HAL
open science

A Reproducible Research Methodology for Designing and Conducting Faithful Simulations of Dynamic HPC Applications

Luka Stanisic

To cite this version:

Luka Stanisic. A Reproducible Research Methodology for Designing and Conducting Faithful Simulations of Dynamic HPC Applications. Distributed, Parallel, and Cluster Computing [cs.DC]. Université Grenoble Alpes, 2015. English. NNT : 2015GREAM035 . tel-01248109v2

HAL Id: tel-01248109

https://theses.hal.science/tel-01248109v2

Submitted on 21 Jan 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Présentée par

Luka STANISIC

Thèse dirigée par **Jean-François MÉHAUT**
et codirigée par **Arnaud LEGRAND**

Préparée au sein du **LIG, Laboratoire d'Informatique de Grenoble**
et de l'**École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

A Reproducible Research Methodology for Designing and Conducting Faithful Simulations of Dynamic Task-based Scientific Applications

Thèse soutenue publiquement le **30 octobre 2015**,
devant le jury composé de :

M. Martin QUINSON
Full Professor, École Normale Supérieure de Rennes, Président
Mrs. Sherry X. LI
Senior Scientist, Lawrence Berkeley National Laboratory, Rapporteur
M. Raymond NAMYST
Full Professor, Université de Bordeaux/Laboratoire Bordelais de Recherche en Informatique, Rapporteur
M. Grigori FURSIN
Research Scientist, Inria Saclay, Examinateur
M. Jean-François MÉHAUT
Full Professor, Université de Grenoble/Laboratoire d'Informatique de Grenoble, Directeur de thèse
M. Arnaud LEGRAND
Research scientist, CNRS/Laboratoire d'Informatique de Grenoble, Co-Directeur de thèse

Acknowledgements

First I would like to thank the members of the jury for accepting to be the part of this committee. They provided me with very useful remarks, questions and suggestions for the future directions of this work.

I would like to thank my family and friends who supported me and who came to the thesis defense from far away. I also thank my friends from Grenoble, especially Ben and Nata, who made this 3 years of PhD one of the most wonderful periods of my life.

I will never forget the incredibly pleasant and motivating surrounding, created by all members and former members of MESCAL, MOAIS, NANOSIM/CORSE teams that I worked with. I would particularly like to thank Brice VIDEAU, who though me many technical and research related things and who was regularly encouraging me. A special thanks goes to Jean-François MÉHAUT, who was always there when needed the most and whose advises helped me to always stay on the right track.

Finally and foremost, I would like to thank Arnaud LEGRAND whose incredible energy, character and knowledge inspired and guided me from the beginning until the end of the thesis. I feel extremely privileged to get so much of his time and to work beside him throughout these years.

Abstract

The evolution of High-Performance Computing systems has taken a sharp turn in the last decade. Due to the enormous energy consumption of modern platforms, miniaturization and frequency scaling of processors have reached a limit. The energy constraints has forced hardware manufacturers to develop alternative computer architecture solutions in order to manage answering the ever-growing need of performance imposed by the scientists and the society. However, efficiently programming such diversity of platforms and fully exploiting the potentials of the numerous different resources they offer is extremely challenging. The previously dominant trend for designing high performance applications, which was based on large monolithic codes offering many optimization opportunities, has thus become more and more difficult to apply since implementing and maintaining such complex codes is very difficult. Therefore, application developers increasingly consider modular approaches and dynamic application executions. A popular approach is to implement the application at a high level independently of the hardware architecture as Directed Acyclic Graphs of tasks, each task corresponding to carefully optimized computation kernels for each architecture. A runtime system can then be used to dynamically schedule those tasks on the different computing resources.

Developing such solutions and ensuring their good performance on a wide range of setups is however very challenging. Due to the high complexity of the hardware, to the duration variability of the operations performed on a machine and to the dynamic scheduling of the tasks, the application executions are non-deterministic and the performance evaluation of such systems is extremely difficult. Therefore, there is a definite need for systematic and reproducible methods for conducting such research as well as reliable performance evaluation techniques for studying these complex systems.

In this thesis, we show that it is possible to perform a clean, coherent, reproducible study, using simulation, of dynamic HPC applications. We propose a unique workflow based on two well-known and widely-used tools, Git and Org-mode, for conducting a reproducible experimental research. This simple workflow allows for pragmatically addressing issues such as provenance tracking and data analysis replication. Our contribution to the performance evaluation of dynamic HPC applications consists in the design and validation of a coarse-grain hybrid simulation/emulation of StarPU, a dynamic task-based runtime for hybrid architectures, over SimGrid, a versatile simulator for distributed systems. We present how this tool can achieve faithful performance predictions of native executions on a wide range of heterogeneous machines and for two different classes of programs, dense and sparse linear algebra applications, that are a good representative of the real scientific applications.

Résumé

Le calcul à hautes performances s'est vu contraint d'évoluer de façon radicalement différente durant la dernière décennie. La miniaturisation et l'augmentation de la fréquence des processeurs a atteint ses limites en raison des consommations d'énergies déraisonnables induites. Cette contrainte énergétique a conduit les fabricants de matériel à développer de nombreuses architectures alternatives afin de répondre aux besoins toujours croissants de puissance de calcul de la communauté scientifique. Cependant, programmer efficacement des plates-formes aussi diverses et exploiter l'intégralité des ressources qu'elles offrent s'avère extrêmement difficile. L'approche classique de conception d'application haute performance consistant à se reposer sur des codes monolithiques offrant de nombreuses opportunités d'optimisation est ainsi devenue de plus en plus difficile à appliquer en raison de difficulté d'implémentation, de portabilité et de maintenance. Par conséquent, les développeurs de telles applications considèrent de plus en plus couramment des approches modulaires et une exécution dynamique des différents composants. Une approche populaire consiste à implémenter ces applications à relativement haut niveau, indépendamment de l'architecture matérielle, en se reposant sur un paradigme basé sur la notion de graphe de tâches où chaque tâche correspond à un noyau de calcul soigneusement optimisé pour chaque architecture cible. Un système de runtime peut alors ensuite être utilisé pour ordonnancer dynamiquement ces tâches sur les différentes ressources de calcul à disposition.

Garantir l'efficacité de telles applications sur un large spectre de configurations reste néanmoins un défi majeur. En effet, en raison de la grande complexité du matériel, de la variabilité des temps d'exécution des calculs et de la dynamicité d'ordonnancement des tâches, l'exécution des applications n'est pas déterministe et l'évaluation de la performance de ces systèmes est très délicate. Par conséquent, il est nécessaire de disposer d'une méthodologie systématique, rigoureuse et reproductible pour conduire de telles études et évaluer la performance de tels systèmes.

Dans cette thèse, nous montrons qu'il est possible d'étudier les performances de telles applications dynamiques à l'aide de simulations, et ce de façon fiable, cohérente et reproductible. Nous proposons dans un premier temps une méthode de travail originale basée sur deux outils couramment utilisés dans notre communauté, Git et Org-mode, et permettant de mettre en oeuvre une recherche expérimentale reproductible. Cette approche simple permet de résoudre de façon pragmatique des problèmes tels que le suivi de la provenance des expériences ou la réplication de l'analyse des données expérimentales. Dans un second temps, nous contribuons à l'évaluation de performance d'applications dynamiques en concevant et en validant une simulation/émulation hybride à gros grains de StarPU, un runtime dynamique utilisant le paradigme de graphes de tâches et particulièrement adapté à l'exploitation d'architecture hybrides. Cette simulation est réalisée à l'aide de SimGrid, un simulateur polyvalent de systèmes distribués. Nous présentons comment notre approche permet d'obtenir des prédictions de performances d'exécutions réelles fiables sur un large panel de machines hétérogènes. Nous appliquons notre approche à deux classes de programmes différentes, les applications d'algèbre linéaire dense et creuse, qui sont représentatives d'un grand nombre d'applications scientifiques.

Contents

1 Introduction **1**
1.1 Contributions . . . 2
1.1.1 Methodology for conducting reproducible research . . . 4
1.1.2 Simulating dynamic HPC applications . . . 5
1.2 Thesis organization . . . 6

2 Background **7**
2.1 Programming challenges for HPC application developers . . . 7
2.1.1 HPC applications . . . 7
2.1.2 Different architectures used in HPC . . . 8
2.1.3 Exploiting machine resources . . . 9
2.1.4 Dynamic task-based runtimes . . . 9
2.1.5 Linear algebra applications . . . 12
Dense linear algebra . . . 12
Sparse linear algebra . . . 13
2.2 Experimental challenges for HPC application developers . . . 13
2.3 Conclusion . . . 15

3 Related Work **17**
3.1 Reproducible research . . . 17
Code and data accessibility . . . 19
Platform accessibility . . . 19
Setting up environments . . . 20
Conducting experiments . . . 20
Provenance tracking . . . 20
Documenting . . . 20
Extendability . . . 20
Replicable analysis . . . 21
Conclusion . . . 21
3.2 Performance evaluation and simulation . . . 21
3.2.1 Different simulation approaches . . . 21
Emulation . . . 21
Cycle-accurate simulation . . . 22
Coarse-grain simulation . . . 22
Hybrid approaches . . . 23
3.2.2 Simulating resources . . . 23
Modeling communications . . . 23
Modeling CPU . . . 24
Modeling GPU . . . 25
3.2.3 Simulating applications . . . 25
Simulating MPI applications . . . 25
Simulating task-based runtimes . . . 27

3.2.4 SimGrid: a toolkit for Simulating Large Heterogeneous Systems 28

4 Methodology **31**
4.1 A Git and Org-mode based workflow 33
4.1.1 Git branching structure 34
4.1.2 Using Org-mode for improving reproducible research 35
Environment capture 35
Laboratory notebook 36
Using literate programming for conducting experiments 37
4.1.3 Git workflow in action 38
Managing experiments with Git 38
Reproducing experiments 40
Fixing code 41
Making transversal analysis 41
Writing reproducible articles 41
4.2 Publishing results 42
4.2.1 The partially opened approach with figshare hosting 43
4.2.2 The completely open approach with public Git hosting 43
4.3 Conclusion 43

5 Porting StarPU over SimGrid **45**
5.1 Choosing a runtime candidate for simulation 45
5.2 Porting StarPU over SimGrid 46
5.3 Modeling the StarPU runtime 47
5.3.1 Synchronization 47
5.3.2 Memory allocations 47
5.3.3 Submission of data transfers 47
5.3.4 Scheduling overhead 49
5.3.5 Duration of runtime operations 49
5.4 Modeling communication 49
5.4.1 Different PCI bus models 50
Fatpipe model 51
Complete graph model 51
Treelike model 51
5.4.2 Model based on calibration 51
5.5 Modeling computation 52
Modeling parameter dependent kernels 54
Modeling kernels with complex codes 55
Limitations due to the simplistic machine models 55
5.6 Conclusion 55

6 Performance Prediction of Dense Linear Algebra Applications **57**
6.1 Experimental settings 57
6.1.1 Applications 57
6.1.2 Machines 58
6.2 Modeling kernel variability 59
6.2.1 Analyzing kernel duration distributions 60
6.2.2 Using histograms to approximate distributions 63
6.3 Evaluation methodology 65
6.4 Where the model needs to be carefully adapted 66
6.4.1 GPU memory limit 66
6.4.2 Specific GPUs/CUDA version 68
6.4.3 Elaborated communication model for complex machines 68
6.5 Accurate performance predictions for hybrid machines 69

6.6 Using both CPUs and GPUs for computation . . . 69
6.7 Where the model breaks and is harder to adapt: NUMA machines . . . 72
6.8 Typical studies enabled by such approach . . . 74
6.8.1 Studying schedulers . . . 74
6.8.2 Studying hypothetical platforms . . . 75

7 Performance Prediction of Sparse Linear Algebra Applications 77
7.1 `qr_mumps`, a task-based multifrontal solver . . . 77
7.2 Porting `qr_mumps` on top of SimGrid . . . 79
7.3 Experimental settings . . . 80
7.4 Modeling `qr_mumps` kernels . . . 80
7.4.1 Simple negligible kernels . . . 82
7.4.2 Parameter dependent kernels . . . 82
Simulation . . . 85
7.4.3 Matrix dependent kernels . . . 85
7.4.4 Accounting for kernels variability . . . 86
7.5 Evaluation methodology . . . 86
7.6 Simulation quality evaluation . . . 88
7.6.1 Evaluation on the Fourmi machine . . . 88
7.6.2 Evaluation on the Riri machine . . . 90
7.7 Typical studies enabled by such approach . . . 92
7.7.1 Memory consumption . . . 92
7.7.2 Extrapolation . . . 93

8 Conclusion and Future Work 95
8.1 Methodology for conducting reproducible research . . . 95
Current limitations and future work . . . 96
8.2 Simulating dynamic HPC applications . . . 97
8.2.1 Current limitations . . . 98
Modeling memory distance . . . 98
Modeling contention . . . 98
Simulation of sub-optimal native executions . . . 99
Model universality . . . 99
8.2.2 Future work . . . 101
Scaling to larger platforms . . . 101
Controlling the simulation quality . . . 102
Opening new horizons . . . 102

A References 105
A.1 Publications . . . 105
A.1.1 International peer reviewed journals . . . 105
A.1.2 International peer reviewed conference proceedings . . . 105
A.1.3 Short communications in conferences and workshops . . . 105
A.1.4 Master Thesis . . . 105
A.2 Bibliography . . . 106

B Additional figures 119

List of Figures

1.1 Diagram illustrating contributions (in gray) of our work that are related to methodology and performance evaluation. In this thesis, for the simulation aspect of our work we consider solely the case where applications rely on a runtime. Our workflow for doing reproducible research is however completely general. 3

2.1 An example of DAG: task graph of the tiled Cholesky factorization of 5×5 matrix with the block dimension 960, implemented in StarPU. 10
2.2 Comparing different scheduling algorithms of MUMPS and SuperLU sparse linear solvers as done by Amestoy et al. in [ADLL01]. MUMPS has very irregular patterns, which makes it a good candidate for using dynamic task-based runtime systems. . 14

3.1 Ideally, the experimenter would keep track of the whole set of decisions taken to conduct its research as well as all the code used to both conduct experiments and perform the analysis. Figure inspired by Roger D. Peng et al. [Rog09]. 18

4.1 Reproducibility issue on ARM Snowball: 4 consecutive experiments with identical input parameters behaving differently; 42 repetitions for each array size depicted by boxplots show no noise within each single experiment. 32
4.2 Proposed Git branching scheme with 4 different types of branches. 34
4.3 Typical Git experimentation workflow with different phases, using two machines. . 39
4.4 Restart or reproduce experiments starting from a certain commit. 40
4.5 Handling source modifications that occurred during the experimentation. 41

5.1 Implementing the simulation mode requires some code modifications of StarPU. This simple example illustrates the SimGrid (lines 2-14) and Native (lines 16-25) modes for executing StarPU's conditional wait. 48
5.2 Communication and topology modeling alternatives. In the crude modeling, a single link is used and communications do not interfere with each others. The pragmatic complete graph modeling allows for accounting for both the heterogeneity of communications and the global bandwidth limitation imposed by the PCI bus. Complex machine architecture demand for even more elaborated models. 50
5.3 Excerpt of the Idgraf platform description file generated using the treelike model. This fragment defines the route used when communicating between GPU6 and GPU2 using CUDA. 52
5.4 Architecture of the Idgraf machine with 12 cores, distributed on 2 NUMA nodes, 8 GPUs and their interconnect. 53

6.1 Tiled Cholesky factorization. 58
6.2 For dense linear algebra applications, most of the processing power is provided by GPUs. These plots depict the performance of the Cholesky application on the Mirage machine (see Table 6.1). A clearer view of these performance when restricting to CPU resources (8 cores) is provided in Figure 6.17 (4+4 cores). 59

6.3 Analysis of the kernel duration distribution as done by Haugen et al. in [HKY+14]. A normal law approximates the sample distribution very accurately. However we believe this is valid only for simple multi-core CPUs. 60
6.4 GEMM kernel durations on a GPU in a single $72,000 \times 72,000$ Cholesky factorization, presented as a time sequence. Most of the values are around 2.84 microseconds, but there is a significant number of higher durations as well. 61
6.5 Distribution of GEMM kernel durations on a GPU in a single $72,000 \times 72,000$ Cholesky factorization. Top plot presents the distribution constructed for all observations, while bottom ones are reconstucted for two separate groups of observation depending on their duration value. 62
6.6 Approximating GEMM duration with two types of histograms. 63
6.7 Initial results of StarPU simulation for the simplistic use cases were already very accurate. More complex scenarious required more sophisticated models. 67
6.8 Illustrating the influence of modeling runtime. Careless modeling of runtime may be perfectly harmless in some cases, it turns out to be misleading in others. 67
6.9 Transfer time of 3,600 KB using cudaMemcpy2D depending on the pitch of the matrix. 68
6.10 Performance of the LU application on Hannibal (QuadroFX5800 GPUs) using different modeling assumptions. 68
6.11 Simulating machines with complex architectures such as Idgraf (see Figure 5.4) require more elaborated models. 69
6.12 Checking predictive capability of our simulator in a wide range of settings. 70
6.13 Comparing execution traces (native execution on top vs. simulated execution at the bottom) of the Cholesky application with a $72,000 \times 72,000$ matrix on the Conan machine but using only GPU resources for processing the application. 71
6.14 Illustrating simulation accuracy for Cholesky application using different resources of the Mirage machine. 71
6.15 Comparing execution traces (native execution on top vs. simulated execution at the bottom) of the Cholesky application with a $72,000 \times 72,000$ matrix on the Mirage machine using 8 cores and 3 GPUs as workers. Adding 8 cores, improved the performance by approximatelly 20% compared to the performances obtained in Figure 6.13 71
6.16 Comparing execution traces (native execution on top vs. simulated execution at the bottom) of the Cholesky application with a $48,000 \times 48,000$ matrix on the Mirage machine using 8 cores and 3 GPUs as workers. Executing kernels on CPUs is much longer since Intel MKL libraries were not used, however simulation predictions are still very precise. 72
6.17 Illustrating the impact of deployment when using 8 cores on two NUMA nodes on the Mirage machine. 73
6.18 Simulation predictions of Cholesky application with a $32,000 \times 32,000$ matrix (block size 320×320) on large NUMA Idchire machine are precise for a small number of cores, but scale badly. The reason is that the memory is shared, while models are not taking into account various NUMA effects. 73
6.19 Cholesky on Attila: studying the impact of different schedulers. 75

7.1 Sequential version (*left*) and corresponding STF version from [ABGL14] (*right*) of the multifrontal QR factorization with 1D partitioning of frontal matrices. 78
7.2 Typical elimination tree: each node corresponds to a front and the resulting tree is traversed from the bottom to the top. To reduce the overhead incurred by managing a large number of fronts, subtrees are pruned and aggregated into optimized sequential tasks (Do_subtree) depicted in gray. 78
7.3 Processing a front requires a complexe series of Panel and Update tasks induced by the staircase structure. The dependencies between these operations expressed by the STF code leads to a fine-grain DAG dynamically scheduled by the runtime system. 78

7.4 Distribution of the `qr_mumps` kernel duration when factorizing the *e18* matrix (see Table 7.2). The distribution shapes is similar for other matrices. Most kernels have a (difficult to model) multi-modal distribution. 81
7.5 Duration of the `Panel` kernel as a time sequence for the *e18* matrix. The patterns suggest that this duration depends on specific parameters that evolve throughout the execution of the application. 81
7.6 Analysing linear model for `Panel` kernel. 84
7.7 Automatically generated code for computing the duration of `Panel` and `Update` kernels. 85
7.8 Makespans on the 8 CPU cores Fourmi machine for 10 different matrices. Native results on 3 largest matrices are not presented, because they are too long, since the factorization exceeds RAM memory capacities of the Fourmi machine. 88
7.9 Gantt chart comparison on the 8 CPU cores Fourmi machine. 88
7.10 Comparing kernel distribution duration on 8 CPU cores Fourmi machine. 89
7.11 Comparing `Panel` as a time sequence on 8 CPU cores Fourmi machine. Color is related to the task id. 90
7.12 Results on the Riri machine using 10 or 40 CPU cores. When using a single node (10 cores), the results match relatively well although not as well as for the Fourmi machine due to a more complex and packed processor architecture. When using 4 nodes (40 cores), the results are still within a reasonnable bound despite the NUMA effects. 91
7.13 Memory consumption evolution. The blue and red parts correspond to the `Do_subtree` and `Activate` contribution. 92
7.14 Extrapolating results for *e18* matrix on 100 and 400 CPU cores. 93

8.1 Elimination tree for *cat_ ears_ 4_ 4* matrix, rotated for 90 degrees to fit the page. The graph is extremely badly balanced and has a comb-like structure with a huge number of `Do_subtree` kernels presented as grey nodes. 100

B.1 Script for running experiments and automatically capturing meta-data. 120
B.2 Output of the execution containing both meta-data and the experiment results. . 121
B.3 Documentation part of the laboratory notebook. 122
B.4 Notes about all experimentation results stored in laboratory notebook. 123
B.5 Conducting experiments directly from the laboratory notebook. 124
B.6 Org-mode article contains both text and the analysis code. 125

List of Tables

2.1 Ranked first on the TOP500 list, illustrating the diversity of architectures used to construct modern HPC platforms. 7

5.1 Typical duration of runtime operations. 49

6.1 Machines used for the dense linear algebra experiments. 59

7.1 Machines used for the sparse linear algebra experiments. 80
7.2 Matrices used for the sparse linear algebra experiments. 80
7.3 Linear Regression of `Panel` kernel. 83
7.4 Linear Regression of `Update` kernel. 85
7.5 Summary of the modeling of each kernel based on the *e18* matrix on Fourmi (see Section 7.3 for more details). 86

Chapter 1

Introduction

Computers have become an indispensable research tool in many scientific fields such as physics, medicine, engineering, etc. These machines can execute trillions of operations per second and therefore perform computations that are far beyond from what humans could manually do. Still, even such tremendous computational power is insufficient for solving certain computational problems. Programs performing simulations in particle physics, earthquakes or astronomy all study very large systems and have thus to perform vast amount of computation on huge data. High-Performance Computing (HPC) is a computer science discipline that focuses on these groups of applications as well as on the platforms required to execute them.

High power machines, often called *supercomputers*, went through an extremely rapid evolution in the past 50 years. Thanks to the miniaturization and frequency scaling of the microchips, the performance of the computers doubled approximately every 18 months. However, this trend stopped a decade ago due to the technological limitations, namely energy consumption and heat. In order to pursue performance growth, manufacturers started to produce computers with multiple cores per processor and started to add the accelerator units. This allowed for breaking the problems into smaller ones that can be executed in parallel. Therefore, modern HPC machines comprise thousands to millions of cores, interconnected by fast networks.

The HPC community maintains the list of the fastest 500 computers in the world: the TOP500 [Top]. This list ranks the computers by the achieved maximum number of FLOPS (FLoating-point Operations Per Second) measured with the LINPACK benchmark [Don88]. Today's fastest supercomputer is "Tianhe-2" in China with 3,120,000 processor cores and it reaches a peak performance of approximately 33 PetaFlops (10^{16} floating-point operations per second). If the previous trends continue, it is expected that 1 ExaFlops (10^{18} floating-point operations per second) will be reached in 2020.

The major challenge for achieving such a high performance is energy constraint. Supercomputers and data centers already consume as much electricity as a small city and the price for powering them for a few years is the same as the initial price of the hardware that it is composed of. Therefore, it is estimated that the power budget for the future fastest computers should not exceed 20 MegaWatts. This requires performance efficiency of 50 GFLOPS/Watt, which is far above the current maximum of 5,2 GFLOPS/Watt achieved by the L-CSC computer [Greb]. Hence, there is a huge energy-efficient computing movement in HPC that established a new supercomputer ranking. The Green500 list [Grea] is organized similarly to the TOP500, except that computers are ranked according to their energy efficiency.

The large diversity of hardware architectures and their respective complexity, on both TOP500 and Green500 lists, clearly indicate that no consensus has still been reached on the architecture of the future supercomputers. For example, the top three machines on the Green500 list all use differently designed accelerators produced by different vendors [Greb]. The technology is evolving extremely fast and it is hard to predict what kind of resources will be available to the applications developers.

Implementing codes whose performance is portable across such diverse and complex platforms

becomes extremely challenging. Having hundred times more processor cores rarely provides hundred times faster execution due to the overheads of parallelization, communication and critical path limits of the application. Such scaling issues are hard to overcome, as exploiting efficiently all resources provided by modern computer platforms is not trivial.

Until a few years ago, the dominant trend for designing HPC libraries mainly consisted of designing scientific software as a single whole that aims to cope with both the algorithmic and architectural needs. This approach may indeed lead to extremely high performance because the developer has the opportunity to optimize all parts of the code, from the high level design of the algorithm down to low level technical details such as data movements. However, such a development often requires a tremendous effort, and is very difficult to maintain. Achieving portable and scalable performance has thus become extremely challenging, especially for irregular codes.

There is a recent and general trend in using instead a modular approach where numerical algorithms are written at a high level independently of the hardware architecture as Directed Acyclic Graphs (DAG) of tasks. A task-based runtime system then dynamically schedules the resulting DAG on the different computing resources, automatically taking care of data movement and taking into account the possible speed heterogeneity and variability. In such a way, runtimes abstract the underlying architecture complexity through a common application program interface (API).

However, dynamic task-based runtimes are hard to develop and the evaluation of their performance raises two major challenges. First, these complex runtimes aim to support diverse platform and parameter configurations. Therefore, to ensure that they achieve good performance on a wide range of settings, numerous experiments need to be executed. Second, these runtimes use dynamic scheduling techniques, which leads to non-deterministic executions. From one execution to another, tasks are not executed in the same order or on the same resources, leading to different makespans, which makes performance evaluation and comparison even more difficult than in classical deterministic settings. Furthermore, this non-determinism often brings heisenbugs that are very hard to locate.

Hence, there is a huge need for a reliable experimental methodology allowing to produce reproducible results to evaluate the correctness of the execution and the good performance of the runtimes on a wide range of settings. A possible solution is to conduct such experimental studies using simulation, which can address the aforementioned issues.

In this thesis, we propose a sound methodology for experimentally evaluating the performance of HPC application implemented on top of dynamic task-based runtimes, using faithful simulation predictions. In our approach, the target high-end machines are calibrated only once to derive sound performance models. These models can then be used at will to quickly predict and study in a reproducible way the performance of resource-demanding parallel applications using solely a commodity laptop. This allows for obtaining performance predictions of classical linear algebra kernels accurate within a few percents and in a matter of seconds, which allows both runtime and application designers to quickly decide which optimization to enable or whether it is worth investing in additional hardware components or not. Moreover, this allows for conducting robust and extensive scheduling studies in a controlled environment whose characteristics are very close to real platforms while having a reproducible behavior.

1.1 Contributions

Figure 1.1 (left part) recaps the previously described system, where an application can be executed either directly on the operating system (OS) or on top of a runtime. The goal of our research was however not to directly contribute to the development of any of these blocks, but to propose a good methodology and performance evaluation techniques to study them (right part of Figure 1.1).

In such a complex, multi-layer systems, it is impossible to master every part. Therefore, during our research we closely collaborated with domain experts, most notably with:

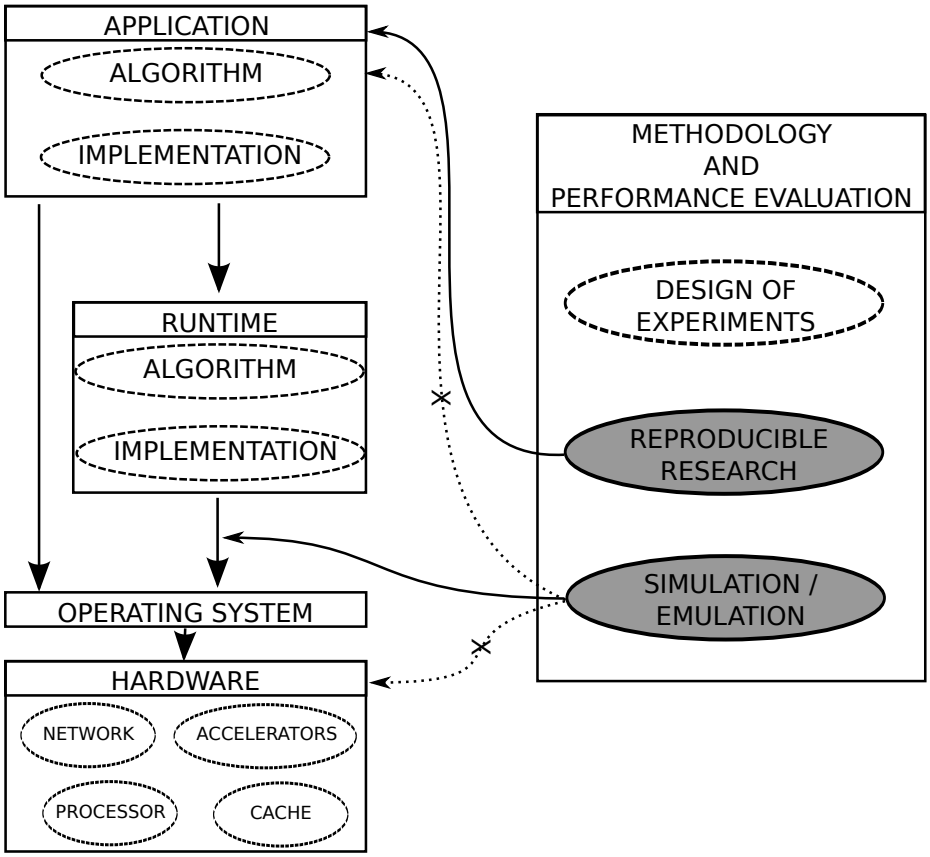


Figure 1.1: Diagram illustrating contributions (in gray) of our work that are related to methodology and performance evaluation. In this thesis, for the simulation aspect of our work we consider solely the case where applications rely on a runtime. Our workflow for doing reproducible research is however completely general.

1. Application developers: Emmanuel Agullo, Suraj Kumar, Paul Renaud-Goud, Abdou Guermouche, Alfredo Buttari, and Florent Lopez.
2. Runtime developers: Samuel Thibault, and Marc Sergent.
3. Operating systems and computer architecture experts: Brice Videau, Jean-François Méhaut, Augustin Degomme and numerous other researchers from teams MESCAL, MOAIS, NANOSIM, CORSE and TIMA in Grenoble as well as experts from ARM and the Barcelona Supercomputing Center.

Even though we studied several aspects in the area of methodology and performance evaluation, the main contributions of this thesis are twofold. First, we developed a unique workflow for doing **reproducible research** on a daily basis when conducting empirical studies on modern computer architectures. Second, we crafted a faithful **simulation of a dynamic runtime system** that aids at its evaluation.

1.1.1 Methodology for conducting reproducible research

Computers, operating systems and software running on it have reached such a level of complexity that it has become very difficult (not to say impossible) to control them perfectly and to know every detail about their configuration and operation mode. Consequently, it becomes less and less reasonable to consider computer systems as deterministic. Experiments that focus on measuring the execution time or the performance of the application running on such setups have thus become non replicable by essence. Hence, there is an urgent need for sound experimental methodology.

We inspired on other tools and approaches developed in the last few years in the domain of reproducible research to develop a lightweight experimental workflow based on standard tools widely used in our community. We used such workflow extensively throughout three years of research, and we strongly believe that our scientific work greatly benefited from it. This workflow was published in:

- [1] L. Stanisic, A. Legrand, and V. Danjean. An effective git and org-mode based workflow for reproducible research. *ACM SIGOPS Operating Systems Review*, 49:61 – 70, 2015. Special Topic: Repeatability and Sharing of Experimental Artifacts.

- [4] L. Stanisic and A. Legrand. Effective reproducible research with org-mode and git. In *1st International Workshop on Reproducibility in Parallel Computing*, Porto, Portugal, Aug. 2014.

- [7] L. Stanisic, and A. Legrand. *Actes du 10ème Atelier en Évaluation de Performances*, chapter Good practices for reproducible research, pages 29–30. Inria, 2014.

Moreover, we presented our solution on numerous occasions in order to encourage researchers from our community to redefine their practices and develop similar workflows that will aid them in their daily work. The most important events where our approach was presented are:

- LIG day on trace production in Grenoble, March 2015.
- Plafrim day on performance of parallel codes in Bordeaux, December 2014.
- REPPAR workshop on reproducibility in parallel computing in Porto, August 2014.
- AEP workshop on performance evaluation in Nice, June 2014.
- Join Laboratory for Petascale Computing (JLPC) summer school on performance metrics, modeling and simulation of large HPC systems in Nice, June 2014.

- SyncFree European project meeting in Paris, May 2014.
- COMPAS conference in Neuchâtel, April 2014.
- ANR SONGS plenary meetings in Lyon, June 2013, and in Nice, January 2014.

Additionally, our approach served as a base for a platform sharing HPC application traces, which is a joint project between researchers in Grenoble and Bordeaux [Traa].

Finally, we made a regular usage of a laboratory notebook comprising information about all the experiments we conducted during the last three years. We opened this laboratory notebook and made it publicly available so that anyone can inspect and possibly build upon our results [SSW].

1.1.2 Simulating dynamic HPC applications

Runtime systems are a promising approach for efficiently exploiting the heterogeneous resources offered by modern computers. However, their development and optimal utilization is not easy to achieve, and it requires constant experimental validation on a wide range of different setups. Evaluating the performance of such systems is extremely challenging due to the diversity of the experimental machines as well as the complexity of the code of the both application and the runtime. Moreover, for the runtimes that use dynamic scheduling, the executions are non-deterministic which makes evaluation even more difficult.

To address these issues, we developed a coarse-grain simulation tool. Our solution can be executed quickly and on a commodity machine, in order to evaluate the performance of the long runtime execution on a large, hybrid clusters. Additionally, the simulation provides reproducible results, which makes debugging of both the code and the performance much easier. Our tool provides very accurate performance predictions for applications running on top of dynamic task-based runtime and it was presented in the following papers:

- [3] L. Stanisic, E. Agullo, A. Buttari, A. Guermouche, A. Legrand, F. Lopez, and B. Videau. Fast and Accurate Simulation of Multithreaded Sparse Linear Algebra Solvers. 2015. Submitted to the ICPADS conference.

- [2] L. Stanisic, S. Thibault, A. Legrand, B. Videau, and J.-F. Méhaut. Faithful Performance Prediction of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures. *Concurrency and Computation: Practice and Experience*, page 16, May 2015.

- [5] L. Stanisic, S. Thibault, A. Legrand, B. Videau, and J.-F. Méhaut. Modeling and simulation of a dynamic task-based runtime system for heterogeneous multi-core architectures. In *Euro-par - 20th International Conference on Parallel Processing*, Euro-Par 2014, LNCS 8632, pages 50–62, Porto, Portugal, Aug. 2014. Springer International Publishing Switzerland.

This work was also presented on several occasions:

- ANR SOLHAR plenary meeting in Lyon, June 2015.
- Mont-Blanc meeting in Barcelona, October 2014.
- EuroPar conference Porto, August 2014.
- JointLab for Petascale Computing (JLPC) workshop in Nice, June 2014, and in Barcelona July 2015.
- ANR SONGS plenary meetings in Lyon, June 2012, in Nice, January 2014, and in Nancy, January 2015.

Finally, we emphasize that our solution is not merely a proof of concept. This tool has been fully integrated into a runtime, and can thus be used for performance evaluation studies of both applications (block *Application* in Figure 1.1) and runtime (block *Runtime* in Figure 1.1). Some researchers have already benefited from this tool, while investigating different scheduling algorithms for the studied application [ABED+15].

1.2 Thesis organization

The rest of this thesis is organized as follows: Chapter 2 briefly describes the evolution of the HPC applications and machines. It states the challenges of efficiently programming in such context and introduces dynamic task-based runtimes as a possible solution. Developing such programs is a process that requires constant empirical validation, which may be prohibitive for large computer platforms. Therefore, simulation can be used to overcome such limits. In Chapter 3 we present state of the art related to the tools for doing reproducible research and to the techniques for doing different types of simulation. Chapter 4 addresses the question of developing a lightweight and effective workflow for conducting experimental research on modern parallel computer systems in a reproducible way. Our approach builds on two well-known tools, Git and Org-mode, and enables to address, at least partially, issues such as running experiments, provenance tracking, experimental setup reconstruction or replicable analysis. In Chapter 5, we present in detail how we ported a dynamic task-based runtime on top of a simulator. We also present the models that are essential to obtain good performance predictions. In Chapter 6, we show that our approach allows for obtaining predictions accurate within a few percents for two dense linear algebra applications on wide range of hybrid machines, within a few seconds on a commodity laptop. We validate our models by systematically comparing traces acquired in simulation with those from native executions. In Chapter 7, we show that it is also possible to conduct faithful simulation of the behavior of an irregular fully-featured library both in terms of performance and memory on multi-core architectures. Finally, in Chapter 8, we conclude with the contributions of our study, as well as its current limitation and future directions.

Chapter 2

Background

2.1 Programming challenges for HPC application developers

2.1.1 HPC applications

HPC applications are computer programs that require high-level computational power. They include a wide range of scientific applications from various domains, such as molecular modeling, weather forecast, quantum mechanics, simulation for engineering/finance/biology, etc. For example, the SPECFEM3D [PKL+11] application simulates seismic wave propagation on local to regional scales using continuous Galerkin spectral-element method. Another good representative is BigDFT [Big12] that proposes a novel approach for electronic structure simulation based on the Daubechies wavelets formalism [GNG+08, Nus82].

Executing these applications on a commodity machine is possible, but it could take many months or even years to complete, which is too long for any practical purposes. One CPU being insufficient for such high software requirements, supercomputers were introduced. These machines typically comprise thousands of nodes interconnected through high-speed networks and possibly equipped with accelerators. For example, SPECFEM3D was executed in February 2013 on the IBM BlueWaters machine using 693,600 cores.

To continue with the progress in their domains, researchers need their applications to focus on more details which makes applications more complex, hence the need for even more computational power. Nevertheless, with the technology we currently have at our disposal, this requirement is not easy to fulfill. Table 2.1 shows the list of the officially most powerful machines in the world in the last years (ranked first on the TOP500 list), where the multiplicity of solutions can be clearly observed. Indeed, the choice of components and methods with which future supercomputers will be constructed is constantly debated between HPC experts, many new technologies emerging and showing promising performance results.

Table 2.1: Ranked first on the TOP500 list, illustrating the diversity of architectures used to construct modern HPC platforms.

Date	Name	CPU	Accelerator	Interconnect
06/2013	Tianhe-2	Intel Xeon	Xeon Phi	TH Exp.2
11/2012	Titan	AMD Opteron	Kepler	Cray Gemini
06/2012	Sequoia	IBM Power BQC	/	Custom
11/2011	K computer	Fujitsu SPARC64	/	Tofu
11/2010	Tianhe-1A	Intel Xeon	Fermi	Proprietary

2.1.2 Different architectures used in HPC

In the previous decades, computing power of individual CPUs mainly improved thanks to the frequency increase, miniaturization, and hardware optimizations (cache hierarchy and aggressive cache policies, out-of-order execution, branch prediction, speculative execution, etc.). However, frequency increase and hardware optimizations are now facing hard limits and incur unacceptable power consumption. Power consumption grows more than quadratically with the growth of frequency. Additionally, speculative execution performs many useless operations and although they seem free in terms of scheduling on the CPU resources, in terms of power utilization they waste a lot of energy. If supercomputers are to achieve a predicted performance in the following years, the power efficiency of individual CPUs will have to be reduced by a factor of 30.

Several different approaches are envisioned. The first one attempts to improve the power consumption and increase parallelism of standard Intel/AMD processors currently used in HPC. This is performed through various frequency scaling techniques, increased number of cores per CPU, additional support for vectorized instructions, and many other very sophisticated methods. Still, performance improvements introduced by each new generation of such processors are becoming less and less significant. To achive desired GFLOPS per Watt ratio, radical changes in the approach are needed.

Another approach is to build on existing low-power CPUs commonly used in embedded systems and to try to improve their performance, as proposed by the European Mont-Blanc project [Mon]. The Mont-Blanc project aims at developing scalable and power efficient HPC platform based on low-power ARM technology. ARM (Advanced RISC Machine and, before that, the Acorn RISC Machine) processors are particularly designed for portable devices as they have very low electric power consumption. Nowadays, these CPUs are embedded on almost all mobile phones and personal digital assistants. However, such CPUs have very different characteristics and using them in HPC is not straightforward.

Yet another solution is based on the use of large Non-Uniform Memory Access (NUMA) machines. These machines consist of multiple multi-core processors, each containing its own part of the memory hierarchy, interconnected with the other processors through PCI bus. The time processor needs to access the data in the memory is non-uniform as it depends whether the data is stored in local or distant memory bank. These architectures can achieve great performance for certain applications, however programming them efficiently is very hard as data locality has to be carefully controlled. Since obtaining a solution with optimally distributed data for a single NUMA machine is already a challenge, constructing a supercomputer with multiple nodes based on this architecture is even harder. With the current system this approach would be extremely complex to program, thus it is unlikely to scale well.

Graphics Processing Unit (GPU) offers a very valuable addition to the CPUs in order to provide more computation power. This hardware component was initially designed to accelerate the image creation for the display output. However, in the last decade GPUs have been also extensively used for doing computation. Although they have a limited flow control and cache memory, GPUs offer a large number of small cores. These cores can be used in parallel very efficiently for arithmetic operations on large arrays. This makes GPUs a perfect candidate for running applications containing operations such as matrix multiplication, which is often the case with HPC codes. However, this requires to rearrange applications and distinguish between control parts executed on CPUs and computation intensive parts executed on the GPUs. Moreover, data transfers to the GPUs have to be carefully managed. Another alternative for executing intensive computation parts of the application is to use coprocessors, such as the Intel MIC (Intel Many Integrated Core Architecture), better known under its brand name Xeon Phi.

We presented only a few basic groups of processors and accelerators and each of them contains various derivatives. Regardless of the choice or a mixture of these architectures, it is certain that future supercomputers will require a large number of such components. These nodes will also have to be efficiently connected to minimize the cost of data transfers, using technologies such as InfiniBand, Quadrics, Ethernet, etc. However, programming an application on such a complex hardware is becoming extremely difficult.

2.1.3 Exploiting machine resources

In order to exploit the resources of modern HPC machines, developers need to implement their solutions as a highly parallel, distributed programs.

To benefit from the computing power offered by huge number of nodes present in the supercomputers, it is necessary to execute a parallel application in which work is distributed between processing units. On each node, computation tasks are executed while the input and the output data of the tasks is exchanged with other nodes through network. Such communication is typically performed by passing messages. The programming paradigm that became a standard for such applications is MPI (Message Passing Interface) [DHHW93, GLS99b].

Additionally, with multi-core machines it is also possible to make execution parallel within the node itself, typically by running one or several threads/processes per processor core. These multiple threads on the same node thus share RAM and cache memories, through which data exchange is performed. A widely used API supporting such programming paradigm is OpenMP [DM98].

Another possibility is offered by Cilk [BJK+95] and Cilk++ [Cil], which are extensions to the standard C and C++ programming languages. Using these languages, the programmer can specify using predefined key words what parts of the code should be executed in parallel.

GPGPU (General Purpose computation on the Graphics Processing Units) exploits the huge computational potential of the GPUs, commonly using programming languages such as CUDA [CUD07] or OpenCL [Opeb, SGS10]. CUDA is a programming model developed by Nvidia that provides an API for using their GPUs for parallel computing. OpenCL has a similar goal, but goes one step further as it aims at providing a possibility to program on heterogeneous platforms in general, using any accelerator. Although OpenCL provides more opportunities, often CUDA outperforms it in terms of performance due to the differences in memory model and better Nvidia compilation optimizations.

Implementing an application using any of the aforementioned approaches is not trivial, as it requires from developers to know well the concepts and syntax of each language. Moreover, to optimize the application for distributed hybrid systems, developers have to combine several different languages with their distinctive paradigms and ensure that these will work efficiently together. Such a direct, rigid approach typically scales badly for most applications, both in terms of implementation effort and final execution performance.

It is very hard to achieve good utilization of the resources for complex hardware setups. To exploit the tremendous computation power offered by such systems, application needs to be divided in a large number of smaller tasks. These tasks are executed on different processing units, in the order that respects the data dependencies between them. However, implementing dynamic execution of the application is a real challenge. Programming tasks efficiently is a first concern, but managing the combination of computation execution and data transfers can also become extremely complex, particularly when dealing with multiple GPUs. In the past few years, it has become very common to deal with that through the use of an additional software layer, a runtime system, based on the task programming paradigm.

2.1.4 Dynamic task-based runtimes

Whereas task-based runtime systems were mainly research tools in the past years, their recent progress makes them now solid candidates for designing advanced scientific software as they provide programming paradigms that allow the programmer to express concurrency in a simple yet effective way and relieve him from the burden of dealing with low-level architectural details. Runtimes abstract the complexity of the underlying hardware by offering a unique API for the application developers, who implement both CPU and GPU implementations for the tasks.

