variable to *Batsky-Adapter*. When the job is terminated, Batsim notifies the adapter, which in turn notifies the corresponding *BatJob* for its termination.

5. When the job finishes, in (6), Batsim notifies the adapter the termination of the job 1, which in turn notifies the corresponding *BatJob* (7). *Batjob* naturally exits when it receives the exit message from the adapter. Finally, *SlurmD* notifies the *SlurmCtld* that the job is complete (8).

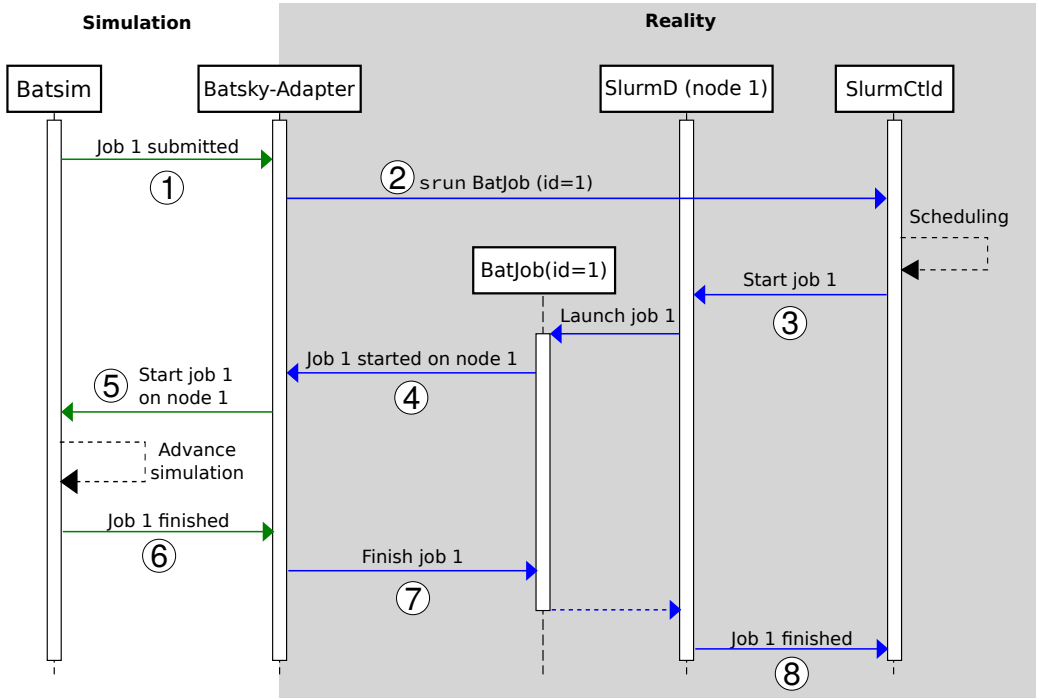


Figure 6.4.: Sequence diagram of the simulation of a job with Batsky and Slurm.

Time Synchronisation

The time interception enables to control the time of the simulation and therefore to speed up the execution time.

The simulation time is lead by the events of the simulation. If the next event of the simulation is in one hour, the simulation jumps directly to the next event, increasing the simulation time by one hour. When Slurm polls the system time it is intercepted

and Batsky sends the time of the simulation. Therefore, between two consecutive Slurm's calls to poll the system time, the simulation could have advanced of one hour.

One challenge of Batsky is to keep the RJMS in a consistent environment regarding the time of the simulation. For instance, at the beginning of the simulation, the Slurm's setup routine performs an initialization phase, in which the *SlurmCtld* connects to every *SlurmD* daemons. These phases are often defined by a number of communications that are not relevant to the simulation. For instance, when executing or terminating a job, Slurm does numerous control checks with the daemons. If during these phases, the simulation time jumps are too high, Slurm detects than an issue.

In order to keep the simulation time consistent, Batsky defines a number of phases where the time flows normally. During these phases, Batsky uses the real system time — The calls to the time functions are still intercepted, but Batsky provides the system time instead of the simulation time. From Batsky point of view, it is difficult to detect when the real system time phases must end. Therefore, we use a fixed amount of time during which Batsky the times flows normally.

Using the real time increases the simulation time because, during this period of time, the time is not accelerated. Using a long time period for the slow phases increases the simulation time but ensures the consistency of the system. On the other hand, a short time period dedicated to these phases enables speeding up the simulation, at the cost of risking to break the system consistency.

A calibration of Batsky is necessary to find the parameters giving the proper balance between consistency and simulation time.

6.4 Discussion

Simunix and Batsky are two different approaches to *emulate* RJMS. Even if both approaches take a similar methodology: modifying the behavior of a program without requiring modification of its source code. *Simunix* takes a complete approach and intercepts every class of functions necessary to have complete control over its execution. Batsky on the other hand intercepts only the function related to time to be able to control it, and therefore speed up the execution of the targeted RJMS.

Simunix's approach strongly relies on the simulation capabilities of RSG. The second version of *Simunix* enabled to emulate Slurm with few nodes. Furthermore, the approach requires no modifications of the source code of the target program, as a proof of concept three different versions of Slurm were simulated without modifications [GF15]. Recently, the *sgwrap* project takes a similar approach to reproduce the concept of *Simunix*. Intercepting every function necessary to have complete

control over the execution of a distributed program is complex. Any failure in the implementation can lock the whole simulation, and render the whole project unusable until a fix is provided.

On the other, Batsky leverages the fact that RJMSs, at some point, executes a user-provided job, to inject a Batsky related job. This job enables Batsky to, first get information about the scheduling decisions (such as the allocated node for a job), and second get control over the RJMS execution. The approach seems to be applicable for others RJMS, in [Lar20] a similar approach has been used for the *kubernetes* [@kubernetes] scheduler.

Simunix global approach has the advantage to be extendable to other programs (written in C, and using the *libc*). Additionally, *Simunix* relies on RSG which is a more generic tool designed to be used on diverse distributed software. Batsky, takes a hybrid approach and is tailored to be specifically used with RJMS. Batsky presents a better compromise to study RJMS for two (connected) reasons: it brings a way to plug a real RJMS on Batsim to benefits from the Batsim job models, and the approach seems usable with other RJMSs. Indeed, although both RSG and Batsim are based on SimGrid, the level of abstraction proposed by Batsim is more appropriate to study RJMSs. Batsim's ecosystem is dedicated to the study of RJMS, it provides tools and practical features dedicated to RJMS study: it handles the workload (1.3), and proposes evaluation tool such as *evalys* [@evalys].

6.5 Conclusion

Simulation is a good option to experiment with RJMS, the approach has been widely used for scheduling algorithms. One of the major issues with simulation is that the abstraction used to design the simulation doesn't always incorporate all the complexity of a real RJMS execution. Nevertheless, some scenarios require to experiment with a real RJMS: testing, tuning the configurations.

A promising approach is to rely on simulation, which involves using a real RJMS on a model (or an abstraction of the platform). Current emulation methodologies strongly rely on containers or virtual machines to create a platform different from the platform at hand. However, using containers or virtual machines has limitations: the experiments are complex and often require a bootstrap platform with few nodes to simulate an RJMS at scale.

In this chapter, we propose two approaches to emulate a real RJMS on a single computer. *Simunix* and Batsky: *Simunix* aims at simulating a whole distributed application on a single computer by controlling every function related to communications, processes, threads, and time. Batsky leverage the design of RJMS to take

control over its execution. We believe that the two approaches are promising for the following reasons:

- Both approaches rely on SimGrid which enables to simulate a large number of different platforms with different network topologies for instance.
- Both approaches enable to study RJMS independently of a specific RJMS implementation, or version.

Batsky seems to be a promising approach for simulating RJMS, besides the approach has been shown useful for the scheduler of *kubernetes*. As future research, concerning Batsky, we outline two directions. In the first place, a more complete use case is necessary to understand the potential of Batsky— using Batsky on a larger workload and with a larger platform. The second direction is to evaluate the accuracy of the simulation, by comparing a Batsky execution to a real use case, or a real Slurm installation (on Grid'5000 for instance).

In this chapter, the two approaches are presented from a high-level point of view, to keep the description simple and encourage a better comparison between both approaches. The implementation of the two approaches rely on technicals tools and software that are presented in chapter 7.

7 Tools for Emulation: Interception, Remote SimGrid and *sgwrap*

7.1 Introduction

This chapter focuses on the technical tools supporting the projects *Simunix* and Batsky presented in the previous chapter (chapter 6). All the tools and projects are available online (appendix A.2).

The first presented tools are the different mechanisms used to create a sandbox to change the behavior of a program without modifying its source code. Both *Simunix* and Batsky use one of these mechanisms. To the best of our knowledge, three different mechanisms exist and are detailed in this chapter.

The second tool is called RSG, which is a simulation toolbox build upon the SimGrid simulation toolkit.

Finally, the third tool is a library named *sgwrap* that implements the function of the *libc* using RSG. The section details the modelization of the *libc* function with the simulation concepts provided by RSG. The two former tools are solely used for the *Simunix* project.

7.2 Interception Methods

This section regroups the different mechanism existing that enable to change the behavior of a function (or a set of functions) called by a program without changing its source code.

LD_PRELOAD

LD_PRELOAD [Pul09] is an environment variable recognized by the loader that enables to load a shared object in priority. The loaded objects (such as functions) will override the original functions, that should have been loaded without *LD_PRELOAD*. Once loaded, the program will call the function defined in the shared object pointed by the *LD_PRELOAD* variable instead of the original one. To take advantage of this feature, one needs to create a shared object with the implementation of the function one wishes to intercept. This variable is usually used to override a function, for profiling purposes for instance.

In the case of *sgwrap*, we use this variable to force the program to load function that, have the same behavior of the original ones, but instead of using the *libc*, we use RSG. Concretely, when a program calls the function, for instance, *gettimeofday* which polls the system time, we replace this function with a similar name which calls the simulation time instead (using RSG's API for instance).

One disadvantage of this method is that all the original functions remain unreachable, that is to say, inside the overridden function it is not possible to use the original function without starting an infinite recursion loop. However, *sgwrap* needs to use the real network to communicate with the SimGrid simulation, therefore it is necessary to be able to access the original *libc* functions. To cope with this issue, we need to set up two things:

- Because the original function's names have been overridden, we need to be able to access to the original function. We use the *dlopen* function that enables to load, at execution time, a shared object to call its function.

- We use a global variable that enables to detect if a function is being intercepted (and is waiting to return). In this case, the variable should be set to *true* to tell *sgwrap* that it needs to call the real system function.

ELF hooking

The Executable and Linkable Format (ELF) is a common file standard use by the UNIX family of OS. It is a format defining an executable file. The ELF hooking [@elfhook; @LIEF] methods works by, at execution time, rewriting the binary file of the executable to change some function of the executable with the function one wishes to execute instead. As for the *LD_PRELOAD* mechanism, the user of this method needs to provide a shared object file containing the functions to change — This is can be done by writing a shared library for instance. However, it is not necessary that the function to replace holds the same name as the replaced function (as it is the case for *LD_PRELOAD*). This enables to directly call if needed, the original *libc* function within the shared object containing the functions replacing the original.

This method is used in the second version of *Simunix*. The direct advantage of using this method is that, unlike the *LD_PRELOAD* method, there is no risk of infinite loop recursion. However, this approach is more complicated, as it directly relies on manipulating the ELF file of the executable that is intercepted. If the ELF file doesn't respect some constraints, the method can fail to intercept some functions. The ELF file generation depends on a lot of parameters, such as compilation flags. If an error occurs (for instance, the wrong flag is set), it can be difficult to track.

Libc linking

It consists to directly change the *libc* of the program by providing a custom *libc* with the modifications necessary. The custom *libc* keeps the same structure apart from the function that one wishes to change the implementation. In order to access to the unwritten function, the *libc* provide for each function a similar function prefixed with two underscores — *sleep* can also be called with *__sleep*.

This is the method used in the Batsky (6.3) project. The difficulty of this approach is to be able to change the *libc* of a program without changing the *libc* of the whole system. Indeed, the *libc* is generally installed on the system and dynamically linked to the programs. The Batsky approach to change the *libc* of Slurm is to use the package manager Nix, which enables to have different versions of the same package (in our the *libc*) without side effect.

7.3 Remote SimGrid

Remote SimGrid (RSG) contribution is twofold:

- First, it defines a set of high-level simulation concepts that should ease the modeling of most distributed applications.
- Second, it provides a toolbox implementing these concepts using a well-known simulation framework: SimGrid [Cas+14].

RSG has been originally written by Martin QUINSON. To fully support *Simunix*, many features of RSG has been developed during this work. The project has been rewritten to increase its maintainability, the basic architecture of the last version has been developed by Millian POQUET and Adrien FAURE.

This section is organized as follows. We first present the simulation concepts to describe distributed applications. And we finally details explains how our prototypal SimGrid-based toolbox called (RSG) works.

7.3.1 RSG Simulation Concepts

A simulation is composed of several **actors** that execute user-provided functions. The actors have to explicitly use the provided API to express their computation, communication, disk usage, and other **activities**, so that they get reflected within the simulator. These activities take place on **resources** such as *hosts*, *links* and *storage* units. The simulator predicts the time taken by each activity and orchestrates the actors accordingly, waiting for the completion of these activities.

When communicating, data is not directly sent to other actors but posted to a **mailbox** that serves as a *rendez-vous* point between communicating actors. Actors

issue *put* requests in a mailbox, that are matched with complementary *get* requests. The concept of mailboxes can be paralleled with many others. The phone number, which allows the caller to find the receiver. In TCP, the pair (hostname, host port) to which you can connect to find your peer. Finally, in HTTP, URLs through which the clients can connect to the servers. One big difference with most of these systems is that no actor is the exclusive owner of a mailbox, neither in sending nor in receiving. Many actors can send it into and receive from the same mailbox. TCP socket ports for example are shared on the sender side but usually exclusive on the receiver side (only one process can receive from a given socket at a given point of time).

