# Modeling, Prediction and Optimization of Energy Consumption of MPI Applications using SimGrid

Franz Heinrich

Communauté
UNIVERSITÉ Grenoble Alpes

# THÈSE

Pour obtenir le grade de

## DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

**Franz Christian HEINRICH**

Thèse dirigée par **Arnaud LEGRAND, CNRS**

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

# Modélisation, prédiction et optimisation de la consommation énergétique d'applications MPI à l'aide de SimGrid

# Modeling, Prediction and Optimization of Energy Consumption of MPI Applications using SimGrid

Thèse soutenue publiquement le **21 mai 2019**,
devant le jury composé de :

**Amina GUERMOUCHE**
Maîtresse de Conférences, Télécom SudParis, France, Examinatrice
**Laurent LEFÈVRE**
Chargé de Recherche, Inria / ENS de Lyon, France, Rapporteur
**Jean-François MÉHAUT**
Professeur, Université Grenoble-Alpes, France, Président
**Martin SCHULZ**
Professeur, Technische Universität München, Allemagne, Rapporteur
**Arnaud LEGRAND**
Directeur de Recherche, LIG, CNRS, France, Directeur de thèse

# Abstract

The High-Performance Computing (HPC) community is currently undergoing disruptive technology changes in almost all fields, including a switch towards massive parallelism with several thousand compute cores on a single GPU or accelerator and new, complex networks.

The energy consumption of these machines will continue to grow in the future, making energy one of the principal cost factors of machine ownership. This explains why even the classic metric "flop/s", generally used to evaluate HPC applications and machines, is widely regarded as to be replaced by an energy-centric metric "flop/watt".

One approach to predict energy consumption is through simulation, however, an accurate simulation of the system is crucial to estimate the energy faithfully. In this thesis, we contribute to the performance and energy prediction of HPC architectures. We propose an energy model which we have implemented in the open source SimGrid simulator. We validate this model by carefully and systematically comparing it with real experiments. We leverage this contribution to both evaluate existing and propose new DVFS governors that are designed to suit the HPC context.

# Résumé

La communauté du calcul haute performance (HPC) est actuellement en pleine mutation, avec des évolutions technologiques majeures telles que le parallélisme massif apporté par des milliers de cœurs de calcul sur un seul accélérateur de type GPU ou bien les réseaux d'interconnexion à très haut débit.

La consommation d'énergie de ces machines est appelée à continuer à croître dans les années à venir, faisant de l'énergie l'un des principaux facteurs de coût. Cela explique pourquoi la métrique classique "flop/s", généralement utilisée pour évaluer les la performance des applications et des infrastructures HPC, est progressivement remplacée par des métriques centrées sur l'énergie comme le "flop/watt".

La simulation est une approche possible pour prédire la consommation d'énergie de ces infrastructures. Cependant, il est nécessaire de mettre en place une simulation fidèle du système pour obtenir une prédiction de performance fiable. Dans cette thèse, nous contribuons à la prédiction de la performance et de la consommation énergétique des architectures HPC. Nous proposons un modèle d'énergie que nous avons implémenté dans le simulateur open source SimGrid. Nous validons ce modèle avec soin en le comparant systématiquement avec des expériences réelles. Nous utilisons cette contribution pour évaluer des algorithmes déjà existant de régulation de la fréquence afin de réduire la consommation énergétique et nous proposons de nouveaux gouvernors DVFS spécialement conçus pour le contexte HPC.

# Acknowledgements

As a student, I heard many stories from and about PhD students and how they had to cope with working on a scientific project for years. Making this experience myself was quite humbling, and I think it is reasonable to say that without the help of many amazing people I would not have managed to achieve results I present in this thesis. I therefore decided to dedicate more space than just the generic one-page acknowledgements to the people that have helped and accompanied me throughout my PhD because I believe that these people have absolutely deserved it.

## My Advisor

► **Arnaud Legrand** At some point during my thesis, Tom said, "After three years as a PhD student, you have no respect left for your advisor." The contrary is true and after 4.5 years of working with you, I find it suprisingly difficult to write these lines because words cannot really convey what this fantastic journey with you means to me personally.

Instead of going into details, I would like to express my admiration for you, not only because of your scientific brilliance and rigor but also because of your great leadership and proximity. I feel the deepest gratitude for all the time and effort it took you to guide me in my endeavour (and the patience that I undoubtedly tested numerous times along the way). Your kindness and politeness that you exhibit when dealing with your PhD students and colleagues have impressed and influenced me and clearly are one of the reasons why your PhD students sometimes have to fight against all the other people that want to collaborate with you!

## The PhD Committee

► **Martin Schulz, Laurent Lefèvre, Amina Guermouche and Jean-François Méhaut** I am truly honored that all of you, despite having a tight schedule on your own,

accepted to review this manuscript and that you attended my defense. I also highly appreciate your feedback on this work and the discussions that followed my presentation.

## My Team: POLARIS

► **Tom Cornebize** Be it on business trips or during a private weekend in Aix-les-bains, I've enjoyed spending time with you. Even better, you have been the best intern I have ever had (also the only one): autonomous, good ideas and great rigor. I think your work is highly interesting and I hope you will find enough motivation to keep this high standard!

► **Vincent Danjean** Thank you for sharing your huge knowledge of operating systems / Debian with me - this has helped me in several cases quite significantly with my research. I have been particularly impressed by the lesson you taught me when you resolved a day-long bug hunt I was on in just a few minutes. Another thing that impressed me is your millionaire's cake. You surely could be one if you started selling it! Thanks for the recipe!

► **Augustin Degomme** Your technical knowledge of SMPI is highly appreciated, not only during several debugging sessions where you have helped me out. Thank you for all your efforts and for being the witty, always friendly guy that you are!

► **Nicolas Gast** Thanks for telling me about the ADMM algorithm and hence giving me a great idea for a DVFS manager! Also for being the funny, humble and active person that you are. It was a pleasure to work with you!

► **Bruno Gaujal** Thanks to you, I can now say that I am somewhat able to understand French people speaking with a slight accent from the south. Besides that, I appreciate your humour but also your knowledge and enthusiasm!

► **Florence Perronin** Thanks for being there for the PhD students that seek help, be it scientifically or personally, and your very pleasant company during breaks!

► **Stéphan Plassart** Even though you arrived only in late 2016, you quickly became one of the closest colleagues and also a good friend. I don't exaggerate when I say that I was able to improve my French significantly thanks to your corrections and patience during our lunch- and coffee-breaks. I'm glad you listened when things were not working, answered questions on the French system, explained things you

are passionate about (or not… like the CVEC) and showed me Aix-les-bains. We still need to play a first round of golf together soon!

► **Annie Simon** The POLARIS team members can consider themselves lucky to have such a caring assistant. I have come to highly appreciate your humour and talking to you was always a more-than-pleasant distraction from my scientific work. You are truly the good soul of this team, and I hope that they get to work with you for a long time to come!

► **Lukas Schnorr** Thank you for being a great friend and colleague! I have loved and missed (after your departure) our discussions on reproducibility, experiments and everything else. Getting to know you closer was a real pleasure and not only your professionalism, deep understanding of your domain but also your kindness towards others is what makes you a fantastic teacher. I hope we get to see each other again rather sooner than later!

► **Jean-Marc Vincent** Thank you so much for your help with my presentation and your comments on my work. This has undoubtedly prepared me for my defense. It has always been a pleasure to discuss with you, not only because of your humble character, but because you are so knowledgeable in so many fields and always willing to share.

► **Others** I would like to thank all the other current and former members of POLARIS (Guillaume Huard, Philippe Waille, Pedro Bruel, Bruno Donassolo, Vinìcius Garcia Pinto, Luca Stanisic, our interns and everyone I forgot) for the great time in the lab!

## POLARIS's Twin-Team: DATAMOVE

The DATAMOVE team shares almost everything with POLARIS: The same corridor, offices, food, events, …

► **Pierre-François Dutot** Thanks for all the information you gave me on Hawaii and wine tasting in France!

► **Grégory Mounié** Your explanations on OS concepts and your great new ideas on how to debug seemingly unexplicable behaviors have helped me with my scientific work. Additionally, even though my last rehearsal was very spontaneous, you still took the time and helped me by asking great questions and pointing out what remained unclear to you. Thanks for your support!

► **Pierre Neyron** Guten Tag! Surprisingly, you are the only member of our teams that constantly claims to be the wrong person to talk to: You reject all blaming when the internet access is broken or when the printer is malfunctioning. The reason for this could be that you *really* are not the right person for this! When it comes to Grid'5000, I can attest that you are not only the right person to talk to, but even extremely helpful, especially when I needed something urgently. You have solved numerous problems quickly and efficiently, and I applaud you here for your CNRS award that you just won. Congrats! If there was another medal for advice on sports / mountains or teaching PhD students nordic skiing, I would highly recommend you for that one as well.

► **Olivier Richard** Unfortunately, we have never really worked together, but your funny personality makes you a great person to have around. Thanks for all the discussions, be it in the lab or on business trips!

► **Julio Toss** Thanks for helping me settle in Grenoble. Coming in, without speaking any French, was quite unsettling but you have helped me to get started here. It is very unfortunate that you had to leave in 2016 already, but I have not forgotten our trips and our endless org-mode discussions!

► **Brice Videau** Thanks a lot for your help with the calibration procedures and explaining the tools I needed to know for debugging. I hope that at some point, we can work on a joint project together!

► **Others** I would like to thank everyone else, especially Fanny Dufossé, Bruno Raffin, Denis Trystram, Frédéric Wagner but also Carmen Chan, Tristan Ezequel, Adrien Faure, Nicolas Michon, Michael Mercier, Clement Mommessin, Baptiste Pichot, Millian Poquet, Danilo Santos, Théophile Terraz and Salah Zrigui for the great time in the lab. Every single one of you has helped me in one way or another and made me laugh numerous times!

## Scientific Collaborations

► **Anne-Cécile Orgerie** Thanks for your kindness and your willingness to contribute with hands-on help and great advice on energy-related subjects! Working with you has been very pleasant for me and I hope we will continue to work together in the future.

► **Professor Martin Quinson** Thank you for hosting me for almost a month in Rennes! Not only during my stay with you, but during the entirety of my thesis, I

have enjoyed your down-to-earth mentality, your humour, kindness and certainly your helpfulness with SimGrid related issues.

► **Sascha Hunoldt** Your honesty and ambition is impressive and your comments on my paper have helped me improve it for the second (and accepted) version. I think that the scientific community needs more rigorous researchers like you. Thank you for working with me!

► **Frédéric Suter** Thank you for all the comments and help with the SimGrid code base. Your efforts and explanations have often saved me from hyperventilating!

## Friends & Family

► **My Family** I would like to thank my family and in particular my parents for their support and their help in various ways. It is highly appreciated!

► **Jan-Philipp Kayser, Alexander Kruck, Hajo Trimborn** Thank you all for your friendship that cannot be put in a few sentences and the great time we always spend together.

► **Jeffrey Overbey** All your advice, be it scientific or on grammar and writing, has been highly valuable to me. Besides being a brilliant software engineer and researcher, your company during our travels has been highly appreciated!

► **Samantha Ho** You have been there for me even during the roughest of times, and your support has helped me so much. Thank you!

## Infrastructure

► **Grid'5000** Experiments presented in this thesis were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations.

# Contents


**1 Introduction** **1**

**2 Context** **5**
2.1 High Performance Computing (Until) Today . . . . . 5
2.1.1 Scientific Applications in a Nutshell . . . . . 5
2.1.2 Architectures: Computation . . . . . 7
2.1.3 Architectures: Communication . . . . . 12
2.1.4 Programming paradigms . . . . . 14
2.2 High Performance Computing Tomorrow: Exascale Computing . . . 19
2.2.1 Applications . . . . . 20
2.2.2 Architectures: Computation . . . . . 23
2.2.3 Architectures: Communication . . . . . 28
2.2.4 Programming Paradigms . . . . . 30
2.3 Conclusion . . . . . 31

**3 Related Work** **33**
3.1 Simulation Based Performance and Energy Prediction . . . . . 33
3.2 Conclusion . . . . . 35

**4 The SimGrid Project** **37**
4.1 Overview of the SimGrid Project . . . . . 37
4.1.1 History and Impact . . . . . 37
4.1.2 Software Architecture . . . . . 38
4.1.3 Software Engineering . . . . . 40
4.2 A General Introduction to SimGrid . . . . . 41
4.2.1 Modeling Virtual Resources . . . . . 41
4.2.2 Modeling Applications . . . . . 45
4.3 SMPI: Simulating MPI Applications . . . . . 46
4.3.1 Emulation of MPI code . . . . . 47
4.3.2 Modeling of MPI Communications . . . . . 52
4.3.3 Scalability . . . . . 53
4.4 Runtime Support (StarPU-SimGrid): Simulating Dynamic GPU-based Applications . . . . . 57
4.5 Contributions to the SimGrid Project . . . . . 58

4.5.1 Platform description . . . 58
4.5.2 PAPI support . . . 59
4.5.3 Privatization . . . 59
4.5.4 Energy plugin . . . 60
4.5.5 DVFS plugin . . . 60
4.5.6 Load Balancing . . . 60

**5 Experimental Methodology 63**
5.1 Experimental Setup . . . 63
5.2 Factors Influencing the Experimental Results . . . 63
5.2.1 Hardware . . . 65
5.2.2 Date . . . 65
5.2.3 Operating System / Software Stack . . . 67
5.2.4 Kernel . . . 67
5.2.5 Application . . . 70
5.2.6 Execution . . . 70
5.2.7 Output . . . 71
5.2.8 Data Analysis . . . 71
5.3 Network Calibration . . . 72
5.3.1 Network . . . 72
5.3.2 Hardware Limitations . . . 72

**6 Contribution: Modeling Multi-Core CPUs 73**
6.1 Problem . . . 73
6.2 Proposed Solution . . . 75
6.3 Performance Evaluation / Effectiveness . . . 79
6.4 Limitations . . . 80
6.4.1 Technical limitations . . . 80
6.4.2 Scaling limitations . . . 81
6.4.3 Future Work . . . 82

**7 Contribution: Modeling Intra-Node Communications 83**
7.1 Problem . . . 83
7.2 Solution . . . 83
7.3 Performance Evaluation / Effectiveness . . . 85
7.4 Limitations . . . 85

**8 Contribution: Modeling Multi-Core CPU Power Consumption 89**
8.1 Problem . . . 89
8.2 Proposed Solution . . . 92
8.2.1 Calibrating the Energy Consumption . . . 94
8.2.2 Predicting the Energy Consumption with SimGrid . . . 96
8.3 Performance Evaluation/Effectiveness . . . 99

8.4 Use Case: Capacity Planning for HPL . . . 101
8.5 Limitations . . . 102
8.5.1 Model Limitations . . . 102
8.5.2 Experimental Limitations . . . 104
8.6 Conclusions . . . 105

**9 Contribution: Optimizing the Power Consumption With DVFS 107**
9.1 Context . . . 107
9.1.1 Iterative Applications . . . 107
9.1.2 DVFS, a Means to Reduce the Power Consumption of HPC Applications . . . 109
9.2 Related Work . . . 115
9.2.1 Adagio (Application Level DVFS) . . . 115
9.2.2 Load Balancing with Adaptive MPI (AMPI) . . . 117
9.2.3 Residual Load Imbalance . . . 119
9.3 Contributions . . . 119
9.3.1 DVFS Governor: AdagioImproved . . . 120
9.3.2 DVFS Governor: Lagrange . . . 122
9.4 Performance Evaluation/Effectiveness . . . 126
9.4.1 DVFS/Adagio/AdagioImproved/Lagrange . . . 126
9.4.2 Efficiency Comparison DVFS / Load Balancer . . . 128
9.5 Limitation and Future Work . . . 129
9.5.1 Unavailable Frequencies and More Complex Architectures . . . 129
9.5.2 Current Limitations of Our Implementation . . . 130
9.6 Conclusions . . . 131

**10 Conclusion and Future Work 133**
10.1 Thesis Summary . . . 133
10.2 Limitations . . . 134
10.2.1 Model Limitations . . . 134
10.2.2 Application Limitations . . . 136
10.2.3 Experimental Limitations . . . 136
10.3 Future Work . . . 137
10.3.1 Extending the Work of this Thesis . . . 137
10.3.2 SimGrid . . . 138
10.3.3 Joint Work with the SimGrid Userbase . . . 140

**Bibliography 141**

# 1 Introduction

Over the past few decades, High-Performance Computing (HPC) has supported a great number of scientific discoveries in both academic and industrial contexts. Today, a continuously growing number of researchers from largely different fields have come to appreciate the computational power provided by supercomputers for their studies of intricate and important questions that were often formerly infeasible, such as climate change or the molecular structure of human diseases. In the industrial world, an increasingly fierce global competition and faster innovation cycles require researchers and engineers to rely on complex simulations to test and optimize products and materials in order to gain an advantage over the competition.

Both worlds have in common that their science is often limited by the existing technology and algorithmic solutions that do not support computations at the desired level of detail. Continuing the development of machines and algorithms is therefore a necessity to provide researchers with the needed computational power. Today, the race to exascale, i.e., the next generation of supercomputers, is in full swing. Alas, to achieve a performance on the order of $1 \times 10^{18}\,\mathrm{flop/s}$ sustainably, a disruptive technology change in almost all fields, including hardware (computation units, networks), software (e.g., batch schedulers) and user applications is required. With a much greater number of computation cores, possibly in the hundreds of millions or even billions, exascale machines are expected to deliver high performance at the cost of a much higher power consumption than today's machines. For this reason, energy is expected to become the main optimization goal since power contributes greatly to the total cost of operation. This can be easily understood when considering Germany as an example: $1\,\mathrm{kW\,h}$ was priced at $0.07$ € in $2000$ but has since more than doubled and cost $0.157$ € in 2016 [Sho+17, Chapter I]. Increasing electricity prices in combination with growing machine sizes have caused power expenditures to grow incessantly. This is illustrated in Figure 1.1, which compares the yearly total electricity cost of the Leibniz-Rechenzentrum (LRZ) in Garching near Munich, Germany, to the cost incurred by its HPC system.

To reduce future spendings on electricity or at least keep increases to an acceptable minimum, efforts to improve power efficiency must be undertaken. With the abovementioned cost for $1\,\mathrm{kW\,h}$, a constant power reduction of $1\,\mathrm{MW}$ would

have saved 1,375,320 € in 2016 and even more today. It is therefore important to improve general hardware efficiency but also to reconsider how these platforms are used and programmed. Significant time is spent running tests on production platforms and moving these tests away from the machines will improve system efficiency ("science-per-watt"), but not necessarily the power consumption. It is therefore an important goal to ensure that application developers rely on algorithms that minimize power consumption, for instance by overlaping communication times with computation or by reducing the amount of data movements through improved data locality. Improving the flop/W ratio of an application is not trivial. Application developers must therefore be provided with the right tools and lightweight models that consider power efficiency [Don+11, p. 38].

One possible approach is the optimization and evaluation of applications through simulation. Once the target platform has been modeled, a simulator can estimate an application's performance on that particular machine. This can help to move away not only performance testing but also debugging, since a simulator can be expected to return deterministic results and reproducing a bug can be easier. Besides raw performance estimations, a simulator can also provide further beneficial information, for example, when and where network bottlenecks (bandwidth, congestion) occur. This can be particularly helpful with iterative developments of next-generation machines through co-design, because application developers can assess more quickly what recent developments mean for their application's performance whereas hardware developers can find out what changes (for example, to the network configuration) could speed up targeted applications and what does not (e.g., more nodes or a more expensive network topology).

This thesis contributes to the utility of simulation as a mean for building energy efficient applications, with the core contribution being a deliberately simple energy model. To test its accuracy, we implemented a prototype in SimGrid, a simulator that is particularly well suited for faithful simulation of MPI applications thanks to its well-tested network models. Our model computes the energy for every node individually and uses the load of the multi-core processors of each node as a basis for this estimation. The load changes often and the current energy consumption is therefore often recomputed over the entire execution time of the application, making precise performance predictions for applications running on multi-core systems crucial to our approach.

With a well functioning energy model in place, we study in a second contribution the potential (automatic) energy savings of adaptative MPI applications by changing a node's frequency. We implemented several, rather simple governors from the linux kernel but also a previously published application-space governor called Adagio [Rou+09]. We present also some ideas to improve Adagio and finally de-

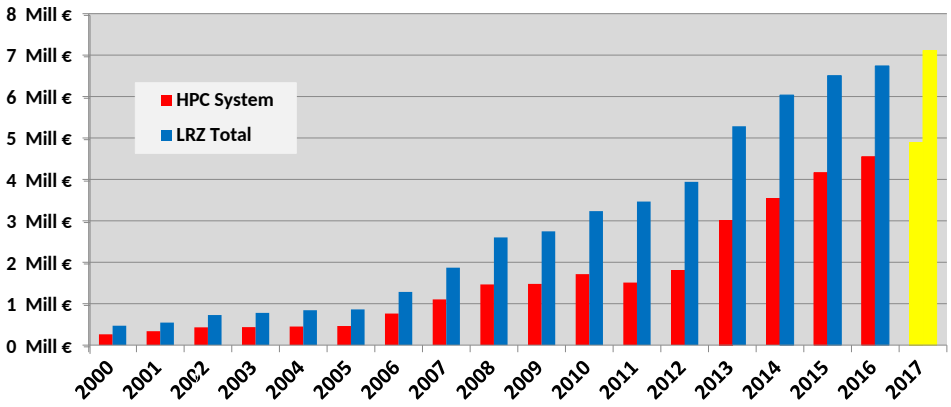


**Figure 1.1:** The energy bill for data centers has drastically increased since the early 2000s, making energy savings an important goal for hardware vendors and application developers. This statistic shows the increase of the Germany-based Leibniz-Rechenzentrum [Sho+17, Figure 1].

velop our own governor that is based on mathematical optimization. We eventually conclude our study with a comparison to load balancing, which we found to yield significant higher savings for irregular applications.

### Thesis Structure

The remainder of this thesis is structured as follows: Chapter 2 presents the current state of High-Performance Computing and the changes we need to reach a sustainable exaflop-performance. The variety of components, the cost of operating such a machine and the need of application developers to optimize their code for each platform (and energy savings) motivates the work presented in this thesis. Chapter 3 then gives an overview of the existing work, especially simulators developed by other projects. Our own simulator, SimGrid, is introduced in Chapter 4. This chapter emphasizes especially SMPI, a simulator for MPI applications that ships with SimGrid, and the issues that are inherent to "online emulation". Our methodology is outlined in Chapter 5. The first contribution, multi-core modeling for SMPI, is presented in Chapter 6. Inter-node communication has been validated many times with SMPI, but *intra-node* communication requires some additional effort. Our approach is explained in Chapter 7. Chapter 8 then finally deals with the problem of energy prediction: A key strength of our proposed model, which computes the energy consumption based on the current load, is its simplicity. Our last contribution is presented in Chapter 9 and shows how we studied energy savings obtained through dynamic changes of frequency. This thesis is then concluded by Chapter 10, which also presents future work.

# 2 Context

High-Performance Computing (HPC) platforms have been developed to satisfy the demand for immense computing power. Adopted by a wide range of scientific domains, HPC has always been subject to change but the evolution over the last decade has been particularly far-reaching.

This chapter gives a general overview of HPC and is structured as follows: Section 2.1 presents the current-state of HPC up to the petaflop-era. Subsequently, Section 2.2 contrasts this with new requirements and developments for the next-generation of HPC (called exascale).

## 2.1 High Performance Computing (Until) Today

### 2.1.1 Scientific Applications in a Nutshell

Not even half a century ago, computational cost caused many scientific problems to be impenetrable, but the advent of computers has given researchers a new instrument that has rendered many of these problems tractable, even though especially the most demanding and challenging problems are limited to the most powerful machines, and hence a small subset of all platforms, because only they can can provide enough resources to satisfy these applications' requirements.

Unsurprisingly, the HPC community consists of researchers from both academia and industry and covers a large variety of fields, such as engineering (e.g., aerodynamics in aviation, material science), geology (oil exploration, earthquakes), meteorology (weather forecasts), medicine (e.g., HIV, brain simulation), biology (e.g., drought-resistance in plants, protein docking), or globally relevant questions such as climate change.

These fields of interest contain profoundly different scientific problems that require computational power, but to investigate them, scientists use very similar approaches: They commonly rely on simulations that model parts of reality with the aid of discretized partial differential equations.

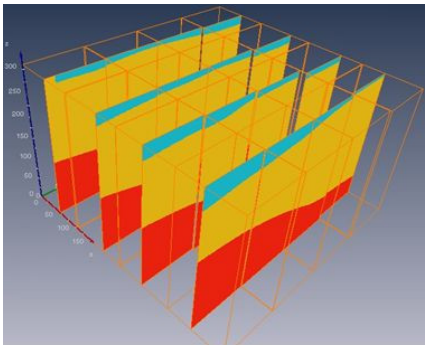

**Figure 2.1:** A domain decomposition of rocky ground used for an earthquake simulation with Ondes3D. The physical domain is cut into cuboids which are evenly assigned to processors.

As a consequence, the general structure of HPC applications is mostly regular: The physical domain (i.e., the object of interest), known at the start, is first discretized, i.e., decomposed into a finite amount of smaller elements with an easier geometric shape (cubes, rectangles, ...). Figure 2.1 exemplifies this process with a decomposition into cuboids for a geological problem. Each cuboid is subsequently assigned to a single processor. In a second step, each thus obtained element (part of the initial problem) must be initialized, for instance from observations (e.g., geological composition, depicted in different colors in Figure 2.1) or a random field. In the final step, the simulation is started and computations are applied on a per-element basis.

It is important to note that the resources required by steps two and three depend on the application and the decomposition's resolution. Consider a "relatively" coarse-grained $1000 \times 1000 \times 1000$ decomposition resolution, with each element containing only five double values. Clearly, $5 \times 10^9$ values would have to be allocated and initialized, consuming a total of $40$ GB of memory. A refined resolution is normally applied to all dimensions and so doubling the resolution will require $8 \times 40 = 320\,\mathrm{GB}$ to be allocated.

The resource consumption can therefore explode easily. To mitigate this, the entire domain is typically distributed over several machines (interconnected by a network). With each machine storing just the fraction of the domain that it processes, regular communication between machines in order to exchange data (such as computation results) or for synchronization (if required) becomes inevitable.

The following discussion gives an overview on current HPC node hardware (Section 2.1.2), network setup (Section 2.1.3) and the main programming paradigms for HPC programming (Section 2.1.4).

### 2.1.2 Architectures: Computation

The scale of supercomputers with hundreds or thousands of nodes is too large to be bought by single research entities such as research teams or university departments, even though these machines are normally already built with off-the-shelf components for cost reasons. In most cases, government organizations such as the NSF or the DoE in the U.S. finance the majority of the expenses required for acquisition and maintenance of a supercomputer. Once the production phase has been entered, access to the supercomputer is normally granted to selected researchers after evaluation of their proposed project.

More often than not, HPC nodes are constructed to provide general computational power to as many disciplines as possible and hence do not favor a specific kind of computation over others. In some rare cases, however, supercomputers are explicitly procured for and restricted to a very limited research field (such as daily weather simulations). This restriction implies that application requirements are rather well known in advance and specialization (leading to, e.g., increased performance or lower energy consumption) can thus be justified.

The three generic computational units (CPU, GPU and Many-Core-Coprocessors) found today in HPC nodes are presented in the following discussion.

#### CPU

CPUs are considered the "heart" of a computer and can be found in every PC, laptop or server. Their development is a very costly undertaking and requires high-tech knowledge in many fields, such as circuit design and manufacturing.

It is hence not surprising that most HPC systems rely on the well-known x86 architecture, as produced by Intel and AMD, for their main processors.

For several decades, processor frequency used to double every 18-24 months. Alas, a processor's power consumption grows quadratically with the frequency and as a consequence, even limited clock speed increments quickly result in higher heat dissipation and require more costly cooling to prevent damage. Vendors have for this reason turned towards multiplying computation units, e.g., by adding several CPUs to a node or adding more cores to a CPU. Having multiple computation units in the same node means that they share the same address space and can access an application's data that is stored in the system's main memory (RAM) directly, without further transfer over a network.

Sharing the main memory between CPUs creates several side-effects: First of all, transferring data from the main memory is very slow and often forces the CPU to wait. To mitigate this, CPUs try to reduce RAM accesses by using a cache hierarchy with different cache sizes and performances per level. Figure 2.2 (page 9) illustrates the cache hierarchies of a commodity laptop bought in 2015 and Figure 2.3 (page 10) a recent HPC node. One can distinguish in total 4 levels of caches:
The first level (L1) cache, exclusive to each core, is typically very fast but also very small with only a few kB. The L1 cache exists for instructions (L1i) and for data (L1d). The second level (L2) cache is still private to each core but its increased size comes at a lower performance. While the L1 caches feature the same size for laptop and HPC node, the L2 cache is already four times larger for each HPC-core than for the laptop. Finally, the third level (L3) cache is shared between all cores of the same CPU and can store several MB of data. The HPC node has an overall larger L3 cache per CPU (22 MB vs. 8 MB), but at the same time it needs to serve in total 16 CPU cores and hence four times more than the corresponding laptop cache. It is thus essential to keep this hierarchy when programming applications. Ensuring both spatial and temporal locality helps to avoid costly cache misses.

All cores can access the main memory to read and write data independently of each other. Therefore, data coherency must be guaranteed among all cores to avoid computations with different values for the same variable. Updates to data stored in RAM must hence trigger an invalidation of any corresponding cached copy. This means that in the worst case, when all cores have a copy in their caches, writing a single value back to RAM requires to broadcast the invalidation to all cache levels of all cores. Consequently, CPU core count scalability is adversely impacted by larger caches and most CPUs have therefore relatively few cores but very large caches, even though the design of their hierarchy (including number and size of caches) can vary.

Privileging caches over main memory greatly contributes to application performance. Over the years, many intricate techniques that help with this have been developed. Cache-prefetching, for instance, reduces the amount of direct memory accesses and cache-misses by intelligently loading data from the main memory into caches before it is actually required.

Unfortunately, even with good cache usage, waiting times for the processor can still occur, for example, when a dependency on a not yet available result of another operation exists.

Instruction Level Parallelism (ILP) aims to reduce waiting times by identifying instructions that can be executed in parallel. Some ILP-based techniques are instruction pipelining, out-of-order execution or speculative execution. The introduction

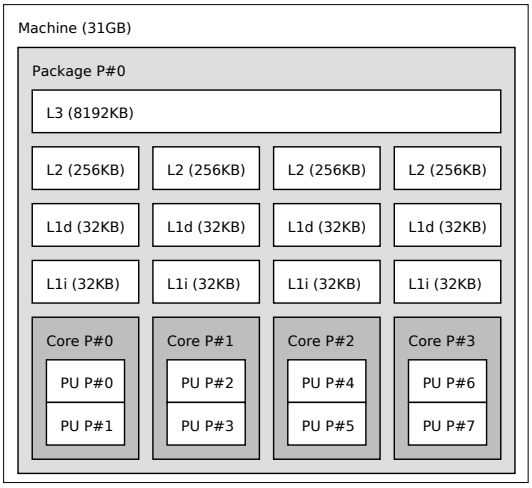


**Figure 2.2:** A visual representation of the cache hierarchy and sizes of a workstation laptop as obtained through hwloc-ls.

of HyperThreading has optimized this even further so that significant speed-ups in the future are not to be expected.

Besides the instruction level, the data-level can be exploited for further parallelism, typically through vectorization. The idea is that a single instruction (such as addition or multiplication) must be applied to all vector coordinates. Several extensions of the x86-architecture (e.g., legacy MMX, multiple versions of SSE and the new standard Advanced Vector Extensions (AVX) and its different versions), introduced supplementary registers and operations. The AVX-512 extension provides $32\,\mathrm{registers}$ with $16\,\mathrm{bit}$ each. These can be used to hold $8\,\mathrm{single}$ precision or $4\,\mathrm{double}$ precision floating points. To take full advantage of vectorization, the instruction (the operator to be applied) as well as the data must be identified. In some cases, a compiler may be able to automatically determine the instructions and data but data-dependencies are difficult to resolve with certainty, so a compiler may or may not vectorize code correctly. In practice, this has largely remained the programmer's responsibility.

Vectorization is an important performance optimization for scientific applications, as they are well-known to be often regular and therefore can profit frequently from good vectorization.

A CPU's performance is improved by several levels of parallelism (multi-CPU, multi-core, instruction and data-level) and many on-chip features. CPUs are also built to be backward compatible with legacy software (e.g., the new AVX is backwards compatible to SSE), so removing a once introduced feature is difficult.