Runtimes employ a very modular approach. First, applications are written at high level, independently of the hardware architecture, as DAG of tasks where each vertex represents a computation task and each edge represents a dependency between tasks, possibly involving data transfers. Figure 2.1 shows a typical example of such task graph for the Cholesky factorization application (, whose internals are detailed in Subsection 6.1.1). A second part is in charge of

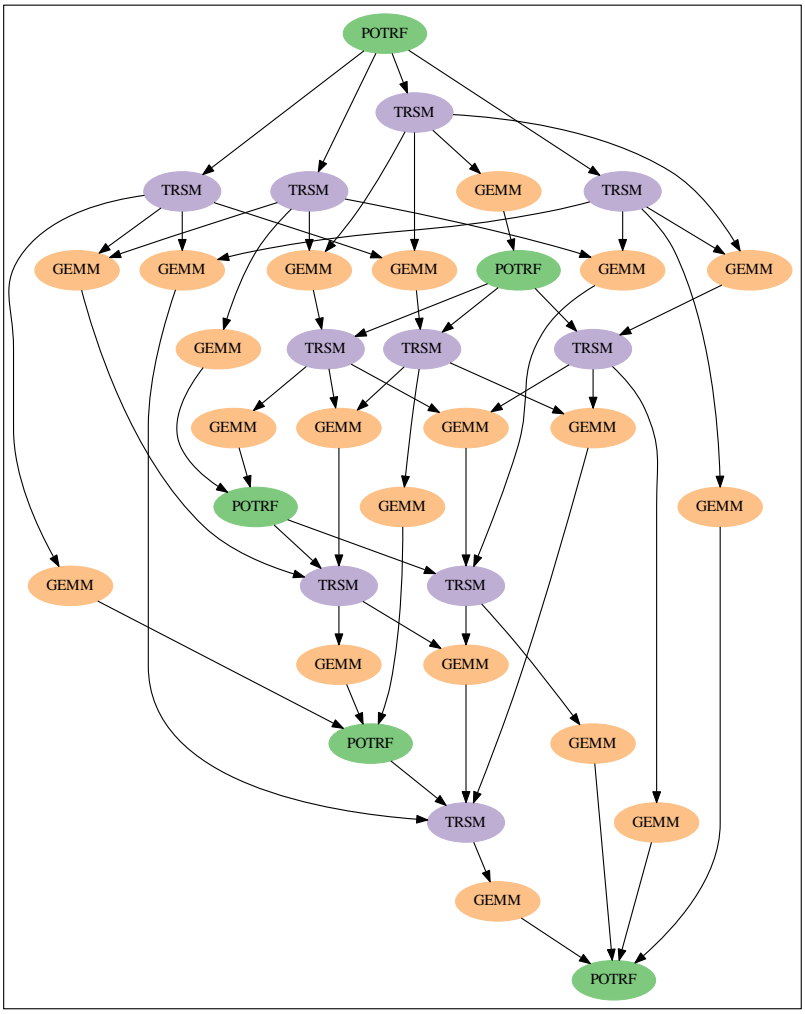


Figure 2.1: An example of DAG: task graph of the tiled Cholesky factorization of 5×5 matrix with the block dimension 960, implemented in StarPU.

scheduling the DAG, i.e., to decide when and where (on which processing unit) to execute each task. In the third part, a runtime engine takes care of implementing the scheduling decisions, i.e., to retrieve the data necessary for the execution of a task (taking care of ensuring the data coherency), to trigger its execution and to update the state of the DAG upon its task completion. The fourth part consists of the tasks code optimized for the target architectures. In most cases, the bottom three parts need not to be written by the application developer since most runtime systems embed off-the-shelf generic scheduling algorithms. For example, work-stealing [ABP01], MCT [THW02b] (Minimum Completion Time) or HEFT [THW02a] (Heterogeneous Earliest Finish Time) very efficiently exploit target architectures. Such runtimes can also take into account the NUMA effects on architectures with large number of CPUs using shared memory [PJN08, BFG+09]. Application programmers are thus relieved from scheduling concerns and technical details.

As a result, the concern becomes choosing the right task granularity, task graph structure, and scheduling strategies optimizations. Task granularity is of a particular concern on hybrid platforms, since a trade-off must be found between large tasks which are efficient on GPUs but expose little task parallelism, and a lot of small tasks for CPUs. The task graph structure itself can have an influence on execution time, by requiring more or less communication compared to computation, which can be an issue depending on the available bandwidth of the target system. Last but not least, optimizing scheduling strategies has been a concern for decades, and the introduction of heterogeneous architectures only makes it even more challenging.

However, no consensus has still been reached on a specific paradigm of the runtime. For example, PTG (Parametrized Task Graph) approaches [BBD+12] consist of explicitly describing tasks (vertices of the DAG) and their mutual dependencies (edges) by informing the runtime system with a set of dependency rules. In such a way, the DAG is never explicitly built but can be progressively unrolled and traversed in a very effective and flexible way. This approach can achieve a great scalability on a very large number of processors but explicitly expressing the dependencies may be a hard task, especially when designing complex schemes.

On the other hand, the STF (Sequential Task Flow) model simply consists of submitting a sequence of tasks through a non blocking function call that delegates the execution of the task to the runtime system. Upon submission, the runtime system adds the task to the current DAG along with its dependencies which are automatically computed through data dependency analysis [AK02]. The actual execution of the task is then postponed to the moment when its dependencies are satisfied. This paradigm is also sometimes referred to as *superscalar* since it mimics the functioning of superscalar processors where instructions are issued sequentially from a single stream but can actually be executed in a different order and, possibly, in parallel depending on their mutual dependencies.

Many runtimes with different paradigms (, optimized for slightly different use cases,) have been proposed. A complete review of them is out of the scope of this thesis and thus we present only a few of the most currently used ones: QUARK [YKD] and PaRSEC [BBD+13] from ICL, Univ. of Tennessee Knoxville (USA), Supermatrix [CVV+08] from University of Texas (USA), Charm++ [KK96] from University of Illinois (USA), StarPU [ATNW11] from Inria Bordeaux (France), KAAPI [GBP07] from Inria Grenoble (France), StarSs [PBAL09] from Barcelona Supercomputing Center (Spain).

Such wide range of tools and no standard API are the main reason why most of the HPC applications haven't been ported yet on top of any runtime. Although this provides many benefits on the long run, an initial coupling of the application with a specific runtime requires a significant development effort. Still, even though there is a huge number of applications coming from completely different domains and trying to answer various scientific questions, their core is often very similar to linear algebra problems. Indeed, the parts of the real scientific application that generally consume most of the machine resources are related to certain arithmetical operations. In order to decrease application execution time, one usually needs to optimize multiplication of densely or sparsely filled matrices on multiple processing units.

2.1.5 Linear algebra applications

The researchers studying linear algebra problems are facing the same performance scalability and hardware diversity issues, and thus they followed the same path and redeveloped their algorithms to be based on dynamic task-based runtimes. Since such applications have typically much smaller code base, implementing them on top of runtime requires less effort. Consequently, the current trend is to evaluate the potential of runtimes using linear algebra kernels. Once results are validated and good application performance is ensured, research will move to more complex use cases, which are real HPC applications.

We focus on dense and sparse algorithms, which are two major groups of linear algebra applications.

Dense linear algebra

Solving dense linear algebra problems is the core of many applications in the fields of computer science, natural sciences, mathematics and social sciences. Therefore, efficiently computing solutions on ever changing computer architectures is of critical importance for obtaining a good, portable performance of the whole application. To this end, many software packages have been developed.

BLAS (Basic Linear Algebra Subprograms) [LHKK79, DDCHD90] provides C and Fortran routines for computing common linear algebra operations such as vector addition, dot products, linear combinations, scalar multiplication and matrix multiplication. These operations are highly optimized for the underneath architecture, taking advantage of many hardware extensions such as SIMD (Single Instruction Multiple Data) instructions. There are many implementations of BLAS, the most popular being ACML (AMD Core Math Library) [ACM14] and MKL (Intel Math Kernel Library) [MKL12] that are proprietary vendor libraries well optimized for their CPUs, and open source solutions OpenBLAS [Opea] and ATLAS [WPD01]. Moreover, there are various extension of the standard BLAS interface such as cuBLAS (NVIDIA CUDA Basic Linear Algebra Subroutines) [CUB] used for GPUs.

Many software packages are built on top of BLAS-compatible libraries. The most widely used for HPC applications is probably LAPACK (Linear Algebra PACKage) [ABB+99] library. The main goal of this project is to provide the programmers with standard building blocks for performing basic vector and matrix operations. With these efficient and scalable routines, developers can ensure good performance of their applications and portability across all the machine containing BLAS libraries. Therefore, both BLAS and LAPACK became a part of the standard software stack for all HPC machines.

With the evolution of computers, these libraries had to evolve as well and be able to support executions on distributed machine. This resulted in extensions such as BLACS (Basic Linear Algebra Communication Subprograms) [DW97] and ScaLAPACK (Scalable LAPACK) [BCC+97]. BLACS is a message passing interface for linear algebra kernels. ScaLAPACK is a subset of LAPACK, implemented for parallel distributed memory machines. It allows for solving dense and banded linear systems, least squares problems, eigenvalue problems, and singular value problems.

However, the diversity of the computer architectures used nowadays for achieving maximal performance required appropriate extension of the BLAS and the LAPACK. This lead to creation of several spin-off projects, such as MAGMA, PLASMA and DPLASMA. MAGMA (Matrix Algebra on GPU and Multicore Architectures) [Mag] is a collection of linear algebra libraries for heterogeneous architectures. This project aims at exploiting the huge computational power provided by GPUs, for doing linear algebra operations. It is based on the cuBLAS library, using the same interfaces as the current standard BLAS and LAPACK libraries. PLASMA (Parallel Linear Algebra for Scalable Multi-core Architectures) [ADH+09] aims at optimizing linear algebra operations on multi-core machines. There is also an extension of this library specialized for distributed memory systems called DPLASMA [BBD+11]. Finally, there are efforts for uniting the aforementioned approaches in order to exploit simultaneously all the resources provided by hybrid, multi-core, distributed memory machines. Examples of such initiatives are MORSE (Matrices Over Runtime Systems at Exascale) [MOR] project, and recently its sub-project Chameleon [Cha].

To maximize parallelism and achieve better performance, these libraries are generally based on so-called *tile* algorithms [BLKD09, QOQOG+09]. In this approach, big matrices are divided into smaller sub-matrices (tiles), which can then be treated separately by the application. This allows for breaking the problem into fine-grain tasks that can be executed in parallel.

However, for obtaining optimal utilization of processing units and hence the performance of the whole application, these fine grain tasks need to be dynamically scheduled. Therefore, the previously described linear algebra libraries often rely directly on runtime systems. For example, PLASMA [ADH+09], DPLASMA [BBD+11], and Chameleon [Cha], are executed on top of QUARK [YKD], PaRSEC [BBD+13], and StarPU [ATNW11], respectively.

Finally, recent advances in dense linear algebra community lead up to the point that the OpenMP board has included similar features in the latest OpenMP standard 4.0 [Ope13] making it possible to design more irregular algorithms, such as sparse direct methods, with a similar approach.

Sparse linear algebra

Applications doing sparse matrix factorizations, sparse linear solvers, face very irregular workloads. Hence, programming a parallel solution that efficiently exploits machine resources is extremely challenging.

Interestingly, to better extract potential parallelism, these solvers were already often designed with, to some extent, the concept of *task* before having in mind the goal of being executed on top of a runtime system. It is the case of the SuperLU [Li05] supernodal solver and the MUMPS [ADKL01, AGLP06] multifrontal solver. These two solvers achieve parallelism in different ways, as SuperLU is based on a right-looking supernodal technique with static pivoting, while MUMPS uses a multifrontal approach with dynamic pivoting [ADLL01]. Such difference causes a contrasting application executions, as shown in Figure 2.2. SuperLU (bottom plot) has a very regular pattern with clearly distinct computation/communication phases, while MUMPS (top plot) has these two operations completely overlapping. The solvers based on such a dynamic task scheduling like MUMPS, can thus greatly benefit from using runtime systems.

Therefore, this application was recently ported on top of the StarPU runtime system [ABGL13] (see Section 7.1 for more details). Another example is the PaSTIX solver that has been extended [LFR+14] to rely on either the StarPU or the PaRSEC runtime systems for performing supernodal Cholesky and LU factorizations on multi-core architectures possibly equipped with GPUs. Finally, Kim et al. [KE14] presented a DAG-based approach for sparse LDL^T factorizations where OpenMP tasks are used and dependencies between nodes of the elimination tree are implicitly handled through a recursive submission of tasks, whereas intra-node dependencies are essentially handled manually.

2.2 Experimental challenges for HPC application developers

The increasing complexity of both hardware and software environment makes programming HPC applications very hard. Moreover, if developers aim at having a solution that can be correctly executed on various systems, while at the same time preserving a good performance, implementing the applications becomes even more challenging. To achieve this goal it is essential to apply a highly modular programming approach, dividing the application into tasks that can be executed in parallel. Such solutions have to be validated in various contexts, which can only be acquired through an iterative process that involves many empirical testings.

Using dynamic task-based runtimes and their API that abstracts the underlying system diversity can save time for the application development. Still, finding the right way to couple the application and the runtime is often not trivial. Developers need to adapt their codes to the specific runtime paradigms, as well as choose the right task granularity, task graph structure, and scheduling strategies optimizations. This demands understanding many sophisticated runtime concepts, which is often, at least partially, based on trial and error method. Moreover, imple-

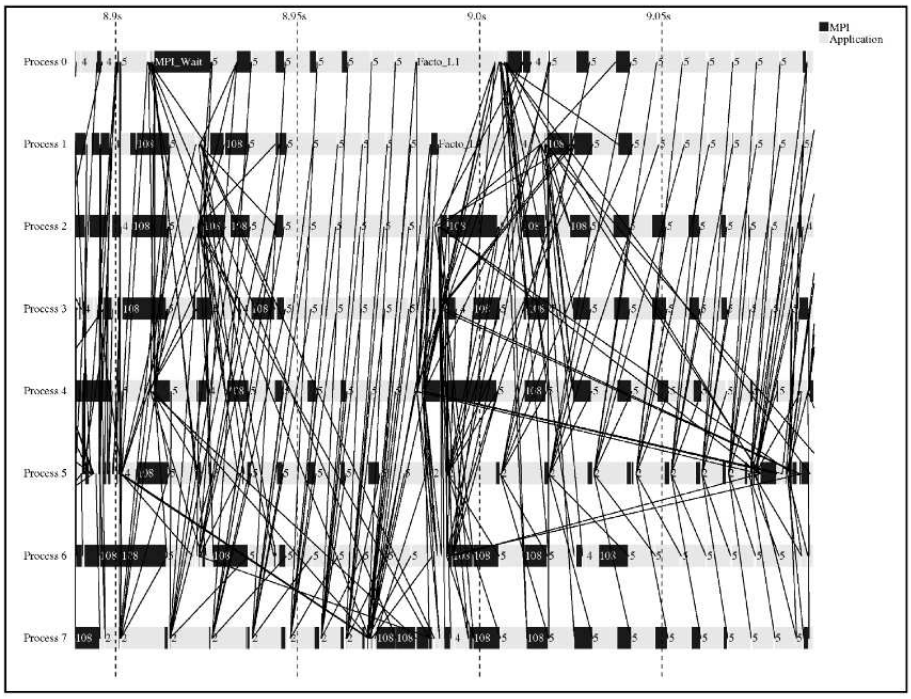


Fig. 3. Illustration of the asynchronous behaviour of the MUMPS factorization phase.

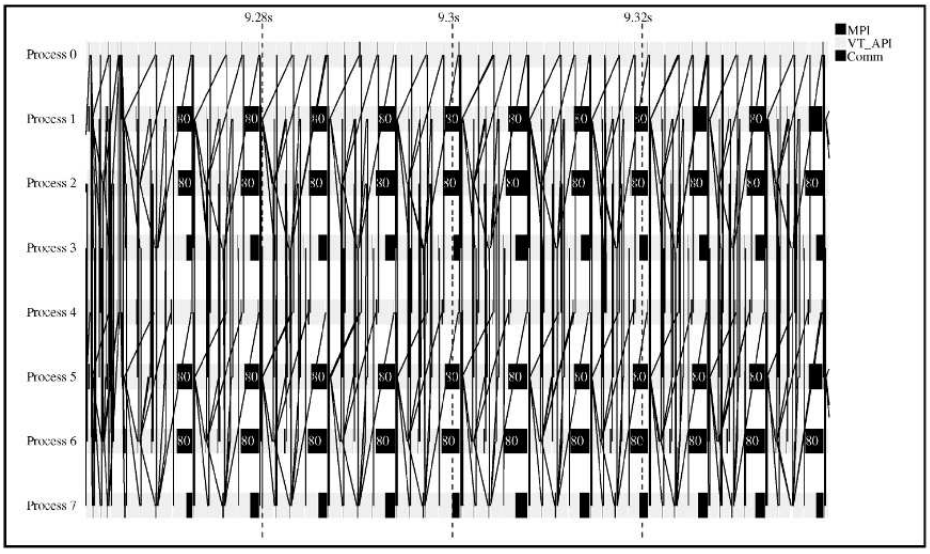


Fig. 4. Illustration of the relatively more synchronous behaviour of the SuperLU factorization phase.

Figure 2.2: Comparing different scheduling algorithms of MUMPS and SuperLU sparse linear solvers as done by Amestoy et al. in [ADLL01]. MUMPS has very irregular patterns, which makes it a good candidate for using dynamic task-based runtime systems.

menting runtime itself is a challenge, as it has to ensure both correct execution of all applications on top of it and their good performance. Therefore, the runtime and its performance have to be regularly evaluated for numerous scenarios by executing codes on different machines.

Therefore, HPC developers for both applications and runtimes have a great need for a reproducible executions of their programs in order to fully understand and improve both code and its performance. Since execution time of the tasks exhibit certain variability, dynamic schedulers take non-deterministic and opportunistic scheduling decisions. The resulting performance is thus far from deterministic, which makes performance comparisons sometimes questionable and debugging of non-deterministic deadlocks extremely hard.

There is equally a need for the extensive experimental studies. Constantly, when doing any change to their code, developers have to test the new solution on a wide range of diverse computer platforms and with large number of different parameter values. This is necessary to ensure that the resulting design choices are generic, and not only suited to the few basic setups at hand.

However, the desired machines are not always available to experiment on. In the HPC domain, researchers often work with expensive hybrid prototype hardware that has a short lifetime due to the rapid technological evolution. Therefore, research centers are typically moderating their expenses on buying new machines. Moreover, the different paths taken by manufacturers in the pursuit for better performance as well as the recent advances in computer architectures, lead to a large diversity of machine architectures used nowadays in HPC. Consequently, researchers have only a very limited number of different computers at their disposal and even these are typically shared by many users. In such context, a dedicated access to the machine has to be planned well in advance. Additionally, getting accurate measurement results for all combinations of input parameters is nontrivial and requires reserving the target system for a long period, which can become prohibitive.

Moreover, even with an unlimited access, running experiments on such machines is very costly in terms of resources, time and energy. This is especially noteworthy for supercomputers, as they contain huge number of nodes and disks, connected via large networks and cooled down with very sophisticated cooling systems.

2.3 Conclusion

The described context emphasizes on the experimental nature of the studies on dynamic HPC applications, especially the ones based on runtime systems. Indeed, compared to other computer science domains, research in this field is extremely empirically oriented. However, there are no standard experimental methods for conducting such studies.

Simulation is a technique proven to be extremely useful to study complex systems and which could be a very powerful way to address these issues. Performance models can be collected for a wide range of target architectures, and then used for simulating different executions, running on a single commodity platform. Since the execution can be made deterministic, experiments become completely reproducible and the experimental setup is better controlled. This eases the debugging of both the code and the performance of the application. If designed with a coarse-grain approach, simulations can be much faster and require less resources than the real executions on the target machine. This allows conducting larger experimental campaigns on a commodity machine, thus minimizing the number of costly hours spent on a supercomputers. Additionally, it is possible to try to extrapolate target architectures, for instance by trying to increase the available PCI bandwidth, the number of GPU devices, etc. and thus even estimate performance which would be obtained on hypothetical platforms.

Therefore, our goal is to contribute to the experimental/analysis methodology of dynamic HPC applications through simulation. Taking into account the evolution of the HPC hardware and software, we decided to focus on applications running on top of dynamic task-based runtimes as we strongly believe that they will soon become a mainstream solution for efficiently implementing HPC applications.

Chapter 3

Related Work

3.1 Reproducible research

In the recent years, the more and more frequent discovery of frauds or mistakes in published results has shed the light on the importance of reproducible research [Ioa05, Rep, Ste11]. For example, cancer studies of researchers leaded by Anil Potti from Duke University were published in prestigious journals and used for three trials on patients, even though their results were not reproducible at all. Some other researchers instantly reacted, pointing out several inconsistencies with the data, but their opinion was ignored. It was only later that the scientific misconduct of the study was disclosed and all publications retracted. Therefore, rigorous protocols have to be put in place to avoid both accidental errors and voluntary manipulations of the published results. It is expected from biologists or chemists to provide a detailed explanation on dozens of pages for all methods and data that was part of their research. Without that no paper should be published in a respected conference or journal.

However, computer science or at least the part of it that can be considered as experimental science is relatively a young discipline and still doesn't have the standards regarding reproducibility. Since it is not mandatory, many decide to conveniently ignore this important aspect. They hurry towards new results solely concentrating on their primary research topic, without noting anything about the path they are passing through. In our field, researchers are more focused (and rewarded) for positive results and novelties rather than on presenting failed attempts or redoing, verifying and enlarging the work of others. Conferences and journals are full of success stories and new algorithms, although undoubtedly there is a tremendous work and many failures behind it. There are many useful lessons to take from "bad", abnormal, negative, unexpected results as well. Thus, consolidating existing work would produce better science and by that help the whole community. To illustrate the current state in our field, researchers from the University of Arizona [CPM+14] studied 613 articles from 8 of the top ACM conferences. They demonstrated how for the most of these papers it was impossible to compile and run the source code (without even looking at the results it would produce). Similar conclusion can be drawn from the studies done in peer-to-peer [NBL+06] or MPI [CARL14] communities.

As a common excuse for ignoring the reproducibility of their work, most scientists blaim the incredibly fast evolution of technology. It is true that both hardware and software of modern computers rapidly changes and becomes increasingly complex. Dynamic and opportunistic optimizations are done at every level and even experts have troubles fully understanding them. Such systems can no longer be considered as deterministic, especially when it comes to measuring execution times of parallel multi-threaded programs. Controlling every relevant sophisticated component during such measurements is almost impossible even in single processor cases [MDHS09], making the full reproduction of experiments extremely difficult. However, if studying computers has become similar to studying a natural phenomena then it should use the same principles as other scientific fields that had them defined centuries ago. Although in computer science many conclusions are commonly based on experimental results, articles generally poorly detail the exper-

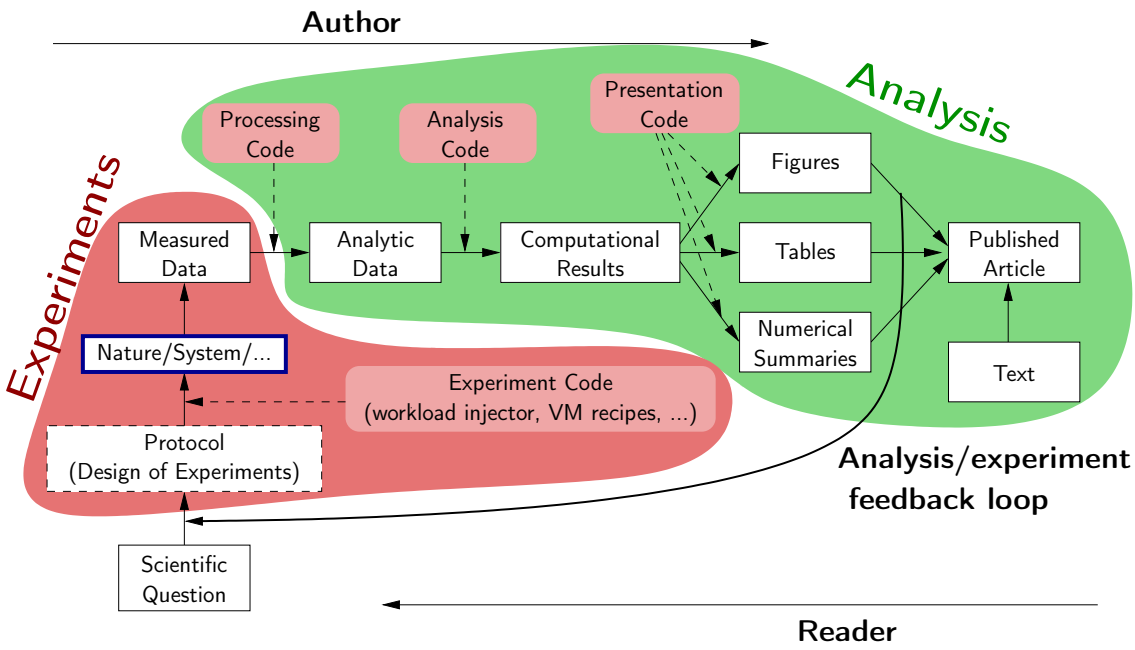


Figure 3.1: Ideally, the experimenter would keep track of the whole set of decisions taken to conduct its research as well as all the code used to both conduct experiments and perform the analysis. Figure inspired by Roger D. Peng et al. [Rog09].

imental protocol. Left with insufficient information, readers have generally troubles reproducing the study and building upon it. Yet, as reminded by Drummond [Dru09], *reproducibility* of experimental results is the hallmark of science and there is no reason why this should not be applied to computer science as well.

The ultimate goal of reproducible research is to bridge the current gap between the authors and the article readers (see Figure 3.1) by providing as much material as possible on the scientist choices and employed artifacts. As shown in the figure, the path between defining the scientific question one wants to answer to publishing its results has many phases. Typically, researchers will iterate many times through the whole process and it is very important that they ensure reproducibility or replicability of the results. The difference between terms the *reproducibility* and *replicability* and their importance has been a topic of many discussions. In the rest of this document, we will relate to the definitions given by Dror G. Feitelson [Fei15]:

> *Reproducibility is the reproduction of the gist of an experiment: implementing the same general idea, in a similar setting, with newly created appropriate experimental apparatus.*
>
> *Replicability is the recreation of the same experimental apparatus, and using it to perform exactly the same experiment.*

If these definitions were to apply to the Figure 3.1, one would conclude that the experiments part can be either reproducible or replicable depending on the type of experiments that are considered. However, analysis is replicable by essence.

Hence, a new movement promoting the development of reproducible research tools and practices has emerged, especially for computational sciences. In this context, scientist not only conduct numerous simulations but also rely on complex repetitive computation intensive workflows to analyze large amount of data. As a consequence, the tools developed in this community generally focus on replicability of data analysis and provenance tracking [SLP14]. Projects like Kepler [Kep] or Taverna [Tav13] are very convenient for the scientists doing research in bioinformatics, astronomy or neuroscience, but unfortunately completely unadapted for researchers in our community.

These huge, integrated frameworks do not leave enough flexibility for users, imposing many rules and it is not in our culture to work with such tools. There are numerous platforms for sharing scientific results and workflows focusing on their replicability, e.g., RunMyCode [HPS14], ActivePapers [Hin11], myexperiments [RGS09] and many more. Even commercial publishers start to promote their own (private) systems, such as Elsevier executable papers [Exe] . However, these are at the moment inappropriate for large, complex source codes that depend on many external libraries.

The main shortcomings of all these tools are that they are mostly made for deterministic processes, fixed input data and working on a commodity machine. However, conducting experiments on prototype, hybrid computers with a limited access and control rights require different kind of workflows and techniques. Although HPC experiments involve running complex codes, sometimes on distributed systems, they do not focus on execution results, but rather on the time taken to run a program and how the machine resources were used. It is important to state that these experiments are not replicable by essence. Nevertheless in such cases, researchers should still at least aim at full reproducibility of their work.

One way to help with reproducibility is to use *literate programming*. This principle of writing the code while at the same time documenting it, was defined more than 30 years ago by Donald Knuth [Knu84]. Such a promising method was supposed to completely change the way people program, as "surely nobody wants to admit writing an *illiterate* program". Still, this never happened, probably because writing codes with many files, function calls and wrappers is too complicated for such approach as it requires constant jumping from one part of the code to another. On the other hand, literate programming is much easier to apply on the use cases that are naturally sequential, e.g., for doing data analysis.

For every process during the study, the minimum is to save all code and input data. However, this is far from enough to actually guarantee the reproducibility/replicability. We will now describe aspects related to software, methodology and provenance tracking, which are often neglected by researchers in our community. Additionally, we provide few most commonly used solutions to address these specific problems with the explanations on why these are not enough for our research context.

Code and data accessibility

It is widely accepted that tools like Git or svn are indispensable in everyday work on software development. Additionally, they help at sharing the code and letting other people contribute. GitHub [Gitb] and frameworks with similar philosophy are addressing these needs and thus becoming increasingly popular solutions in the community. Using such tools for managing experiments is however not that common. Public file hosting services, such as Dropbox or Google Drive have become a mostly used way to share data among scientists that want to collaborate. The unclear durability of such service and the specific requirements scientists have in term of size and visibility has lead to the development of another group of services (e.g., figshare [Fig] and zenodo [Zen]) that are focused on making data publicly and permanently available while ensuring it is easily understandable to everyone.

Platform accessibility

Many researchers conduct their experiments on large computing platforms such as Grid5000 [BCAC+13] or PlanetLab [PACR03] and which have been specifically designed for large scale distributed/parallel system experimentation. Using such infrastructures eases reproduction but also requires to manage resource reservation and to orchestrate the experiment, hence the need for specific tools [RSRVO13b, ABD+07]. However, the machines we considered in our study are generally recent prototypes, some of them being rather unique and meant to be accessed directly without any specific protocol.

Setting up environments

It is necessary to carefully configure machines before doing experiments. Among the few tools specifically designed for this purpose and based on recipes, we can cite Kameleon [RRE14], which allows for reconstructing an environment step by step. Another approach consists in automatically capturing the environment required to run the code (e.g., as done by CDE [SLP14, chap.4], ReproZip [CSF13] and CARE [CAR]) or to use virtual machines so that code can be later re-executed on any other computer. In our use case, experimental platforms are already set up by expert administrators and we have neither the permission nor particular interest to modify their configuration.

Conducting experiments

Numerous tools for running experiments in a reproducible way were recently proposed [BRNR15]. These tools are not specifically designed for HPC experiments but could easily be adapted. Another set of related tools developed for computational sciences comprises Sumatra [SLP14, chap.3] and VisTrails [SLP14, chap.2]. Such tools are rather oriented on performing a given set of computations and do not offer enough control on how the computations are orchestrated to measure performances. They are thus somehow inadequate in our context. Some parts or ideas underlying the previously mentioned tools could have been beneficial to our case study. In our experiments, simple scripts were sufficient although they often require interactive adaptations to the machines on which they are run, making experiments engine that aim at automatic execution difficult to use.

Provenance tracking

Knowing how data was obtained is a complex problem [BNG15]. The first part involves collecting meta-data, such as system information, experimental conditions, etc. In our domain, such part is often neglected although experimental engines sometimes provide support for automatically capturing it. The second part, frequently forgotten in our domain, is to keep track of any transformation applied to the data. In such context, the question of storing both data and meta-data quickly arises and the classical approach to solve these issues involves using a database. However, this solution has its limits, as managing source codes or comments in a database is not convenient and is in our opinion handled in a much better way by using version control systems and literate programming.

Documenting

While provenance tracking is focused on how data was obtained, it is generally not concerned with why the experiments were run and what the observations on the results are. These things have to be thoroughly documented, since even the experimenters tend to quickly forget all the details. One way is to encourage users to keep notes when running experiment (e.g., in Sumatra [SLP14, chap.3] and BURRITO [GS12]), while the other one consists in writing a laboratory notebook (e.g., with IPython [PG07]).

Extendability

It is hard to define good formats for all project components in the starting phase of the research. Some of the initial decisions are likely to change during the study, so the system has to be easy to extend and modify. In such a moving context, integrated tools with fixed database schemes, as done for example in Sumatra, seemed too rigid to us although they definitely inspired several parts of our workflow. A more flexible solution without SQL is used in other projects, such as cTuning [Fur12].

Replicable analysis

We believe that researchers should only trust figures and tables that can be regenerated from raw data that comprise sufficient details on how the experiments were conducted. Therefore, ensuring replicable analysis is essential to any study. A popular solution is to rely on open-source statistical software like R and knitr that simplify figure generation and embedding in final documents [SLP14, chap.1].

Conclusion

Although there is an urgent need for changing practices in computer science, how we should proceed is not yet clear. There are widely recognized books and manuals [Jai91, Mon05] providing useful general directions and hints, however finding a right way to implement them on a specific tool and problem is not straightforward. We presented some existing solutions partially addressing different challenges raised by the need for reproducible research and the list is not exhaustive. However, none of them is completely satisfying the needs and constraints of our experimental context. Therefore, we developed an alternative approach based on well-known and widely-used tools, described in more details in Chapter 4.

3.2 Performance evaluation and simulation

In all scientific domains, empirical evaluation of certain systems is hard and even sometimes impossible due to various technological, time or resource limitations. Many different alternative techniques based on simulation are possible and extensively used in biology, nuclear physics, chemistry, etc. The same applies to the HPC field and performance evaluation of modern computers and parallel programs running on them. Indeed, in such context studying certain problems *in vivo* would require extremely powerful production machines and very long time, both not always available to the researchers. Thus, many different simulation approaches and tools have been proposed in our community and we devote the rest of this section to their presentation.

3.2.1 Different simulation approaches

Emulation

A first possible approach to predict the performance of a machine is to use an emulation. Emulators are software that allow a computer system to mimic the behavior of another one. In this case, the program creates an extra layer between an existing computer platform (host platform) and the platform to be reproduced (target platform). Then, the host machine runs through this layer the desired code, planned to be executed on the target platform. The observed performance of host machine represent a good estimation of the possible target behavior. However, since everything goes through an additional software layer, the execution is generally slowed down. Still, the emulator typically knows the speed and the characteristics of both host and target machines, thus to mimic the predicted behavior it can adjust the real speed by multiplying it with the necessary factors.

In the past, many emulation approaches have been proposed, e.g., MicroGrid [XDCC04] that allows studying various MPI applications in grid environments. This tool has a well developed CPU controller, which permits to study a large heterogeneous grid of processors using a small cluster. Another, more modern, approach that is suited not only for grid but also for Cloud, Peer-to-Peer and HPC is Distem [SBJN13]. Distem provides to users the possibility to run experiments with their codes on a large machine, mimicking the behavior of another large computer. It allows many changes to the experimental environment both in terms of processor speed and network topology, transforming a homogeneous machine into a completely heterogeneous platform. Although they enable many interesting research using unmodified code base, the main limit of these tools is that for simulating behavior of a computer cluster they also require a cluster.

Indeed, in emulation generic unmodified codes are executed and that is very positive for the users as it saves their time. On the other hand, these tools still contain certain approximations and simplifications, since it is impossible to completely mimic every detail of the target machine hardware. Such estimations introduce experiment bias that is very hard to control. For example, if there are two processes of the parallel applications that are run on the cores of the same node, they typically share several levels of cache. Hence, they could influence each other by evicting cache lines of their neighbor in order to fetch their own data. Although one could have an intuition about how big would be the influence of such phenomena for a specific execution, quantifying and controlling it in general is much harder. Indeed, many emulators report that if the codes that are executed with their tool are memory bound, the results are generally not so accurate as memory behavior is difficult to predict. Emulating the complete memory hierarchy of the target machine is hardly an option, especially for the large scale emulators, as it is very challenging to take into account all influential factors.

Cycle-accurate simulation

A solution for decreasing and better controlling the prediction error, not only for CPU caches but for many other issues as well, is to use cycle-accurate simulations. This implies simulating program execution as if it was run on the processing units cycle-by-cycle (or packet-by-packet when simulating networks). These fine-grain discrete event simulators are very complex but allegedly they provide very accurate results. Nevertheless, their problem is that they take up to 1 million times [LEE+97] more than the original runtime of the application. To execute the whole application, one would need even larger machines than the ones that are being studied. For that reason, cycle-accurate simulators are typically used only to study a few seconds of the program execution. This rises the accuracy question of such predictions, since they are based only on isolated part of the whole application [WM08]. In such conditions, even a minor error of the model on a micro level can lead to very inaccurate predictions of the whole execution on a macro level. Since this error propagation from micro to macro is very hard to control, the only way to decrease its influence is to have very precise fine-grain models for all parts of the target system. However, such models are extremely difficult to instantiate as they greatly depend on numerous environment parameters that are beyond the control and expertise of the simulator developers. Thus, the sensibility and bad scalability of such techniques makes them often inadequate and difficult to rely on. Cycle-accurate simulators can be great for understanding whether buying a CPU with larger cache or a better arithmetic logic unit (ALU) would help, however it is not clear how this approach can bring benefit to an MPI application developer.

Coarse-grain simulation

Another path, is to construct discrete event simulators that are much more coarse-grain. In such approach, complex components of the system are abstracted with simple models. For example, CPUs can be described with a single parameter that is a number of floating point operation instructions per second (FLOPS) that it can perform. Even though this is a very rough approximation, when studying large systems, such level of granularity can provide satisfactory predictions of the CPU behavior. The main advantage of this approach is that it is lighter and often faster than the real execution on the target machine. Therefore, such solutions scale much better, which allows for performing huge experimental campaigns using simulation that normally would not be possible directly on the target system. Moreover, as the models are coarse-grain they are much easier to instantiate. However, this approach has its weaknesses as well. Even though much better control of the bias is provided, the accuracy loss due to modeling approximations are still hard to evaluate. Additionally, finding the right level of abstraction and the accurate models for every component of the system is often not trivial.

Hybrid approaches

There is a whole spectrum of solutions in between these major approaches. Many tools are trying to find a good trade-off, combining different modeling techniques in the pursuit for the optimal simulator/emulator of the problem of their interest. Integration of coarse-grain and fine-grain simulations are currently investigated for example within Structural Simulation Toolkit [RHB+11] project. SST is highly modular framework that aims at providing to the users different tools for studying both individual components of the computer systems and the parallel MPI application. There are many tools targeting performance simulation of various aspects of HPC components and applications. In the following we present some of these solutions stating their strong points and weaknesses.

3.2.2 Simulating resources

Modeling communications

When simulating behavior of the target system, network topology and data transfers can be modeled using different approaches. The first way is to emulate the whole network with its traffic, which is very costly in terms of performance and thus rarely used in the HPC domain. Tools such as ModelNet [VYW+02] aim at predicting performance of Internet-like environments. However, to doing so, they need a whole cluster of machines to be executed on. These hardware requirements are very hard to meet, thus the utilization of such emulators remains narrow.

Packet-level simulators, such as ns-3 [NS3], are another particularly popular approach in the network protocol community and are widely considered as very accurate. In such solutions, transfer of every single packet containing both data and control part is simulated. The OMNeT++ [Var01] framework has been used by some research groups to build their own network simulations in various contexts [MR09, PWTR09a]. However, all these approaches have a common shortcoming. Since HPC applications typically transfer a huge amount of data between the workers, this generates enormous number of messages passing through the network. Therefore, such packet-level modeling of communication is hardly usable for parallel and distributed applications as their simulation is too long, typically several orders of magnitude longer than the real execution [FC07]. This results in scalability constraints that are often too prohibitive to the users, who thus favor faster but less accurate approaches. These fine-grain simulators are more appropriate for studying network protocols than for HPC codes.

An alternative approach is to rely on simple delay-based, analytical models that ignore complex network phenomena which are not crucial for the targeted machine. The LogP [CKP+93] model characterizes networks with very few parameters: L for latency, o for the transmission overhead, g for the gap that is a reciprocal to the bandwidth and P for the number of processors. This simplified approach proved to be sufficient for many studies involving communications, and also served as a solid base for some other more advance solutions such as LogGP [AISS95] and LogGPS [IFH01]. Many modern simulators use a similar approach, sometimes adding more parameters to the model, but sometimes also introducing additional simplifications. This allows for very scalable tools which are suited to research on large systems with many processors and an over-provisioned interconnect. Still, even though these simulators tend to provide good estimations for the very high-end platforms, in the cases where network is the performance bottleneck of the application, such approaches generally lead to completely inaccurate predictions. For example, PeerSim [MJ09], SimBA [TKE+07], EmBOINC [ETRA09] do not correctly take into account all bandwidth effects, while LogGOPSim [HSL10], BigSim [ZKK04], MPI-SIM [BDP01a] completely ignore network contention. Such constraints limit the usability of these simulators to the scenarios where a well balanced application is executed on a machine with more than enough resources and there is no external source of interference. Unfortunately, these idealized preconditions are less and less satisfied on modern HPC platforms.

Finally, another solution could be to model network links with flows, as it is done with TCP simulations. The communications through the interconnect are described via flows that share bandwidths, simulating the network contention. Such approach is very flexible, as it allows for

easily accounting for various phenomena present in the communication of the distributed programs running on clusters. However, implementing and instantiating flow models is not trivial.

Modeling CPU

The constant need for more computational power on one side and the technical and energy limitation on the other, result in modern CPUs having an extremely complex architecture. Several tools have been proposed for measuring various aspects of processor performance, most popular being PAPI [BDG+00] and likwid [THW10]. However, in order to improve the performance of their applications even further, researchers need to understand the CPU behavior in details and to this end, many rely on emulators and cycle-accurate simulators.

Developing such tools that would work not only on the current but also on the next generation processors requires great expertise. One of the most popular and recent solution is the gem5 [BBB+11] simulator that is a product of a joint effort of numerous academic and industrial institutions. It allows for investigating various different computer architectures, including ARM, ALPHA, MIPS, Power, SPARC, and x86, through diverse CPU and interconnect models. There are many other cycle-accurate CPU simulators and emulators (e.g., MARSS [PACG11], SESC [RFT+05]), and each of the proposed solutions has its own specificities. However, one common characteristic for all of them is that they have very long simulation time. This allows executing only a fractions of the total application, thus making confidence in final results questionable. Additionally, these simulators/emulators are often proprietary and uniquely designed for particular architectures which limits their usage even more.

An interesting solution that tries to limit this bad effect is Sniper [CHE+14]. This x86 simulator is based on interval code model [GEE10], which is a higher level abstraction of multi-core and multiprocessor systems. This approach is based on analyzing the performance of the intervals between two miss events of the processor, such as branch mispredictions or TLB/cache misses. The authors reported a huge speedup in the simulation duration compared to the standard cycle-accurate simulators, for the price of only several percents of the prediction error.

Another alternative technique is proposed in the PMaC framework [SCW+02]. In this macroscopic approach, the authors try to characterize the code as a whole with numbers that can later be related to platform characteristics to evaluate performances. A binary instrumentation tool based on cache simulation is executed to generate a signature for each sequential code block (typically large for loops inside the program). These descriptions contain information about the number of performed floating point operation, number of memory references, cache misses, etc. The characterization of the target machine is performed using the MultiMAPS [SCW+02] memory benchmark that makes repeated data array accesses to measure the speed of the different levels of cache hierarchy. Finally, the two results are convoluted by merging processor and memory requirements of the application to the machine capabilities [TCSS07]. The same principle is later used for modeling network and then these two are again merged for the complete prediction of the parallel application execution on a parallel system. Such framework for performance modeling and prediction is faster than cycle-accurate simulation and more informative than simple benchmarking of the application. The idea behind it is very intuitive and well structured.

However, this approach seems quite difficult to apply on recent computer architectures. We tried to employ similar techniques in our study on Intel and ARM caches [8] as a part of SONGS [SON] and Mont-blanc [Mon] projects, but could not really apply it. The main problem is that modern architectures have large number of parameters that dictate the CPU performance, some of which are out of researchers' control. Good and stable performance is ultimately possible, but only after a lot of hardware and environment tuning has been done. Such modifications are specific to the experiment setup and are rarely portable. These issues can be very clearly observed on ARM processors, that are rapidly changing their design in the last years. Even though researchers from PMaC team reported good results on these architectures [LTJ+14], in our studies we identified several sources that can cause big performance variations [6].

Therefore, we believe that using code characterization and cache simulation can only provide good CPU models for highly optimized programs and not too complex CPUs. Unfortunately, this

is not the case during the period of code development and experimentation on various machine architectures.

There are other simple approaches, such as the one proposed by BigSim [ZKK04]. Although this simulation framework provides multiple methods for modeling CPUs, the preferred one is when users themselves provide the code execution times. The Dimemas [BLGE03] solution, is somehow similar and requires two input files from the users. The first one is the trace of the application execution, clearly indicating the parts where code is executed on the CPU. The second file is used for scaling, where the characteristics of the new architectures are described through coefficients with which the execution time of the corresponding code should be multiplied.

Finally, there is also a basic approach where necessary code is executed on the real machine, measuring its duration. These benchmarked values are then used to construct models of the code blocks corresponding to the computational parts of the applications. These models are later consulted during the simulation, increasing simulation timer for the benchmarked values.

Modeling GPU

Using general-purpose programming models on graphics processing units (GPGPU) opens new possibilities for improving the performance on parallel applications. Additionally, GPU micro-architectures are typically less complex than the CPU ones, making them easier to model and simulate. Still, several micro-benchmarking studies [WPSAM10, ZLN+15] showed that understanding and predicting code behavior on GPUs is not trivial and that there are many unexpected phenomena to take into account.

Therefore, many detailed micro-architecture level simulators of GPUs have been developed in the last years. For example GPGPU-Sim [BYF+09], one of the most commonly used cycle-accurate GPU simulator, runs directly NVIDIA's parallel thread execution (PTX) virtual instruction set and simulates every detail of the GPU. It is thus very useful for obtaining insights into architectural design problems for GPUs. However, no comparison to an actual GPU is provided in [BYF+09] and although the trends predicted by GPGPU-Sim are certainly interesting, it is not clear that it can be used to perform accurate performance prediction of a real hardware. A few other GPU-specific simulators have therefore been developed (e.g., Barra [CDDP10] for the NVIDIA G80 or Multi2Sim [UJM+12] for the AMD Evergreen GPU). Such specialization allows Multi2sim to report predictions within 5 to 30% of native execution for several OpenCL benchmarks. While this prediction is quite impressive, it comes at the price of a very long simulation time as every detail of the GPU is simulated. The average slowdown of simulations versus native execution is reported to be $44,000\times$ while the one of GPGPU-Sim on a similar scenario is about $90,000\times$[UJM+12].