Distributed applications are composed of multiple processes that can be distributed over several machines while SimGrid is implemented as a single, centralized process. This makes it impossible to directly use SimGrid to study the independent processes of a distributed application. Remote SimGrid (RSG) is a technical solution designed to allow the study of actual distributed applications with SimGrid. RSG is free (licensed under GNU AGPL-3.0) and available online [@rsg].

Figure 7.1 illustrates the core architecture of Remote SimGrid. RSG provides a **rsg_server** program in which the SimGrid simulation takes place, and a **librsg** library used by the processes of the distributed application under study. The SimGrid actors defined in *rsg_server* incorporate a *Remote Procedure Call (RPC)* server, which offers clients to remotely perform an action on the server or to access some data on it. Here, *rsg_server* creates and attaches a unique *RPC* server to every actor of the simulation. As a result, each actor is remotely controllable by the attached *RPC* client. The *RPC* client is implemented in the *librsg* library, which exposes SimGrid concepts (defined in section 7.3.1) through its own API.

7.3.2 Implementation Details

The API of RSG allows to issue *put/get* requests on a mailbox synchronously or asynchronously.

SimGrid predicts the time taken by each activity and orchestrates the actors accordingly. To do so, SimGrid must know which activities every actor wants to do, so that it can compute what activity will finish next (and when). As a consequence, user-provided functions used as actors must not fall into a pattern where they do not call the provided API anymore — aka fall into an infinite loop. While this may sound trivial, we have observed that forgetting about this is a common pitfall when using RSG for the first time. As a rule of thumb, we advise that the inter-process synchronizations of the studied application should either be fully simulated (via mailboxes), or that synchronization loops should contain API calls that advance the simulation time (typically, let the actor *sleep* for a while) — as this will pass the control to another actor, that will hopefully unlock the situation.

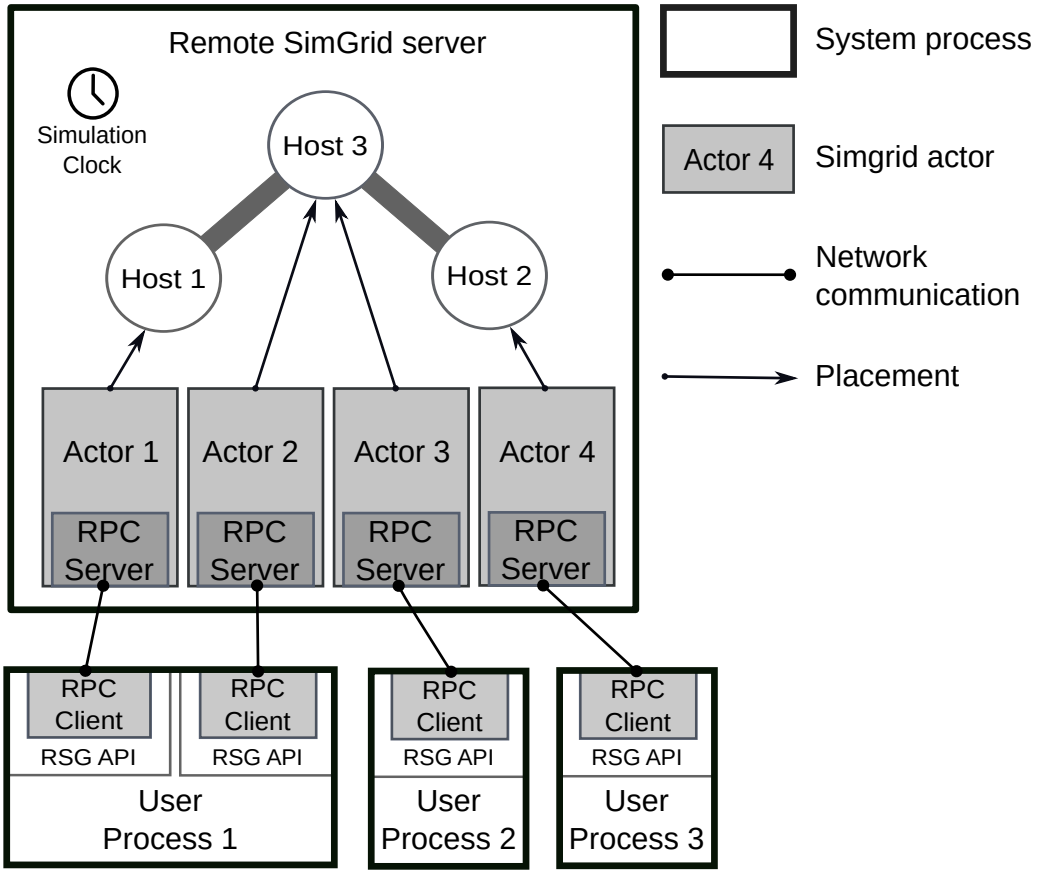


Figure 7.1.: Overview of a simulation with Remote SimGrid. The simulation consists of 3 hosts and 4 actors. Actors are bound to hosts, representing *where* the actors' actions take place. Each actor is controllable remotely thanks to an RPC mechanism. A single actor is automatically spawned per user process by default. This is not limiting, as actors can spawn other actors — e.g., here, actors 1 and 2 are part of the same system process.

7.3.3 Related Work

To the best of our knowledge, only a few solutions exist to sandbox existing applications in simulation. Most works in the literature mandate the use of a model of the application. These models are usually manually built, as in [New+15] where the authors maintain a model of the twitter infrastructure to formally verify that sough properties are met. However, maintaining such models up-to-date with their implementations requires a lot of work.

Other works propose to automatically build the application model. For example, MPISE [Fu+15] couples static analysis and symbolic execution to build a model that can then be used to verify some properties. This approach is very promising, but the obtained models are so far limited to qualitative studies while our goal is to also conduct quantitative *what-if* studies on the simulator. It is also unclear how this approach could be generalized to non-MPI applications.

DCE [Taz+13] aims at studying slightly modified arbitrary applications with the ns-3 [RH10] network simulator, and is therefore close to *sgwrap* presented in section 7.4. DCE focuses on real network protocol implementations and thus allows the study of kernel code. To that extent, the network protocols implementations of the kernel are provided as a shared library, that is also in charge of the synchronization with ns-3. Specific code is injected (using *LD_PRELOAD*) at the *libc* level to call the DCE kernel instead of the usual one.

In DCE, the authors chose to group all emulated processes under study in a single system process. This reduces context switching overhead during the simulation and arguably eases the application debugging with a single process debugger. We took the opposite design decision for sake of simplicity and robustness. Indeed, this process folding requires to privatize the global variables in the application but also in all libraries. This requires some OS-level tricks that are difficult to automatize and often interact badly with other tools that can be used when debugging the application (e.g., the DMTCP [AAC09] checkpointer). Besides, the ease of debugging implied by grouping a distributed application into a single process is debatable, as both have the same overall number of threads and can be fully deterministic.

7.4 C Standard Library Interceptions

wide part of the distributed RJMSs is programmed using the C library. This is also the case for most RJMS: Slurm, Flux, and Torque to cite a few.

In this work, we propose to model with RSG the concepts featured by the library C (*libc*) that helps the development of most distributed applications, namely the processes API (POSIX), the threads and the synchronizations, the function to poll the system time, and finally, the socket API enabling to communicate through the network (and the processes on the same machine). Coupling this modelization of the *libc* functionalities with a method for intercepting the *libc* function, we are able to run real-world applications without modifying the source code.

We propose an implementation of this modelization into a project called *sgwrap*. The implementation is inspired by the *Simunix* project. *sgwrap* is a collection of four C++ libraries dedicated to model *libc* functions with RSG. Each library features the interception of one of the features provided by the C library. Namely, time management (sleeping or getting system time), the threads and synchronizations, the network sockets, and the system processes (forking for instance). The libraries are designed to be executed together, however, this architecture enables to select the intercepted features by activating only the required libraries.

We built the network interception part of *sgwrap* upon an existing C library called *socket_wrapper* [@cwrap], which redirects the network traffic of a distributed ap-

plication through inter-process communication (IPC). The redirection is made by using library preloading, as described previously. To summarize, *socket_wrapper* uses *LD_PRELOAD* to redirect all sockets into IPC sockets.

7.4.1 Choosing the Intercepted Functions

The first class of functions to intercept are the functions related to the time. One objective of the simulation is to test new scenarios, during an experiment the time is important information. RSG (thanks to SimGrid) predicts the time that communications will take on the simulated platform. Therefore, the time of the simulation needs to be injected in the emulated program.

The second class of functions that need to be intercepted are *blocking* functions. Indeed, with RSG one actor of the simulation is executed at the time until it gives the control back to RSG. RSG decides which actor is running, and at each call to RSG (such as sending on a mailbox, or sleeping) the process gives the control back to RSG. If the simulated process initiates a blocking function, such as waiting for a signal or a Mutexes, the process never gives the control back to RSG which hard-locks the simulation. To satisfy this constraint *sgwrap* must implement every blocking function of the *libc* used by the program. This includes the communication functions (such as the sockets), the synchronization functions (Mutexes), the signals, etc.

The third class of functions contains the functions related to processes and threads management. The *libc* provides a wide API to manage, and control processes and threads. It is a common pattern to use the *fork* function to create new processes (a child). Furthermore, the *libc* provides a function to permit the parent and the child to communicate using dedicated channels. To keep the simulation consistent, *sgwrap* needs to keep track of the new processes, and include them in the simulation — each new process (forking) or thread needs to have a dedicated RSG actor. Otherwise, if a child's process is not integrated into the simulation, a parent may try to communicate with it and locks the whole simulation.

7.4.2 System Time Interception

The *libc* features different functions to, either retrieve the current time of the system or to wait for a specific duration (sleeping). RSG provides both possibilities, from any actor it is possible to ask the current time of the simulation or to sleep for a specific duration. Modeling the time is directly made by RSG, and doesn't present any difficulty as it is a direct match between the *libc* and RSG's simulation concepts.

7.4.3 BSD Socket Interception

Sockets are the core part of the application programming interface (API) provided by the operating system (OS) to interact with the network. A *socket* is an abstract

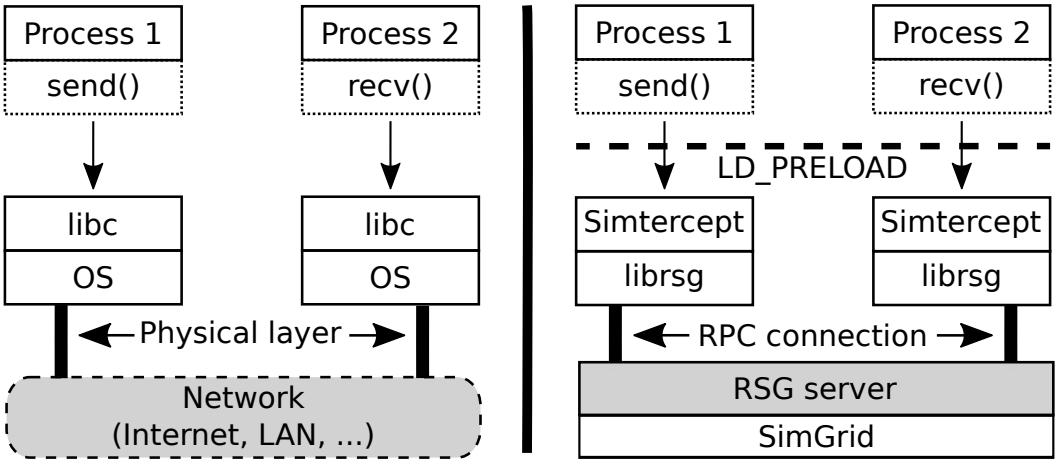


Figure 7.2.: Representation of the usage of *sgwrap* on a simple application composed of two processes. The left side shows how processes communicate usually — without *sgwrap*. On the right side, communications are simulated using Remote SimGrid. This is done by *sgwrap*, which intercepts calls to the BSD socket API and calls Remote SimGrid accordingly.

representation of an endpoint of one communication across a network or the Internet. Almost all distributed applications are programmed with sockets — unless they are executed without an operating system. Usually, applications do not manage sockets by communicating directly with the OS (via system calls) but prefer to rely on the standard library of their programming language. The C standard library (libc) typically encapsulates such communications with the OS in functions such as *listen* or *send*. Interestingly, most high-level programming languages decide not to reimplement such functions but to wrap their API around calls to libc functions. This means that almost all distributed applications, regardless of their programming language, use the same limited set of functions from the *libc* to interact with the network.

In this section, we present a methodology to study using simulation any distributed application that uses *libc* functions to interact with the network. This methodology is implemented in *sgwrap* [@simtercept], which is a free library based on Remote SimGrid.

sgwrap works by rerouting the network traffic of the target application into a SimGrid simulation. This is done by providing a custom implementation of the *libc* functions related to sockets that use Remote SimGrid internally. The application to study is executed in such a way that its *libc* functions are overridden by ours. Technically, this is done by executing the application with an adequate *LD_PRELOAD* environment variable, which changes how the system loads the program in memory. Figure 7.2 illustrates the general idea of *sgwrap*.

The remaining of this section describes how we used the generic concepts of Remote SimGrid to implement the OS behavior behind the *socket* API of the *libc*. The

following paragraphs mention the different stages of a socket life-cycle and explain how each one of them is simulated.

Socket Initialization. When an application wishes to create a socket, it calls the *socket* function that asks the OS to create an internal representation of the desired socket. Once the socket is created, the system returns a unique socket identifier to the caller. This identifier will be used in every subsequent function call to identify the connection. *sgwrap* intercepts the creation of the socket and creates and stores a virtual socket instead. The virtual socket is a structure containing metadata and information on the socket, such as its type or connection state. It allows *sgwrap* to keep track of the sockets. When a process desires to create a server (i.e., a socket willing to start multiple connections), it needs to assign an address to the socket, and then to set the state of the socket to *listening*. The functions *bind* and *listen* serve this purpose. *sgwrap* intercepts these calls, registers the socket binding address, and creates a *pairing* mailbox corresponding to the address — that is used during the connection stage.