CPUs have thus a very general nature that makes them well-suited for problems that are not massively parallel.

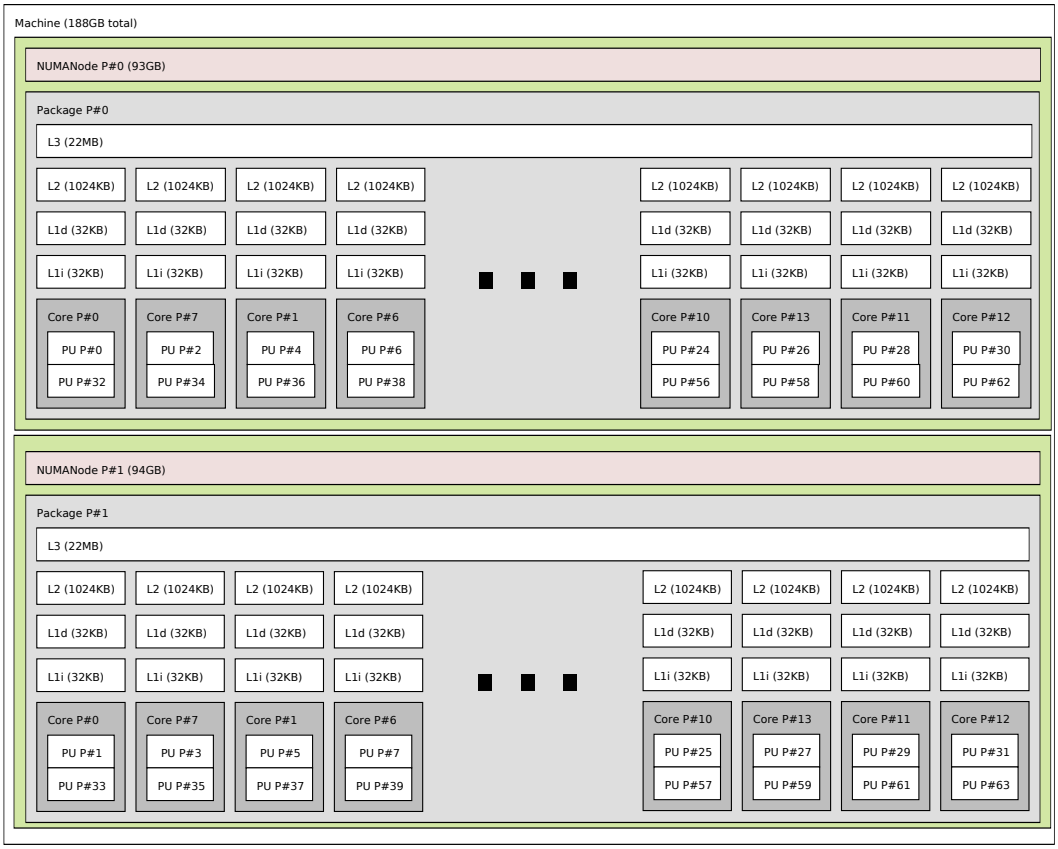


**Figure 2.3:** A visual representation of the cache hierarchy and sizes of a node as obtained through hwloc-ls.

The following section contrasts CPUs with GPUs, devices that are highly specialized on supporting massive parallelism.

### GPU

The hardware-implementation of the large x86 instruction set and necessary optimizations (e.g., ILP) consume power and space on the die for its transistors. This makes CPUs very complex and forces them to dedicate about 70 % of the space (and hence energy) to hardware-internal decoding of the control flow.

For signifcant performance gains, this share needs to be reduced as much as possible, for instance by removing unnecessary features. As a side-effect, space is released on the die since each core shrinks in size, making it possible to fill in more compute cores. For many general-case applications, the removal of features such as ILP comes at a loss of performance. On the other hand, applications that satisfy some assumptions (e.g., that they are very regular and do not require ILP so much) will receive a large performance boost.

In these special cases, it is hence a good idea to use simplified hardware with more compute units for improved performance. GPUs, originally developed to render 3D scenes in video games, have a lean implementation, a commodity hardware and are massively parallel. Consider for instance Nvidia's flagship GPGPU at the time of writing, the Tesla V100, that comes with $5120\,\mathrm{cores}$.

Unsurprisingly, as GPUs are especially powerful vector machines, an application that is vectorized easily will profit from executing on a GPU as they are expert in rapidly executing the same operation over and over again. Alas, the execution itself is not as seamless as for CPUs, because a GPU does not execute the binary itself. Instead, a programmer needs to use OpenCL [Inc] or CUDA [NVI] to start threads on the device and to transfer all required data in and out. Furthermore, data locality is very important as caches are small (relative to the number of cores) and loading data is hence expensive. Programming GPUs is discussed in more details in Section 2.1.4.

Regardless of elevated requirements for programming (that possibly necessitate extensive and expensive (re-)training), imposed restrictions regarding suitable applications and even high energy consumption (up to $300\,\mathrm{W\,s}$), GPUs have recently become a common addition to HPC nodes as they are cheap, fast and significantly more energy efficient (when using $\mathrm{flop/W}$ as a metric) than CPUs.

#### Many-Core Chips

*Many Integrated Core* (MIC) accelerator cards are the third option and provide a middle way between the general but slow and energy inefficient CPU and the fast but highly specialized GPU. MICs are (like GPUs) add-on chips that communicate over specific busses (e.g., bandwidth-limited PCIe) and they come with their own memory on the chip. They were also widely considered a viable alternative to GPUs after Intel's well-known *Xeon Phi* was introduced. The Xeon Phi in particular is presented here, since it has been the most popular MIC for a long time, even though the Phi add-on model line has been abandoned today.

The Xeon Phis were the first to support the new vectorization standard AVX-512, implemented for all $48$ to $72$ cores with $4\,\mathrm{threads}$ per core, making up to $288\,\mathrm{logical}$ cores available. This is significantly less than the thousands of particularly floating-point oriented cores used by GPUs, but as a trade-off, each MIC core is capable of dealing with control statements much more efficiently.

Unlike GPUs, the Phis run their own Linux-based micro operating system on the chip and must hence be booted. This combined with their support for the x86 architecture allows them to execute binaries directly on the chip. This enables users also to launch individual processes on each core, converting the Phi effectively into a mini-cluster. As another side-effect, this also lowers the entry barrier significantly as applications using the accelerator only need to be written in a commonly used language (plain C, C++ or Fortran) and a specialized compiler (e.g., provided by Intel) will further help the user. Although this is certainly helpful to obtain quick performance boosts, exploiting the chips capacity entirely requires programmers to prepare / rewrite the code in a way that the compiler can identify every optimization potential. It can generally be said that (similar to GPUs) MICs work great with applications that have already been optimized for vectorization.

#### Conclusion

With three very different types of computation units (CPU, GPU, MIC), each exhibiting its own characteristics and advantages, machines can be configured very differently. System vendors tend to add accelerators to more machines and so it is not surprising that the November 2018 edition of the Top500 lists `138 (27.6%)` machines as using accelerators.

### 2.1.3 Architectures: Communication

#### Network Technology

Many applications implement intensive communication patterns between processes launched on the same or different nodes in order to send/receive data that is required for further computation or to simply synchronize with others. Messages containing data have no upper size boundary and can reach several GB. For large amounts of data, the transfer time is predominantly determined by the bandwidth, which must hence be large enough. On the other hand, for very small messages (e.g., for synchronization or updates to a few variables), bandwidth is significantly less important as the total transfer time will be very small. Instead, the speed is mostly determined by the network's latency.

In environments such as HPC machines, where distance is relatively small and all routers and switches are under the full control of the data center, latency is mainly influenced by the technology used. Today, it is dominated by Gigabit Ethernet and Infiniband, although some other competitors such as Intel's Omnipath and custom

interconnects exist. Ethernet is used in private networks at home or in professional environments and therefore is a non HPC specific, very low cost solution that uses the entire TCP/IP stack (and all of the overhead it comes with) and can currently reach bandwidths up to $400\,\mathrm{Gbit/s}$. On the other hand, Infiniband and OmniPath are proprietary solutions that are specific to HPC and significantly more expensive. What makes them interesting to the HPC community is that the network stack has been reduced by removing parts from the TCP/IP stack that are useful in dynamic and large-scale networks such as the internet but that are not required for a well-controlled or (comparatively) simple environment such as a HPC machine. Those parts include loss mechanisms, flow control and routing. Other parts, including 0-copy RDMA and deterministic routing have been specialized.

In some cases (e.g., for Cray machines), when the enormous cost can economically be justified, a custom-tailored interconnect is employed to reduce the latency even further.

Choosing one of these software/hardware stacks also means to a trade-off between cost and performance / power consumption.

### Topology

Even for small-scale clusters, fully connected networks (Figure 2.4 (a)) are infeasible due to quadratical growth of required cables and ports. (For $n$ nodes, $\frac{n\cdot(n-1)}{2}$ cables and $n-1$ ports per node are required.) Instead, nodes are connected through intermediate routers / switches and form a physical topology. Over the years, many topologies have been used in HPC, some of which are illustrated in Figure 2.4, including meshes (c), multidimensional tori (d), hypercubes (e), fat-trees (f) and clos-networks. Researchers evaluate a topology by looking at several properties, such as simple and very fast routing algorithms (i.e., quick computation of the shortest path between two nodes), bisection bandwidth, diameter, scalability and cost. When setting up a physical network, it is important to find a good balance between these properties to guarantee good performance for as many types of applications as possible, since changing the topology is either impossible or very difficult as this requires physical reconfiguration. The best topology for a given application often depends on its implemented communication patterns and may not be the perfect fit for other applications.

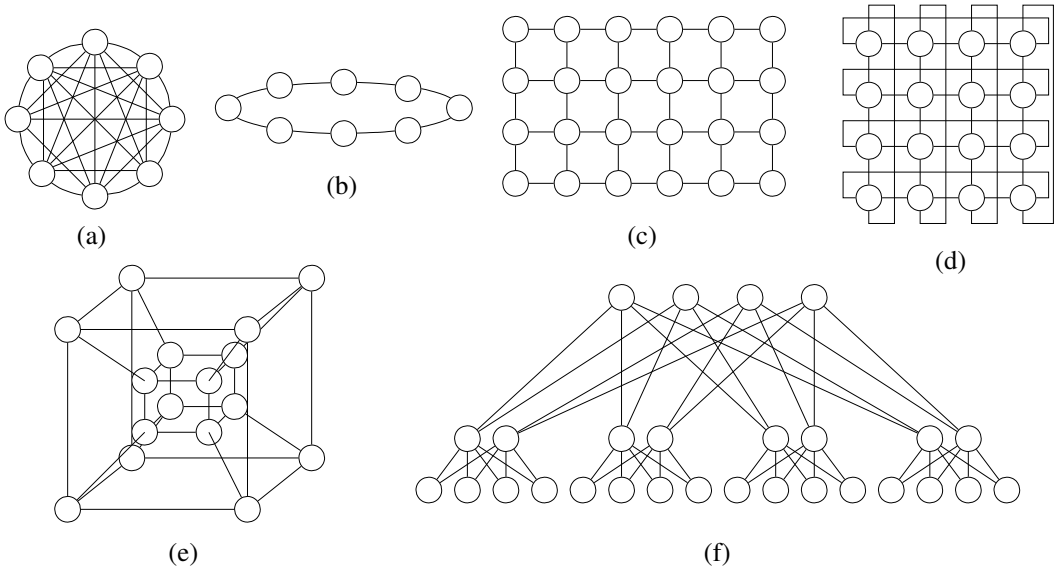


**Figure 2.4:** Several classic network topologies. Shown are: (a) fully connected network, (b) ring, (c) mesh, (d) torus, (e) hypercube and (f) fat-tree [CLR08, Figure 3.1].

#### Observations

An application's performance certainly depends on its own communication patterns but is also impacted by network technology and topology. The former is generally under control of the researcher whereas the latter is provided by the machine vendor and normally cannot be re-configured arbitrarily.

The multitude of technologies makes it necessary to determine the best choice by looking at the mix of applications that are supposed to use the machine. Unfortunately, since there are so many tunable parameters, evaluating / predicting an application's performance on a given network is rather challenging.

### 2.1.4 Programming paradigms

Naturally, scientists want their applications to run even faster and increasing the parallelism is often a promising approach to achieve this. To support this parallelism, more compute units need to be added and hardware needs to be updated, for instance by moving to heterogeneous nodes (i.e., CPU + accelerator).

Efficient usage of heterogeneous machines with several interconnected nodes, each possibly containing one or more multi-core CPUs and other specialized components such as GPUs, MICs etc., is a difficult undertaking that often requires profound knowledge of the underlying hardware and problem. Consider a simple matrix-multiplication: Moving it from the main CPU to an accelerator may speed-up the

*algorithm* itself; however, the data must be moved to the accelerator first, causing an overhead. For somewhat small matrices, this may overall result in a slow-down.

The rapid innovation and development cycles of hardware make it very difficult if not impossible for manpower-limited research-teams to implement proper support for each type of hardware. But also the software itself may pose problems: A software's architecture may have been designed years ago, without now emerging technologies in mind. It may hence require significant refactoring efforts to even implement basic support for massive parallelism.

Furthermore, programmers do not only want to exploit the parallelism that is today available *intra-node*, they also want to use interconnected nodes to distribute computations across several nodes (*inter-node*). It is common to rely on different paradigms that aid programmers with these two tasks.

#### Inter-node

For a given problem, sequential algorithms are often already known or easier to come up with than parallel algorithms. A natural approach for the development of a parallel algorithm is hence to parallelize the sequential algorithm by distributing data among all nodes so that every node can start computing as quickly and as much as possible. When the computation requires data that is only available on other nodes, inter-node communication based on message passing can be used to retrieve this data.

The Message Passing Interface (MPI) standard [For] has been developed by the MPI-Forum for this purpose. It is freely available in its most recent version 3.1 at the time of writing. Open source implementations are actively developed and released by several projects, such as Open MPI [Gab+04], MVAPICH [LWP04] and MPICH [Gro02], but proprietary implementations exist as well, e.g., IntelMPI.

MPI abstracts the complexity of the network by providing a well-defined API to the user that includes standard send/recv operations, numerous collective communication operations (broadcast, all-to-all, ...) but also advanced features such as buffered sends, 0-copy RDMA etc., while making little to no comments on *how* to concretely implement them.

Another great benefit of MPI are so-called *virtual topologies*. These (logical) topologies allow algorithm designers and programmers to define a notion of relationship (neighborhood) among processes. This is not inherently necessary, because every

process can always send a message to every other process, but it permits algorithm designers to structure the program more cleanly, giving them a better view of the communication patterns. Programmers, on the other hand, can implement the computation patterns more easily. Rank ids have no longer to be cumbersomely calculated by (for example) their position in the network but all communication partners are stored in the topology. Collective communications can also be applied to virtual topologies and can therefore be more readily used, leading to a more high-level programming style. Structuring the communication relieves the network and hence improves the overall performance as well.

Lastly, the runtime itself may benefit from declaring virtual topologies as this can help with mapping the processes to the computational units on all nodes.

### Intra-node

Multi-core CPUs are now commonly found in each node and can be exploited by a parallel application. All cores provide unified (but not necessarily uniform) access to the same main memory whereas their caches are (on some levels) private and shared on others.

An application can take advantage of a multi-core CPU in several ways: Firstly, if the application is already parallelized with MPI, one or more MPI-processes can be started on each of the cores. However, processes do not share the same memory address space and consequently, making data from one process available to another may incur additional copies / transfers even though the data is available in a cache that both processes can access. This remains even true if the values are already stored in the memory.

Secondly, if the application is fully sequential and re-writing is not an option, developers can substitute libraries with existing multi-threaded versions (e.g., Intel's Math Kernel Library (MKL) for BLAS algorithms).

Finally, the application can be (re-)written using threads, but this is cumbersome and error-prone due to low-level programming. A more high-level approach is to parallelize using the OpenMP (Open Multi-Processing) standard's `#pragma` annotations for C, C++ and Fortran, which is natively supported by some compilers, for example GCC since version 5 Unlike MPI, OpenMP relies exclusively on threads and creates and manages threads for the user. Alas, good performance results can often only be obtained after thorough tuning of the application and parameters.

Unlike MPI, OpenMP can only be used to parallelize an application on the same node as there is no notion of data transfer. Still, data can be shared between threads directly through the memory, as their address space is shared. For this reason, MPI can (theoretically) replace OpenMP but not vice versa. Nevertheless, with their concepts being very different, it is difficult to know whether MPI performs better than OpenMP on a particular platform. Many scientific application developers have therefore resorted to maintaining an MPI-only and an MPI+OpenMP version. In the latter case, it is a common practice to launch one MPI process per socket which in turn starts OpenMP threads on every core of that socket.

#### GPU programming

It was already mentioned in the last section that multi-core CPUs can be exploited without any modifications to the code by using multi-threaded libraries, such as MKL. This is also true for GPUs, if the library in question supports GPU offloading. Otherwise, substantial development needs to be done in order to incorporate the data transfer from and to the GPU as well as kernels that can be executed on the GPU's cores. Since Nvidia is the most common supplier for high-performance GPUs, this is most commonly done through its proprietary API called *CUDA* that integrates with C, C++ and Fortran, although many other languages are also supported through third-party wrappers. CUDA only supports Nvidia GPUs, so the Open Compute Language (OpenCL), which comes with the benefit of additional support for other devices, is a viable alternative that is already used by simulations as this portability allows applications to run also on machines with other devices, AMD GPUs.

As a consequence, it is the programmer's responsibility to handle communication with other nodes through MPI and node-internal communication with GPUs via CUDA. Furthermore, exploiting the massive parallelism of a GPU is not an easy feat, as it requires first of all to keep the load high enough on each node to occupy all GPU cores. Secondly, this data must also be granular enough to assign a piece of work to every core. Further tuning options exist that are known to be important for performance, for instance the number of threads (for oversubscription) or streams.

Scientists from application domains such as physics and biology are often not familiar with these new concepts and have to undergo further training before making modifications to their applications. Since it is difficult to predict the outcome of their modifications, implementing accelerator support in their application may be postponed as long as possible.

### MIC

Older code-bases can often exploit MICs without rewriting the code through linked libraries, similar to GPUs and multi-core CPUs. MKL, developed by Intel, is used for BLAS functionality and supports exploitation of many-core chipsets for several (but not all) operations. However, not all supported operations, such as matrix-multiplication, are always executed on the accelerator as MKL determines whether the parameter size is large enough to offset the penalty for transferring the data to the MIC.

As mentioned before, a MIC can also be used as a mini-cluster by directly launching MPI processes on its cores, even though generally with very limited amounts of RAM available (with a few GB of high-bandwidth RAM normally significantly less but much faster RAM than the host node). As a consequence, computations requiring massive amounts of data cannot be executed on the MIC without transferring data from and to the host's main memory (or other nodes). Messages from and to processes outside of the MIC must pass through the PCIe bus and this extra traversal increases latency. Although this is negligible for large messages, it can add significantly to the time it takes a small message to arrive. With bandwidth being limited as well, however, the PCIe bus can become a bottleneck when all cores are sending large (i.e., bandwidth-intensive) messages.

Running MPI on MICs directly therefore causes a topology that has largely differing cost for sending messages: Sending messages to other processes running on the same MIC is significantly less expensive than using first the bus and then the network. Furthermore, due to the little on-card memory, sending large messages must take into account whether the receiver is on a MIC and can actually receive (store) this message.

With all these constraints, it is in practice rather uncommon for MICs to be a plug-in solution. Like GPUs, programmers need to consider MICs and their properties when writing code that is supposed to exploit a MICs capabilities completely. This implies that significant effort for rewriting an application may be required.

### Conclusion

Programmers want to make an optimal choice so that their application can be executed at optimal performance. Unfortunately, the multitude of libraries and programming frameworks on the one hand side and hardware on the other hand

makes it very difficult to predict in advance what performance can be expected from a choosen software/hardware stack.

## 2.2 High Performance Computing Tomorrow: Exascale Computing

We have seen in Section 2.1 the components the HPC community relies on to build machines that can operate sustainably at the petascale ($1 \times 10^{15}$ operations per second) level. Scientists can exploit this immense computational power in many ways, e.g., by adding more detail to their scientific model or by increasing the decomposition's resolution. Currently, scientists are still limited by technology and by improving the machines further, questions that cannot be resolved at this time become feasible. For this reason, the HPC community races to develop exascale machines ($1 \times 10^{18}$ operations per second) but at the same time, several issues thwart quick development. First and foremost, the physical size of today's fastest machines already fills entire machine rooms and so increasing a machine's performance by adding $10$ times more nodes cannot work due to space constraints. The second problem is that the energy consumption of an exascale machine with possibly billions of computation cores would be infeasible if no measures to reduce the consumption are taken. Initially, the U.S. Department of Energy proposed the threshold for exascale systems to be at most $25\,\mathrm{MW}$ for financial and political reasons [Don+11, p. 4], but environmental and practical reasons (e.g., redundant power supply, cost for diesel or battery aggregates, ...) certainly exist as well. Today, this threshold is expected to be set more realistically around $30\,\mathrm{MW}$ to $40\,\mathrm{MW}$. Thirdly, all previously presented issues persist and become even more challenging for exascale: Rigid applications will not scale, heterogeneous nodes will be more difficult to manage as they will exhibit even more massive parallelism and the network will experience unprecedented levels of contention due to message exchanges and largely increased amounts of data.

Profound technology changes are necessary to move from (sub-)petascale to exascale. It has been estimated that this current overhaul of technology is comparable in disruptiveness to the major transition in the 90's that saw vector computing replaced with parallel computing [Don16, Section 4.2.4].

The following sections follow the same pattern as the above discussion. First, applications are discussed in Section 2.2.1, followed by computation units in Section 2.2.2. Communication and networks are discussed in Section 2.2.3 and, finally, programming for exascale is presented in Section 2.2.4.

### 2.2.1 Applications

For sustained exascale performance, developers will be forced to take on a more active role when it comes to managing and optimizing an applications execution. New responsibilities include now load balancing and resilience design. Co-design, on the other hand, is an important paradigm that will be required to be applied during development of the machine and application for successful exascale performance.

#### Load Balancing

Applications running on petascale machines already have to deal with load balancing to obtain maximum throughput. Unfortunately, it is not always clear *why* a load imbalance exists. Processes/threads could be idle waiting for data from another process, maybe because the network exhibits a bottleneck and is overly congested, or have just fewer computations to do. In the future, higher core- and thread-counts (expected to reach even billions [Don+11, Section 3.1]), required for exascale performance, will exacerbate this problem.

Load imbalance is rarely easy to fix through modifications to the source code, as it is often inherent to applications, both regular and irregular, and amplified by hardware variability.

In the case of regular applications, algorithm designers frequently begin with sequential algorithms that are then parallelized, requiring large synchronizations through collective communications. Nowadays prevalent complex node architectures (e.g., CPU + accelerator) can be the cause as well since managing all important aspects (e.g., parameters such as block size) can be difficult using only MPI + OpenMP.

For irregular applications that mix different models (e.g., different gasses in turbulent combustion or changing layers of rock/soil in geology), load balance deteriorates even more: Physical effects depend on local conditions and exhibit different behavior in non-uniform conditions. For example, when simulating combustion, turbulent conditions cause local variations in the fuel-air ratio which causes the burning velocity to change locally. This, in turn, causes cellular flames (depicted in Figure 2.5), a highly irregular state. This irregularity explains why these applications seemingly can never have enough compute power: in $2011$, a time frame as small as $1.9\,\mathrm{ms}$ of methane-combustion in a gas-turbine was simulated using MPI and took over $1.3 \times 10^6$ million core hours (using a total of $16\,384$ cores) [MDV11, p. 1346].

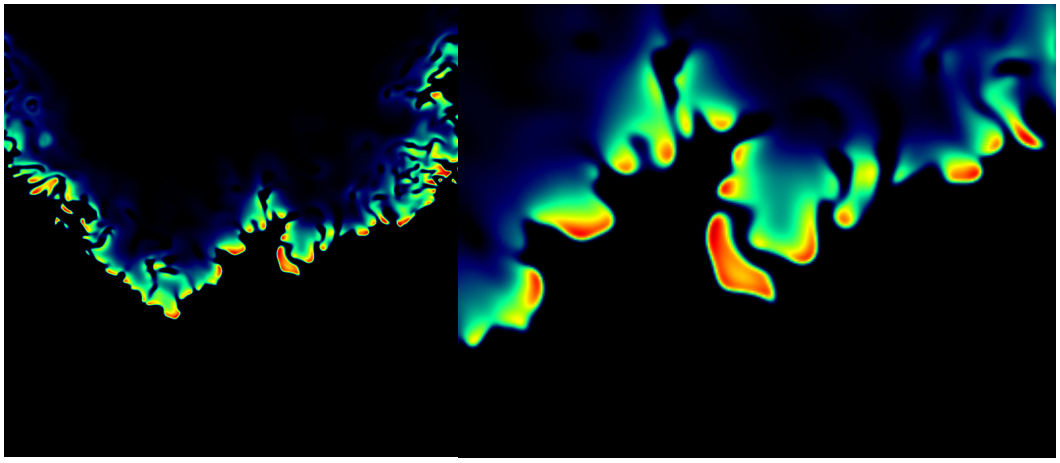

**Figure 2.5:** An illustration of (simulated) cellular flames where temperatures differ locally due to differences in burning velocity. The right picture highlights that parts of the flame can have highly irregular shapes and can be detached from the rest [Day+09].

Even with a thousand times more computational power than petascale machines, exascale machines will still not be powerful enough for many of today's scientific questions as time complexity increases exponentially with the time and space order of the underlying physics. The full simulation of a gas turbine combustor, for instance, is expected no earlier than 2045 and only if machine floprates continue to grow linearly.

With the immense cost of running a simulation at the exascale, many scientists see these platforms mostly as a means to run hundreds of *petascale* simulations at the same time instead of running one simulation at the exascale.

### Resilience

Tens of thousands of nodes, hundreds of millions of cores and billions of threads lead to a low mean-time between failures for exascale machines, maybe even as low as 30 min [Cap+14, Section 2.2]. Each of these may have various and unforeseeable reasons and are not limited to any level of the stack. They may occur at the data center level (e.g., fatal outages), hardware level (e.g., hardware failure), or even bugs on the software level (e.g., operating system, support libraries, ...). Each failure can potentially be catastrophic for the execution of a job in the sense that the application needs to restart completely, implying full loss of already achieved progress. It is therefore desireable to implement strategies that make an application resilient against failures. This allows an application to terminate correctly, even though intermittent failures occurred.

A popular approach to mitigate full loss is to checkpoint(-restart) the application [Cap+14, Section 4.1.1]. A snapshot of the application state is saved periodically and once a failure has occurred, the application is interrupted and reset to the state saved in the checkpoint. There is still some loss, but it is reduced significantly. To reduce the great cost (e.g., no unreceived messages at the time of checkpointing, heavy IO traffic, ...) it comes with, many variations of this approach have been proposed. In fact, checkpoint-restarting can be so expensive that it can be cheaper to use replication.

Other issues such as memory corruption are more serious, impossible to evade and difficult to recognize, as they occur silently. They are also difficult to recover from, since the corrupt data may not be restorable. Consider high-energy neutrons, stemming from cosmic radiation: They are known culprits for apparently random flips of bits in RAM when sufficiently small circuits are used [Cap+14, Section 2.1]. An application that uses numerically robust algorithms may ignore or even restore corrupt values within a data vector (e.g., a column of a matrix), but control variables such as loop counters are more difficult to deal with and applications are especially vulnerable in this regard. An application may hence crash on a node or enter a deadlock state when a control variable is corrupted.

The MPI standard and consequently most scientific applications are still not ready for fault-tolerance. Progress is stifled by the fact that a best practice has not yet been found; each class of applications seems to work best with a different approach. Plus, a half-baked standardization attempt is feared to come with a performance degradation that is unacceptable to many users or vendors. Significant efforts have been made, however, and ULFM (User Level Fault Mitigation) [Bla+13] is a proposal developed by the MPI Forum's "Fault Tolerance Working Group" that aims to ensure that node failures can no longer cause MPI calls to crash or deadlock the application by waiting for messages from the dead node indefinitely.

Other approaches to save data from being lost include storage of (important) data in multiple locations (possibly with varying levels of quality/redundancy) and algorithm-based fault tolerance (ABFT). ABFT exploits properties of the underlying computational algorithm in a way that faulty data can be restored, e.g., through redundant or supplementary data and further knowledge of the computational structure. This approach is quite limiting with respect to the algorithm design and does in general not work for every application.

Significantly more research is still required as it remains unclear what the most usable yet conservative approach is. Breakthroughs are therefore not to be expected. Instead, small improvements and adoption by the user base over time is expected to help.

#### Co-Design

In the past, vendors developed machines that could be easily deployed generically on several sites. For sustainable performance at exascale, decisions on hardware design must consider relevant scientific applications and their potential of exploiting the platforms computational power while at the same time application developers (and possibly algorithm designers) need to know about best practices but also idiosyncracies of platforms. Bringing these different groups together, i.e., scientists investigating the scientific question, software architects building the application and hardware architects building the platform, allows each group to benefit from an improved feedback-loop for their own work and test / influence the work of others, during (and not after) the construction of the machine.

This paradigm, called "co-design", fundamentally changes how machines are constructed: They are built around applications, rather than forcing applications to merely "fit in". Taking account of these benefits, IBM installed two "centers of excellence" at Lawrence Livermore National Laboratory and Oak Ridge National Laboratory that bring application and hardware developers closer together [Ead16]. This collaboration can be expected to give rise to new and more specialized hardware.

### 2.2.2 Architectures: Computation

The computational power of machines has continuously and rapidly grown over the last decade, see Figure 2.6. In June 2005, more than $100\,\mathrm{Tflop/s}$ were achieved for the first time by the fastest system, powered by $65\,536$ cores and consuming a total of $716\,\mathrm{kW\,h}$. In November 2018, the fastest system contained (with a total of $2\,397\,824$ ) more than $30$ times more cores and had a 1094 times higher theoretical peak performance at $200\,794.9\,\mathrm{Tflop/s}$ while the power consumption increased by a factor of 14 to $9783\,\mathrm{kW}$. [1] This means that this machine consumes *per hour* almost twice the power consumption a german household with at least $3$ persons had over the entire year in $2017$, namely $5000\,\mathrm{kW\,h}$ [Off17].

Clearly, this is an enormous energy consumption and must not scale proportionally with increases in performance. With $20\,\mathrm{MW}$ to $25\,\mathrm{MW}$ being the target for power consumption [Don+11, p. 4], exascale machine must be able to perform *at least* $50\,\mathrm{Gflop/W}$. With a power cap of twice this limit, i.e., $40\,\mathrm{MW}$, the performance must be at least $25\,\mathrm{Gflop/W}$. In 2018, Nvidia's flagship GPU, the Tesla V100, delivered

[1] November 2018 list at www.top500.org

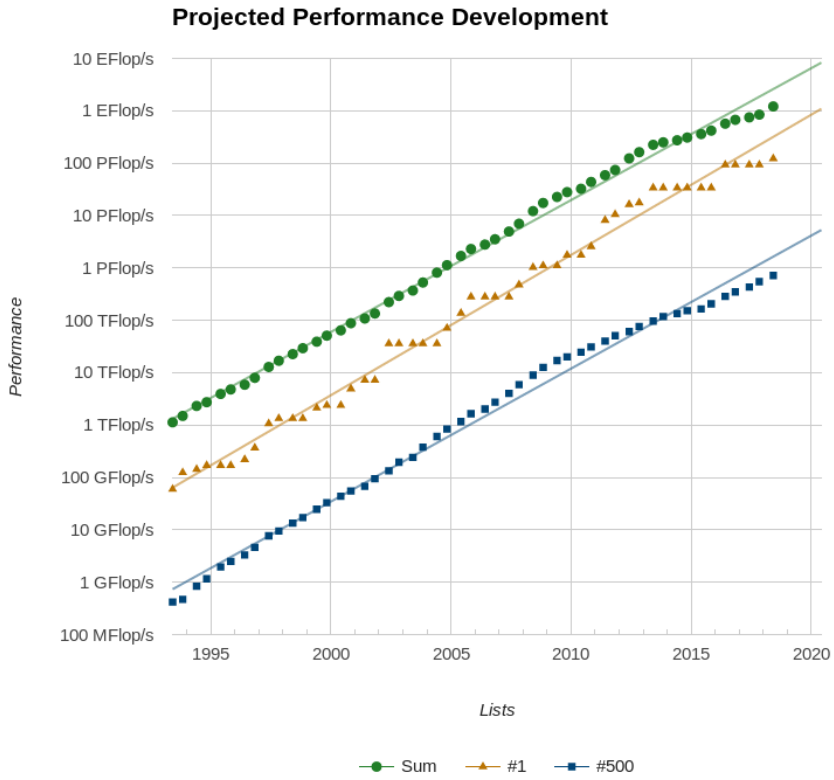


**Figure 2.6:** Development of the Top500 list over time and extrapolation for the near future. From www.top500.org

up to 7.8 Tflop/s (double precision) with a consumption of 300 W s, i.e., it delivers up to 26 Gflop/W.