Another approach, similar to the one described for the CPUs, is a simple delay-based model. Parts of the application that are normally executed on GPUs are carefully benchmarked on the target machine. These results are used to construct performance models needed by the simulator so it can compute the timings that should be injected during the simulation. Such approach is even more valid for GPUs than for CPUs, as their architecture is much simpler, making the computation block durations easier to predict.

3.2.3 Simulating applications

Simulating MPI applications

There are different types of HPC applications. Some are iterating through clearly divided phases, where first processes compute certain tasks and after that they are all exchange messages with the necessary data. In the other group, there are applications that can be described as tasks and try to execute these tasks as soon as possible without predefined order, using opportunistic scheduling.

MPI applications typically have very regular computation/communication block patterns and are thus a good candidate for accurate simulations. There are two main approaches for simulating MPI applications: *off-line* and *on-line*.

Off-line (or "post-mortem") simulation is based on replaying previously obtained traces. This means that the application is executed once on a real platform, either a target or a host machine,

logging beginning and end time of every computation and communication, size of the exchanged messages, memory footprint, etc. Later on a host machine, this log is injected into simulator, possibly modifying the timings according to the different characteristics of the target machine. Such solution allows fast simulations of large platforms providing very useful information to the MPI application developers. Therefore, many tools based on this approach have been developed in the recent years [ZKK04, TLCS09, NnFG+10, ZCZ10, HGWW09, BLGE03].

Although this approach can provide very accurate predictions, it is limited to the specific settings. Indeed the order of different computation blocks is not deterministic for all applications and it may depend on the order in which certain messages are sent or received, which itself depends on network characteristics.

Moreover, if one wants to modify platform description, typically increasing the number of nodes, the application patterns are likely to change as well. There are techniques proposed by [WM11] or [CLT13] that allow for extrapolating the behavior at large scale of a parallel code from a few traces at lower scale. However, the confidence in such predictions is questionable as different setups often introduce some new phenomena that is difficult to predict. For example, the number of nodes will dictate the algorithms' load balancing of the work, creating more or less computation/communication blocks for each node. Additionally, this will influence the size of the messages exchanged between the nodes, and in fact this size depending on the MPI implementation can determine which mode of communication is used [BDG+13]. These modes have very different latency and bandwidth characteristics, and they can even have different patterns for the collective communication operations.

Still, there are use cases where the studied applications are deterministic and there is very little contention due to interactions between the worker processes. The SuperLUDIST [LD03] simulator was used in the recent study by Cicotti and al. [CLB09] for simulating parallel sparse Cholesky and LU factorizations. The objectives of this research are very similar to ours (presented in Chapter 7), as the authors aim at predicting the performance of sparse linear algebra applications. Their simulation framework consists of the tools used for benchmarking machine characteristics and of a simulation module specifically built for this application. The approach is based on three types of models: memory, kernel and communication. Memory models are generated from the cache simulator that preserves the whole state of the memory hierarchy, simulating every read and write operation. Such approach contains all the shortcomings of the cycle-accurate CPU simulators described in 3.2.2. The kernel models are based on the micro-benchmarks of BLAS routines. When running simulation, timings measured by benchmarks for specific dimensions are injected into simulator and in case there is no value for a requested dimension, the estimation computed by a linear interpolation is provided. Although such approach may provided satisfactory results in certain cases, we strongly believe that proper modeling (especially for the parameterized kernels) demands for a more profound analytical and statistical studies. Finally, communication models are taking into account different size of the messages, but completely ignore possible network contentions. The predictions presented by the authors are extremely accurate and the case studies seem very useful. However, such approach is unlikely to work with other, more irregular and dynamic sparse solvers such as MUMPS.

The main disadvantage of a simulation completely based on input traces or component descriptions is that it implies deterministic execution of the application. Such approach disregards all the noise and perturbations introduced by the operating system and the machine environment in general, which have an important influence on the application performance on modern computers.

On-line MPI simulations try to cope with this challenges through the actual executions of the code. The computation parts are executed on the host machines, while communication calls are intercepted and passed to the simulator that again uses models to predict transfer durations. This approach ensures that the computation/communication block patterns are correct even if the number of nodes is increased or if the order of certain operations is changed. However, since the intensive computations are actually executed, this solution is much slower than the classical off-line one. Therefore, simulators using this approach [PWTR09b, DHN96, BDP01b, Rie06] typically add certain optimizations to improve simulators' performance and decrease simulation time.

Simulating task-based runtimes

Task-based runtime execution consists of running through a DAG that contains all the data dependencies between the tasks. During the application execution, these tasks are scheduled to be executed on the appropriate workers, typically CPUs or GPUs. The decision on which task is going to be run on which worker is made dynamically, following the current state of the whole system and some scheduling heuristics. Even if the heuristics stay the same, from one run to another, the state of the system will evolve differently, since it is influenced by the environment in which the code is executed. Consequently, the tasks are not going to be executed in the same order on the same resources.

Simulating such an irregular execution is much harder than for classical MPI applications. However, solutions can again be divided into off-line(trace based) and on-line.

The BigSim [ZKK04] simulator proposes replaying traces generated by the CHARM++ runtime [KK96]. This tool can use parallel discrete event simulation in order to scale better for large problems. Still, to simulate a huge cluster of machines, BigSim requires another (smaller) cluster. Traces generated by the simulator are very rich and enable various analysis and visualizations studies of the executed code. This can be very useful to the program developers, as after the initial trace acquisition, they can replay it on a smaller machine and rapidly evaluate their applications. The trace on which simulation predictions are based however represents a unique execution of the runtime. Therefore, any specificities of that single run with all the captured and uncaptured phenomena will be present in all the following simulations, which is a considerable limitation of this approach.

Another off-line approach is used with TaskSim [RCV+12] that has recently been coupled with the NANOS++ runtime system (on which OmpSs [ABI+09] is based) to provide predictions built from multiple levels of abstraction. This trace-driven simulator uses runtime as a dynamic component for potential rescheduling of the task, based on the machine characteristics [RDC+11]. However, coming from computer architecture experts, this tool aims at being an alternative to classical cycle-accurate simulators, such as Dinaro IV [Din]. Even though the reported performance is much better than cycle-accurate simulations, this solution would not scale for larger runtime executions on big platforms. Additionally, to the best of our knowledge, this tool addresses so far only multi-core machines, without GPUs.

Nowadays, many researchers working with HPC applications, especially parallel linear algebra, tend to implement them using dynamic scheduling of computation kernels in order to maximize the performance on various architectures. In such context, simulations via trace replay is inadequate, as even the real executions greatly vary from one run to another due to the opportunistic scheduling. In order to fully preserve dynamic behavior, the runtime must be the part of the simulation.

Researchers from University of Tennessee Knoxville developed such a simulation tool [HKY+14] on top of three widely used runtimes: OmpSs, StarPU and QUARK. Their solution is based on carefully benchmarking computational kernels on the target machine, and then on injecting timings according to the distributions of the measured values. During the simulation, the actual runtime is executed on the host machine, and it is responsible for all data and task managements. Since computational kernels are not executed, such solution makes time to run the simulation much smaller than the native execution and the authors reported very accurate simulator predictions. Moreover, their solution manages three different runtimes, that all have different internal structure, which is a strong point of this work. However, certain abstractions and simplifications necessary to implement this solution limit its usage. First, extending this tool on hybrid or distributed machines would be extremely hard as so far it does not include any network model, not to mention network contention model. Second, even though the authors try to minimize the influence of the cache contention caused by the parallel thread execution, they do not address the question of the non-uniform time to access the data present on NUMA machines. When executing applications on machines that consists of multiple nodes, the time to fetch the data from the local and distant CPU cache is different, making the time to execute a computational kernel variable. Finally, the authors measured that in their experiments standard BLAS and LAPACK routines have a stable behavior

that can be approximated with a normal distribution. The observed variations are due to the cold or warm cache state at the beginning of the kernel execution. However, our studies showed that such behavior is present only in the well-balanced executions of relatively small problems. Longer executions on complex platforms tend to demonstrate much more variability, and abstracting those with a simple distribution is not possible. Moreover, kernels running on GPUs tend to have even more irregular performance, as it greatly depends on the location of the data (see Section 6.2).

On-line simulation of the runtime appears to be the right approach, but implementing an ad hoc simulator on top of a complex runtime code base is not straightforward. Such solutions are however short-lived as they vastly depend on the specific version of the runtime and any major evolution of runtime typically requires significant changes to the simulator. Additionally, any tool produced in this way is limited to only certain use cases. Once it can provide the accurate predictions, researchers immediately demand for more complicated scenarios and machine setups, which are again hard to extend due to the poor modularity of the initial approach. Therefore, it is highly advisable to build solution using an already developed simulation framework that has a solid internal simulation engine with the appropriate API for linking with the runtime.

3.2.4 SimGrid: a toolkit for Simulating Large Heterogeneous Systems

In the context of tuning HPC runtimes, expectations in term of simulation accuracy are extremely high. It is thus difficult to rely on a simulator that may provide the right trends but with a 50% over/under estimation. Choosing the right level of granularity or the correct scheduling heuristic can not be done without precise and quantitative predictions. Such inaccuracies can come typically from an inadequate level of details and should be avoided. Therefore, we propose to use a top-down modeling approach such as promoted by the SimGrid project [CGL+14], which provides a *versatile* simulation toolkit to study the behavior of large-scale distributed systems like grids, clouds, or peer-to-peer systems. It is performance oriented and scalable, using a delay-based approach rather than slow cycle-accurate simulations of the resources.

SimGrid builds on fluid network models that have been proven as a reasonable alternative to both simple analytic models and expensive, difficult-to-instantiate packet-level simulations [VSCL13] and have been extended to simulate accurately MPI applications on Ethernet networks [BDG+13]. Fluid models are used by simulators in different domains [BCM+03, CLQ08, OPF10, GB02, ZKK04]. These models represent communications by *flows* that are simulated as single entities rather than as sets of individual packets. The bandwidth allocated to flows is constrained by the network resource capacity. Transfer time of a message of the size S between hosts i and j is computed with the following formula:

$$T_{i,j}(S) = L_{i,j} + S/B_{i,j},$$

where $L_{i,j}$ and $B_{i,j}$ represent the end-to-end network latency and bandwidth, respectively, of the route connecting i and j hosts. While latency parameter is independent and computing it may be straightforward, bandwidth depends on all other flows that share parts of the route. Hence, to determine how much bandwidth is allocated to each flow, the following formal problem has to be solved:

> *Consider a connected network that consists of a set of links L, in which each link l has capacity B_l. Consider a set of flows F, where each flow is a communication between two network vertices along a given path. Determine a "realistic" bandwidth allocation ρ_f for flow f, so that*:

$$\forall l \in L, \sum_{f \text{ going through } l} \rho_f \leq B_l \ .$$

Therefore, with a computed bandwidth allocation and the size of the data that needs to be transferred by each flow, one can determine which flow will complete first. Each time a flow has

completed its transfer or another flow has started, the bandwidth allocation is reevaluated. This allows for quickly stepping forward through simulated time, which makes this approach very scalable. Fluid network models generally assume steady-state of the flows, and thus such models ignore all transient phases between two steady-state operation points. However, this approach is are very flexible and allows for easily accounting for the network topology and heterogeneity as well as many non-trivial phenomena (e.g., RTT-unfairness of TCP or cross-traffic interferences) [VSCL13] at a very low simulation cost.

SimGrid can simulate computation blocks in different ways, depending on the application requirements. For example, both *off-line* and *on-line* simulations of the MPI applications are allowed. In a post-mortem simulation, computation timings and communication pattern are injected from the previously obtained trace (possibly adjusting the values according to the target machine characteristics). On the other hand, in on-line simulations the real code will be executed on the host machine, thus directly injecting measured values into simulator. Many other scenarios are also feasible for other use cases, and such flexibility is a strong point of this simulator.

In our research, we decided to investigate how SimGrid could be used to simulate runtime systems. We believe that a coarse-grain approach used by this framework is the right method to abstract the huge complexity of the studied systems and the heterogeneity of the machines. Moreover, SimGrid is modeled with threads rather than states and transitions, which is the easiest way to simulate dynamic applications. Additionally, SimGrid is portable across different platforms and operating systems, and due to its modular internal organization it is very easy to extend. All this makes him a perfect candidate for our task.

Chapter 4

Methodology

Our research is centered on performance evaluation of modern computer systems. To validate the approach and models we developed, we had to perform numerous measurements on a wide variety of machines. Some of them are part of larger platforms (e.g., Grid5000) for which a have whole team of engineers is dedicated to provide optimal and reproducible environment to their users. However, even these systems go through occasional hardware renewals, which makes experiments impossible to reproduce. Another group of machines we used belong to regional centers comprising both production and experimental clusters, where researchers share resources and where the access is not fully dedicated. Finally, during our research we also had the access to the newest ARM processors or NVIDIA K40 accelerators which we plugged in custom machines by hand. We were doing that only temporarily for a few weeks by installing all necessary software layers and running several sets of experiments. After the experiments were completed, we gave these chips to other researchers so that they can perform their own studies. In such process, we completely lost control on these experimental setup and reproducing experiments became quite haphazard.

Another important challenge for maintaining the reproducibility of our experiments comes from the heterogeneity of the platforms. Experiments are conducted on prototype hardware that sometimes contains a huge number of cores on the same node and a custom interconnect. Others have GPUs with different CUDA installations. Moreover, when dealing with processors initially designed for mobile phones and tablets like ARM processors, an other type of problems arises. These machines have limited resources, both in terms of memory, CPU power and disk space. Even though these machines support operating systems based on Unix, software packages still have very restricted availability. Therefore, experimental workflow on such computers has to be lightweight and with minimal dependencies.

Additionally, on some platforms we have only limited control and access to the environment setup, as the machines are strictly managed by the administrators who maintain and update its configuration so that it matches the needs of most users. Even that would not be so harmful if the image of the whole system could be saved and later redeployed, like on Grid5000 clusters or on some machine belonging to the local laboratory. However, this is often not the cases, and the experimenters have to work with environments on which they have very little control.

In such context, a presumably minor misunderstanding or inaccuracy about some parameters at small scale can result in a totally different behavior at the macroscopic level. Mytkowicz et al. [MDHS09] show how unpredictable and unexpected the measurement bias can be. Using different compilation flags and slightly changing the linking order for a simple single-threaded benchmarking application can not only produce different results, but also lead to completely incorrect conclusions. We also encountered problems with inconsistent results in the initial phase of our research, when we conducted a comparative study of CPU cache performance on various Intel and ARM micro-architectures [8]. For that we used a simple memory benchmark, similar to MultiMAPS [SCW+02], which measures memory bandwidth when accessing consecutive fields of a varying size array, in a loop. Once all data has been accessed for the first time, if the whole array can fit the L1 cache then any new access will be fast, resulting in high memory bandwidth values.

On the other hand, if the array size is larger than the size of L1 cache, repetitively accessing all fields in a loop will result in many cache misses and thus will degrade the performance.

Figure 4.1 shows the result of 4 consecutive experiments on an ARM Snowball processor using exactly the same source code and inputs. 42 repetitions for each memory size (on each plot) are represented by boxplots. One can observe that there is very little variability in each experiment set, but that the expected performance drop occurs at different places. Extreme values of array memory size always exhibit the same behavior, but the middle part (from 50% to 100% of the L1 cache size) is unpredictable. Without going into details, after some efforts, we finally found the source of this surprising phenomenon comes from the way operating system allocates physical memory pages on ARM processor. In general, operating systems allocate nonconsecutive physical memory pages, choosing them randomly from a pool of available pages. Since the set-associativity of that generation of ARM processors is 4, while the L1 cache size is 32KB, without doing the appropriate page coloring [Pag] bad choice of the physical pages will causes much more cache misses, hence the drop of overall performance. During one experiment run, the same pages are reused as we do malloc/free repeatedly on each array size. Hence, the arrays start from the same physical memory location for each memory size during one experiment, which explains why there is no variability in the results despite the randomization of the measurements.

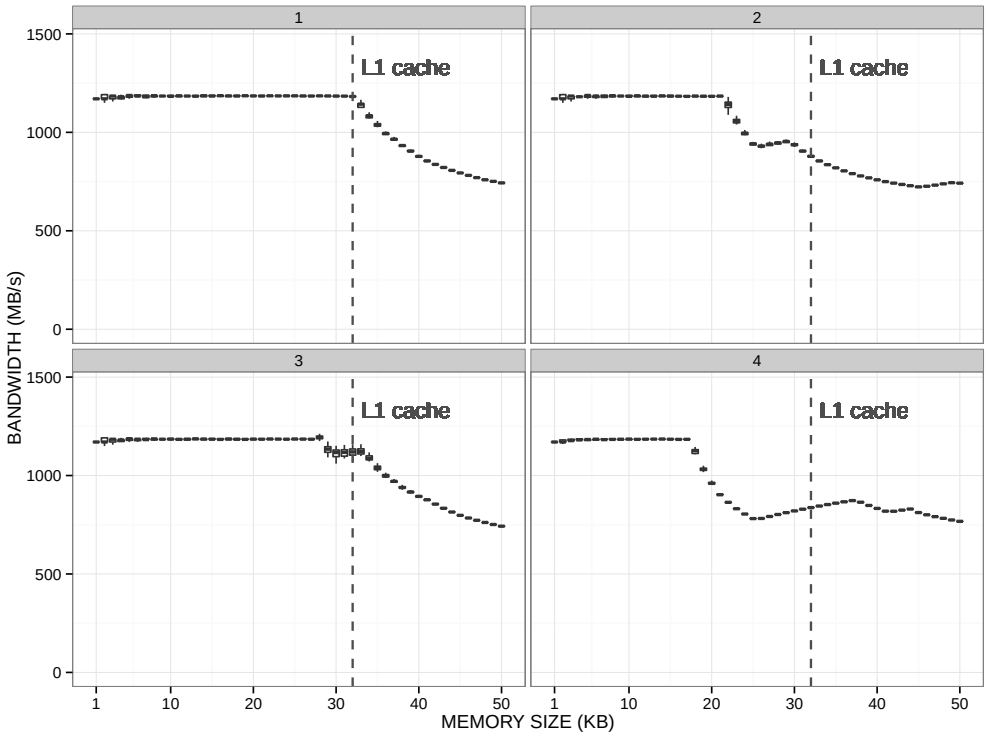


Figure 4.1: Reproducibility issue on ARM Snowball: 4 consecutive experiments with identical input parameters behaving differently; 42 repetitions for each array size depicted by boxplots show no noise within each single experiment.

What we learned from this example is that it is crucial to carefully collect all the useful metadata and to use well-planed experiment designs along with coherent analyses. Such details are essential for a better understanding of the results and a good reproducibility of the experiments.

There is no standard way to perform research and measurements in our context. Many solutions were presented in 3.1, but neither of them is applicable for us. Our goal was however not to implement a new tool for conducting experiments, but rather to find a good combination of already existing ones that would allow a painless and effective daily usage. We were driven by

purely pragmatic motives, as we wanted to keep our workflow as simple and comprehensible as possible while offering the best possible level of confidence in the reproducibility of our results. Thus, we decided to work with a minimalistic set of simple, lightweight, and well-known tools.

We use **Org-mode**, initially an Emacs mode for editing and organizing notes, that is based on highly hierarchical plain text files which are easy to explore and exploit. Org-mode has also been extended to allow combining plain text with small chunks of executable code (Org-babel [SDDD12] snippets). Such feature builds on the literate programming principles introduced by Donald Knuth [Knu84] three decades ago, and for which there has been a renewed interest in the last years. Although such tool is very convenient for conducting experiments and for writing scientific documents, its use is not so common yet.

In addition, for version control system we decided to rely on **Git**, a distributed revision control tool that offers an incredibly powerful and flexible branching mechanism.

We propose and describe in Subsection 4.1.1 a novel Git branching model for managing experimental results synchronized with the code that generated them. We have been using such branching model for three years now and we identified a few typical branching and merging operations, that we are currently packaging for the Debian Linux system.

Such branching model eases provenance tracking, experiments reproduction and data accessibility. However, it does not address issues such as documentation of the experimental process and writing the conclusions about the results, nor the acquisition of meta-data about the experimental environment. To this end, in Subsection 4.1.2 we complete our branching workflow with an intensive use of Org-mode, which enables us to manage and keep in sync experimental results and meta-data. It also provides literate programming through a laboratory notebook which is very convenient and eases the edition of reproducible articles. We explain in Subsection 4.1.3 how the laboratory notebook and the Git branching can be nicely integrated to ease the set up of a reproducible article.

Through the whole Section 4.1, we demonstrate the effectiveness of this approach by providing examples. We illustrate several points in the discussion by pointing directly to specific commits inside our project repository.

Although we opened our whole Git repository for illustration purposes, this is not required by our workflow. There may be situations where researchers may want to share only parts of their work. We discuss in Section 4.2 various code and experimental data publishing options that can be used within such a workflow.

4.1 A Git and Org-mode based workflow

In this section, we present our workflow for conducting experiments and writing articles about the results, based on a unique Git branching model coupled with a particular usage of Org-mode. Although these are well-known and widely used tools, to the best of our knowledge no one so far has proposed using them in a similar manner for doing reproducible research. The approach we present is lightweight, to make sure the experiments are performed in a clean, coherent and hopefully reproducible way without being slowed down by a rigid framework. It fully fulfilled our needs for conducting large experimental campaigns and we believe it would equally help anyone doing research in a such context. However, the main ideas behind our solution are general and can be applied to other fields of computer science as well, possibly with some domain specific adjustments as the environment in which experiments are performed can be very different.

We remind the reader that every document and series of commits described herein can be found at [SSW]. Links to Git commits with examples are provided in the rest of this thesis and we encourage readers to inspect them. All our documents are plain text files and can thus be opened with any text editor or browser. However, since most of these files are Org-mode documents, we suggest to open them with a recent Emacs and Org-mode installation rather than within a web browser. Options for pretty printing in web browsers exist, but are not fully mature yet. We are currently working on making it easier for non Emacs users to easily exploit such data, following the principles used by GitHub developers.

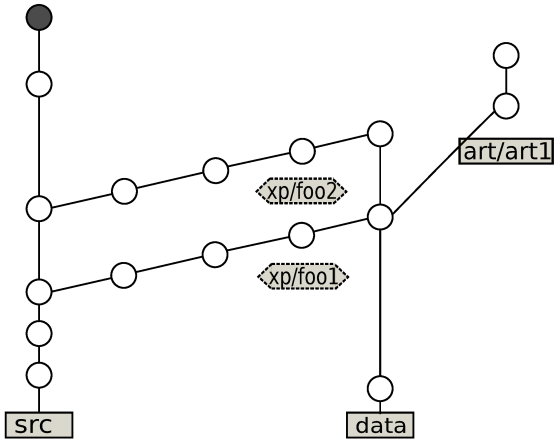


Figure 4.2: Proposed Git branching scheme with 4 different types of branches.

4.1.1 Git branching structure

Relying on a revision control system for under development code is nowadays a common practice. It is thus a good habit to ensure that all source code changes are committed before running experiments. These restrictions are already implemented in many tools, such as Sumatra [SLP14, chap.3]. Knowing which revision of the source code was used to produce a given data makes it theoretically possible to reproduce experimental results. However, in practice it can be quite burdensome.

A first problem is that code and experimental results are often treated as two separate kinds of objects. Being only loosely coupled, the link between some data and the code that produced it is often hard to find and exploit, and sometimes it can even be completely lost. Therefore, we suggest to store both of them in the same Git repository so as to ensure that they are always perfectly synchronized. This greatly increases the confidence level in the results and makes it easier to obtain the code that produced a particular data set.

Storing everything in the same repository can quickly lead to an anarchic and unexploitable system and hence it requires some organization and convention. To this end, we propose an approach that consists in 4 different types of branches, illustrated in Figure 4.2. The first branch, named *src*, includes only the source code, i.e., the code and scripts required for running the experiments and simple analyze. The second branch, *data*, comprises all the source code as well as all the data and every single analysis report (statistical analysis results, figures, etc.). These two branches live in parallel and are interconnected through a third type of branches going from *src* to *data*, the *xp#* branches (the "#" sign means that there are multiple branches, but all with the same purpose). These are the branches where all the experiments are performed, each *xp#* branch corresponding to one set of experimental results. The repository has typically one *src* and one *data* branch, both started at the beginning of the project, while there is a huge number of *xp#* branches starting from *src* and eventually merged into *data* that accumulates all the experimental results. All these together induce a "ladder like" form of Git history. Finally, the fourth type of branches is the *art#* branches which extend the *data* branch. They comprise an article source and all companion style files, together with a subset of data imported from the *data* branch. This subset contains only the most important results of the experiments, that appear in the tables and figures of the article.

By using Git as proposed, it is extremely easy to set up an experimental environment and to conduct the experiments on a remote machine by pulling solely the head of the *src* or of an *xp#* branch. This solves the problem of long and disk consuming retrieving of the whole Git repository, as the *src* and *xp#* branches are typically very small.

On the other hand, one might want to investigate all the experimental data at once, which can be easily done by pulling the head of *data* branch. This is meant for researchers that are

not interested in the experimentation process, but only in the analysis and cross-comparison of multiple sets of results. For such users, the *src* and *xp#* branches are completely transparent, as they will retrieve only the latest version of the source code (including analysis scripts) and the whole set of data.

Another typical use case is when one wants to write an article or a report based on some experiment results. A completely new branch can then be created from *data*, selecting from the repository only the data and analysis code needed for the publication and deleting the rest. This way, the complete history of the study behind the article is preserved in the git structure (e.g., for the reviewers) while the article authors can download only the data set they really need.

When used correctly, such Git repository organization can provide numerous benefits to the researchers. However, it is not sufficient in our setting, since commit messages in Git history give only coarse-grain indications about source code modifications. There is still a lot of information missing about the environment setup of the machines, why and how certain actions were performed and what the conclusions about the results are. We address all these questions with Org-mode files, as described in the following subsection.

4.1.2 Using Org-mode for improving reproducible research

As mentioned in Section 3.1, several tools can help to automatically capture environment parameters, to keep track of the experimentation process, to organize code and data, etc. However, none of them addresses these issues in a way satisfying our experimental constraints, as these tools generally create new dependencies on specific libraries and technologies that sometimes cannot be installed on experimentation machines. Instead, we propose a solution based on plain text files, written in the spirit of literate programming, that are self-explanatory, comprehensive and portable. We do not rely on a huge cumbersome framework, but rather on a set of basic, flexible `shell` scripts, that address the following challenges.

Environment capture

The environment capture aims at getting every detail about the code, the libraries in use and the system configuration. Unlike the parallel computing field where applications are generally expected to run in more or less good isolation of other users/applications, there are several areas of computer science (e.g., networking, security, distributed systems, etc.) where fully capturing such platform state is impossible. However, the principle remains the same, as it is necessary to gather as much useful meta-data as possible, to allow comparison of experimental results with each others and to determine if any change to the experimental environment can explain potential discrepancies. This process should not be burdensome, but automatic and transparent to the researcher. Additionally, it should be easy to extend or modify, since it is generally difficult to anticipate relevant parameters before performing numerous initial experiments.

Thus, we decided to rely on simple `shell` scripts, that just call many Unix commands in sequence to gather system information and collect the different outputs. The meta-data that we collect typically concern users logged on the machine during the experiments, the architecture of the machine (processor type, frequency and governor, cache size and hierarchy, GPU layout, etc.), the operating system (version and used libraries), environment variables and finally source code revisions, compilation outputs and running options. This list is not exhaustive and would probably need to be adjusted for experiments in other domains.

Once such meta-data is captured, it can be stored either individually or accompanying results data. One may prefer to keep these two separated, making the primary results unpolluted and easier to exploit. Although some specific file systems like HDF5 (as used in ActivePapers [Hin11]) provide a clean management of meta-data, our experimental context, where computing/storage resources and our ability to install non-standard software are limited, hinders their use. Storing such information in another file of a standard file system quickly makes information retrieval from meta-data cumbersome. Therefore, we strongly believe that the experiment results should stay together with the information about the system they were obtained on. Keeping them in the same

file makes the access straightforward and simplifies the project organization, as there are less objects to handle. Furthermore, even if data sustains numerous movements and reorganizations, one would never doubt which environment setup corresponds to which results.

In order to permit users to easily examine any of their information, these files have to be well structured. The Org-mode format is a perfect match for such requirements as its hierarchical organization is simple and can be easily explored. A good alternative might be to use the yaml format, which is typed and easy to parse, or to develop a new specific format similar to efforts made in bioinformatics with the ISA software suite [RSBM+10]. However, we decided to stay with Org-mode (which served all our needs) to keep our framework minimalist.

A potential issue of this approach is raised by large files, typically containing several hundreds of MB and more. Opening such files can temporary freeze a text editor and finding a particular information can then be tedious. We have not yet met with such kind of scenario, but it would certainly require some adaptations to the approach.

In the end, all the data and meta-data are gathered automatically using scripts (e.g., 41380b54a7{run-experiment.sh#l220} or in Appendix B.1), finally producing a read-only Org-mode document (e.g., 1655becd0a{data-results.org} or in Appendix B.2) that serves as a detailed experimental report.

The motivations for performing some experiments and observations about the results are stored separately in the laboratory notebook.

Laboratory notebook

A paramount asset of our methodology is the laboratory notebook (labbook), similar to the ones biologist, chemists and scientist from other fields use on a daily basis to document the progress of their work. For us, this notebook is a single file inside the project repository, shared between all collaborators. The main motivation for keeping a labbook is that anyone, from original researchers to external reviewers, can later use it to understand all the steps of the study and potentially reproduce and improve it. This self-contained unique file has two main parts. The first one aims at carefully documenting the development and use of the researchers' complex source code. The second one is concerned with keeping the experimentation journal.

Documentation This part serves as a starting point for newcomers, but also as a good reminder for everyday users. The labbook explains the general ideas behind the whole project and methodology, i.e., what the workflow for doing experiments is and how the code and data are organized in folders. It also states the conventions on how the labbook itself should be used. Details about the different programs and scripts, along with their purpose follow. These information concern the source code used in the experiments as well as the tools for manipulating data and the analysis code used for producing plots and reports. Additionally, there are a few explanations on the revision control usage and conventions. Moreover, this part of the labbook contains a few examples how to run scripts, illustrating the most common arguments and format. Although such information might seem redundant with the previous documentation part, in practice such examples are indispensable even for experienced users, since some scripts have lots of environment variables, arguments and options. It is also important to keep track of major changes to the source code and the project in general inside a ChangeLog. Since all modifications are already captured and commented in Git commits, the log section offers a much more coarse-grain view of the code development history. There is also a list with a brief description of every Git tag in the repository as it helps finding the latest stable or any other specific version of the code.

Experiment results All experiments should be carefully noted in this part, together with the key input parameters, the motivation for running such experiment and the remarks on the results. For each experimental campaign there should be a new entry that answers to the questions "why", "when", "where" and "how" experiments were run and finally what the observations on the results are. Inside the descriptive conclusions, Org-mode allows for using both links and

git-links connecting the text to specific revisions of files. These hyperlinks point to crucial data and analysis reports that illustrate a newly discovered phenomenon.

Managing efficiently all these different information in a single file requires a solid hierarchical structure, which once again motivated our use of Org-mode. We also took advantage of the Org-mode tagging mechanism (not to be mistaken for Git tags), which allows for easily extracting information, improving labbooks' structure even further. For example, tags can be used to distinguish which collaborator conducted a given set of experiments and on which machine. Although such information may already be present in the experiment files, having it at the journal level proved very convenient, making the labbook much easier to understand and exploit. Experiments can also be tagged to indicate that certain results are important and should be used in future articles.

Several alternatives exist for taking care of experiment results and progress on a daily basis. We think that a major advantage of Org-mode compared to many other tools is that it is just a plain text file that can thus be read and modified on any remote machine without requiring to install any particular library, not even Emacs. Using a plain text file is also the most portable format across different architectures and operating systems.

To illustrate our approach we provide two examples of labbook files. The first one comprises only the documentation parts related to the code development and usage (30758b6b6a{labbook} or in Appendix B.3) and is obtained from the *src* branch or from the beginning of an *xp#*, while the second one (01928ce013{labbook#l272} or in Appendix B.4) has a huge data section comprising the notes about all the experiments performed since the beginning of the project.

Using literate programming for conducting experiments

In our field, researchers typically conduct experiments by executing commands and scripts in a terminal, often on a remote machine. Later, they use other tools to do initial analysis, plot and save figures from the collected data and at the end write some remarks. This classical approach has a few drawbacks, which we try to address using Org-babel, Org-mode's extension for literate programming.

The main idea is to write and execute commands in Org-babel snippets, directly within the experimentation journal, in our case the labbook. This allows for going through the whole experimentation process, step-by-step, alternating the execution of code blocks and writing text explanations. These explanations can include reasons for running a certain snippet, comments on its outputs, plan for next actions or any other useful remarks. At the end, this process can be followed by a more general conclusion on the motives and results of the whole experimentation campaign. Conducting experiments in such manner provides numerous benefits comparing to the usual way scientists in our field work.

The first problem with the classical approach is that researchers save only the experiment results (possibly with some meta-data), while all other seemingly irrelevant outputs of commands are discarded. However, in case of failures, these outputs can occasionally be very helpful when searching for the source of an error. Although, such outputs, along with the commands that produced them, can sometimes be found in a limited terminal history, their exploration is a very tedious and error-prone process. On the other hand, when using Org-babel, all snippet results are kept next to it, which simplifies the tracing of problems.

Second, since the preparation and management of experiments is a highly repetitive process, grouping and naming sequences of commands in a single snippet can be very beneficial. This allows for elegantly reusing such blocks in future experiments without writing numerous scripts or bulky "one-liners".

Additionally, Org-babel permits to use and combine several programming languages, each with its own unique purpose, inside the same file. This again decreases the number of files and tools required to go through the whole experimentation process, making it simpler and more coherent.

Last, and probably the most important point, using this approach avoids documenting an experimental process afterwards, which is generally tedious and often insufficient. Researchers

are frequently in a hurry to obtain new data, especially under the pressure of strict deadlines. They do not dedicate enough time to describe why, where and how experiments were performed or even sometimes what the conclusions about the results are. At that time, the answers to these questions may seem obvious to the experimenters, hence they often neglect noting it. However, after a few days or months, remembering all the details is not so trivial any more. Following literate programming principles and taking short notes to explain the rationale and usage of code snippets, while executing them, is quite natural and solves the previous issues. From our own experience, it does not significantly slow down the experimental process, while it provides huge benefits later on.

Finally, the outcome of this approach is a comprehensible, well-commented executable code, that can be rerun step-by-step even by external researchers. Additionally, it can also be exported (tangled), producing a script that consists of all snippets of the same language. Such scripts can be used to completely reproduce the whole experimentation process.

An example of this approach is provided in 0b20e8abd5{labbook#l950} or in Appendix B.5. It is based on Shell snippets, and although it can be rerun only with the access to the experimental machines, it provides both a good illustration of Org-babel usage for conducting experiments and a faithful logging of the commands run to obtain these experimental data, which is paramount for a researcher willing to build upon it.

4.1.3 Git workflow in action

We now explain the typical workflow usage of our Git branching scheme, that is also tightly linked to the experimentation journal in Org-mode.

Branching and merging is a technical operation that can become cumbersome and generally scares new Git users. That is why such Git interactions should be made as easy as possible with new specific Git commands, which we are currently packaging. We introduce such commands along with their intended use without going into the details of their implementation.

Managing experiments with Git

On Figure 4.3 we explain the typical workflow usage of our branching scheme. Although it is self-contained and independent from any other tool, we found it very practical to couple it with our laboratory notebook.

Before even starting, the whole project needs to be correctly instantiated on every machine, as shown in Phase 0. The `git setup url` command will clone the project from server, but without checking out any of the branches. Additionally, it will take care of several configuration options regarding our future Git commands, ensuring that the workflow works smoothly.

When everything is set, the researcher can start working on a code development inside the *src* branch, committing changes, as shown in Phase 1. These modifications can impact source code, analysis or even the scripts for running the experiments. Later, such modifications should be tested and the correctness of the whole workflow should be validated. Only then can one start conducting real experiments by calling `git xp start foo`. This command will create and checkout a new branch *xp/foo* from the *src*. Then, this command will create, commit and push a new folder for storing the results. We used the convention that these two (branch and folder) should always have the same name, which eases the usage of both Git and labbook. Next, the newly created branch is pulled on a remote machine B, using `git xp pull foo`. It will fetch only the last commit of the *xp/foo* branch. As a result, machines for experimentation, such as machine B, get only the code required to run the experiments, with neither the Git history nor any experimental data. Meanwhile, machine A and all other users that want to develop code, do the analysis and write articles will continue using the standard `git pull` command to get a complete Git repository, although it can sometimes be quite memory and time consuming.

In Phase 2, we first verify that there has not been any code modification before running the experiment and we also automatically ensure that the latest version of the code has been compiled. Then, experiments are run, generating new data. The resulting Org-mode data files, containing

Phase	Machine A	Machine B	Git server
0	**git setup url**	**git setup url**	
1	**git xp start foo**	**git xp pull foo**	xp/foo, src, data
2		**git status** *do experiments* **git add results** **git commit** **git push**	xp/foo, src, data
3	**git pull** *do analysis* *write to labbook* **git commit** **git push**		xp/foo, src, data
4	**git xp finish foo**		xp/foo, src, data

Figure 4.3: Typical Git experimentation workflow with different phases, using two machines.

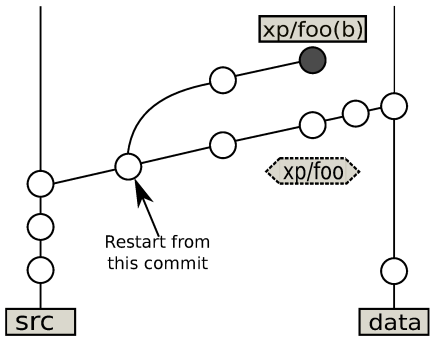


Figure 4.4: Restart or reproduce experiments starting from a certain commit.

experiment outputs together with the captured environment meta-data, are then committed to the Git repository. Such process may be repeated, possibly with different input parameters. Finally, the commited data is pushed to the server.

After that, the experiment results can be pulled on the machine that is used to do the analysis (Phase 3). Important conclusions about the acquired data should be saved either in separate reports, or even better as a new *foo* entry inside the experiment results section in the labbook. Results of the analysis could later trigger another round of experimentation and so on.

Finally, when all desired measurements are finished, *xp/foo* is merged with the *data* branch using `git xp finish foo`, as depicted in Phase 4. This command also deletes the *foo* branch, to indicate that the experimentation process is finished and to avoid polluting the repository with too many open branches. Still, a simple Git tag is created on its place, so if needed, the closed branch *foo* can easily be found and investigated in future. Note that while in the *src* branch, the labbook only has the documentation part, the *xp#* branches are populated with observations about the experiments. Therefore, the merged labbook in the *data* branch holds all the collected experimental entries with comments, which makes their comparison straightforward.

One interesting option is to go through the entire workflow depicted in Figure 4.3 directly within the labbook, using the literate programming approach with Org-babel we described in Section 4.1.2.

Reproducing experiments

The main goal of such workflow is to facilitate as much as possible the reproduction of experiment by researchers. This can be done by calling the `git xp start --from foo` command, from the machine we want to repeat the experiments. As displayed in Figure 4.4, this command will checkout the desired revision of the code and create a new branch and folder based on the initial *xp#* branch. From there, conducting the new experiments, noting the observations and later merging with *data* branch is performed as usual.

It may happen that software components of the machines used for experiments are replaced between two series of experiments. In many cases, this is done by the machine administrators and the researchers conducting the experiments may have no permission to revert it. There could thus be some important changes in the environment and repeated experiments might produce different results from the initial ones. Unfortunately, when dealing with experiments that cannot be run on virtual machines, nothing can be done to avoid this problem. The best we can do is to carefully track all the software used in the experiments. Therefore, if any significant deviation of the experimental results occurs, we can compare the meta-data and find the source of discrepancy.

It is also worth mentioning that if researchers want to reconduct a previous experiments, but on a completely new machine, they will use this exact same approach.

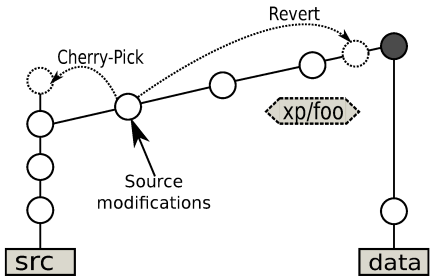


Figure 4.5: Handling source modifications that occurred during the experimentation.

Fixing code

Sometimes, while conducting the experiments on a remote machine, the researcher may need to make few small source code modifications. These modifications have to be committed (since measurements are never done using an uncommitted code), even though in most cases they represent an *ad hoc* change, specific to an individual machine and its current installation. These minor, local hacks would pollute the *data* branch and thus it has to be ensured that they are not propagated when branches are merged. This protection is thus implemented inside the `git xp finish foo` command.

At the end of the *foo* branch, all source code changes are reverted. This means that we create an "anti-commit" of all the previously committed source modifications inside that experimental branch, as shown in Figure 4.5. This way modifications remain local to *foo* and the experimental setup can still be reproduced by pulling the revision before the revert operation.

If the researcher eventually recognizes that some of the source code modifications done inside *foo* branch could be useful for the whole project, there are two ways to insert them in the *src* branch. The first one involves rewriting Git history and it is not advised as it can introduce incoherence between Git repositories. The second option is to simply cherry-pick the desired commits. Although this approach produces some redundancy, it is easier, safer and enables to keep the Git history comprehensible and consistent.

Making transversal analysis

Such Git organization, environment parameter capture and careful note taking in labbook also simplifies the comparison of data sets. Since the *data* branch aggregates all the *xp#* branches, it is the best location to analyze and compare these different results with each others. The numerous plain-text meta-data are easily exploited and incorporated in the analysis. Since each data file comprises the revision of the source code used to generate it, it is easy to backtrack to specific commits and to exploit the labbook to explain unexpected behaviors.

Writing reproducible articles

Using the described workflow on a daily basis makes the writing of reproducible articles straightforward. Figure 4.2 shows how researchers can create a new branch from the *data* branch that contains only useful experimental results by using the `git article start art1` command. This command deletes all the unnecessary data from the new branch, keeping only the experiments that are previously tagged in labbook with *art1* keyword.

Occasionally, some important experiment results may have been overlooked or conducting additional measurements becomes necessary. In such case, these new results can be added later through merging with the updated *data* branch.

Using *art1* branch is convenient for the collaborators writing an article that are not concerned with the analysis part. For them it is possible to pull only the Git history of the article, not the

whole project.

The problem may occur if one wants to present experiment results that are stored in large files. Even though *art1* branch is much smaller than the *data* branch, its size still may be non-negligible. Since certain contributors to the article may want to have the access only to its text and figures, doing long cloning and storing of the experiment data for them is unnecessary and burdensome. A possible solution (, also used for writing this thesis) is to create a new repository for the paper, while using tools such as git submodules to synchronize with the *art1* branch. Therefore, only researchers needing data to perform the analysis would checkout and pull any changes of the main project. For other authors this stays transparent as they are not obliged to load the submodule. They can work solely with the new "mini project" created for the specific paper that contains only the text file and the final figures.

Regardless of whether the paper is written in the *art1* branch or in a separate project, the same principles used for conducting experiments with Org-babel can be applied when writing an article. This approach allows combining the text of the paper with data transformations, statistical analysis and figure generation, all using generally different programming languages. A major advantage of this methodology is that a lot of code and text can be recycled from previous analysis scripts and from the labbook.

Keeping everything in the same file rather than to have it scattered in many different ones, makes everything simpler and greatly helps the writers. We did not encounter any particular issue when multiple authors simultaneously worked on the same paper. This also simplifies modifications and corrections often suggested by reviewers since every figure is easily regenerated by calling the code snippet embedded next to it in the article.

The final result of the whole workflow, is an article containing all the raw data that it depends on together with the code that transformed it into tables and figures, possibly along with the whole history with the detailed explanations of how this data was obtained. This is very convenient not only for the authors but also for the readers, especially reviewers, since all experimental and analysis results can be inspected, referenced, or even reused in any other research project.

4.2 Publishing results

Making data and code publicly available is a good practice as it allows external researchers to improve or build upon our work. However, at least in our domain it is not that commonly done, in particular because it is not that trivial to do when the study was not conducted with a clean methodology in mind from the beginning. If such intentions are not envisioned from the beginning of the project, it is generally very tedious to document and package afterwards. Gathering all the data required for an article can be cumbersome, as it is typically spread in different folders on different machines. Explaining experiment design and results is even harder, since notes that were taken months ago are often not precise enough. In the end, few researchers somehow manage to collect all the necessary elements and put them in a tarball, to accompany the article. Nevertheless, such data without appropriate comments is hardly understandable and exploitable by others. This lowers the researchers' motivation to share their data as it will not be widely used.

The question of what parts of this whole history should go public can remain a sensitive topic. We think that, at the very least, the data used to produce the article figures and conclusions should be made available. Of course, providing only already post-processed .csv tables with only carefully chosen measurements can make the article replicable, but will not guarantee anything about reproducibility of the scientific content of the paper. Therefore, meta-data of all experiments should be made available as well. Likewise, it is desirable to provide more material than what is presented, as it allows for illustrating issues that cannot be included in the document due to lack of space. The most extreme approach would be to publish everything, i.e., the whole laboratory notebook, acquired data and source code archived in a revision control system. Yet, some researchers may have perfectly valid reasons for not publishing so much information (copyright, company policy, implementation parts that the authors do not wish to disclose at the moment, etc.).

The methodology we propose allows for easily choosing which level of details is actually published. From the wide spectrum of possible solutions, we present two we used so far.

4.2.1 The partially opened approach with figshare hosting

When we first started writing the article on the modeling and simulation of dynamic task-based runtimes [5], our Git repository was private and we had not considered to open it. To publish our experimental data, we decided to use Figshare, which is a service that provides hosting for research outputs, that can be shared and cited by others through the DOI mechanism.