Connection. Communicating sockets from two processes go in pairs. One socket needs to be identified as the client and the other as the server. A server socket, after having been assigned to an address, and after having been put into the *listening* state, can *accept* new connections. For this to take place, the client socket needs to know the address on which the server socket is listening. The client can then ask the server socket to engage a new end-to-end connection. The functions in charge of setting up the connection are *connect* (on the client side) and *accept* (on the server side).

sgwrap models this connection mechanism thanks to different mailboxes. First, a *pairing* mailbox is used so that the server detects the connection from the client. The *pairing* mailbox name is based on the listening address of the server. The server *gets* on the pairing mailbox, while the client *puts* a pairing message on it. This pairing message contains a unique identifier generated by the client, that is used right away so that the server can acknowledge information to the client. This is done on an *acknowledgement* mailbox, whose name exactly corresponds to the unique identifier generated by the client. Finally, the information acknowledged by the server allows to create two final *transfer* mailboxes, that will be used to do the actual data transfers between the two sockets — one mailbox per direction of the connection.

Data transfer. Once a connection is established between a client and a server, the two sockets can exchange data with the complementary *send* and *recv* functions. Both functions take a *buffer* and a *size* as parameters. Usually, the system is in charge of caching the received data on a socket, so that successive calls to the *recv* function fill the user buffer with the remaining data. *sgwrap* intercepts these two functions and exchange data through the *transfer* mailboxes created in the connection stage.

As message transfers on mailboxes are based on *messages* rather than *bytes*, *sgwrap* stores the received message in a buffer to comply with the expected *recv* behavior.

Other functions Many other functions exist, *poll*, formerly *select*, and *addsockopt* that enables to tune the behavior of the communications. The function *poll* waits for one of the descriptors provided as a function parameter. *sgwrap* can model this behavior thanks to the function *wait_any* provided by Remote SimGrid. *wait_any* allows waiting for communications on an ensemble of mailboxes.

Accessing the real network. *sgwrap* intercepts all functions related to the network, it includes the function used by RSG that should not be intercepted to access the RPC servers (7.1). To cope with this issue, we benefit from the fact that when a network function is called with a socket unknown to *sgwrap*, the original function is called. The key is to deactivate *sgwrap* (this is done by removing environment variables configuring *sgwrap*) during the connections between the RSG client and RPC servers. Once the connection is up (one socket exists and is connected) and is unknown by *sgwrap*, we can activate *sgwrap* again to continue the interception and the simulation.

7.4.4 System Process Interception

This section describes how we use the RSG's simulation concepts to model the system process, and therefore being able to simulate them.

The *libc* features a set of functions to create, kill, and synchronize processes. New processes can be created by cloning an old process through the function *fork* [NL16]. A new process is creating and has the same — but isolated — state. When the system clones a process, the resources allocated to the process are not duplicated, instead, the resources are shared. This is true for the process's files, the signals handler, and for the virtual memory. However, the virtual memory is set into a copy-on-write state, which means that the memory is copied once a process tries to write it.

As the only way to create a new process is to clone an old one, the *libc* also features a set of functions to override the current process to start a new executable file — or a script file —, the *exec* family of functions. At the call of one of these functions, the process state is replaced by a new state corresponding to the new program, note that the shared resources, such as file descriptors, for instance, are not necessarily freed. One of the *execv* family functions is commonly used in conjunction with *fork*, to create a new process and then start a new program in the newly created process.

The function *pipe* creates a unidirectional data channel for communication between two processes. This function creates two file descriptors, one to send data, the second one dedicated to the reception.

Lastly, the *libc* provides a unique identifier to each process, called the *pid*. This information can be accessed from inside the process with the function *getpid*. Additionally, the function *getppid* enable a process to access its parent *pid*.

Modelling *fork*. With the simulation's concepts proposed by RSG, forking is similar to the creation of a new actor. However, to fully respect the forking process, the new actors need to have an isolated state. In order to be able to model the *fork* system call, RSG provides a function to create an actor into a newly forked process.

The function works as follows, first the current actor sends a message to the RSG server to create a new actor (using its RPC server.). When the RSG server receives this request, it creates a new SimGrid actor (containing an RPC server as depicted in Figure 7.1), and returns to the caller the id of the newly created actor. At this step, the RPC server of the new SimGrid actor is waiting for the remote actor to initiate the connection. Once the first request to create a new actor has returned, the remote original actor calls the real *fork* system's call. The initial process (also called the parent) send an acknowledgment to its RPC server, and returns from the function. The child process (the new remote actor), inherits the environment of its parent, and thus, is still connected to the RPC server of its parent actor. To cope with this issue, the child remote actor deletes the connection with its RPC server, and then initiate the connection with the new waiting RPC server. Note that, the child process does not close its parent connection because the RPC server would interpret that as the completion of the parent actor, and shut the connection down.

From *sgwrap* point of view, the fork is intercepted and has to handle two issues.

1. The first issue is as explained in section 7.2, that the *LD_PRELOAD* variable prevents RSG from using the real *libc* fork function, and instead the function will be recursively called indefinitely.

2. The second issue is related to the socket interception described in the previous section. During the RSG fork (described previously), the RSG client needs to use the network to establish a connection with its RPC server. As explained, at the end of the section 7.4.3, we disable the socket interception during the fork function. Once the remote actor is connected to the RSG server, we activate the network interception.

Modelling Exec Functions. On a call to *execv* function, the process will switch into a new state instantiating a new program. The current environment variables of the process remain untouched, and all active file descriptors stay open. However, after using *execv*, the RSG client of the actor will be destroyed (but the connection will remain active). *sgwrap* defines a *constructor* function (*__attribute__((constructor))*), which is a special function that is called when a shared library is loaded (during program startup). By setting the appropriate environment variables, *sgwrap* is able to connect back to the RSG connection of the actor during the call to the *constructor*.

Modelling PID and PPID. Within the RSG simulation, each actor has a unique ID. When *sgwrap* intercepts a call to *getpid*, the id of the actor is provided as PID. The PPID, is undefined by default (set to *-1*) if the actor is directly created by RSG. When a process uses the function *fork*, and therefore creates a child process, the PID of the parent is stored into an environment variable. Enabling a way for *sgwrap* to provide the PPID, when it intercepts a call to *getppid*.

Data transfer. The pipe interception (described earlier) is coupled with the socket interception (described in the previous section). This is mainly due to the fact that, once it is created, the functions used to transfer data within the pipe are the same as the function used for the sockets (*write/send* and *read/recv*).

At the interception of a call to the *pipe* function, *sgwrap* creates two new identifiers for the file descriptors and creates one mailbox associated with the two file descriptors. Once the pipe is created, when the process initiates a data transfer (with *send* or *receive* for instance), *sgwrap* intercept these calls and transfer the data through the RSG mailbox associated to the file descriptors.

7.4.5 Threads Interception

Threads can be seen as lightweight processes. A thread can be created by a process and is also scheduled by the system. Unlike the processes, all the threads of the same process share the same fundamental parts (same program, same virtual memory, same file descriptors, and same stack). Thus, using threads facilitates communication as everything is shared. However, the order of the execution of each thread is not controlled by the process and is subject to race conditions — two threads using the same variable at the same time.

To deal with the race conditions, the system provides different synchronization mechanisms accessible using the C library. One can cite, the mutexes and the condition variables, which are both intercepted by *sgwrap*.

- **A mutex** can be seen as a barrier used to protect the critical part of a program.
- **A condition variable (CV)** is another synchronization mechanism that enables a thread to wait for new events to happen.

Modelling Threads. *sgwrap* intercepts the calls to the function *pthread_create* to, instead of creating a thread, create a new RSG actor. At the creation of a new thread, the caller gives a function that will be executed by the new thread, *sgwrap* provides the thread function to RSG to be the actor main function. From *sgwrap*'s point of view, a thread is a new RSG actor. However, the creation of a new actor in RSG triggers the creation of a new thread, in this case, RSG needs to be able to use the original system's function *pthread_create*. Similarly, the *fork* function, this is done by using a global variable in *sgwrap*, to detect cases where the original function must

be called without being intercepted. Additionally, for the creation of a new remote actor, RSG needs to create a new socket without being intercepted, as for the *fork* function it is done by temporally removing the environment variable enabling the network interception.

Modelling Mutexes and Condition Variables. Modeling mutexes and condition variables are directly done using the RSG API. Indeed, as SimGrid directly features both synchronization mechanisms into its API to synchronize actors. The functions defined by SimGrid are thus implemented into RSG. This direct match enables *sgwrap* to call the appropriate function when a function call to the mutex or condition variable is intercepted.

7.5 Conclusion

The objective of this chapter is to separate the tools and mechanisms from the methodology developed and used to simulate RJMS. This separation enables to provide an in-depth description of the technical projects developed for this dissertation.

8 Reproducibility of Experiments with Variations

Reproducibility in computer science should be a main concern for the credibility of every scientific contributions.

But, what is the real purpose of reproducibility? It is the capitalisation on the work of all the people of the scientific community, to move forward scientific knowledge; so the science is not restarting from scratch every time that a scientist disappear. It is common that a project dies, because the only people that have essential unwritten knowledge are hit by a truck; or because it has been left unmaintained for too long [Hin19]. Because computer science is based on software, it is possible to avoid this issue with good practices in software engineering, like composability and good documentation. But, while the science evolves, simulations become more accurate, and the scientific software stack becomes more complex. Thus, even if the project is documented, going from source code to runnable experiment, i.e. the building process, can be tedious. Moreover, good software engineering is not recognized as it should in computer science research community. Thus, there is no incentive to take the time to do it right. So, reproducibility in computer science needs to be backed with new methodologies, good practices, and appropriate tools, to be able to build and transmit scientific knowledge more efficiently.

Reproducibility is a wide notion that needs to be specified. Feitelson [Fei15a] has defined a taxonomy of the different way to reproduce scientific results. In this taxonomy, The first level of reproduction is the **Repetition**, i.e. do exactly the same experimental process to obtain the same results. The second level, **Replication** is similar but the experience's input is changed.

Currently, most of the reproducibility tools only support these two levels by capturing the software environment. Indeed, software environment is hard to reproduce, and without it, it is impossible to run the experiment. Also, experiment software environment tightly depends on the Operating System (OS) distribution it was built on. It is sometimes impossible to install it on an other distribution because of inter dependency issues. One approach to solve this problem is to snapshot the software environment into an image. But, even if an image of the experiment runtime environment is provided by the original author, continuing his work requires more than just repeating the experiment; to be able to corroborate someone's approach, we not only need to rerun experiments, but we also need to modify them: test new

variations, add more parameters, and develop new features. This is the next level of reproducibility in the aforementioned taxonomy, called **Variation**.

Enabling scientists to reproduce an experiment with variation requires that the reproducer is able to rebuild experiment software with some modifications (even if the software is unmaintained and the tools necessary for building it, are long gone). It means that the reproducibility with variation can be achieved if the reproducer is able to reproduce not only the experiment "production" environment, but also the "development" environment which is necessary to modify the software. Moreover, when a variation of previous experiment produces new results, this experiment should also be reproducible.

In this context, we are proposing a new way of seeing reproducibility through the scientific software development lifecycle. Each step in this lifecycle requires a software environment. We define a software environment by a set of applications and libraries, with all their dependencies, and their configurations, required to achieve a step in a scientific workflow.

All the experiments of this dissertation have been designed with a special attention to their reproducibility. All software have been made publicly available. A.2 lists of the different repositories used for the work of this dissertation, which has been made following the methodology presented in this chapter.

This chapter has been made in collaboration with Michael MERCIER and Olivier RICHARD, and led to one publication [MFR18]. This work has also led to one tutorial[1] about creating reproducible experiment with Nix, the tutorial has been written in collaboration with Millian POQUET, and presented at Inria Rennes in 2019.

8.1 Software Development Workflow and Reproducibility

In computer science, a scientific workflow contains a software development lifecycle that starts by setting up a development environment with build tools and dependencies. Then, this environment is used to build a production environment that will, in turn, be used to run the actual experiment. But, software development is an iterative process: one can produce different versions of the production environment, or even modify the development environment to update or to add tools. This process is in the middle of the scientific workflow and all the software environments produced, for development and production should be captured to enable reproducibility. The Figure 8.1, exhibits that the first two levels of reproducibility can be achieved with

[1] https://nix-tutorial.gitlabpages.inria.fr/nix-tutorial/

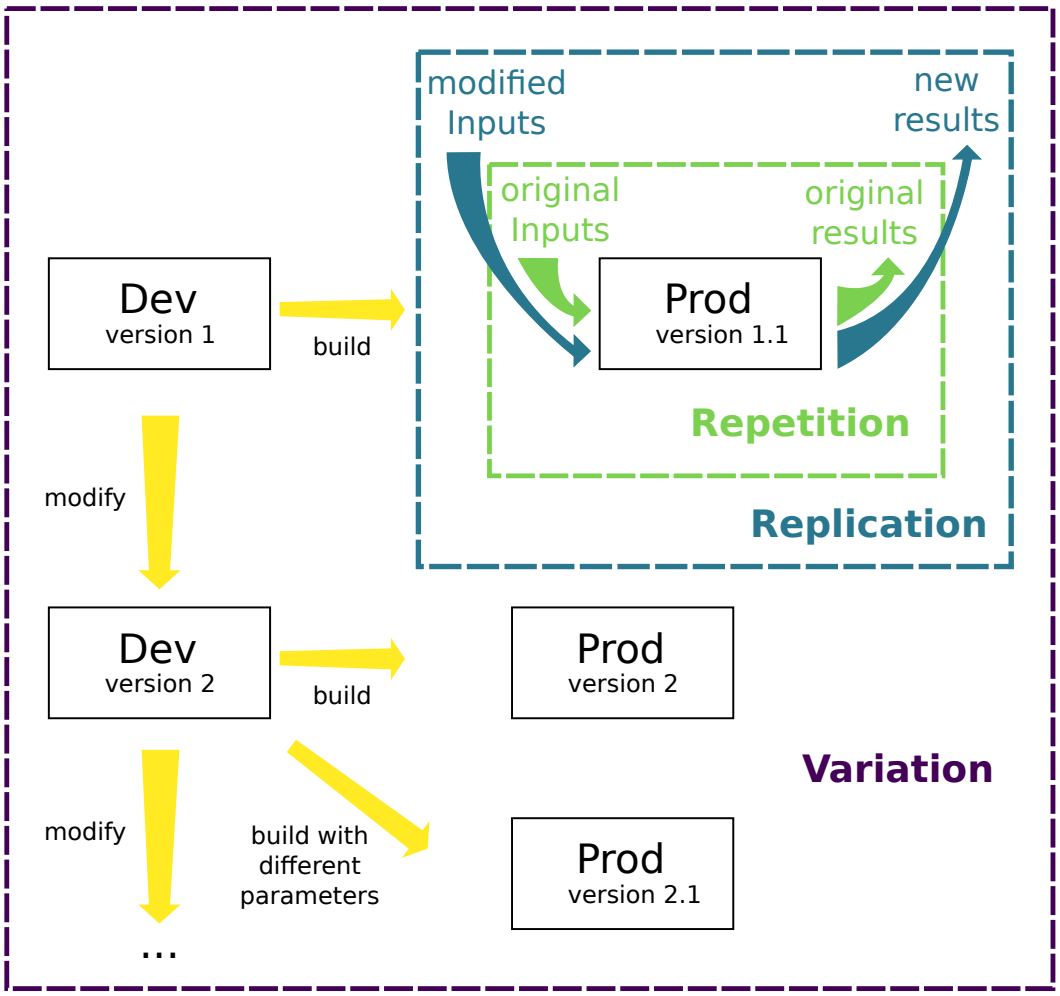


Figure 8.1.: The different level of reproducibility in regard to the development lifecycle: Variation requires to enclose the development environment and to provide a way to modify it while keeping reproducibility.

only one environment, but **reproducibility with variation requires taking into account the whole development process.**

To contribute to the scientific workflow, it has to be reproducible by itself: Thus, internal (e.g. colleague, intern) and external (e.g. scientist from other laboratory, reviewer) contributors have the capability to reproduce this workflow.