A twofold increase in energy efficiency is therefore still required and currently popular computation nodes with a CPU/GPU pair must be revised for further energy savings, e.g., by using other (more energy efficient) types of accelerators.

### CPU

We've already seen in Section 2.2.2 that CPUs are mainly well suited for problems that are not massively parallel. For exascale machines, several concepts exist that enhance the CPU's usability. First of all, since the clock rates are no longer getting faster and the transistor size is shrinking, the parallelism can be increased, for instance by adding more vector registers to each core, more cores to a single die or by adding several CPUs to the same node. Programmers can then exploit the CPU for more parallelism as well. Unfortunately, CPUs with high core count are

prohibitively expensive to acquire and operate; the price for the Xeon Platinum 8176 processor with 28 cores started at $9284.99 and an average power consumption of 165 W on intel.com on 2018/12/29. [2]

To increase energy savings, leaner CPU architectures are also considered as a replacement of the (bloated) x86 architecture. IBM's PowerPC architecture, for instance, has been used for years in the BlueGene supercomputer-series. Other architectures, such as ARM, are investigated as well. The success of an architecture in a large market, like the phone market in the case of ARM, ensures two important factors: First, existing demand for a processor and competition among manufacturers reduces the price itself. Second, to survive fierce competition, constant funding must be made available for further research and development of the architecture. These architectures offer hence a better value for money than expensive custom architectures that require funding for research and development to be provided by the HPC community itself.

To investigate the usability of devices coming from the embedded systems world, the Mont Blanc project has been started and investigated several production-level applications on a prototype system running on ARM Cortex-A15 CPUs and ARM Mali-T604 GPUs [Raj+16].

In this context, it is interesting to know that in order to improve energy-efficiency, ARM introduced heterogeneity at the core level through its big.LITTLE concept, i.e., by implementing cores with different computational power on the same die. The more powerful a compute core, the more powerhungry it normally is, meaning that for simple tasks energy can now be conserved by using only the weak (and energy-efficient) cores.

### GPU

GPUs were originally designed to accelerate 3D graphics in computer games and have seen continuous development, driven by the competition in the gaming market. They have been an integral part of home computers for a long time and are available at a reasonable price. For these reasons, the HPC community started to exploit GPUs more than a decade ago.

In response, manufacturers have begun to develop GPGPUs (general purpose GPUs) that target computation-heavy applications but vendors also include features to

[2] Seen on https://www.intel.com/content/www/us/en/products/processors/xeon/scalable/platinum-processors/platinum-8176.html

speed up several common types of computations. For instance, with the advent of deep-learning/AI, GPUs are specifically designed to support important operations at maximum performance, e.g., $\frac{1}{4}$ precision operations are marketed by Nvidia for its Tesla V100 GPGPU with a maximum performance of $125\,\mathrm{Tflop/s}$, single-precision at $15.7\,\mathrm{Tflop/s}$ and double-precision at $7.8\,\mathrm{Tflop/s}$. [3]

Programmers and algorithm designers hence have to carefully evaluate if mixed-precision computations can be used in their algorithm. As a result, the performance could be improved easily by, e.g., a factor of 2:1, by using single instead of double precision. This factor depends on the GPU generation, as even Nvidia's own GPUs delivered unstable performance across several generations, with Tesla's 8:1 performance ratio being especially noteworthy [Don+17, p. 57].

### Many-core

Today, only few systems (when compared to GPUs) rely already on MICs, but its advantages, such as high parallelism, applicability of standard programming techniques and (compared to CPUs/GPUs) low energy consumption, make this technology a good candidate to achieve sustained exascale performance.

After several revisions, including bootable Xeon Phi CPUs that removed the PCIe bottleneck, Intel decided to discontinue its *Xeon Phi* accelerator series. For this (and sometimes political reasons), leading systems, such as the chinese Tianhe-2 supercomputer, have already started to replace the Xeon Phi's with next-generation MICs from other manufacturers or GPUs. The November 2018 edition of the Top500 lists only one single system (called Trinity) in the Top10 that still uses Xeon Phi Co-Processors. [4] In the case of Tianhe-2, replacing the Xeon Phis from 2013 with the 128-core Matrix-2000 MIC increased its benchmarked performance from $33.9\,\mathrm{Pflop}$ to $61.4\,\mathrm{Pflop}$, an improvement of $81.12\,\%$; at the same time, its energy consumption increased comparatively little by only $4\,\%$. [5]

Another MIC nowadays used is the PEZY-SC2 [TOR+17] (PEZY stands for *Petascale Exascale Zetascale Yotascale*). It contains $2048$ cores, each running at $1\,\mathrm{GHz}$, and delivers a total of $4\,\mathrm{Tflop/s}$ (Rpeak) for double precision computations. $6$ MIPS64 cores manage the compute cores and make an external CPU unnecessary. With a consumption of only $180\,\mathrm{W}$, the PEZY-SC2 plays an important role regarding power efficiency. The 1st ranked machine in the November 2018 issue of the Green500

[3] From https://www.Nvidia.com/en-us/data-center/tesla-v100/
[4] See https://www.top500.org/lists/2018/11/
[5] See https://www.top500.org/lists/2018/06/

list was noted to achieve $17.604\,\mathrm{Gflop/W}$ with a PEZY-SC2, whereas the 2nd place achieved $15.113\,\mathrm{Gflop/W}$ with an Nvidia GPU.

Further development of the PEZY MIC is underway and the PEZY-SC3 was announced to further increase the core-count with $4096$ to $8192$ cores [TOR+17].

Other MIC products include the Kalray MPPA (massively parallel processor array). Its third generation (called Coolidge) was announced by the manufacturer in 2017 to consist of $80$ or $160$ compute cores that run at $1.2\,\mathrm{GHz}$. Additionally, the same amount of co-processors is used to boost performance of deep learning or computer vision. For these applications, the Coolidge card can perform at up to $5\,\mathrm{Tflop/s}$ at less than $20\,\mathrm{W}$, i.e., with a computational performance of $250\,\mathrm{Gflop/W}$ [Kal17].

In the GPU market, it is accepted that Nvidia dominates AMD, but it is too early to say which MIC will become the most popular one. Further development with especially rapid innovation cycles can hence be expected.

#### FPGA

Field-programmable gate-arrays (FPGA) are integrated circuits that are first manufactured and can then be programmed by a customer to match their specific needs. Implementing circuits is generally done through hardware description languages (HDL) and require expert knowledge. To improve performance through optimized placement and routing on the die, compilation times can frequently take more than $4$ to $10$ hours. This is very different to e.g., CPUs that can more easily be programmed through C or C++.

FPGAs are therefore also predestined to implement rather simple operations that have to be executed quickly and over and over on large amounts of data. With the function being implemented *in the circuit*, there is no overhead of translating *instructions*. This is fundamentally different from CPUs, GPUs and MICs which are all programmed by software and hence suffer this overhead as they require translation from instructions.

FPGAs can also be attached to virtually any input / output sources, allowing them to achieve extremely high bandwidth and extremely low latency as they are not forced to communicate via PCIe or NVlink. This is why they're used in many data intensive projects, such as signal analysis in astronomy. Astron, the Netherlands Institute for Radio Astronomy, developed for this purpose UniBoard I and more recently UniBoard II, a board with several FPGAs assembled together [Rad]. Astron

claims on its website that UniBoard II can process a total data rate of $5\,\mathrm{TB/s}$ which they compare to the Amsterdam Internet Exchange points total $2\,\mathrm{TB/s}$, i.e., the four FPGAs could process more than twice of the entire exchange points current data throughput.

Even though FPGAs are known for high performance, GPUs can sometimes outperform them, most notably on floating-point operations where GPUs excel.

An FPGAs main advantage, however, is that it can require less power than a GPU (even on floating-point operations) since the GPU depends on a host for communication, which needs to be accounted for as well.

More efforts to establish FPGAs in HPC include for instance Intel's recently announced Xeon-FPGA Hybrid Chip [Huf18] and ongoing endeavours to make FPGAs high-level programmable via OpenCL and C/C++ [Cza+12].

It is clear that FPGAs are great to provide the throughput that some applications require, but whether they will be largely used is unclear since programmers have just now become used to programming GPUs and would need to be largely retrained on an even more complicated matter. Furthermore, whether FPGAs can lead to a change of node layouts (currently CPU + accelerator) remains unclear, but efforts are certainly made in this direction, e.g., by connecting FPGAs directly to routers.

### 2.2.3 Architectures: Communication

The network assures communication between all parts of the machine and hence plays a critical role in each machine. To support the future massive parallelism, new technologies had to be developed to, on the one hand side, prepare the network stack for this task and on the other hand to assure the connectivity of thousands of cores on a chip, with very little space available, in a network-on-chip.

The following sections give a brief overview over the network and network-on-chip technologies and used topologies.

#### Topology

Desireable properties, such as fast routing, have leveraged regular topologies. In some cases (e.g., the 5D-torus of the BlueGene/Q), topologies can isolate an allocated node-partition used by a single application, i.e., they evade cross-traffic from nodes allocated to other jobs. Alas, scheduling jobs such that properties of

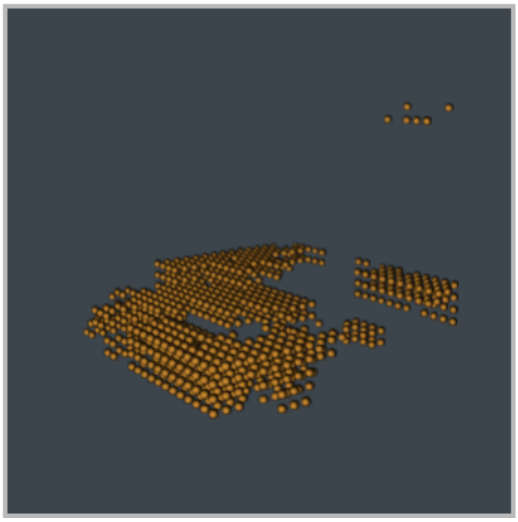

**Figure 2.7:** Visualization of nodes allocated to a job running on BlueWaters 3D-torus topology. Obtained through private communication from Greg Bauer.

the underlying topology are retained while maximizing machine utilization is very hard and machine-fragmentation is well-known to occur often, leading to poor resource utilization.

Figure 2.7 shows irregularly allocated nodes for a job, running on the BlueWaters system that features a 3D-torus. In such a case, mapping the generally regular (MPI) virtual topology to the physical machine results in a mapping that does no longer respect the regularity, thus decreasing network performance.

Future machines can choose to either use more and more powerful nodes ("fat nodes") with very high core-count and several accelerators or to increase the number of nodes. In this latter case, an exascale network must have several important properties, such as high-bandwidth, scalability, low latency and resilience to link failures. Furthermore, network size should not be determined by the topology (as for example for regular topologies) but through factors such as the available (power) budget or datacenter space [Koi+13, Section 1]. Networks account for up to $33\,\%$ of the total machine cost and up to $50\,\%$ of the total system power consumption [BH14, Section 1], making optimizations here also financially worthwhile.

New topologies have hence been proposed, such as the DragonFly [Kim+08], Slim-Fly [BH14] or SmallWorld [Koi+12] topologies.

These topologies use high-radix routers to increase connectivity and reduce the diameter (i.e., the length of the longest path is minimized). Dragonfly topologies can scale up to $256\,000$ nodes when $64$-radix routers are used, all while keeping the diameter of the network at $3$. With the same diameter, Slimflies can scale up to millions of nodes; for a diameter of $2$, it still supports more than $100\,000$ nodes.

Improving the interconnect is important to reduce for example the latency of remote memory accesses. Nevertheless, the main increase in parallelism still takes place intra-node, with thousands of cores on a single chip. An optimization goal for the best connection of these cores in a Network on Chip (NoC) is a low diameter and high-radix. Due to the die specificities (e.g., limited space) and required high-energy efficiency, we see the same topologies (dragonfly, slimfly, ...) being proposed as an improvement (albeit generally with small modifications to account for die specificities) for intra-node communication as for intra-node networks [Bes+18].

### 2.2.4 Programming Paradigms

The previous discussion has shown that exascale machines have added complexity on all levels: A magnitude more nodes require more inter-node communication, irregular topologies make it more difficult to map virtual topologies and massive on-chip parallelism on very different accellerators (FPGAs, GPUs and MICs) and chip-level heterogenity (big.LITTLE) require applications to be programmed with highly diverse devices in mind. Unfortunately, this extreme level of parallelism and performance requirements force programmers to take over even more responsabilities, such as load-balancing, fault-tolerance, data transfer and core-specificities (e.g., faster execution on a GPU than on a CPU for a specific kernel).

At the same time, application developers are required to optimize their code towards more energy efficiency as the consumption of power will be limited in exascale machines. Simple throughput, measured in $\mathrm{Gflop/s}$ (i.e., operations per time unit), will be no longer the optimization goal; instead, it is expected that the focus will shift towards $\mathrm{Gflop/W}$ (i.e., operations per energy unit) [Don16, Section 1.7].

Programmers are responsible to ensure proper functioning and high performance of their application even on the most complex machines. Consequently, higher amounts of testing (and, naturally, debugging) efforts will incur significantly higher costs by blocking human and machine resources [Don+11, Section 4.2.5.1]. A direct consequence of this is less time for actual science and development of the application. The time spent testing and developing is exacerbated by the current approach of using MPI plus one *or more* of CUDA, OpenMP, OpenCL, etc.. Having to implement kernels possibly several times bloats code, makes it less maintainable and may duplicate code. The MPI+X approach also requires programmers to implement the aforementioned responsabilities such as load-balancing themselves. At this point, a paradigm change towards runtime systems seems inevitable.

Runtime systems such as StarPU [Aug+10], Charm++ [KK93] or PaRSEC [Bos+13] aim to reduce this development overhead by automating important aspects of

application performance, such as scheduling, job placements (with data locality) and load-balancing. By employing a runtime, programmers trade in their fine-grained control over all important aspects of the execution for higher portability and code maintainability/cleanliness. Runtimes force developers to write their code in a task-based manner, i.e., functions are now essentially tasks that are not called directly but submitted to the runtime for execution. Once a task has been submitted, the runtime decides the appropriate device for execution (GPU, FPGA, ..., given an implementation on that device exists) and automatically takes care of scheduling, data transfers and load-balancing. Unfortunately, most existing implementations require a complete unrolling of the task graph to schedule tasks at runtime, making this approach at the moment inadequate for extreme core numbers [Don+17]. The choice of the right runtime is difficult as they are all still under heavy development and subject to change. Due to a lack of standardization, adopting a runtime system today may entail significant refactoring in the future if a system is abandoned or changes its API significantly.

Finally, runtimes can improve the overall exploitation of resources considerably, but it is not their responsibility to find the best values for all performance relevant parameters of the computation kernels on each resource type (CPU, GPU, ...), such as compiler options or algorithm selection. Unfortunately, the search space grows exponentially with the number of parameters and allowed values, making it prohibitively expensive to search exhaustively. Thankfully, tedious manual testing can be avoided through interpretation as a search problem, known as *autotuning*, and application of appropriate algorithms [Ans+14; WPD01].

## 2.3 Conclusion

The race for exascale requires disruptive technology changes in almost all areas, from hardware to software. With new technologies replaceing the old software-hardware-stack, the entire eco-system is completely revolutionized. This causes even formerly barely important optimization goals, such as energy consumption, to become the main optimization goal.

Limited experience with this all-new and highly intricate technology-stack complicates performance analysis and energy prediction. Allocating expensive resources for testing purposes needs to be reduced to a minimum, and a possible approach to achieve this is to use faithful models and then evaluate these through a simulated application run.

SimGrid (see Chapter 4) is a simulator with particularly faithful and validated models, particularly for network communication. We consequently extended this simulator with an energy model and will show that even adaptative applications can be faithfully simulated.

# 3 Related Work

The following overview over existing work related to system simulation is paraphrased from our work [Hei+17b].

## 3.1 Simulation Based Performance and Energy Prediction

The energy consumption of data centers is constantly growing, with U.S. data centers expected to consume about $73 \times 10^9$ kW h in 2020 [Sho+17, Section 1]. Unsurprisingly, optimizing energy efficiency has become a major undertaking for data centers operators [Sho+17; Wil18], but also application developers are more and more required to consider energy aspects.

An often published approach to assess core application characteristics such as the (network) performance on a specific machine is through simulation. In a cloud context, several simulators were already presented that come with models for energy consumption [OPF10; Tig+12]. These energy models are at the basis of energy-related studies, such as cloud management strategies with the help of dynamic frequency scaling models [Gué+13] implemented in CloudSim [Cal+11]. GreenCloud [KBK12], an extension of the (now unsupported) NS2 simulator, can be used to evaluate energy-aware networking approaches designed for cloud infrastructures.

DCSim [Tig+12] is a tool particularly suited for studies of management strategies of dynamic, virtualized resources and supports per-host power predictions through its energy models.

Unfortunately, the faithfulness of the communication models implemented in some of these simulation frameworks is undermined by grave errors that are particularly critical in an HPC centric context [Vel+13a], while other frameworks seem to be more realistic but have not been empirically validated. The most realistic results are arguably obtained by packet- and cycle-level models, however, only in the case that they are correctly instantiated and used [Now+15], which is particularly difficult in an HPC context. Furthermore, they are hardly usable in HPC contexts, even when

correctly used, as their low-level models cause the performance to deteriorate when used at scale.

We saw in the previous chapter that applications written with high-performance computing in mind usually rely on MPI. This large class of applications makes the performance prediction of MPI applications on complex platforms particularly interesting. This is reflected by the large number of simulators, among others Dimemas [Bad+03] (developed by the Barcelona Supercomputing Center), BigSim [ZKK04] (University of Illinois at Urbana-Champaign), LogGOPSim [HSL10] (ETH Zürich), SST [Jan+10] (Sandia National Laboratory) and the xSim [Eng14] project (Oak Ridge National Laboratory). Recently proposed simulators include CODES [Mub+17] (Lawrence Livermore National Laboratory) and HAEC-SIM [Bie+15] (TU Dresden and University of Basel).

Studies often focus on how the performance of a particular MPI applications evolves when scaled strongly or weakly, but the impact of other parameters (e.g., network topology, link bandwidth and congestion) is also of interest.

Surprisingly, models capable of predicting multi-core architectures are only employed by a few simulators, such as Dimemas [Bad+03], which can discriminate local (using shared memory) and remote (using the network) communications but does not account for performance degradation incurred by processes contending on cache or memory. To predict intricate applications at scale, Dimemas can be combined with the PMAC framework [Sna+02] and its elaborate cache hierarchy model. Alas, only application traces can be replayed by these two tools, which is often too limiting when dynamic applications (i.e., applications that adapt to the underlying platform) or different scales need to be simulated.

Models for energy prediction (or other facilities that can be used to conduct energy-related studies) are, to the best of our knowledge, not implemented in any of these tools with the notable exception of HAEC-SIM.

Alas, the models were tailored towards their quite specific use case and HAEC-SIM only supports a small subset of the MPI API. The NAS-LU benchmark was used for validation at small scale with only 32 processes. HAEC-SIM obtains rather promising and faithful prediction trends but power estimation errors (when compared to reality) not uncommonly range from 20 % to 30 %.

## 3.2 Conclusion

We already discussed in Chapter 2 that systems have become extremely complex and are still expected to continue to do so. Predicting even a simple application's performance manually has become almost impossible today. HPC simulators are a promising approach for this and other reasons, such as debugging an application when the sheer complexity of a platform triggers transient bugs that only appear under specific conditions. On real machines, recreating this bug can take tremendous amounts of time as for example the reserved nodes differ from execution to execution. A deterministic simulator can help significantly to first identify the problem and then, once a patch has been written, even verify that the problem is now solved.

In the future, the main focus will be on energy and therefore adding the capability to simulators to predict the energy consumption of HPC applications is mandatory, as this will open further fields of study such as improving energy-efficiency.

# 4 The SimGrid Project

## 4.1 Overview of the SimGrid Project

### 4.1.1 History and Impact

Development of SimGrid was started in the early 2000's by Henri Casanova in order to study scheduling heuristics developed for a grid computing context. Arnaud Legrand (CNRS, Université Grenoble-Alpes), Martin Quinson (École Normale Superieure, Rennes) and Frédéric Suter (CNRS, CC-IN2P3, Lyon) joined the project subsequently and act today as project leaders. Development is constantly backed by several PhD students and postdocs, tenured researchers and software engineers. Over the years, around 80 people from various fields and institutions have contributed to SimGrid's code base.

Researchers not directly part of the SimGrid-project principally make contributions because they need a bug fixed or a feature added that allows them to investigate a specific aspect of their domain, mostly in HPC, P2P and cloud computing. Many interesting aspects of these domains can be investigated through simulation and it is especially readily used to investigate questions that would require access to a large experimental setup (e.g., HPC machine, several clouds, P2P network) that is often not available to researchers. Some examples of questions that have been investigated with SimGrid are algorithmic performance (including performance of communication algorithms [Deg+17, p. 2394]) but also networks (routing, performance bottlenecks, impact of latency or bandwidth, link failure, ...) [Yas+19] and impact of storage I/O [Leb+15].

SimGrid is a long-term project and not a one-shot prototype. Its measurable impact since the early 2000's is visualized in Figure 4.1. For each year, this figure illustrates the number of publications the SimGrid project is aware of that only cite (but don't contribute to) SimGrid ($> 500$), use SimGrid as a research tool in their own scientific investigation ($\approx 320$) or detail specific and scientifically relevant new developments ($\approx 60$). A comprehensive list of the publications on the SimGrid development and some of its components is available online [Tea].

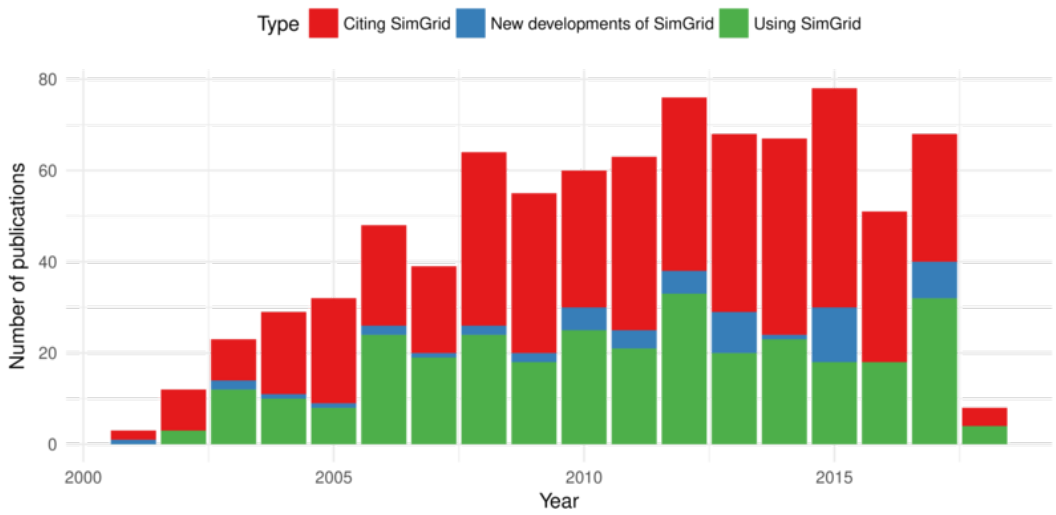


**Figure 4.1:** Impact of the SimGrid project per year, measured by the amount of citations (red), papers detailing new components/modules of the SimGrid project (blue) and scientific work that uses SimGrid as a tool for research (green) [Tea].

### 4.1.2 Software Architecture

SimGrid was founded as a research project itself, written in ANSI-C (to optimize performance) and with an ad-hoc architecture. Continuous development and added functionality have made changes necessary to keep the code maintainable. In 2015 , and therefore around the same time this dissertation project was started, it was finally decided to rewrite SimGrid with C++ so that entry barriers for users and new developers could be lowered and code maintenance be reduced and overall made less tedious. On this occasion, a new architecture (illustrated in Figure 4.3) was also conceived and replaces step-by-step the old architecture [Cas+14] (see Figure 4.2). This major refactoring of SimGrid's internals made this dissertation project more difficult, as the main components that were required for this work were subject to change (and hence not yet stable) or had to be rewritten completely by myself. SimGrid's refactoring is still ongoing and pursued in an iterative approach, i.e., once implemented and tested, changes are merged and published with the next, quarterly SimGrid 3 release. Once the refactoring is complete, SimGrid 4 will be released.

In the following, a brief description of the main components of this architecture is given. A more detailed presentation of the general functioning of some of these components will be given in Section 4.2 and Section 4.3.

Today, self-implemented user applications only have dependencies on the publicly available S4U ("SimGrid for you") API. Previously, user applications using SimDag accessed the simulation kernel directly whereas applications using MSG ("meta-SimGrid ", a legacy interface to describe simple distributed algorithms) and SMPI

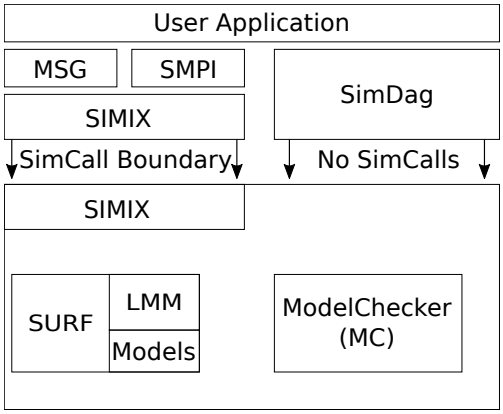


**Figure 4.2:** Overview over legacy SimGrid components.

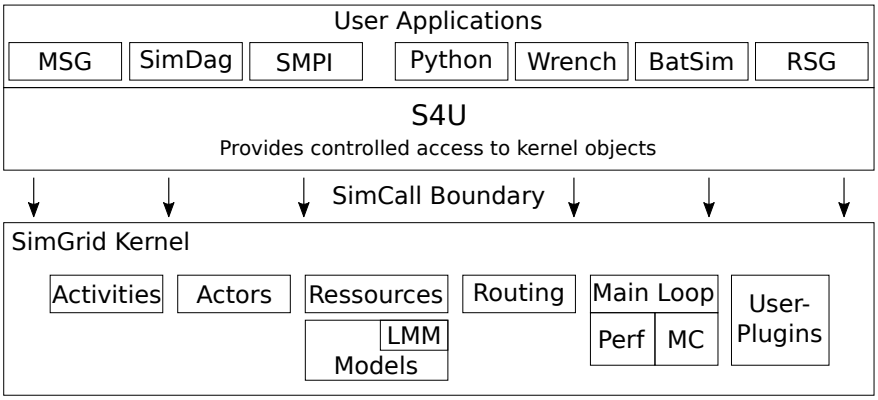


**Figure 4.3:** Highlevel overview over SimGrid 4. Only the new S4U layer can communicate with the kernel.

(the SimGrid interface that facilitates simulation of MPI applications) relied on SIMIX. SIMIX corresponds to an OS kernel as it manages resources (hosts, actors, ...) and contains public (accessible by user code) and private functions. To prevent user applications from modifying simulator internal variables directly, interacting with the kernel is enforced via simulation calls (called *simcalls* for short) which are issued by the public part of SIMIX towards the private part [Cas+14]. Decoupling user applications from the simulation kernel will be important in the future, for instance to distribute the simulation on several machines in which case several processes won't be able to directly access the memory.

The SURF layer corresponds in a wider sense to a virtual machine. It is most importantly responsible for the virtual time management. Furthermore, it is responsible for resource creation (instantiation of models, see Section 4.2 for a short introduction) and allocation (e.g., compute power when several processes contend for the CPU). Resource allocation is currently implemented with a linear max min model (LMM), which is solved internally.

SimGrid 4 contains a largely modularized simulation kernel. Large parts of the former SURF layer, such as routing or actors, were moved into their own modules which can now easily be exposed for extension, whereas they were hidden within the SURF and SIMIX layers in legacy SimGrid. In addition to these extension points, signals have been implemented in most of the kernel modules. Signals allow users to write plugins that implement listeners to specific events (e.g., new actor, start of the simulation, ...). The usercode can therefore implement user-dependent behavior more easily as it is notified of previously "internal" events. As a consequence, user contributions can be distributed more easily, as most of the code will no longer require to make changes to SimGrid's source files.

### 4.1.3 Software Engineering

The goal for each simulator is to compute high-quality results. Unfortunately, in some "corner case" scenarios even small, inconspicuous changes to the simulator can affect results unintentionally, yet significantly. Hence, it is important to automatically test and detect these errors in order to maintain a good level of software quality. This becomes even more important with the possibility of portability-based errors, for instance between the platforms that SimGrid supports (currently Linux, BSD, Windows, MacOS) and non-obvious connections between the different software layers or APIs.

To cover a wide range of possible errors, a standard SimGrid build (with SMPI enabled) currently uses 665 functional tests (mostly execution of examples or tailored tests) that are checked after each push to the main `git` repository. Each test is only marked as failing when the result does not equal the expected output. To test correct return values and functioning of specific functions, SimGrid's developers currently add unit tests that will provide more thorough and isolated testing of internal mechanisms. In addition to these homemade tests, SimGrid launches an extensive test suite at certain hours (currently every night). This suite included at the time of writing 49 "Proxy-Apps" with code sizes ranging from 255 lines of code (`zlatest`) to 109 477 (`CLAMR`). Furthermore, real projects such as `BigDFT` (a quantum chemistry code) and the runtime StarPU are also regularly tested.

To manage these tests and their execution on the different platforms automatically, SimGrid employs several Continuous Integration projects such as Jenkins[1], AppVeyor[2] and Travis[3]. Figure 4.4 gives an overview of all currently tested platforms and their configuration.

[1] See `https://www.jenkins.io`
[2] See `https://www.appveyor.com`
[3] See `https://travis-ci.org`

| Name of the Builder | OS | Compiler | Boost | Java | Cmake | NS3 | Python |
|---|---|---|---|---|---|---|---|
| **simgrid-centos7-x64** | CentOS Linux release 7.6.1810 (Core) 64 bits | GNU 7.3.1 | 1.67.0 | 1.8.0.191 | 3.13.1 | | ✔ |
| **simgrid-debian-stable** | Debian 9.7 (stretch) 64 bits | GNU 6.3.0 | 1.62.0 | 1.8.0.181 | 3.7.2 | ✔ | |
| **simgrid-debian8-64-dynamic-analysis** | Debian testing (buster) 64 bits | GNU 8.2.0 | 1.67.0 | 11.0.2 | 3.13.2 | ✔ | ✔ |
| **simgrid-fedora-rawhide-64** | Fedora release 30 (Rawhide) 64 bits | GNU 9.0.1 | 1.69.0 | 1.8.0.192 | 3.13.4 | | ✔ |
| **simgrid-fedora26** | Fedora release 29 (Twenty Nine) 64 bits | Intel 19.0.0.20180804 | 1.66.0 | 1.8.0.191 | 3.12.1 | | ✔ |
| **simgrid-freebsd-64** | FreeBSD 12.0-RELEASE | Clang 9.0.0 | 1.69.0 | 1.8.0.192 | 3.13.3 | | ✔ |
| **simgrid-manjaro** | ManjaroLinux 18.0.2 (Illyria) 64 bits | Clang 7.0.1 | 1.69.0 | 11.0.1 | 3.13.3 | | ✔ |
| **simgrid-nixos** | NixOS 19.03pre168320.2d6f84c1090 64 bits | GNU 7.4.0 | 1.67.0 | 1.8.0.202 | 3.12.1 | | |
| **simgrid-opensuse** | openSUSE Tumbleweed 20190205 64 bits | GNU 8.2.1 | 1.67.0 | 11.0.2 | 3.13.2 | | |
| **simgrid-osx-highsierra** | Mac OS X 10.13.6 64 bits | AppleClang 10.0.0.10001145 | 1.68.0 | 11.0.2 | 3.13.4 | | ✔ |
| **simgrid-ubuntu-bionic-64** | Ubuntu 18.04 (bionic) 64 bits | GNU 7.3.0 | 1.65.1 | 10.0.2 | 3.10.2 | ✔ | |
| **simgrid-ubuntu-xenial-32** | Ubuntu 16.04 (xenial) 32 bits | GNU 5.4.0 | 1.67.0 | 1.8.0.191 | 3.5.1 | ✔ | |
| **simgrid-win10** | Windows 10 v17763 - WSL Ubuntu 18.10 64 bits | GNU 8.2.0 | 1.67.0 | 11.0.1 | 3.12.1 | | ✔ |
| **travis-linux** | Ubuntu 16.04 (Xenial) 64 bits | GNU 5.4.0 | 1.58.0 | 11.0.1 | 3.12.4 | | |
| **travis-mac** | Mac OSX High Sierra (kernel: 17.4.0) | AppleClang 9.1.0.9020039 | 1.67.0 | 10.0.1 | 3.11.4 | | |
| **appveyor** | Windows Server 2012 - VS2015 + mingw64 5.3.0 | GNU 7.2.0 | 1.60.0 | 1.8.0.162 | 3.12.2 | | ✔ |

**Figure 4.4:** Overview of used platforms and essential configurations for building and testing SimGrid as used on 2019-02-08 [Tea19].