Although our article was managed within an internal Git, publishing to figshare required to select, archive and upload all the data files and to finally annotate them in the web browser. This could probably have been automated, but the REST API was not completely stable at that time, so we had to do everything manually. Likewise, the Fidgit project [Fid] could help, but it was at the early development stage and requires the whole Git repository to be hosted on GitHub, which may raise other technical issues (in particular the management of large files, whose size cannot exceed 100MB).

Hosting all our raw data on figshare also required adjusting our reproducible article. Data are first downloaded from figshare, then untared and post-processed. To this end, we again used the literate programming feature of Org-babel and the way we proceeded is illustrated in e926606bef{article#l85} or in Appendix B.6.

Finally, this resulted in a self-contained article and data archive [Z214]. This approach was not so difficult to use, although the interaction with figshare was mostly manual, hence not as effective as it could have been.

4.2.2 The completely open approach with public Git hosting

Using the previous approach, we somehow lost part of the history of our experimental process. Some data sets were not presented, some experiments where we had not properly configured machines or source codes were also missing. Nevertheless, it is clear that with highly technical tools and hardware such as the ones we experimented with, good results are not only the consequence of an excellent code, but also of expertise of the experimenters. Making failures available can be extremely instructive for those willing to build upon our work and thus publishing the whole labbook and Git history seemed important to us. In our case, this did not require additional work except to move our private Git repository to a public project [SSW]. With all the information we provide and an access to similar machines and configurations, others should be able to repeat our experiments and to reproduce our results without much burden.

In the end, it is important to understand that even though we decided to completely open our labbook to others, this step is not a prerequisite for writing reproducible articles. The level of details that is made public can be easily adapted to everyone's preferences.

4.3 Conclusion

In this chapter, we did not intend to propose new tools for reproducible research, but rather investigate whether a minimal combination of existing ones can prove useful. The approach we describe is a good example of using well-known, lightweight, open-source technologies to properly perform a very complex process like conducting computer science experimentation on prototype hardware and software. It provides reasonable reproducibility warranties without taking away too much flexibility from the users, offering good code modification isolation, which is important for *ad hoc* changes that are ineluctable in such environments. Since all the source code and data are in Git repository, reconstructing experimentation setup is greatly simplified. One could argue that not all elements are completely captured, since operating system and external libraries can only be reviewed but not reconstructed. To handle this, researchers could build custom virtual appliances and deploy them before running their experiments but this is not an option on all

experimental machines. Using virtual machines to run the experiments is not an option either, since in our research field we need to do precise time measurements on real machines and adding another software layer would greatly perturb performance observations.

We used this methodology for conducting our research on how to faithfully predict the performance of complex HPC applications through simulation. Indeed, as explained earlier, such study requires extensive experimental and simulation campaigns. The proposed workflow based on Git and Org-mode proved very efficient. However, the main difficulty was to find a right level of granularity and which notes, tests and errors should be recorded and which should be ignored. Capturing absolutely everything can be counterproductive, as important information will be hidden in the large amount of data and it would be hard to exploit it. Finding the right trade-off is specific to every domain and scientist, and it evolves through the time.

Our approach was also a base for a joint project between researchers from Grenoble (MESCAL and MOAIS teams) and Bordeaux (HeiPACS and STORM teams). We have created a common Git repository on GitHub for sharing traces and analysis of different application executions [Traa]. All files are commited to the repository except very large traces that are stored at zenodo while our Git contains only links to such data. This archive is envisioned to promote better collaboration not only between researchers of these two centers, but also between application developers and trace visualization experts. Although we haven't used the Git branching scheme presented in Figure 4.2, we applied the same principles regarding code and data organization. More importantly, we encouraged all participants to use Org-mode for environment capture and keeping laboratory notebook by documenting their experiment results. A very useful feature provided by GitHub (that will probably soon be available on other project hosting frameworks) is the pretty printing of Org-mode files. This enables a "forum-like" discussions about the experiment analysis through directly editing the Org-mode document in a web browser [Dis].

Chapter 5

Porting StarPU over SimGrid

In this chapter, we show how we crafted a coarse-grain hybrid simulation/emulation of StarPU [ATNW11], a dynamic runtime system for heterogeneous multi-core architectures, on top of SimGrid, a simulation toolkit specifically designed for distributed system simulation. We start by briefly presenting StarPU and the main principles behind porting StarPU over SimGrid, then we present in more details three different modeling aspects that we had to carefully take into account.

5.1 Choosing a runtime candidate for simulation

As discussed throughout the Chapter 2, task-based runtime systems recently became a very popular solution to bridge the gap between performance and portability of HPC applications. From a wide range of different runtimes, we choose to focus our study on StarPU mainly for the two following reasons.

First, StarPU is a dynamic task-based runtime system specifically designed to exploit heterogeneous multi-core architectures. It is a task programming library that handles low level concerns such as task dependencies, optimization of heterogeneous scheduling, and optimization of data transfers between main memory and discrete memories. In StarPU, tasks use abstractions named *codelets*. This task decomposition allows the developer to propose multiple codelets specifically optimized for each architecture (e.g., using CUDA or OpenCL for GPUs). StarPU can then select at runtime the best codelet to use to execute each task. Through this unified abstraction, StarPU ensures portability of the code. On the other hand, portability of performance is obtained with the help of schedulers that efficiently exploit the heterogeneity of the machines. StarPU exploits all resources to perform the computation, adequately choosing which task should be executed on which processing unit, but also minimizing the amount of data transfers. Finally, this runtime is structured in such a way that it provides the right level of granularity and abstraction as well as the ability to obtain any relevant machine or task information, which makes it a good candidate for simulation.

Another important reason for choosing this runtime was the motivation of the developers to work on such a project. They strongly believed that both their users and themselves could greatly benefits from having an accurate simulator. Their expertise was crucial for rapidly and correctly porting StarPU on top of SimGrid and for further improving of the models. Additionally, the StarPU development team puts a lot of efforts on having a stable code base, which drastically eases building other software layers on top of it. This is why there is a sustained effort in Bordeaux to put linear algebra applications such as Chameleon [Cha], `qr_mumps` [ABGL13] or FMM [ABC+14] on top of StarPU. However, there is nothing specific to StarPU in the approach we followed. The same kind of study could as well be done with other runtimes. In the following sections, we present the main challenges that need to be addressed when porting a simulation of any dynamic task-based runtime.

5.2 Porting StarPU over SimGrid

StarPU scheduling is dynamic and opportunistic, hence it is not deterministic. Replaying an execution trace, as it can be done for classical MPI applications (described in Subsection 3.2.3), is thus not an option. The decisions taken when simulating should be as close as possible to the ones taken in a native execution. Therefore, the most natural approach is to execute the StarPU code related to scheduling decisions and to replace actual task execution with SimGrid calls, as would be done in emulation. To be more precise, by emulation and simulation we consider:

> *Emulation: executing real applications in a synthetic environment, generally slowing down the whole code.*
>
> *Simulation: use a performance model to determine how much time a process should wait.*

To make sure that simulation is carried out in a reproducible and controlled manner, SimGrid exports a specific thread API (similar to the POSIX one) that allows the SimGrid kernel to control the scheduling of all application threads. In simulation, such threads run in mutual exclusion and are scheduled upon completion of simulated data transfers and simulated computations. Therefore, any direct regular call to the POSIX threads has to be abstracted as well (e.g., calling SimGrids' `xbt_mutex_acquire()` instead of standard `pthread_mutex_lock()`). Likewise, in simulation mode, any memory allocation on CPUs or GPUs has to be faked as no actual data processing is done and no actual GPU is necessarily available on simulation machines. They are thus replaced by a call to `MSG_process_sleep()` to only simulate their overhead. Last, since schedulers may use runtime statistics to take scheduling decisions, time has to be abstracted as well to make sure that simulation time (as returned by `MSG_get_clock()`) is used instead of system time (as returned by `gettimeofday()`). This fine-grain thread control is the key point that previous emulation attempts from the ICL of the University of Tennessee Knoxville were missing, resulting in ample inaccuracies due to improper synchronization between runtime threads and simulated time.

Therefore, when running on top of SimGrid, StarPU applications and runtime are *emulated* since the actual code is executed, but any operation related to thread synchronization, actual computations of CPU-intensive kernels, or data transfer is in fact *simulated*.

In other words, the control part of StarPU is executed to dynamically inject computation and communication tasks in the simulator. Additionally, for all synchronizations, transfer requests and memory allocation/deallocations, runtime is modified to inject delays that increase simulation time and account for the overhead of such operations.

In order to know how much time the tasks and delays will take, SimGrid relies on the *kernel* and *platform* calibrations performed by StarPU [ATN09]. The result of the platform calibration is an architectural description of the machine, specifying each CPU core or GPU with its processing power and the interconnect of the whole machine. We choose to simply represent each processing unit as a SimGrid host with specific characteristics. Each host comprises one or several threads that manage synchronization and signaling to StarPU, whenever transfer or computation kernel ends. Additionally, platform description contains measured latencies and bandwidths for communications between stated processing units and other machine resources. All these characteristics are benchmarked beforehand on a target platform by StarPU or by few simple scripts. To simulate a desired machine, one would need to run only once these short calibrations. As a result, such approach is very different from the classical ones described in Section 3.2 where architecture is modeled in detail and coarse-grain performances are derived from fine-grain simulation of the internals.

From a modeling perspective, we consider three main components to take into account: the StarPU scheduling and control part, the communications between resources and finally the computations on different computing units.

5.3 Modeling the StarPU runtime

StarPU has been extended in collaboration with the StarPU developers to have two modes of execution: *Native* and *SimGrid*. Assuming in the Native mode means to execute StarPU as usual, running the tasks on processing units, transferring the data between them, etc. When running in SimGrid mode, StarPU is executed on one machine, but only to simulate the behavior of another machine.

Some parts of StarPU are modified to use SimGrid function calls instead. Indeed, since the StarPU runtime is dynamic, it is essential to design a faithful emulation of the control part to produce scheduling decisions as close as possible to reality. Otherwise, this would damage simulation prediction accuracy of the whole application execution. We will describe in the following several important aspects of the runtime control part that have to be correctly managed.

5.3.1 Synchronization

To offer deterministic and reproducible executions, SimGrid provides its own thread abstractions. Therefore, we made explicit modifications to the runtime to call SimGrid functions instead of pthreads. We illustrate this kind of modifications with the conditional wait function shown in Figure 5.1, where standard `pthread_cond_wait()` (line 22) is simply changed for SimGrids' `xbt_cond_wait()` (line 10) in simulation mode. A similar principle is applied not only for synchronization, but throughout the StarPU source code for all necessary modifications of the runtimes' control part.

Although these changes were performed manually, it could have been done automatically for pthreads. However, there are other operations (e.g., test and set, atomics) whose implementation is not as simple as the example in the Figure 5.1. For such cases, writing the scripts to automatize the generation of SimGrid code would be more burdensome than just modifying the functions one by one.

5.3.2 Memory allocations

When executing StarPU in SimGrid mode, memory is not really allocated for data, as no actual computation or communication is going to be performed. Memory allocation and deallocation calls are thus changed to only inject delays into simulator. These timings depend on memory type (RAM or GPU memory) and on the size of the buffer that needs to be allocated.

However, SimGrid still keeps track of the information about the size of the allocated data, as memories on target machine have fixed capacities. This is very important for ensuring that simulation will never exceed the memory limits of the simulated platform. Overlooking this aspect can lead to completely misleading predictions.

Nonetheless, there are some runtime related structures used in controlling and guiding StarPU execution. For data structures it is still necessary to do real memory allocations, since StarPU is emulated and the data inside these structures is needed for correctly executing the runtime. Thus, it is impossible to automatically replace all *malloc* calls inside StarPU with SimGrid functions. The modifications have to be performed carefully by a StarPU expert who knows exactly which memory allocations belong to application data and which ones are part of the runtime core.

5.3.3 Submission of data transfers

The modeling of data transfers between RAM and GPU memories will be elaborated in Section 5.4. However, not only communications, but also the submission of GPU transfers take a certain amount of time. Neglecting these could change scheduling decisions and prediction of the total program duration. Therefore, some fixed delays need to be injected into simulation for every data transfer submission.

Changing the CUDA function calls for SimGrid code was once again done manually. Since StarPU has a good level of abstraction and provides wrapper functions for any GPU transfer, this

```
1  #ifdef STARPU_SIMGRID
2  int starpu_pthread_cond_wait(starpu_pthread_cond_t *cond,
3                               starpu_pthread_mutex_t *mutex)
4  {
5          _STARPU_TRACE_COND_WAIT_BEGIN();
6
7          if (!*cond)
8                  STARPU_PTHREAD_COND_INIT(cond, NULL);
9
10         xbt_cond_wait(*cond, *mutex);
11
12         _STARPU_TRACE_COND_WAIT_END();
13
14         return 0;
15 }
16 #else
17 int starpu_pthread_cond_wait(starpu_pthread_cond_t *cond,
18                              starpu_pthread_mutex_t *mutex)
19 {
20         _STARPU_TRACE_COND_WAIT_BEGIN();
21
22         int p_ret = pthread_cond_wait(cond, mutex);
23
24         _STARPU_TRACE_COND_WAIT_END();
25
26         return p_ret;
27 }
28 #endif
```

Figure 5.1: Implementing the simulation mode requires some code modifications of StarPU. This simple example illustrates the SimGrid (lines 2-14) and Native (lines 16-25) modes for executing StarPU's conditional wait.

Operation	Transfer queue management	GPU memory allocation (`cudaMalloc`)	GPU memory deallocation (`cudaFree`)	Pinned RAM allocation (`cudaHostAlloc`)
Time	10μs	175μs	125μs	650μs/MB

Table 5.1: Typical duration of runtime operations.

code modification had to be done in a single place. On the other hand, automatizing this process for another runtime with different internal organization would probably require implementing a fake CUDA API, which would be quite tedious.

5.3.4 Scheduling overhead

Since StarPU is emulated, the actual scheduling of tasks is performed as well. However, the time to compute a scheduling decision is not the same on the target simulated machine and on the machine used for simulation. In our current solution, we decided to leave out these discrepancies by completely ignoring the scheduling overhead in simulation mode, as its influence to the final simulation accuracy seems minimal.

It is important to note that scheduling time is precisely measured in native traces. Thus, it could be possible to take this into account when simulating target machine by injecting certain delays every time StarPU has to choose a resource for running a task. However, implementing this is not straightforward and for the studies we have performed so far, the cost of scheduling is negligible and often covered by other operations that are executed in parallel.

5.3.5 Duration of runtime operations

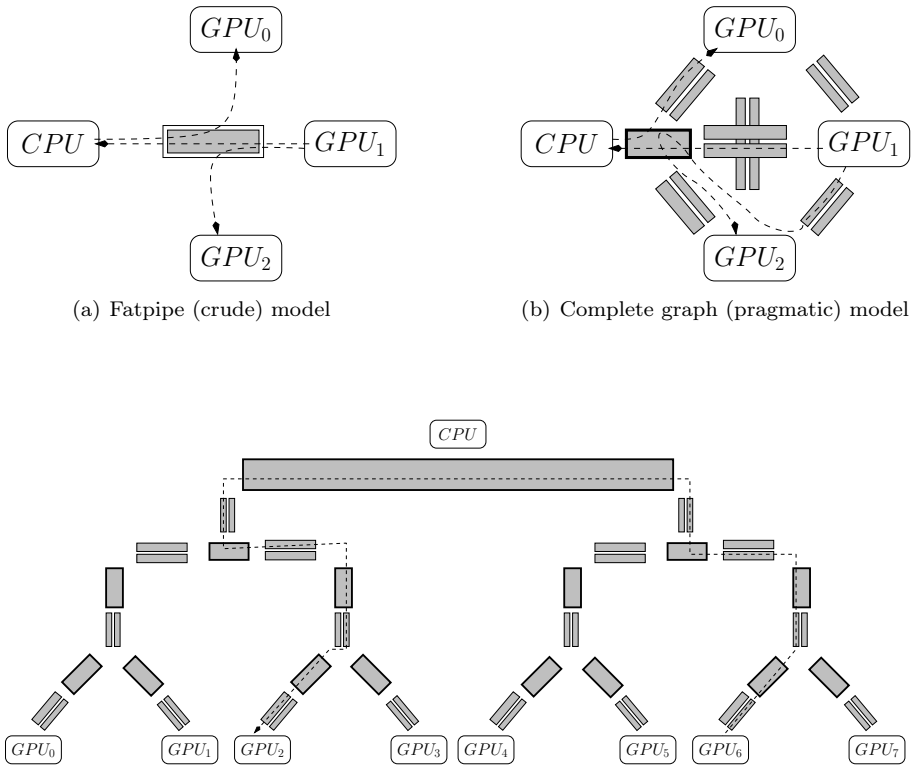
We described how process synchronizations, memory allocations and submission of data transfer requests are all faked in simulation mode, whereas such operations in native execution do take time and have some influence on the overall performance. Several delays were thus included in the simulation to account for their overhead and Table 5.1 depicts typical duration of such operations. It is interesting to note that pinned RAM allocation is linear with memory size, since it has to pin each physical page of the allocation, while other allocations have more standard, constant costs.

We measured the timings from Table 5.1 offline and realized that they are very similar for all GPUs and CUDA versions we tested. Thus, we pragmatically decided to always inject the same values for every simulation regardless of the target machine. This is improved further by benchmarking the values for each GPU during the machine calibration and later adding this information to the platform description. SimGrid would then use the benchmarking results and inject the corresponding delays during the simulation, possibly even taking into account the variability of the measured values. However, this would require some additional changes to the default StarPU calibration and only for minimal benefit. Thus, we do not plan to implement such extensions yet.

The other modifications to the StarPU code that had to be performed in order to get a viable simulation are more related to the modeling of the communication between processing units and to the modeling of the computation of kernels.

5.4 Modeling communication

In parallel applications, data is generally distributed across the memory of the computing units that execute the tasks. Since, different tasks require different parts of the whole data, this implies many exchanges of data blocks. When running a StarPU program, these data transfers are explicit. However, if the transfer is done between CPU cores that share the same memory, no actual data will be moved, only the pointers will be updated. This results in a problem regarding NUMA architectures, where the time to retrieve data from the local and distant cache is not equal. In order to correctly model this phenomena, one would have to precisely trace all the memory accesses.



(a) Fatpipe (crude) model

(b) Complete graph (pragmatic) model

(c) Treelike (elaborated) model

Figure 5.2: Communication and topology modeling alternatives. In the crude modeling, a single link is used and communications do not interfere with each others. The pragmatic complete graph modeling allows for accounting for both the heterogeneity of communications and the global bandwidth limitation imposed by the PCI bus. Complex machine architecture demand for even more elaborated models.

This process is hard to implement and very costly in terms of performance. Therefore, for now we concentrated solely on transfers involving GPUs as those are explicit and the data is *de facto* moved via the PCI bus. In the rest of this thesis, when discussing communication, we will refer only to CPU-GPU (RAM-GPU) and direct GPU-GPU transfers.

Due to the relatively low bandwidth of the PCI bus, applications running on hybrid platforms often spend a significant fraction of the total time transferring data back and forth between the main RAM and the GPUs. Modeling communication between computing resources is thus of primary importance. SimGrid models heterogeneity and contention in a simple way with a linear model, fair with no performance degradation as if flow control was perfect (see Subsection 3.2.4 for more details). SimGrid also has a very flexible platform description that allows for running simulations of various computer architectures, without requiring any modifications to the simulator. In such a context, there are many ways to represent PCI bus.

5.4.1 Different PCI bus models

Since links through which the transfers are performed are just an abstraction, there is more than one way to describe the network architecture. We have started with the simplest models that were enough for initial use cases. Later, when the complexity of the experimental machine architectures and the size of the application problem both increased, we had to upgrade our network models. Figure 5.2 shows three different types of PCI bus models we have developed so far.

Fatpipe model

The macro data-flow model [CJLL95, PY90] which is used in most theoretical DAG scheduling studies assumes that communication delays are paid each time a task and one of its successors are not assigned to the same processing units. This implicitly assumes a homogeneous and contention free network.

Therefore, as a first approximation (see Figure 5.2(a)), we mimicked this model by introducing a single non-shared link whose latency and transfer rate correspond to typical characteristics of the PCI bus. The non-shared characteristic of the link means that any connection gets the full capacity of the link regardless of the number of other competing flows. However, such modeling does not account for important architectural aspects. First, the bandwidth between CPU and GPU is asymmetrical. Second, communication characteristics are not uniform among all pairs of CPUs and GPUs, as it depends on the chipset architecture. Therefore, the fatpipe model is effective only for very simplistic machines (the hypothetical ones) for which researchers want to evaluate the characteristics of their application, scheduler or hardware.

Complete graph model

We decided to account for the shortcomings of the previous approach by using a dedicated uplink and a downlink with different characteristics for each pair of resources (see Figure 5.2(b)). Furthermore, any communication between two resources has to go through a common shared link (in bold), which represents the maximum capacity of the PCI bus. This creates a complete graph with communication bandwidths and latencies measured for every possible transfer between computation units, together with one link shared by everyone.

Even though the actual communication on the real machine is performed differently as there is no direct connection between memories, this pragmatic approach provides very good results. However, modeling network in such a way is limited to the machines on which all resources use the same PCI bus. For bigger platforms with more complex architectures, more sophisticated models are needed.

Treelike model

In HPC community, researchers often work with prototype hardware and non-standard architectures. The hwloc [BCOM+10] output in Figure 5.4 depicts the Idgraf machine that has two CPUs with 6 cores each, 8 GPUs and a very particular interconnect. Such architectures demand for more elaborated models. The two previously presented solutions both use a single shared link for modeling PCI bus contention. However, this approach cannot be applied for complex machines such as the one on Figure 5.4, since communication between two pairs of resources can go through completely different routes. This results in sharing bandwidths only on particular links while on others there is no interference at all.

Figure 5.2(c) shows a treelike model we created for Idgraf. This model imitates the machine architecture, where routes between different memories consist of many connected unidirectional and bidirectional links. This way we ensure that transfers will be slowed down only on certain parts of their route, depending on the current traffic on that section of the network. Such a complex network topology is generated automatically during the phase of the machine calibration. In Figure 5.3 we display only a small piece of platform description file, the one defining the route between GPU6 and GPU2 (also presented as a dotted line on Figure 5.2(c)).

Although this last approach seems intuitive as it follows the architecture of the machine, benchmarking the latency and the bandwidth of every link stays a challenging task.

5.4.2 Model based on calibration

Whichever network model is chosen from the previous subsection, it has to be instantiated with the benchmarked values from the target machine. For that we use machine calibration functions that were already part of the StarPU code. Indeed, even before running in native mode, StarPU

```
<route src='CUDA6' dst='CUDA2' symmetrical='NO'>
   <link_ctn id='CUDA6-CUDA2'/>
   <link_ctn id='PCI:0000:0d:00.0 down'/>
   <!--   Switch PCI:0000:[0d-0d] through -->
   <link_ctn id='PCI:0000:[0d-0d] down'/>
   <!--   Switch PCI:0000:[0b-0d] through -->
   <link_ctn id='PCI:0000:[0b-0d] down'/>
   <link_ctn id='PCI:0000:[0a-0d] through'/>
   <link_ctn id='PCI:0000:[0a-0d] down'/>
   <link_ctn id='PCI:0000:[00-13] through'/>
   <link_ctn id='PCI:0000:[00-13] down'/>
   <link_ctn id='Host'/>
   <link_ctn id='Host'/>
   <link_ctn id='PCI:0000:[80-8a] up'/>
   <link_ctn id='PCI:0000:[80-8a] through'/>
   <link_ctn id='PCI:0000:[81-84] up'/>
   <link_ctn id='PCI:0000:[81-84] through'/>
   <link_ctn id='PCI:0000:[82-84] up'/>
   <!--   Switch PCI:0000:[82-84] through -->
   <link_ctn id='PCI:0000:[84-84] up'/>
   <!--   Switch PCI:0000:[84-84] through -->
   <link_ctn id='PCI:0000:84:00.0 up'/>
  </route>
```

Figure 5.3: Excerpt of the Idgraf platform description file generated using the treelike model. This fragment defines the route used when communicating between GPU6 and GPU2 using CUDA.

benchmarks the PCI bus and all possible transfers in order to optimize its scheduling. We can thus reuse this gathered information when running simulations as well. This step is necessary as benchmarked values are typically much more precise than the data provided by hardware vendors.

Another important aspect that needs to be managed for parallel transfers is the contention. Two transfers occurring at the same time and sharing a link will certainly interfere with each other, thus degrading the performance for both. However, depending on the resources involved in the communication, data transfers may also be serialized or not. For example, although most CUDA transfers are serialized whenever they involve the same resource, on some systems it is possible to transfer both from GPU_0 to GPU_1 and from GPU_1 to GPU_0 at the same time. Thus, these serialization rules have to be additionally taken into account inside the network description.

Apart from the serialization, the performance of transfers can also heavily depend on both the GPU type and the CUDA version. All this requires a sound model calibration protocol and very careful capturing of the environment setup when doing the experiments. In such an experimental context, our methodological approach (described in Chapter 4) proved to be very helpful.

Finally, once data is transferred to the memory of the specified processing unit, it can be used for computing the kernel and this is the last step that needs to be addressed by the simulation.

5.5 Modeling computation

When running simulation, the actual result of the application is of no interest. Hence the execution of each computational part is replaced by a virtual delay accounting for its duration.

The basic model in SimGrid to manage computation is based on FLOPS. Each processing unit in the system is characterized by its speed (FLOPS rate), and during the simulation the user provides how much floating-point operations a certain kernel needs to execute. From these two values, SimGrid simply computes the duration (possibly taking into account contention on

Figure 5.4: Architecture of the Idgraf machine with 12 cores, distributed on 2 NUMA nodes, 8 GPUs and their interconnect.

computational resources if used by several threads at the same time) and increases the simulation time. Such approach is effective when simulating distributed, peer-to-peer or grid applications, where a very rough estimation of the computation duration is sufficient. However, for HPC applications, much more accurate predictions are needed, as computational blocks (kernels) are smaller and occur more often. The kernel performances are measured on a real machine, and then models accounting for both their mean duration and their variability are created.

The easiest way to obtain the duration of the kernels is from a native execution trace. The captured timings are then just injected into simulator, using the replay technique so commonly applied for MPI applications. However, as discussed in Subsection 3.2.3, this approach has many shortcomings, from which the main one is that it requires executing StarPU on the target machine every time there is a code modification. It also strongly bias the simulation toward one particular execution. Working in such a way is not only cumbersome, but it questions the very purpose of using the simulation in the first place.

Instead, we propose a different approach, aiming at finding good, possibly parameterized, models for this simulation purpose. Since the code of the computational kernels is rather stable, these results are valid even if there are changes to the rest of the application or runtime. However, to instantiate models one still needs to access the target machine and run calibration of the kernels. Such calibration has to be performed only once and after that, numerous simulations can be executed. Nonetheless, different types of computation kernels require different types of models.

Modeling parameter dependent kernels

HPC applications, particularly linear algebra applications, usually work with large matrices that are divided into smaller blocks to maximize parallelism. Then, certain operations encapsulated into runtime tasks are performed on the blocks. These operations can be implemented to rely on standard libraries such as the BLAS, the LAPACK or the MKL. Whenever there is a basic matrix operation on a single block, applications use standard functions provided by these libraries (`GEMM`, `POTRF`, `GEMQRT`, etc.). The rationale is that the performance of such routines can be optimized for each specific machine architecture. Nevertheless, depending on the operation and the structure of the matrix, these kernels can be very diverse.

When the blocks are densely filled, the time to execute a certain operation on a block of a matrix will mostly depend on its geometry. For example, when doing a general matrix multiplication on a square matrix, `GEMM` subroutine can be characterized by a single parameter, the block dimension. Typically, in the StarPU execution of dense linear algebra applications, such as Cholesky or LU factorization, the block dimension is an input parameter defined by the user and it will not change during the execution. Therefore, when simulating these applications, one only needs to calibrate the kernels for a single parameter value. In practice, there are few optimal dimensions, depending on whether the factorization is performed using CPUs, GPUs or both, and only kernels with such block dimensions need to be calibrated. Such coarse-grain approach does not allow for predicting the influence of a longer cache line or of a different block size. However, it allows for performing numerous simulations varying the whole matrix size or the scheduler, since the basic operation on a block will always be the same.

On the other hand, calibrating kernels that work on sparse matrices is much harder. Since they are filled with many zeros, the time to execute a kernel will depend not only on the size of the block, but also on its structure and on the number of non-zero values inside it. Benchmarking such kernels with multiple parameters is thus much more complicated and time consuming, as calibration needs to cover a huge experimental space. Additionally, in order to get accurate models that faithfully reflect kernel executions, the calibration has to be done non-uniformly following certain properties of the sparse linear algebra applications. Still, it is generally plausible to derive linear or polynomial models of such kernels, as we show in the Section 7.4.

Modeling kernels with complex codes

Not all runtime tasks perform regular operations on a problem of a fixed size. Some kernels are responsible for memory management of the application, which may be very complex and depend on the state of the other kernels executed in parallel. There are also tasks that assemble results from several other kernels, and for which the amount of data that will have to be processed is not known in advance.

The input parameters of these type of kernels are not representative of the amount of work they perform. Therefore, the duration of such tasks is impossible to predict based solely on their inputs, and supplementary parameters are needed. These additional parameters describe certain aspects of current the system, and can be related to the machine characteristics, the state of the applications, etc. Such parameters are very specific to each task, and in order to identify them, a good knowledge of the kernel code and the application behavior is required. Therefore, such kernels often need alternative approaches for calibration and modeling.

Limitations due to the simplistic machine models

Our modeling approach is coarse-grain and is thus based on simplistic models of the machines. We decided to neglect certain hardware phenomena. For example, memory hierarchy of the machine and how the data is distributed have a big influence on the time to execute a kernel. Moreover, we performed kernel calibrations on a single CPU in a dedicated surrounding, which ignores any hardware contention that can greatly decrease the performance. A detailed list of the limitations of our approach and the envisioned future improvements are presented in Subsection 8.2.1. Still, the chosen level of abstraction and the kernel models that are presented in this section proved to be sufficiently accurate for studying many applications.

5.6 Conclusion

In the previous chapter, we presented the main principles behind the SimGrid simulation of the StarPU runtime. We gave several implementation details and discussed central elements for simulating dynamic task-based runtime systems for hybrid machines. Coding the main part of this solution took a few days and represent no more than a few hundreds lines of code in total code base few hundreds thousands of lines. This simple approach immediately provided good results for basic use cases. However, validating, correcting and enlarging this approach has been a constant process that lasted for almost three years.

The same ideas could be applied to other runtime systems and simulators. However, the amount of work would mostly depend on the paradigms and the internal structure of such software. We believe that the crucial advantage of our solution was a coarse-grain approach as well choosing good runtime and simulator candidates. Both StarPU and SimGrid are modular with a good level of abstraction, which eased the development of this new tool. Our solution enables to obtain very accurate predictions for different machines and applications, and we present the evaluation results in the next two chapters.

Chapter 6

Performance Prediction of Dense Linear Algebra Applications

Designing and implementing simulation of the StarPU runtime was performed in parallel with numerous experiments that guided us towards all the features that had to be improved, always aiming at having more accurate simulation predictions. In this chapter, we present the most relevant experimental results obtained during the study.

We start by presenting the applications and the architecture of the machines used to evaluate our approach. Then, we provide details about the experiment design and the typical workflow, although the main principles behind the methodology are already described in Chapter 4.

Even though during a single application run, dense linear algebra kernels are executed on the same problem size, their duration has certain variability. We address this issue and propose an approach to account for it inside the simulation using histograms of duration distributions.

As previously presented, a careless modeling of any aspect of the runtime, of the communications or of the computations can lead to gross inaccuracies for particular combinations of machines and applications. We show in this chapter that we managed to cover the most important issues, which enables us to obtain excellent prediction of performance. We present results of the simulation of heterogeneous machines with a wide range of different GPUs. Additionally, we show how our method could be equally applied to the executions that use both CPUs and GPUs for doing computation. We equally exhibit current limits of our work regarding large NUMA machines, for which we still do not have sound models. Finally, we demonstrate typical use cases for such a tool, and from which StarPU users already benefit.

6.1 Experimental settings

To make sure our approach is reliable and applicable on various use cases, we had to validate it on a wide range of experimental settings. The most important elements of our experiments are the applications and the simulated machines, and thus they will be presented in more details.

Additionally, it is worth mentioning that all the systems used in the experiments had a Unix operating system, however with very different distributions. Also we used different libraries for the computation kernels (BLAS, LAPACK, MKL, etc.) depending on their availability on the target machine. The same applies to different *gcc* compilers, used for compiling both StarPU and SimGrid. However, all these software diversities we kept track of do not have a decisive influence on the conclusions that are presented in the following sections.

6.1.1 Applications

We used two well-known and efficiently implemented applications on top of StarPU: *Cholesky* and *LU* factorization. The Cholesky decomposition is typically used for finding numerical solutions for partial differential equations and thus it is a good representative for many scientific

```
for k = 1..nblocks do
    A[k][k] <- POTRF(A[k][k])
    for i = k+1..nblocks do
        A[i][k] <- TRSM(A[k][k],A[i][k])
    for i = k+1..nblocks do
        for j = k+1..i do
            A[i][j] <- GEMM(A[i][k],A[j][k],A[i][j])
```

Figure 6.1: Tiled Cholesky factorization.

applications. Figure 6.1 shows a pseudocode of tiled Cholesky implementation that was used during our research. Details of the algorithm, its parallelization and possible optimizations have already been extensively explained in many publications [ABED+15, Gus03, BLKD09] and we only briefly present here the main function of the kernels:

1. `POTRF`: The kernel performs the Cholesky factorization of a diagonal (lower triangular) tile of the input matrix and overrides it with the final elements of the output matrix.

2. `TRSM`: The operation applies an update to an off-diagonal tile of the input matrix, resulting from factorization of the diagonal tile done by `POTRF` and overrides it with the final elements of the output matrix. The operation is a triangular solve.

3. `GEMM`: The operation applies updates to an off-diagonal tile of the input matrix, resulting from factorization of the tiles to the left of it. The operation is a matrix multiplication.

Note that this is a simple StarPU implementation of the Cholesky algorithm, in particular it does not use the SYRK subroutine.

The second application we used in our experiments is the LU decomposition based on Gaussian elimination, another common algorithm, introduced by Alan Turing in his 1948 paper [Tur48]. In terms of StarPU implementation, this application has a structure similar to Cholesky, only this time relying on the `SCAL`, `GER`, `TRSM` and `GEMM` functions.

Concerning task granularity, for the executions running on GPUs we fixed a relatively large block size (960×960) as it is representative of what is typically used to achieve good performances [AAD+10, AAD+11]. In most of the experiments, the CPUs were only controlling the execution and scheduling of tasks while GPUs had the roles of workers, meaning that the whole computation was done entirely on the GPUs. We initially focused on this kind of scenario as GPUs have a performance that is easier to predict and provide a significant fraction of computational power for dense linear algebra kernels. However, in Subsection 6.7 we also report experiments conducted on a NUMA machine with a large number of cores (with no GPUs) and for which we thus used a smaller block size (320×320) that is much better suited to CPU resources. Finally, we also investigated situations involving both CPUs and GPUs doing parallel computations.

Running such applications on top of runtime typically raises questions about the optimal block size, the maximal GFLOPS rate, the influence of the scheduler, the transfers overhead, the memory contention, etc. Moreover, researchers are often interested on how their solution would behave on various architectures and platforms.

6.1.2 Machines

For dense linear algebra applications, the computational power of GPUs is generally much higher than the one of CPUs. Figure 6.2 proves such statement by providing the GFLOPS rate for the Cholesky application with StarPU when varying the size of the matrix from 1MB to several GiB. All three experimental campaigns were performed on the same machine but used different computation resource sets: 8 cores on the left plot, 3 GPUs on the middle plot, and both CPUs and GPUs on the one on the right. As expected, the hybrid solution provides the best performance

Name	Processor	Number of Cores	Frequency	Memory	GPUs
Hannibal	Intel Xeon X5550	2×4	2.67GHz	2×24GiB	3×QuadroFX5800
Attila	Intel Xeon X5650	2×6	2.67GHz	2×24GiB	3×TeslaC2050
Mirage	Intel Xeon X5650	2×6	2.67GHz	2×18GiB	3×TeslaM2070
Conan	Intel Xeon E5-2650	2×8	2.0GHz	2×32GiB	3×TeslaM2075
Frogkepler	Intel Xeon E5-2670	2×8	2.6GHz	2×16GiB	2×K20
Pilipili2	Intel Xeon E5-2630	2×6	2.6GHz	2×32GiB	2×K40
Idgraf	Intel Xeon X5650	2×6	2.67GHz	2×36GiB	8×TeslaC2050
Idchire	Intel Xeon E5-4640	24×8	2.4GHz	24×31GiB	/

Table 6.1: Machines used for the dense linear algebra experiments.

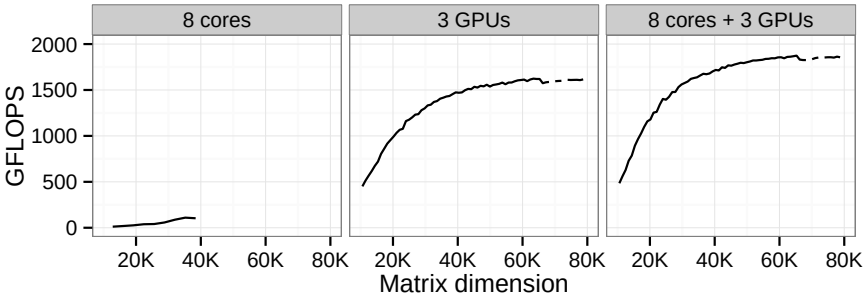


Figure 6.2: For dense linear algebra applications, most of the processing power is provided by GPUs. These plots depict the performance of the Cholesky application on the Mirage machine (see Table 6.1). A clearer view of these performance when restricting to CPU resources (8 cores) is provided in Figure 6.17 (4+4 cores).

and manages to take advantage of both resource types. Nevertheless, since GPUs provide the major fraction of the processing power, we mostly concentrate on executions that use only GPUs for computing kernels, even though we are able to accurately predict fully hybrid setups as well.

Therefore, we choose to use the systems described in Table 6.1 to study the validity our models. All seven machines with GPUs have distinct characteristics and span three different GPU generations, which intends to demonstrate the validity of our approach on a range of diverse machines. Additionally, we have experimented on a large NUMA machine without GPUs, but with 24 nodes each having 8 cores.

For doing the simulations we used either the same target machines or more frequently a commodity laptop. It is an Intel Core i7-3720QM with four 2.6 GHz CPU cores, 8 GiB RAM memory and no installation of CUDA libraries. Still, we remind that the characteristics of the machine used for doing simulation have no influence as the results would be the same on any other machine.

6.2 Modeling kernel variability

Varying distance of the data and contention on different resources (see Section 5.5), introduces a variability of the execution time of the kernels. Moreover, even without these effects, operating system and the non-determinism of today's machines adds some noise to the execution time. Consequently, a kernel execution with fixed parameters will always take slightly different time.

When running simulation, such variability can be modeled in different ways. In the study performed at the University of Tennessee Knoxville [HKY+14], researchers reported that the kernel durations of their executions can be well approximated with a simple normal distribution(see Figure 6.3). This may be valid for the kernels executed on a single multi-core CPUs, as in such

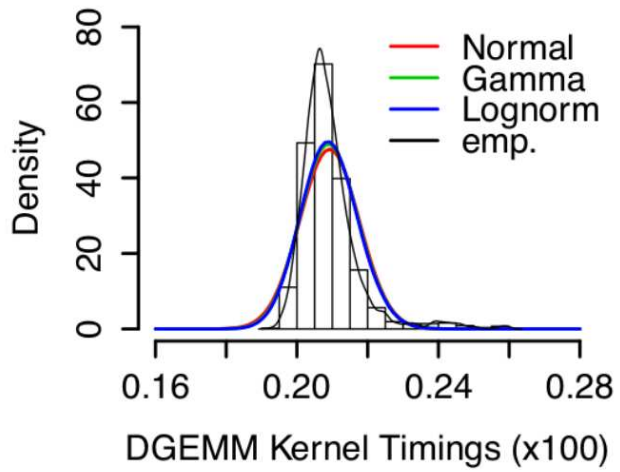


Fig. 4: A plot of the kernel execution times of the DGEMM (part of the Cholesky factorization) kernel along with the fitted distribution curves. The kernel execution times are multiplied by 100 for readability. The simple distributions do not fit quite as well as the DTSMQR kernels, but they seem to model the kernel execution times better than a constant or uniform distribution.

Figure 6.3: Analysis of the kernel duration distribution as done by Haugen et al. in [HKY+14]. A normal law approximates the sample distribution very accurately. However we believe this is valid only for simple multi-core CPUs.

cases the variability is mostly the product of different cache states. However, when working with GPUs, NUMA or hybrid machines, the placement of data and the time of its retrieval is much more complex, which results in much more irregular kernel durations.

In our initial approach, we used the mean duration of each computation kernel. This approximation was producing satisfactory results, as we were experimenting with dense linear algebra kernels that have optimal block size of the matrix, which leads to good utilization of resources and almost no idle time on computational units during the application run. In such context, modeling kernels with a single constant is accurate enough for the intended purposes and the mean value proved to be a good representative in our experiments. However, using a fixed value leads to a deterministic schedule in simulation, which may bias the simulation and does not allow to verify the ability of the scheduling algorithms to handle resource variability. Therefore, we performed a deeper study on kernel durations, its deviation throughout the application execution and finally the best way to capture and replay all these phenomena in the simulation.

6.2.1 Analyzing kernel duration distributions

In order to study kernel execution time more closely, we modified StarPU to capture the timing of every computation during a native execution. Figure 6.4 shows the evolution of 23254 `GEMM` kernels durations as a time sequence during $72,000 \times 72,000$ Cholesky factorization on a multiple GPU Attila machine (see Table 6.1). All observations were measured on single GPU and the kernel was always executed on 960×960 size blocks.

The timings can be clearly divided into two groups. The first one consists of the smaller durations, around 2.84 microseconds, that represent the vast majority of the measured values. This is a minimal duration for the execution of `GEMM` kernel on 960×960 blocks with TeslaC2050 GPU. In the second group are the higher value timings that can be much bigger than 2.84 microseconds and for which one can observe a huge variability. These durations correspond to the cases where the kernel was slowed down by a certain external factor. One reason could be that a matrix block

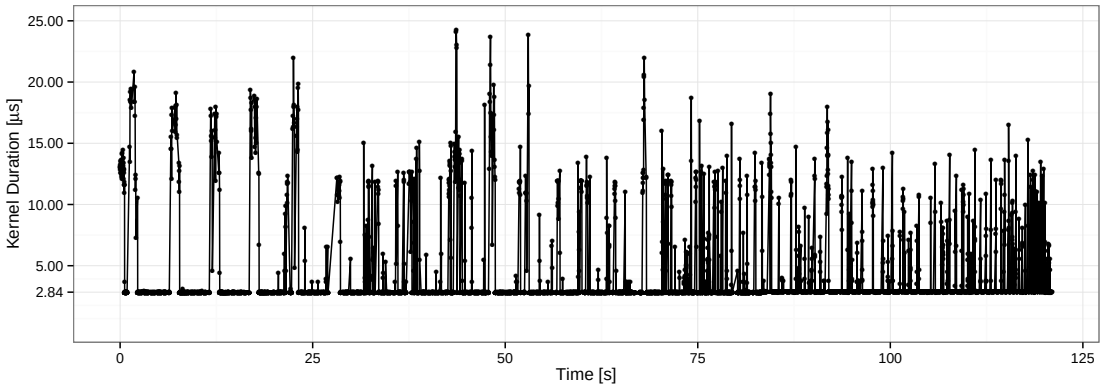


Figure 6.4: GEMM kernel durations on a GPU in a single $72,000 \times 72,000$ Cholesky factorization, presented as a time sequence. Most of the values are around 2.84 microseconds, but there is a significant number of higher durations as well.

was not fully available at the beginning of the execution. Another one is that the computation on GPU was done in parallel with a communication and since they both use the same GPU memory, this introduced performance penalties for both operations. There are even more possible sources of irregular performance, including different synchronization and operating system issues. However, the information about all these parameters are hard to capture and relate to the presented trace. Thus, in the rest of this analysis we assume that the variability in Figure 6.4 comes from independent, uncontrollable factors.

Such hypothesis implies that the durations of kernel executions are also independent. However, this is not true as the larger duration values are often clustered together, as can be observed in Figure 6.4. Due to the structure of the Cholesky application, this phenomenon is clearly noticeable in the beginning of the execution and less towards the end, but for the other applications it could be the opposite. In fact, these timings depend on the state of the system, the location of the data, other kernels executed in parallel, the operating system, etc. Moreover, consecutive executions of the kernel are likely to have an influence on each other.

Nonetheless, since parameters responsible for such performance degradation are hard to quantify, we decided to ignore the dependencies between kernel durations on the Figure 6.4. Such approximation seem plausible for this research, since when studying dynamic runtime schedulers it is more important to ensure that there is a right number of longer duration tasks, than specifying that these sometimes come in groups.

Figure 6.5 depicts the distribution of the GEMM kernels for the same trace. One can observe on the top plot that there is a huge spike for the lower durations that causes the other bars to be almost invisible, although we know from Figure 6.4 that there are many observations bigger than 3 microseconds. To address this issue, in the other plots of Figure 6.5, we zoom on the two groups separately in order to analyze their distributions in more details.

The bottom left one shows the distribution of the lower value timings that occur in 96% of the cases and take 87% of the overall duration of GEMM kernels. Even though the measured values are very stable, with only dozens of nanoseconds of difference, surprisingly one can observe that the distribution is bimodal. We believe that the reason behind it is the imperfection of the tracing tools. Indeed, since the tracing of the begin and the end time of the kernel is performed by the CPU that is in charge of controlling the GPU execution, timestamps may be slightly biased. Having a more precise measurements is possible, but would require to change the core of StarPU profiling, which is not trivial to implement.