So, to provide this capability the development environment should be defined entirely, and every changes should be tracked. This statement also holds for others software environments involved in the authors' workflow, like the ones use for input data generation, or output data analysis.

8.2 Reproducible Software Environments with Nix

Courtès et al. [CW15] emphasize that functional package managers (FPM), like Guix and Nix, are good candidates to share complex and upgradable environment. In the following of this paper, we will focus on Nix, but most of the assertions also hold for Guix. The FPM are applying the concept of mathematical function to software packaging. Each software building process is described through a function. The dependencies are also functions that are given as inputs. This function, or package definition, allows to precisely describe a package: where and how to gather the source code, which commit to use, the dependencies and their versions, and finally how to build the package. When a package is built, the dependency graph is resolved by a lazy evaluation of the function parameters, and all the necessary piece of software are also built. The result of the evaluation of a package definition is called a derivation. A derivation is concretely a set of files that contains the results of the building process of the software, which placed on the special place that contains all the derivations: the store. Finally installing a software is simply exposing a derivation from the store through symbolic links. Nix packages are written in a functional Domain Specific Language (DSL). This ensures that each build is pure, i.e. it only depends on its inputs, and the same inputs give the exact same package even on a different machine.

To implement the workflow described in Section 8.1 with Nix, the critical feature is the capacity to create a software environment without virtualization. This feature is used to create an isolated environment for the package building process. An environment can be seen as a set of derivations and relies on the fact that an FPM can infer all the dependencies of a derivation, and only expose these dependencies on the specified environment. Installing the environment will expose to the user the packages described in the environment. Archiving an environment will extract the whole dependency tree of the environment and create a self-contained archive. The resulting tarball, also called a closure, contains every binary and file necessary to

run the packaged applications. Thus, the environment can be installed on another machine without any external download or building process.

To achieve each level of reproducibility with Nix, the first requirement is to create a package definition for each application and its dependencies. Thankfully, Nix is a very active community and more than 40000 applications are already packaged in the main repository called “nixpkgs”. It is also possible to maintain a private set of packages that import dependencies from “nixpkgs”.

The first level of reproducibility, the Repetition, can be achieved by providing to an external scientist the production closure, that can be installed to repeat the experiment. The environment could also contain all scripts to deploy and run the experiment.

The second level of reproducibility, the Replication, that consists of replaying an experiment while changing its input, have the same requirement as the repetition: Only the production environment is needed.

The third level of reproducibility, the Variation, is where using Nix is the most advantageous. Nix provides the interesting feature, called the “nix-shell”, permitting to enter the package build environment. Hence, packaging a software with Nix have the side effect to provide also the build environment for the users. Nix capacity to define software environment and software package in a unified way gives the scientist the ability to share a reproducible production environment, and the associated development environment, with a single definition.

8.3 Related Work

The Popper method, proposed by Jimenez et al. [Jim+17], describes a structural framework for dependencies and artifacts. They identified a generic workflow describing an experimental methodology, from source code to the final manuscript of a contribution. Our contribution is compatible with this approach, the popper method proposes a structural organization of the experiment, whereas our approach proposes to implements a part of this workflow using Nix.

Repeatability has been the focus of previous works on reproducibility. The platform presented in [Ric+15], has the ability to instantiate an experiment environment in their infrastructure from a previously captured environment. The approach is interesting as it provides a way for a scientist to repeat experiments that requires specific hardware. Our approaches could be complementary to cover both hardware and software to provide a higher level of reproducibility. Boettiger et al. [Boe15] survey how to use docker to do reproducibility, and also introduce the development environment. From our implementation with Nix, the docker approach shows similarities. However, Nix closure is more adapted than the Docker images for

application packaging because Docker provides an inappropriate level of abstraction: Docker is about constructing and configuring a complete OS, instead of declaring application dependencies.

Constructing reproducible experiment and workflow with an FPM has already been explored. In [Wur+18], they build a toolset upon the GUIX FPM, to facilitate the usage of bioinformatics common pipelines. They argue that using an FPM is a good foundation for reproducible computational experiment workflow.

The Blue Brain project [DDS15], is a big project, with a complex software stack, that aims to build a mammalian brain with a computer. In addition to a structured development workflow (using git, agile methodologies, code reviewing), they decided to package their workflow with Nix. They identified nine properties that are facilitated with Nix, from reproducibility to deployment and cross compilation. Their approach is based on internal needs and specific use case, whereas our contribution focuses on the role of the development lifecycle in the reproducibility, with Nix as a possible implementation.

8.4 Discussion

The proposed workflow does not consider input and output data of the experiment. One approach is to package the experiment data inside the production environment, i.e., the environment containing all the software and tools necessary during the experiment phase (the production phase). It is a viable solution for a small amount of data, but most of the experiments manage data separately with other tools.

The data related to the experiment is not only input and output data, the different piece software that are used to develop and run the experiment also have external inputs, like the source code of the experiment, the dependencies, the build tool binaries, and the configuration files, i.e., all the other artifacts necessary to build the pieces of software of an experiment. With Nix, it is easy to extract these artifacts for the software necessary during the experiment (the production environment). Nix is capable to export all the dependencies of an environment in a closure that can be imported on any machine where Nix is installed. This feature of Nix is very hard to achieve with other kinds of tools because of the lack of a clear dependency definition. However, even if it is straightforward for the production environment, how to extract this closure for the development environment (the environment containing the build tools) is unclear for now.

Capturing and archiving the environment closures is necessary for the Variation, for instance, the lifespan of Internet links is only a few months [Law+01]. Even if Nix is capable to rebuild everything from source, the source code repository can be unavailable, breaking the environment reproducibility. The problem that emerges is

that closures also have to be safely archived, versioned, and accessible for a long time period. A mid-term solution would be to store those closures using trusted centralized archives like Internet Archive[2] and Software Heritage[3].

Nix itself has limitations, his usage requires to understand the concepts behind the FPM and to learn the Nix language. Also, even if the Nix build system provides the framework to achieve reproducible build of packages, the bit-wise reproducibility [Lam+] depends on the way the software itself is built, i.e., patching the Makefile may be required.

In the case where the experiment results depend on the OS Kernel (e.g., performance evaluation) this approach is not sufficient. Indeed, Nix packaging handles the whole application software but not the OS Kernel. So, the proposed workflow needs to be supplemented with a building process definition of the entire OS — and not just the application layer — to be able to reconstruct a complete OS image with a variation. A Linux distribution based on Nix, called NixOS, is a good candidate.

When specific hardware is necessary to achieve reproducibility, an additional layer of control is needed. Testbeds like Grid'5000 [Bal+13], Chameleon [Kea+19], and Emulab [PSM10], are giving this level of control with the capability to create and deploy OS image on the fly on different hardware. Additionally, one also need to provide a description of the hardware used during the experiment, or the allocated resources.

8.5 Conclusion

The reproducibility with variations is the next level of reproducibility that the Computer Science community should aim at. The variations require to take into account the software development workflow, including the capability to modify and rebuild environments. The use of functional package managers is a promising approach. This kind of tool permits to achieve this next step to the reproducibility with variations, with a unified way to describe environments and packages, and a simple method to backup and to restore them.

[2] https://archive.org
[3] https://www.softwareheritage.org/

9 Conclusion

The role of the **R**esource and **J**ob **M**anagement **S**ystem (**RJMS**) is central to a cluster: It manages the resources and the jobs. Current RJMS needs to embrace both the evolution of the clusters and the jobs. The scheduling policy is the cornerstone of the RJMS, it decides when and where a job will be executed on the cluster. The production constraints of a computing cluster impose the scheduling policy to take efficient decisions in a short amount of time. Finding a good policy is challenging and has been the subject of numerous works during the past decade. Evaluating the relevance to use a scheduling policy in production from the previous study is challenging, it outlines the difference between the real world between the ideal world of the studies.

Chapter 2 presents the state of the art of this dissertation focusing on the tools and methodologies to conduct experiments with RJMS. It exists four different methodologies, the *emulation*, *in-vivo*, the *benchmarking* and the *simulation*. Each methodology illustrates if the experiment uses a model or the reality regarding the RJMS, and the platform. However, these four categories alone are not sufficient to categorize all the experiments done in the literature, because the limit between model and reality is not always well determined. Instead, several experiments mix approaches that rely on model and reality, for instance, simulating a cluster with virtual machines is both reality (the operating system) and model (the physical resources are virtual). For this reason we introduce the idea of **Hybridization** (fig. 2.1) mixing approaches based on reality and model.

9.1 Contributions and Future Work

The contribution of this dissertation is twofold: We present a new scheduling policy for HPC jobs, and through two different works we improve the tools used for experimenting with RJMS. More specifically, the first work focuses on creating a model for job scheduling simulation, and the second work proposes a new tool to *emulate* a distributed system, and therefore an RJMS.

9.1.1 Scheduling with Job Redirection

Chapter 3 introduces a new scheduling policy to schedule jobs with redirection. Redirected jobs are killed and restarted on a dedicated resources pool. The objective of this policy is to improve the slowdown metric, which is a measure of the system's reactivity form the user's perspective. Through a large experiment campaign of

simulations, the policy has been shown efficient for the slowdown metric, at the cost of increasing the waiting time.

Redirection is a practical extension of a former theoretical work studying rejection as a way to increase the scheduling performances [LST16]. This work aims to bridge the gap between theoretical and practical works, and shows that a theoretical idea can be adapted in a more practical environment using recent simulation techniques, and simulation models closer to a production system (parallel jobs on multicore clusters).

Despite the current evaluation, it is difficult to adapt the redirection in a production cluster as the simulations are not fully representative of a real cluster. For instance, the migration time of a redirected job is not simulated. Therefore it is difficult to evaluate the relevance of the redirection in a real setup. The remaining of this dissertation presents two different, yet complementary approaches for the evaluation of scheduling policies and RJMS. The first approach proposes to increase the realism of the scheduling simulation, and the second approach proposes new tools and methodologies to study real RJMSs.

Additionally to the future works presented in chapter 3's conclusion, the methodology developed in this work can be leveraged to conduct experiments more representative of a production cluster.

9.1.2 Simulation Model with Job Resources Consumption

Simulation has many advantages it is fast to execute on a single computer, it can represent a lot of different scenarios and it is easily reproducible. *Simulation*, by essence, strongly relies on models. Therefore, the ability of a simulation to accurately predict a scenario is dependant of the soundness of the underlying models. Current simulators for HPC scheduling use simple model, the model simulates the jobs as a fixed amount of time regarding their context of execution (the behavior of the job, the capacity of the underlying platform and the other jobs executing at the same time). However in production, the execution time of the jobs is dependant of external factors, such as their placement on the cluster or the other jobs using the same shared resources at the time (such as the interconnect or the parallel file system etc.). In this work, we propose to extend the scheduling simulation with job models that depends on their context of execution.

The first part of this work introduces and demonstrates the simulation capability of the Batsim simulator (chapter 4). Based on the SimGrid simulation toolkit, Batsim has the ability to simulate a platform and workloads containing models for the jobs. Batsim currently features three job models, the *delay*, the *Time Independent Traces (TiT)* and the *ptask* model. One difficulty to find model for scheduling simulation is

that a single simulation can embed million of jobs. The model must be simultaneously representative of a real job, and fast enough to keep reasonable simulation time.

Chapter 5 presents the second part of this work, it focuses on the evaluation of the *ptask* model. Among the presented models, the *ptask* model present two advantages: It is reasonably fast and has been created to represent parallel jobs. However, the model has not been evaluated for HPC jobs. The evaluation methodology used aims at evaluating the ability of the model to accurately predict the running time of an HPC application under (periodic) network interference. The methodology used compares the execution of the model (in simulation) with a real application (in a real cluster) under different network interferences. The results shows that the *ptask* model is able to reproduce the network interference of the HPC application without being reliably accurate regarding the predicted run time of the application.