It should be noted that testing more exotic and rarely used platforms such as FreeBSD has proven to be useful, as development is typically done on machines with a common configuration, such as Debian machines with GCC. Tests tend to break more often on these platforms, revealing more profound issues that may also impact common platforms but in more subtle ways.

The remainder of this chapter is structured as follows: Section 4.2 gives a general introduction to SimGrid and explains how platforms and applications can be modeled. Section 4.3 explains SMPI, which was used and extended for this thesis. Simulation support for accelerators with runtimes is briefly discussed in Section 4.4 and finally, non-scientific contributions I made to the development (often necessary due to the C++-rewrite) of SimGrid are presented in Section 4.5.

## 4.2 A General Introduction to SimGrid

### 4.2.1 Modeling Virtual Resources

To accurately consider different aspects of a platform, SimGrid relies on models of the most important parts, such as the network and hosts (CPUs). For almost every part (model, routing), several exchangeable implementations exist that normally differ in their outcome and computational complexity. This allows users to use the implementation that is suited the most to their workload.

The following sections give a brief overview over the most important models of nodes and network. To illustrate concrete modeling of a platform, an example platform file is discussed.

### Computation Unit modeling

Several computation units (CPU, GPU, MIC, FPGA) were presented in Section 2.1.2 and Section 2.2.2, with a specific focus on their individual advantages or disadvantages. Their performance differs largely and is dependent on the application's cache-usage, computation type (e.g., double, single or half-precision floating points) and supported degree of parallelism.

In SimGrid, computation units are not explicitly declared as GPUs or CPUs. Instead, abstract modeling with very few parameters is used to represent their computational capacities. The user can declare the number of cores that are available and hence how many computations can be executed in parallel *at full speed*. If more computations are executed than cores are present in the machine, the total computational power is distributed fairly over all computations. This means that scheduling among processes is ignored and therefore *all computations make progress at any time*. Their computational capacity (frequency) is declared in $\mathrm{flop/s}$ instead of $\mathrm{MHz}$. This allows users to differentiate between two CPU models that ship with the same clock frequency but have possibly very different performances due to different internal features.

SimGrid's models also provide a notion of *performance-state*. This means that several frequencies can be declared per host, but only one can be active at a time (i.e., core-dependent frequencies are not directly supported). This provides several advantages: First, a computation unit (CPU) that supports frequency scaling can be easily modeled. Secondly, this provides ways and means to model non-trivial computational effects as well. Heavy I/O can for example slow-down computations and could be modeled by declaring a specific frequency that represents this slowdown. The user can subsequently manually enter a modeled state through an API call to S4U so that SimGrid can take the state's specific performance into account.

SimGrid's simple models do not consider important features such as TurboMode (for Intel-CPUs), memory or cache latency. These features are too device-specific and due to their extreme complexity very difficult to implement correctly. The downside of this missing support is that application performance is sometimes not correctly estimated, most notably when multi-core CPUs are used. To help ameliorate this situation, I contributed a first step towards a better solution that is presented in Chapter 6.

On the other hand, the advantage of this model for multi-core computational units is first of all its low computational cost. For long computations, there is no need to

account for all the details of the rapid frequency changes incurred by TurboMode. Only the average time spent in each state is important. Additionally, the user can remodel the platform easily and hence adapt it to new devices without changing the implemented model itself.

### Network modeling

To accurately predict a network's performance, its topology and the impact of contention must be taken into account. SimGrid treats each communication as a single *network flow,* therefore avoiding a split of the communication into many individual packets, which is computationally overly costly and prohibits large-scale simulations. With this approach, only one flow needs to be managed instead of possibly thousands or millions of individuel packets. Choosing a flow-based representation hence comes with the additional benefit that the SimGrid-internal overhead for managing (storing) the communication is *independent* of the message size, a feature crucial to simulations sending large messages.

Under the assumption of steady-state, SimGrid applies a bandwidth-sharing algorithm to compute incurred contention. This algorithm considers non-trivial phenomena [Vel+13b] such as network heterogeneity, Round-Trip-Time-unfairness of TCP [Mar+07] or reverse-traffic interference [Heu+11]. SimGrid recomputes the bandwidth-share for each flow every time a communication finishes (and hence releases resources) or begins (and therefore consumes resources). Transient phenomena of the network protocol, such as the time it takes to converge to a stable bandwidth, are currently not considered by the model, see Figure 4.5. SimGrid's predictions are thus expected to be rather optimistic when compared to reality.

### Platform Description

To instantiate the models of computation units and network, the simulator requires the user to supply a high-level description of the target platform, written in a human-readable format, which is then used to create simulator-internal datastructures. This description includes information structured through so-called *zones*. Zones can represent for instance different rooms of several or a single data center, different racks of a cluster or just groups of nodes connected to different routers. Each zone contains either more zones or a description of the hardware setup it contains, i.e., the network (links including their bandwidth and latency, routers, routing algorithms or tables) and the nodes (especially number of cores, available frequencies). The

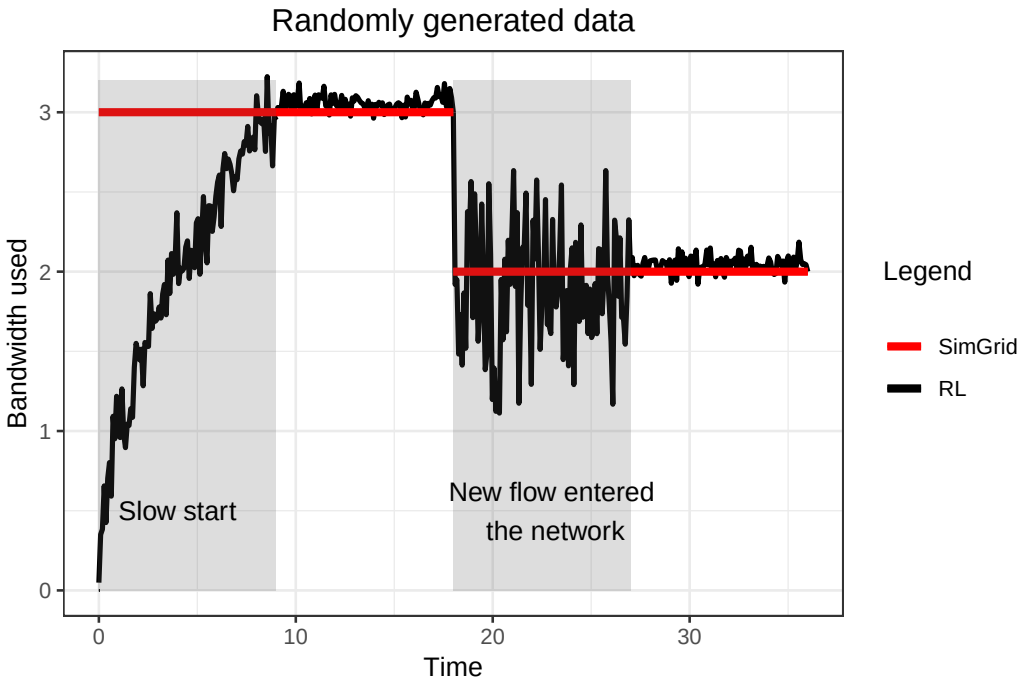


**Figure 4.5:** SimGrid's models simplify real-life network usage by ignoring the bandwidth adjustment phase in the beginning and when a new message enters the network, it reduces the bandwidth immediately, without going through another adjustment phase.

physical location of a node in a rack or the length of a network link can influence energy consumption and performance, respectively, but is currently ignored.

The standard format for supplying the platform description is XML, even though SimGrid also supports a programmatic (C/C++) approach. An overly simple platform is shown in Figure 4.6. The platform only contains one zone with two nodes that can compute $1\,\mathrm{Gflop/s}$. The nodes (identified by the "host" tag) are connected through a single link with bandwidth $1\,\mathrm{GB/s}$ and a latency of $25\,\mathrm{ms}$. Note that the link itself is not explicitly connected to any of these hosts. The link is just declared to be part of the route from host S1 to host C1 (and vice versa). In fact, links in SimGrid are not modeled as "cables" and have no designated endpoints. This counter-intuitive modelisation allows users to use links to model specific features of the platform more easily. For instance, by declaring that the same link connects hosts A and B as well as hosts C and D, communication between A and B is impacted by communication between C and D (and vice versa) because both communication flows now have to share the bandwidth of this link.

```
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid/simgrid.dtd">
<platform version="4.1">
  <zone  id="AS0"  routing="Full">
    <!---   S1 <-> link 1 <-> C1  -->
    <!---        (1GBps, 25ms)     -->

    <host id="S1" speed="1Gf"/>
    <host id="C1" speed="1Gf"/>
    <link id="1" bandwidth="1GBps" latency="25ms"/>
    <route src="S1" dst="C1">
      <link_ctn id="1"/>
    </route>
  </zone>
</platform>

```

**Figure 4.6:** A simple platform, taken from SimGrid's collection of examples, called `onelink.xml`. Two hosts with 1 Gflop/s and one link with 25 ms latency and 1 GB/s are declared. This link is furthermore used to connect both hosts.

### 4.2.2 Modeling Applications

Once the platform has been modeled, it can be used to simulate a parallel application. With SimGrid, this can be done in two different ways: (offline) trace-based replay of application behavior and direct (online) simulation of applications.

The easiest approach is the trace-based replay. A trace file is normally obtained by tracing the targeted application with a trace-tool. Once obtained, traces can be replayed on different simulated platforms. This allows the user to study the impact of changing platform parameters such as network bandwidth, network topology or node speed on the application. Trace files are static and therefore prohibit the simulation of adaptive applications as they only contain the behavior of *one* execution. To illustrate this, consider an application that sends and receives asynchronous messages, for example, to overlap communication times with computations. This application's behavior may change on a different platform because computation and communication operations may change their order.

In order to simulate applications with adaptive behavior, a new concept is required. SimGrid allows users to model the application as a set of actors interacting with each other. Programmers fully control actors by explicitly providing each with a function (written in C or C++) that contains the actor's logic (e.g., computations, message exchanges with other actors, synchronization, ...).

Although SimGrid allows users to simulate a parallel system, it is a sequential simulator that always executes actors mutually exclusively, i.e., one at a time, *within the same process* but within *distinct contexts*. SimGrid implements several different context-strategies (listed in Table 4.1) out of which the user can choose a single

| Context name | Comment |
|---|---|
| thread | Implemented as pthreads; very slow but great for debugging |
| java | Java threads (for simulations written in Java) |
| ucontext | Fast System V contexts, non-portable (only Linux/BSD). |
| boost | Very fast implementation; requires additional libraries |
| raw | Implementation in Assembler, no system calls, x86/amd64 only |

**Table 4.1:** Exhaustive list of all currently by SimGrid supported contexts, i.e., mechanisms to virtualize user code

one for the entire simulation. Their main differences are performance and memory consumption but also debuggability.

SimGrid's sequential execution raises the question of how it manages the execution of all actors. At first, every actor is ready at time 0. SimGrid picks an actor deterministically and transfers control to the actor; it starts execution. Once the actor enters a command that requires the state of the platform to be modified (e.g., computations, communication, synchronization) it yields and SimGrid transfers control to another actor that is ready to execute. Once all actors are blocked, SimGrid uses the resource sharing models (network, computation units, etc.) to determine which actions will complete first. Time can then be advanced to the completion date of these actions. Since these actions have terminated, the corresponding actors are now ready to execute again. The above procedure is then repeated.

## 4.3 SMPI: Simulating MPI Applications

To study HPC applications through simulation, good support for MPI is crucial since this class of applications predominantly relies on MPI for communication. SimGrid's SMPI component has therefore been specifically designed to predict the performance of MPI applications. In SimGrid's current version (3.21, released on 2018-10-05), SMPI supports most of the MPI-2 standard (and a subset of the MPI-3 standard) and faithful predictions of unmodified MPI applications can be obtained [Deg+17].

We already discussed in Section 2.1.4 that MPI is mostly used for inter-socket communication whereas OpenMP is used for intra-node parallelization. Implementing OpenMP support in SMPI is much more difficult due to its use of `#pragma` statements that are parsed and substituted with parallel code by the OpenMP compiler. This means that implementing support for OpenMP requires a different approach than MPI. SimGrid therefore currently does not support the execution of OpenMP within the simulated application at all. As it does furthermore not support CUDA

| Operation Name | Implementations | Operation Name | Implementations |
|---|---|---|---|
| Allgather | 16 | Allgatherv | 7 |
| Allreduce | 15 | Alltoall | 14 |
| Alltoallv | 10 | Barrier | 3 |
| Broadcast | 17 | Gather | 2 |
| Reduce | 9 | ReduceScatter | 2 |
| Scatter | 2 | | |

**Table 4.2:** SMPI implements several algorithms for each collective operations to allow users to simulate their MPI runtime more closely.

or OpenCL, accelerators can currently not be emulated directly. An alternative approach is briefly presented in Section 4.4.

### 4.3.1 Emulation of MPI code

The discussion above has already made clear that *online* simulation of *already existing* MPI code is particularly useful. This is a form of emulation and three requirements must be given: First, the user code needs to be virtualized so that SimGrid can control the execution. Second, the time spent by the application idling, computing or communicating must be evaluated and third, since all actors share the same address space in memory and therefore also global and static variables, privatization of global variables is required to prevent incoherent memory states. The techniques to address these issues are subsequently briefly presented.

#### Virtualization of userspace code

The simulated application should not be able to tell a real MPI implementation and SMPI apart. SMPI was therefore itself built as a functional MPI implementation and manages ranks, communicators and communications and time by itself. Naturally, its implementation as a *simulator* comes with several constraints that require virtualization, i.e., replacement of certain functionalities with an SMPI-enabled equivalent. SMPI is unaware of any MPI-implementation detail and virtualization was done *on the MPI level* and not on a specific *implementation* level.

The first difficulty is to retain a valid memory state at all times. To achieve this, SMPI needs to make copies of memory in certain cases. Consider, for instance, sending and receiving a message: MPI copies the message from one process to another, possibly on another machine. This copying (via `memcpy()`) is also required in SMPI, but for a different reason, since all ranks share the same address space: If not

copied, the receiver might access the buffer after the sender already called `free()` on it. Providing the sender and receiver with their own (private) copy therefore prevents invalid memory states and possibly segfaults.

Correct estimation of runtime and a reduced execution time is a second reason for virtualization. In SMPI, the duration of computations is measured by accessing the *thread* specific clock via `CLOCK_THREAD_CPUTIME_ID`. This clock is however not advanced by functions such as `sleep()` and `usleep()`. It is therefore important to replace these functions with an implementation that injects a virtual delay into SMPI so that the time sleeping can be counted. This approach provides a second benefit: By substituting them, these functions are prevented from blocking needlessly the entire simulation process for the specified amount of time, once *for every rank that issues the function call*. SMPI's replace does not actually sleep but merely injects the time as a virtual delay back into the simulator.

SMPI provides two commands (implemented as shell-scripts) that automatically take care of the virtualization. They are named similarly to those of popular MPI implementations, namely `smpicc` and `smpirun`. The application is compiled with `smpicc` (which links automatically against `libsimgrid`, the library that contains SimGrid's fake MPI-, `sleep()` – and `usleep()` implementations, and sets a few variables after which it passes the source code to the original compiler). After the compilation, the user starts the application by running `smpirun` with several parameters such as the number of nodes, the hostfile, the platform file or tracing options. These additional options are filtered by SMPI and are not available in the user application's `main()` method, as every process is passed its own `argc` and `argv[]` parameters.

Virtualization on the MPI level has furthermore the advantage that functionality from *several* implementations can be supported exchangeably. For this, recall from the discussion in Section 2.1.4 that algorithmic implementations of MPI functions, especially of collective operations, depend on the MPI implementation. Each implementation can either offer users a range of algorithms to select from or do this automatically (at runtime), based on parameters such as message size or the geometry of a communicator. It is clear that the performance of these algorithms depends on the machine (especially the network technology / topology) and the application's communication pattern. Their performance may therefore vary greatly and must be considered for faithful simulation results. SMPI supports a large amount of collective algorithms and selector logic from several implementations, e.g., `MVAPICH` [LWP04], `OPENMPI` [Gab+04], or `MPICH` [Gro02]. Table 4.2 gives an overview of the variety of implemented algorithms, broken down by the type of the collective operation.

### Time Evaluation

Once launched, SMPI executes the binary of the simulated application and executes every instruction, just as if the binary was executed normally with MPI.

As explained above, every call to an MPI function is handled by SMPI. In its implementation of each MPI function, SMPI takes care of the time spent in MPI and the expected behavior (e.g., send / receive a message). To keep track of the time that was spent *outside* of MPI functions, SMPI relies on timers. By starting a timer before leaving an MPI call and stopping the same timer on entering the next MPI function, SMPI can benchmark the time spent computing between two consecutive calls to MPI. This time is then converted into flop, as this is SimGrid's unit of measure for computations, and consequently injected back into the simulator as a virtual delay to mark this rank as computing.

By emulating the application, SMPI cannot only take changes to the source code immediately into account but even allows application developers to simulate adaptive algorithms, i.e., algorithms that have situation dependent behavior. Naturally, this advantage comes with a downside: The overall execution time is the sum of each rank's individual execution time and therefore increases with the number of ranks.

### Privatization of global and static variables

Developers of MPI-based applications normally assume that each rank will be running within its own process. Developers hence use global and static variables without fear for race conditions since each process works on its own copy. Alas, SMPI only guarantees an individual execution context (stack, registers, code pointers) for each rank, not its own process. As a consequence, global and static variables in usercode are shared because all contexts are executed within the same process, i.e., all contexts share the same address space. Incoherent memory states are therefore possible and require particular treatment.

*Privatization* is a technique employed to avoid these issues by isolating all variables that would be shared between the threads, i.e., global and static (in the context of functions) variables. Several ways to resolve this issue exist, such as compile-time privatization or through addition of variable modifiers such as `__thread` (in C, non-standardized) / `thread_local` (standardized since C++11) directly in the source code.

This can be done automatically, for instance via LLVM's abstract syntax trees or through the tool `f2c` for FORTRAN code that names global variables in a very particular way. Since this technique can also be used to improve the performance of HPC code (it allows application developers to replace heavy UNIX processes by lightweight user threads), a few specific compilers (e.g., `mpc` [CPJ11] or IntelCC with the undocumented `-fmpc-privatize` compile-flag) support this approach. This is however not standard, which hinders the portability (between platforms and compilers) of SMPI.

But most importantly, the solution based on variable modifiers alters the application behavior. Indeed, this approach generally modifies the memory layout and the memory access pattern, as well as the potential compiler optimizations, which modifies the performance of the code. Since SimGrid regularly benchmarks the performance of the application during the simulation, heavily modifying the original compiling tool chain significantly biases the measures performed by SimGrid and affect the performance prediction in an unpredictable way.

SimGrid therefore does not rely on these compiler-based solutions. Instead, it relies on the two following different and "less-intrusive" strategies based on the system calls `mmap()` and `dlopen()`, the latter being now the default [Bed+13]. Both solutions are mutually exclusive and are always setup before any user-code is run.

► **Solution via mmap()** Immediately after startup, a backup of the application's `data` segment with all static and global variables is made and stored. This ensures that all copied variables are initialized with their default value. Each MPI rank then creates another copy in the heap through a standard `malloc()` (this copy is private to the rank). Every time the rank resumes after a context-switch, it employs `mmap()` to remap the `data` segment to its private copy. All ranks hence possess their own `data` segment and never share the state of global/static variables with other ranks.

`mmap()` only modifies the target (physical address) of a virtual address. The *virtual address* itself remains unchanged. As a consequence, each global/static variable has the same *virtual* address within all threads but the physical address is always different.

When using `mmap()`, it must be guaranteed that no values cached for one thread are used during the execution of the other. Since the virtual address remains the same, no knowledge about which thread cached the variables exists. This requires caches to be flushed and re-loaded upon each context-switch. This becomes

particularly noticeable when simulating memory-intensive applications: Frequent cache-invalidation is costly and impacts the performance negatively. To offset this, a calibration must be used. A solution will be presented in Chapter 6.

► **Solution via dlopen()** Another solution (which has recently become the default in SimGrid) is the use of `dlopen()`. This function can be used to dynamically open and close shared objects. In this approach, the fact that there is no restrictions on how often *the same* object can be loaded, is exploited. Variables are hence effectively privatized with the following approach (`#processes` is the number of processes, specified by the user):

1. At first, we generate a *new* `main()`-function that is then compiled as a new binary. The new function contains boiler-plate code that, when executed, loads the *original* binary *for each process* via `dlopen()`.

2. The user, who is unaware of Step 1, launches the simulation. Instead of executing the user-specified binary, the binary from step 1 is executed. Consequently, the `data` segment of the process running the simulator now contains static and global variables for this (very small) generated `main()`-method instead of the user-specified binary.

3. The `dlopen()` loads the binary `#processes`-times and ensures that *no address conflicts* occur between loaded instances by assigning distinct addresses. `dlopen()` normally only loads the binary once, as it is shared, but the binary can be loaded multiple times by creating a temporary copy for each rank (e.g., by suffixing the binary with the rankid). This copy is then deleted immediately after it has been loaded. All global and static variables are now effectively privatized.

4. By calling `dlsym(handle_obtained_by_dlopen, "main")` for every rank, an entry point into the original binary is obtained. This entry point allows a rank to execute the code found in the original `main()` method.

If required, libraries can be privatized as well. The main advantage of this method is that it does not require `mmap()` calls. The system can therefore differentiate instances of (now privatized) global variables through their distinct virtual addresses. This implies that after a context-switch, the cache does not require to be flushed (as for the `mmap()`-based solution). The cache can hence be fully exploited.

Unfortunately, this approach may be impractical for large-scale executions or for SimGrid-runs of applications with large binary sizes on machines with little RAM.

The reason for this is that now not only the data segment is replicated, but as described in Step 3, the code segment is replicated as well. Since the (same) binary is loaded `#processes`-times, this may become a limiting factor when studying large legacy applications at scale.

### 4.3.2 Modeling of MPI Communications

When used in applications, MPI's principal responsibility is to send messages from one rank to another. Unsurprisingly, to maximize their performance, MPI implementations and the network layer come with built-in, complex optimizations, which must be taken into consideration in order to achieve faithful performance predictions.

One such optimization is the protocol controlling the handshake mechanism. Once the sender has posted its send, the MPI implementation selects, based on the message size, whether it wishes to send the message in *eager* mode (i.e., immediately sends the message, regardless of whether the receiver is ready or not) or *rendez-vous* (i.e., the sender requests first the permission to send (request to send, RTS) from the receiver, who has to acknowledge this (clear to send, CTS)). The RTS/CTS part can make up a substantial part of the transmission time for small messages, which can in the worst case even triple. The eager mode is therefore especially useful to reduce the overhead of small messages by using more memory to buffer the message on the receiver's side until it is ready to process it. For larger messages, eager mode becomes unattractive because of the significant overhead of allocating a (large enough) temporary buffer and copying the entire buffer over once the matching receive has been posted. In this case, more specialized transmission mechanisms such as RDMA are available and result in better overall performance. The rendez-vous mode, on the other hand, requires less memory for buffers (since the receiver is ready to receive the message) but the latency increases due to the RTS/CTS part. Since rendez-vous is typically used for large messages, the added latency is of little importance and therefore often acceptable.

Many MPI implementations have options that activate asynchronous send and receive operations even for normally blocking operations. This should not be confused with eager mode: even though the function returns instantaneously, the message itself is not necessarily sent. However, applications can now overlap communication time with computation and hence make more progress. These options (send/receive modes, asynchronous sending, ...) are generally determined on the implementation level, but optimizations that exploit certain characteristics of the underlying network (e.g., OmniPath, Infiniband, ...) often influence the perfor-

mance as well. SimGrid currently ships with specific models for Ethernet [Vel+13b] and Infiniband [Vie10] networks.

To instantiate SMPI correctly, a calibration that identifies all communication modes must be executed on the real machine [Deg+17, p. 2391f.]. Figure 4.7 is a visualization of the calibration we ran on the taurus cluster in Lyon. It shows that communication performance is indeed largely impacted by the selected mode, for instance when moving from the mode in red to the mode in brown, but that each mode can be individually modeled through linear regressions. To account for these and other phenomena (e.g., protocol overhead), SMPI extends SimGrid's fluid network model (see Section 4.2.1) with a generalized LogGPS model [Deg+17, p. 2391f.] that is configured through the options shown in Figure 4.8. The correct values for these options are generally determined through the abovementioned calibration that will be briefly explained in Section 5.3. The meaning of these options is as follows: `smpi/os`, `smpi/ois` and `smpi/or` are used to control the overhead, i.e., the incompressible time taken on the sender/receiver side regardless of the transmission mode, of (i)send and receive operations of different modes. The syntax is `interval_start:startup_cost:cost_per_byte`. Here, `interval_start` is the smallest message size required to trigger this mode (but only the largest thus declared mode is activated), `startup_cost` is the constant additional cost charged *once* for every message using this mode and `cost_per_byte` determines incurred costs for every *byte* of the message, e.g., for copying the message to the network card. The `bw-factor` and `lat-factor` options affect the passage of the message through the network. They allow users to account for MPI and network overhead by reducing the maximum bandwidth that a communication can reach and increasing the latency per mode. Clearly, effective bandwidth cannot be larger than the physical bandwidth defined for a link and therefore values for `bw-factor` must be $\leq 1$ whereas the latency increases and therefore must be $\geq 1$. The option `smpi/async-small-thresh` is unfortunately misnamed and sets the upper bound up to which eager mode is used. Finally, `smpi/send-is-detached-thres` is the maximum message size for asynchronous message mode, i.e., even blocking operations will return immediately even though the message is not necessarily sent.

### 4.3.3 Scalability

Recall that aside from emulation, which was discussed above, a second way of executing an application exists: The replay of a previously obtained time indendepent (TI) trace that contains the application's captured behavior. TI traces contain all important events of the execution, in the order of their occurrence: e.g., length of computation blocks (in flop), size of messages including communication-

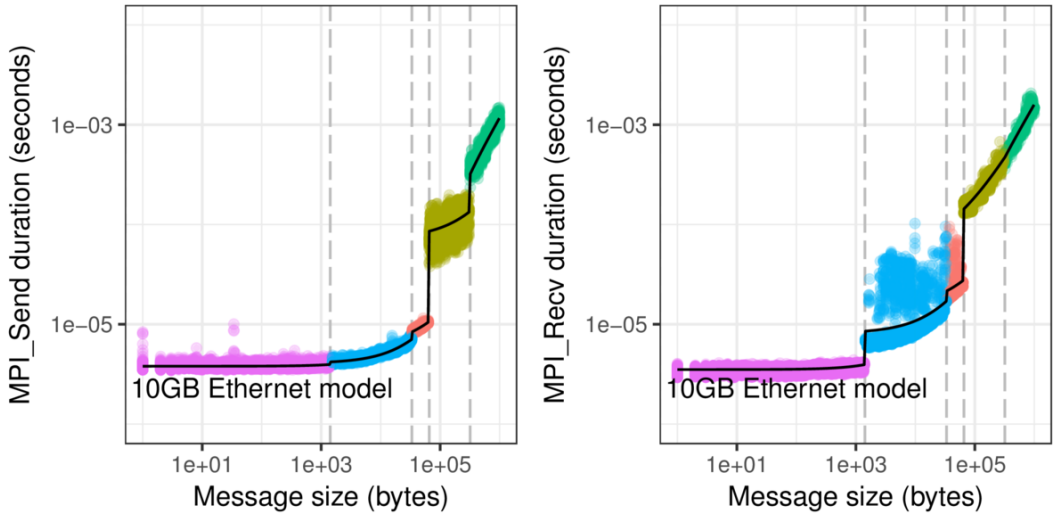


**Figure 4.7:** Faithful prediction of MPI applications requires to account for message-size dependent protocol and mode changes. By calibrating our cluster through a series of experiments [Deg+17], we determined that on this particular cluster and MPI implementation five main modes (each colored differently) with sometimes significant differences exist.

type (send, isend, broadcast, ...), tags, receiver, datatype etc.. Since each entry is time independent, the actual duration (at the time of recording) is not tracked.

Each of these two execution modes has its own benefits and disadvantages. The emulation is more responsive while the replay is significantly faster and can adapt to other platforms as well (if it is time-independent), but it comes with a significant overhead, namely to obtain the traces in the first place. Additionally, these traces can get prohibitively large for long executions and any change to the application or execution-based parameters (e.g., the number of processes) requires to generate another trace for these changed parameters. This means that for studies that frequently change algorithms or number of processes, it may not be feasible to use replay.

SimGrid supports the replay of several applications at the same time, permitting researchers to simulate actual workloads on shared machines (i.e., with realistic cross-traffic etc.). This is for instance used in BatSim [Dut+16a], a simulator for scheduling workloads that is based on SimGrid.

Thankfully, not only replay allows SimGrid to run simulations at large scale but also in some cases emulation. Emulations with several tens of thousands of processes have been shown to be possible [Deg+17]. However, some modifications to the application's source code may be necessary in order to accelerate the emulation significantly or to reduce the memory footprint so that a single node's memory suffices.

```
<config id="General">
 <prop id="smpi/os" value="0:3.79946267082783e-06:1.09809596167633e-10;1420:4
    .06752467953734e-06:8.98782555257323e-11;33500:6.01204627458251e-06:7
    .10122202865045e-11;65536:7.28270630967833e-05:1.9683266729216e-10;320000:0:0"
    />
 <prop id="smpi/ois" value="0:3.65848336553355e-06:1.33280621516301e-10;1420:3
    .83673729379869e-06:7.84867337035856e-11;33500:5.57232433176236e-06:6
    .5668893954931e-11;65536:4.17803219267394e-06:2.37460347640595e-12;320000:4
    .70677307448713e-06:3.38065421824938e-13"/>
 <prop id="smpi/or" value="0:3.51809764924934e-06:3.01847204118237e-10;1420:8
    .16124874852713e-06:2.66840481979518e-10;33500:1.49347740713389e-05:1
    .97645004617501e-10;65536:5.88893263987424e-05:1.29160163208845e-09;320000:0:0
    "/>
 <prop id="smpi/bw-factor" value="0:0.0489825651012801;1420:0
    .824385608826111;33500:0.600278012183156;65536:1;320000:0.536759617074721"/>
 <prop id="smpi/lat-factor" value="0:1;1420:2.16408517748122;33500:1
    .76905573216394;65536:2.9114462429055;320000:2.5981998109037"/>
 <prop id="smpi/async-small-thres" value="65536"/>
 <prop id="smpi/send-is-detached-thres" value="320000"/>
 [...]
</config>
```

**Figure 4.8:** The (shortened) calibration output obtained for the Grid5000 taurus cluster that serves as input to SimGrid. Note that the last value of `smpi/or` and `smpi/os` shows values that are 0: These are correct, since these messages will be sent synchronously (as defined `smpi/send-is-detached-thres`) and the entire cost is hence already accounted for.

```
9 init
9 compute 2.86401e+07
9 barrier

9 isend 10 4 165288 0
9 compute 1.33016e+11
9 recv 8 3 165288 0
9 wait 10 1 209
9 finalize

```

**Figure 4.9:** An example of a Time Independent Trace for a single rank. The first column denotes the rank id and the second column the action, each with their own parameters (e.g., flop for the `compute` action or receiver, tag, message size and data type for the `isend` operation)

### Speeding up computations

Recall that SMPI executes all instructions of all MPI processes. Instead of doing so on multiple nodes, like MPI, only a single node is used. Naturally, this increases the execution time significantly but not all computations are necessary since HPC applications have often very regular behavior. Serendepitously, when simulating an application, not all data is required to be consistent since only the characteristics of the application are of interest whereas the computed results are irrelevant. This leverages the following two solutions for a reduction of execution time: Online benchmarking and offline modeling kernels.