The bottom right plot of Figure 6.5 shows how the distribution of bigger duration values is much more irregular even if the density plot tends to alleviate such effect. These timings occurred in only 4% of the kernel executions during the application run. We point out that the y-axis on

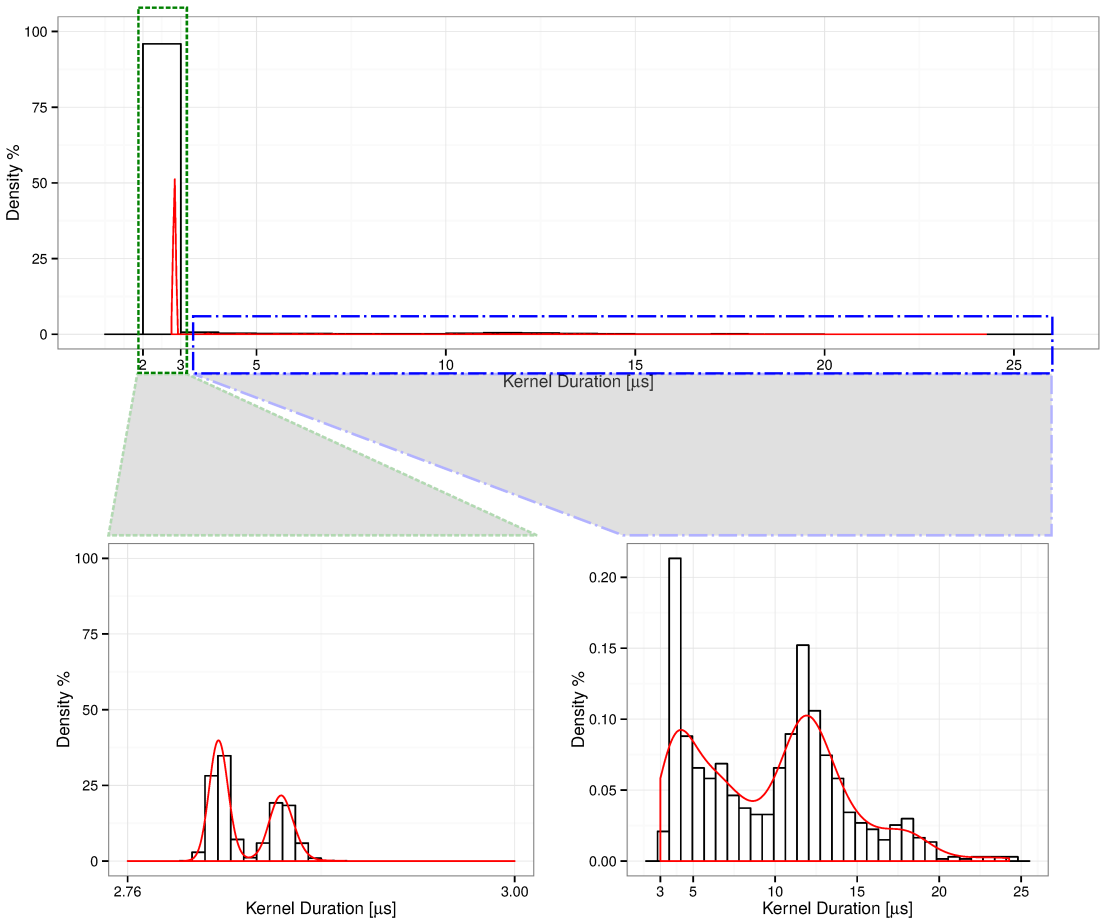
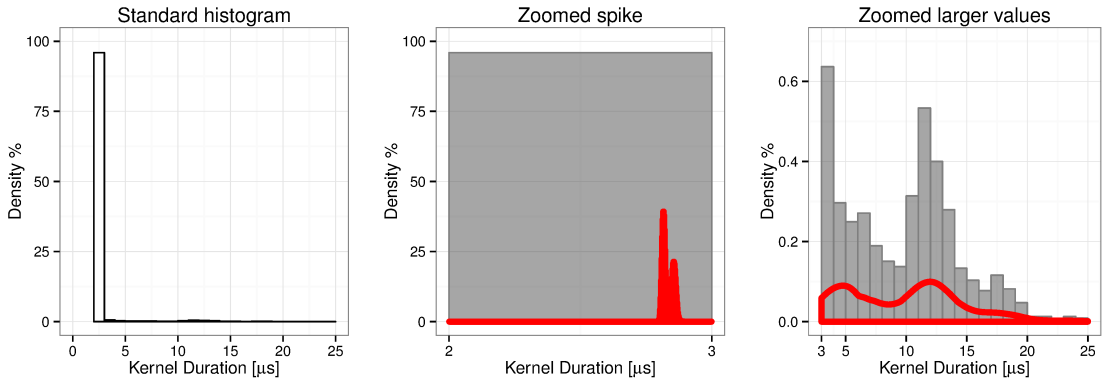
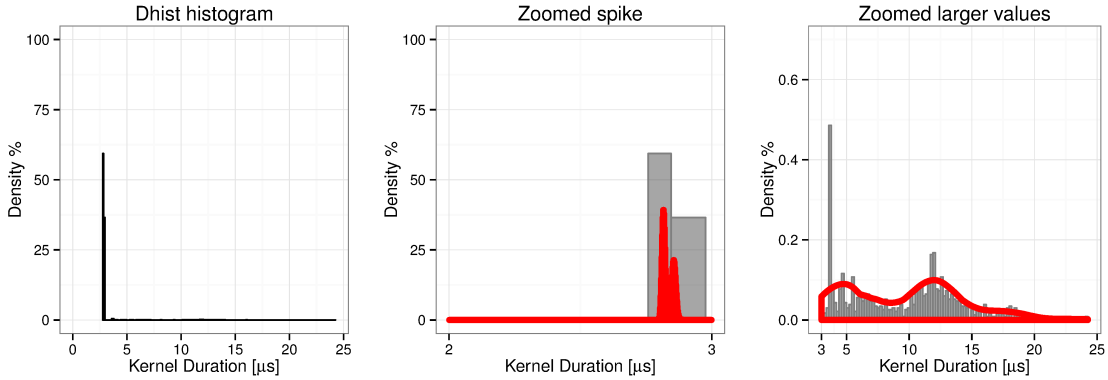


Figure 6.5: Distribution of `GEMM` kernel durations on a GPU in a single $72,000 \times 72,000$ Cholesky factorization. Top plot presents the distribution constructed for all observations, while bottom ones are reconstucted for two separate groups of observation depending on their duration value.



(a) With standard histograms, the bins have uniform width, thus the distribution is poorly captured especially the spike with the lower duration values.



(b) With irregular histograms, the bins are chosen more precisely, with non-uniform widths, thus the histogram approximation is more accurate.

Figure 6.6: Approximating `GEMM` duration with two types of histograms.

left and right plots of Figure 6.5 is not the same, as the ranges are from 0 to 100% and from 0 to 0.2%, respectively. Even though the higher duration values are rare compared to the first group, their cumulative duration is 13% which is not negligible.

The kernel performance we presented is specific to the GPUs. However, when experimenting with NUMA machines we also observed multimodal distributions. Accurately mimicking the behavior of kernel execution inside the simulation requires to carefully capture all the details of every execution mode. The easiest way is to replay the trace in simulation, but this approach has many drawbacks (as described in more details in Section 3.2). Another path is to create models based on distribution of the measured kernel durations. However, information about the parameters responsible for such variability is hardly accessible. One possible solution is to try to approximate duration distributions using histograms.

6.2.2 Using histograms to approximate distributions

The results presented in Figure 6.5 are captured for a `GEMM` kernel during a single Cholesky execution. However, even if the matrix dimension changes (and block dimension stays the same), the kernel duration tends to exhibit similar performance behavior, since it is executed for exactly the same data size. Therefore, it is enough to run a single Cholesky execution or even a well designed calibration by the benchmarking script and to capture durations for every kernel. Such collection

of data can then be used to analyze the computation duration distributions and to model it using histograms. These models can then be used during the simulation to generate pseudo-random variables from the histograms and inject them into simulator.

The main idea is to produce histograms and then save its parameters (bins and densities) into a separate text file. Later when running simulation and a specific kernel has to be executed (e.g., `GEMM` kernel on a GPU), SimGrid will consult the file with histogram characteristics in order to compute how much time the kernel execution will take. First it will pseudo-randomly choose a bin according to its probability (e.g., there are 0.4% chance that the kernel duration is between 12 and 13 microseconds). Then it will uniformly pick a value from the chosen bin and return it to the simulator as a predicted duration of the kernel.

This allows simulation to mimic the behavior of the real machine, where kernel durations may greatly vary. Such advanced approach is more compatible with simulations of dynamic runtimes than simple constant injection. However, it also has two major drawbacks.

The first shortcoming of this solution is that it assumes that the kernel durations used to construct histograms are independent. Clustered higher value timings from the beginning of trace shown in Figure 6.4 prove the opposite, but as already discussed this approximation is sufficient for the studied use cases. The problem occurring in the simulation is that bins are chosen randomly only according to their probability. In such a way, the character of the external parameters is completely neglected, even though these factors actually dictate the possible kernel slowdown and thus which bin should be selected. However, implementing this correlation would require knowing many aspects of the system and in the context of the modern machines it is hardly possible. Fortunately, in practice the runtime compensates quickly any longer executions of kernels by using dynamic scheduling. Therefore, this drawback has a very small effect on the final results, and we can continue using histograms without harming the simulation accuracy.

The second, much bigger problem, comes from the way standard histograms are computed. These histograms use uniform bin-width in order to produce figures that visually provide the most information to the researchers doing the analysis. However, this type of histograms proved to be a very bad solution for the distributions containing high, narrow spikes. More generally, such standard histogram representations are not optimal for abstracting the kernel duration distribution in models that are later used for simulation, as it is illustrated in Figure 6.6(a). This figure shows the distribution of the durations of `GEMM` kernel coming from the same trace as the one on Figure 6.5. The output of the standard histogram computed by R, a programming language for statistical computing, is displayed in the left plot of Figure 6.6(a). One can observe the two previously described groups of values. The first one is presented as a huge spike between 2 and 3 microseconds occurring in 96% of the cases, and its zoomed version is shown in the middle plot. The second group is with larger but infrequent values, thus they are hardly noticeable on the histogram in the left plot due to the big difference in probability to the spike. Therefore, we zoom on these kernel durations (changing y-axis range) on right plot of the same figure. Since the histogram uses a uniform bin-width (in this case of 1 microsecond) and there is a huge disproportion of the number of occurrences in the two groups of values, the kernel characteristics are not well captured and approximating this region by a uniform distribution is a very poor approximation.

One can clearly detect the bias standard histograms introduce by comparing the bottom left plot of Figure 6.5 and the middle plot of Figure 6.6(a). The first one shows histogram distribution generated only from the observations smaller than 3 microseconds, while the second one is only the zoom on this range while the histogram is generated from all observations of the trace (both small and big duration values). To make the comparison between two figures easier, density distributions computed for the bottom plots of Figure 6.5 are presented as fat lines on middle and right plots of Figure 6.6(a). Indeed, when histogram is computed for the whole trace, all small values fit into a single bin. Although such abstraction is correct, it is not precise enough for our simulation purposes. The reason behind it is that if during the SimGrid execution this bin is chosen, the simulator will pick any random value uniformly between 2 and 3 microseconds. Consequently, not only that the kernel variability is not well simulated, but overall simulation accuracy decreases. There are even cases where the simulator injects values that are much lower than the best possible ones on the real machine, which may hinder the scheduling of the runtime and provide completely

unrealistic predictions. Therefore, when we applied these models for simulating Cholesky and LU applications, we observed much worse predictions accuracy compared to the simulations using a simple constant (mean) value for each kernel.

An intuitive solution for this problem is to divide observations in two groups and analyze them separately (as it was performed in the bottom two plots of Figure 6.5). Indeed, for this case or for the histograms with manually chosen bins, the simulation predictions are much more accurate, as we could carefully tune the values to optimally fit the distribution. However, other kernels may have completely different performance distributions, possibly with more than two groups or some other specific behavior. Moreover, manually analyzing traces is very tedious as it has to be done for every new kernel, block dimension and machine. Therefore, this process clearly needs to be fully automatized.

Another approach is to use irregular histograms [DM09] that choose bins non-uniformly, capturing better the distributions. The proposed solution aims at combining good properties of equal-width histograms, that are capturing well the low density regions, and the equal-area histograms which are good for capturing spikes in the distribution. The results of such approach applied on the same `GEMM` trace are displayed in Figure 6.6(b). The bars on the left plot are very narrow and hard to inspect, due to the high precision of the bin parameters. However, zooming on two regions reveals that this representation indeed approximates real distribution much better than the standard histograms. Instead of having a single huge bin for the smaller values, middle plot of Figure 6.6(b) reveals that irregular histogram has two bins with the boundaries chosen in a much better way, although there is still a room for improvement. Additionally, the distribution of higher durations (right plot) is also captured in more details. A small shortcoming of this approach is that the histogram descriptions doubles, increasing the size of the file containing its outputs. Still, for all applications and matrices we have studied so far, this negative effect is negligible. Finally, when using estimations provided by this type of histograms in simulation, application performance predictions proved to be very accurate.

Although this technique allows for obtaining different simulated schedules by changing the seed of the simulation, no significant gain in term of accuracy could be observed for the applications and machines we used so far. The makespan is always very similar in both cases (mean duration vs. random duration following an irregular histogram approximation of the original distribution). Nonetheless, the main reason behind it is that the linear algebra applications used in our experiments are highly optimized for the StarPU runtime. We strongly believe that in some more complex use cases, using fine models such as irregular histograms may provide more accurate predictions.

6.3 Evaluation methodology

We conducted series of experiments to (in)validate our modeling approach, searching for the machines and parameter setup where simulation results would not match the reality. If found, we would investigate model weaknesses or sometimes even the reasons why real executions where not behaving as expected. All conclusions were drawn from analyzing and comparing the GFLOPS rate, the makespans and the traces of StarPU on one hand (*Native*), and of StarPU on top of SimGrid (*SimGrid*) on the other.

Before running applications, StarPU needs to obtain a calibration of the platform, which consists in measuring bandwidths and latencies of communication between each processing unit, together with evaluating timings of computation kernels. Such information is used to guide StarPU schedulers' decisions when delegating tasks to available workers. StarPU has thus been extended to generate at the same time a SimGrid description of the platform (.xml file), which is later used in simulation. It is important to understand that only the calibration, which is meant to be run once on the target system, is used in the *SimGrid* simulation and that it is not linked to the application being studied. The only condition is that the application can use only computation kernels that have been measured. Such a clear separation allowed the simulations presented in this thesis to be easily performed on a personal laptop. This separation also allows for simulating

machines we don't have access to, knowing merely their characteristics.

Therefore, our typical experimentation workflow consists in iterating through the following phases:

1. We calibrate the target machine and the kernels using the StarPU calibration.
2. We do many simulations of the target machine on a commodity laptop.
3. We experiment on a real target platform.
4. We do various analysis and comparison of the two experimental sets, validating simulation predictions or potentially searching for the cause of the results discrepancy.

This whole work was done in the spirit of open science and reproducible research. Both StarPU and SimGrid software are free and available online. All the source code and experiment results presented in this paper are publicly available [SSW]. Supplementary data, which is not presented in this thesis are also available at the same location along with all the scripts, raw data files and traces which allow to reproduce this work. The methodology used during the whole project is a good example of conducting an exhaustive, coherent and comprehensible research (see Chapter 4 for more details).

Finally, assessing the impact of our various modeling attempts is quite difficult. Some of them are specifically linked to the modeling of the StarPU runtime, while others are more linked to the modeling of communications or to the computation variability. Obtaining a good predictive power is the combination of a series of improvements. Hence, comparing different runtime modeling options with a native execution while having a poor modeling of communications and computations would not be very meaningful. So instead, we evaluate our different runtime modeling options while using the best options for communication and computation modeling we found. Likewise, when we evaluate various communication modeling options, we always use the best modeling option of runtime and computations, which allows us to evaluate how much accuracy may be lost by overlooking this particular aspect.

6.4 Where the model needs to be carefully adapted

The first implementation of StarPU-SimGrid with initial simplistic models already provided accurate simulation predictions for simple use cases such as the one in Figure 6.7. This is a typical analysis performed throughout our study, where we compare the performance results obtained by the native execution of StarPU on the target machine (*Native* with a solid line) to the SimGrid prediction for the same machine and parameters (*SimGrid* with dotted line). Following the trends in our community, we do not contrast overall duration of the application execution (makespans), but instead we focus on the effective GFLOPS rates. We also vary the matrix size (on the x-axis), by increasing the number of blocks, keeping the block size itself fixed (960×960 for GPUs, and 320×320 for CPUs). Therefore, the line for both Native and SimGrid in Figure 6.7 actually represents a set of 30 measurements, for the matrix dimension going from 960 to 28800. The fact that two lines are very close to each other signifies good results, as SimGrid accurately predicts the performance of the target machine.

However, when we enlarged the evaluation by increasing the matrix size or by changing the environment setup in certain ways, the results were not so precise any more. In the rest of this section, we present three examples of the challenges we faced as well as the techniques we used to overcome them.

6.4.1 GPU memory limit

Probably the most influential parameter for accurate modeling of runtime proved to be the size of GPU memory. When there is not enough space to keep all the necessary matrix blocks, hardware limits force the scheduler to swap data back and forth between the CPUs main memory and

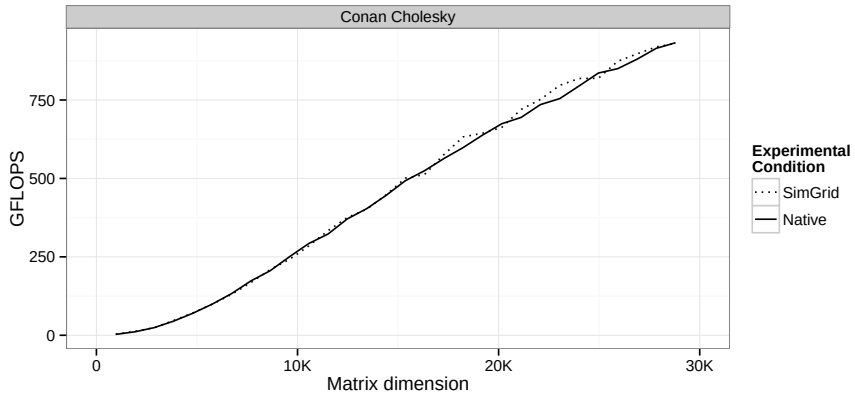


Figure 6.7: Initial results of StarPU simulation for the simplistic use cases were already very accurate. More complex scenarious required more sophisticated models.

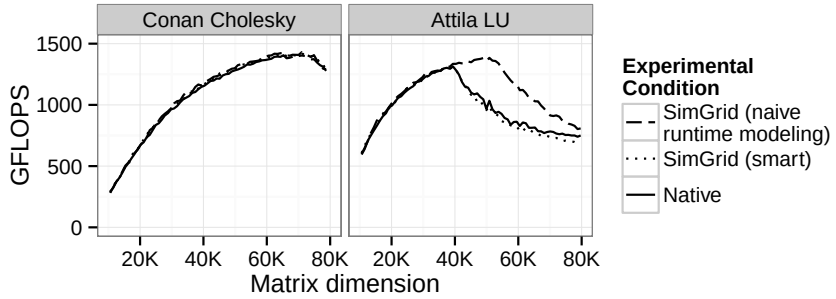


Figure 6.8: Illustrating the influence of modeling runtime. Careless modeling of runtime may be perfectly harmless in some cases, it turns out to be misleading in others.

GPUs. These data movements saturate the PCI bus, producing a tremendous impact on overall performance. It is thus critical to keep track of the total size of memory load allocated by StarPU during the simulation, in order to ensure the scheduler behaves in the same way for both real native executions and simulations.

Figure 6.8 illustrates the importance of taking into account this parameter. Each curve depicts the GFLOPS rate of experiments for 72 different matrix dimensions (the matrix dimension 80,000 corresponds to ≈25GiB). The *Native* solid line shows the execution of StarPU on the native machine, while the other two are the results of the simulation: *naive* for execution without runtime adjustments and *smart* with GPU memory limit included. The left plot depicts a situation where all these optimizations have very little influence as both *naive* and *smart* lines are almost overlapping with the *Native* line. The reason behind it is that the Conan machine has GPUs with bigger memory capacities and StarPU manages to control execution of Cholesky application without any unnecessary memory swapping. On the other hand, for some other combinations of machines and applications (right plot), having a precise modeling of runtime is critical as otherwise, simulation may highly overestimate the performance for large matrix sizes. Nonetheless, we remind that the excellent predictions achieved in these examples are also the result of the careful modeling of communications and computations.

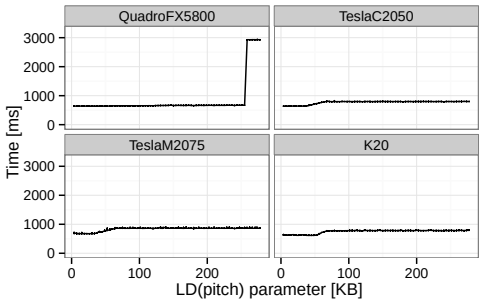


Figure 6.9: Transfer time of 3,600 KB using `cudaMemcpy2D` depending on the pitch of the matrix.

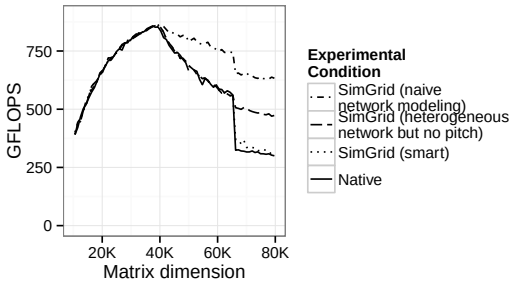


Figure 6.10: Performance of the LU application on Hannibal (QuadroFX5800 GPUs) using different modeling assumptions.

6.4.2 Specific GPUs/CUDA version

It was previously discussed how the difference between GPUs or CUDA versions can significantly influence the application performance. Each platform needs thus to be carefully calibrated whenever the environment is changed. Additionally, to move chunks of matrices between resources, StarPU relies on the `cudaMemcpy2D` function. The performance of this function is not exactly the same as the one of `cudaMemcpy`, which was used in the original calibration process. More important, it turns out that the pitch (i.e., the stride of the original matrices) can have a substantial impact on transfer time on some GPUs (see Figure 6.9) whereas it can be relatively safely ignored on others. Therefore, communication time is modeled as a piece-wise linear function of data payload whose slope and intercept depend on the pitch of the matrix.

Again, for a given application and a given target architecture, it may not be necessary to take care of all such details to obtain a good prediction. For example, as illustrated on Figure 6.10, a naive network modeling such as the one on Figure 5.2(a) proved excellent predictions when matrix dimension is smaller than 40,000. Beyond such size, a more precise modeling of the network (as in Figure 5.2(b)) is necessary. Beyond 66,240, the behavior of `cudaMemcpy2D` changes drastically and has to be correctly modeled to obtain a good prediction of the performances.

This is certainly not the only example of an unexpected behavior related to a specific GPU type. It is thus important that our models can easily integrate such knowledge into simulation and finally provide accurate predictions even for this kind of scenarios.

6.4.3 Elaborated communication model for complex machines

As discussed in Subsection 5.4.1, we have developed three different types of communication models. Figure 6.11 shows results of simulating the Idgraf machine with each of these network models. We remind that Idgraf machine has 8 GPUs connected in a very peculiar way and thus time to transfer data from one GPU to another varies greatly on their distance and the contention on the particular links (more details on its architecture is given in Figure 5.4).

As expected, more sophisticated models provide much more accurate predictions, especially for larger matrix size. However, there is still a room for improvement as can be observed from all simulation results for matrix dimension larger than 90,000. In such cases, the huge matrix size and the limited GPU memory lead to many data transfers, which completely saturates the PCI bus. Even the treelike model does not handle well this amount of network contention, although it was specifically designed for this particular machine.

We have illustrated only a few representative examples from a list of numerous issues that had to be addressed during our study. After improving the implementation and applying the

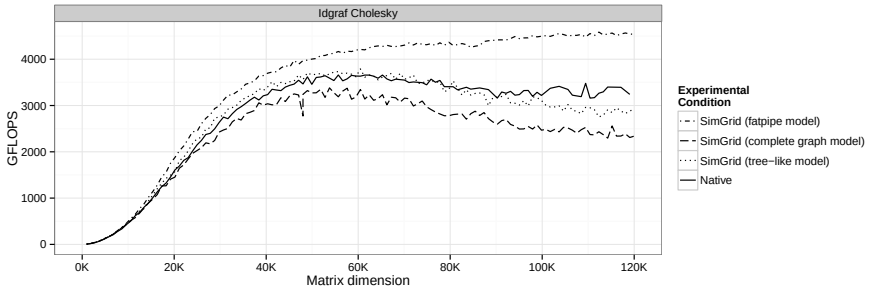


Figure 6.11: Simulating machines with complex architectures such as Idgraf (see Figure 5.4) require more elaborated models.

appropriate models, we are able to provide very accurate SimGrid predictions of the StarPU runtime. In the following sections, we present the final results for a wide range of machines using the optimal models.

6.5 Accurate performance predictions for hybrid machines

Figure 6.12 depicts the performance as a function of the size of the matrix for the two applications LU and Cholesky and for the seven different hybrid systems described in Table 6.1. For all combinations, the prediction obtained with SimGrid is very accurate. There are a couple of scenarios for which the error is larger than a few percents but such discrepancies are actually due to the fact that the prototype experimental machines are sometimes perturbed by other users, operating system, etc. Regardless of that, these errors stay always lower than 6%, which is still very precise. Additionally, the trend is perfectly predicted as well as the size beyond which performance drops.

A closer look at traces allows for seeing that this approach does not only provide a good estimation of the total runtime but also offers an accurate simulation of the scheduling details. In Figure 6.13, we compare traces from *Native* execution with *SimGrid* simulation, focusing only on the most important states. *DriverCopy* corresponds to the CPU managing a data transfer to the GPU, while `POTRF`, `TRSM` and `GEMM` are the three kernels that compose of the Cholesky application. One can observe that GPUs perform all the computations, while CPUs provide them with data. Additionally, CPU0 is responsible for doing all the scheduling. Since even with the same parameters, native traces differ from one execution to another, a point-to-point comparison with a simulation trace would not make sense. However, we can check that both the *Native* and the *SimGrid* traces are extremely close, which allows for studying and understand the potential weaknesses of a scheduler.

6.6 Using both CPUs and GPUs for computation

It was illustrated earlier in Figure 6.2 that CPUs do not have a major influence on the performance of the dense linear algebra applications used in this work. However, this may be different for other kind of applications with less optimized kernels. Therefore, we also investigated the situation where both CPUs and GPUs are used for doing computation. The results are depicted in Figure 6.14, which shows once again how our *SimGrid* predictions compare favorably to real experiments in hybrid setups.

Again, to illustrate the fact that not only the makespan is accurately predicted but that the whole scheduling is correctly modeled as well, we compare two execution traces (see Figure 6.15). These traces correspond to the experiments performed for the Mirage machine that has 3 GPUs and 12 cores. 8 of them were used for doing computations, 3 for GPU transfer management and

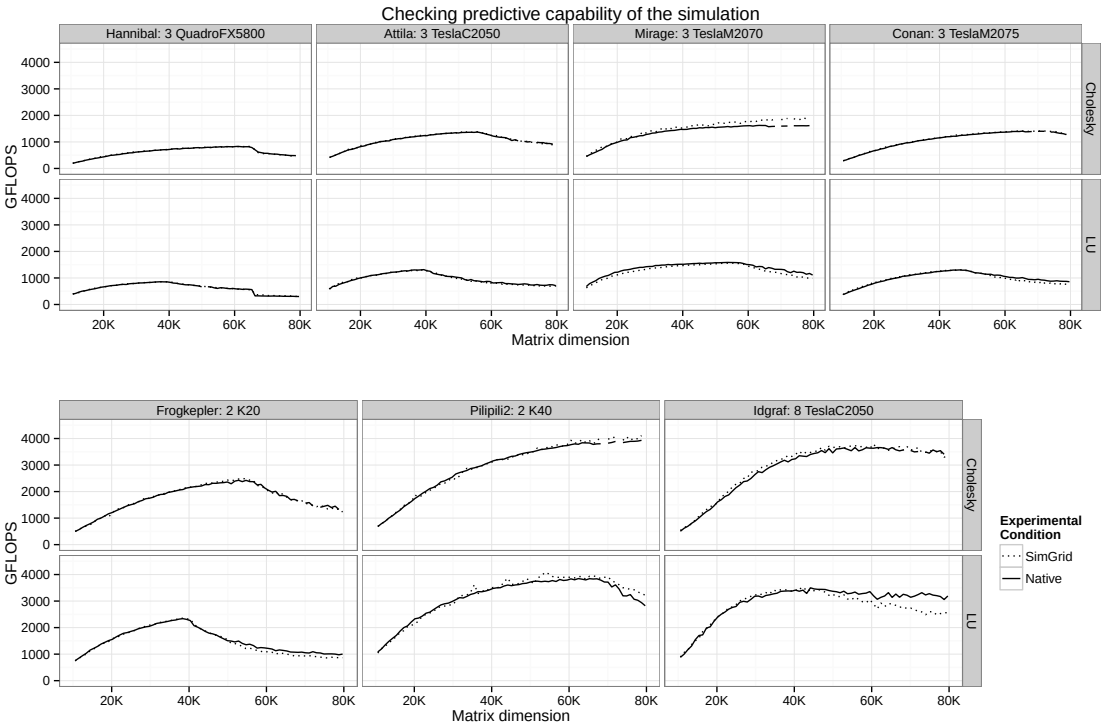


Figure 6.12: Checking predictive capability of our simulator in a wide range of settings.

1 was dedicated to scheduling and control of the runtime. These traces can be compared to the ones of Figure 6.13, since in both cases, the same application and matrix size are used (only the GPU slightly differs). The conclusion is that adding 8 CPU cores improved the performance by approximately 20% and such conclusion can be drawn solely on simulations.

To convince the readers even further, we provide another trace comparison in Figure 6.16, again based on the same experimental setup (the Mirage machine using 8 cores for computations, 3 for GPU transfers and 1 for scheduling) and application (Cholesky) only using a different matrix size and a different implementation of the kernels. In this execution we did not use the MKL but simple non-over optimized kernels and thus executing kernels on CPUs was 10 times slower than in all other results from CPUs presented in this thesis. Although these results are not necessarily interesting in terms of performance, we still think that it is important to show that we manage to obtain accurate performance prediction in such context as well since not all users may have access to the proprietary Intel libraries.

The general shape of the schedule in Figure 6.16 is the same in both *Native* and *SimGrid* traces and one can observe several characteristics of the scheduling algorithm:

- The shortest kernel (`POTRF`) was executed mostly on CPU0, except in the very beginning and at the end where it was executed on GPUs. This is due to the fact that although the execution of `POTRF` is faster on the GPUs, GPUs are relatively more efficient for `GEMM` operations than CPU resources. The GPU resources should thus not be "wasted" for such kernels when the application reaches steady-state. However, using the GPUs for such kernels at the beginning and at the end of the schedule makes sense, since it allows for releasing available tasks as soon as possible in the beginning and to improve the execution of the critical ones in the end.

- The `TRSM` kernels are executed on every CPU worker in the first part of the execution, while they are performed regularly and much faster on the GPUs in the rest of the execution.

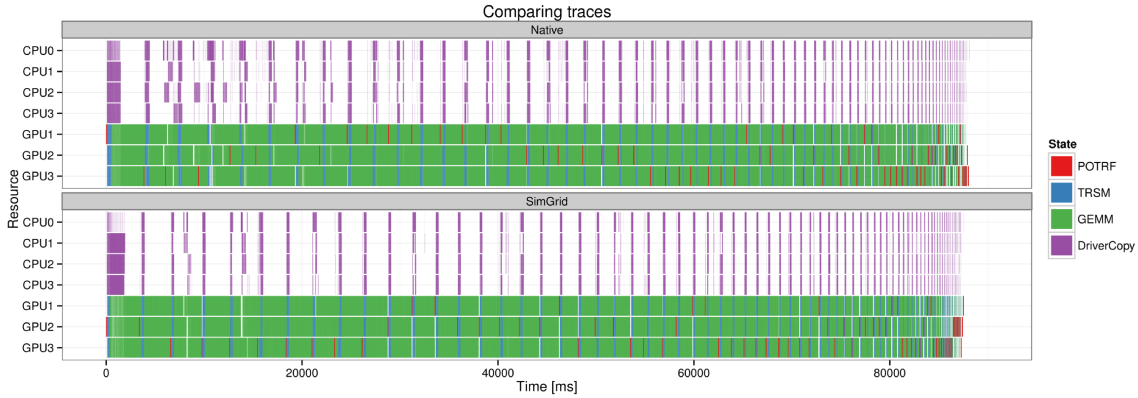


Figure 6.13: Comparing execution traces (native execution on top vs. simulated execution at the bottom) of the Cholesky application with a $72,000 \times 72,000$ matrix on the Conan machine but using only GPU resources for processing the application.

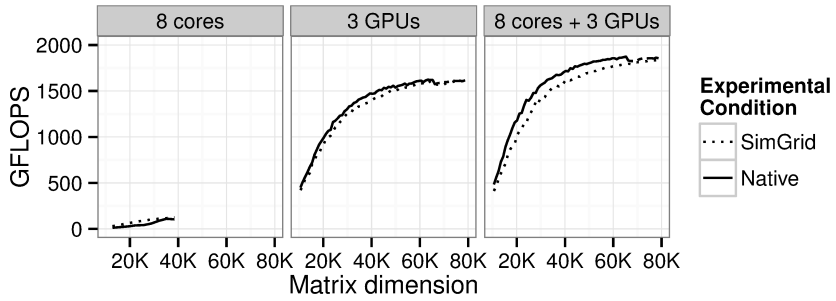


Figure 6.14: Illustrating simulation accuracy for Cholesky application using different resources of the Mirage machine.

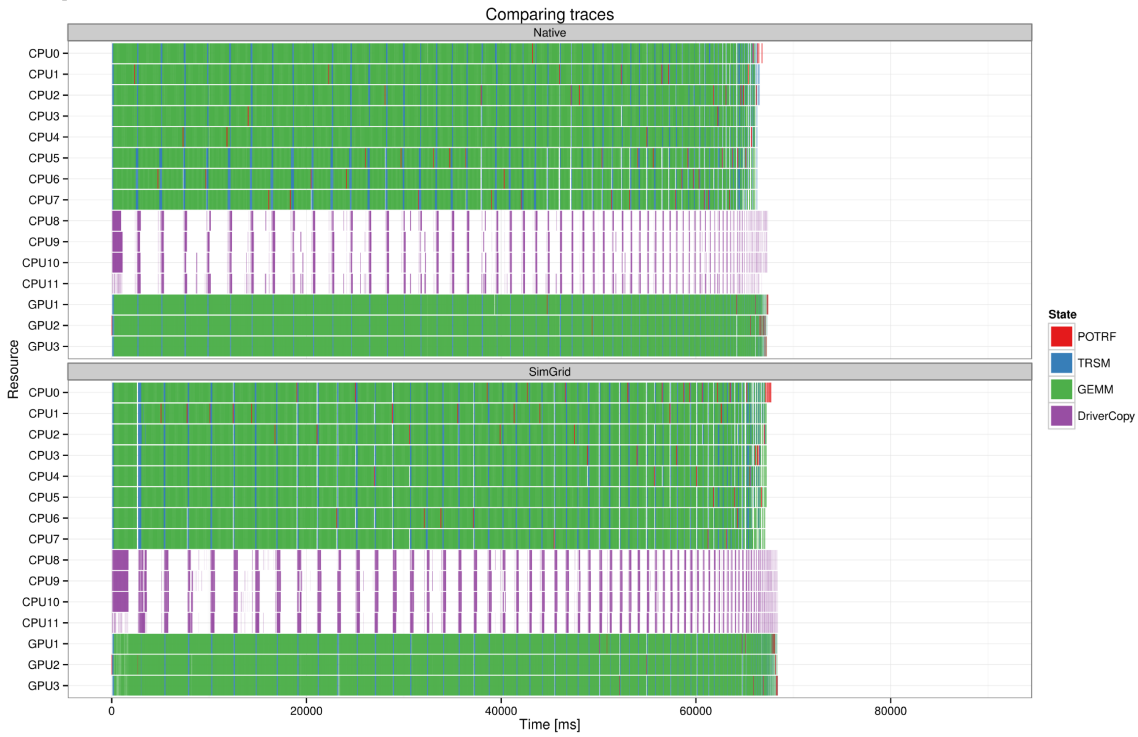


Figure 6.15: Comparing execution traces (native execution on top vs. simulated execution at the bottom) of the Cholesky application with a $72,000 \times 72,000$ matrix on the Mirage machine using 8 cores and 3 GPUs as workers. Adding 8 cores, improved the performance by approximatelly 20% compared to the performances obtained in Figure 6.13.

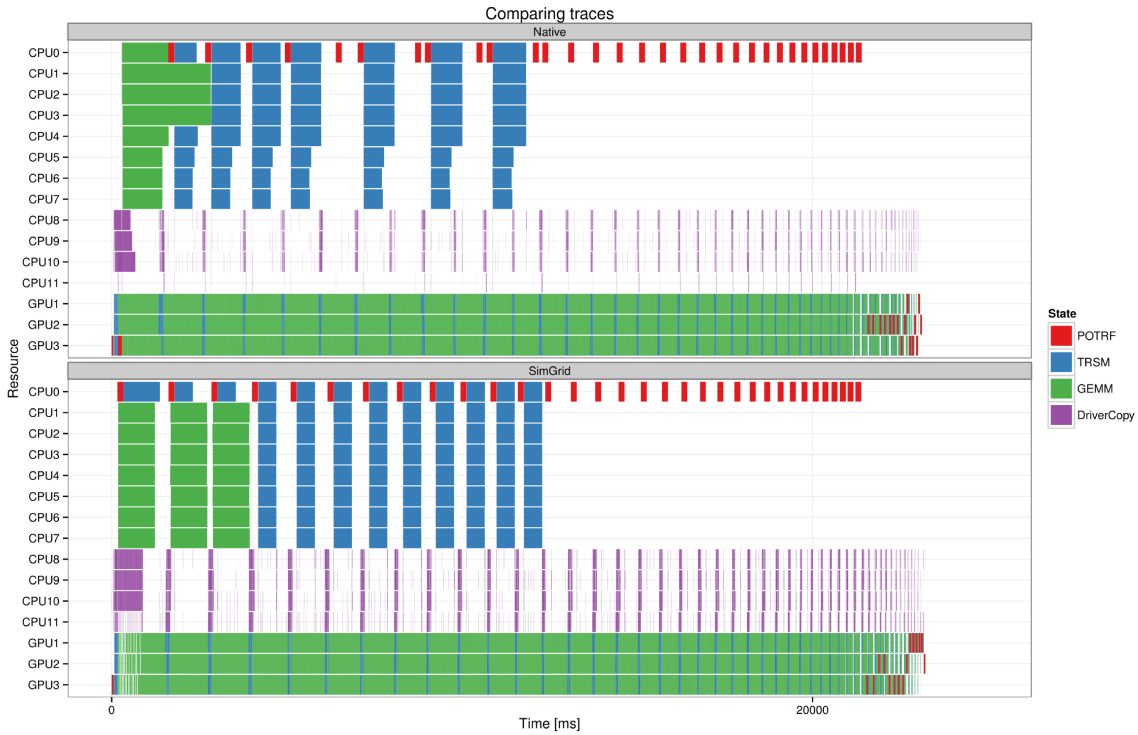


Figure 6.16: Comparing execution traces (native execution on top vs. simulated execution at the bottom) of the Cholesky application with a $48,000 \times 48,000$ matrix on the Mirage machine using 8 cores and 3 GPUs as workers. Executing kernels on CPUs is much longer since Intel MKL libraries were not used, however simulation predictions are still very precise.

- Only few `GEMM` kernels are run on CPUs in the initial phase of the execution, while they are constantly executed on GPUs. Additionally, execution times of this kernel on GPUs decreases during the application run.

All these phenomena are also present in the *SimGrid* trace. As expected, scheduling is not identical, since StarPU is dynamic and with two native executions with the same parameters traces would not be exactly the same neither. For example, there is a slight difference in the distribution (the number of times) of `TRSM` kernels' allocation between CPUs and GPUs. It can be explained by the fact that the execution time variability of this kernel was not accounted for in this simulation. There was thus interest for the scheduler to execute 9 series of such kernels on the CPUs in simulation although only 7 of them were done on the CPUs in the native execution. We remind that this number varies from one native run to the other and that the simulation is thus only slightly idealizing the real conditions in a deterministic way. However, all the trends of the real execution are correctly accounted by the simulation.

6.7 Where the model breaks and is harder to adapt: NUMA machines

The current shortcoming of our model is for the simulation of NUMA architectures. When executing an application using only CPU resources, there is no explicit data transfers, as the workers use shared memory to exchange data. Yet, the time to access the shared data depends on the used memory banks. Additionally, effective memory bandwidth depends on efficient utilization and is particularly sensitive to suboptimal block size and memory strides. Such aspects are extremely difficult to model and are currently ignored in our simulations. Although this is not too harmful

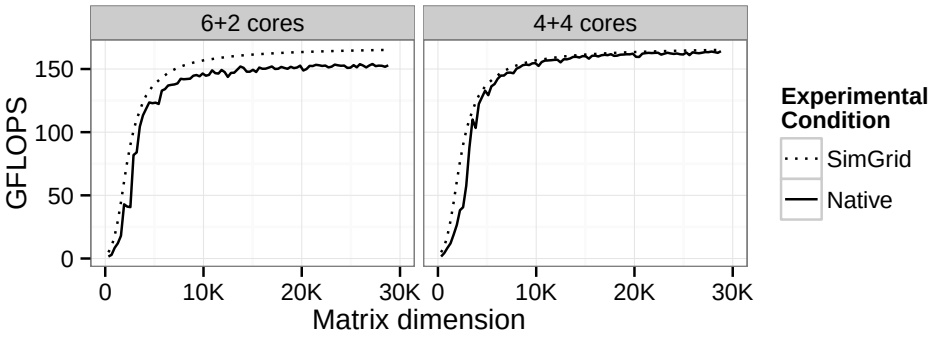


Figure 6.17: Illustrating the impact of deployment when using 8 cores on two NUMA nodes on the Mirage machine.

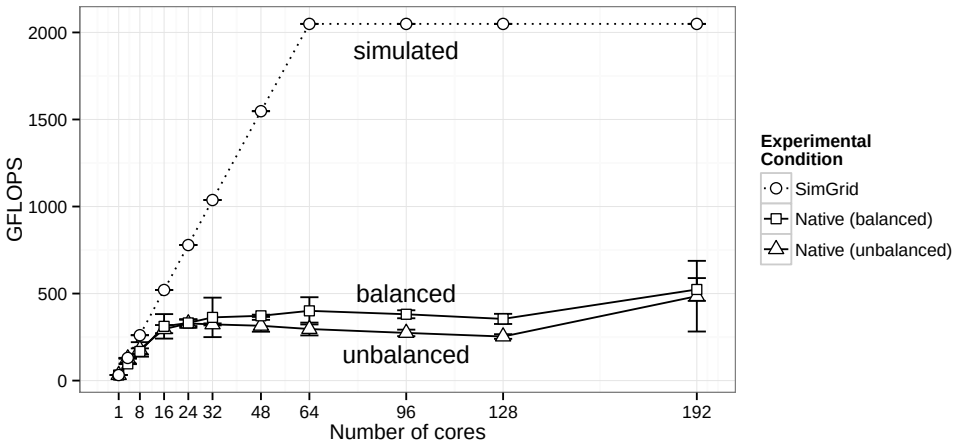


Figure 6.18: Simulation predictions of Cholesky application with a $32,000 \times 32,000$ matrix (block size 320×320) on large NUMA Idchire machine are precise for a small number of cores, but scale badly. The reason is that the memory is shared, while models are not taking into account various NUMA effects.

for systems with relatively few cores (like the ones we used in the previous experiments), it can be much more annoying with larger architectures.

We present a set of experiments that illustrates the impact of the NUMA effect on the realism of our predictions. On Figure 6.17, there are two different experimentation setups on the same Mirage machine, which has 12 cores that are distributed on two NUMA nodes (6+6). In these experiments, we used only 8 cores, since the other 4 are normally in hybrid executions dedicated to GPU transfers and scheduling as it is generally how the best performances are obtained in practice. The plot on the left use an improper balancing of computing threads as 6 threads are pinned to one node and only 2 on second node. On the other hand, the plot on the right is well balanced (4+4). Since the simulation does not currently take the NUMA topology into account, the *SimGrid* prediction is identical in both the left and the right plot. As one could however expect, resources are more efficiently used in the second case and thus the performance of *Native* executions are better in the second scenario and match the *SimGrid* predictions. In such small platforms, our approach can thus provide a sound estimation of the performance that one should expect but cannot account for the performance loss due to a bad deployment. In more extreme setups, our predictions are however likely to be too optimistic.

To illustrate even further such difficulty, we conducted a similar experiment on Idchire that has 24 NUMA nodes each with 8 cores. On Figure 6.18, one can once again observe that there is a significant difference in terms of performance between a good and a bad balancing of the cores for the native executions. On such platforms, our SimGrid results provide decent predictions only for execution with a very small number of cores, while for the other setups it greatly overestimates the capabilities of the system. This is explained by our current inability to account for the performance degradation of interfering memory-bound kernels, the NUMA effects and inter-node traffic.