As future works, we outline two research directions:

- **Model calibration.** In its current state, the *ptask* model shows lack of accuracy. Identifying the causes of the model's inaccuracy is difficult because SimGrid is a complicated software. One promising approach is to use the tracing capability of SimGrid— that enables to trace the network activity of the platform's network links during the simulation — in order to pinpoint the inaccuracy's cause. Unfortunately, this tracing capability is currently broken. With more insight on the model activity on the platform, it would be possible to add parameters to adjust the behavior of the model and increase its accuracy.

  Additionally, the *ptask* model makes the assumption that the application always uses 100 % of the most limited resource. In practice, this behavior is not necessarily true (or needs to be verified). One possible parameter to limit this effect could be to add the possibility to limit the resources the application can use, for instance, using 80 % of the total host speed instead of 100 %).

- **Extending the model's validation.** The next proposed direction is to extend the model's validation. In the presented evaluation, we generate synthetic interferences to stress the network during the application's execution. However, scheduling simulation involves the execution of multiple applications. The model's validation can be extended in three ways:

  1. Evaluating the model's accuracy with, instead of the synthetic network interferences, use another application. Future works in this direction replace the network interferences with other PDGEMM executions, to validate that the behavior of the model is consistent when multiple applications are involved.

  2. Secondly, it exists a large number of HPC applications. The in this work, we validate the *ptask* model against a single, simple, application.

PDGEMM is not fully representative of a real production application. More investigation with different HPC application is necessary to understand which applications can be simulated with a *ptask*. One lead, is to compare the *ptask* to the different application from the Nasa Parallel Benchmarks (NPB) [Bai+91], or to proxy applications.

3. Finally, with the emergence of data-intensive workloads, data management is an overgrowing concern for HPC centers [Asc+18]. In [Mer19] the author uses Batsim to simulate data-intensive jobs (Big Data jobs) along with HPC jobs. The main limitation of this work, is that the HPC jobs use a delay model (section 4.3), and therefore the impact of the data movements on HPC workloads is not observable. However, [Mer19] results on Big Data jobs show that Batsim's approach is promising to simulate data-intensive scenarios. Further validation on the data transfer models is necessary to increase our confidence in simulation regarding this aspect.

9.1.3 Experimenting with Real RJMS: *Simunix* and Batsky

Experimenting with real RJMS has the advantage of being closer to a production system than simulation. System administrators, for instance, can leverage these work to tune the configurations of a system, or developers can test new features. However, the tools and methodology used to set up an experiment with a real RJMS are expensive in time, and often complex.

In this work, we propose two approaches to use a real RJMS on a single computer.

- The first approach, *Simunix* (detailed in section 6.2), aims at reproducing the behavior of the operating system by intercepting the *libc*'s functions dedicated to communications and time control. *Simunix* leverages the Remote SimGrid (RSG) simulation toolbox to model the behavior of the computing platform. *Simunix* project has been rewritten as a generic library, called *sgwrap* (section 7.4). Hence, we propose *sgwrap* a library designed to simulate the C library features related to distributed applications (BSD sockets, threads and synchronization, and processes) and the functions related to time management. *sgwrap* uses a different approach: cut into several libraries, *sgwrap* can be used to intercept a subpart of the proposed features by only activating libraries handling a specified feature, for instance intercepting only time and threads.

  To support *Simunix*'s approach, we propose Remote SimGrid (detailed in section 7.3) that is a simulation toolbox presenting simulation concepts suitable for distributed applications. RSG is not dedicated to simulating RJMSs, instead,

it is intended to works for any distributed software. For instance, ongoing work from Millian POQUET focuses on the simulation of the OpenMPI runtime[1].

- The second approach detailed in section 6.3, called Batsky, takes a transversal approach: what if we only simulate the time during the experiments? The main idea of Batsky is to be able to execute the RJMS Slurm on an environment where the jobs and the time of the experiment are managed by a simulator. The main difficulty of Batsky is to be able to control the simulation time to accelerate the simulation, but to keep the system consistency. With its specific design, Batsky is able to dedicate control of the time and the execution of the jobs to an external simulator (Batsim), while executing a real RJMS. The current prototype is able to execute a workload of 4 jobs, on 10 simulated nodes in less than a minute.

As future works, we outline two research directions.

- **Remote SimGrid** is a promising tool to simulate real distributed applications. With the actor model featured by RSG, SimGrid has the necessary functionalities to simulate a distributed application. However, RSG lacks a terminated proof of concept. Currently, two projects have a significant advancement, the Simunix project that was able to simulate the RJMS Slurm with few nodes; and an OpenMPI plugin to perform the network communications of OpenMPI with RSG. Additionally, one distributed software could be a good candidate to be simulated with RSG: the CEPH storage system [@ceph].

  In the context of RJMS, the first project Simunix showed good promising results to simulate Slurm on a single computer. As future works, reproducing the approach with *sgwrap* is good to simulate Slurm with RSG. However, new development is necessary to complete the project.

- **Batsky: scalability, reproducibility and accuracy evaluation.** The approach taken by Batsky is new, and the current state of the project shows a proof a concept. However, studying the scalability of the approach is a key point to understand the scope of the studies that will be possible with Batsky. This can be done by using larger workloads and more simulated nodes. In a second time, evaluating (and improving) the accuracy and the reproducibility of Batsky is a necessary step to be able to use it as a reliable scientific tool.

9.1.4 Reproducibility of Experiments with Variation

Most computer science research relies on software, as it is the case for the different works in this dissertation. Some software has been designed and programmed for the occasion, others are their dependencies. Special attention was given to the

[1] https://framagit.org/simgrid/openmpi-rsg-plugins

reproducibility of the experiments done for this works. A general methodology has been extracted from the different works; leveraging the functional package manager (FPM) approach, we were able to extract a general methodology to package and design experiments involving computing programs. The methodology is based on packaging a program, but also the whole set of software needed by the different environments, such as its development environment or its execution environment. Furthermore, we identified that, with this methodology, we were able to propose the reproducibility with *variations*. *Variation* is the idea that experimenters can easily replay past experiments, but also modify either the inputs to conduct complementary experiments, or modify the code of the experiment to add new features or fix bugs.

As future works, we want to propose the **software environment transposition.** The variation is an interesting idea, and it is easy to implement with the functional package manager *Nix*. Reproducible software environments is a real asset for developers and experimenters. However, it doesn't provide information regarding the execution's environment, that is to say, the place in which the programs are executed. Nowadays, it is common to switch between different execution environments. More especially, to have a different environment for the development, the tests, and production. Each environment has different requirements: Development is often carried on a laptop, testing can be done a dedicated infrastructure, and finally, the final release is made in a dedicated platform (such as a Cloud or Fog computing). Furthermore, with the evolution of deployment methodologies, one execution environment can use containers, while others may use bare metal or traditional virtual machines. Future research in this area would involve the transpositions of one software environment from one execution environment to another.

Appendix

A

A.1 PDGEMM

A.1.1 Nonblocking Broadcast

The parallel task model makes strong assumption about the progress of the parallel applications, as it bundles the communication and the computations and adapts their speed so that every activity finish at the same time. This supposes that the real application has a communication and computation overlapping (the communications occur during the computation phases). The proposed PDGEMM algorithm uses the broadcast MPI collective which is a blocking call, the application stops until the broadcast is completed.

However, MPI features the collective IBroadcast (nonblocking broadcast) which is non blocking. Instead, this MPI function should starts the communication in background and return immediately providing a request object. The request object can be used to wait the end of the communication when it is needed.

Algorithm 5 shows a version of PDGEMM using nonblocking broadcast. First, at the iteration k the only requirement is that the data transfers needed for matrix products (line 23 to 31 of the Algorithm 3) need to be completed. Therefore it is possible at iteration k to initiate the transfer for the data needed at iteration $k+1$. In that way, the data transfers for iteration $k+1$ can occur during the computation phase of the k^{th} iteration. This is possible at one condition, the data for the first iteration needs to be already present on each node, so that we can compute the first iteration.

Algorithm 5 PDGEMM using nonblocking broadcast

procedure PDGEMM(id_ranks,$world\_size$,$matrix\_block_A$, $matrix\_block_B$)
  $q \leftarrow \sqrt{world\_size}$
  $row\_group \leftarrow id\_rank/q$ ▷ Row group of the process
  $col\_group \leftarrow id\_rank \mod q$ ▷ Column group of the process
  $group\_row\_id \leftarrow$ get_id(row_group) ▷ Process unique id in its row group
  $group\_col\_id \leftarrow$ get_id(col_group) ▷ Process unique id in its col group
  $res_C \leftarrow [block\_length \times block\_length]$
  $buffA_{product} \leftarrow [block\_length \times block\_length]$
  $buffB_{product} \leftarrow [block\_length \times block\_length]$
  $buffA_{bcast} \leftarrow [block\_length \times block\_length]$
  $buffB_{bcast} \leftarrow [block\_length \times block\_length]$
  if $group\_row\_id == root\_row$ **then**
    broadcast_on_lines($matrix\_block_A$, $source = True$)
  else
    broadcast_on_lines($buffA_{product}$, $source = False$)
  end if
  if $group\_col\_id == root\_col$ **then**
    broadcast_on_cols($matrix\_block_B$, $source = True$)
  else
    broadcast_on_cols($buffB_{product}$, $source = False$)
  end if
  for $i \leftarrow 1, i < \sqrt{world\_size} + 1, i++$ **do**
    $req \leftarrow$ Request[2] ▷ Table of two empty requests
    $root\_col\_prod \leftarrow root\_col$;
    $root\_row\_prod \leftarrow root\_row$;
    $root\_col \leftarrow k \mod q$;
    $root\_row \leftarrow k \mod q$;
    if $group\_row\_id == root\_row$ **then**
      $req[0] \leftarrow$ nonblocking_bcast_on_lines($matrix\_block_A$, $source = True$)
    else
      $req[0] \leftarrow$ nonblocking_bcast_on_lines($buffA_{bcast}$, $source = False$)
    end if
    if $group\_col\_id == root\_col$ **then**
      $req[1] \leftarrow$ nonblocking_bcast_on_cols($matrix\_block_B$, $source = True$)
    else
      $req[1] \leftarrow$ nonblocking_bcast_on_cols($buffB_{bcast}$, $source = False$)
    end if
    if $group\_row\_id == root\_col\_prod$ & $group\_col\_id = root\_row\_prod$ **then**
      $res_C \leftarrow res_C + matrix\_block_A * matrix\_block_B$
    else if $group\_col\_id == root\_col\_prod$ **then**
      $res_C \leftarrow res_C + buffA_{product} * matrix\_block_B$
    else if $group\_row\_id == root\_row\_prod$ **then**
      $res_C \leftarrow res_C + matrix\_block_A * buffB_{product}$
    else
      $res_C \leftarrow res_C + buffA_{product} * buffB_{product}$
    end if
    wait_all_requests(req)
    swap_buffers($buffA_{gemm}$, $buffA_{bcast}$)
    swap_buffers($buffA_{gemm}$, $buffA_{bcast}$)
  end for
end procedure

A.2 Reproduce Experiments

The experiments done for this dissertation are available online. The software used are packaged using the Nix package manager, as for the software execution environments.

- Chapter on Redirection 3: https://gitlab.inria.fr/adfaure/evipar. The experiments have been conducted thanks to the Grid'5000 infrastructures. The repository contains the sources to replay the experiments, and the data to analyze without replaying all the simulations.
- Chapter on Job profiles 4: https://gitlab.inria.fr/adfaure/ptask_tit_eval The repository contains the necessary source code, and Nix environments to replay all the simulations (including generating the Time Independent Traces with SimGrid) and the data analysis.
- Chapter on Ptask evaluation 5: https://gitlab.inria.fr/batsim/ptask-eval The experiments of this chapter contains two independent parts: The real executions and the simulation, and therefore several software environment are necessary. The environment for the real execution (that is deployed on the computing nodes of Grid'5000), the simulation environment containing the SimGrid simulation source code, and the data analysis environments.
- Presented in chapter 6, the Batsky project is available at: https://github.com/oar-team/arion-batsky The repository contains the necessary nix files to replay the experiments, and simulate Slurm (on 10 nodes) with Batsky.
- Chapter on tools for emulations 7 presents *sgwrap*, the source code can be found online at https://framagit.org/simgrid/sgwrap

List of Figures

1.1 Overview of a Resource and Job Management System. The users submit their applications, called the jobs, to the RJMS. The jobs scheduler finds a number of nodes available according to the job's requirements and attributes a starting date at each one of them. The resources manager manages the platform, monitors the nodes, and controls the job's executions 3

1.2 Exhaustive job definition. 4

1.3 Example of a Gantt chart. It depicts the result a of scheduling policy. The jobs are represented by the rectangles, the *x-axis* shows the time, while the *y-axis* shows the set of allocated resources. 5

1.4 Experiment methodology for RJMS using workload replay. The experiment's setup contains a module (or program) that submits jobs. The studied RJMS (or simulator) receives the jobs as if they were submitted by the users. Once all the jobs are submitted and scheduled, one can analyze the results in respect of the desired metric(s). 7

2.1 *Hybridization* mixes approaches based on reality and model. The blue clouds locate the contributions of this work according to the *hybridization*. The approaches using a model of RJMS are detailed in section 2.2. The approaches based on real or *hybridized* (left column) RJMS are detailed in section 2.3. Simulators with platform model is the focus of chapters 4 and 5, RSG and Batsky approaches are detailed in 6. Chapter 3 evaluates a new scheduling policy with a *Discrete Event Simulator*. 13

3.1 Division of the nodes of a cluster of m processors in respect of the input parameter α, with $0 \leqslant \alpha \leqslant 1$. Both group are managed by an independent scheduler. 30

3.2 **Lower is better** — For each of the 20 workloads extracted from Curie, we computed $BSLD_{avg}$ on the upper side and $BSLD_{max}$ on the midel line, the average waiting time ($WAIT_{avg}$) on the lower line. Each violin is created by 20 such workloads under a certain set of parameters. Each plot under the same line shows the same objectives, with different perspectives. On the left plots, the redirection threshold is in the x-axis and the y-axis represents the observed metric. Each column of the grid represents a different allocation percentage α. While on the right plots, the allocation percentage is in the x-axis, each column grid represents a different threshold. 36