► **Online Benchmarking** If certain computations are known to have stable execution times over the course of the application run, benchmarking can be used to avoid executing them every single iteration. This process is online and implemented in SMPI, however, the user must designate the code (e.g., a loop) to be benchmarked by adding one of two currently supported macros (`SMPI_SAMPLE_LOCAL`, `SMPI_SAMPLE_GLOBAL`) to the source code. The user chooses the former macro when the execution time of the code is dependent on the rank and a per-rank benchmark therefore mandatory. The latter macro is therefore used when the execution time is expected to be uniform among all ranks. This reduces the number of benchmarks significantly, as the number of benchmarks is no longer a multiple of the number of ranks.

Once the modifications have been made and the application (re-)compiled with `smpicc`, the application can be executed as always. Each time the instrumented code is reached, SMPI automatically initializes its timers and executes the code. This is done until enough samples have been collected to obtain a reasonable evaluation of the expected duration. In this case, the code is no longer executed and only the average benchmark result is injected as a virtual delay into SimGrid. This means that once the benchmarking has finished, each iteration requires *constant* work, which is why this approach is suited only to code with stable behavior.

► **Offline Kernel modeling** For some computations that cannot use benchmarking due to non-uniform execution times, for instance, because they depend on parameters, a model of the corresponding kernel can often be created through a statistical analysis. Unfortunately, this process needs to be done offline and requires significantly more effort than benchmarking as SMPI currently does not provide an automated way for this. However, once the model has been obtained and implemented (e.g., by defining a new macro that replaces calls to the function with the model and a subsequent call to SimGrid), no executions of the kernel are

necessary and the model can be used to predict the runtime by solely inspecting the parameters.

The BLAS routine used for matrix-matrix-multiplications (`dgemm`) is a good example to demonstrate the power of kernel modeling. With a complexity of $\Theta(n^3)$, with $n$ designating the size of the matrix, multiplications rapidly become computationally expensive when the matrix becomes too large. Instead, a model (typically obtained through a linear regression) can be used. The cost to estimate the duration is then constant as only the model needs to be evaluated based on the parameters.

#### Reducing the consumption of memory

To emulate an application even of moderate scale, it is often insufficient to only take care of the computational cost of the application. A reduction of memory consumption is almost inevitable since without further tweaks, each rank consumes as much memory during the emulation as it would during a real-world simulation run. The available memory on the single node hosting the emulation can quickly even make runs with a few tens or hundreds of ranks impossible, as the memory allocated by all ranks may reach hundreds of GB or even several TB.

Under the assumption that the application's behavior does not depend profoundly on the data it operates on, memory can be saved via SMPI's `SHARED_MALLOC` macro. This macro is especially useful to avoid allocating unimportant datastructures for each rank and works as follows: When called for the first time (this is determined on a file/line basis), `SHARED_MALLOC` allocates one chunk of memory, just as a normal `malloc` would do. However, on all subsequent calls (at that code location), handles to the previously obtained allocation are returned, i.e., the allocation is shared by all ranks.

## 4.4 Runtime Support (StarPU-SimGrid): Simulating Dynamic GPU-based Applications

We have already discussed in Section 2.2.4 that runtime systems will see increased importance in the future, especially on heterogeneous systems that contain complex nodes (e.g., CPU + GPU) and that will benefit from other advantages provided by runtimes, such as scheduling, load-balancing or data-transfers. Unfortunately, as was previously mentioned in Section 4.3, SimGrid currently does not directly provide support for accelerators.

Implementing explicit support for each runtime in SimGrid (so that SimGrid can intercept and then forward calls from the application to the runtime) is tedious and would make support for runtimes depend on SimGrid's development. To ensure that not only current but also future versions of runtimes (with possibly different APIs) are supported as well, it is easier to have the runtime implement explicit support for the simulator.

As a proof-of-concept, support for SimGrid was implemented into StarPU [Aug+10], a runtime developed by Inria Bordeaux, and it was shown that faithful predictions are possible [Sta15]. In fact, the task-based approach leverages a clean separation of the control and compute part, which is not the case for SMPI. When used with SimGrid, StarPU only switches into simulation mode internally and does not notify the application about this. In order to use SimGrid, StarPU requires a single calibration run on the platform that is being simulated. This allows StarPU to assess the performance of available devices (e.g., GPUs) for submitted tasks and create offline kernel models. With this calibration, StarPU can then compute the computational cost for kernels during simulation runs and inject it directly into SimGrid. This means that SimGrid does not itself emulate the application code but only keeps track of the computations and messages as reported by StarPU.

## 4.5 Contributions to the SimGrid Project

It was already mentioned in Section 4.1.2 that SimGrid is currently undergoing a major rewrite through iterative refactoring. This changing infrastructure and complex simulated software-codes made it often necessary to contribute additional, refactored or fully rewritten code, tests, or bugfixes in parts of the code that I had started to (co-)maintain, most notably SMPI.

The development of SimGrid itself has taken an important amount of time during this dissertation project. A short overview over my main contributions is hence given in the following sections.

### 4.5.1 Platform description

The XML description is both verbose and rigid, which makes it ill-suited for the modeling of large and complex platforms. Usage of the scripting language `Lua` has been tested and was found to be promising to describe complex platforms programmatically and could be extended to be used for routing-algorithms as well. Since its current implementation needs to be maintained *besides* the XML-based default, meaning that all changes need to be implemented once for XML and

once for Lua, it was decided to not improve support for `Lua` but rather move to a `python`-based implementation in the future as bindings can be automatically generated from the C++ API. This has the benefit that by using existing, well-tested python-based XML-parsers, the C-based `FleXML` parser can be removed in the future. Furthermore, python is more universally known and may hence be easier to use.

### 4.5.2 PAPI support

During my research, several cases of application-dependent issues were encountered when simulating with SMPI, such as seemingly unexplicably optimistic performance estimations. The investigation proved to be extremely difficult. To determine the cause, further information, as for example provided by hardware counters, was necessary and lead us to identify cache-related issues (see Chapter 6 for details).

One way of retrieving these counters is through PAPI [Ter+10], a well-known, robust and portable API that provides means to obtain performance information by inspecting and reporting hardware counters of the CPU. Support for PAPI was contributed to SMPI and can be used to collect PAPI-counters for each actor (and not just for the entire simulation). Since SimGrid's own code can impact counters as well (e.g., total number of instructions), counter values must be stored *before* the execution of an actor and immediately *after* the execution has finished. The difference of these two counter values is then attributed to the actor itself and stored in a trace.

To investigate the actor's behavior with PAPI, counters of interest must first be declared. Currently, this is only possible for all ranks (i.e., all ranks inspect the same counters), but first steps were already taken to assign a different set of counters to each rank. However, this was not required for my work and hence not fully pursued.

### 4.5.3 Privatization

Privatization has been discussed in Section 4.3.1. These techniques have been implemented for several years but the implementation was prohibitively static and did neither support the introduction of daemons in SimGrid (i.e., of non-MPI based processes that execute work in the background) nor the dynamic addition of processes. Through a refactoring process, limiting code such as fixed-size arrays and double indirections were identified and removed.

Furthermore, instead of being stored in global variables, the privatization segments are now directly associated to each actor through a member in the corresponding class.

Although these refactorings were important to support daemons (required for our investigation of DVFS governors in Chapter 9), other projects have already benefitted from these changes as well: most notably, they leveraged scheduling-simulations with multiple applications executing at the sime time as done by the BatSim [Dut+16a] project.

This support for dynamicity is also required for future support of functions that spawn child-processes. These functions are also required by the MPI standard, for instance `MPI_Comm_spawn()`, and can now be implemented in SMPI.

### 4.5.4 Energy plugin

It was already discussed in Section 2.2 that energy consumption will play a critical role in the future. An interesting question is therefore the energy efficiency of an application or of an algorithm. The energy-model that was developed and evaluated during this dissertation (see Chapter 8 for details) was implemented in SimGrid as a plugin. This plugin does not depend on SMPI and is therefore available for any simulation using SimGrid.

### 4.5.5 DVFS plugin

To investigate further options for increased energy efficiency, a new plugin providing support for several DVFS governors will be presented in Chapter 9. This plugin allows users to select DVFS governors on a per-host basis and provides several classical algorithms (performance, powersave, on-demand, ...).

### 4.5.6 Load Balancing

Distributing the load inequally over all nodes is a well-known cause for subpar performance, even on medium-sized machines. The upcoming massive increase in parallelism of exascale machines will further exacerbate the situation (see also Section 2.2.1 for a brief problem presentation). The HPC community has hence declared load-balancing to be of critical importance for exascale performance [Don+11]. Unfortunately, understanding load related issues is often a very tedious task but without this knowledge the development of efficient algorithms is almost impossible. The development of better tools is therefore necessary [Don+11, pp. 40, 54]

and the simulation approach as provided by SimGrid certainly provides valuable insight. Alas, there was no high-level (i.e., without directly querying the SimGrid-core) API call to obtain the load of a particular node. To alleviate this situation, I developed a plugin that can be used to obtain the load of one or more arbitrary hosts at any time.

Unfortunately, (extreme) scale comes with (extreme) complexity. Load imbalances can therefore be easily misunderstood and solutions built into applications may work on one but not on other machines. Programmers should hence not attempt to implement their own load-balancer in their applications [Don+11, p. 57] as this may in fact lead to adverse results due to the complexity and diversity of platforms. On real machines, runtime systems such as StarPU [Aug+10] or Charm++ [Acu+14] are therefore required to relieve the programmer of the responsibility to load balance the application (see also the brief discussion in Section 2.2.4).

Naturally, given the importance of the subject, the evaluation of load-balancing techniques is very interesting. Rafael Keller Tesser studied the impact of selected Charm++/AMPI [Acu+14] based load balancers with SimGrid in his dissertation project [Kel18]. Unfortunately, this implementation was based on a forked SimGrid version that was quickly outdated due to the rapid development of SimGrid and its APIs. As a contribution to a joint work [Tes+18], I rewrote the entire code-base, including the load-balancing algorithm from Charm++. Chapter 9 details how this contribution was later used in my own research.

# 5 Experimental Methodology

Parts of this chapter were published previously as part of a preprint [Hei+17a].

## 5.1 Experimental Setup

All experiments for this thesis were executed using a cluster provided by the Grid'5000 infrastructure project [Bal+13]. Grid'5000 provides clusters at seven sites within France plus one in Luxemburg. We were only able to choose among Lyon-based Grid'5000 clusters, as only they offered a hardware wattmeter so that we could measure the power consumption of a node during our experiments. Power consumption of CPUs can be obtained on modern processors through hardware counters, however, we are interested in the total node consumption, making the usage of a wattmeter more convenient. We therefore chose particularly the Taurus cluster[1] because it was the largest and most "recent" (from 2011 ) cluster at the time. The measured power values were accessed through a specific server that queried 4 wattmeters located on-site. For each plug, a total of 3600 measurements per second were made, each with an accuracy of 0.125 W, that were subsequently averaged and returned to the wattmeter server as a single value.[2]

## 5.2 Factors Influencing the Experimental Results

In science, experiments are normally executed in highly controlled environments to exclude outside influences as much as possible. Unfortunately, computers are highly complex machines that are very difficult to use for experiments in such a "laboratory-like" manner. Improving the reliability of experiments requires diligence and consideration of numerous factors. An overview of these factors is given in Figure 5.1. For each "category" shown in this diagram, we will briefly discuss what the impact is and some issues we encountered. We have not observed other key factors, but this might become the case with new and more complex systems.

[1] See also https://www.grid5000.fr/mediawiki/index.php/Lyon:Hardware for more details.

[2] As of September 2018, these wattmeters have been replaced with newer equipment that allows users to retrieve 20 to 50 values per second, but this was too late for our experiments.

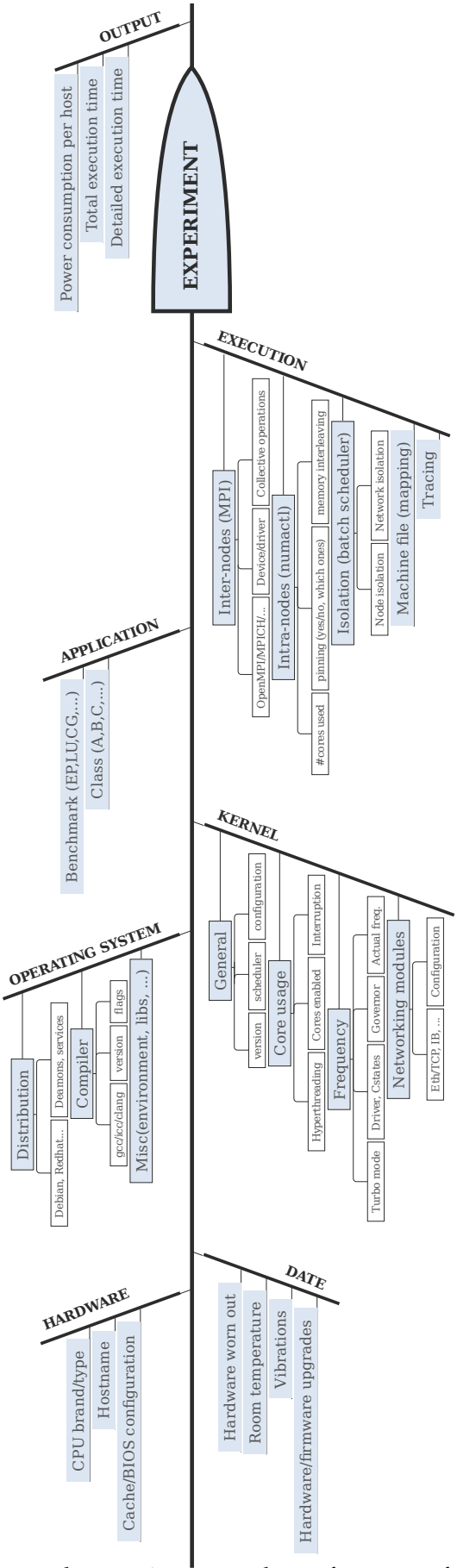


**Figure 5.1:** Factors that can have an impact on the performance of an HPC application.

### 5.2.1 Hardware

The first factor is the hardware used to run the experiments. It is known that even homogeneous machines can exhibit variability in performance and power consumption [Ina+15], for instance because the hardware was produced in different batches or factories. Another reason can be the position in the rack . As we will see in Chapter 8 (especially Figure 8.4 and Figure 8.5), this is also the case for the taurus cluster we used, which consists of in total 16 nodes, each with 2 `Intel Xeon E5-2630` CPUs that in turn have 6 physical cores (+ 6 Hyperthreads). Each CPU has a total of 3 cache levels: A 32 kB large first level (L1) cache, a 256 kB second level (L2) cache and a 15 MB third level (L3) cache. A total of 32 GB of main memory is available per node, split over two benches, with node interleaving disabled. All servers are interconnected through a (Full-Duplex) DELL Force10 S4810 switch with a maximum capacity of 1.28 Tbit. Up to 64 nodes can be connected to this switch with 10 Gbit Ethernet links. The port the node is connected to can influence the performance as well. Alas, we have no information on the wireings and therefore are unable to log this information. If in doubt about homogeneity, an individual profile should be created for each machine [Dav+12].

Figure 5.2 visualizes the setup of the Lyon site in 2016. As can be seen, two more clusters (Orion and Hercule) are connected to this switch as well. With a total of 4 nodes per cluster, Hercule and Orion are relatively small. Nevertheless, in order to avoid any interference of these clusters with our experiments, we fully reserved them as well, even though we did not use them. We were unable to ensure that neither the service machines nor the administration network were in use during our experiments, as we had no control over these, but we believe that executing all experiments on several dates, including nights and weekends, and obtaining consistent results is a strong hint that this was not the case.

### 5.2.2 Date

The specific date (and time) of a measurement can play an important role as well, since the state of important components can change over time. For example, a mechanical disc's performance may deteriorate due to more read/write errors or mechanical problems caused by wear and tear. The machine room's configuration may have changed (e.g., room temperature, node placement within the rack, ...) and especially temperature can cause a node's CPU to run slower [Myt+09, p. 266], consume more power or even to see a change in its clock drift [All87].

The research presented in this thesis focusses on rather macroscopic measurements and we have had no reason to believe that any of these reasons might be responsible

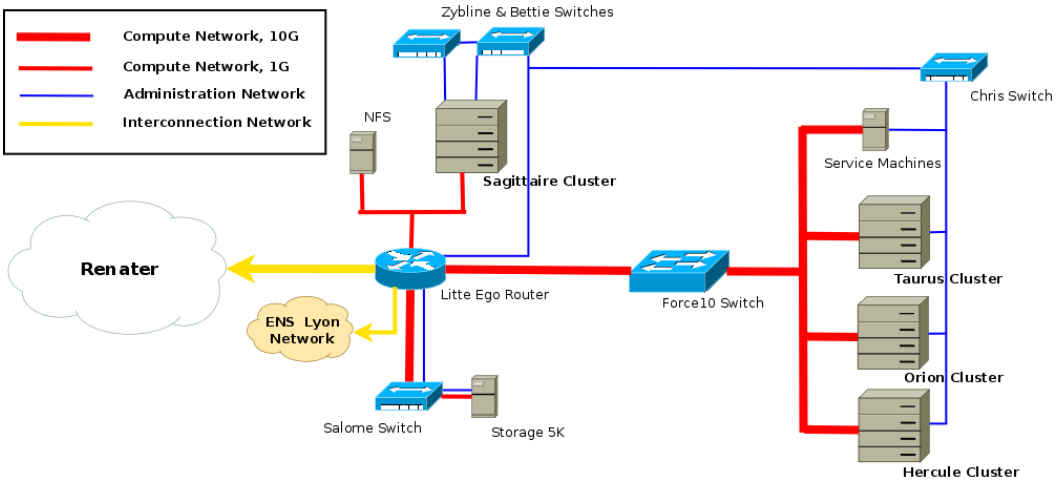


**Figure 5.2:** In Lyon, several cluster are connected to the same switch, which is why we reserved all of them except for the service networks and machines (due to lack of permissions) [Tea12].

| Machine | `memcopy` | `dumb` | `mcblock` |
|---|---|---|---|
| taurus-1 | 4558.395 | 2705.507 | 6478.633 |
| taurus-3 | 4100.912 | 2622.598 | 6132.107 |
| taurus-12 | 4622.562 | 2694.097 | 6412.366 |

**Table 5.1:** Even though all machines were provisioned with identical hardware, their bios settings proved to be different. This table illustrates the different results we obtained when testing for memory performance through the `mbw` benchmark on 2016-08-12.

for changes from one experiment to another. Yet, the hardware we used changed over time, for instance because the hardware wore out or through bios and firmware upgrades. We will see in Chapter 8 (Figure 8.5, page 96) that some changes to the hardware necessitated a re-calibration as the power consumption was largely increased for some nodes (`taurus-5` and `taurus-12`).

Firmware and bios upgrades (or changes) may also be responsible for different performances. For some experiments with the NAS-LU benchmark, we saw as much as 18 % difference between nodes, even though we had carefully setup all nodes in the very same manner. Among others, we used the `mbw` (Memory BandWidth) tool on the affected and non-affected machines. Table 5.1 illustrates `mbw`'s results on in total three machines: while `taurus-1` and `taurus-12` yield almost identical results, `taurus-3` performs significantly slower on the `memcopy` and `mcblock` tests.

We believed initially that the configuration controlling the "node interleaving" option was set to different values, but we found with the help of the Grid'5000 support team that this was not true and instead, an option called `MemOpMode` (Memory Operating Mode [DEL]) was configured differently. For `taurus-3`, it was set to

"AdvECCMode", which trades in performance for reliability by combining both 64-bit DRAM controllers in 128-bit mode. The other nodes were set to "Optimizer-Mode", an option that lets both controllers operate independently for maximum performance.

As a consequence of this bug, the Grid'5000 project adapted their verification scripts and can now detect different settings automatically.

### 5.2.3 Operating System / Software Stack

The third identified factor is the operating system. The Grid'5000 platform allows its users to deploy their own OS image on all reserved nodes. The platform provides pre-configured, versioned images that can be easily booted, however, they are also updated regularly. We wanted to avoid this and instead stored an image of the Debian Stretch operating system permanently that has all required libraries already installed. We can therefore guarantee that all of our experiments used the same libraries (and the same versions). This is important because e.g., library and compiler versions or their configurations through flags can also have a major influence on application performance and energy consumption: For instance, an update to a specific library may support automatic offloading to accelerators while a newer compiler may (no longer) manage to exploit a vector unit. We therefore keep track of the environmental variables (and possibly temporarily set compiler flags) through the `env` command but in general, the environment is not specifically modified between experiments. We are hence still subject to measuring bias [Myt+09] but in our case, its impact should be minimal due to the homogeneous results of our experiments.

Unfortunately, since we had started from a Grid'5000-based image for convenience, changes to the platform caused Grid'5000-dependent `systemd` services to timeout a few minutes after the machine was booted. Before the timeout, the service kept on using the CPU and experiments that were started immediately after startup were therefore affected in a seemingly random way due to a performance degradation that was irreproducible unless the machine was rebooted. A delay of a few minutes can help to prevent these negative impacts.

### 5.2.4 Kernel

The fourth category deals with the kernel version and its configuration. In fact, an application we used (NASPB-LU) is an extreme example of a bug in the Linux scheduler that was responsible for a slowdown of up to 2700 % when pinning the execu-

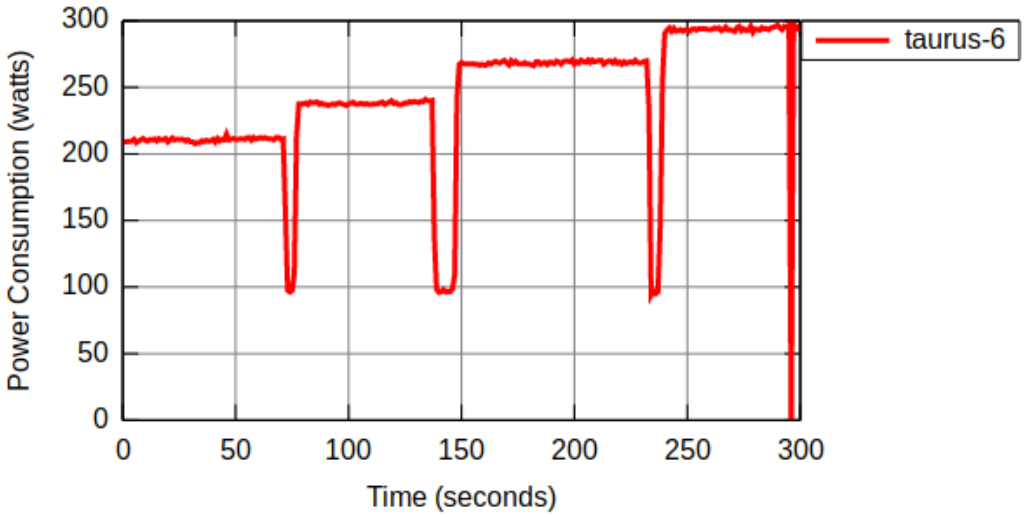


**Figure 5.3:** Different load and different core configurations can change the energy consumption significantly. In the beginning, 12 cores experience 80 % load, the second plateau represents 100 % load. For the third measurement, the CPU's turbo mode was enabled and for the last measurement, hyperthread was enabled as well. Note that the drops to around 100 W are short breaks in between the measurements and the system was idle. These tests were executed with FIRESTARTER [Hac+13] on 2018/08/13 from 11:15:00am to 11:20:00am on `taurus-6`.

tion to a particular subset of cores on NUMA machines, e.g., via `numactl` [Loz+16, e.g. Sec.3.2].

The drivers the kernel uses for hardware, such as the CPU, can be configured. Several drivers for the CPU are available and their performance and energy consumption can vary largely. Since we used Intel CPUs, the specialized `intel_pstate` driver was available but we selected the generic `acpi/cpufreq` driver to be able to set the frequency more easily through the `cpufreq` tool and the `userspace` governor. Modern machines (including the ones we used) support several frequencies, hyperthreading and a "turbomode", i.e., a mode that increases the frequency of individual cores through overclocking for a limited amount of time. These settings have a direct impact on the runtime of HPC applications and their energy consumption. Figure 5.3 illustrates four cases we ran with the FIRESTARTER [Hac+13] CPU benchmark: First, all 12 real cores are activated (no turbomode) and utilized with 80 % load. Second, the load is increased to 100 %, causing the dissipated energy to rise significantly but it still stays well below 250 W s. When turbomode is activated, which is shown in the third case, the energy consumption continues to grow which is also the case when hyperthreading is enabled (last case). With 100 % load, turbo mode and hyperthreading enabled, the consumption reaches almost 300 W s.

Since turbomode and hyperthreading are difficult to model, we disabled both altogether and fixed the used frequencies. However, we still allowed the CPU to

| Interrupt Name (IR-PCI-MSI) | CPU0 | CPU1 | CPU2 | … | CPU9 | CPU10 | CPU11 |
|---|---|---|---|---|---|---|---|
| 512000-edge 0000:00:1f.2 | 394 | 0 | 0 | | 0 | 0 | 0 |
| 35651584-edge eth0-TxRx-0 | 899873 | 0 | 0 | | 0 | 0 | 0 |
| 35651585-edge eth0-TxRx-1 | 1225593 | 248116 | 0 | … | 0 | 0 | 0 |
| 35651586-edge eth0-TxRx-2 | 1062111 | 0 | 95 | | 0 | 0 | 437964 |
| 35651587-edge eth0-TxRx-3 | 1016571 | 0 | 0 | … | 416930 | 0 | 0 |
| 35651588-edge eth0-TxRx-4 | 1063962 | 61012 | 0 | | 0 | 0 | 0 |
| 35651589-edge eth0-TxRx-5 | 1024896 | 0 | 0 | … | 0 | 0 | 0 |
| 35651590-edge eth0-TxRx-6 | 1063529 | 0 | 0 | | 0 | 0 | 0 |

**Table 5.2:** This (arbitrary) excerpt from `/proc/interrupts` shows how interrupts were not correctly re-balanced when disabling all cores (except for CPU0) and then immediately re-activating them. As can be seen, CPU0 handles almost all interrupts exclusively (the interrupts handled by other cores were handled before the cores were disabled and re-activated) continued to almost exclusively handle all interrupts, resulting in a performance degradation.

enter all energy-saving modes (called "C-states"). Our choices regarding frequency, hyperthreading and turbomode were important for the power consumption when running an application, but they had no effect on the idle consumption. Naturally, the energy-saving modes had a large impact on the idle power consumption: When all C-states are disabled and the CPU is asked to poll constantly (for maximum responsiveness), the observed consumption was more than $220\,\mathrm{W\,s}$.

During our first experiments, hyperthreading was disabled by default within the bios of all nodes and was only subsequently activated by the Grid'5000 project. Once it was activated, we adapted our scripts to disable hyperthreading immediately after deploying the nodes. Interestingly, we noticed that when *all* cores but Core0 are disabled and then all *physical* cores (i.e., no hyperthreading) are re-activated, interrupts were not correctly re-balanced and were only executed by Core0 even though the `irqbalance` package was installed. In our experiments, this caused a performance degradation that reached up to $30\,\%$ when using $12$ nodes and $144$ processes ($1$ process per physical core). We found that this occured since we disabled first *all* cores and then only enabled the correct number of cores ($1$,$4$,$8$,$12$) for each experiment (i.e., we had a growing number of cores for each experiment). By changing this to a decreasing number of cores for each experiment, i.e., by disabling first only the number of cores that are not required, this bug can be circumvented. The interrupts are re-balanced correctly once all cores (including hyperthreads) are re-enabled.

Table 5.2 presents an excerpt from `/proc/interrupts` that illustrates how interrupts are handled once this bug is active: The non-zero values outside the `CPU0` column were interrupts that were correctly handled *before* all cores were disabled. Subsequently, with the bug enabled, all interrupts are only handled by `CPU0` which is why the numbers in this column are significantly larger. It is hence important

to know *how* cores have to be disabled; in our case, we found that disabling *only* the hyperthreading is sufficient. We did not specifically track the interrupts before and after an experiment, even though this is simple and could help later to answer questions related to (unexpected) performance degradations.

When experiments are executed on several nodes, their performance is impacted by the networking module, as for instance different protocols can achieve a different usable bandwidth and latency. Network cards have a relatively constant energy consumption (independent of load) and can consume up to half of a CPU's power, but recent developments such as On/Off links can reduce this consumption [MN15, Chapter 2]. However, this reduction comes at a cost of a reduced responsiveness when the network card enters an energy saving mode (because a wake-up latency is added). To avoid this and the corresponding changes to our models, we took no further measures of energy savings regarding the interconnect.

### 5.2.5 Application

The fifth category deals with the user application itself. Generally, communication and computation bursts alternate (even though communication is nowadays often overlayed with computation). Its performance and energy consumption is impacted by the executed instruction mix (e.g., integer or floating point, memory and cache accesses and therefore also cache misses, ...). Applications can behave homogeneously along time and even across several nodes for the same workload. When studying the same application for different workloads, instruction mixes may differ and cache-effects can vary due to larger/smaller input.

### 5.2.6 Execution

The sixth category contains elements that are relevant for the execution of the application, such as the MPI implementation, its version and its configuration, such as the concrete algorithmic implementation of a collective communication function. Many MPI implementations select this implementation based on the context (e.g., the network partition). Major implementations (such as Open MPI) ship for this reason with several implementations for a single MPI collective. As previously mentioned (see Table 4.2 on page 47 for a breakdown), SMPI supports a large number of collective algorithms and for good performance predictions, the algorithmic implementation should be configured accordingly. For our experiments with HPL, simulation results did not depend on the right algorithmic implementation because HPL ships with its own implementation of all used collectives.

Before the execution of an experiment, we decided how many cores and processes we want to use and subsequently used Open MPI's mapping support to pin each process to a core (i.e., process migrations were prohibited) via the `--bind-to core` option whereas the mapping of ranks to cores was fixed via `--map-by core`. We also configured Open MPI to continue using the tcp stack via `--mca btl tcp,self`.

All nodes were using the NFS drive only for `/home` and accessed all shared libraries locally. Since all used applications consist of small binary files, the delay caused by accessing the binary over the network is minimal. Generated data (such as traces), however, were first stored in memory, then dumped to the local disk and finally transferred to the NFS for permanent storage as they are usually large enough to impact the overall performance.

### 5.2.7 Output

The last category is the output of the executed application. We generally capture and store all output (`stdin` and `stderr`). When executed via Open MPI, each line receives its own timestamp via the `--timestamp-output` option.

### 5.2.8 Data Analysis

In a final step, the obtained experimental data is analyzed. Although this stage has no influence on the *actual* behavior (e.g., performance, power consumption) of the application, analyzing obtained data incorrectly can impact the *reported* behavior (e.g., through plots or statistics). The data analysis is therefore just as important and must be executed with diligence.

During this thesis, we have used the popular `org-mode` plugin [Sch+12] for `emacs` to analyze data (mostly through R [R C16]) and to generate plots, often using a *literate programming* style. All code for figure generation is stored in a single, sectioned and commented `.org` file. Since we published also experimental data, our analyses can be verified by interested researchers and figures can be reproduced easily (with the same data). Although this does not *prevent* errors from being made, it helps to *find* them and especially to answer questions on the used techniques.

## 5.3 Network Calibration

### 5.3.1 Network

Recall from Section 4.3.2 that SMPI requires several configuration options, e.g., which transmission protocol is used for a specific message size. Finding these values is done in two steps: First, a calibration script [3] is executed on two nodes of the target platform that runs several rather simple tests, for example *Ping* (to determine the time spent in MPI calls) and *PingPong* (to determine transmission delay) [Deg+17, p. 2392]. In a second step, the raw measurements (and not just averaged values, as this could cause for example behavior smoothing) are analyzed with a script written in `R` [R C16], resulting in a piecewise linear regression fitted to the data. This yields output similar to the one in Figure 4.7 (page 54). Visualizing the results helps the user to determine if the breakpoints for the transmission protocols are set correctly or if they need to be adapted.