These are known limits of our approach that may be overcome by keeping track of data localization and trying to model data movement more precisely. However, although the performance loss incurred by interfering kernels that contend on the memory hierarchy can be measured, it is quite difficult to model. We are thus still investigating how to account for such situation.

6.8 Typical studies enabled by such approach

Confidence in the simulation precision allows researchers to completely rely on the SimGrid mode execution of StarPU. When doing exploration studies, they can run only simulations, which is much faster and does not require access to the remote machine. Only when they know exactly what code and parameters they want to study, they can do native experiments in order to validate the simulation predictions and obtain real machine results. Such approach can be applied on numerous use cases including various scheduler studies. Additionally, platform description of the machine can be manually changed, so users can get a performance estimation for a machine that does not exist or that they do no have access to.

6.8.1 Studying schedulers

One of the main challenges that StarPU developers encounter is how to efficiently exploit all the available heterogeneous resources. To address it, they develop different scheduling techniques that may be specifically tailored for a given type of machine architectures.

For example, the reason for the performance drop observed on Figure 6.12 and which is more and more critical with newer GPUs can be explained by the need to move data back and forth between the GPUs and the main memory whenever matrix size exceeds the memory size of the GPUs. The scheduler we used in Figure 6.12 is the *DMDA* (*Deque Model Data Aware*) scheduler. Although it schedules tasks where their termination time (including data transfer time) will be minimal, it does not take care of the number of available data buffers on each GPU. Such greedy strategy may be harmful as one GPU may be overloaded with work and forced to evict some data, as it cannot handle the whole matrix. Two other strategies *DMDAR* and *DMDAS* were designed

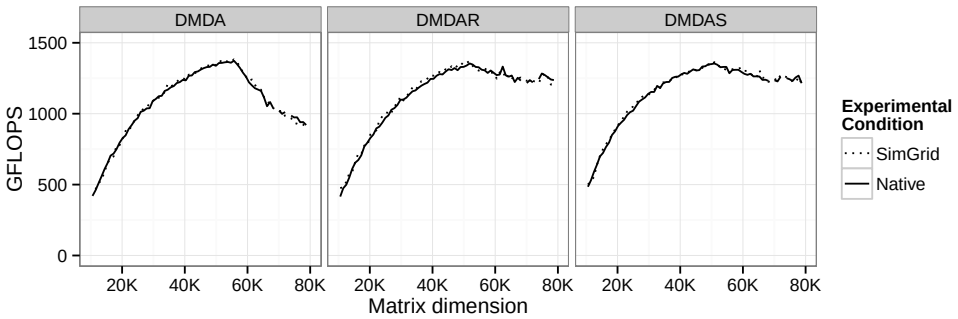


Figure 6.19: Cholesky on Attila: studying the impact of different schedulers.

to execute in priority tasks whose data is already on the GPU, before tasks whose data is not yet available.

Therefore, we decided to check whether these two other schedulers could stabilize performances at the peak or not. To this end, we first ran the corresponding simulations and obtained a positive answer (Figure 6.19). Later, when the target system became accessible again, we confirmed these results by running the same experiments and as can be seen on Figure 6.19, our simulation predictions were again perfectly accurate.

Researchers studying different scheduling algorithms on the StarPU implementation of MAGMA/MORSE [MOR] (now Chameleon [Cha]) applications also benefit from this simulation approach [ABED+15]. They have started to use extensively SimGrid simulations for screening experiments. Indeed a major advantage of doing simulations rather than running real experiments, is that simulations are fast, reproducible (, which simplifies the analysis) and do not require an access to the remote experimental cluster. Since our simulations provides reliable predictions, it is possible to screen a wide range of parameters and quickly see whether a given approach seems effective or not. Such exploratory measurements thus help refining the set of configurations that are worth being tested in real environments.

6.8.2 Studying hypothetical platforms

Apart from rapidly testing different scheduling alternatives on calibrated platforms, SimGrid can as well be used for investigating potential performance of fictional machines. One can modify an existing platform description or create a new one, either by hand or with a script, adding more processing units, changing the latencies and bandwidths of the network or changing the network topology.

A good example of such approach can be observed in the study performed by Agullo et al. [ABED+15]. In their work, the authors were comparing theoretical performance boundaries of Cholesky factorization with the ones observed on real heterogeneous machine. As a middle solution, the authors additionally display a simulated performance prediction of the same target machine, only with a slightly modified platform description, having the communication cost equal to zero. Although this means that all the data transfers between RAM and GPUs are instantaneous, which is impossible on a real machine, these results are still very interesting as they show the maximal potential of the processing units of the target platform as well as the scheduler behavior.

Another feature enabled by SimGrid is extrapolation, performed by increasing the number of resources and investigating possible performance. However, such simulation predictions should be taken with caution since they are based on a calibration of the initial machine, which can not estimate well PCI bus contention, NUMA effects or any other phenomena that is introduced with scaling. Therefore, the reliability of these predictions would be questionable, which is contrary to

the accurate results were presented throughout this chapter.

Chapter 7

Performance Prediction of Sparse Linear Algebra Applications

After having achieved accurate simulation predictions for dense linear algebra applications such as Cholesky and LU decompositions, we decided to focus on sparse linear solvers as we wanted to validate that our approach could be applied to more irregular workloads as well.

In this study we consider `qr_mumps` [ABGL14, ABGL13], an implementation of the MUMPS sparse direct solver on top of the StarPU runtime system. In this approach a multifrontal QR factorization (a highly irregular algorithm) is programmed using the STF model (see Subsection 2.1.4).

In the rest of the chapter, we provide details about this application, and present few necessary adjustments needed for simulating its execution with SimGrid. Later, we follow the same organization as in Chapter 6. First, we describe the experimental settings, some specificities of the kernel modeling and the methodology. Then we present an experimental evaluation and a few typical use cases. Due to the difference of complexity between dense and sparse algorithms, we had to adapt our evaluation to this specific context. In particular, modeling of the computational parts had to be modified, as the duration of `qr_mumps` kernels completely depend on kernel parameters that vary during the execution. Additionally, as we validate our approach on a set of of sparse matrices, which all generate distinctive factorization and very different DAGs, we reduced our experimental campaign to fewer target machines. Finally, to demonstrate the diversity of the studies offered by simulation approach, we present different use cases from the ones of Section 6.8: we show how our tool allows for conducting studies related to the memory footprint of the application, as well as extrapolation of the target machines.

7.1 `qr_mumps`, a task-based multifrontal solver

The multifrontal method, introduced by Duff and Reid [DR83] as a method for the factorization of sparse, symmetric linear systems, can be adapted to the QR factorization of a sparse matrix thanks to the fact that the R factor of a matrix A and the Cholesky factor of the normal equation matrix A^TA share the same structure under the hypothesis that the matrix A is *Strong Hall*. As in the Cholesky case, the multifrontal QR factorization is based on the concept of *elimination tree* introduced by Schreiber [Sch82] expressing the dependencies between elimination of unknowns. Each vertex f of the tree is associated with k_f unknowns of A. The coefficients of the corresponding k_f columns and all the other coefficients affected by their elimination are assembled together into a relatively small dense matrix, called *frontal matrix* or, simply, *front*, associated with the tree node (see Figure 7.2). An edge of the tree represents a dependency between such fronts. The elimination tree is thus a topological order for the elimination of the unknowns; a front can only be eliminated after its children. We refer to [ADP96, Dav11, But13] for further details on high performance implementation of multifrontal QR methods.

```
forall fronts f in topological order
   ! allocate and initialize front
   call activate(f)

   forall children c of f
      forall blockcolumns j=1...n in c
         ! assemble column j of c into f
         call assemble(c(j), f)
      end do
      ! Deactivate child
      call deactivate(c)
   end do

   forall panels p=1...n in f
      ! panel reduction of column p
      call panel(f(p))
      forall blockcolumns u=p+1...n in f
         ! update of column u with panel p
         call update(f(p), f(u))
      end do
   end do
end do
```

```
forall fronts f in topological order
   ! allocate and initialize front
   call submit(activate, f:RW, children(f):R)

   forall children c of f
      forall blockcolumns j=1...n in c
         ! assemble column j of c into f
         call submit(assemble, c(j):R, f:RW)
      end do
      ! Deactivate child
      call submit(deactivate, c:RW)
   end do

   forall panels p=1...n in f
      ! panel reduction of column p
      call submit(panel, f(p):RW)
      forall blockcolumns u=p+1...n in f
         ! update of column u with panel p
         call submit(update, f(p):R, f(u):RW)
      end do
   end do
end do
call wait_tasks_completion()
```

Figure 7.1: Sequential version (*left*) and corresponding STF version from [ABGL14] (*right*) of the multifrontal QR factorization with 1D partitioning of frontal matrices.

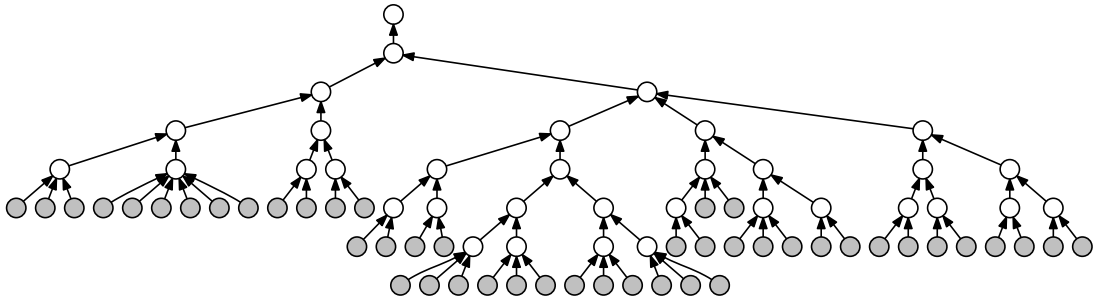


Figure 7.2: Typical elimination tree: each node corresponds to a front and the resulting tree is traversed from the bottom to the top. To reduce the overhead incurred by managing a large number of fronts, subtrees are pruned and aggregated into optimized sequential tasks (`Do_subtree`) depicted in gray.

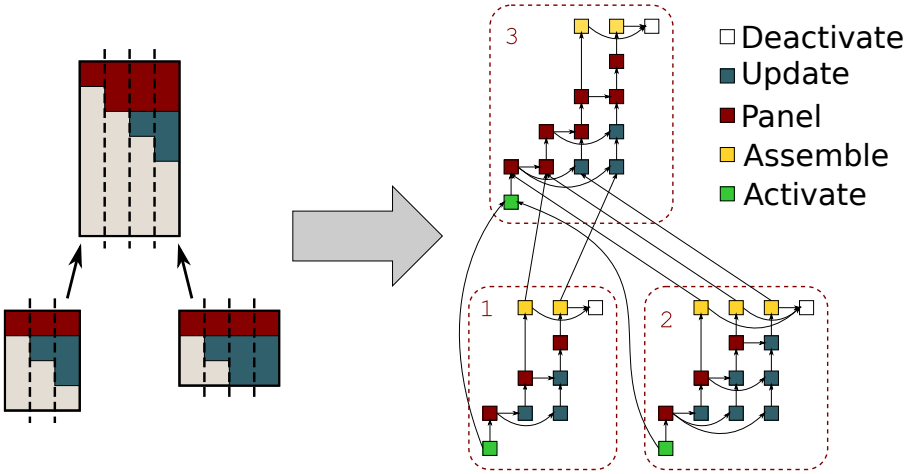


Figure 7.3: Processing a front requires a complexe series of `Panel` and `Update` tasks induced by the staircase structure. The dependencies between these operations expressed by the STF code leads to a fine-grain DAG dynamically scheduled by the runtime system.

The multifrontal QR factorization then consists in a tree traversal following a topological order (see line 1 in Figure 7.1 (left)) for eliminating the fronts. First, the activation (line 3) allocates and initializes the front data structure. The front can then be assembled (lines 5-12) by stacking the matrix rows associated with the k_f unknowns with uneliminated rows resulting from the processing of child nodes. Once assembled, the k_f unknowns are eliminated through a complete QR factorization of the front (lines 14-21). This produces k_f rows of the global R factor, a number of Householder reflectors that implicitly represent the global Q factor and a *contribution block* formed by the remaining rows. These rows will be assembled into the parent front together with the contribution blocks from all the sibling fronts.

One distinctive feature of the multifrontal QR factorization is that frontal matrices are not entirely full but, prior to their factorization, can be permuted into a staircase structure that allows for moving many zero coefficients in the bottom-left corner of the front (see Figure 7.3) and for ignoring them in the subsequent computation. Although this allows for a considerable saving in the number of operations, it makes the workload extremely irregular and the cost of kernels extremely hard to predict even in the case where a regular partitioning is applied to fronts, which makes it challenging to model.

Figure 7.1 (right) shows the 1D STF version [ABGL14, ABGL13] (used in the present study) of the multifrontal QR factorization described above. Instead of making direct function calls (`Activate`, `Assemble`, `Deactivate`, `Panel`, `Update`), the equivalent STF code submits the corresponding tasks (see Figure 7.3). Since the data onto which these functions operate as well as their access mode (Read, Write or Read/Write) are also specified, the runtime system can perform the superscalar analysis while the submission of task is progressing. For instance, as an `Assemble` task accesses a block-column `f(i)` before a `Panel` task accesses the same block-column in Write mode, a dependency between those two tasks is inferred. Since the number of fronts in the elimination tree is commonly much larger than the number of resources, the above-mentioned partitioning strategy is not applied to all of them. A technique similar to that proposed by Geist and Ng [GN89] and described in [But13] under the name of *logical tree pruning* is used. Through this technique, a layer in the elimination tree is identified such that each subtree rooted at this layer is treated in a single task with a purely sequential code. This new type of tasks, which are named `Do_subtree`, is represented in Figure 7.2 as the gray layer. It was previously shown [ABGL14] that the STF programming model allows for designing a code that achieves a great performance and scalability as well as an excellent robustness when it comes to memory consumption.

As a conclusion, the multifrontal method provides two distinct sources of concurrency: tree and node parallelism. The first one stems from the fact that fronts in separate branches are independent and can thus be processed concurrently; the second one from the fact that, if a front is large enough, multiple threads can be used to assemble and factorize it. Modern implementations exploit both sources of concurrency which makes scheduling difficult to predict, especially when relying on dynamic scheduling, which is necessary to fully exploit the parallelism delivered by such an irregular application.

7.2 Porting `qr_mumps` on top of SimGrid

Porting `qr_mumps` on top of SimGrid required only two minor modifications to the original `qr_mumps` code. The compilation process (along with the necessary environment variables) had to be adapted to the simulation mode of StarPU and the `main` function of the `qrm_test` program used to execute factorization with `qr_mumps` had to be changed for the `starpu_main` subroutine. Indeed, SimGrid has its own `main` function that is required to initialize the simulation before running the simulated application.

Compared to the previously described study on the simulation of dense linear algebra applications with StarPU and SimGrid (see Chapter 5 and 6), the main difficulty arises from the application structure and from the fact that tasks (computation kernels) are called with a wide range of input parameters. When working with dense matrices, it is common to use a global fixed block size and a given kernel type (e.g., `dgemm`) is therefore always called with the same parameters

Name	Processor	Number of Cores	Frequency	Memory	GPUs
Fourmi	Intel Xeon X5550	2×4	2.67GHz	2×12GiB	/
Riri	Intel Xeon E7-4870	4×10	2.4GHz	4×256GiB	/

Table 7.1: Machines used for the sparse linear algebra experiments.

throughout the execution, which makes its duration relatively stable and easy to model. In the `qr_mumps` factorization, the amount of work that has to be done by a given kernel greatly depends on its input parameters. These parameters may or may not be explicitly given to the StarPU runtime and we thus had to rework the `qr_mumps` task submission model to ensure StarPU can propagate these information to SimGrid. Some parts of the StarPU code responsible for interacting with SimGrid were also modified to detect specific kernel parameters and predict durations based on such parameters. We have also extended the StarPU tracing mechanism so that they are traced as well, which is indispensable to obtain traces that can be both analyzed and compared between real and simulated executions.

7.3 Experimental settings

To evaluate the quality of our approach, we used two different kind of nodes from the Plafrim [PLa] platform (see Table 7.1) The Fourmi nodes proved to be easier to model as their CPU architecture is well balanced with 4 cores sharing L3 cache on each of the 2 NUMA nodes. Such configuration leads to little cache contention. However, the RAM of these nodes is limited and thereby limits the matrices that can be factorized to a certain size. Although the huge memory of the Riri machine puts almost no restriction on the matrix choice, its memory hierarchy with 10 cores sharing the same L3 cache leads to cache contention that is harder to model.

The matrices we used for evaluating our approach are presented in Table 7.2 and come from the UF Sparse Matrix Collection [UFM] plus one from the HIRLAM [HIR] research program.

Table 7.2: Matrices used for the sparse linear algebra experiments.

Matrix	m	n	nz	GFLOPS
tp-6	143 000	1 010 000	11 500 000	277.7
karted	46 500	133 000	1 770 000	279.9
EternityII_E	11 100	262 000	1 570 000	566.7
degme	186 000	659 000	8 130 000	629.0
hirlam	1 390 000	452 000	2 710 000	2401.3
TF16	15 400	19 300	216 000	2656.0
e18	24 600	38 600	156 000	3399.1
Rucci1	1 980 000	110 000	7 790 000	12768.1
sls	1 750 000	62 700	6 800 000	22716.6
TF17	38 100	48 600	586 000	38209.3

7.4 Modeling `qr_mumps` kernels

Figure 7.4 shows the distribution of duration for each kernel for a typical `qr_mumps` factorization of the *e18* matrix. As expected, since some kernels have non fixed input parameters, variability is very important compared to the dense case and most kernels exhibit several modes. `Do_subtree` and `Activate` have only a few samples, but one can already suspect that they cannot be modeled by a simple random variable. For the `Panel`, `Update` and `Assemble` operations, the situation is even more certain: the several modes and particular shapes clearly originate from the structure of the application.

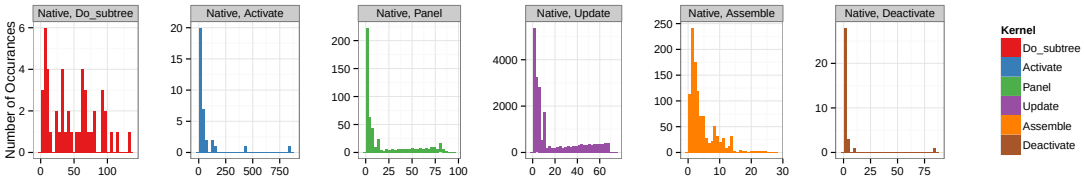


Figure 7.4: Distribution of the `qr_mumps` kernel duration when factorizing the *e18* matrix (see Table 7.2). The distribution shapes is similar for other matrices. Most kernels have a (difficult to model) multi-modal distribution.

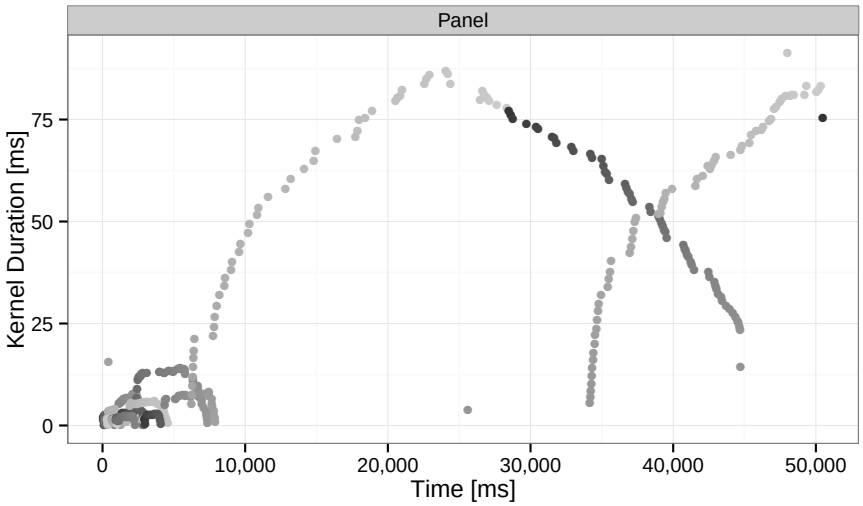


Figure 7.5: Duration of the `Panel` kernel as a time sequence for the *e18* matrix. The patterns suggest that this duration depends on specific parameters that evolve throughout the execution of the application.

To illustrate the structure of the application execution, we propose to focus on the `Panel` kernel. Indeed, this factorization function is called regularly and can thus be thought as a "temporal marker" that provides a signature of the application execution. To analyze this signature, we use a time sequence (see Figure 7.5) that depicts the evolution of the duration of `Panel` calls throughout the whole application. The x-axis represents the moment each `Panel` task started its execution while the y-axis represents its duration. The color is related to the task id of the task, which is stable during two executions of the application. The trace analyzed in this plot is the same as the one used in Figure 7.4, only with more details (starting time and task id) on the `Panel` kernel. Figure 7.5 illustrates the fact that the duration of the kernel implicitly depends on the time it started, more precisely on the task parameters that depend on the application structure. Such kind of visualization analysis allows for checking whether the phenomenon is structured or not, hence whether some external parameter influence the duration and should be taken into account. In the case of `Panel`, these parameters can be obtained by inspecting the matrix structure, without even doing the factorization for real. This eases the modeling of such kernels since such information can be used to define the region in which such parameters evolve and to design an informed experiment design to characterize the performance of the machine.

We repeated similar analyses for other kernels and discovered that `Update` had a similar structure. The other three kernels (`Do_subtree`, `Activate` and `Assemble`) show different behavior and the explaining parameters were more difficult to identify. In the following we describe our modeling choices for each kernel, detailing how and why particular parameters proved to be crucial.

7.4.1 Simple negligible kernels

As illustrated by Figure 7.4 (top-right histogram), the `Deactivate` kernel has a very simple duration distribution. We remind that this kernel is responsible solely for deallocating the memory at the end of the matrix block factorization. It is thus not surprising that these tasks are very short (a few milliseconds) and are negligible compared to the other kernels. Furthermore, there are only a few instances of such tasks even for large matrices and the cumulative duration of the `Deactivate` kernels is thus generally less than 1% of the overall application duration (makespan). Therefore, we decided to simply ignore this kernel in the simulation, injecting zero delay whenever it occurs. So far, this simplification has not endangered the accuracy of our simulation tool.

7.4.2 Parameter dependent kernels

Certain `qr_mumps` kernels (`Panel` and `Update`) are mostly wrappers of LAPACK/BLAS routines in which the vast majority of the total execution time is spent. Their duration depends essentially on their input arguments, which define the geometry of the submatrix on which the routines work. Although these routines execute very different kind of operations, they can be modeled in a similar way.

`Panel` The duration of the `Panel` kernel mostly depends on the geometry of the data block which it operates upon, i.e., on MB (height of the block), NB (width of the block) and BK (number of rows that should be skipped). This kernel simply encapsulates the standard `dgeqrt` LAPACK subroutine that performs the QR factorization of a dense matrix of size $m \times n$ with $m = MB - (BK - 1) \times NB$ and $n = NB$. Therefore, its *a priori* complexity is:

$$T_{\texttt{Panel}} = a + 2b(NB^2 \times MB) - 2c(NB^3 \times BK) + \frac{4d}{3} NB^3,$$

where a, b, c and d are machine and memory hierarchy dependent constant coefficients. Note that formula above matches the theoretical complexity of the operation.

Such linear combination of parameter products fits the linear modeling framework and the summary of the corresponding linear regression is given in Table 7.3. For each parameter combination in the first column (NB^3, $NB^2 \times MB$, and $NB^3 \times BK$), an estimation of the corresponding coefficient is provided along with the 95% confidence interval. These values correspond to the a,

Table 7.3: Linear Regression of `Panel` kernel.

	`Panel` Duration	
NB^3	1.50×10^{-5} (1.30×10^{-5}, 1.70×10^{-5})	***
$NB^2 * MB$	5.49×10^{-7} (5.46×10^{-7}, 5.51×10^{-7})	***
$NB^3 * BK$	-5.52×10^{-7} (-5.57×10^{-7}, -5.48×10^{-7})	***
Constant	-2.49×10^{1} (-2.83×10^{1}, -2.14×10^{1})	***
Observations	493	
R^2	0.999	
Note:	*p<0.1; **p<0.05; ***p<0.01	

$2b$, $2c$ and $\frac{4d}{3}$ from the previous formula. The standard errors are always at least one order of magnitude lower than the corresponding estimated values, which means that the coefficient estimates is quite good. Furthermore, the three stars for each parameter in the last column indicate that the estimates of the coefficients are all significantly different from 0, which means that these parameters are significant. This hints that the model is minimal and that we can not simplify it further by removing parameters without damaging its precision. Finally, the most important indicator is the adjusted R^2 value. The coefficient of determination, denoted R^2, indicates how well data fit a statistical model. This coefficient ranges from 0 to 1. An R^2 of 0 indicates that the model explains none of the variability of the response data around its mean while an R^2 of 1 indicates that the regression line perfectly fits the data. In our case R^2 is extremely good, as it is almost 1, which indicates that our model has a very good predictive power.

We also checked the linear model hypothesis by analyzing the corresponding standard plots provided by the statistical language R and which are displayed in Figure 7.6. The first one indicates that the residuals are indeed unstructured and homoscedastic. The second one allows for checking the normality assumption. Although it does not hold perfectly in our case, it is known to not harm the quality of the regression and of the model. Finally, the third one allows for checking through a sequence plot that the residuals are not structured along time and that there has been for example no temporal perturbation.

Note that although the results presented in this section only concern a single `qr_mumps` factorization of *e18* matrix, they are perfectly general. For all the other matrices and experiments we conducted, the regression is always just as good with very high R^2 value (above 0.98).

`Update` The duration of the `Update` kernel also depends on the geometry of the data upon which it operates, defined by the same MB, NB, and BK parameters. This kernels simply wraps the LAPACK `dgemqrt` routine which applies k Householder reflections of size $m, m-1, ..., m-k+1$ on a matrix of size $m \times n$ where m, n and k are equal to $MB - (BK-1) \times NB$, NB and NB, respectively. Therefore, its *a priori* complexity is defined as:

$$T_{\texttt{Update}} = a' + 4b'(NB^2 \times MB) - 4c'(NB^3 \times BK) + 3d'NB^3$$

The same approach can thus be used and the R regression summary is provided in Table 7.4. The coefficient estimates are obviously different from the ones of the `Panel` kernel since the nature of the two kernels is different. The most notable difference is certainly the NB^3 coefficient whose influence cannot be accurately estimated and may thus appear as insignificant. However, this can be explained by the fact that for this particular matrix, the parameter range of BK is limited, which leads to a confounding of the effects of NB^3 with the ones of $NB^3 \times BK$.

It is also interesting to note that this model is based on a much larger set of observations, which is expected since one `Panel` is followed by many `Update`s. Finally, the R^2 value reported in Table 7.4 is again very close to 1, which shows the excellent predictive power of this simple model.

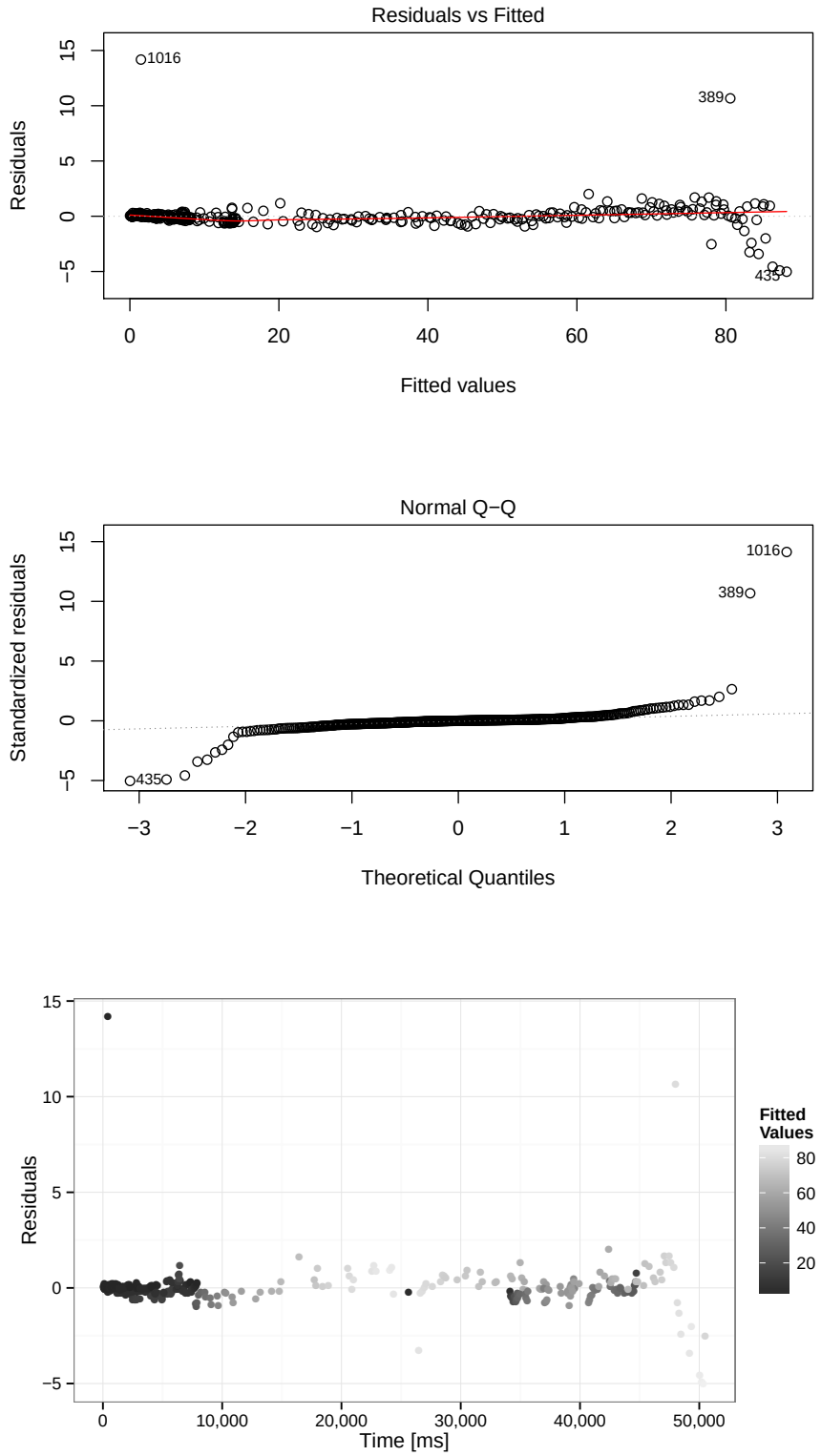


Figure 7.6: Analysing linear model for `Panel` kernel.

Table 7.4: Linear Regression of `Update` kernel.

	`Update` Duration	
NB^3	1.59×10^{-9} $(-6.93 \times 10^{-8}, 7.25 \times 10^{-8})$	
$NB^2 * MB$	4.37×10^{-7} $(4.36 \times 10^{-7}, 4.37 \times 10^{-7})$	***
$NB^3 * BK$	-4.37×10^{-7} $(-4.38 \times 10^{-7}, -4.36 \times 10^{-7})$	***
Constant	8.33×10^{-1} $(7.12 \times 10^{-1}, 9.54 \times 10^{-1})$	***
Observations	20,893	
R^2	0.998	
Note:	*p<0.1; **p<0.05; ***p<0.01	

```
/* Injecting panel time */
static inline double xbt_panel_time(double MB, double NB, double BK)
{
  // Computed from matrix: e18.mtx
  // Adjusted R-squared: 0.999
  return  -24.89 + (NB*NB*NB)*(1.50e-05) +
    (NB*NB)*MB*(5.49e-07) + (NB*NB*NB)*BK*(-5.52e-07);
}
/* Injecting update time */
static inline double xbt_update_time(double MB, double NB, double BK)
{
  // Computed from matrix:  e18.mtx
  // Adjusted R-squared:  0.999
  return  0.83 + (NB*NB*NB)*(1.59e-09) +
    (NB*NB)*MB*(4.37e-07) + (NB*NB*NB)*BK*(-4.37e-07);
}
```

Figure 7.7: Automatically generated code for computing the duration of `Panel` and `Update` kernels.

Simulation

From the previous R linear regressions (Table 7.3 and 7.4), we automatically generate C code for the simulation of these kernels (see Figure 7.7) and link it to SimGrid. When simulating `qr_mumps`, whenever `Panel` or `Update` is called, their parameters are given as an input to these functions. The calling thread is then blocked during the corresponding duration estimation, thereby increasing the simulation time without actually executing the tasks.

It is important to understand that these two kernels are the most critical ones regarding overall simulation accuracy and that the precision of their estimation greatly influences both the makespan and the dynamic scheduling.

7.4.3 Matrix dependent kernels

Modeling kernels based on their signature is quite natural but it is unfortunately not applicable to all kernels. Some of them are more than simple calls to regular LAPACK/BLAS subroutines. These tasks execute a sequence of operations that depends on the matrix structure as well as on the organization of the previously executed tasks. Therefore, such kernels cannot be modeled simply from their input parameters. The actual amount of work performed by the task can however be estimated by taking into account the size of the submatrix and its internal structure. Such complexity is required as large parts of the matrix blocks are filled with zeros and are thus skipped. Three kernels require a specific expertise on the multifrontal QR method and on the `qr_mumps` code to provide such workload estimates.

- **`Do_subtree`**: Both an estimation of the number of floating point operations it needs to perform and the number of nodes it has to manage are required to model the duration of this kernel.

- **`Activate`**: The duration of this kernel is mainly governed by two factors: the number of coefficients that have to be set to zero and the number of non-zero coefficients it has to assemble from children nodes.
- **`Assemble`**: The complexity of this kernel is directly linked to the total number of non-zero coefficients that needs to be copied to the parent node. However, such memory intensive operation is more subject to variability than the other computing kernels.

Analyzing the execution of these kernels and constructing models can then be performed as in Subsection 7.4.2 using the R language and simple linear regressions. Table 7.5 presents a summary of the prediction quality for each kernel as well as the minimal number of parameters that have to be taken into account. For all of them, the adjusted R^2 is close to 1, which indicates an excellent predictive power.

Table 7.5: Summary of the modeling of each kernel based on the *e18* matrix on Fourmi (see Section 7.3 for more details).

	`Panel`	`Update`	`Do_subtree`	`Activate`	`Assemble`
1.	*NB*	*NB*	#FLOPS	#Zeros	#Coeff
2.	*MB*	*MB*	#Nodes	#Assemble	/
3.	*BK*	*BK*	/	/	/
R^2	0.99	0.99	0.99	0.99	0.86

7.4.4 Accounting for kernels variability

Due to the specificity of the kernel models, we did not perform a histogram study for sparse kernels similar to the one we did for dense linear algebra kernels presented in Section 6.2. During the whole sparse application run there are typically only a few kernel executions with exactly the same values for each parameter. Thus, it makes no sense from a statistical point of view to create histograms for each one of them, as they would be based only on few observations. However, we strongly believe that the simulation results presented in the following sections are not significantly biased by the fact that the kernel with specific parameters is always replaced with the same constant computed from the model.

One should still add a supplementary variability to the injected values, based on the residuals of the linear model. In other words, to each value computed from the formulas presented in Figure 7.7 a random value that sampled from the residuals could be added. Later, by modifying the seed of the random generator, researchers could execute large number of experiments that will all have a slightly different kernel timings and consequently different scheduling and overall makespan. Such studies are very interesting for testing the robustness of the scheduler. Although we have not yet performed such research due to lack of time, implementing such extension to our current solution would be straightforward.

7.5 Evaluation methodology

The execution time of a single kernel on a certain machine greatly depends on the machine characteristics (namely CPU frequency, memory hierarchy, compiler optimization, etc.). Obtaining accurate timing is thus a critical step of the modeling. To predict the performance of the factorization of a set of matrices on a given experimental platform, we need to first benchmark the kernels identified in the previous section.

For the kernels that have clear dependency on the matrix geometry (`Panel` and `Update`), we wrote simple sequential benchmarking scripts, that pseudo-randomly choose different parameter values, allocate the corresponding matrix and finally run the kernel, capturing its execution time. However, for kernels `Do_subtree`, `Activate` and `Assemble` whose code is much more complex and

depends on many factors, including even dependencies on previously executed tasks, creating a simplistic artificial program that would mimic such a sophisticated code is very difficult.

Indeed, since each sparse matrix has a unique structure, the corresponding DAG is very different and the kernel parameters (such as height and width) greatly vary from one matrix factorization to another. For example, the `qr_mumps` factorization of a certain matrix may execute a very large number of "small" `Do_subtree` kernels (each with a small amount of work), while some others matrices (, such as *e18* as shown in Figure 7.2), have much fewer instances of this kernel, but with a bigger workload. Consequently, it is very hard to construct a single linear model that is appropriate for both use cases. The inaccuracies caused by such model imperfection can produce either underestimation or overestimation of the kernel duration and thus of the whole application makespan as well. Therefore, to benchmark such kernels we rely on traces generated by a real `qr_mumps` execution (possibly on different matrices as the ones that need to be studied).

The result of this benchmark is analyzed with R to obtain linear models that are then provided to the simulation. At each step of the regression, we control that the models are adequate through a careful inspection of the regression summaries and of the residual plots. These models are then linked with the simulator and the experimental platform is then of no longer use as `qr_mumps` can then be run in simulation mode on a commodity laptop. Using a recent and more powerful machine only improves the simulation speed and possibly allows for running several simulations in parallel. In such simulations, we recall that the code of `qr_mumps` and of StarPU is run for real (the application and the runtime are emulated) but all computation intensive and memory consuming operations are faked and converted into simple simulation delays. SimGrid is solely used for managing the simulated time and the synchronization between the different threads.

To evaluate the validity of our approach, we need to compare real execution outcomes with simulations outcomes. Therefore, we execute `qr_mumps` for the different matrices and collect not only the execution time but also an execution trace with information for each kernel as well as when memory is allocated and deallocated. When running in simulation mode, due to the structure of our integration, we can collect a trace of the same nature and thus compare the real execution to the simulation in details.

Finally, our experimentation workflow follows the same four steps as for the dense linear algebra application study:

1. We run once a designed calibration campaign on the target machine that spans the desired parameter space corresponding to the different matrices.

2. We analyze the benchmarking outputs and execution traces, fitting the observations into linear models for each kernel. We add such models to SimGrid.

3. We run simulations on a commodity machine.

4. We validate the simulation accuracy by comparing makespans, traces and memory consumption with the native executions.

Following the principles and the Git/Org-mode workflow presented in Chapter 4, all the results that are given in this document are also available online [SSW] for further inspection. We also provide on Github an example of an easy to access pretty-printed report on the trace [Trab] (similar to the one in Appendix B.2) as well as information on how the trace was captured, as a part of the joint trace collection project [Traa].

We remind that this repository additionally contains source code of `qr_mumps`, StarPU and SimGrid along with all the scripts for running the experiments, the calibrations and conducting the analysis, making our work as reproducible as possible. Supplementary data (e.g., produced by "unsuccessful" experiments and that can be very informative to the reader) can also be found at the same location.

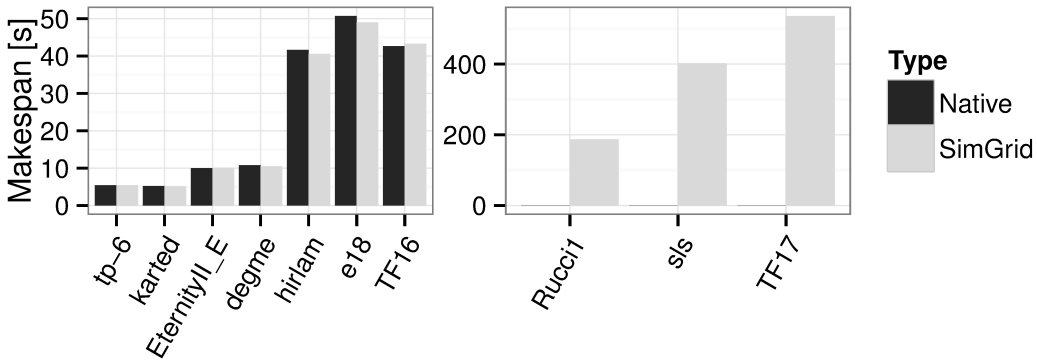


Figure 7.8: Makespans on the 8 CPU cores Fourmi machine for 10 different matrices. Native results on 3 largest matrices are not presented, because they are too long, since the factorization exceeds RAM memory capacities of the Fourmi machine.

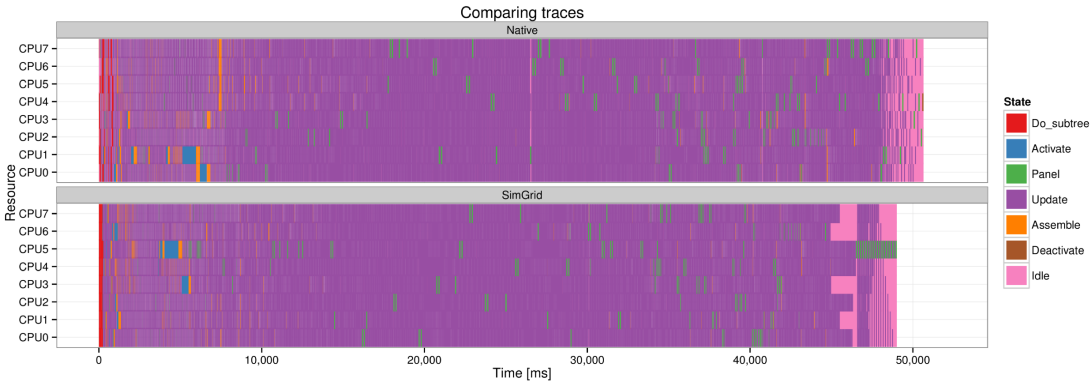


Figure 7.9: Gantt chart comparison on the 8 CPU cores Fourmi machine.

7.6 Simulation quality evaluation

7.6.1 Evaluation on the Fourmi machine

Figure 7.8 depicts the overall execution time of `qr_mumps` when factorizing the different matrices of Table 7.2. The SimGrid predictions are very accurate, as they are never bigger than 3%. It should be noted that our predictions are systematically slight underestimations of the actual execution time as our coarse-grain approach ignores the runtime overhead and a few cache effects.

Still, focusing solely on a single number at the end of the execution hides all the details about the operations performed during the execution. Therefore, we also investigated the whole scheduling in details, comparing Native to SimGrid execution traces. An example of such investigation for the *e18* matrix (it is the one exhibiting the largest difference between Native and SimGrid makespans) is shown in Figure 7.9. To make the Gantt charts as readable as possible, we retain only the modeled kernels and idle state, filtering overlapping states related to the runtime control.

`qr_mumps` starts by executing many `Do_subtree` kernels and executes all the remaining ones soon after. Most of the time is spent running `Update` operations while the `Panel` operations are executed regularly. Towards the end, there are fewer and fewer tasks with more and more dependencies between them, and many cores have thus to remain idle. The Native and SimGrid traces are extremely close. A noticeable difference can be seen at the very beginning as in the simulation all

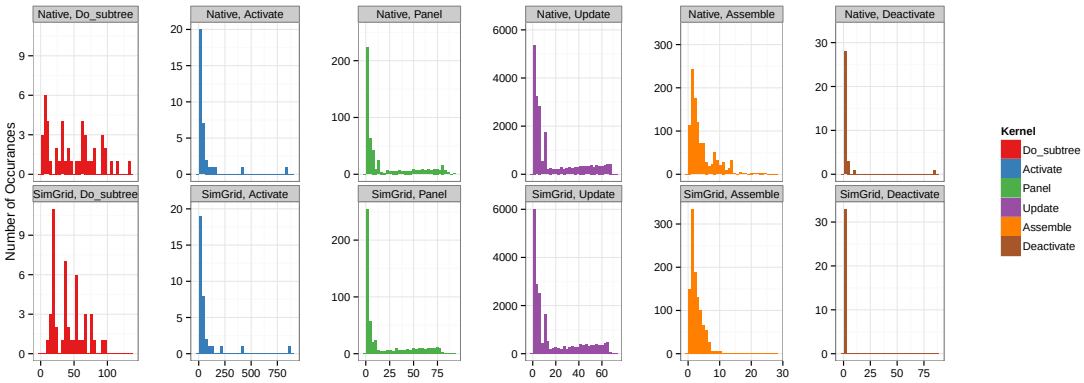


Figure 7.10: Comparing kernel distribution duration on 8 CPU cores Fourmi machine.

workers start exactly at time 0, they all pick only `Do_subtree` tasks that are the leaves of the DAG and thus the first ready tasks. Although the real execution tends to run in the same manner, it is not so strict and several workers pick `Activate` tasks available right after the first `Do_subtree` terminates. Another small discrepancy between the two traces can be noticed at the end of the execution where the idle time is distributed in a quite different way. This is however related to the difference of scheduling decisions taken by the runtime. Indeed, we remind that the StarPU runtime schedules tasks dynamically and thus even two consecutive Native executions can lead to quite different execution structures even if their total duration is generally similar. Idle time distributions similar to the one of the SimGrid trace can also be observed in real executions.

However, the gantt charts of such densely packed traces of dynamic schedules can sometimes be misleading. The first issue comes from the fact that there is a huge number of very small states that have to be aggregated during the graphical representation which is limited by the screen or printer resolution. Many valuable information can be lost in such process and the result may be biased [MSL13]. Second, it is very hard to quantify the resemblance of two traces corresponding to dynamic schedules, as even when the task graph is fixed, the tasks will naturally have very different starting times from an execution to another. Therefore, we compared different and more controlled aggregates.