3.3 **Lower is better, under horizontal line means that the redirection is more effective** — For each of the 20 workloads extracted from a cluster, we computed the ratio of $BSLD_{avg}$ without redirection over $BSLD_{avg}$ with redirection, and we did the same for the $BSLD_{max}$. Each box is induced by 20 such ratios. The black line is the median ratio and the red square the average ratio. The horizontal lines represent the quartiles. The figure presents the results for each cluster, the two upper figures deal with Curie, the middle one corresponds to Intrepid while the bottom one concerns Ricc. 38

3.4 **Lower is better, under horizontal line means that the redirection is more effective** — For each of the 20 workloads extracted from a cluster, we computed the ratio of $BSLD_{avg}$ and we did the same for the average waiting time. Each violin is induced by 20 such ratios. These plots have the same structure than Figure 3.3, but focus on the *waiting time* objective. Ratio of the average waiting time in blue (dark gray) $BSLD_{avg}$ in green (light gray) 39

4.1 *Ptask* communication matrix. The arrows show the direction of the communications. 47

4.2 Example of how the application progress is computed using the *ptask* model. At $t = 0$, the purple task (plain borders) uses the two nodes at their full capacity, while the link is used at $60\ \%$. Then, at $t = 1$, a new task arrives (dashed borders) performing a data transfer on the link, taking half the of the link's bandwidth. Hence, the link becomes the bottleneck of the purple task which has to adapt its load on the two hosts. Finally, at $t = 6$, the green task (dotted borders) arrives with some work on the two nodes, taking $50\ \%$ of the host's speed. 49

4.3 Batsim overview, the scheduler takes decisions from the network protocol. The simulation inputs are a platform description and a workload of jobs handled by Batsim. The simulated platform is handled by SimGrid. The results of a simulation comprises data about the simulation, such as jobs resources allocations and execution times. 51

4.4 Example of a simplified exchange between Batsim and the scheduler taking the scheduling decisions. The protocol is based on request-answer, Batsim waits for a response for each request (the grey, and white layers represent a request-answer phase). Batsim sends events to the scheduler, such as new job submissions, or job completions. The scheduler takes all scheduling decisions. 51

4.5 Network topology of the Graphene platform. The platform is organized in 4 irregular cabinets, connected to a top switch with a $10\ G$ Ethernet link. 56

4.6 Mean (the error bar are the standard deviation) of the (real world) duration for the simulation of the entire workload over 10 different executions. The figure on the right zooms over the delay and the *ptask* profiles. 57

4.7 The figure presents only the schedules for the workload containing 512 jobs. Each Gantt chart depicts the scheduling decisions taken by the scheduler process, the first two upper charts correspond to the profile of execution delay, followed by the *ptask* execution profile. The last two charts on the bottom are the Gantt charts obtained with the TiT profiles. 58

4.8 Histogram of the waiting times and execution times for the different profiles. *ft.C.8* takes 8 machines for its execution, *ft.C.16* takes 16 machines, and *ft.C.32* takes 32 machines. The histogram on the left shows the distribution of the running time per profile, the x-axis is the running time in seconds. The histogram on the right shows the distribution of the waiting times. The color shows the different scheduling algorithm used, forced contiguity in purple (bottom) , not forced contiguity in blue (upper histogram). 59

5.1 At iteration k, the processes on the k^{th} line broadcast their block to the other processes of their line for the matrix A. Once the fist broadcast is done, the processes on the k^{th} column broadcast their block to the other process of their column. 67

5.2 Final configuration for the experiment. Figure 5.2a depicts the physical setup of the nodes, while the fig. 5.2b shows the logical view we achieve. 69

5.3 *Tcpkali* evaluation (upper figure). Interference pattern examples (lower figure). 70

5.4 **Mean runtime of PDGEMM under different interference patterns and broadcasts (blocking or nonblocking).** 73

5.5 ***Paravance.*** Network traffic monitored on one MPI host, and on *tcpkali* server's host for **blocking broadcast**. The upper figure plots one instance for each interference pattern without subdivision, whereas the lower figure plots it with 50 subdivisions. 74

5.6 ***Paravance.*** Network traffic monitored on one MPI host, and on *tcpkali* server's host for **nonblocking broadcast**. The upper figure plots one instance for each interference pattern without subdivision, whereas the lower figure plots it with 50 subdivisions. 75
5.7 **Comparison of monitoring traces of *Grisou* and *Paravance*.** The plots show the configuration of PDGEMM with blocking broadcasts and 50 subdivisions. It appears that without interferences, *Paravance* and *Grisou* have similar executions, the network activities are alike and have the same execution times. However, under constant interferences (bottom line), the bandwidth of PDGEMM is significantly lower on *Paravance* than on *Grisou* and have a different execution time. 78
5.8 **Comparison of the *ptask* with *Paravance* and *Grisou* for configurations with 50 subdivisions**. The error bar are only present for *Paravance* and *Grisou*, they show the confidence intervals ($95\ \%$) of the mean runtime. The *15s interference / 15s idle* pattern has been removed to increase readability. 82
5.9 Progress of PDGEMM and *ptask* over the simulated time until its completion ($progress = 100$). 83
5.10 Model parameters. 85
5.11 Theoretical model's results for the different calibrations. The x-axis shows the different values of α, corresponding to the different interference patterns. 86
5.12 Illustration of how the *ptask* simulates a periodic application. The dotted blue line represent an example of a real activity. The solid purple line shows the activity of *ptask*'s simulating the blue line. 87

6.1 High-level illustration of a *Simunix* simulation involving two programs. The execution of the two programs are on the same computer, and therefore the SimGrid execution and the two programs share the same operating system. Each process is executed into a different sandbox, and the black arrows depict the function replacing the original *libc* function and interacting with SimGrid. The sandbox is permeable: the function that is not intercepted by the sandbox are executed normally by the OS, on the other hand, the intercepted functions use the SimGrid simulation. Note that to provide process isolation for each separated actors, the sandbox accesses the SimGrid simulation using the RSG project. 94
6.2 Example of usage of *Simunix* with Slurm. 96
6.3 Batsky overview. The *Batsky-Apdater* is the bridge between the simulation and the reality, it controls the time provided to *SlurmCtld*, and gathers information from the BatJob. The white arrows represent network traffic between the different components. 97

6.4 Sequence diagram of the simulation of a job with Batsky and Slurm. . 98

7.1 Overview of a simulation with Remote SimGrid. The simulation consists of 3 hosts and 4 actors. Actors are bound to hosts, representing *where* the actors' actions take place. Each actor is controllable remotely thanks to an RPC mechanism. A single actor is automatically spawned per user process by default. This is not limiting, as actors can spawn other actors — e.g., here, actors 1 and 2 are part of the same system process. . . . 107

7.2 Representation of the usage of *sgwrap* on a simple application composed of two processes. The left side shows how processes communicate usually — without *sgwrap*. On the right side, communications are simulated using Remote SimGrid. This is done by *sgwrap*, which intercepts calls to the BSD socket API and calls Remote SimGrid accordingly. . . . 110

8.1 The different level of reproducibility in regard to the development lifecycle: Variation requires to enclose the development environment and to provide a way to modify it while keeping reproducibility. 119

List of Tables

2.1 Experiment classes depending on the application and the platform (real or model) [GJQ09]. 11

2.2 Comparison of five open source simulators for the simulation of RJMS. 15

3.1 List of the clusters used for replay and their important characteristics. . 34

5.1 Mean runtime of all PDGEMM executions for each configuration (number of subdivisions, interference patterns and broadcast types) on ***Paravance***. The last column is the percentage of increase compared to the execution of the same category without interference. 72

5.2 Mean runtime of all PDGEMM executions for each configurations (number of subdivisions, interference patterns and broadcast types) on ***Grisou***. The last column is the percentage of increase compared to the execution of the same category without interference. 76

5.3 Predicted runtime for the *ptask* model under each interference pattern, and two methods for simulating *tcpkali*. The obtained results are identical for the two methods: Using a *ptask* or the SimGrid's *s4u* API doesn't impact the simulation's results. The last column is the percentage of increase compared to the execution of the same category without interference. 81

Bibliography

[@ceph] *ceph*. https://ceph.io/ (cit. on p. 129).

[@cwrap] *cwrap, Testing your full software stack on a single machine.* 2019 (cit. on p. 108).

[@elfhook] Redirecting functions in shared ELF libraries. *Redirecting functions in shared ELF libraries*. 2013 (cit. on p. 104).

[@evalys] *Evalys*. https://github.com/oar-team/evalys (cit. on p. 100).

[@kubernetes] *Kubernetes*. https://kubernetes.io/ (cit. on p. 100).

[@LIEF] *LIEF*. https://lief.quarkslab.com/doc/latest/Intro.html (cit. on p. 104).

[@rsg] *Remote SimGrid (git repository)*. 2019 (cit. on p. 106).

[@simtercept] *Remote SimGrid (git repository)*. 2019 (cit. on p. 110).

[AAC09] Jason Ansel, Kapil Arya, and Gene Cooperman. „DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop“. In: *2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)*. IEEE. Rome, Italy, 2009, pp. 1–12 (cit. on p. 108).

[AGZ94] Ramesh C. Agarwal, Fred G. Gustavson, and Mohammad Zubair. „A high-performance matrix-multiplication algorithm on a distributed-memory parallel computer, using overlapped communication“. In: *IBM Journal of Research and Development* 38.6 (1994), pp. 673–682 (cit. on p. 64).

[Ahn+20] Dong H. Ahn, Ned Bass, Albert Chu, et al. „Flux: Overcoming scheduling challenges for exascale workflows“. In: *Future Gener. Comput. Syst.* 110 (2020), pp. 202–213 (cit. on pp. 18, 25).

[Álv+17] Gonzalo Pedro Rodrigo Álvarez, Erik Elmroth, Per-Olov Östberg, and Lavanya Ramakrishnan. „Enabling Workflow-Aware Scheduling on HPC Systems“. In: *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2017, Washington, DC, USA, June 26-30, 2017*. Ed. by H. Howie Huang, Jon B. Weissman, Adriana Iamnitchi, and Alexandru Iosup. ACM, 2017, pp. 3–14 (cit. on p. 22).

[Ana+12] *True Elasticity in Multi-Tenant Data-Intensive Compute Clusters*. Oct. 2012 (cit. on p. 27).

[Asc+18] M. Asch, Terry Moore, Rosa M. Badia, et al. „Big data and extreme-scale computing“. In: *Int. J. High Perform. Comput. Appl.* 32.4 (2018), pp. 435–479 (cit. on pp. 2, 128).

[Bai+91] David H. Bailey, Eric Barszcz, John T. Barton, et al. „The Nas Parallel Benchmarks“. In: *Int. J. High Perform. Comput. Appl.* 5.3 (1991), pp. 63–73 (cit. on pp. 18, 55, 128).

[Bal+13] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, et al. „Adding Virtualization Capabilities to the Grid'5000 Testbed“. In: *Cloud Computing and Services Science*. Ed. by Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan. Vol. 367. Communications in Computer and Information Science. Springer International Publishing, 2013, pp. 3–20 (cit. on pp. 17, 123).

[Bea+20] Olivier Beaumont, Louis-Claude Canon, Lionel Eyraud-Dubois, et al. „Scheduling on Two Types of Resources: A Survey“. In: *ACM Comput. Surv.* 53.3 (2020), 56:1–56:36 (cit. on p. 2).

[Béd+13] Paul Bédaride, Augustin Degomme, Stéphane Genaud, et al. „Toward Better Simulation of MPI Applications on Ethernet/TCP Networks“. In: *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation - 4th International Workshop, PMBS 2013, Denver, CO, USA, November 18, 2013. Revised Selected Papers*. Ed. by Stephen A. Jarvis, Steven A. Wright, and Simon D. Hammond. Vol. 8551. Lecture Notes in Computer Science. Springer, 2013, pp. 158–181 (cit. on p. 79).

[Bha+13] Abhinav Bhatele, Kathryn Mohror, Steve H. Langer, and Katherine E. Isaacs. „There goes the neighborhood: performance degradation due to nearby jobs“. In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*. Ed. by William Gropp and Satoshi Matsuoka. ACM, 2013, 41:1–41:12 (cit. on pp. 2, 8, 44).

[Boe15] Carl Boettiger. „An introduction to Docker for reproducible research“. In: *ACM SIGOPS Operating Systems Review* 49.1 (2015), pp. 71–79 (cit. on p. 121).

[Bro+18] Kevin A. Brown, Nikhil Jain, Satoshi Matsuoka, Martin Schulz, and Abhinav Bhatele. „Interference between I/O and MPI Traffic on Fat-tree Networks“. In: *Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018, Eugene, OR, USA, August 13-16, 2018*. ACM, 2018, 7:1–7:10 (cit. on p. 44).

[BSS13] Arun Balasubramanian, Alan Sussman, and Norman M. Sadeh. „Decentralized Preemptive Scheduling Across Heterogeneous Multi-core Grid Resources“. In: *Job Scheduling Strategies for Parallel Processing - 17th International Workshop, JSSPP 2013, Boston, MA, USA, May 24, 2013 Revised Selected Papers*. Ed. by Narayan Desai and Walfredo Cirne. Vol. 8429. Lecture Notes in Computer Science. Springer, 2013, pp. 22–41 (cit. on p. 27).

[Cam11] Pedro Antonio Madeira de Campos Velho. „Evaluation de précision et vitesse de simulation pour des systèmes de calcul distribué à large échelle. (Accurate and Fast Simulations of Large-Scale Distributed Computing Systems)“. PhD thesis. Grenoble Alpes University, France, 2011 (cit. on p. 46).

[Cap+05] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, et al. „A batch scheduler with high level components“. In: *5th International Symposium on Cluster Computing and the Grid (CCGrid 2005), 9-12 May, 2005, Cardiff, UK*. IEEE Computer Society, 2005, pp. 776–783 (cit. on p. 22).

[Cas+14] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. „Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms“. In: *Journal of Parallel and Distributed Computing* 74.10 (June 2014), pp. 2899–2917 (cit. on pp. 16, 19, 45, 46, 105).