### 5.3.2 Hardware Limitations

In some cases, intermediate hardware such as switches can limit the total performance of the network, e.g., when the switch has a lower capacity than all links connected to it [Deg+17, p. 2392]. To determine these bottlenecks and their capacity, network saturation tests have to be executed. The resulting capacity can then be modeled in SimGrid by using a so-called *limiter link* (see also Section 4.2.1). A limiter link is a concept that limits communication speeds by adding a common bottleneck. For this, the limiter link is declared as an actual link and its bandwidth is set to the maximum switch capacity. To limit the total communication speed, all routes that are known to go through the switch are declared to use the limiter link as well, i.e., all communications that go through the switch now have at least one link in common that becomes a bottleneck once the switch's maximum capacity is reached.

We executed such a saturation test that showed that no such limiter was needed because the switch we used for our experiments (DELL S4810) was well-provisioned and our testing environment was isolated from third party impact since we reserved all nodes connected to the switch.

[3] For all required scripts and examples, see https://gitlab.inria.fr/simgrid/platform-calibration

# 6 Contribution: Modeling Multi-Core CPUs

This chapter contains figures and text that was published in [Hei+17b].

## 6.1 Problem

As we have seen in Sections 2.1.2 and 2.2.2, nodes consist today of multi-core CPUs that are often supplemented with accelerators. Traditional single-core nodes, on the other hand, are dying out. To support these complex platforms, modelization of multi-core nodes is imperative. Alas, previously published work on SMPI and its scientific validation dealt only with networking aspects and did not make use of multi-core or many-core nodes.

Node-internal parallelism, however, comes with its own challenges. If more than one CPU is available, two processes executing on the same node could be using a distinct CPU (as opposed to different cores) and would therefore be impacted differently by cache-effects than when both processes execute on the same CPU but on different cores. These effects can be considerable and must be accounted for. Recall (from Section 4.2.1) that SimGrid does not support the configuration of a number of *CPUs* but that it only allows a user to configure the number of *cores* a node contains, making it impossible to account for cache-effects.

In SimGrid's model for computational resources (e.g., cores), each resource is assigned a capacity specified in $\mathrm{flop/s}$, which is then distributed among all processes using that resource: Let $C$ denote the capacity of a node, $p$ the number of processes and $n$ the number of total cores. If less processes use the CPU than cores are available ($p \leq n$), each process advances at rate $C$ (i.e., each process runs on a dedicated core). In the other case, i.e., more processes use the CPU than cores are available ($p > n$), SimGrid simplifies resource allocation by ignoring scheduling policies and other overhead (e.g., for context-switching) and shares the total amount of computational power ($C \cdot p$) evenly among all processes, i.e., each process progresses at rate $\frac{C \cdot p}{n}$.

For CPU-bound applications, this model works rather well, but for memory bound applications inaccurate simulation results can be the consequence of online simula-

tion because the simulation cannot be executed under the exact same conditions as a real-life run.

The following two examples illustrate this:

1. Consider an application (such as the quantum chemistry code BigDFT [Gen+08]) that contains coarse-grained, memory-bound computation blocks. In this application, each process shall use the L3 cache and/or the memory bus heavily. Normally, the L3 cache is shared between all cores of the same CPU (see Figure 2.3 (page 10)). In consequence, processes running in parallel on the same node will contend for this cache-level. SimGrid, however, executes contexts (representing a process) in mutual exclusion. This means that, when measuring the duration of the computation, no contention on the L3 cache occurs (due to the lack of parallelism). In this particular scenario, the measured time tends for this reason to be (very) optimistic.

2. Now consider an application with many small computation phases relying heavily on the (per-core) L1 cache, such as the NAS-PB LU benchmark. Each phase shall be followed by a call to an MPI function (for instance, `MPI_Iprobe` to check for progress of an ongoing communication). Each time the MPI function is entered, SimGrid will cause the currently executing context to yield and transfers control to another context. If privatization of global/static variables via `mmap()` (see the discussion in Section 4.3.1) is used, the context-switch will remap the `data` segment in memory via `mmap()`. Recall that the main disadvantage of `mmap()` for privatization is the mandatory cache-flushing. All cache levels are hence always cold after a context-switch and in this particular example, the now executing process has to retrieve data from the main memory instead of just the L1 cache. This means that immediately after the context-switch, the executing process suffers a performance degradation for a very short time. Since many small computation phases are used (and therefore many context-switches are executed), the performance loss is non-negligible when compared to a real execution, where no call to `mmap()` is necessary and the MPI process runs on a dedicated core with a dedicated L1 cache. The estimated time therefore tends to be (very) pessimistic in this case.

Both scenarios are common in HPC applications of interest, but it is difficult to know beforehand in which situation an application falls. To offset incorrect timings caused by online emulation and privatization, SMPI must speed-up or slow-down some parts of a user's code. Which option is required depends not only on the application's memory access pattern, but also on the node's memory hierarchy (e.g.,

cache sizes) as small caches may force the real application to access the memory more frequently as well. Lastly, the number of processes contending for the caches plays an important role as well.

## 6.2 Proposed Solution

In HPC, compute kernels are generally regular and their performance depends mostly on the machine and the workload. When emulating an application, the time of the computations can be measured, but it will represent the performance of the kernels *in emulation mode* on the host. Unfortunately, this is not helpful as we are interested in the performance of the application on the target machine under *real-life conditions*. The approach we propose computes first correcting factors that allow SMPI to convert the measured time to the corresponding time on the target machine. Computing these factors on the kernel level is possible, however, working on the kernel level is complicated as it requires some advance knowledge of the code. We therefore propose to compute the correcting factors based on the executed code between two MPI calls (called a *code region*). We also assume that the used code is regular, i.e., that the execution time of code regions does not change along time (only for different workloads). The comparison of MPI and SimGrid executions is *machine dependend* since the complexity of a kernel can normally be described through a polynomial function whose coefficients are machine dependend.

The first step towards an unbiased emulation and correct performance estimations is to run and trace the application with MPI on a single node. One process per core is started and the workload is choosen small enough to execute it. [1] The trace recorded during this real-life (RL) execution is illustrated exemplarily in Figure 6.2 (designated as $\texttt{Calibration}^{\texttt{RL}}$). It contains the rank, start time, execution duration (in micro-seconds) and state (MPI function name / computation) for all MPI calls and computations.

The second step consists of executing the same application and the same workload via SMPI. This yields another trace, generated by SMPI's own tracing mechanism. An excerpt (labelled as $\texttt{Calibration}^{\texttt{SMPI}}$) is shown in Figure 6.2. This trace contains columns that do not exist in the MPI trace: filename and linenumber (the rank-column was left out for illustration purposes). They are used to identify what we call a *code region*: Consecutively executed computations, possibly spanning several files and functions, uninterrupted by any MPI call. While blank for entries marking simple computations, these columns contain the exact location for every

[1] The exact commands we used can be found in this file: https://gitlab.inria.fr/fheinric/paper-simgrid-energy/blob/master/paper-cluster-figures.org, section "computation"

MPI call. This means that each region can be identified through the MPI calls surrounding it. In Figure 6.1, three regions are identified and visualized through different colors.

The node (or even the hardware) used for the MPI and SMPI runs do not necessarily need to be the same: SimGrid already allows users to simulate other hardware by adjusting the speed parameter for the executing host. However, only the host that runs the MPI part can be *simulated*, while no other host than the host that runs the SMPI part can faithfully execute the simulation.

After downloading both traces to the same machine, a script written in `R` is used to automatically align them (see Figure 6.2). The total duration (over all executions) of each code region $c$ is subsequently computed and compared to the other trace: The resulting speed-up / slow-down factor $s$ is set as $s(c) = \frac{t_{SMPI}(c)}{t_{RL}(c)}$. Here, $t_{SMPI}(c)$ denotes the total time spent in $c$ by all ranks in $\texttt{Calibration}^{\texttt{SMPI}}$. $t_{RL}(c)$ is defined accordingly for $\texttt{Calibration}^{\texttt{RL}}$.

Once computed, these factors are saved in a `.csv` file that consequently serves as input to an arbitrary number of simulations of the calibrated application. Figure 6.2 presents an excerpt of a file we obtained for `NAS-LU`.

The values of $s(c)$ can be interpreted as follows: When SMPI executed the region $c$ faster overall, a slow-down is required and therefore $s(c) > 1$ results. Every time $c$ gets executed, SMPI measures the execution time $t'$ and increases it by injecting a virtual delay of $s(c) \cdot t'$ into the simulator. In the example, region $18$ was executed faster by SMPI and hence requires a slowdown of a factor of $1.9696$.

Analogously, when SMPI executed $c$ slower than MPI, $s(c) < 1$ causes the injected virtual delay to shrink to $s(c) \cdot t'$. In Figure 6.2, region $43$ requires a speed-up of $0.8933$ because SMPI required more time to execute this region.

To use this approach, no changes to the code base are necessary. The location tracing mechanism was built through macros defined in SMPI's `mpi.h` header. This header is included in every MPI application in both Fortran and C/C++. At compile time, the macros only add a new function call

```
smpi_trace_set_call_location(__FILE__, __LINE__);
```

before each MPI function. This function only stores the location of the last two called MPI functions and then returns. This information can subsequently be accessed by the tracing mechanism. Obtaining the filename and linenumber at runtime is hence computationally very cheap.

```
...
Region 43 if( iex .eq. 0 ) then
Region 43     if( north .ne. -1 ) then
                call MPI_RECV( dum1(1,jst),
              >     5*(jend-jst+1),
              >     dp_type,
              >     north,
              >     from_n,
              >     MPI_COMM_WORLD,
              >     status,
              >     IERROR )
Region 2        do j=jst,jend
Region 2           g(1,0,j,k) = dum1(1,j)
Region 2           g(2,0,j,k) = dum1(2,j)
Region 2           g(3,0,j,k) = dum1(3,j)
Region 2           g(4,0,j,k) = dum1(4,j)
Region 2           g(5,0,j,k) = dum1(5,j)
Region 2        enddo
Region 2      endif
Region 2
Region 2      if( west .ne. -1 ) then
                call MPI_RECV( dum1(1,ist),
              >     5*(iend-ist+1),
              >     dp_type,
              >     west,
              >     from_w,
              >     MPI_COMM_WORLD,
              >     status,
              >     IERROR )
Region 3        do i=ist,iend
Region 3          g(1,i,0,k) = dum1(1,i)
Region 3          g(2,i,0,k) = dum1(2,i)
Region 3          g(3,i,0,k) = dum1(3,i)
Region 3          g(4,i,0,k) = dum1(4,i)
Region 3          g(5,i,0,k) = dum1(5,i)
Region 3        enddo
Region 3      endif
...
```

**Figure 6.1:** Excerpt of the NAS LU-PB (`exchange_1.f`) highlighting code regions between any two MPI calls.

Calibration$^{RL}$ trace (MPI)

| rank | start (s) | duration (mus) | state |
|---|---|---|---|
| ... | ... | ... | ... |
| 1 | 1.643388 | 1293 | mpi_allreduce |
| 1 | 1.644681 | 62 | Computing |
| 1 | 1.644743 | 82 | mpi_barrier |
| 1 | 1.644825 | 6454 | Computing |
| 1 | 1.651279 | 549 | mpi_recv |
| 1 | 1.651828 | 474 | Computing |
| 1 | 1.652302 | 53 | mpi_send |
| 1 | 1.652355 | 2 | Computing |
| 1 | 1.652357 | 15 | mpi_send |
| 1 | 1.652372 | 359 | Computing |
| 1 | 1.652731 | 11 | mpi_recv |
| 1 | 1.652742 | 462 | Computing |
| 1 | 1.653204 | 15 | mpi_send |
| 1 | 1.653219 | 1 | Computing |
| 1 | 1.653220 | 9 | mpi_send |
| 1 | 1.653229 | 376 | Computing |
| 1 | 1.653605 | 22 | mpi_recv |
| 1 | 1.653627 | 465 | Computing |
| 1 | 1.654092 | 16 | mpi_send |
| 1 | 1.654108 | 1 | Computing |
| ... | ... | ... | ... |

Calibration$^{SMPI}$ trace (uncorrected SMPI)

| start (s) | duration (mus) | state | Filename | Line |
|---|---|---|---|---|
| ... | ... | ... | ... | ... |
| 0.550426 | 1130 | mpi_allreduce | l2norm.f | 57 |
| 0.551556 | 18 | Computing | | |
| 0.551574 | 47 | mpi_barrier | ssor.f | 74 |
| 0.551621 | 5303 | Computing | | |
| 0.556924 | 617 | mpi_recv | exchange_1.f | 30 |
| 0.557541 | 608 | Computing | Region 3 | |
| 0.558149 | 4 | mpi_send | exchange_1.f | 113 |
| 0.558153 | 12 | Computing | Region 17 | |
| 0.558165 | 4 | mpi_send | exchange_1.f | 130 |
| 0.558169 | 652 | Computing | Region 18 | |
| 0.558821 | 8 | mpi_recv | exchange_1.f | 30 |
| 0.558829 | 587 | Computing | Region 3 | |
| 0.559416 | 5 | mpi_send | exchange_1.f | 113 |
| 0.559421 | 12 | Computing | Region 17 | |
| 0.559433 | 5 | mpi_send | exchange_1.f | 130 |
| 0.559438 | 699 | Computing | Region 18 | |
| 0.560137 | 9 | mpi_recv | exchange_1.f | 30 |
| 0.560146 | 597 | Computing | Region 3 | |
| 0.560743 | 4 | mpi_send | exchange_1.f | 113 |
| 0.560747 | 14 | Computing | Region 18 | |
| ... | ... | ... | ... | ... |

Merging traces

Region-based speedup/slowdown factors

```
"bcast_inputs.f:37:exchange_3.f:42",0.1655     Region 1
"exchange_1.f:30:exchange_1.f:48",14.6704      Region 2
"exchange_1.f:30:exchange_1.f:113",1.2967      Region 3
"exchange_1.f:30:exchange_1.f:130",1.2994      Region 4
. . .
"exchange_1.f:113:exchange_1.f:130",11.7101    Region 17
"exchange_1.f:130:exchange_1.f:30",1.9696      Region 18
...
"exchange_3.f:288:exchange_1.f:30",0.8933      Region 43
...
```

**Figure 6.2:** Trace merging process used for the NAS-LU benchmark to compute region-based speedup/slowdown factors and correct the simulation.

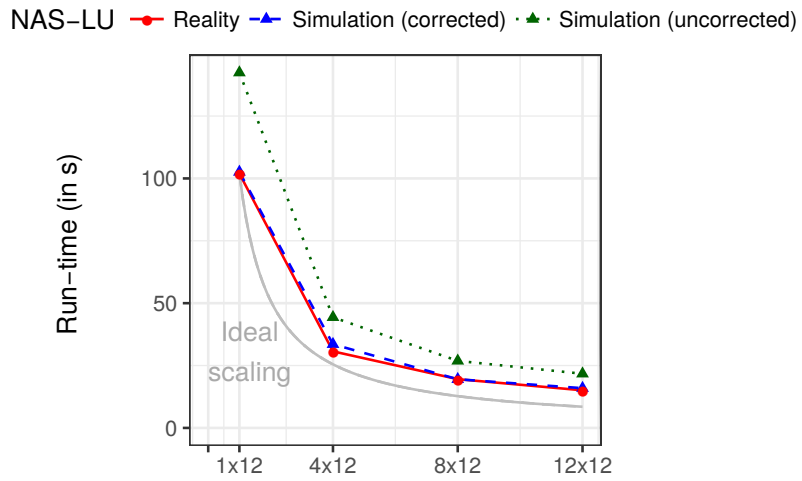


**Figure 6.3:** Comparison of calibrated (blue) and uncalibrated (green) runs of LU with real experiments (red) and ideal scaling (grey).

## 6.3 Performance Evaluation / Effectiveness

To evaluate our approach, an application needs to be of one of the two cases mentioned in the previous section. We found that NAS-EP and HPL were not of any of these types, as they are mostly computation-bound. The calibration files we obtained for NAS-EP had most factors very close to 1, implying that the applied corrections are not very important for accurate results.

On the other hand, when `mmap()` privatization is used, NAS-LU requires calibration. Indeed, NAS-LU is a memory-bound application as it frequently loads and writes matrix elements.

Figure 6.3 illustrates four scenarios, executed on one, four, eight and twelve nodes, each consisting of 12 cores: Ideal scaling (i.e., increasing resources decreases runtime proportionally) is depicted in grey, the real-life MPI execution in red, the uncorrected simulation in green and the corrected simulation in blue. The red and blue lines almost perfectly overlap as the corrected simulation yields near-perfect results. Represented by the green line, SMPI's uncalibrated computed runtime estimation yielded an error of around 20 % to 30 %.

Another interesting question is which regions contribute the most to skewed runtime predictions. In Figure 6.2, values for regions range from significant speed-ups like 0.16 (region 1) to strong slow-downs (14.67, region 2) while other regions are relatively close to 1 (regions 3, 4 and 43). In our analysis, we found that regions that require a strong speed-up were executed very few times (even sometimes only once) while the regions with large slow-down factors require very little time for

execution but are called excessively (possibly several hundred thousand times). We found that regions with extreme values contribute only neglegibly to the runtime estimation and are hence less important.

## 6.4 Limitations

### 6.4.1 Technical limitations

As was already explained in Section 6.2, we use macros to insert a function call with the `__FILE__` (i.e., current source file) and `__LINE__` (i.e., current line in that source file) macros as parameters.

Our *technical* implementation can fail for applications (such as HPL) that wrap some or all of their MPI calls in an additional layer, which is often done to facilitate e.g., logging, portability and maintainability. Note that in this case, the `__FILE__` and `__LINE__` macros will now always evaluate to the filename and line of the file containing that wrapped MPI call. This means that the start and the end of code regions are now identified by a generic location, which can lead to distinct code regions to be considered to be the same.

A solution to this problem could be to use the entire call-stack as a means to identify code regions, as implemented for example by SST/DUMPI [Jan+10]. For our studies, however, this was not needed.

Another limitation of our approach is that only regions that are actually *executed* can also be identified. Switching a boolean parameter could possibly execute another conditional branch and lead to very different regions. When a region was not executed during the calibration, no correction factor can be applied. This may lead to the need to re-calibrate even for small parameter or plattform changes.

It may also be necessary to calibrate every *kernel* of the application and not just every *region* (i.e., all instructions that just happen to lie between two consecutive MPI calls). This approach is much more intrusive (since the actual source code needs to be modified) but has the advantage to separate e.g., cpu-bound kernels from memory-bound ones. The impact of contended memory can therefore be isolated to a specific part of the application, which may be important when the workload and therefore the time spent within the kernels changes.

We already mentioned in Section 6.2 that we assume that the behavior of kernels does not change along time. Indeed, since the correcting factors are computed

by averaging all obtained samples, a temporal evolution of kernel runtime would cause our approach to be able to apply only a single corrective factor. This becomes especially problematic when a specific workload executes a kernel many times but mostly with either very fast or very slow executions: Since the average was computed on a particular ratio of these fast/slow executions, any shift in that ratio could not be offset since the right factors would not be known.

### 6.4.2 Scaling limitations

The basis of this approach is a comparison between a real execution and a simulated execution on a single node. It is not surprising that, once calibrated, the simulation of a single node returns almost exactly the same results as the real execution since we slowed-down or sped-up each code region to have the same *overall execution time* for simulation and real-life execution.

These results are, however, not obvious when changing the number of processes, possibly spanning several nodes, or the workload. The computed factors are intuitively affected by cache re-use and cache locality. It is therefore reasonable to suppose that they are sensitive regarding the workload. For weak scaling, i.e., when the workload per processor is kept constant and added processors are used to solve a larger problem, this approach can be expected to work. For strong scaling studies, i.e., when the problem size does not vary with the addition of new processors, this approach may not be feasible because each processor works on a smaller part of the problem, which may lead to different cache usage.

Figure 6.3 shows a strong-scaling study we conducted with NAS-LU. Our approach continued to work even though the workload was split between all processors involved in the computation. We believe that this may be explained by well-optimized code that results in small to no differences in correction factors for varying workloads.

Another limitation is the need to obtain the traces first. Since trace size typically varies with workload size (since more events are to be traced), storing and analyzing these traces requires sufficient disk space and also enough memory, respectively.

In our experiments with workload size `C` and 12 cores on a single node, the `.csv` file obtained by converting the original trace reached 7 GiB. Note that the host's 32 GiB were sufficient to compute the application *and* hold the entire trace in RAM before flushing it to the disk. Intermediate flushing may be necessary for larger workloads and can cause the *traced* real-life execution to perform slower.

We have only validated our approach with class C. Future processors that contain more cores should in principal also be able to use this approach. For NAS-LU, 144 processes produced a trace of 5.5 GiB for class B and 11 GiB for class C when traced with TAU but class D was impossible to trace due to the memory footprint. This is an issue of TAU and other tools, such as Scalasca, are able to acquire even larger instances [Mar14, Table 3.29].

When dumping traces to disk during the MPI run, the calibration would be computed with this overhead, meaning that SMPI may not perfectly simulate the *untraced*, faster execution. Another approach is to compress traces while gathering them, for example with ScalaTrace [Noe+09], however, the used heuristics make assumptions that do not hold true for every application. This approach is hence not universally possible.

Another limitation is that our method is oblivious to time. The correction factor for a region is computed once and applied after each execution. A rank executing a kernel that is slowed down at the wrong time might cause other ranks to wait for it, which may lead to further overall slowdown.

### 6.4.3 Future Work

We have calibrated the application and executed the simulation on the same node of the `taurus` cluster and obtained good results. SimGrid permits users to simulate applications on machines that are different than the original machine, for instance their laptops. Even though this should work just as well, we have not yet tried this.

# 7 Contribution: Modeling Intra-Node Communications

This chapter contains figures and text that was published in [Hei+17b].

## 7.1 Problem

Recall from Section 4.3.2 that SMPI extends SimGrid's network model (based on fluid flows) with a hybrid model that integrates LogGPS-like parameters. This allows SMPI to predict communication costs more accurately. In previously published work, this model has been extensively tested and evaluated using *network* communication, i.e., for communication between several nodes. However, when communicating within the same node, messages can be passed through the shared memory instead of the network card. This comes with several implications: First, the available bandwidth and latency is different. Second, protocol changes (e.g., from eager to asynchronous) can occur at different message sizes as message which means, third, that the observed performance will differ and therefore, previously obtained linear models must be recomputed.

## 7.2 Solution

To distinguish between local and remote communications within SMPI SimGrid uses a so-called loopback link when communicating from a host to itself. The characteristics (bandwidth, latency) of these loopback links should thus be specified. If the bandwidth and latency are not known up-front, they can be determined experimentally (through measurements). A tool such as `iperf` that relies on TCP/IP for all communication can help to give a first idea of the *physical* bandwidth of the loopback. The *effective* bandwidth that can be reached through MPI can differ, since MPI can change the protocol.

Once bandwidth and latency are known, a loopback link can be added for each node to the platform file. It is declared just like any other link (see Figure 4.6 on page 45). Although it is impossible to expressly designate it as a node-internal loopback link

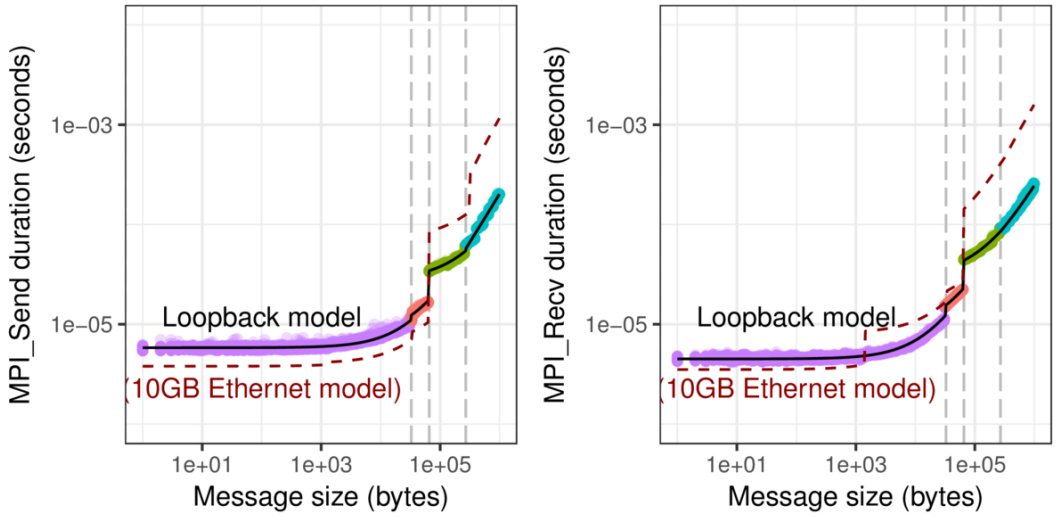


**Figure 7.1:** Intra-node communications can rely on different protocols than inter-node communications. To calibrate the loopback link, we executed the same calibration procedure as for inter-node communications (see Figure 4.7 on page 54). As can be seen, there is signifcantly less jitter for local communications than for inter-node communications. We furthermore found that small messages were sent *faster* over network links than shared memory since the send operation was executed asynchronously for remote destinations. For large messages, the loopback was almost an order of magnitude faster which is expected due to the much faster bandwidth.

(because SimGrid does not support such a notion), it can be regarded as such when the only route that uses this link is used for communication from a node to itself.

Finally, the transfer modes (eager, asynchronous, ...) must be calibrated analogously to the network (Section 5.3) by running the calibration procedure on two cores of the same node. This allows SMPI to use the right mode for each message size and account for the communication overhead (start-up cost and cost-per-byte).

As can be seen in Figure 7.1, the loopback calibration experienced significantly less noise throughout all identified modes than the calibration for the network (Figure 4.7 on page 54), even though we had isolated the entire cluster for the Ethernet calibration. The number of modes remains unchanged and only minor modifications to the break points were necessary. At the same time, the computed regressions have very little in common: Small messages appear to be *faster* when transmitted via the interconnect because they are small enough and hence sent asynchronously. For large messages, communications going through the shared memory can be up to one order of magnitude faster which is consistent with expectations.

## 7.3 Performance Evaluation / Effectiveness

We evaluated the effectiveness of our solution using the HPL benchmark. Figure 7.2 compares the measured runtime on the actual cluster with two simulations: One simulation ignores the specific characteristics of local communications (and hence treats them like inter-node communications) and the other one uses a correctly defined loopback (i.e., with a bandwidth of around $41\,\mathrm{Gbit/s}$) but does *not* use different values for the options given in Figure 4.8 (page 55). Section 7.4 will explain why we made this decision. This means that startup cost and overhead per-byte remain the same for local and remote communications. The experiments showed that this approach is sufficient for HPL. The most important change is the increased bandwidth and calibrated simulation results match the real execution perfectly. When the bandwidth is too slow, all twelve processes use a single link that becomes the bottleneck. In this case, the prediction is off by almost $25\,\%$ for the single-node because the difference between $10\,\mathrm{Gbit/s}$ ethernet links and loopback links are too significant. The importance of the bandwidth for the single-node scenario is confirmed by Figure 7.3, which compares the same real-life results with a loopback that has with $25.6\,\mathrm{Gbit/s}$ only $62.5\,\%$ of the measured bandwidth and one that is with $5.12\,\mathrm{TB/s}$ a thousand times faster. Speeding the loopback up further reduces the execution time, and vice-versa for a reduction of bandwidth. Even with $25.6\,\mathrm{Gbit/s}$ bandwidth, the loopback is still $2.5\,\mathrm{times}$ faster than Ethernet links which is why the estimated timing is still close to the actual result. Both figures have in common that when the respective scenarios are executed on more than one node, local communications play almost no role because the HPL implementation we used does not particularly exploit locality. Each node still has the same amount of processes but their messages become slower and more distinct (albeit slow) Ethernet links are available for the communication. In all of these scenarios, SMPI achieves almost perfect results regardless of whether it is configured specifically for local communication.

## 7.4 Limitations

Figure 7.1 visualizes not only the calibration results for the intra-node calibration but gives for comparison the linear model found for the Ethernet interconnect, see Figure 4.7 (page 54). As one can see, the computed piecewise regressions are different and hence not exchangeable. It may therefore be necessary to allow users to configure the communication overhead based on the sender/receiver of a message. Currently, SMPI supports startup- and per-byte overhead for messages on a global and per-node basis but accounting for different costs based on the used communication method (network, shared memory) or the route is at this

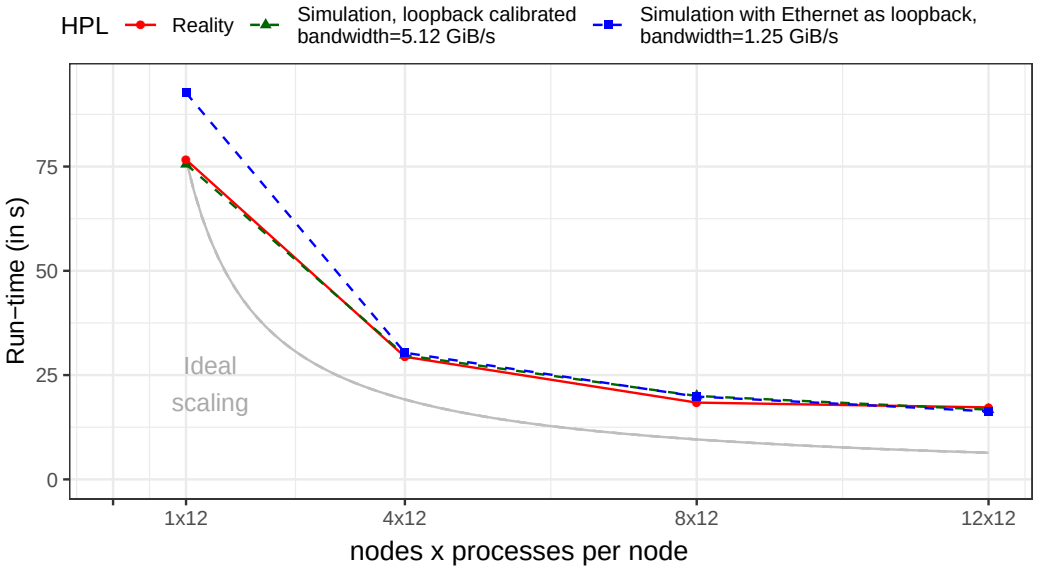


**Figure 7.2:** HPL does not exploit locality and therefore, only the single-node execution is largely overestimated when the loopback link remains configured with the same speed as the more than four times slower network.

time impossible. Furthermore, the configuration options that are used to adapt the available bandwidth and latency based on the message size (e.g., to account for protocol overhead, see Section 4.3.2) can currently only be set *globally* and hence both inter- or intra-node communications use the same values. To obtain faithful predictions for applications that, unlike HPL, exploit locality, the whole range of message sizes or are more communication-sensitive (i.e., communication cannot be largely overlapped with computation), it may be necessary to use a different set of values based on the sender and receiver of a message. Note that supporting different values per sender/receiver is more general than to declare the same options again, only this time for local communications, as it takes into account that platforms may employ heterogeneous hardware (e.g., different motherboards, multiple network types) that require different configurations. Recall from Section 4.2.1 ("Platform Description") that the entire platform must be defined within one or more so-called zones. We believe that the abovementioned modifications are relatively straightforward when attaching these options directly to each zone declared in the platform [Bob+12]. This is planned for the near future. For the applications we used for our experiments, this was not necessary as faithful predictions only required correct bandwidth and latency values for the loopbacks.

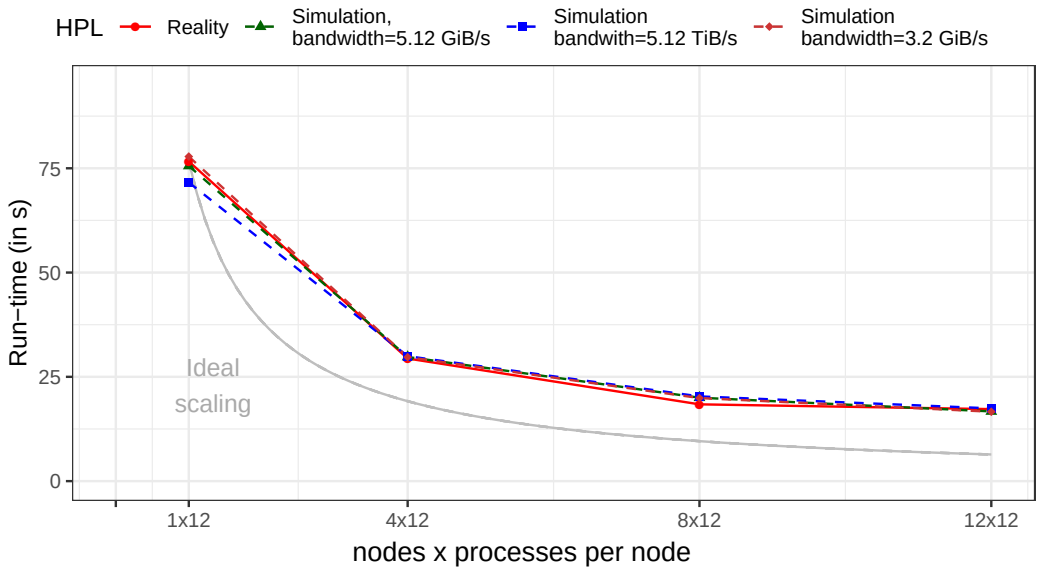


**Figure 7.3:** When executed on only one node, HPL's performance is influenced by loopback bandwidth: A thousandfold increase to $5.12\,\mathrm{TB/s}$ reduces the runtime by almost $5\,\mathrm{s}$ (from $76.64\,\mathrm{s}$ to $71.61\,\mathrm{s}$) whereas lowering the bandwidth to around two-third ($3.2\,\mathrm{GB/s}$) causes the runtime to be overestimated (with a total of $77.8\,\mathrm{s}$). When executed on more than one node, local communication becomes less important and hence the impact of these settings is not noticeable.