For example, Figure 7.10 compares the distributions of the duration of each kernel for a real execution and a simulation (we use the same two traces of the *e18* matrix that were used earlier). The upper row presents kernel distributions from the Native trace, while in the bottom row are the ones predicted by the SimGrid simulation. The modeling technique described in Section 7.4 proves very satisfactory, since distributions match quite accurately. The only kernel for which we can observe a slightly larger discrepancy is `Assemble`, which was indeed very hard to model and had the worst R^2 value. However, in practice this kernel is rarely on the critical path of the `qr_mumps` execution and is often overlapped by other kernels. Additionally, the overall duration of all `Assemble` tasks is relatively small compared to others and thus such inaccuracies of the model do not greatly affect the final simulation prediction.

Studying distribution applies a temporal aggregation and discards any notion of time such as when a specific task was executed. This can hide interesting facts about certain events or a particular group of tasks that occurred in a distinct period of time during the whole application run. Figure 7.11 tracks the execution of the `Panel` kernel and indicates the duration of the tasks at each time. To ease the correspondence between the real execution and the simulation, the color of each point is related to the task id of the task. Both colors and the pattern of the points suggest that the traces match quite well. Even though the scheduling is not exactly the same, it is still very close. Similar analysis have been performed for all the other kernels as well and the results were very much alike.

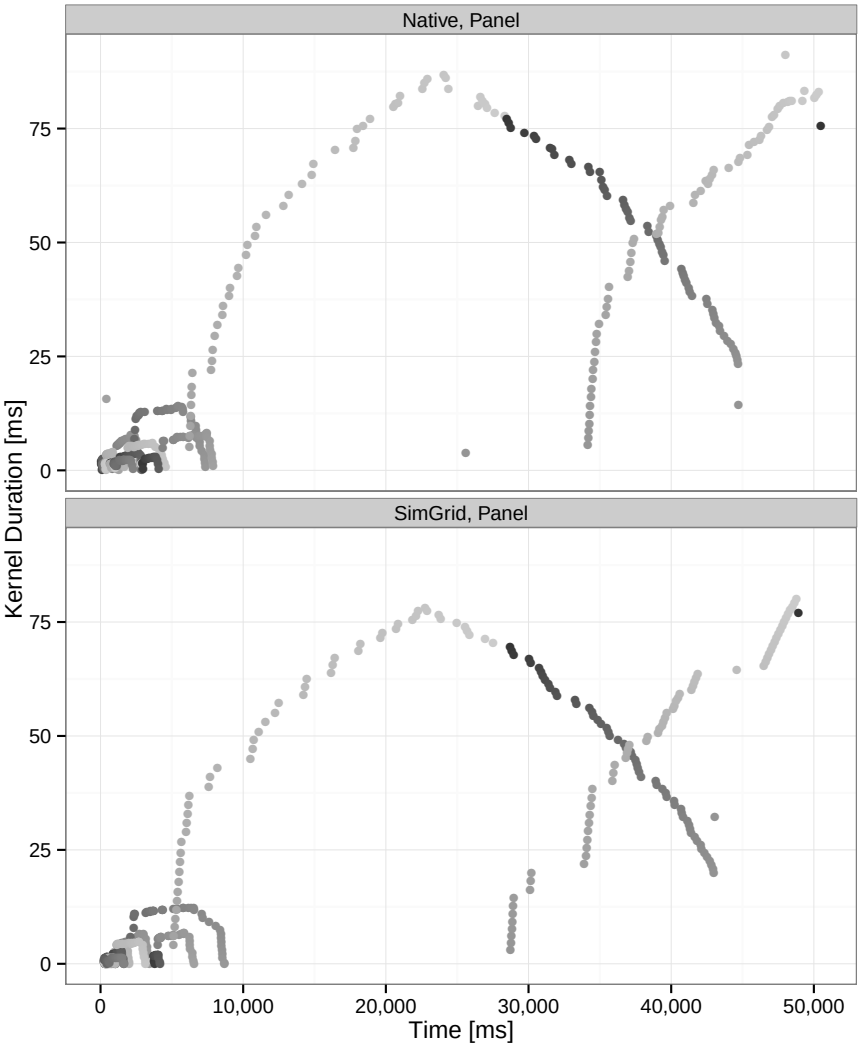


Figure 7.11: Comparing `Panel` as a time sequence on 8 CPU cores Fourmi machine. Color is related to the task id.

7.6.2 Evaluation on the Riri machine

To validate our approach, we have experimented on a different architecture. The results are presented in Figure 7.12 and show a more important error of the SimGrid predictions, that is now averaging 8.5%. Such inaccuracy mostly comes from the fact that Riri has a specific architecture, where the 10 cores used for the experiments are all share the same L3 cache. The pressure on the cache produced by all the workers executing kernels in parallel decrease the overall performance, which is not correctly captured by our models. Still, the SimGrid predictions stay reasonably close to the Native ones and can thus be very useful to users and developers.

One step further in our experimental campaign was to compare executions on the full machine, using all 40 cores. As expected, the largest prediction error doubled (Figure 7.12), but the results can still be considered as good since all the tendencies are well captured. In particular, non trivial results can be obtained such as the fact that the *TF17* matrix benefits much more from using several nodes than the sls matrix.

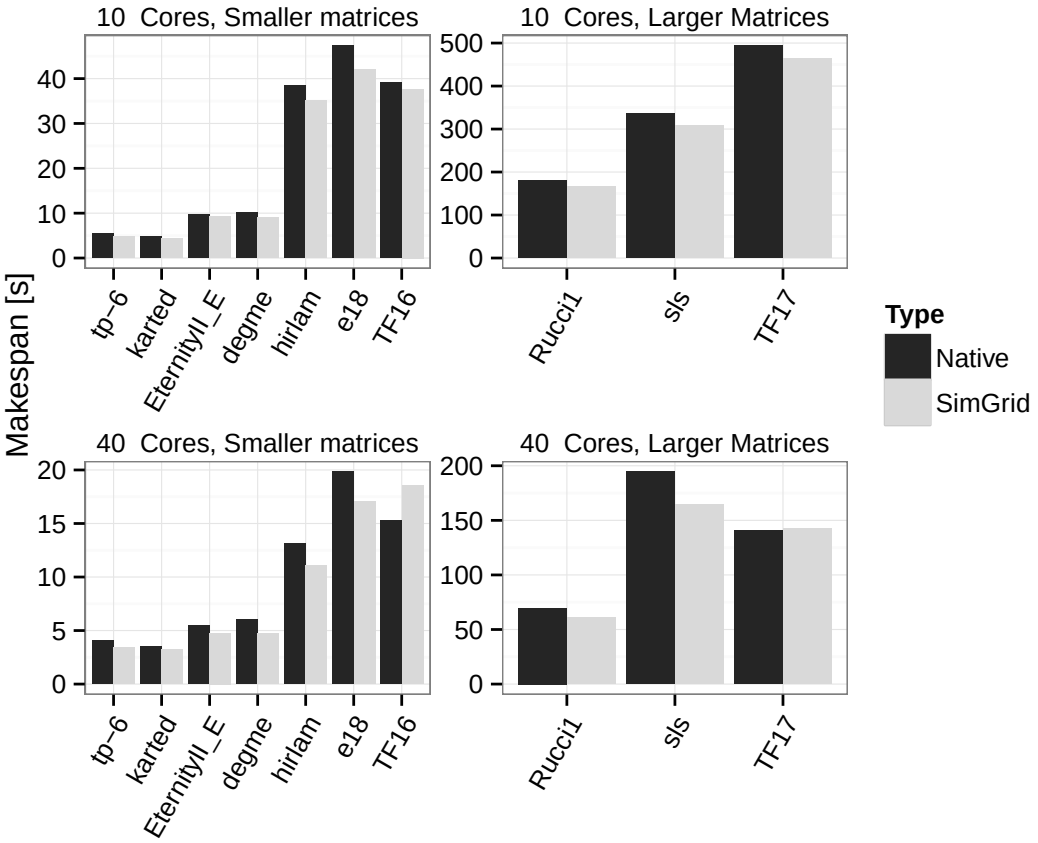


Figure 7.12: Results on the Riri machine using 10 or 40 CPU cores. When using a single node (10 cores), the results match relatively well although not as well as for the Fourmi machine due to a more complex and packed processor architecture. When using 4 nodes (40 cores), the results are still within a reasonnable bound despite the NUMA effects.

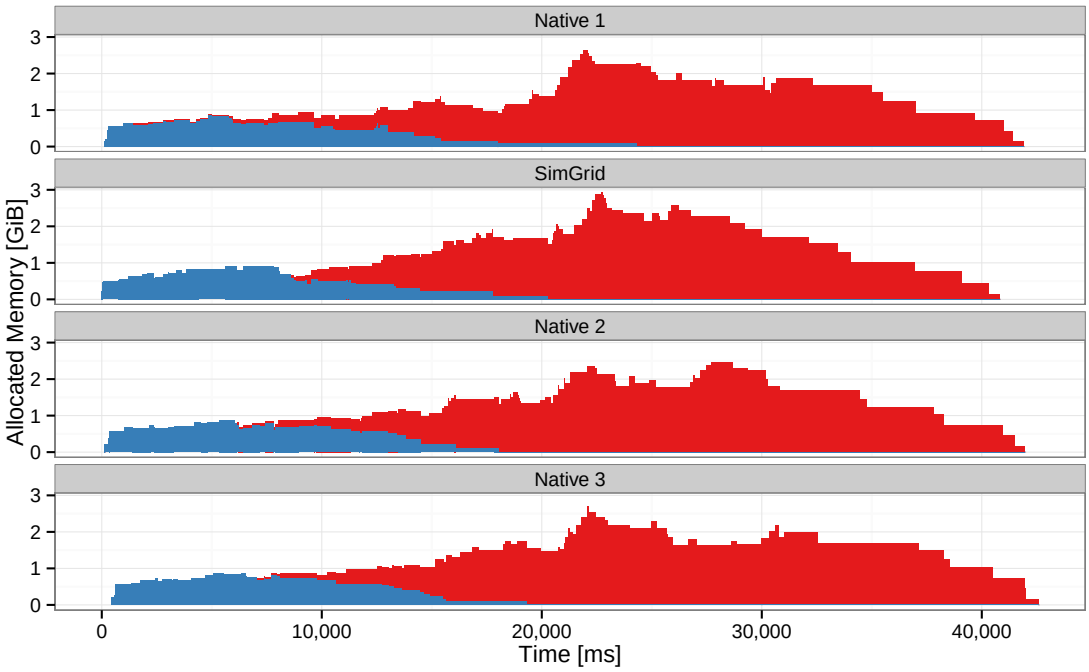


Figure 7.13: Memory consumption evolution. The blue and red parts correspond to the `Do_subtree` and `Activate` contribution.

7.7 Typical studies enabled by such approach

7.7.1 Memory consumption

Working on several parts of the elimination tree in parallel provides more scheduling opportunities, which improves processor occupancy. But it also increases memory requirements, which can have a very negative influence on performance. The most critical criteria regarding memory is the peak memory consumption of the application. A single wrong scheduling decision can dramatically increase it and potentially result in memory swapping between the RAM and the disk.

Since the amount of work that can be executed in parallel is generally limited and very matrix dependent, finding the right trade-off between memory consumption and the efficient use of the whole set of available cores is crucial for obtaining the best performance [MSV13]. To evaluate a new factorization algorithm or a different scheduling strategy, one thus has to perform a large number of costly experiments on various matrices. Using simulation can greatly reduce the cost of such study as it does not require the access to the actual experimental machines, often shared between many users. It can be performed much faster and several simulations with different parameters can even be run in parallel. In our solution no actual memory is neither allocated nor deallocated for the data, as the corresponding *malloc* calls are only simulated by SimGrid. However, the size for the required array allocation and deallocation is still traced. This allows for reconstructing memory usage, providing the memory peak prediction of the simulation. Since SimGrid faithfully represents the runtime execution (as it was presented in the previous subsections), its execution will go through the DAG in a very similar way to the Native run. Therefore, the memory peak predicted by SimGrid will be very close to the one observed in Native experiments.

Beyond the memory peak, it is also interesting to study the evolution of memory consumption throughout the execution. Figure 7.13 shows the evolution of the total amount of memory allocated by `qr_mumps` when factorizing the *hirlam* matrix for three Native executions and for a simulation. The three Native executions correspond to three consecutive runs performed with exactly the same source code and environment. Such analysis allows for identifying where the scheduling was not

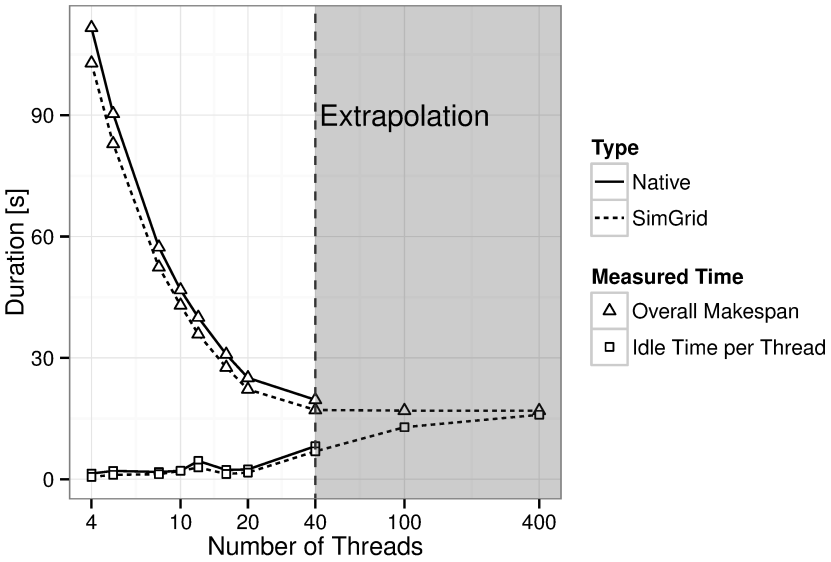


Figure 7.14: Extrapolating results for *e18* matrix on 100 and 400 CPU cores.

optimal and what parts of the application should be improved to increase the overall performance. Although the memory consumption evolution is very similar between different experiments it is far from being identical since runtime scheduling decisions are made dynamically. The evolution predicted by SimGrid is remarkably difficult to distinguish from the three other ones, which shows that our approach allows for faithfully predicting memory consumption during the application execution.

7.7.2 Extrapolation

When sufficient care is taken on benchmarking and kernel profiling, we believe that our approach allows for performing faithful performance prediction of the execution on large computer platforms. Based on models obtained from a single CPU calibration, we can extrapolate the simulation on a larger numbers of cores. These simulation results are certainly not as accurate as the ones presented in Section 7.6, but can still show general trends. New phenomenon that haven't been observed yet may occur at large scale but the simulation somehow allows for obtaining an "optimistic" performance prediction that will be achieved "if nothing goes wrong". Researchers can thus observe how their matrix factorization would perform in an ideal context. Since certain parts tend to be generally underestimated (e.g., data fetching and contention) in the simulations, SimGrid results provide theoretical performance bound, above which application could not pass on the target machine.

Figure 7.14 shows the performance obtained when factorizing the *e18* matrix with different numbers of cores. Together with the overall makespan, we indicate how much time in average each core spends idle. With a small number of workers, each thread has enough work as `qr_mumps` is well parallelized. However, the execution is limited by the critical path in the DAG of tasks and thus above a given platform size, most of the cores remain idle, waiting for task dependencies to be satisfied. Increasing the number of threads beyond this point does not improve the overall performance, but only decreases the efficiency of the workers. After comparing our simulation results to the Native execution for up to 40 cores on the riri machine, we decided to use the same kernel models and investigate what performance could be expected from a larger machine comprising the same kind of nodes. The simulation results actually predict that, for this matrix, the makespan will not improve any further and that most cores will be idle. Investigating more

in detail the trace simulated for 100 cores would allow to know whether the critical path is hit or if further improvements can still be expected with a better scheduling.

Chapter 8

Conclusion and Future Work

In this thesis we focused on two important aspects in the domain of methodology and performance evaluation for conducting empirical studies of dynamic HPC applications. In the following chapter, we conclude on these two aspects separately. We start by describing the advantages and the drawbacks of our methodology for conducting reproducible research. Then, we recapitulate the achieved results and present the current limitations of our simulation of dynamic task-based runtime systems. Finally, we detail several possible research directions opened by the various studies we conducted for this thesis.

8.1 Methodology for conducting reproducible research

We believe that the methodology we applied throughout three years of this thesis is a good example of how studies in our field should be performed. Before each measurement, we automatically log information about the machine, such as the current CPU frequency and governor, the memory hierarchy, the versions of Linux and gcc, etc., (see for example an excerpt of the script and the output file in Appendix B.1 and B.2). We also systematically and automatically recompile software before using it to conduct experiments, and keep all the configuration and compilation outputs. Additionally, we enforce that all changes to the source code are committed in the revision control system and we keep track of the hash of the Git/SVN version of the source code. We even went one step further in using versioning systems, as we stored all the experiment data in the same repository as the source code of our project. This enables provenance tracking, i.e, to bind the results that were obtained with the corresponding code, so that they can be later easily investigated, compared or reproduced. Finally, we save the measurement results (makespan, GFLOPS rate) together with execution traces. Since during the development period, the workflow and the data format can go through several changes and adjustments, we used a laboratory notebook to keep track of the most important modifications.

Only by applying such approach, we were able to manage more than 10,000 experiments and 40GiB of data. All these experiment results carry a valuable information. Some of them are good results that were later published, others are unsuccessful, bogus experiments that are equally important as the reason behind their failure provides knowledge about the studied system. Therefore, it is an imperative to keep all these experiment results well organized and easy to access as well as to keep notes about every experimental campaign, for which we once again used a laboratory notebook.

In our research, we were not only concerned about the reproducibility of our experiments, but also about the replicability of the analysis we were performing on the collected data. Hence, all the papers we have written, **including this thesis**, combine within a single plain text file the body of the article along with the complete analysis of the data, all of which are later exported into a standard pdf document (see Appendix B.6). It has a hierarchical structure, with different types of code, including Shell (to manipulate data files), R (for plotting figures) and LaTeX (to finely control formatting details). We made all raw data and traces publicly available in the Git

repository of the project on [SSW]. To replicate the article, all these data should first be cloned and unpacked. Then, raw data files that contain many additional metadata are parsed and filtered to extract all useful information into csv files. Finally, R is used to load, process and plot data, ensuring that all figures are consistent throughout the whole article. This way, our experimental and analysis results can be inspected, referenced, or even reused in any other research project.

Our methods drove a great attention in the community, especially among young researchers, and we were asked to present it on numerous occasions. This had an impact on many researchers, who later applied some of our techniques and methodology to their daily workflows, which greatly improved clarity, exploitability and mostly reproducibility of their work. One of the examples is the current collaboration between researchers from Grenoble and Bordeaux for sharing execution traces [Traa].

Finally, since our solution is based on simple, widely known tools, even people from other fields could apply it. As an illustration, our publication in the SIGOPS Operating Systems Review [1]) raised the interest of a researcher who wants to apply our workflow on his studies on East-Asian languages.

Current limitations and future work

The main drawback of our approach is that it has many not so common conventions along with a steep learning curve, hence it is difficult for new users. Moreover, it requires an expertise in Org-mode, preferably using Emacs text editor, together with a good understanding of Git. We acknowledge that some researchers are more used to other editors such as Vi/Vim and will not switch them easily. Although it is still possible to use them in our context, as Org-mode is a plain text file that can be edited anywhere, it would be much harder to benefit from many of its special features. We believe that the tools we used provide benefits that are worth investing time but we also understand the need to simplify its use. There are thus currently many initiatives to port Org-mode to make it work completely in Vi or in web browsers. Some of them already work, but are not fully mature or complete yet. We are thus quite confident that Org-mode will be completely Emacs independent in the near future.

There is also a problem regarding the management and storing of large data files in repositories, and which is well-known to the community. This has been already solved for the Mercurial revision control tool, but even after an exhaustive research we could not find a satisfactory solution for Git. Many tools have been proposed, e.g., `git-annex`, but they all have their shortcomings. Such tools are generally meant to be alternatives to synchronization services like Dropbox and Google Drive rather than to help dealing with large data traces originating from remote machine experiments. Having large Git repositories of several GiB does not hinder daily committing, but can significantly slow down pull and checkout operations of branches comprising a huge number of data sets (typically the *data* and *art#* branches).

It is also still unclear how this approach would scale to a large number of users working simultaneously, doing code modifications and experiments in parallel. In theory, it should work if everyone has sufficient experience of the tools and workflow, but we have never tried it with more than few persons. Another interesting feature that we have not experienced yet is collaboration with external users. These researchers could clone our project, work on it on their own, try to reproduce the results and build upon our work, potentially improving the code and contribute data sets back. Even though such utilization should work smoothly, there could be some pitfalls that we have not anticipated yet.

One could also ask the question of whether providing so much information is of any interest as too much information may make the most important things harder to distinguish. Regardless of the answer to this question, we believe anyway that beyond the actual experimental content of our open laboratory notebook, its structure and the techniques we used to keep track of information or to make analysis could be useful to others.

In the near future, we plan to finish implementing simple scripts that will completely automate our workflow. These scripts will be packaged and available on the debian Linux system, in the same way as the `git-flow` approach for software development, only this time for managing experimental

research.

8.2 Simulating dynamic HPC applications

In this thesis, we have explained how to model and simulate a task-based runtime system running on hybrid multi-core architecture comprising several GPUs. Unlike fine-grain GPU simulators that have been proposed in the past and which focus on architectural details of GPUs, our coarse-grain approach relies on SimGrid and allows for accurately predicting the actual running time and to perform extremely quickly extensive simulation campaigns to study various alternatives. We demonstrated the precision of our simulations using the critical method, i.e., by testing our models and by conducting as much experiments as possible in a large variety of settings (two standard dense linear algebra applications, seven different generations of GPUs, several scheduling algorithms) until we find a situation where our simulation fails at producing a good prediction, in which case we fixed our modeling. Such a tool is extremely interesting for both StarPU developers and users as it allows (i) for easily and accurately evaluating the impact of various parameters or scheduling alternatives (ii) for tuning and debugging applications on a commodity laptop (instead of requiring a dedicated access to a high-end machine) in a reproducible way (iii) for obtaining reliable performance estimations that may allow for detecting problems with some real experiments (perturbation, configuration issue, etc.).

It is important to mention that the time to run each simulation of these applications is much shorter than the one needed to conduct a real experiment. Compared to architecture-level simulators whose average slowdown of simulations versus native execution is of the order of magnitude of several dozens of thousands, our coarse-grain simulation allows for obtaining a speedup of ten to a hundred depending on the workload and on the speed of the machine. Additionally, since the target system is not required anymore, it is easy to run series of simulations in parallel.

Furthermore, we extended this work on dynamic dense linear algebra applications, by considering a sparse multifrontal linear algebra solver `qr_mumps`. Modeling the irregular internals of such application is much more challenging and required a careful study. We show through extensive experimental results that we manage to accurately predict both the performance and the memory usage of such applications. Once again, our proposal allows for quickly simulating such dynamic applications using only commodity hardware instead of expensive high-end machines. As an illustration, factorizing the *TF17* matrix on a 40 core machine requires 157s and 58GiB of RAM while simulating its execution on a laptop only takes 57s and 1.5GiB of RAM. Being able to quickly obtain performance and details of memory consumption of such applications on a commodity laptop is a very useful feature to the `qr_mumps` users and developers, as they can easily test the influence of various scheduling, parameters or even code modifications.

The SimGrid simulation mode is thus integrated in the latest versions of StarPU and of `qr_mumps`. Implementing such a faithful simulation tool was however not straightforward. Although the final solution contains relatively small number of code line changes (approximately a few hundreds) compared to the huge code base of SimGrid (106,350 lines) and StarPU (172,251 lines), programming them required much effort. Developing such a complex software requires a good understanding of paradigms and a good integration of the tools on all layers: application, runtime and simulation. This high complexity probably explains the few successful attempts in simulating runtimes. The main reason why our solution worked and provided such accurate predictions is a good choice of runtime and simulators, as both StarPU and SimGrid have a good, modular design of internals that allows for coupling with other tools.

The motivation for developing our solution came from the StarPU users and developers that had a great need for such a tool for their daily work. Therefore, we developed a stable tool that could be used by many researchers that want to evaluate the performance of their application, hence our solution quickly became much more than just a prototype. The results presented in this thesis are only a first step, as our approach can be easily extended to many other interesting studies.

8.2.1 Current limitations

The simulation tool we have crafted is coarse-grain and is thus based on certain runtime and hardware abstractions. Although our models are good enough for achieving faithful simulation predictions for the programs and machines we have studied so far, these models are still not perfect. There are several minor sources of inaccuracy of our solution, such as the fact that not all parts of the StarPU are simulated, there is the environment noise, the error of the kernel models, etc. These can be mitigated but hardly completely avoided, as the systems we are studying are very complex and their performance is non-deterministic. There are however other more fundamental shortcomings that we have neglected so far and that can possibly be accounted for with more advanced modeling.

Modeling memory distance

First, even before executing a kernel, the matrix block needs to be fetched in local memory. If data is not available from previous tasks, this requires explicit transfers from the main memory or from the GPUs, applying the communication models described in Section 5.4.

The problem however arises for the CPUs that share parts of the memory hierarchy, which is especially noticeable for NUMA machines. On such architectures, the data needed to execute a kernel can be stored either in local or in distant cache, or in the main memory. Therefore, the time to fetch the data is non-uniform. The StarPU runtime is based on a paradigm where CPU workers use shared memory for communication and there are thus no explicit data transfers. Consequently, during the whole application run, the exact same kernel with the exact same parameters will take varying time to be executed, since its input data will be located on different places. Additionally, a lot of data fetching from the distant caches could cause a memory bandwidth contention of the PCI bus, slowing down kernels even more. These effects could be mitigated through optimized scheduling and careful data distribution, but it is very hard (if not impossible) to completely discard such factors. Moreover, the larger machine, the more significant this effect becomes.

Without knowing the exact mapping of the memory throughout the whole application execution, the NUMA effect is very hard to model and integrate into the simulation. Therefore, many researchers working in this domain [SCW+02, CLB09, RCV+12] couple their application or runtime simulator to another cycle-accurate cache simulator, responsible for managing the memory hierarchy with all data and its distance. We believe that such approach suffers from many restrictions (in particular in terms of speed and scalability), and that much more coarse-grain methods should be applied. However, we have not yet found a satisfying alternative solution and we continue exploring different options. Still, for most of the applications we have studied so far, the aforementioned issue have a limited influence, thus we were able to completely ignore these memory related effects without harming the accuracy of the simulation predictions.

Modeling contention

In parallel applications, kernels are often slowed down by their neighbors that also execute certain operations in the same time. Combined with differing memory distance of the data, this introduces variability to the kernel durations. It is possible to account for such variability in the simulation considering that it as a random noise and a possible solution for dense linear algebra applications is presented in Section 6.2. However, better modeling these phenomena is much more challenging. The performance degradation due to contention occurs on different levels depending on the shared resource.

Multithreaded applications may be run with more than one thread executed per CPU core or GPU device. SimGrid supports such contentions through fair dividing of the resource capacity between the threads, similar to the bandwidth sharing in communication flows. However, we have not yet tested such feature for the processing units, as these applications introduce additional complexity and many new phenomena. Although this path undoubtedly carries many interesting research topics, in the scope of this work we have decided to focus solely on one thread per core executions.

When calibrating kernels on a single core and later using these values in the simulation, we assume that there is no interference between CPUs. However, when running parallel application, multiple kernels will be executed on cores that share some parts of the memory hierarchy. Consequently, these parallel kernels will compete for CPU caches, possibly evicting some data and slowing down each others. This is the main explanation why most of our simulation predictions tend to slightly underestimate the execution time of the kernels which produces shorter overall makespan.

Finally, there is another hypothesis in our models which implies that computation and communication on processing unit are independent and can be executed in parallel without damaging the performance of neither of those operations. Such assumption is not completely true, especially on certain generations of GPUs. However, measuring and injecting this degradation is not trivial, and since it would add only small benefit to the simulation accuracy, we decided to ignore it for the moment.

Simulation of sub-optimal native executions

Despite all these assumptions, our approach provides users with a sound baseline to compare with. The differences between Native and SimGrid can reveal not only modeling/simulation weaknesses but also application or machine issues. An example of such case is presented in Subsection 6.4.2, where it is demonstrated how a certain model of GPUs has a sub-optimal time of transfer when using a large pitch of the matrix as an input parameter.

We can refer to another illustrative example for `qr_mumps` application. When studying the matrix *cat_ ears_ 4_ 4* (also from the UF Sparse Matrix Collection), one can observe large discrepancy of native makespan and the one predicted by the simulation. When investigating the elimination tree in more details, it reveals that the matrix factorization is not optimally balanced between tasks (see Figure 8.1). There is a huge number of very small `Do_subtree` kernels (grey nodes on the Figure 8.1) and the elimination tree is unnecessary deep, which all makes the execution longer. Consequently, the range of kernel parameters during *cat_ ears_ 4_ 4* factorization are very different from the one of other matrices and thus the resulting models for such kernels are not completely appropriate, which damages simulation accuracy. However, before improving modeling one would probably want to fix the native execution for this matrix, generating a completely different (more balanced) DAG. Perhaps with such modification, no changes to the simulation would be needed after all.

There is another recurrent case where the native `qr_mumps` execution runs into problems which make its simulation inaccurate. For very large matrices, the factorization operation may require more memory than what is physically available on the system. As a result, part of the data is automatically swapped to disk by the virtual memory manager which causes disk thrashing. This produces tremendous decrease of overall performance because the access to data on disk is orders of magnitude slower than to the main memory. If paging is handled by the operating system and runtime has no control of it, the exact location of pages (memory or disk) is unknown during the simulation and SimGrid can thus not predict the time to fetch data. An alternative that could circumvent this issue is to use out-of-core implementations, which has already been implemented for the MUMPS solver [ALG06], but not yet on top of the StarPU runtime. Such an implementation could then benefit from the recently developed SimGrid's disk models.

Model universality

A key difficulty of our approach relies in obtaining of a general model for every kernel that can be applied to the simulation of any matrix. Since each sparse matrix presented in Table 7.2 has a unique structure, the DAG of tasks generated to solve it on a parallel machine also depends on these characteristics. Therefore, the parameters of the kernel (such as height and width) greatly vary from one matrix factorization to another. For example, the `qr_mumps` factorization of the *cat_ ears_ 4_ 4* executes a large number of "small" `Do_subtree` kernels (each with a small amount of work), while most others matrices, such as *e18* (see Figure 7.2), have much fewer instances

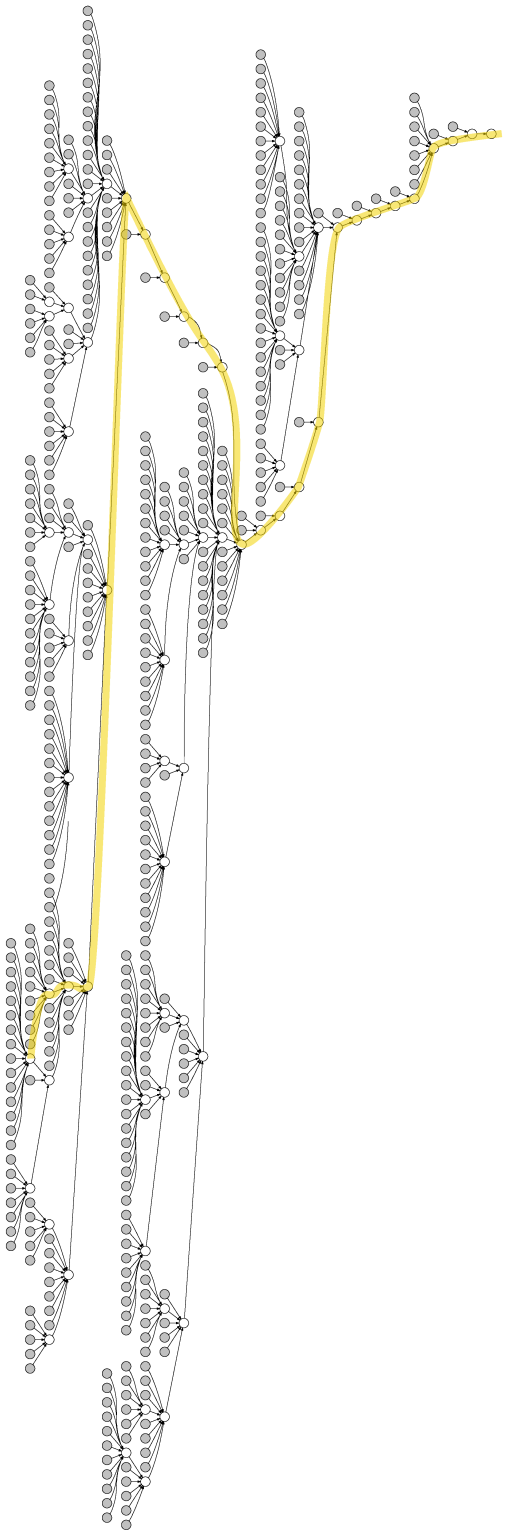


Figure 8.1: Elimination tree for *cat_ ears_ 4_ 4* matrix, rotated for 90 degrees to fit the page. The graph is extremely badly balanced and has a comb-like structure with a huge number of `Do_subtree` kernels presented as grey nodes.

of this kernel, but with a bigger internal workload. Consequently, it is very hard to construct a single linear model that is appropriate for both use cases. The inaccuracies caused by such model imperfection can produce either underestimation or overestimation of the kernel duration and thus of the whole application makespan as well.

The simplest solution for this problem would be to use piecewise linear models instead of simple linear ones. This should account for the particular parameter distribution and group the related observations. Constructing such models in practice however proved to be very challenging, as the observations are not following any natural law that divides them into groups and they are also greatly affected by external factors (kernels executed at the same time on the other cores, operating system, etc.). Thus, choosing where to put breakpoints to separate different segments is unclear. We could decide to completely rely on statistical tools that would help us find the best fitting piecewise linear model from the traces. However, such model would be very dependent on the measured values and thus not robust enough for a more general use. The decision where the breakpoints are made would have a huge influence on the simulation prediction.

8.2.2 Future work

Scaling to larger platforms

Our first goal in the future will be to upgrade the current solution so it allows for obtaining faithful simulation predictions on much larger platforms.

To this end, we plan to improve models for CPU cache contention and non-uniform memory access time. Indeed, when applications execute numerous computation kernels in parallel, many of them are slowed down compared to their optimal duration. If such decrease of performance is not correctly modeled, the error is propagated influencing the scheduling of the future kernels. This does not only produce overly optimistic simulation predictions of the overall application duration, but also generates unrealistic traces of the execution.

We believe that the problem on modeling contention on CPUs and CPU caches can be addressed by calibrating parallel kernel execution. One could analyze native execution traces and observe which combination of kernels is mostly run at the same time. Then, a parallel benchmarking program could be run, where kernels would be adequately slowed down by their neighbors. Models constructed from such calibration would represent better the native execution. However, implementing such approach is not straightforward, as choosing which kernels to run in parallel and with which parameters, depends greatly on application inputs (in the case of sparse linear algebra application the main input is the matrix). Therefore, constructing a single model for each kernel becomes even more complex. Additionally, for certain applications, such solution only alleviates the problem of running the application with different numbers of CPU cores, as changing the number of workers may also lead to different DAG of tasks with different characteristics. This is the case for `qr_mumps` where the depth of the elimination tree depends on the number of workers and thus two executions with different numbers of processing units will lead to the execution of the kernels with different sets of input parameters. Still, we believe that doing a more elaborate parallel kernel calibration would improve simulation accuracy and we plan to do it in the near future.

Similar to the shared caches, the problem of varying data distance on NUMA architectures could be addressed by an advanced calibration of parallel kernels. However, benchmarking NUMA machines is even more complex as it requires distributing data over nodes following the same patterns as in the native execution. This demands tracing memory banks used during the native run. Even if such a sophisticated calibration is carefully performed and accurate kernel models are derived from it, this approach would require mapping the whole memory hierarchy inside the simulation. Even though we believe this approach is possible and it could provide more accurate simulation predictions, implementing it is very challenging and deserves its own proper study. For now in our solution we ignored these memory related effects, as all aforementioned issues have a limited influence on the problems we have studied so far. However, this stays as a significant limitation of our approach if one wants to evaluate the performance of systems on a larger scale.

Another missing feature for predicting performance of large clusters is the ability to simulate distributed applications. Indeed, all our studies so far were within a single node (or using NUMA shared memory). Still, StarPU was recently extended to exploit clusters of hybrid machines by relying on MPI. Since SimGrid's ability to accurately simulate MPI applications has already been demonstrated [BDG+13], combining both works should allow for obtaining good performances predictions of complex applications on large-scale high-end HPC infrastructures. Implementing this solution however is not straightforward as it requires integrating two different SimGrid APIs: *MSG* used in our solution for simulating hybrid machines and *SMPI* used for simulating MPI applications. Although both of these APIs rely on the same SimGrid core beneath, combining them was never intended in the initial SimGrid design and thus many small adjustments to the code are required. Moreover, to simulate StarPU MPI applications several technical challenges have to be overcome, such as segment data sharing for privatizing global variables, initialization of the StarPU application, initialization of MPI, etc.

There is an ongoing work on porting StarPU MPI applications on top of SimGrid, and most of the aforementioned issues have already been solved. There are still a few technical details to fix, and after these are resolved we will soon start to experiment with various applications and machines. We believe that this approach can also provide very faithful simulation predictions, but it needs to be validated on a wide range of settings. If the results are positive, then this approach should be even more useful to the community, as the resource saving when executing simulation on a commodity machine instead of running a real executions on a large cluster is even larger than in the case of single-node hybrid setups.

Controlling the simulation quality

An important aspect of the simulation that we have not addressed during this thesis is the validity domain of our simulation predictions. Our models are checked and calibrated on specific machines and specific inputs. They can easily be extrapolated to other setups but the accuracy of the obtained predictions is then more questionable. For example, our kernel models for `qr_mumps` are calibrated with specific matrices. We can use these models to simulate the factorization of a new matrix that we never calibrated before. However, if this matrix has some peculiar structure that leads to the execution of the kernels with input parameters that are very different from the parameter space explored with the initial calibration, the simulation might provide inaccurate predictions.

To validate the faithfulness of our simulation tool, we constantly compared native and SimGrid traces (see Figure 6.13, 6.15, 6.16 and 7.9). However, event for two stable, consecutive native executions, due to the small variability of kernel durations and the non-determinism of the modern machines, the scheduling decisions and hence the resulting traces are always quite different. Although two traces may visually look alike, we have not found an appropriate criteria to formally quantify this resemblance. We have considered different techniques of trace aggregation and statistical analysis, which all ended up to be only partially suitable for these use case. We have dedicated a significant time of our research to this topic, but have not yet found a satisfying solution. We believe that this specific question deserves a full study of its own and that it is the key to an in-depth understanding of the performance of such dynamic applications.

Opening new horizons

Building on the very accurate simulation predictions of the basic dense and sparse linear algebra applications, we would like to extend our study to other, more complex, use cases. The first step will be to investigate advanced versions of the `qr_mumps` solver, as such research requires little additional implementation effort.

In the research performed during this thesis, we have decided to fix the width of the block when factorizing parts of the matrix in order to simplify the modeling problem. In the near future, we intend to investigate the executions that rely on 2D partitioning of the sparse matrices into tiles, which allows for breaking down the panel factorization and thus for increasing the parallelism of

the application. In such executions, the number of parameters for the computation kernels will increase and we will have to revise our models. Still, the main principles of the `qr_mumps` execution are not significantly changed, therefore we believe that with the adjusted models performance predictions will be equally accurate as the ones for the 1D code.

`qr_mumps` was recently extended with a memory-aware scheduling algorithm [ABGL14] that applies strict memory bounds on the application. This solution add limitations that can ensure that the application will never exceed the RAM memory of the machine, while at the same time maximizing the parallelism of the execution. However, this algorithm is implemented partially in the application itself (part that keeps control of the allocated memory) and partially in the StarPU tasks of the application (part which indicates allocations and deallocations of the memory needed for doing a factorization). Since StarPU kernels are not executed in the simulation but only replaced by the delay representing their duration, simulating this version of `qr_mumps` would require some modifications to the source code so that simulation can maintain the same memory limitations as the native execution.

When we conducted our research, the GPU version of `qr_mumps` was still under development. Therefore, we decided to fix the evaluation of our approach to a more stable code version, where computations are performed solely on CPUs. In such context, since all workers use the same shared memory, there is no explicit data transfers and thus no need for advanced communication models as described in Section 5.4. Still, based on experience with dense linear algebra applications and the excellent results observed for heterogeneous machines (see Figure 6.12), we strongly believe that for `qr_mumps` using GPUs, our simulation predictions will be equally accurate.

Apart from different version of `qr_mumps`, we would also like to extend our study to other more complex applications implemented on top of StarPU, e.g., Fast Multipole Methods (FMM) (, introduced by Greengard and Rokhlin [GR87],) which can be applied on particle simulations problems where computing pair-wise interactions is reduced from quadratic to linear complexity. There are many implementations of FMM on top of different runtimes [CWO+10, CWO+10, YB12] including the one on top of StarPU [ABC+14]. This type of applications have even more irregular workloads than sparse linear algebra kernels. Hence, simulating them certainly contains many new challenges that are hard to anticipate.

Finally, the ultimate goal of our research is to simulate executions of full-fledged dynamic scientific HPC applications running on top of hybrid clusters. Thanks to the efforts of the StarPU developers on the stability of their software, real applications should be ported on top of StarPU in a near future and would then easily benefit from our work. Another possibility could be to directly target the simulation of specific applications such as BigDFT. BigDFT has been already successfully simulated with SMPI SimGrid [BDG+13], but only for single-core multi-node machines. A support for GPUs is also available in BigDFT, but as it is built directly on top of CUDA and not of a runtime system such as StarPU, one would have to significantly modify BigDFT to benefit from our work. It is also important to note that the development of direct GPU-GPU communications over the networks in BigDFT is underway. Being able to simulate such kind of operations would not only be valuable to the BigDFT developers, but also interesting from the modeling and performance evaluation point of view. Moreover, the control flow of BigDFT is currently static but the developers are considering moving to more dynamic task-based approaches using OpenMP. Such an irregular dynamic application would be a perfect use case for our future studies. Implementing simulation of an OpenMP application (even if using a restricted set of the OpenMP standard) would however require significant development work. A possible solution could reside in the recent efforts of specific OpenMP compilers, such as KSTAR [KST] project that aims at directly compiling OpenMP programs to StarPU and KAAPI.

Appendix A

References

A.1 Publications

A.1.1 International peer reviewed journals

[1] L. Stanisic, A. Legrand, and V. Danjean. An Effective Git And Org-Mode Based Workflow For Reproducible Research. *ACM SIGOPS Operating Systems Review*, 49:61 – 70, 2015. Special Topic: Repeatability and Sharing of Experimental Artifacts.

[2] L. Stanisic, S. Thibault, A. Legrand, B. Videau, and J.-F. Méhaut. Faithful Performance Prediction of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures. *Concurrency and Computation: Practice and Experience*, page 16, May 2015.

A.1.2 International peer reviewed conference proceedings

[3] L. Stanisic, E. Agullo, A. Buttari, A. Guermouche, A. Legrand, F. Lopez, and B. Videau. Fast and Accurate Simulation of Multithreaded Sparse Linear Algebra Solvers. July 2015. Submitted to the ICPADS conference.

[4] L. Stanisic and A. Legrand. Effective Reproducible Research with Org-Mode and Git. In *1st International Workshop on Reproducibility in Parallel Computing*, Porto, Portugal, Aug. 2014.

[5] L. Stanisic, S. Thibault, A. Legrand, B. Videau, and J.-F. Méhaut. Modeling and Simulation of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures. In *Euro-par - 20th International Conference on Parallel Processing*, Euro-Par 2014, LNCS 8632, pages 50–62, Porto, Portugal, Aug. 2014. Springer International Publishing Switzerland.

[6] L. Stanisic, B. Videau, J. Cronsioe, A. Degomme, V. Marangozova-Martin, A. Legrand, and J.-F. Mehaut. Performance Analysis of HPC Applications on Low-Power Embedded Platforms. In *DATE - Design, Automation & Test in Europe*, pages 475–480, Grenoble, France, Mar. 2013.

A.1.3 Short communications in conferences and workshops

[7] L. Stanisic and A. Legrand. *Actes du 10ème Atelier en Évaluation de Performances*, chapter Good practices for reproducible research, pages 29–30. Inria, 2014.

A.1.4 Master Thesis

[8] L. Stanisic. Towards Modeling and Simulation of Exascale Computing Platforms. Master's thesis, Universite Joseph Fourier, June 2012.

A.2 Bibliography

[AAD+10] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In Wen-mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, September 2010.

[AAD+11] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov. QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators. Anchorage, Alaska, USA, 5 2011.

[ABB+99] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' Guide (Third Ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[ABC+14] E. Agullo, B. Bramas, O. Coulaud, E. Darve, M. Messner, and T. Takahashi. Task-Based FMM for Multicore Architectures. *SIAM Journal on Scientific Computing*, 36(1):66–93, 2014.

[ABD+07] J. Albrecht, R. Braud, D. Dao, N. Topilski, C. Tuttle, A. C. Snoeren, and A. Vahdat. Remote control: distributed application configuration, management, and visualization with plush. In *Proceedings of the 21st conference on Large Installation System Administration Conference*, LISA'07, pages 15:1–15:19, Berkeley, CA, USA, 2007. USENIX Association.

[ABED+15] E. Agullo, O. Beaumont, L. Eyraud-Dubois, J. Herrmann, S. Kumar, L. Marchal, and S. Thibault. Bridging the Gap between Performance and Bounds of Cholesky Factorization on Heterogeneous Platforms. In *Heterogeneity in Computing Workshop 2015*, Hyderabad, India, May 2015.

[ABGL13] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. Multifrontal QR factorization for multicore architectures over runtime systems. In *Euro-Par 2013 Parallel Processing*, pages 521–532. Springer Berlin Heidelberg, 2013.