[Cha+99] Steve J. Chapin, Walfredo Cirne, Dror G. Feitelson, et al. „Benchmarks and Standards for the Evaluation of Parallel Job Schedulers“. In: *Job Scheduling Strategies for Parallel Processing, IPPS/SPDP'99 Workshop, JSSPP'99, San Juan, Puerto Rico, April 16, 1999, Proceedings*. Ed. by Dror G. Feitelson and Larry Rudolph. Vol. 1659. Lecture Notes in Computer Science. Springer, 1999, pp. 67–90 (cit. on p. 15).

[Cho+13] Brian Cho, Muntasir Rahman, Tej Chajed, et al. „Natjam: Design and Evaluation of Eviction Policies for Supporting Priorities and Deadlines in Mapreduce Clusters“. In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. Santa Clara, California: ACM, 2013, 6:1–6:17 (cit. on p. 27).

[CLH19] Tom Cornebize, Arnaud Legrand, and Franz C. Heinrich. „Fast and Faithful Performance Prediction of MPI Applications: the HPL Case Study“. In: *2019 IEEE International Conference on Cluster Computing, CLUSTER 2019, Albuquerque, NM, USA, September 23-26, 2019*. 2019, pp. 1–11 (cit. on p. 79).

[Cru+19] Eduardo H. M. Cruz, Matthias Diener, Laércio Lima Pilla, and Philippe O. A. Navaux. „EagerMap: A Task Mapping Algorithm to Improve Communication and Load Balancing in Clusters of Multicore Systems“. In: *TOPC* 5.4 (2019), 17:1–17:24 (cit. on pp. 22, 43).

[CW15] Ludovic Courtès and Ricardo Wurmus. „Reproducible and User-Controlled Software Environments in HPC with Guix“. In: *Euro-Par 2015: Parallel Processing Workshops*. Ed. by Sascha Hunold, Alexandru Costan, Domingo Giménez, et al. Cham: Springer International Publishing, 2015, pp. 579–591 (cit. on p. 120).

[DDS15] Adrien Devresse, Fabien Delalondre, and Felix Schürmann. „Nix based fully automated workflows and ecosystem to guarantee scientific result reproducibility across software environments and systems“. In: *Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering*. ACM. 2015, pp. 25–31 (cit. on p. 122).

[Deg+17] Augustin Degomme, Arnaud Legrand, George S. Markomanolis, et al. „Simulating MPI Applications: The SMPI Approach“. In: *IEEE Trans. Parallel Distrib. Syst.* 28.8 (2017), pp. 2387–2400 (cit. on p. 47).

[Des+11] Frederic Desprez, George S. Markomanolis, Martin Quinson, and Frédéric Suter. „Assessing the Performance of MPI Applications through Time-Independent Trace Replay“. In: *2011 International Conference on Parallel Processing Workshops, ICPPW 2011, Taipei, Taiwan, Sept. 13-16, 2011*. Ed. by Jang-Ping Sheu and Cho-Li Wang. IEEE Computer Society, 2011, pp. 467–476 (cit. on p. 47).

[Don+11] Jack Dongarra, Pete Beckman, Terry Moore, et al. „The International Exascale Software Project roadmap“. In: *The International Journal of High Performance Computing Applications* 25.1 (Jan. 2011), pp. 3–60 (cit. on pp. 1, 25).

[Dut+16] Pierre-François Dutot, Michael Mercier, Millian Poquet, and Olivier Richard. „Batsim: A Realistic Language-Independent Resources and Jobs Management Systems Simulator“. In: *Job Scheduling Strategies for Parallel Processing - 19th and 20th International Workshops, JSSPP 2015, Hyderabad, India, May 26, 2015 and JSSPP 2016, Chicago, IL, USA, May 27, 2016, Revised Selected Papers*. Ed. by Narayan Desai and Walfredo Cirne. Vol. 10353. Lecture Notes in Computer Science. Springer, 2016, pp. 178–197 (cit. on pp. 12, 14, 22).

[Eme13] Joseph Emeras. „Workload Traces Analysis and Replay in Large Scale Distributed Systems“. Theses. Université de Grenoble, Oct. 2013 (cit. on pp. 2, 7, 43).

[Fau+20] Adrien Faure, Giorgio Lucarelli, Olivier Richard, and Denis Trystram. „Online Scheduling with Redirection for Parallel Jobs“. In: *2020 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2020, New Orleans, LA, USA, May 18-22, 2020*. IEEE, 2020, pp. 326–329 (cit. on p. 26).

[FC07] Kayo Fujiwara and Henri Casanova. „Speed and accuracy of network simulation in the SimGrid framework“. In: *Proceedings of the 2nd International Conference on Performance Evaluation Methodolgies and Tools, VALUETOOLS 2007, Nantes, France, October 22-27, 2007*. Ed. by Peter W. Glynn. ACM International Conference Proceeding Series. ICST/ACM, 2007, p. 12 (cit. on p. 46).

[Fei01a] Dror G. Feitelson. „Metrics for Parallel Job Scheduling and Their Convergence“. In: *Job Scheduling Strategies for Parallel Processing*. Ed. by Dror G. Feitelson and Larry Rudolph. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 188–205 (cit. on pp. 6, 26).

[Fei01b] Dror G. Feitelson. „Metrics for Parallel Job Scheduling and Their Convergence“. In: *Job Scheduling Strategies for Parallel Processing, 7th International Workshop, JSSPP 2001, Cambridge, MA, USA, June 16, 2001, Revised Papers*. 2001, pp. 188–206 (cit. on p. 26).

[Fei15a] Dror G Feitelson. „From repeatability to reproducibility and corroboration“. In: *ACM SIGOPS Operating Systems Review* 49.1 (2015), pp. 3–11 (cit. on p. 117).

[Fei15b] Dror G. Feitelson. *Workload Modeling for Computer Systems Performance Evaluation*. Cambridge University Press, 2015 (cit. on pp. 7, 54).

[Fei19] Dror Feitelson. *Parallel Workloads Archive*. 2019 (cit. on pp. 7, 33).

[FOH87] Geoffrey C. Fox, Steve W. Otto, and Anthony J. G. Hey. „Matrix algorithms on a hypercube I: Matrix multiplication“. In: *Parallel Computing* 4.1 (1987), pp. 17–31 (cit. on p. 64).

[FPR18] Adrien Faure, Millian Poquet, and Olivier Richard. „Évaluation d'algorithmes d'ordonnancement par simulation réaliste“. working paper or preprint. Apr. 2018 (cit. on p. 45).

[Fu+15] Xianjin Fu, Zhenbang Chen, Yufeng Zhang, et al. „MPISE: Symbolic execution of MPI programs“. In: *2015 IEEE 16th International Symposium on High Assurance Systems Engineering*. IEEE. 2015, pp. 181–188 (cit. on p. 107).

[FW98] D. G. Feitelson and A. M. Weil. „Utilization and predictability in scheduling the IBM SP2 with backfilling“. In: *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*. Mar. 1998, pp. 542–546 (cit. on p. 28).

[Gal+20] Cristian Galleguillos, Zeynep Kiziltan, Alessio Netti, and Ricardo Soto. „AccaSim: a customizable workload management simulator for job dispatching research in HPC systems“. In: *Cluster Computing* 23.1 (2020), pp. 107–122 (cit. on pp. 8, 14, 44).

[Gau+15] Éric Gaussier, David Glesser, Valentin Reis, and Denis Trystram. „Improving backfilling by using machine learning to predict running times“. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*. 2015, 64:1–64:10 (cit. on p. 33).

[Gau+18] E. Gaussier, J. Lelong, V. Reis, and D. Trystram. „Online Tuning of EASY-Backfilling using Queue Reordering Policies“. In: *IEEE Transactions on Parallel and Distributed Systems* 29.10 (Oct. 2018), pp. 2304–2316 (cit. on p. 28).

[Geo+15] Yiannis Georgiou, David Glesser, Krzysztof Rzadca, and Denis Trystram. „A Scheduler-Level Incentive Mechanism for Energy Efficiency in HPC“. In: *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2015, Shenzhen, China, May 4-7, 2015*. IEEE Computer Society, 2015, pp. 617–626 (cit. on p. 19).

[Geo+17] Yiannis Georgiou, Emmanuel Jeannot, Guillaume Mercier, and Adèle Villiermet. „Topology-aware resource management for HPC applications“. In: *Proceedings of the 18th International Conference on Distributed Computing and Networking, Hyderabad, India, January 5-7, 2017*. ACM, 2017, p. 17 (cit. on p. 22).

[Geo10] Yiannis Ioannis Georgiou. „Contributions for Resource and Job Management in High Performance Computing“. PhD. Université de Grenoble, Nov. 2010 (cit. on p. 25).

[GF15] David Glesser and Adrien Faure. *Simunix, a large scale platform simulator*. Atos. Nov. 19, 2015. URL: https://slurm.schedmd.com/SLUG16/slug16_simunix.pdf (visited on Aug. 17, 2020) (cit. on pp. 94, 99).

[GH12] Yiannis Georgiou and Matthieu Hautreux. „Evaluating Scalability and Efficiency of the Resource and Job Management System on Large HPC Clusters“. In: *Job Scheduling Strategies for Parallel Processing, 16th International Workshop, JSSPP 2012, Shanghai, China, May 25, 2012. Revised Selected Papers*. Ed. by Walfredo Cirne, Narayan Desai, Eitan Frachtenberg, and Uwe Schwiegelshohn. Vol. 7698. Lecture Notes in Computer Science. Springer, 2012, pp. 134–156 (cit. on p. 20).

[GJQ09] Jens Gustedt, Emmanuel Jeannot, and Martin Quinson. „Experimental Methodologies for Large-Scale Systems: a Survey“. In: *Parallel Process. Lett.* 19.3 (2009), pp. 399–418 (cit. on p. 11).

[Gle16] David Glesser. „Road to exascale : improving scheduling performances and reducing energy consumption with the help of end-users“. PhD. Université Grenoble Alpes, Oct. 2016 (cit. on pp. 6, 25, 94).

[Hin19] Konrad Hinsen. „Dealing With Software Collapse“. In: *Comput. Sci. Eng.* 21.3 (2019), pp. 104–108 (cit. on pp. 9, 117).

[IH09] Teerawat Issariyakul and Ekram Hossain. „Introduction to Network Simulator 2 (NS2)“. In: *Introduction to Network Simulator NS2*. Boston, MA: Springer US, 2009, pp. 1–18 (cit. on p. 45).

[JDC18] Ana Jokanovic, Marco D'Amico, and Julita Corbalán. „Evaluating SLURM Simulator with Real-Machine SLURM and Vice Versa“. In: *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, PMBS@SC 2018, Dallas, TX, USA, November 12, 2018*. IEEE, 2018, pp. 72–82 (cit. on pp. 12, 21, 22, 92).

[Jim+17] Ivo Jimenez, Michael Sevilla, Noah Watkins, et al. „The popper convention: Making reproducible systems evaluation practical“. In: *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*. IEEE. 2017, pp. 1561–1570 (cit. on p. 121).

[JMT14] Emmanuel Jeannot, Guillaume Mercier, and Francois Tessier. „Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques“. In: *IEEE Trans. Parallel Distrib. Syst.* 25.4 (2014), pp. 993–1002 (cit. on p. 43).

[Kas19] Ayham Kassab. „Optimization of parallel scheduling and the use of renewable energy sources to power computing centers. (Optimisation de l'ordonnancement de calculs parallèles et de l'engagement de sources d'énergie renouvelable pour l'alimentation des centres de calcul)“. PhD thesis. University Bourgogne Franche-Comté, France, 2019 (cit. on p. 2).

[Kea+19] Kate Keahey, Joe Mambretti, Paul Ruth, and Dan Stanzione. „Chameleon: A Large-Scale, Deeply Reconfigurable Testbed for Computer Science Research“. In: *27th IEEE International Conference on Network Protocols, ICNP 2019, Chicago, IL, USA, October 8-10, 2019*. IEEE, 2019, pp. 1–2 (cit. on p. 123).

[KOB00] Jacques Chassin de Kergommeaux, Benhur de Oliveira Stein, and Pierre-Eric Bernard. „Pajé, an interactive visualization tool for tuning multi-threaded parallel applications“. In: *Parallel Comput.* 26.10 (2000), pp. 1253–1274 (cit. on p. 55).

[KP00] Bala Kalyanasundaram and Kirk Pruhs. „Speed is as powerful as clairvoyance“. In: *J. ACM* 47.4 (2000), pp. 617–643 (cit. on p. 25).

[KSS19] Dalibor Klusácek, Mehmet Soysal, and Frédéric Suter. „Alea - Complex Job Scheduling Simulator“. In: *Parallel Processing and Applied Mathematics - 13th International Conference, PPAM 2019, Bialystok, Poland, September 8-11, 2019, Revised Selected Papers, Part II*. Ed. by Roman Wyrzykowski, Ewa Deelman, Jack J. Dongarra, and Konrad Karczewski. Vol. 12044. Lecture Notes in Computer Science. Springer, 2019, pp. 217–229 (cit. on pp. 8, 14).

[Lam+] Chris Lamb, Holger Levsen, Mattia Rizzolo, and Vagrant Cascadian. *reproducible builds*. https://reproducible-builds.org/ (cit. on p. 123).

[Lar20] Théo Larue. „Development and evaluation of a Kubernetes cluster simulator based on Batsim“. MA thesis. Ensimag, Grenoble INP, Grenoble, France, 2020 (cit. on p. 100).

[Law+01] Steve Lawrence, David M Pennock, Gary William Flake, et al. „Persistence of web references in scientific research“. In: *Computer* 2 (2001), pp. 26–31 (cit. on p. 122).

[Leg15] Arnaud Legrand. *Scheduling for Large Scale Distributed Computing Systems: Approaches and Performance Evaluation Issues*. 2015 (cit. on pp. 20, 45, 46).

[LF03] Uri Lublin and Dror G. Feitelson. „The workload on parallel supercomputers: modeling the characteristics of rigid jobs“. In: *Journal of Parallel and Distributed Computing* 63.11 (2003), pp. 1105–1122 (cit. on p. 7).