# 8 Contribution: Modeling Multi-Core CPU Power Consumption

We have already seen in the introduction (Chapter 1) that the energy consumption has become one of the main problems in HPC. A supercomputer consumes power mainly through its components, such as the network, compute- and storage nodes. This base consumption is further increased by indirect consumption, such as cooling equipment (which can account for up to 33 % of the total consumption [ODL14, p. 8]), power losses caused by physical effects that occur during distribution as well as conversion and various other components such as uninterruptable power supply [Sho+17, p. 955].

In this chapter, a model for the prediction of energy consumption of multi-core CPUs is presented. In standard multi-core nodes (i.e., without accelerator), CPUs are often the most important source of energy consumption [ODL14, p. 3].

This chapter contains figures and text that were published during this dissertation project [Hei+17b; Hei+17a].

## 8.1 Problem

Models of the energy-consumption of CPUs in compute nodes often distinguish two types of consumption: *Static* and *dynamic* consumption [ODL14, p. 2].

*Static* consumption denotes the energy that is consumed when the CPU is on but idle *and* no measures for power-saving have been taken, for example, because not enough time has passed to justify entering a power-saving mode. It is generally set to

$$P_{static} = V \cdot I_{leak}$$

whereas $V$ is the supply voltage and $I_{leak}$ the inevitable power leakage [Ren12, p. 9].

The *dynamic* part comes into play when the CPU is actually executing instructions. This dynamic part has been found to be

$$P_{dynamic} = A \cdot C \cdot V^2 \cdot f \tag{8.1}$$

with $A$ denoting the percentage of active gates, $C$ the total capacitance load, $V$ the supply voltage and $f$ represents the frequency [ODL14, p. 4].

Note that $V$ and $f$ are not fully independent of each other. In fact, through elimination of $V$, the above formula can be further simplified to [DWF16, Eq. 20]

$$P_{dynamic} \approx A \cdot C \cdot f^3. \tag{8.2}$$

Alas, the simple structure of these formulas is deceptive. The first variable, $A$, can be interpreted as the switching activity and is highly dependent on the current activity of the code (e.g., floating point operations or memory accesses require different gates), the general architecture of the CPU itself, the compiler, operating system and finally also on the activity of the other cores. Determining the correct value can be expected to pose a problem for application developers, as no easy way of measuring it is known and deep insight into a platform's CPU architecture (e.g., AMD64, ARM, ...) is required. The second variable, $C$, is the total capacitance load and hence the power consumed when switching a gate. It is a constant for a particular CPU, but its actual value is determined by the used manufacturing process.

For a long time, $V$ and $f$ were mostly fixed. This has changed fundamentally with the introduction of dynamic voltage and frequency scaling (DVFS), a nowadays common feature of almost all CPUs. DVFS allows the OS kernel to adapt the voltage and frequency based on the workload and is controlled by a so-called *governor*, essentially an algorithm that makes decisions in order to save power. (For more details on DVFS, see the discussion in Chapter 9.) In practice, not all values for $f$ and $V$ can be attained. In fact, CPUs generally only support a limited set of frequencies called *P-states*. They are denoted $P_0, ..., P_m, ..., P_n$ and $m, n$ depend on the processor model. In this case, $m$ stands for the $m + 1$ user-selectable and sustainably usable frequencies for each core. For example, a CPU could provide P-states $P_0, ..., P_m$ that would correspond to $1200\,\mathrm{MHz}$ to $2300\,\mathrm{MHz}$, with increases by $100\,\mathrm{MHz}$. The remaining $n - m$ frequencies are reserved for specific technologies such as Intel's Turbo Boost. These techniques, more commonly known as *dynamic overclocking*, use dynamic increases in voltage to achieve higher frequencies. These per-core frequencies can be sustained as long as power, voltage and thermal constraints of the CPU are satisfied. The frequency other cores execute at hence influence also

the possible frequencies for dynamic overlocking. This means that the frequencies $P_{m+1}, ..., P_n$ can only be selected for a relative short amount of time.

Modern CPUs also feature per-core power-saving states, called *C-states*, that are used to reduce $P_{static}$, i.e., the cost of the CPU being on but not executing any computations. The available C-states of a CPU are denoted $C_0, ..., C_m$, but not all of them have to be actually available on a particular machine. Furthermore, only $C_0$ does *not* denote a power-saving state: It is in fact the state that the CPU enters when it is computing. The other states are ordered ascendingly by their energy savings, i.e., $C_i$ is supposed to save more energy than $C_{i-1}$ (if it exists). Since each core can be used independently of the others for computations, it makes sense to allow each core to enter C-states independently of the others as well. These per-core C-states are called *CC-states* (Core C-states). The operating system kernel can request a C-state for each core individually. This requested C-state is called *LC-state* (logical C-state) and may not always correspond to the actual CC-state of a core because there are architecture-dependent constraints for each CC-state: They enforce that no core can thwart another core from doing its task for the sake of energy savings. For instance, Intel CPUs disable the clock at C-state C3, but this is impossible when at least one core on the die is currently computing (and hence in C-State C0). This means that even though the LC-state may be set to 3, the CC-state will not be set to 3 until all cores can accept the clock to be disabled. Finally, each processor has one unique PC-state (Processor C-states). The PC-state is equal to the minimum CC-state of all cores of the CPU [ODL14; Kid08].

Whether or not a core can enter a given C-state can change rapidly due to these complex dependencies on the other cores. Considering C-states in a power consumption model correctly hence requires much more work, especially since wakeup-latencies must also be accounted for. Furthermore, since parts of the circuit are actually disabled, any model that accounts for the number of gates that are currently active would have to consider the disabled parts of the circuit, and this would need to be updated every time a core changes its CC-state.

Even though it may be possible to build a precise model at the architectural level, it would require endusers to find out how many gates are switched in a specific case for every single CPU model they use. Additionally, accounting for all of the above would render the model itself very complex since many parameters would be required. This would make the model not only difficult to instanciate and hard to control but also validation would become extremely difficult.

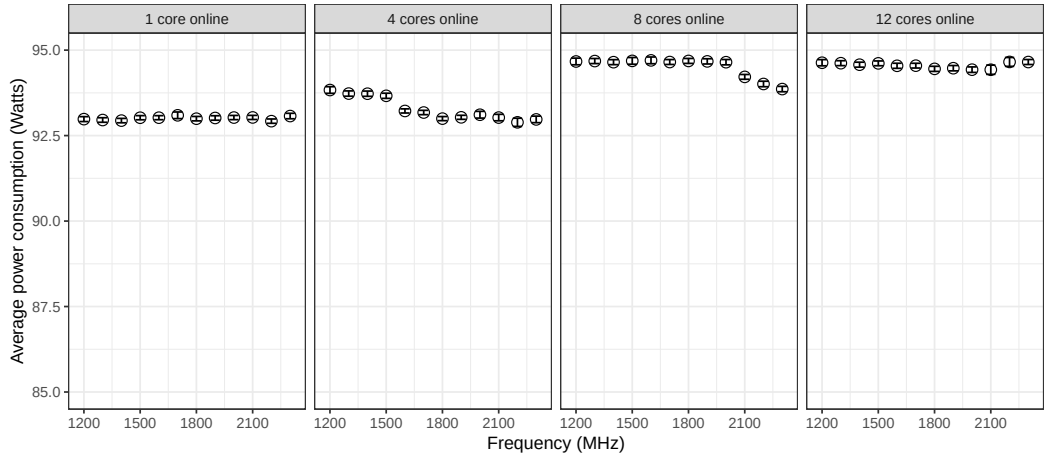


**Figure 8.1:** We measured idle power on every machine (here: `taurus-1`) for ten minutes per frequency and active cores. These measurements were repeated three times and no-earlier than 3 weeks after the previous measurement. Note that the y-axis begins at $85\,\mathrm{W}$. The variation per core-count is therefore minimal and on the order of about $1\,\mathrm{W}$.

## 8.2 Proposed Solution

A non-linear connection between frequency and power consumption as in Equation 8.2 can be easily verified through experimentation. Figure 8.2 visualizes the results for an experimental campaign (see Chapter 5) with the CPU-bound `NAS-EP` benchmark and a fixed workload (class C). The application was executed for each available frequency (fixed to a value in the $1200\,\mathrm{MHz}$ to $2300\,\mathrm{MHz}$ range) with either $1, 4, 8$ or $12$ cores activated. The plot for a single core seems to remain almost linear, but when all $12$ cores are activated, a clear non-linear relationship becomes apparent. However, a linear relationship seems to be possible when comparing the distances (on the y-axis) between measured values for every single, *fixed* frequency as they appear to be equidistant.

This is confirmed by Figure 8.3, which illustrates the same data (some frequencies were skipped to reduce overplotting) but keeps the frequency constant and only varies the number of cores. In other words, for each fixed frequency, the energy consumption of a node is linear in the number of cores that are in use. When extrapolating a power value for the entire node when $0$ cores (i.e., CPU is idle) are used, a significant difference between this obtained value (called $P_{static}$, borrowing from the CPU-only value above) and $P_{idle}$ becomes apparent. This can be explained with the power-saving measures (e.g., C-states) that are taken when the node is idle for too long. The extrapolated value hence represents a consumption during extremely short times: Either the CPU stays idle and then goes into a sleep state, i.e., the consumption drops to $P_{idle}$ or at least one core starts computing. Note that

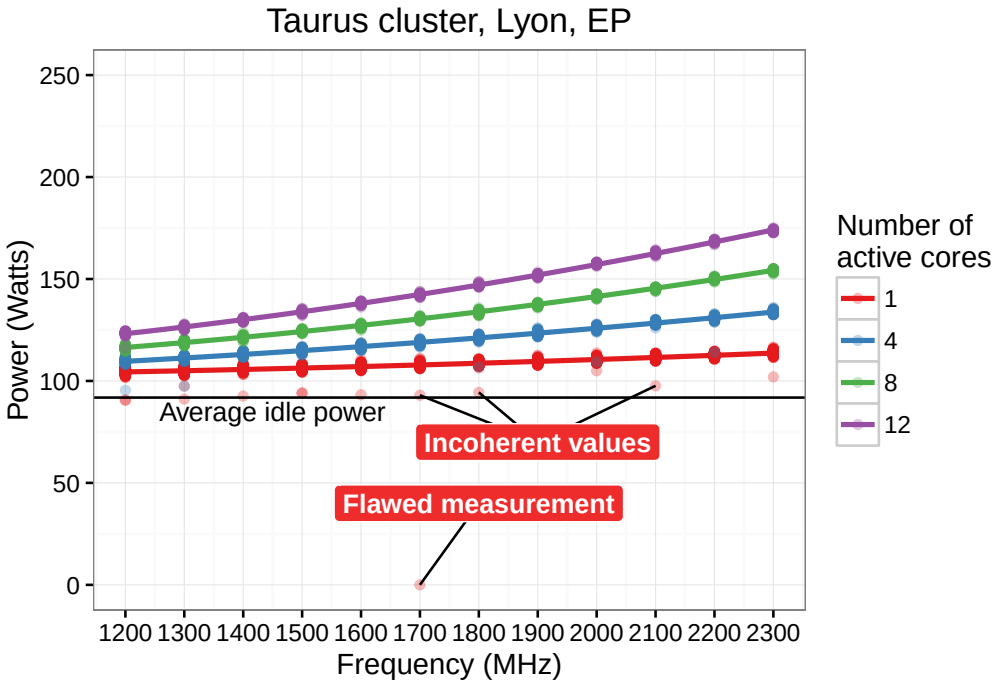


**Figure 8.2:** Changing the frequency while keeping the load constant causes the consumed energy to grow quadratically. The energy that is required to keep an idling node on (despite all energy-efficiency measures, such as C-states), is additionally shown here as $P_{idle}$.

even though $P_{static}$ was extrapolated with data based on a specific workload, it is reasonable to expect that it is independent of the workload.

Following directly from these observations, the power consumption of a CPU with usage $u$ (in percent, via $\frac{\min(\#cores, \#computations)}{\#cores} \cdot 100\,\%$) can be predicted using the following model for a machine $i$ executing a workload $w$ (e.g., memory-intensive or not, ...) at fixed frequency $f$:

$$P_{i,f,w}(u) = P_{i,f}^{\text{static}} + P_{i,f,w}^{\text{dynamic}} \cdot u \tag{8.3}$$

As just discussed, $P_{static}$ does not depend on the workload but on the machine and even the frequency. Allowing $P_{static}$ to vary is important for complexer node-types, such as ARM's big.LITTLE, where higher frequencies cause the stronger computation cores to take over, causing $P_{static}$ to vary.

By ignoring the details of the energy-saving techniques discussed above, only very few variables are required, which are furthermore measureable by application developers. This model is therefore very easy to instanciate for any user.

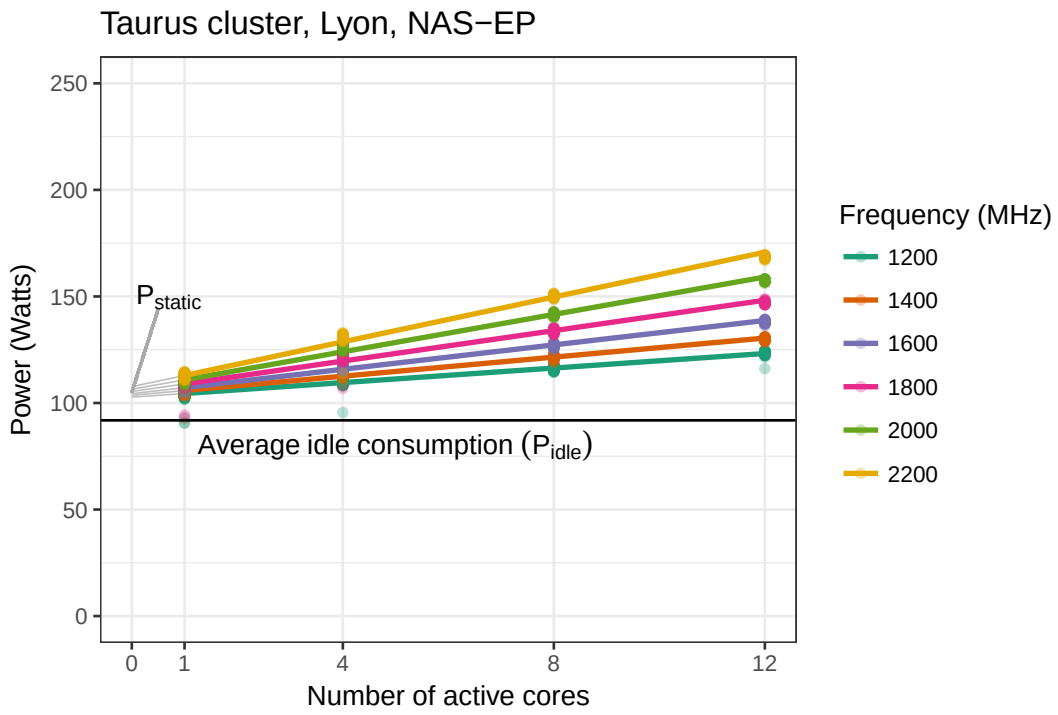


**Figure 8.3:** Varying the number of cores while keeping the workload constant (NAS-EP, class C) reveals a linear connection between load and power consumption. Note that not all frequencies are shown in this figure to reduce overplotting. The energy that is required to keep an idling node on (despite all energy-efficiency measures, such as C-states), is shown here as $P_{idle}$.

### 8.2.1 Calibrating the Energy Consumption

There is, however, overhead because this model depends on the workload, host and frequency as non-variable parameters. Unfortunately, characterizing the energy consumption for every workload on every node is required for faithful predictions, as is illustrated by Figure 8.4: At a macroscopic scale (one averaged sample per second), the CPU-bound NAS-EP benchmark always consumes about $40\,\mathrm{W\,s}$ less than the memory-bound NAS-LU or HPL benchmarks, which are separated by $10\,\mathrm{W\,s}$ to $15\,\mathrm{W\,s}$. Most machines seem to consume more or less the same when they execute the same workload but this is not always true, as is the case for `taurus-8` and `taurus-10`. Furthermore, the used software and hardware stack (e.g., compiler and CPU, respectively) impact the power consumption as well (e.g., through optimizations). By measuring the consumption for a workload on a specific machine, these impacts are accounted for and do not have to be integrated explicitly into the model. This would (just as considering other phenomena, such as C-states) require more variables that are difficult to measure and instanciate, in exchange for a questionable and probably very limited gain in accuracy.

Energy consumption cannot only vary among nodes, but it can vary *on the same node* when enough time passes. Figure 8.5 depicts the results of two experimental

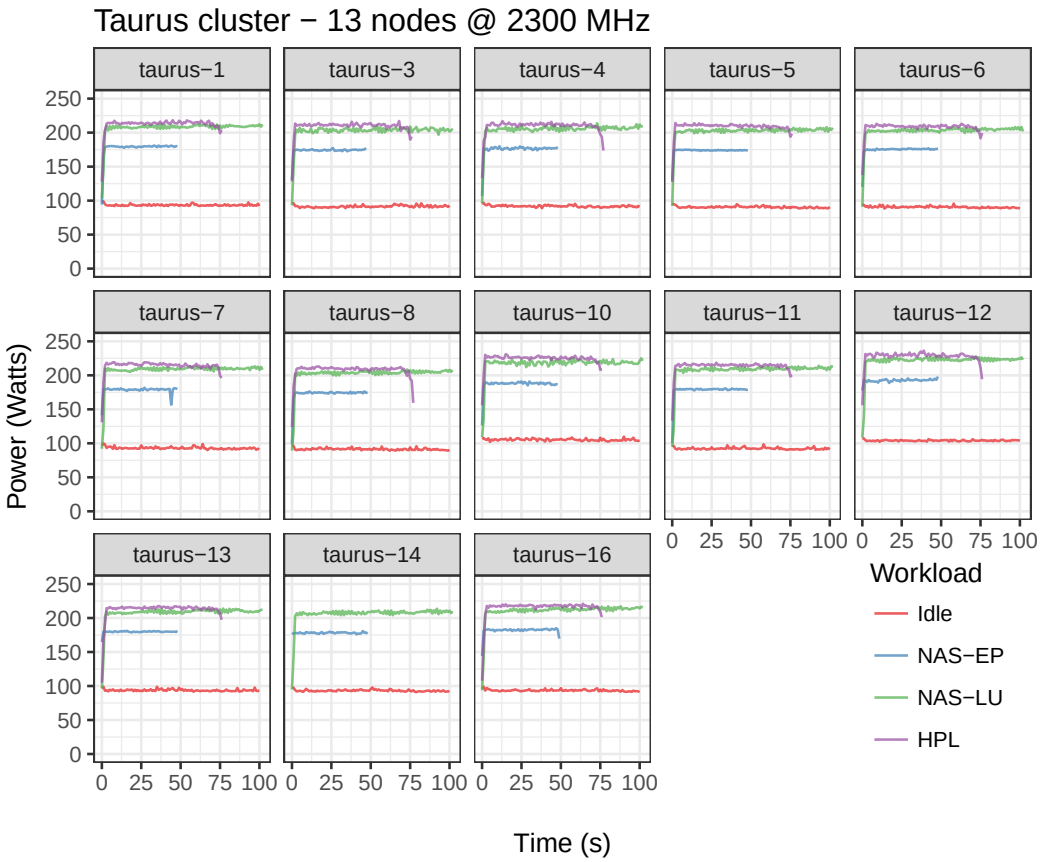


**Figure 8.4:** Power consumption over time when running NAS-EP, NAS-LU, HPL or idling (with 12 active cores and the frequency set to 2300 MHz).

campaigns that were run in May 2014 (by Alexandra Carpen and Sascha Hunoldt) and one in October 2016 (by myself) to determine $P_{idle}$. After 29 months, the idle power consumption of `taurus-12` had increased by 11 W s, while it dropped by 3 W/s for `taurus-5`. In fact, it is not uncommon to find out that the assumption, that a cluster consists of homogeneous nodes, does not hold true after the entire cluster was calibrated [Ina+15]. Thankfully, the measurements we obtained over a duration of two hours are mostly stable. This allowed us to set $P_{idle}$ to the sample mean, which is a valid approximation in this case, after we detected and removed several outliers (with values around 0 W s and 50 W/s) as they could clearly be attributed to wattmeter glitches. This demonstrates that re-calibrating the machines may be necessary from time to time due to (often opaque) internal changes that can cause even idle nodes to consume more. Alas, doing so would consume significant resources and is therefore not always feasible. In many cases, machines are already attached to a monitoring infrastructure that can detect and inform the user about nodes that require individual calibration due to their different behavior.

There are several ways to compute the linear model, and it is not always necessary to measure the intercept ($P_{static}$) itself. In our case, we chose to make two runs of the target workload: One run is limited to a single core and the other run uses

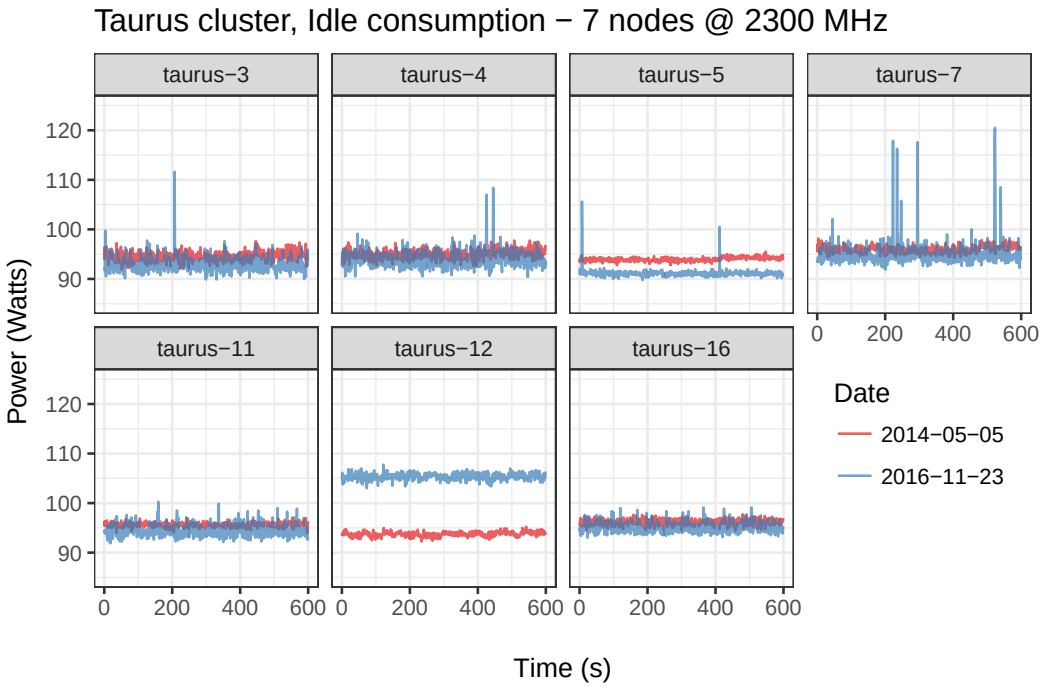


**Figure 8.5:** Comparison of two idle power measurements. The first one was executed in 2014 by Alexandra Carpen-Amarie and Sascha Hunoldt, the one in 2016 by me. The frequency was fixed at 2300 MHz for the used nodes. In this plot, the y-axis starts at around 90 W s to highlight the differences, especially for taurus-12.

all cores. This seemed to be easier to setup in experiments than disabling all C-states to measure $P_{static}$, which is not always possible since this requires superuser privileges. The intermediate values and $P_{static}$ can then be interpolated through the resulting linear function. Interpolating the intercept $P_{static}$ is not a problem because this state is only entered for a fraction of a second before either a new computation is executed or the cores enter an energy-saving state.

### 8.2.2 Predicting the Energy Consumption with SimGrid

The implementation of the model within SimGrid is relatively straightforward. SimGrid is always aware of the exact load of a node and can easily compute the load $u$ (e.g., a $12-$core node with $6$ concurrent computations has a load of $50\,\%$). During a simulation, SimGrid must update the energy consumption every time that the load $u$ changes. In SimGrid, load can only increase or decrease when a computation starts or finishes. This ignores that HPC codes commonly check the status of a message by looping over `MPI_Iprobe`. As long as the message is not available, small computations are executed to overlay the time spend waiting with useful work. While these computations are accounted for in SimGrid, the repeated polling (and the resulting, non-neglegible CPU usage) is not, even though SimGrid already allowed users to configure the time a single call to `MPI_Iprobe` takes. This time, however, was modeled as a purely virtual delay (i.e., the core spends the time

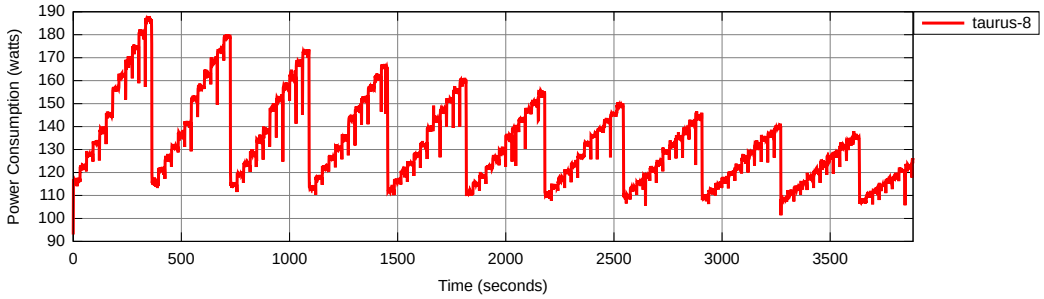


**Figure 8.6:** Power consumption for a single node when polling via MPI_Iprobe. The distinct growing phases represent one frequency, ranging from 2300 MHz to 1300 MHz, and 1 to 12 cores. When the second CPU gets activated, a jump is recognizable, especially for the highest frequency.

sleeping) and hence only the idle-power was consumed because no computation was started. Clearly, the sleep needs to be replaced with a computation, but it is insufficient to just run a computation *at full speed* for the time of the iprobe because the amount of energy consumed by iprobes is different from the main application: Figure 8.6 shows the power consumption of a single node that occupies step by step one more of its cores with repeated polling for new messages. When all cores have been used, the frequency is set to the next lower available frequency and the procedure restarts with one core. When all cores poll repeatedly, the power consumption was around 188 W s, which is significantly higher than the previously accounted idle power consumption of merely 92 W s. HPL, the application that uses `MPI_Iprobe`, consumes around 214 W when all cores are used and run at maximum frequency. For this reason, an option was introduced that allows users to scale the usage of the CPU while polling so that the measured energy consumption of `MPI_Iprobe` is consumed (a similar effect is expected for `MPI_Test` but was not particulary investigated). Note that, when increasing the core count from 6 to 7 cores, slightly more energy is consumed than for any other step. This is due to the architecture of the target platform: The taurus nodes contain two CPUs with 6 cores each, i.e., this jump stems from activating the second CPU. Recall from Section 4.2.1 that SimGrid lacks any notion of CPU. SimGrid can therefore not track which (and how many) CPUs are activated and this jump (of about 5 W s) was ignored.

In our experiments, NAS-EP and NAS-LU were unaffected by this because they don't use `MPI_Iprobe`. HPL, on the other hand, heavily uses this mechanism in its own `MPI_Bcast` implementation, which initially resulted in serious energy mispredictions.

Recall from above that the intercept $P_{static}$ is only used for a fraction of a second. In our implementation, we have replaced the $P_{static}$ with the more conservative value $P_{idle}$. This allows SimGrid to ignore that deeper energy-saving states are only entered one after another. To keep the implementation simple, we furthermore

```
<host id="taurus-8.lyon.grid5000.fr" speed="23E9, 22E9, 21E9, ..." core="12">
  <prop id="watt_per_state" value="92.75:114.62:174.38, 92.75:113.25:168.62, 92.88
    :112.25:162.88, 92.88:110.75:157.12, 92.88:110.38:151.75, 92.88:109.38:147.25,
     92.88:108.62:142.75, 93:107.38:138.25, 93.12:106.75:134, 93:106.5:130.62, 93
    :105.12:127, 93.25:104.62:123.62" />
  <prop id="watt_off" value="10" />
</host>
```

**Figure 8.7:** A sample configuration for the `taurus-8` node. The configured workload here is the EP benchmark and corresponds to the linear regression of Figure 8.3. For each frequency, three values are provided: the idle power $P_{idle}$, the power consumption when the workload is executed on a single core and the power consumption when the workload is executed on all cores. SimGrid allows the user to configure the energy that the node consumes when it is turned off through the "watt_off" option.

ignore latencies for entering and leaving C-states: Latencies depend on the C-state, are architecture dependend and would have required configuration by the user. [1]

The proposed model is cheap to evaluate, which is mandatory when computations are very small and the update mechanism is frequently triggered. In the worst case, this can be necessary hundreds or thousands of times per (simulated) second and MPI process.

Finally, Figure 8.7 presents the configuration of node `taurus-8` for the EP workload. Only the values for the `watt_per_state` tag are subject to change and values are given in the `Idle:1Core:AllCores` format, followed by a comma (“,”) and the values for the next pstate. This example contains (marginally) differing energy values for the idle state because we calibrated even idle consumption for every single frequency, but this is clearly not necessary. Note also that the pstates (listed in the `speed` attribute) were shortened for increased readability.

The configuration shown in Figure 8.7 represents a very simple and uniform workload. The energy consumption during different states, such as booting or when the machine is turned off, may need to be considered as well. The energy consumption (e.g., for wake-up-on-LAN devices) when the machine is turned off can be specified directly in the `watt_off` property. However, states such as “booting” are not supported directly and must be modeled by adding another value to the speed-attribute. Since floprates are not required to be unique, one can use a floprate that has already been declared previously (e.g., the highest one). This state is consequently not interpreted as a CPU frequency that can be selected at any time when running a workload but rather as a special state the system can enter. The corresponding energy consumption can then be declared accordingly. When the

[1] Latencies can be obtained by running the command `cpupower idle-info` on the target platform.

machine is booting, this specific state can be entered and the energy is correctly accounted for.

Using this method, a user can also model additional power consumption when offloading onto an accelerator (see Section 2.1.2 and Section 2.2.2). Of course, this does not allow SimGrid to suddenly execute, e.g., CUDA code and so entering this specific state must be done manually when using SimGrid programmatically. The additional power consumed by the accelerator can thus be accounted for and once the accelerator is done, the user can simply switch back to a "normal" state that just models computations on the CPU and accounts for idle energy usage of the accelerator.

## 8.3 Performance Evaluation/Effectiveness

To validate our model, we calibrated the entire `taurus` cluster (see Section 5.1 for a brief discussion of this setup), i.e., we calibrated each node, the network and calls to `MPI_Iprobe` and ran three popular benchmarks (NAS-EP, NAS-LU, HPL) on partitions with $1$, $4$, $8$ and $12$ nodes. Only NAS-LU required a calibration file for speed-ups / slow-downs that was obtained as detailed in Chapter 6. Subsequently, we used SimGrid to predict the time-to-solution and energy-to-solution. Figure 8.8 compares the simulation results to the results obtained from the real-life experiments. Thanks to the calibration, predictions are in almost all cases indiscernable from measured real-life results. Only NAS-EP ("Embarrassingly Parallel") achieved perfect scaling and the energy consumption hence remains almost constant. NAS-LU and HPL scale sublinearly, which is expected and causes the energy consumption to grow as well. As can be seen in Figure 8.9, the energy prediction accuracy of HPL is largely due to accounting for CPU load caused by `MPI_Iprobe` calls. When polling is modeled with a simple sleep (green dotted line), the energy is underestimated significantly and this error increases with the number of processes. However, modeling the iprobe call with a computation yields almost perfect results (blue dashed line). Therefore, when iprobes are used massively, calibrating SimGrid accordingly is required.