[ABGL14] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez. Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems. Technical Report IRI/RT–2014-03–FR, IRIT, 2014. Submitted to ACM TOMS.

[ABI+09] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Proceedings of the 15th Euro-Par Conference*, August 2009.

[ABP01] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory Comput. Syst.*, 34(2):115–144, 2001.

[ACM14] AMD Core Math Library Users Guide. Available at http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/acml.pdf, 2014.

[ADH+09] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan. Plasma Users' Guide. Technical report, Technical report, ICL, UTK, 2009.

[ADKL01] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[ADLL01] P. R. Amestoy, I. S. Duff, J.-Y. L'excellent, and X. S. Li. Analysis and Comparison of Two General Sparse Solvers for Distributed Memory Computers. *ACM Trans. Math. Softw.*, 27(4):388–421, December 2001.

[ADP96] P. R. Amestoy, I. S. Duff, and C. Puglisi. Multifrontal QR factorization in a multiprocessor environment. *Int. Journal of Num. Linear Alg. and Appl.*, 3(4):275–300, 1996.

[AGLP06] P. R. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel Computing*, 32(2):136–156, 2006.

[AISS95] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages Into the LogP Model – One Step Closer Towards a Realistic Model for Parallel Computation. In *Proc. of the 7th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 95–105, Santa Barbara, CA, 1995.

[AK02] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, 2002.

[ALG06] E. Agullo, J.-Y. L'Excellent, and A. Guermouche. A Preliminary Out-of-Core Extension of a Parallel Multifrontal Solver. In *Euro-Par 2006 Parallel Processing*, pages 1053–1063, Germany, 2006.

[ATN09] C. Augonnet, S. Thibault, and R. Namyst. Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures. In *3rd Workshop on Highly Parallel Processing on a Chip (HPPC)*, August 2009.

[ATNW11] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23:187–198, February 2011.

[BBB+11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[BBD+11] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Hérault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. Yarkhan, and J. J. Dongarra. Distibuted Dense Numerical Linear Algebra Algorithms on massively parallel architectures: DPLASMA. In *Proceedings of the 25th IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW'11), PDSEC 2011*, pages 1432–1441, Anchorage, United States, May 2011.

[BBD+12] G. Bosilca, A. Bouteiller, A. Danalis, T. Hérault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Computing*, 38(1-2):37–51, 2012.

[BBD+13] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. Dongarra. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. Vol. 15, No. 6:36–45, November 2013.

[BCAC+13] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec. Adding virtualization capabilities to the Grid'5000 testbed. In I. Ivanov, M. Sinderen, F. Leymann, and T. Shan, editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing, 2013.

[BCC+97] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[BCM+03] W. H. Bell, D. G. Cameron, A. P. Millar, L. Capozza, K. Stockinger, and F. Zini. OptorSim: A grid simulator for studying dynamic data replication strategies. *International Journal of High Performance Computing and Applications*, 17(4), 2003.

[BCOM+10] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. Hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 180–186, Washington, DC, USA, 2010. IEEE Computer Society.

[BDG+00] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, August 2000.

[BDG+13] P. Bedaride, A. Degomme, S. Genaud, A. Legrand, G. Markomanolis, M. Quinson, M. Stillwell, Lee, F. Suter, and B. Videau. Toward Better Simulation of MPI Applications on Ethernet/TCP Networks. In *4th International Workshop on Performance Modeling, Benchmarking and Simulation of HPC Systems (PMBS)*, November 2013.

[BDP01a] R. Bagrodia, E. Deelman, and T. Phan. Parallel Simulation of Large-Scale Parallel Applications. *IJHPCA*, 15(1):3–12, 2001.

[BDP01b] R. Bagrodia, E. Deelman, and T. Phan. Parallel Simulation of Large-Scale Parallel Applications. *IJHPCA*, 15(1):3–12, 2001.

[BFG+09] F. Broquedis, N. Furmento, B. Goglin, R. Namyst, and P.-A. Wacrenier. Dynamic Task and Data Placement over NUMA Architectures: An OpenMP Runtime Perspective. In M. Müller, B. de Supinski, and B. Chapman, editors, *Evolving OpenMP in an Age of Extreme Parallelism*, volume 5568 of *Lecture Notes in Computer Science*, pages 79–92. Springer Berlin Heidelberg, 2009.

[Big12] The BigDFT Scientific Application. Available at http://bigdft.org, 2012.

[BJK+95] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 1995. ACM.

[BLGE03] R. M. Badia, J. Labarta, J. Giménez, and F. Escalé. Dimemas: Predicting MPI Applications Behaviour in Grid Environments. In *Proc. of the Workshop on Grid Applications and Programming Tools*, June 2003.

[BLKD09] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures. *Parallel Comput.*, 35(1):38–53, January 2009.

[BNG15] T. Buchert, L. Nussbaum, and J. Gustedt. Towards Complete Tracking of Provenance in Experimental Distributed Systems Research. 2015. To appear.

[BRNR15] T. Buchert, C. Ruiz, L. Nussbaum, and O. Richard. A survey of general-purpose experiment management tools for distributed systems. *Future Generation Computer Systems*, 45(0):1 – 12, 2015.

[But13] A. Buttari. Fine-grained multithreading for the multifrontal QR factorization of sparse matrices. *SIAM Journal on Scientific Computing*, 35(4):C323–C345, 2013.

[BYF+09] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS*, pages 163–174, 2009.

[CAR] CARE: tool for monitoring the execution. Available at http://reproducible.io/.

[CARL14] A. Carpen-Amarie, A. Rougier, and F. Lübbe. Stepping Stones to Reproducible Research: A Study of Current Practices in Parallel Computing. In L. Lopes, J. Žilinskas, A. Costan, R. Cascella, G. Kecskemeti, E. Jeannot, M. Cannataro, L. Ricci, S. Benkner, S. Petit, V. Scarano, J. Gracia, S. Hunold, S. Scott, S. Lankes, C. Lengauer, J. Carretero, J. Breitbart, and M. Alexander, editors, *Euro-Par 2014: Parallel Processing Workshops*, volume 8805 of *Lecture Notes in Computer Science*, pages 499–510. Springer International Publishing, 2014.

[CDDP10] S. Collange, M. Daumas, D. Defour, and D. Parello. Barra: A Parallel Functional Simulator for GPGPU. In *IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication*, pages 351–360, 2010.

[CGL+14] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter. Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms. *Journal of Parallel and Distributed Computing*, 74(10):2899–2917, June 2014.

[Cha] Chameleon: A dense linear algebra software for heterogeneous architectures . Available at https://project.inria.fr/chameleon.

[CHE+14] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. An Evaluation of High-Level Mechanistic Core Models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.

[Cil] Cilk Arts. Cilk++. Available at http://www.cilk.com.

[CJLL95] P. Chrétienne, E. C. Jr., J. Lenstra, and Z. Liu, editors. *Scheduling Theory and its Applications*. John Wiley and Sons, 1995.

[CKP+93] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 1–12, New York, NY, USA, 1993. ACM.

[CLB09] P. Cicotti, X. S. Li, and S. B. Baden. Performance Modeling Tools for Parallel Sparse Linear Algebra Computations. In *Parallel Computing: From Multicores and GPU's to Petascale, Proceedings of the conference ParCo 2009, 1-4 September 2009, Lyon, France*, pages 83–90, 2009.

[CLQ08] H. Casanova, A. Legrand, and M. Quinson. SimGrid: a generic framework for large-scale distributed experiments. In *Proceedings of the 10th Conference on Computer Modeling and Simulation (EuroSim'08)*, 2008.

[CLT13] L. Carrington, M. Laurenzano, and A. Tiwari. Inferring large-scale computation behavior via trace extrapolation. In *Large-Scale Parallel Processing workshop (IPDPS'13)*, 2013.

[CPM+14] C. Collberg, T. Proebsting, G. Moraila, A. Shankaran, Z. Shi, and A. M. Warren. Measuring reproducibility in computer systems. Technical report, University of Arizona, March 2014.

[CSF13] F. Chirigati, D. Shasha, and J. Freire. ReproZip: Using Provenance to Support Computational Reproducibility. In *Proceedings of the 5th USENIX Workshop on the Theory and Practice of Provenance*, TaPP '13, pages 1:1–1:4, Berkeley, CA, USA, 2013. USENIX Association.

[CUB] CUDA development toolkit: cuBLAS. Available at http://docs.nvidia.com/cuda/cublas/.

[CUD07] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[CVV+08] E. Chan, F. Van Zee, R. Van De Geijn, P. Bientinesi, E. Quintana-Ortí, and G. Quintana-Ortí. *SuperMatrix: A multithreaded runtime scheduling system for algorithms-by-blocks*, pages 123–132. 2008.

[CWO+10] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, and R. Vuduc. Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures. In *Proc. IEEE Int'l. Parallel and Distributed Processing Symp. (IPDPS)*, Atlanta, GA, USA, April 2010.

[Dav11] T. A. Davis. Algorithm 915, SuiteSparseQR: Multifrontal multithreaded rank-revealing sparse QR factorization. *ACM Trans. Math. Softw.*, 38(1):8:1–8:22, December 2011.

[DDCHD90] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, March 1990.

[DHHW93] J. J. Dongarra, R. Hempel, A. J. Hey, and D. W. Walker. A Proposal for a User-Level, Message-Passing Interface in a Distributed Memory Environment. Technical report, Knoxville, TN, USA, 1993.

[DHN96] P. Dickens, P. Heidelberger, and D. Nicol. Parallelized Direct Execution Simulation of Message-Passing Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1090–1105, 1996.

[Din] Dinero IV Trace-Driven Uniprocessor Cache Simulator. Available at http://pages.cs.wisc.edu/~markhill/DineroIV/.

[Dis] Discussion about trace generated by executing qrm_starpu factorization of e18 matrix. https://github.com/inria-traces/trace.archive/blob/master/data/qrm_starpu/e18/index.org.

[DM98] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55, Jan 1998.

[DM09] L. Denby and C. Mallows. Variations on the Histogram. *Journal of Computational and Graphical Statistics*, 18(1):21–31, 2009.

[Don88] J. Dongarra. The linpack benchmark: An explanation. In *Proceedings of the 1st International Conference on Supercomputing*, pages 456–474, London, UK, UK, 1988. Springer-Verlag.

[DR83] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions On Mathematical Software*, 9:302–325, 1983.

[Dru09] C. Drummond. Replicability is not reproducibility: Nor is it good science. In *Proceedings of the Evaluation Methods for Machine Learning Workshop at the 26th ICML*, 2009.

[DW97] J. Dongarra and R. C. Whaley. A User's Guide to the BLACS v1.1. Technical Report UT-CS-95-281. LAPACK Working Note 94. Available at http://www.netlib.org/blacs/, 1997.

[ETRA09] T. Estrada, M. Taufer, K. Reed, and D. P. Anderson. EmBOINC: An Emulator for Performance Analysis of BOINC Projects. In *Proc. of the Workshop on Large-Scale and Volatile Desktop Grids (PCGrid)*, 2009.

[Exe] Executable paper grand challenge. Available at http://www.executablepapers.com.

[FC07] K. Fujiwara and H. Casanova. Speed and Accuracy of Network Simulation in the SimGrid Framework. In *Proceedings of the 2Nd International Conference on Performance Evaluation Methodologies and Tools*, ValueTools '07, pages 12:1–12:10, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[Fei15] D. G. Feitelson. From Repeatability to Reproducibility and Corroboration. *SIGOPS Oper. Syst. Rev.*, 49(1):3–11, January 2015.

[Fid] Figshare and Git(Hub). https://github.com/arfon/fidgit.

[Fig] Figshare: repository where users can make all of their research outputs available in a citable, shareable and discoverable manner. Available at http://figshare.com.

[Fur12] G. Fursin. Abstract: cTuning.org: Novel Extensible Methodology, Framework and Public Repository to Collaboratively Address Exascale Challenges. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis, Salt Lake City, UT, USA, November 10-16, 2012*, pages 1401–1402, 2012.

[GB02] T. Giuli and M. Baker. Narses: A Scalable Flow-Based Network Simulator. Technical Report cs.PF/0211024, Stanford University, 2002. Available at http://arxiv.org/abs/cs.PF/0211024.

[GBP07] T. Gautier, X. Besseron, and L. Pigeon. KAAPI: A Thread Scheduling Runtime System for Data Flow Computations on Cluster of Multi-processors. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, PASCO '07, pages 15–23, New York, NY, USA, 2007. ACM.

[GEE10] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval Simulation: Raising the Level of Abstraction in Architectural Simulation. In *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 307–318, February 2010.

[Gitb] GitHub: web-based Git repository hosting service. Available at http://github.com.

[GLS99b] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. Scientific And Engineering Computation Series. MIT Press, 2nd edition, 1999.

[GN89] A. Geist and E. G. Ng. Task scheduling for parallel sparse Cholesky factorization. *Int J. Parallel Programming*, 18:291–314, 1989.

[GNG+08] L. Genovese, A. Neelov, S. Goedecker, T. Deutsch, S. Ghasemi, A. Willand, D. Caliste, O. Zilberberg, M. Rayson, A. Bergman, et al. Daubechies Wavelets as a Basis Set for Density Functional Pseudopotential Calculations. *The Journal of Chemical Physics*, 129:014109, 2008.

[GR87] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73(2):325–348, December 1987.

[Grea] Green500: Ranking the worlds most energy-efficient supercomputers. Available at http://www.green500.org.

[Greb] The Green500 List - November 2014. Available at http://www.green500.org/lists/green201411.

[GS12] P. J. Guo and M. Seltzer. BURRITO: Wrapping Your Lab Notebook in Computational Infrastructure. In *Proceedings of the 4th USENIX Conference on Theory and Practice of Provenance*, TaPP'12, pages 7–7, Berkeley, CA, USA, 2012. USENIX Association.

[Gus03] F. Gustavson. High-performance linear algebra algorithms using new generalized data structures for matrices. *IBM Journal of Research and Development*, 47(1):31–55, Jan 2003.

[HGWW09] M.-A. Hermanns, M. Geimer, F. Wolf, and B. Wylie. Verifying Causality between Distant Performance Phenomena in Large-Scale MPI Applications. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 78–84, Feb 2009.

[Hin11] K. Hinsen. A data and code model for reproducible research and executable papers. *Procedia Computer Science*, 4(0):579 – 588, 2011. Proceedings of the International Conference on Computational Science.

[HIR] HIRLAM research program. Available at http://hirlam.org/.

[HKY+14] B. Haugen, J. Kurzak, A. YarKhan, P. Luszczek, and J. Dongarra. Parallel Simulation of Superscalar Scheduling. In *Parallel Processing (ICPP), 2014 43rd International Conference on*, pages 121–130, Sept 2014.

[HPS14] C. Hurlin, C. Pérignon, and V. Stodden. *RunMyCode.org: A Research-Reproducibility Tool for Computational Sciences*. Center for Open Science, 2014.

[HSL10] T. Hoefler, T. Schneider, and A. Lumsdaine. LogGOPSim - Simulating Large-Scale Applications in the LogGOPS Model. In *Proc. of the ACM Workshop on Large-Scale System and Application Performance*, pages 597–604, June 2010.

[IFH01] F. Ino, N. Fujimoto, and K. Hagihara. LogGPS: a Parallel Computational Model for Synchronization Analysis. In *Proc. of the eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP)*, pages 133–142, Snowbird, UT, 2001.

[Ioa05] Ioannidis, John P. A. Why Most Published Research Findings Are False. *PLoS Med*, 2(8):e124, 08 2005.

[Jai91] R. K. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1 edition, April 1991.

[KE14] K. Kim and V. Eijkhout. A parallel sparse direct solver via hierarchical DAG scheduling. *ACM Trans. Math. Softw.*, 41(1):3:1–3:27, October 2014.

[Kep] The Kepler Project. Available at http://kepler-project.org.

[KK96] L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[Knu84] D. E. Knuth. Literate Programming. *Comput. J.*, 27(2):97–111, May 1984.

[KST] KSTAR: An OpenMP compiler for portable and efficient application programming on task-based runtime systems. Available at https://gforge.inria.fr/projects/kstar/.

[LD03] X. Li and J. W. Demmel. SuperLU DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29:110–140, 2003.

[LEE+97] J. L. Lo, S. J. Eggers, J. S. Emer, H. M. Levy, R. L. Stamm, and D. M. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15:322–354, 1997.

[LFR+14] X. Lacoste, M. Faverge, P. Ramet, S. Thibault, and G. Bosilca. Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. In *23rd International Heterogeneity in Computing Workshop, IPDPS 2014*, Phoenix, AZ, 05/2014 2014. IEEE, IEEE.

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

[Li05] X. S. Li. An overview of superlu: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302–325, 2005.

[LTJ+14] M. A. Laurenzano, A. Tiwari, A. Jundt, J. Peraza, W. A. Ward Jr, R. Campbell, and L. Carrington. Characterizing the performance-energy tradeoff of small ARM cores in HPC computation. In *Euro-Par 2014 Parallel Processing*, pages 124–137. Springer International Publishing, 2014.

[Mag] MAGMA: Matrix Algebra on GPU and Multicore Architectures. Available at http://icl.cs.utk.edu/magma/.

[MDHS09] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing Wrong Data Without Doing Anything Obviously Wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 265–276. ACM, 2009.

[MJ09] A. Montresor and M. Jelasity. PeerSim: A Scalable P2P Simulator. In *Proc. of the 9th International Conference on Peer-to-Peer*, pages 99–100, September 2009.

[MKL12] Intel Math Kernel Library. Available at http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/mklman.pdf, 2012.

[Mon] Mont-Blanc: European Approach Towards Energy Efficient High Performance. MontBlanc. http://www.montblanc-project.eu/.

[Mon05] D. C. Montgomery. *Design and Analysis of Experiments, Student Solutions Manual.* Wiley, August 2005.

[MOR] Matrices Over Runtime Systems @ Exascale. http://icl.cs.utk.edu/projectsdev/morse/index.html.

[MR09] C. Minkenberg and G. Rodriguez. Trace-driven Co-simulation of High-performance Computing Systems Using OMNeT++. In *Proceedings of the 2Nd International Conference on Simulation Tools and Techniques*, Simutools '09, pages 65:1–65:8, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[MSL13] L. Mello Schnorr and A. Legrand. Visualizing More Performance Data Than What Fits on Your Screen. In A. Cheptsov, S. Brinkmann, J. Gracia, M. M. Resch, and W. E. Nagel, editors, *Tools for High Performance Computing 2012*, pages 149–162. Springer Berlin Heidelberg, 2013.

[MSV13] L. Marchal, O. Sinnen, and F. Vivien. Scheduling tree-shaped task graphs to minimize memory and makespan. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS, pages 839–850. IEEE Computer Society, 2013.

[NBL+06] S. Naicken, A. Basu, B. Livingston, S. Rodhetbhai, and I. Wakeman. Towards yet another peer-to-peer simulator. In *In Proc. Fourth International Working Conference Performance Modelling and Evaluation of Heterogeneous Networks (HET-NETs' 06)*, 2006.

[NnFG+10] A. Núñez, J. Fernández, J. D. García, F. García, and J. Carretero. New Techniques for Simulating High Performance MPI Applications on Large Storage Networks. *Journal of Supercomputing*, 51(1):40–57, 2010.

[NS3] The ns-3 Network Simulator. Available at http://www.nsnam.org.

[Nus82] H. Nussbaumer. Fast Fourier Transform and Convolution Algorithms. *Berlin and New York, Springer-Verlag.*, 2, 1982.

[Opea] An optimized BLAS library. Available at http://www.openblas.net/.

[Opeb] OpenCL: The open standard for parallel programming of heterogeneous systems. Available at https://www.khronos.org/opencl/.

[Ope13] OpenMP architecture review board. OpenMP 4.0 complete specifications, 2013.

[OPF10] S. Ostermann, R. Prodan, and T. Fahringer. Dynamic Cloud Provisioning for Scientific Grid Workflows. In *Proc. of the 11th ACM/IEEE Intl. Conf. on Grid Computing (Grid)*, October 2010.

[PACG11] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSS: A Full System Simulator for Multicore x86 CPUs. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 1050–1055, New York, NY, USA, 2011. ACM.

[PACR03] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. *SIGCOMM Comput. Commun. Rev.*, 33(1):59–64, January 2003.

[Pag] Page coloring. Available at https://www.freebsd.org/doc/en_US.ISO8859-1/articles/vm-design/page-coloring-optimizations.html.

[PBAL09] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical Task-Based Programming With StarSs. *International Journal of High Performance Computing Applications*, 23:284–299, 2009.

[PG07] F. Pérez and B. E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.

[PJN08] M. Pérache, H. Jourdren, and R. Namyst. MPC: A Unified Parallel Runtime for Clusters of NUMA Machines. In E. Luque, T. Margalef, and D. Benítez, editors, *Euro-Par 2008 – Parallel Processing*, volume 5168 of *Lecture Notes in Computer Science*, pages 78–88. Springer Berlin Heidelberg, 2008.

[PKL+11] D. Peter, D. Komatitsch, Y. Luo, R. Martin, N. Le Goff, E. Casarotti, P. Le Loher, F. Magnoni, Q. Liu, C. Blitz, et al. Forward and adjoint simulations of seismic wave propagation on fully unstructured hexahedral meshes. *Geophysical Journal International*, 186(2):721–739, 2011.

[PLa] PLaFRIM: Plateforme Fédérative pour la Recherche en Informatique et Mathématiques. Available at https://plafrim.bordeaux.inria.fr/.

[PWTR09a] B. Penoff, A. Wagner, M. Tuxen, and I. Rungeler. MPI-NeTSim: A Network Simulation Module for MPI. In *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, pages 464–471, Dec 2009.

[PWTR09b] B. Penoff, A. Wagner, M. Tüxen, and I. Rüngeler. MPI-NeTSim: A Network Simulation Module for MPI. In *Proc. of the 15th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 464–471, December 2009.

[PY90] C. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, 19(2):322–328, 1990.

[QOQOG+09] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. V. D. Geijn, F. G. V. Zee, and E. Chan. Programming Matrix Algorithms-by-blocks for Thread-level Parallelism. *ACM Trans. Math. Softw.*, 36(3):14:1–14:26, July 2009.

[RCV+12] A. Rico, F. Cabarcas, C. Villavieja, M. Pavlovic, A. Vega, Y. Etsion, A. Ramírez, and M. Valero. On the simulation of large-scale architectures using multiple application abstraction levels. *TACO*, 8(4):36, 2012.

[RDC+11] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramírez, and M. Valero. Trace-driven simulation of multithreaded applications. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2011, 10-12 April, 2011, Austin, TX, USA*, pages 87–96, 2011.

[Rep] How Bright Promise in Cancer Testing Fell Apart. Available at http://www.nytimes.com/2011/07/08/health/research/08genes.html?_r=0.

[RFT+05] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005.

[RGS09] D. D. Roure, C. Goble, and R. Stevens. The design and realisation of the virtual research environment for social sharing of workflows. *Future Generation Computer Systems*, 25(5):561 – 567, 2009.

[RHB+11] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob. The Structural Simulation Toolkit. *SIGMETRICS Perform. Eval. Rev.*, 38(4):37–42, March 2011.

[Rie06] R. Riesen. A Hybrid MPI Simulator. In *Proc. of the IEEE International Conference on Cluster Computing*, September 2006.

[Rog09] Roger D. Peng and Sandrah P. Eckel. Distributed Reproducible Research Using Cached Computations. *Computing in Science and Engineering*, 11(1):28–34, 2009.

[RRE14] C. Ruiz, O. Richard, and J. Emeras. Reproducible Software appliances for Experimentation. In *Proceedings of the 9th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (Tridentcom)*, 2014.

[RSBM+10] P. Rocca-Serra, M. Brandizi, E. Maguire, N. Sklyar, C. Taylor, K. Begley, D. Field, S. Harris, W. Hide, O. Hofmann, S. Neumann, P. Sterk, W. Tong, and S.-A. Sansone. ISA software suite: supporting standards-compliant experimental annotation and enabling curation at the community level. *Bioinformatics*, 26(18):2354–2356, September 2010.

[RSRVO13b] C. C. Ruiz Sanabria, O. Richard, B. Videau, and I. Oleg. Managing Large Scale Experiments in Distributed Testbeds. In *Proceedings of the 11th IASTED International Conference*, pages 628–636. IASTED, ACTA Press, feb 2013.

[SBJN13] L. Sarzyniec, T. Buchert, E. Jeanvoine, and L. Nussbaum. Design and Evaluation of a Virtual Experimental Environment for Distributed Systems. In *PDP2013 - 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing, pages 172 – 179, Belfast, United Kingdom, February 2013. IEEE.

[Sch82] R. Schreiber. A new implementation of sparse Gaussian elimination. *ACM Transactions On Mathematical Software*, 8:256–276, 1982.

[SCW+02] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–17, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[SDDD12] E. Schulte, D. Davison, T. Dye, and C. Dominik. A multi-language computing environment for literate programming and reproducible research. *Journal of Statistical Software*, 46(3):1–24, 1 2012.

[SGS10] J. Stone, D. Gohara, and G. Shi. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering*, 12(3):66–73, May 2010.

[SLP14] V. Stodden, F. Leisch, and R. D. Peng, editors. *Implementing Reproducible Research*. The R Series. Chapman and Hall/CRC, April 2014.

[SON] SONGS: Simulation Of Next Generation Systems. Available at http://infra-songs.gforge.inria.fr/.

[SSW] StarPU + SimGrid project website. Available at http://starpu-simgrid.gforge.inria.fr.

[Ste11] Steen, R Grant. Retractions in the scientific literature: is the incidence of research fraud increasing? *Journal of Medical Ethics*, 37(4):249–253, 2011.

[Tav13] The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. *Nucleic Acids Research*, 41(Web Server issue):gkt328–W561, May 2013.

[TCSS07] M. M. Tikir, L. Carrington, E. Strohmaier, and A. Snavely. A Genetic Algorithms Approach to Modeling the Performance of Memory-bound Computations. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 47:1–47:12, New York, NY, USA, 2007. ACM.

[THW02a] H. Topcuouglu, S. Hariri, and M.-y. Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274, March 2002.

[THW02b] H. Topcuouglu, S. Hariri, and M.-y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, Mar 2002.

[THW10] J. Treibig, G. Hager, and G. Wellein. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW '10, pages 207–216, Washington, DC, USA, 2010. IEEE Computer Society.

[TKE+07] M. Taufer, A. Kerstens, T. Estrada, D. Flores, and P. J. Teller. SimBA: A Discrete Event Simulator for Performance Prediction of Volunteer Computing Projects. In *Proc. of the 21st International Workshop on Principles of Advanced and Distributed Simulation*, pages 189–197, 2007.

[TLCS09] M. Tikir, M. Laurenzano, L. Carrington, and A. Snavely. PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications. In *Proc. of the 15th International Euro-Par Conference on Parallel Processing*, number 5704 in LNCS, pages 135–148. Springer, August 2009.

[Top] TOP500 Supercomputer Site. Available at http://www.top500.org.

[Traa] Inria Parallel Program Trace Archive. https://github.com/inria-traces/trace.archive.

[Trab] Report of e18 matrix factorization from Inria Parallel Program Trace Archive. https://github.com/inria-traces/trace.archive/blob/master/data/qrm_starpu/e18/index.org.

[Tur48] A. M. Turing. Rounding-off errors in matrix processes. *The Quarterly Journal of Mechanics and Applied Mathematics*, 1(1):287–308, 1948.

[UFM] The University of Florida Sparse Matrix Collection. Available at http://www.cise.ufl.edu/research/sparse/matrices/.

[UJM+12] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 335–344, New York, NY, USA, 2012. ACM.

[Var01] A. Varga. The omnet++ discrete event simulation system. In *In ESM'01*, 2001.

[VSCL13] P. Velho, L. Schnorr, H. Casanova, and A. Legrand. On the Validity of Flow-level TCP Network Models for Grid and Cloud Simulations. *ACM Transactions on Modeling and Computer Simulation*, 23(3), October 2013.

[VYW+02] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and Accuracy in a Large-scale Network Emulator. *SIGOPS Oper. Syst. Rev.*, 36(SI):271–284, December 2002.

[WM08] V. Weaver and S. McKee. Are Cycle Accurate Simulations a Waste of Time? In *Proc. of the 7th Workshop on Duplicating, Deconstruction and Debunking*, Beijing, China, June 2008.

[WM11] X. Wu and F. Mueller. ScalaExtrap: trace-based communication extrapolation for SPMD programs. In *Proc. of the 16th ACM symposium on Principles and Practice of Parallel Programming (PPoPP'11)*, pages 113–122, 2011.

[WPD01] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[WPSAM10] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 235–246, March 2010.

[XDCC04] H. Xia, H. Dail, H. Casanova, and A. Chien. The MicroGrid: Using Emulation to Predict Application Performance in Diverse Grid Network Environments. In *In Proceedings of the Workshop on Challenges of Large Applications in Distributed Environments (CLADE'04). IEEE Press, 2004. Published in conjunction with the Thirteenth IEEE International Symposium on High-Performance Distributed Computing*, pages 52–63. Society Press, 2004.

[YB12] R. Yokota and L. A. Barba. A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems. *Int. J. High Perform. Comput. Appl.*, 26(4):337–346, November 2012.

[YKD] A. YarKhan, J. Kurzak, and J. Dongarra. QUARK Users' Guide: QUeueing And Runtime for Kernels.

[Z214] Companion of the StarPU+SimGrid article. Hosted on Figshare: http://dx.doi.org/10.6084/m9.figshare.928338, 2014. Online version of [5] with access to the experimental data and scripts (in the org source).

[ZCZ10] J. Zhai, W. Chen, and W. Zheng. PHANTOM: Predicting Performance of Parallel Applications on Large-Scale Parallel Machines Using a Single Node. In *Proc. of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 305–314, January 2010.

[Zen] Zenodo: open dependable home for the long-tail of science. Available at http://zenodo.org.

[ZKK04] G. Zheng, G. Kakulapati, and L. Kalé. BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines. In *Proc. of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, April 2004.

[ZLN+15] Y. Zhibin, E. Lieven, G. Nilanjan, L. Tao, J. Lizy, J. Hai, X. Cheng-Zhong, and W. Junmin. GPGPU-MiniBench: Accelerating GPGPU Micro-Architecture Simulation. *IEEE Transactions on Computers*, 2015.

Appendix B

Additional figures

```
File Edit Options Buffers Tools files Sh-Script Help
echo "##########################################" >> $filedat
echo "** COMPILATION OF STARPU:" >> $filedat
echo "Make..."
if [[ $recompile == 1 ]]; then
    make -j5 >> $filedat
    make install
else
    echo "# used previous compile" >> $filedat
fi
echo "##########################################" >> $filedat
if [[ "$starpu_program" != qrm ]]; then
    if [[ "$starpu_program" != magma ]]; then
        echo "** PROGRAM SCRIPT:" >> $filedat
        cat $starpu_program >> $filedat
        echo "##########################################" >> $filedat
        echo "** PROGRAM BINARY LIBRARIES:" >> $filedat
        ldd $starpu_program_binary >> $filedat
        echo "##########################################" >> $filedat
    fi
fi

##################################################
# Collecting machine info
set +e #All following metadata is not crucial
echo "* MACHINE INFO:" >> $filedat
echo "** HOSTNAME:" >> $filedat
hostname >> $filedat
echo "##########################################" >> $filedat
echo "** MEMORY HIERARCHY:" >> $filedat
lstopo --of console >> $filedat
echo "##########################################" >> $filedat
echo "** STARPU MACHINE DISPLAY:" >> $filedat
./tools/starpu_machine_display >> $filedat
echo "##########################################" >> $filedat
echo "** CPU INFO:" >> $filedat
cat /proc/cpuinfo >> $filedat
echo "##########################################" >> $filedat
echo "** CPU GOVERNOR:" >> $filedat
if [ -f /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor ];
then
    cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor >> $filedat
else
    echo "Unknown (information not available)" >> $filedat
fi
echo "##########################################" >> $filedat
echo "** CPU FREQUENCY:" >> $filedat
if [ -f /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq ];
then
    cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq >> $filedat
else
    echo "Unknown (information not available)" >> $filedat
fi
echo "##########################################" >> $filedat
echo "** GPU INFO FROM NVIDIA-SMI:" >> $filedat
nvidia-smi -q >> $filedat
echo "##########################################" >> $filedat
echo "** LINUX AND GCC VERSIONS:" >> $filedat
cat /proc/version >> $filedat
```

Figure B.1: Script for running experiments and automatically capturing meta-data.

```
File  Edit  Options  Buffers  Tools  files  Org  Tbl  Help
#+TITLE: Native execution on pythagore
#+DATE: lun. avril 14 13:03:35 CEST 2014
#+AUTHOR: luka
#+FILE: SoloStarpuData10.org

* COMPILATION:
** PEOPLE LOGGED WHEN EXPERIMENT STARTED:
luka     :0           2014-04-14 10:24 (console)
luka     pts/0        2014-04-14 10:24
luka     pts/1        2014-04-14 10:31
luka     pts/3        2014-04-14 12:03
##############################################
** ENVIRONMENT VARIABLES:...
** CONFIGURATION OF STARPU:...
** COMPILATION OF STARPU:...
* MACHINE INFO:
** HOSTNAME:
pythagore
##############################################
** MEMORY HIERARCHY:...
** STARPU MACHINE DISPLAY:...
** CPU INFO:...
** CPU GOVERNOR:...
** CPU FREQUENCY:...
** LINUX AND GCC VERSIONS:
Linux version 3.4.63-2.44-desktop (geeko@buildhost) (gcc version 4.7.1 20120723 [gcc-4_7-branch
revision 189773] (SUSE Linux) ) #1 SMP PREEMPT Wed Oct 2 11:18:32 UTC 2013 (d91a619)
##############################################
* CODE REVISIONS:
** GIT REVISION OF STARPU+SIMGRID REPOSITORY:...
** SVN REVISION OF ORIGINAL STARPU CODE:...
** SIMGRID VERSION:...
* RUNNING OPTIONS:
STARPU_HOME=/home/luka/starpusg/starpusg-latest STARPU_HOSTNAME=pythagore
STARPU_HISTORY_MAX_ERROR=50 STARPU_CALIBRATE=0 STARPU_NCPU=0 STARPU_NCUDA=1 STARPU_NOPENCL=0
STARPU_SCHED=dmda ./examples/cholesky/cholesky_implicit -size 9600 -nblocks 10
##############################################
* ELAPSED TIME:
Elapsed:    1.945568416 seconds
* STDERR OUTPUT:
Computation took (in ms)
Synthetic GFlops : 425.57
* STDOUT OUTPUT:
Makespan (in ms): 693.10
* CALIBRATION:
** File: pythagore.latency...
** File: pythagore.affinity...
** File: pythagore.platform.xml...
** File: pythagore.config...
** File: pythagore.bandwidth...
** File: chol_model_22.pythagore...
** File: chol_model_11.pythagore...
** File: chol_model_21.pythagore...
* BENCHMARKING OUTPUT:...
* MAKESPAN APPROXIMATION FROM PAJE TRACE [ms]:...
```

Figure B.2: Output of the execution containing both meta-data and the experiment results.

```
File Edit Options Buffers Tools files Org Tbl Help
#+TITLE:       Lab book StarPU+Simgrid
#+AUTHOR:      Luka Stanisic
#+LANGUAGE:    en
#+TAGS: IMPORTANT(i) TEST(t) DEPRECATED(d) QRM(q) MAGMA(m)
#+TAGS: @PAUL(p) @LUKA(l) @SURAJ(s)
#+TAGS: _WINNETOU(W) _ATTILA(A) _HANNIBAL(H) _CONAN(C) _FROGGY(F) _MIRAGE(M) _FOURMI(O) _K40(K)
_MANUMANU(N) _IDCHIRE(H) _IDGRAPH(G)
#+TAGS: #EUROPAR14(e)

* README:                                                          :@LUKA:
** General:
   - This file corresponds to the lab books like the ones used by biologist, chemists etc.
   - It contains explanations of how things are organized, what is the workflow for doing
experiments, changes made to the code (as accurate as possible) and the observed behavior in the
"* Data" section. Note that this last section is available only in data branch, as we want to
keep master branch clean from any results-it contains only source code and the analysis (R code)
** Experiments workflow:
   1) Create a new branch
   2) Make sure everything is commited
   3) Run run_bench_StarPU script with the desired parameters
   4) Run run_inject_StarPUSG script with the desired parameters
   5) Do the analysis
   6) Add to this file into "* Data" section the entry for the results, using the template
described below
   7) Commit/push the results of this separate branch
   8) Merge this new branch with the remote "data" branch
** Paul help:                                                       :@PAUL:...
** Adding histograms to all data files:...
** Plotting all histograms of a certain file:...
** Extracting all GFlops as .csv and plotting results:...
** Example for using run_bench_StarPU.sh:                           :@LUKA:
   STARPU_NCPU=0 STARPU_NCUDA=3 STARPU_NOPECL=0 STARPU_SIZE=72000 STARPU_BLK=75 STARPU_SCHED=dmda
STARPU_CALIBRATE=1 STARPU_PROGRAM=cholesky ./run_bench_StarPU.sh -d data/dataNew -c -f -v
** Example for using run_inject_StarPUSG.sh:                        :@LUKA:
   STARPU_SCHED=dmda STARPU_PROGRAM=cholesky ./run_inject_StarPUSG.sh -n
data/dataNew/SoloStarpuData0.org -d data/dataNew -c -f -v
** Example for running benchmarking experiments in a for loop:...
** Example for running simulation experiments in a for loop:...
** Running qrm benchmarking:...
** Running qrm simulation:...
* Template for data entry:...
* Organization of git
** remote/origin/master branch:...
** remote/origin/data# branches:...
** remote/origin/data branch:...
* Git TAGs:
** Stable versions:...
** starpu_bench:...
** All tags from git:...
* Organization of code
** scripts:...
** src/...
** analysis/...
** .starpu/...
** backup/                                                     :DEPRECATED:...
* Additional feature:
* Changes:...
* Data: /
```

Figure B.3: Documentation part of the laboratory notebook.

```
File  Edit  Options  Buffers  Tools  files  Org  Tbl  Help
#+TITLE:       Lab book StarPU+Simgrid
#+AUTHOR:      Luka Stanisic
#+LANGUAGE:    en
#+TAGS: IMPORTANT(i) TEST(t) DEPRECATED(d) QRM(q) MAGMA(m)
#+TAGS: @PAUL(p) @LUKA(l) @SURAJ(s)
#+TAGS: _WINNETOU(W) _ATTILA(A) _HANNIBAL(H) _CONAN(C) _FROGGY(F) _MIRAGE(M) _FOURMI(O) _K40(K)
_MANUMANU(N) _IDCHIRE(H) _IDGRAPH(G)
#+TAGS: #EUROPAR14(e)

* README:                                                          :@LUKA:...
* Template for data entry:...
* Organization of git...
* Git TAGs:...
* Organization of code...
* Additional feature:
* Changes:...
* Data:
** data1:                                                     :TEST:@LUKA:...
** data1403                                                   :TEST:@PAUL:...
** dataOld                                                :IMPORTANT:@PAUL:...
** dataBabeltest                                               :TEST:@LUKA:...
** dataLUSUS                                           :TEST:@PAUL:_ATTILA:...
** dataCholeskySUS                                       :TEST:@PAUL:_ATTILA:...
** dataLUHannibal                                      :TEST:@PAUL:_HANNIBAL:...
** dataLUHannibalok                                    :TEST:@PAUL:_HANNIBAL:...
** dataCholwscripts                                      :TEST:@PAUL:_ATTILA:...
** dataChol1306                                        :TEST:@PAUL:_HANNIBAL:...
** dataCompMeanHisto                                   :TEST:@PAUL:_HANNIBAL:...
** dataChol2006                                            :@LUKA:_HANNIBAL:...
** dataCholPaje1                                 :IMPORTANT:@LUKA:_HANNIBAL:...
** dataTest3007                                        :TEST:@LUKA:_HANNIBAL:...
** dataLU2808                                    :@LUKA:_HANNIBAL:...
** dataCUDAfix                                             :@LUKA:_HANNIBAL:...
** dataSleep1                                    :IMPORTANT:@LUKA:_WINNETOU:...
** dataSleep2                                          :TEST:@LUKA:_WINNETOU:...
** dataSleep3                                                  :TEST:@LUKA:...
** dataT2                                              :TEST:@LUKA:_HANNIBAL:...
** dataUsedSize                                  :IMPORTANT:@LUKA:_HANNIBAL:...
** dataBord1                             :IMPORTANT:@LUKA:_ATTILA:_HANNIBAL:
*** git:...
*** Notes:
    - These are the experiments we performed when I was in Bordeaux in November 2013
    - Finally we got native and simgrid matching for large matrix size
    - Everything is very well explained in Report
** dataBord2                        :IMPORTANT:@LUKA:_ATTILA:_CONAN:#EUROPAR14:...
** dataBord3                             :IMPORTANT:@LUKA:_HANNIBAL:#EUROPAR14:...
** dataLU0912*            :IMPORTANT:@LUKA:_ATTILA:_HANNIBAL:_CONAN:#EUROPAR14:...
** dataCUDAbench :IMPORTANT:@LUKA:_ATTILA:_HANNIBAL:_CONAN:_FROGGY:#EUROPAR14:...
** art**         :IMPORTANT:@LUKA:_ATTILA:_HANNIBAL:_CONAN:_FROGGY:#EUROPAR14:...
** dataK20*                              :IMPORTANT:@LUKA:_FROGGY:#EUROPAR14:...
** dataFrogcpu                                     :IMPORTANT:@LUKA:_FROGGY:...
** dataTestN                                            :TEST:@LUKA:_ATTILA:...
** dataMirT                                             :TEST:@LUKA:_MIRAGE:...
** dataMirT2                                       :IMPORTANT:@LUKA:_MIRAGE:...
** dataQt1                                          :TEST:QRM:@LUKA:_FOURMI:...
** dataQall1                                   :IMPORTANT:QRM:@LUKA:_FOURMI:...
** dataK40                                              :TEST:@LUKA:_K40:...
** dataQall2                                   :IMPORTANT:QRM:@LUKA:_FOURMI:...
```

Figure B.4: Notes about all experimentation results stored in laboratory notebook.

```
File  Edit  Options  Buffers  Tools  files  Org  Tbl  Help
** dataFATPIPE                                    :TEST:@LUKA:_ATTILA:_CONAN:
*** git:...
*** Experimentation diary:
    We will try here to run everything directly from my LabBook.org file, using org-babel
feature. This way we can use literate programming approach both for experiments and analysis
directly.

    Process will be divided in several phases, that will be represented here as subsections:

**** Start: Creating new branch...
**** Connecting to the remote machine frontend:...
**** Connecting to the experimentation machine
Lets now connect to the remote machine on which we want to do the experiments:
#+begin_src sh :results output :session org-sh
ssh attila
#+end_src

#+RESULTS:
#+begin_example

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue Jul 29 15:07:20 2014 from morris
#+end_example

#+begin_src sh :results output :session org-sh...

#+RESULTS:
#+begin_example...

Navigating through the folders on remote machine is a bit painful, luckily we will not use this
that often:
#+begin_src sh :results output :session org-sh...

#+RESULTS:
#+begin_example...

Now we are in the "starpu_simgrid" folder on attila machine
**** Pulling the right remote branch...
**** Testing source code...
**** Looking at platform.xml to use FATPIPE:...
**** Testing experiments with CUDA...
**** Changing platform description...
**** Running first real experiment...
**** Running all experiments...
**** Adding data files to the repository...
**** Doing experiments on conan machine...
**** Try simulations on local machine...

**** Analysis...
**** Reverting ...
**** Final remarks...
*** Notes:...
```

Figure B.5: Conducting experiments directly from the laboratory notebook.

```
File  Edit  Options  Buffers  Tools  files  Org  Tbl  Help
  Luka Stanisic\inst{1} \and
  Samuel Thibault\inst{2} \and
  Arnaud Legrand\inst{1} \and
  Brice Videau\inst{1} \and
  Jean-François Méhaut\inst{1}
}

\maketitle
#+END_LaTeX

#+BEGIN_LaTeX...

#+call: Download_data()
#+call: Download_other()
#+call: Init_data()
#+call: R_init() :results output silent
* Initialization                                                    :noexport:...
* Introduction...
* Related Work...
* Porting StarPU over SimGrid...
* Experimental Setting...
* Modeling runtime system...
* Modeling communication in hybrid systems...
* Modeling computation
#+LaTeX: \label{sec.model.comp}

When running simulation, the actual result of the application is of no
interest. Hence the execution of each kernel is replaced by a virtual
delay accounting for its duration. In our initial approach, we used
the mean duration of each computation kernel, which was benchmarked by
StarPU during the calibration phase. Although this was producing
satisfactory results, using a fixed value leads to a deterministic
schedule in simulation. This may bias the simulation and does not
allow to verify the ability of the scheduling algorithms to handle the
variability of the resources.

Therefore, we modified StarPU to capture the timing of every
computation during a *Native* execution. Such collection of data can
then be used to analyze the computation time distribution which can be
approximated using irregular histograms\cite{dhist}, as regular ones
(with uniform bin-width) revealed very inefficient at representing
details of distributions where a small but non-negligible fraction of
values are an order of magnitude larger than the vast majority of
measurements. Such approximation can then be used in the simulation by
generating pseudo-random variables from the histograms.

Although this technique allows to obtain different simulated schedules
by changing the seed of the simulation, no significant gain in term of
accuracy could be observed for the applications and machines we used
so far. The makespan is always very similar in both cases (mean
duration vs. random duration following an approximation of the
original distribution). Nonetheless, we strongly believe that in some
more complex use cases, e.g., sparse linear algebra algorithms, using
fine models like histograms may provide more precise predictions.
#+LaTeX: \vspace{-.3cm}

* Prediction Accuracy in a Wide Range of Settings...
* Conclusion and Future work...
```

Figure B.6: Org-mode article contains both text and the analysis code.