[LMT17] G. Lucarelli, F. Mendonca, and D. Trystram. „A New On-line Method for Scheduling Independent Tasks“. In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. May 2017, pp. 140–149 (cit. on pp. 26, 29).

[LST16] Giorgio Lucarelli, Abhinav Srivastav, and Denis Trystram. „From Preemptive to Non-preemptive Scheduling Using Rejections“. In: *Computing and Combinatorics - 22nd International Conference, COCOON 2016, Ho Chi Minh City, Vietnam, August 2-4, 2016, Proceedings*. 2016, pp. 510–519 (cit. on pp. 29, 126).

[LTZ19] Arnaud Legrand, Denis Trystram, and Salah Zrigui. „Adapting Batch Scheduling to Workload Characteristics: What can we expect From Online Learning?“ In: *IPDPS 2019 - 33rd IEEE International Parallel & Distributed Processing Symposium*. rio de janeiro, Brazil: IEEE, May 2019, pp. 1–10 (cit. on p. 33).

[Luc+15] Giorgio Lucarelli, Fernando Machado Mendonca, Denis Trystram, and Frédéric Wagner. „Contiguity and Locality in Backfilling Scheduling“. In: *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2015, Shenzhen, China, May 4-7, 2015*. IEEE Computer Society, 2015, pp. 586–595 (cit. on p. 54).

[Luc+16] Giorgio Lucarelli, Nguyen Kim Thang, Abhinav Srivastav, and Denis Trystram. „Online Non-preemptive Scheduling in a Resource Augmentation Model based on Duality“. In: *European Symposium on Algorithms (ESA 2016)*. Vol. 57. 63. Aarhus, Denmark, Aug. 2016, pp. 1–17 (cit. on p. 30).

[Luc11] Alejandro Lucero. „Simulation of batch scheduling using real production-ready software tools“. In: *Proceedings of the 5th IBERGRID* (2011) (cit. on pp. 21, 92).

[Mer+17] Michael Mercier, David Glesser, Yiannis Georgiou, and Olivier Richard. „Big data and HPC collocation: Using HPC idle resources for Big Data analytics“. In: *2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017*. 2017, pp. 347–352 (cit. on pp. 17, 18).

[Mer19] Michael Mercier. „Contribution to High Performance Computing and Big Data Infrastructure Convergence“. 2019GREAM031. PhD thesis. 2019 (cit. on pp. 2, 27, 63, 128).

[MF01] A. W. Mu'alem and D. G. Feitelson. „Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling“. In: *IEEE Transactions on Parallel and Distributed Systems* 12.6 (June 2001), pp. 529–543 (cit. on pp. 27, 28).

[MFR18] Michael Mercier, Adrien Faure, and Olivier Richard. „Considering the Development Workflow to Achieve Reproducibility with Variation“. In: *SC 2018 - Workshop: ResCuE-HPC*. Dallas, United States, Nov. 2018, pp. 1–5 (cit. on p. 118).

[Mub+17] Misbah Mubarak, Christopher D. Carothers, Robert B. Ross, and Philip H. Carns. „Enabling Parallel Simulation of Large-Scale HPC Network Systems“. In: *IEEE Trans. Parallel Distrib. Syst.* 28.1 (2017), pp. 87–100 (cit. on p. 45).

[New+15] Chris Newcombe, Tim Rath, Fan Zhang, et al. „How Amazon Web Services Uses Formal Methods“. In: *Commun. ACM* 58.4 (Mar. 2015), pp. 66–73 (cit. on p. 107).

[NL16] Linus Nyman and Mikael Laakso. „Notes on the History of Fork and Join“. In: *IEEE Ann. Hist. Comput.* 38.3 (2016), pp. 84–87 (cit. on p. 112).

[Pol+18] Samuel D. Pollard, Nikhil Jain, Stephen Herbein, and Abhinav Bhatele. „Evaluation of an interference-free node allocation policy on fat-tree clusters“. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*. IEEE, 2018, 26:1–26:13 (cit. on pp. 20, 92).

[Poq17] Millian Poquet. „Simulation approach for resource management. (Approche par la simulation pour la gestion de ressources)“. PhD thesis. Grenoble Alpes University, France, 2017 (cit. on pp. 2, 14, 32, 45, 50).

[PSM10] Andres Perez-Garcia, Christos Siaterlis, and Marcelo Masera. „Designing Repeatable Experiments on an Emulab Testbed“. In: *Broadband Communications, Networks, and Systems - 7th International ICST Conference, BROADNETS 2010, Athens, Greece, October 25-27, 2010, Revised Selected Papers*. Ed. by Ioannis Tomkos, Christos Bouras, Georgios Ellinas, Panagiotis Demestichas, and Prasun Sinha. Vol. 66. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer, 2010, pp. 28–39 (cit. on p. 123).

[Pul09] Kevin Pulo. „Fun with LD_PRELOAD“. In: *linux. conf. au*. Vol. 153. 2009 (cit. on p. 103).

[Qia+17] Peixin Qiao, Xin Wang, Xu Yang, Yuping Fan, and Zhiling Lan. „Preliminary Interference Study About Job Placement and Routing Algorithms in the Fat-Tree Topology for HPC Applications“. In: *2017 IEEE International Conference on Cluster Computing, CLUSTER 2017, Honolulu, HI, USA, September 5-8, 2017*. IEEE Computer Society, 2017, pp. 641–642 (cit. on p. 2).

[RH10] George F. Riley and Thomas R. Henderson. „The ns-3 Network Simulator“. In: *Modeling and Tools for Network Simulation*. Ed. by Klaus Wehrle, Mesut Güneş, and James Gross. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 15–34 (cit. on p. 108).

[Ric+15] Robert Ricci, Gary Wong, Leigh Stoller, et al. „Apt: A platform for repeatable research in computer science“. In: *ACM SIGOPS Operating Systems Review* 49.1 (2015), pp. 100–107 (cit. on p. 121).

[Rod+17] Gonzalo P. Rodrigo, Erik Elmroth, Per-Olov Östberg, and Lavanya Ramakrishnan. „ScSF: A Scheduling Simulation Framework“. In: *Job Scheduling Strategies for Parallel Processing - 21st International Workshop, JSSPP 2017, Orlando, FL, USA, June 2, 2017, Revised Selected Papers*. Ed. by Dalibor Klusácek, Walfredo Cirne, and Narayan Desai. Vol. 10773. Lecture Notes in Computer Science. Springer, 2017, pp. 152–173 (cit. on pp. 21, 92).

[Sar+13] Luc Sarzyniec, Tomasz Buchert, Emmanuel Jeanvoine, and Lucas Nussbaum. „Design and Evaluation of a Virtual Experimental Environment for Distributed Systems“. In: *21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2013, Belfast, United Kingdom, February 27 - March 1, 2013*. IEEE Computer Society, 2013, pp. 172–179 (cit. on pp. 19, 92).

[Smi+18] Staci A. Smith, Clara E. Cromey, David K. Lowenthal, et al. „Mitigating inter-job interference using adaptive flow-aware routing“. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*. IEEE, 2018, 27:1–27:15 (cit. on pp. 2, 8).

[Sul+08] Anthony Sulistio, Uros Cibej, Srikumar Venugopal, Borut Robic, and Rajkumar Buyya. „A toolkit for modelling and simulating data Grids: an extension to GridSim“. In: *Concurr. Comput. Pract. Exp.* 20.13 (2008), pp. 1591–1609 (cit. on pp. 16, 19, 46).

[Tan+11] W. Tang, Z. Lan, N. Desai, D. Buettner, and Y. Yu. „Reducing Fragmentation on Torus-Connected Supercomputers“. In: *2011 IEEE International Parallel Distributed Processing Symposium*. May 2011, pp. 828–839 (cit. on p. 33).

[Taz+13] Hajime Tazaki, Frédéric Urbani, Emilio Mancini, et al. „Direct Code Execution: Revisiting Library OS Architecture for Reproducible Network Experiments“. In: *The 9th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*. Santa Barbara, United States, Dec. 2013 (cit. on p. 108).

[TEF05] Dan Tsafrir, Yoav Etsion, and Dror G. Feitelson. „Modeling User Runtime Estimates“. In: *Job Scheduling Strategies for Parallel Processing, 11th International Workshop, JSSPP 2005, Cambridge, MA, USA, June 19, 2005, Revised Selected Papers*. Ed. by Dror G. Feitelson, Eitan Frachtenberg, Larry Rudolph, and Uwe Schwiegelshohn. Vol. 3834. Lecture Notes in Computer Science. Springer, 2005, pp. 1–35 (cit. on p. 7).

[Ull75] J. D. Ullman. „NP-complete Scheduling Problems“. In: *J. Comput. Syst. Sci.* 10.3 (June 1975), pp. 384–393 (cit. on p. 27).

[Vel+13] Pedro Velho, Lucas Mello Schnorr, Henri Casanova, and Arnaud Legrand. „On the validity of flow-level tcp network models for grid and cloud simulations“. In: *ACM Trans. Model. Comput. Simul.* 23.4 (2013), 23:1–23:26 (cit. on p. 79).

[VL09] Pedro Velho and Arnaud Legrand. „Accuracy study and improvement of network simulation in the SimGrid framework“. In: *Proceedings of the 2nd International Conference on Simulation Tools and Techniques for Communications, Networks and Systems, SimuTools 2009, Rome, Italy, March 2-6, 2009*. Ed. by Olivier Dalle, Gabriel A. Wainer, L. Felipe Perrone, and Giovanni Stea. ICST/ACM, 2009, p. 13 (cit. on p. 46).

[Wan+14] Lei Wang, Jianfeng Zhan, Chunjie Luo, et al. „BigDataBench: A big data benchmark suite from internet services“. In: *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*. IEEE Computer Society, 2014, pp. 488–499 (cit. on p. 18).

[Wur+18] Ricardo Wurmus, Bora Uyar, Brendan Osberg, et al. „Reproducible genomics analysis pipelines with GNU Guix“. In: *bioRxiv* (2018). eprint: https://www.biorxiv.org/content/early/2018/04/21/298653.full.pdf (cit. on p. 122).

[ZF14] Netanel Zakay and Dror G. Feitelson. „Preserving User Behavior Characteristics in Trace-Based Simulation of Parallel Job Scheduling“. In: *IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems, MASCOTS 2014, Paris, France, September 9-11, 2014*. IEEE Computer Society, 2014, pp. 51–60 (cit. on pp. 12, 16).

Abstract

High-Performance Computing (HPC) provides the computational power dedicated to solving complex problems of our society. HPC computers are large scale and distributed infrastructures composed of several thousands of computing cores. The management of theses systems is left to unique software: the Resource and Job Management System (RJMS). The objective of the RJMS is multiple: Managing the physical infrastructure, and handling the user requests to access to the computing power. The scheduling algorithm is the cornerstone of the RJMS, it decides where and when the user's jobs will be executed. Scheduling is a difficult problem; to manage large scale platforms RJMS needs to dispose of efficient yet scalable scheduling heuristics Evaluating and testing new scheduling algorithms is crucial before releasing it in production. Any failure can have a dramatic impact on the HPC platform leading to wasted time, energy, and resources. The lack of a platform dedicated experiments and tests compels RJMS designers and HPC center's administrators to use different tools and methodologies to evaluate new algorithms.

In the first part of this dissertation, we present and evaluate a new scheduling heuristics with job redirection. The evaluation is done using a large simulation campaign, it results that by redirecting jobs can improve the efficiency of the scheduling. In the second part, we focus on and extend the tools and methodologies available to experiment with RJMS. This part is twofold: Firstly, we propose to extend scheduling simulations with job models to simulate network contention between jobs. Secondly, we propose new tools that enable experiment with production RJMS without the need for an HPC platform. This dissertation aims to broaden the experimental landscape of tools and methodologies to experiment with RJMS and therefore help the release in the production of new scheduling algorithms.

Résumé

Les superordinateurs sont des systèmes mutualisant la puissance de milliers de coeurs de calculs dédiés à la résolution des problèmes compliqués de notre société. Le gestionnaire de ressources est un système distribué et complexe chargé de la gestion de ses ressources de calculs. Son rôle est multiple : Gérer la plateforme physique et traiter les requêtes d'accès des utilisateurs au superordinateur. La pierre angulaire du gestionnaire de ressources est son algorithme d'ordonnancement des requêtes des utilisateurs. L'ordonnancement est un problème difficile ; pour gérer efficacement un superordinateur le gestionnaire de ressources doit disposer d'heuristiques d'ordonnancement efficaces permettant de prendre des décisions pertinentes sur des milliers de ressources de calculs. Évaluer et tester de nouvelles heuristiques est fondamental avant de pouvoir les utiliser dans un système en production. Toute panne induite par une nouvelle politique peut avoir des conséquences importantes sur la qualité de service du superordinateur. Il est ainsi nécessaire de disposer d'outils et méthodes dédiés à l'évaluation des algorithmes d'ordonnancement.

La première partie de ce document présente un nouvel algorithm d'ordonnancement, ainsi que son évaluation par le biais de la simulation. L'algorithme en question repose sur la possibilité de rediriger les programmes des utilisateurs en cours d'exécution. L'évaluation est réalisée par le biais d'une large campagne de simulation, et montre que rediriger des programmes permet d'améliorer les performances de l'ordonnancement. L'objectif principal de la seconde partie de ce document est de proposer et développer de nouveaux outils et méthodes pour l'évaluation des gestionnaires de ressources. Cette seconde partie est elle même divisée en deux arcs : Nous proposons dans un premier temps d'étendre les techniques de simulations d'algorithmes d'ordonnancement avec des modèles dédiés aux programmes permettant ainsi la simulation d'interférences réseaux entre les différents programmes. Dans un second temps, nous proposons deux nouvelles approches pour créer des expériences sur un seul ordinateur, en se basant directement sur de vrais gestionnaires de ressources. L'objectif de ces travaux est d'étendre le paysage expérimental des outils et méthodologies nécessaires à l'évaluation de nouveaux algorithmes d'ordonnancement.