In all cases, SimGrid's prediction error is within a few percent. Since every node was calibrated individually, heterogeneity is accounted for. However, we found that in our experiments, calibrating only a single node and using this configuration for all other nodes incurs only $\approx 1\,\%$ of error, effectively rendering it indistinguishable from noise commonly experienced in real experiments (see Figure 8.10).

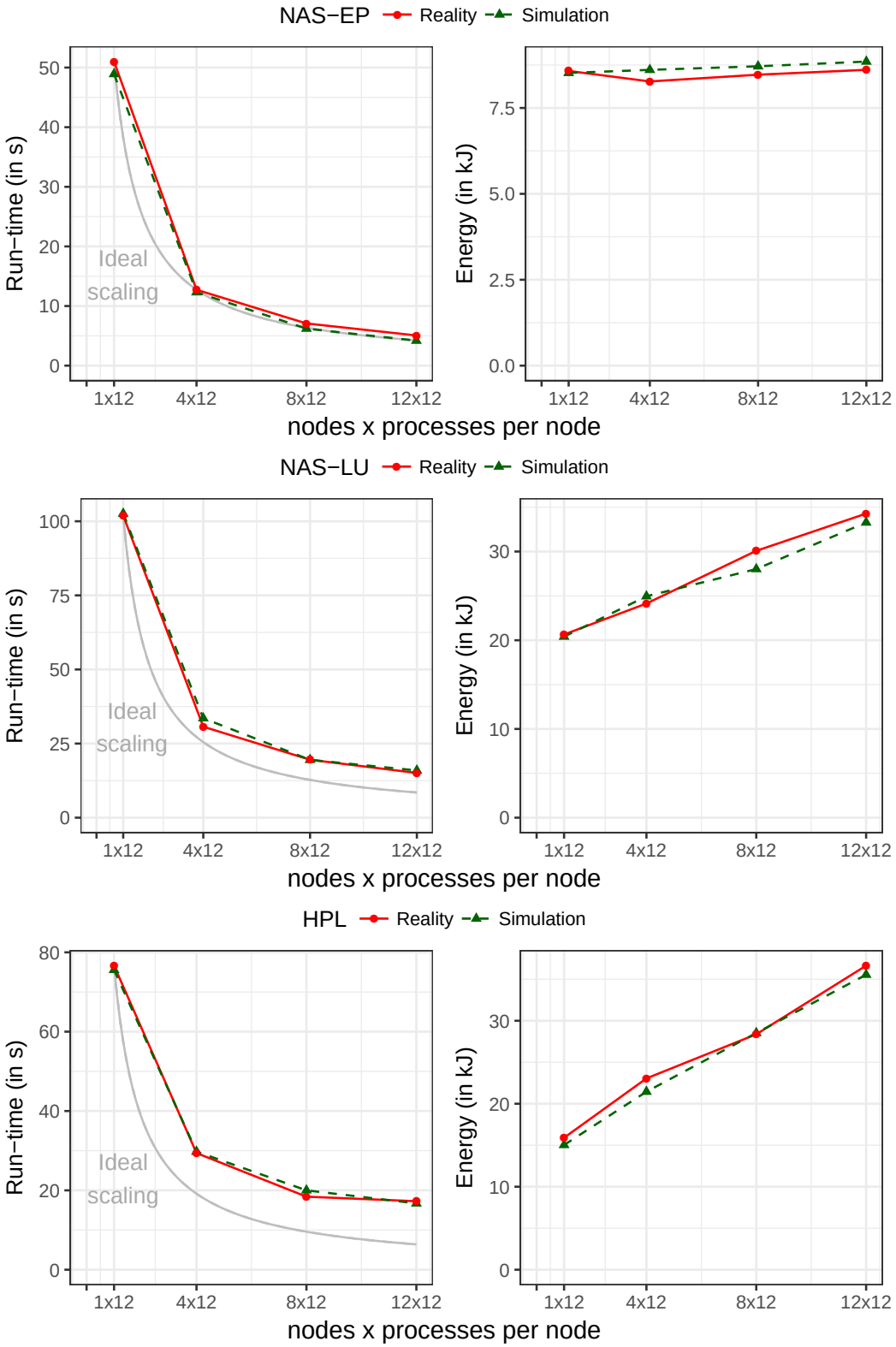


**Figure 8.8:** The validity of this model was tested with three popular benchmarks: NAS-EP, NAS-LU, and HPL. The taurus cluster was used with up to 12 nodes and 12 processes per node.

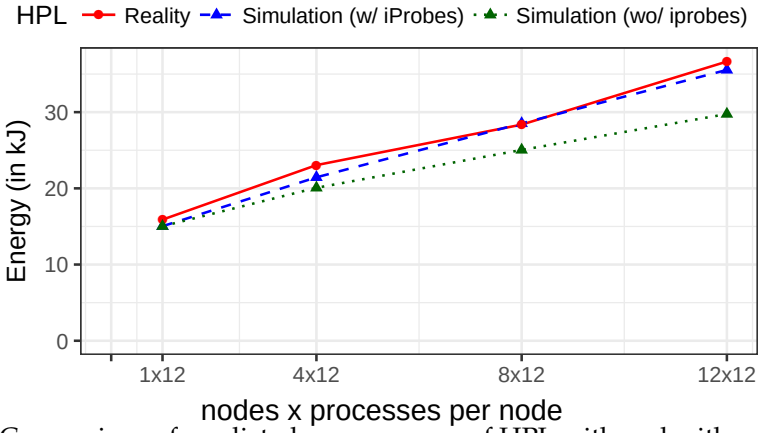


**Figure 8.9:** Comparison of predicted energy usage of HPL with and without accounting for the additional energy consumption of MPI_Iprobe calls.

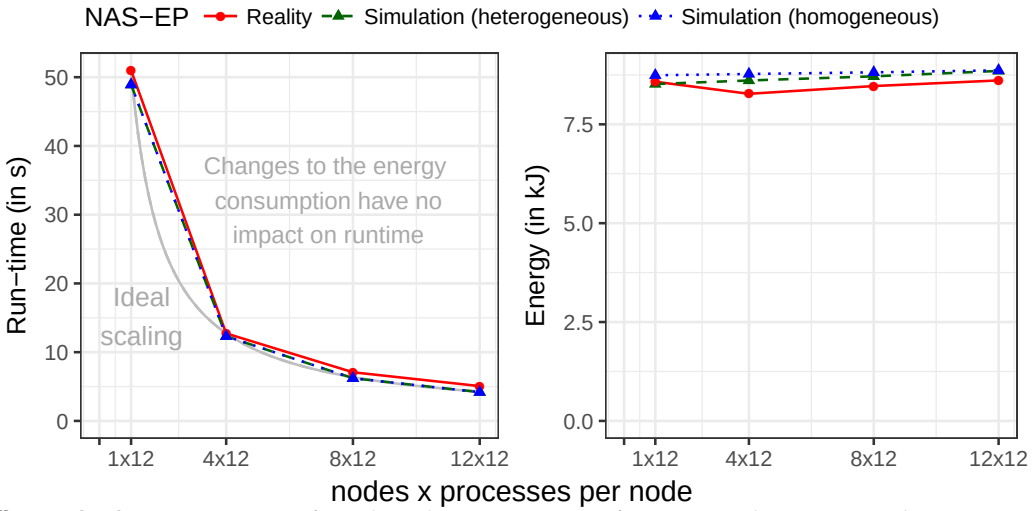


**Figure 8.10:** Comparison of predicted energy usage of NAS-EP when using only one or an individual power model for all simulated nodes.

## 8.4 Use Case: Capacity Planning for HPL

One interesting field of application is capacity planning. In collaboration with Tom Cornebize, we studied how HPL benefits from more resources and the respective energy consumption. As is generally the case for capacity planning, the machine does not (yet) exist. The platform we imagined contains $256$ nodes with $12$ cores per node. This is deliberately similar to the taurus-cluster that was used for the validation study. This time, however, we decided to connect all nodes with a hypothetical fat-tree network, built with 16-port switches on two levels: the top layer consists of $2$ switches whereas the bottom layer comprises $16$ The links were set up as $10\,\mathrm{Gbit/s}$ Ethernet links.

The previous validation study leveraged SMPI's emulation mechanism and hence executed every instruction of the unmodified applications. At most $144$ processes were used and the workload size was limited: For HPL, a matrix of size $20,000 \times$

$20,000$, as previously used, consumes $3.2\,\mathrm{GB}$ of memory. For this (very small) use-case, the main problem was hence not the memory consumption but the emulation time of almost two hours. This is a drastic, but not unexpected increase from the $20\,\mathrm{s}$ it took to execute HPL with $144$ MPI-processes on $12$ nodes of the taurus cluster. This problem exacerbates with more processes, a more complex network and larger problem instances. Furthermore, memory *does* become an issue once the input matrix reaches a certain size: For a still relatively small $65,536 \times 65,536$ matrix, $34.3\,\mathrm{GB}$ of memory are required which surpasses the entire memory of a single taurus node. To alleviate this prohibitive resource hunger, we resorted to the two techniques presented in Section 4.3.3 that allow SMPI to emulate runs at larger scale by exploiting HPL's regularity: First, kernel modeling to reduce the execution time by skipping their execution and second, shared memory usage to reduce the required memory. We have detailed the necessary modifications that finally allowed us to simulate HPL on a single node using a model and the scale of the Stampede supercomputer in a technical report [Cor+17] that is currently under preparation for publication. For the above matrix ($20,000 \times 20,000$), these modifications allowed us to run the (sequentially executed) emulation of $144$ processes on a commodity laptop in under two minutes with as little as $43\,\mathrm{MB}$ of RAM used. Using all nodes and all cores of our (hypothetical) platform, the emulation of $256 \times 12 = 3,072$ MPI processes took around $90$ minutes and required not even $1.5\,\mathrm{GB}$ of RAM. Our study was limited to the two aforementioned input sizes. For each size, in total $5$ scenarios were executed and results are depicted in Figure 8.11. The simulation results on up to $12$ nodes yield the same results as were obtained in real-life (red line in the top figure). This is expected, because all nodes are connected to the same switch in this case and the new network-topology is hence ignored. When using more nodes than can be connected to the same switch, however, a slowdown can be observed due to the added latency. This also increased the rate of power consumption, which is already elevated due to the added nodes. The larger matrix, on the other hand, was more suitable for scaling. The energy consumption continued to grow but this is partly also due to a different ratio of communication and computation.

## 8.5 Limitations

### 8.5.1 Model Limitations

Recall from Section 8.2 that our model is calibrated with and hence depends on the workload $w$. Since the model calculates the power consumption for an entire node, it even implicitly assumes that all cores either execute $w$, a workload with similar characteristics (e.g., memory accesses, cache usage, I/O, ...), communicate or are fully idle and that the workload only changes within these three cases throughout

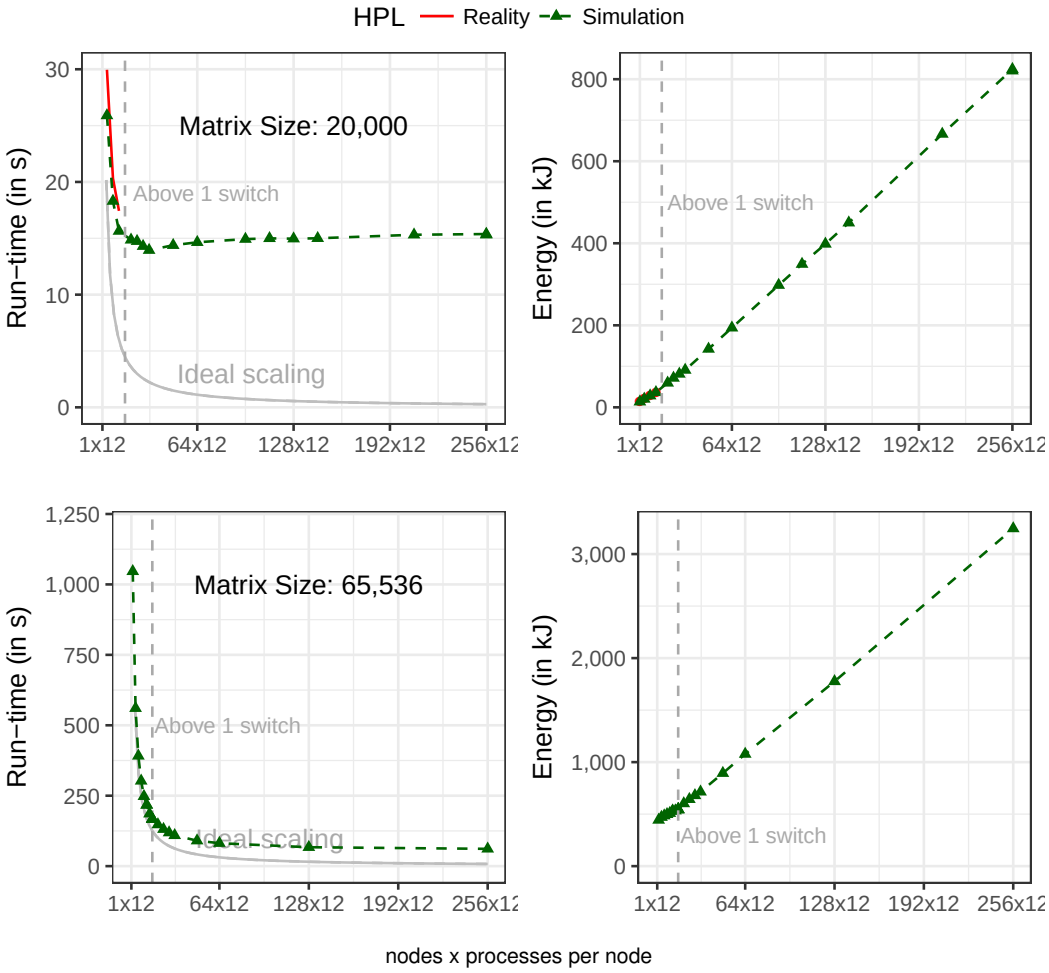


**Figure 8.11:** Time- and energy-to-solution extrapolated for two different matrix sizes with up to $256 \times 12 = 3,072$ MPI processes, interconnected by a fat-tree topology. Once a threshold is reached, adding more nodes does not yield faster performance but only increased energy consumption.

the entire execution. For HPC applications, this is often a reasonable assumption due to their regularity. However, this restriction can be violated in three scenarios that are more difficult or currently impossible to model because the memory and cache usage is difficult to predict.

First, an application can consist of phases, i.e., the characteristics of the currently executed code changes (for instance from a memory or cache heavy computation to an I/O heavy checkpointing procedure (see Section 2.2.1) or when a workload is offloaded to a previously idle GPU). The energy profile of the computational workload of the application therefore does not remain constant throughout the execution. Each of these phases should then be characterized individually. This is already possible and can be done analogously to the previously described "special states" such as booting. However, further problems may be encountered by the user when the phases are of microscopic length (e.g., when kernels constitute each an individual phase) because power measurement and tracing tools that support

this precision are rarely available. As explained in Section 5.1, the wattmeter we used for our experiments returned a single value per second. The measured power is in such a case not clearly associable to a single phase, however, one can track how much time $t_i^p$ was spent during the $i-th$ measurement interval working on phase $p$. Likewise, the amount of energy $e_i$ during the $i-th$ measurement can easily be measured. With a large enough set of samples $(e_i, t_i^1, ..., t_i^N)$, the application of statistical estimators should make it possible to infer the consumption of each *phase*.

In the second problematic scenario, the CPU is shared between different applications, each with different but constant energy consumption. *In-situ* applications are an example. These are applications that can be separated into mainly two parts: The first part (simulation component) generates data that are subsequently statistically analyzed by the second part (statistics component). This case is very different than the previous as several kernels are executed at the same time and hence potentially impact each other. Providing a single energy profile per application might therefore not be sufficient. It may therefore be necessary to obtain an energy and performance profile of the concurrently executed components that depends on the number of cores alotted to each component.

Finally, in the third scenario, execution of kernels and applications is no longer structured, i.e., no assumptions on which workload is executed at a specific time can be made. Highly dynamic applications and runtimes (such as Star-PU, see Section 4.4) often fall into this category because the number of cores and kernels cause a combinatorial explosion of kernels that could potentially execute in parallel. Alas, the cache and memory usage directly influences the energy consumption and the only real option for faithful predictions with our model is to obtain measurements for *all* possible combinations.

### 8.5.2 Experimental Limitations

Recall from Section 5.1 that our wattmeter provides a single, averaged sample per second. It lies in the very nature of an average of a non-constant series that some values must be larger and others smaller than the average. To instantiate our model, the user is required to supply the consumption when all cores are active. This value was measured by running the application on all cores at the same time, but we only later did we find (through simulation) that the node's load changes many times per second when executing NAS-LU. To exemplify this, Table 8.1 lists for each core count the absolute time that only this many cores are active. As one can see, the application computes concurrently on all cores only about $62.76\,\%$ of the entire execution time. This means that, since our real-life watt measurement is an

| Cores | Load | Total time | Percent of total execution time |
|---|---|---|---|
| 0 | 0.000 | 0.105638 | 0.12987747 |
| 1 | 0.083 | 0.316098 | 0.38862918 |
| 2 | 0.166 | 0.016594 | 0.020401624 |
| 3 | 0.250 | 0.247750 | 0.30459819 |
| 4 | 0.333 | 0.284455 | 0.34972544 |
| 5 | 0.416 | 0.473350 | 0.58196389 |
| 6 | 0.500 | 1.607626 | 1.9765085 |
| 7 | 0.583 | 0.977106 | 1.2013107 |
| 8 | 0.666 | 2.194727 | 2.6983244 |
| 9 | 0.750 | 5.018271 | 6.1697528 |
| 10 | 0.833 | 5.915051 | 7.2723061 |
| 11 | 0.916 | 13.126249 | 16.138170 |
| 12 | 1.000 | 51.053753 | 62.768439 |

**Table 8.1:** An entire execution of NAS-LU (class C) on a single node with 12 cores, broken down by the time spent with each possible load factor and the percentage relative to the total execution time of $81.336\,\mathrm{s}$.

average, the actual consumption for all cores must be *higher* because the samples also include the consumption during the remaining $37.24\,\%$ of the execution when less cores are used. This implies for our energy model that the energy consumption we extrapolate when $2$ to $11$ cores are used must be an underestimation of the actual power usage since the slope of the linear function should be steeper. We believe that this error should be accounted for in future versions but that our results are still valid: Table 8.1 shows that only very few cores are idle during this remaining time and that the energy consumption hence remains relatively high. This means that the actual maximal consumption should only differ by a few watts. Another reason is that, when compared to the measured consumption of over $200\,\mathrm{W\,s}$, the actual error should only be a few percent.

## 8.6 Conclusions

The HPC community increasingly focusses on minimizing the power consumption by making hardware (computation units, network) and software (system applications and user applications) more energy efficient. Our approach clearly focusses on aiding the software side, but even network energy predictions with SimGrid have become possible through recent work [Gue+19].

System applications, such as batch schedulers and load balancers, can be improved with new energy-aware capabilities. Batch schedulers will implement intricate allocation policies accounting for energy constraints and we believe that our application-dependend energy model is flexible and faithful enough to support studies eval-

uating their efficiency. In fact, several projects [Dut+16b; Geo+15] are currently underway that aim to emulate batch schedulers such as SLURM or OAR on top of SimGrid and we have no doubt that these projects will be able to benefit from this work.

Assessing the energy consumption of applications for an existing machine can be useful to determine the best configuration to minimize power consumption. Unfortunately, algorithmic changes inside the application can only be faithfully studied if the energy profile used to instantiate the energy model is still valid.

As we already mentioned in Section 8.4, we have studied the performance of HPL and tried to re-obtain the TOP500 results for the Stampede supercomputer in a joint effort with Tom Cornebize [Cor+17]. In this work, power consumption was not investigated and studying HPL under this aspect with the goal to recompute the Green500 (i.e., the list of the most energy-efficient supercomputers) rating is certainly an interesting undertaking.

# 9 Contribution: Optimizing the Power Consumption With DVFS

We already mentioned several times (e.g., in Section 2.2) that minimizing power consumption will become the main optimization goal in the near future due to drastic increases in power consumption. Applications (and libraries) can take countermeasures by implementing power-aware algorithms. This, however, requires significant effort on the application developer's side. Another option is to use methods that can optimize the power efficiency of any or at least a class of applications passively, i.e., without making any modifications to the application. In this chapter, we study iterative HPC applications through an example, a geophysics simulator called Ondes3D, and possible energy gains through *dynamic voltage frequency scaling* (DVFS), a technique that promises to obtain power savings by reducing the frequency at which the CPU (and therefore, one of the most energy-hungry components of the system) operates. We therefore assume that the frequency is set per-CPU and not per-core. We furthermore compare our results to previously published results achieved with load balancing.

## 9.1 Context

### 9.1.1 Iterative Applications

Traditional HPC applications are often written with a particular model, called *bulk synchronous processes* (BSP) [Val90, p. 105], in mind. The BSP model works in three main steps. In the first phase, computations are executed by each process individually, followed by a communication phase and finally, a synchronization step that ensures that all computations and communications have finished, before the entire procedure is repeated. Applications following this model therefore principally work in a iterative way, with the synchronization ensuring that the current iteration has finished and every process is ready to move to the next iteration. A large number of HPC applications employ this model, making it particularly interesting to study how energy can be saved without making profound changes to these applications.

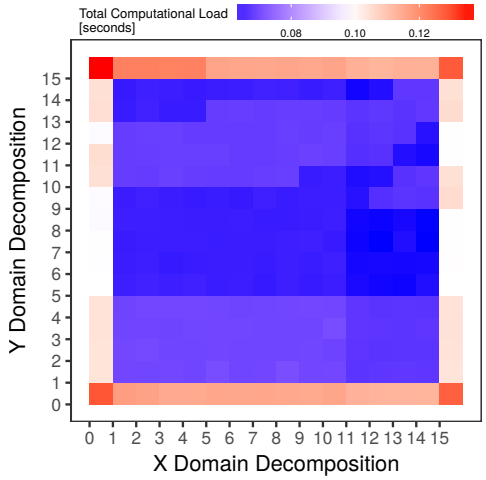


**Figure 9.1:** Illustration of the spatial load imbalance encountered during the first iteration of a run with $16 \times 16 = 256$ processes on the Ligurian workload. The imbalance consists of high load on the border, as more conditions have to be checked, and weaker but varying load (visualized by blue shades) in the interior that depends on the rock geology. The load imbalance is therefore workload dependent [Tes+18, Figure 2 (a)].

We chose Ondes3D[1] ("ondes" is the plural of the French physical term for "wave"), a geophysics simulator developed by the French Geological Survey (BRGM) to assess regional seismic hazards, as an application to test our optimizations as it works strictly with the BSP model. Ondes3D uses the finite-differences method (FDM) to approximate the partial differential equations required to compute the elastodynamic effects propagating through rock media. The application structure of Ondes3D is inherently regular: Ondes3D decomposes the problem domain into cuboids with equal, fixed geometries. Each process then executes the main loop (shown in Listing 1) on a single cuboid. The main loop primarily consists of calls to the kernels `Intermediates`, `Stress`, `Velocity` (which in turn call small FDM kernels) and communication calls, which only exchange messages between neighboring processes. The main loop does not contain any global synchronization, meaning that processes can evolve (slightly) independently. Despite this regularity, significant spatial- and temporal load imbalances prevent Ondes3D to scale well. Figure 9.1 visualizes the spatial load imbalance when simulating the "Ligurian" workload on a $16 \times 16$ process grid. The higher computational load on the processes managing the border cuboids can be explained by additional computations necessitated by boundary conditions. The processes in the interior are less heavily loaded. They nevertheless also experience a load imbalance, albeit of weaker nature, which is caused by heterogeneous geological conditions and therefore depends on the workload. Evidence found by Tesser et al. suggests that temporal load imbalance, depicted in Figure 9.2, is related to low-level optimizations within the CPU (see [Tes+18, Section 2.1] for more details).

[1] The following discussion of Ondes3D is a paraphrased summary of the presentation we published in a joint work with Rafael Keller Tesser [Tes+18, Chapter 2].

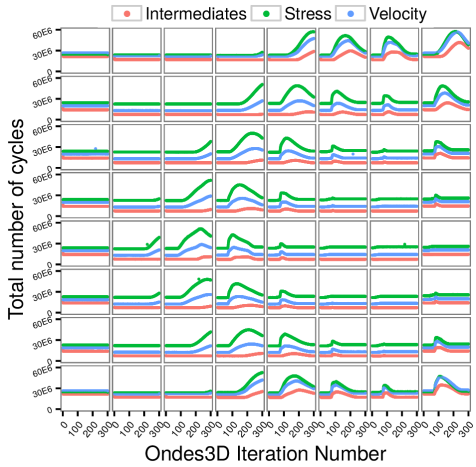


**Figure 9.2:** The evolution of the temporal load imbalance for $8 \times 8 = 64$ processes on the Ligurian workload [Tes+18, Figure 2 (b)].

```
for (ts = 0; ts < N; ts++){
  Intermediates();

  Stress();
  // Intertwined Asynchronous Neighborhood Communication

  Velocity();
  // Intertwined Asynchronous Neighborhood Communication
}
```

**Listing 1:** Ondes3D's main loop consists of three main kernels that each contain further (inlined) kernel calls. The communication calls after the `Stress` and `Velocity` functions have been omitted for simplification [Tes+18, Figure 1 (b)].

### 9.1.2 DVFS, a Means to Reduce the Power Consumption of HPC Applications

#### Application running on a single node

We already discussed in Chapter 8 that the frequency is the dominant factor of CPU power consumption (see Formula 8.2, page 90). Reducing the frequency "at the right time" can therefore help save energy, however, identifying this point in time and determining the right frequency is non-trivial. In fact, running computations at a lower frequency can have an adversarial effect, as is shown by Table 9.1: each row represents a single frequency and the corresponding measured power consumption of `taurus-7` when running the highly CPU bound benchmark `NAS-EP` on all cores (the low memory usage is also reflected by the consumed power). The ratio of the fastest frequency (2300 MHz) to the selected frequency is then used to scale the power consumption accordingly. This allows us to compare the power required to compute the same number of flops. When replacing a single second worth of

| Frequency (MHz) | Power (W) | Scale Factor | Scaled Power (W) | Loss (W) |
|---|---|---|---|---|
| 2300 | 179.25 | 1.000 | 179.250 | 0.000 |
| 2200 | 172.12 | 1.045 | 179.865 | 0.694 |
| 2100 | 166.25 | 1.095 | 182.044 | 2.833 |
| 2000 | 160.81 | 1.150 | 184.931 | 5.687 |
| 1900 | 155.12 | 1.211 | 187.850 | 8.527 |
| 1800 | 150.00 | 1.278 | 191.700 | 12.417 |
| 1700 | 144.75 | 1.353 | 195.847 | 16.588 |
| 1600 | 140.68 | 1.438 | 202.298 | 22.977 |
| 1500 | 136.00 | 1.533 | 208.488 | 29.283 |
| 1400 | 132.38 | 1.643 | 217.500 | 38.231 |
| 1300 | 128.88 | 1.769 | 227.989 | 48.768 |
| 1200 | 125.25 | 1.917 | 240.104 | 60.812 |

**Table 9.1:** Power consumption for the CPU-bound toy benchmark NAS-EP with one process per core (i.e., the machine operates under full load) as measured on `taurus-7` during our energy calibration. The scaled values are always scaled with regards to the fastest frequency (2300 MHz). On this machine, choosing lower frequencies on a fully-loaded machine is highly inefficient and results in significant power loss.

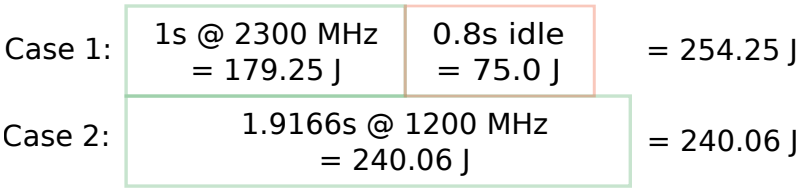


**Figure 9.3:** Reducing the frequency can save energy when power during idle times is counted as well. This is even the case when the finishing time is pushed back (here: running at reduced frequency takes 1.9166s as opposed to the 1.8s including idle time).

computations (of NAS-EP) at 2300 MHz, the power loss can be as high as 60 W (at 1200 MHz), since we now have to compute 1.916 s at 125.25 W (green boxes in Figure 9.3). It is tempting to conclude from this observation that a lower frequency only reduces the *momentary* power consumption of the machine, but that the *overall* power required to finish a task on a fully loaded machine increases. Luckily, this is only true when *merely* the power consumed during the respective computation is considered (i.e., when only the green boxes in Figure 9.3 are compared). In reality, computations are often followed by an (although often brief) period of idle time (red box in Figure 9.3), which still consumes significant energy as the machine is still powered on. When running at a slower frequency, this idle time is reduced (or even disappears completely) as it is instead used for computations. This is illustrated by Case 2, which also proves that a computation may take *longer* than the previous idle and computation time together and still save power.

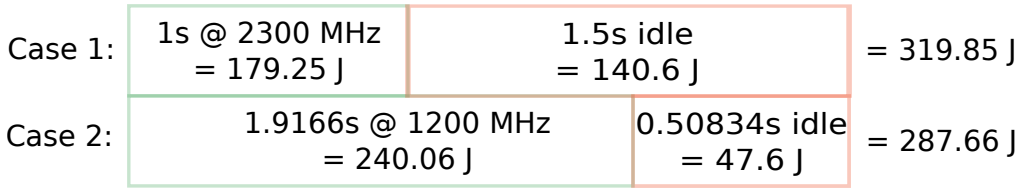


**Figure 9.4:** It is possible to save energy without introducing a delay by overlapping idle time with computations.

### Application running on several nodes

When executing a parallel application, the situation becomes more complicated. In the case of an application running on a single node, the DVFS algorithm only has to determine if energy can be saved by reducing the frequency on that machine, regardless of whether the makespan increases or not. Once the application runs on several nodes, a decision at the node level may be locally optimal, but can create a jitter in the system. To understand this, consider the following scenario: imagine that the idle time in Case 1 of Figure 9.3 comes from a blocking communication call, say a blocking send, that is immediately issued after the computation. After a total of $1.8\,\mathrm{s}$, the process can continue since it no longer has to wait for another process to post the matching receive (or else the idle time would be longer). Note that the other process has no idle time waiting for the (sufficiently small) message. In Case 2, however, the blocking send is posted only after $1.9166\,\mathrm{s}$ and therefore later than in Case 1. If the receiving process has not changed itself, it will now have to wait around $0.1166\,\mathrm{s}$ for the send to be posted. This means that, without further restriction, opportunistic local decisions can impact (and slow down) other processes as well, creating a jitter. Therefore, DVFS managing a parallel application should reduce the power consumption but should not degrade performance in doing so.

Although it is non-trivial to save energy on platforms (which is why HPC systems normally refrain from adapting the frequency), it still is possible. In the scenario of Figure 9.4, for example, it was ensured that the computation time in Case 2 is not extended beyond the idle time, meaning that the process still has to wait and does not slow others down.

We can therefore ascertain the following: let $t_{comp}(f)$ and $t_{idle}(f)$ denote the computation- and consecutive idle time, respectively, when running at frequency $f$. Let furthermore denote $s$ the start time of the computation, $P(f)$ the power required by frequency $f$ and $P_{idle}$ the power when idling. We then have

$$t_{comp}(f) \cdot P(f) + t_{idle}(f) \cdot P_{idle}$$

as the total power consumption when running at frequency $f$ (Case 1). Energy savings without changing the finishing time $t_{comp}(f) + t_{idle}(f)$ are possible if the reduction in energy during the period $[s, s + t_{comp}(f)]$ (green box in Case 1) outweighs the increase in power during the period $[s + t_{comp}(f), s + t_{comp}(f')]$ (the node is idle during the remaining time and power consumption hence does not change). This means that when

$$t_{comp}(f) \cdot (P(f) - P(f')) > (t_{comp}(f') - t_{comp}(f)) \cdot (P(f') - P_{idle})$$

holds, then switching the frequency saves energy.

### Kernel Level DVFS

To start the discussion on DVFS governors, we will now present four DVFS governors that are implemented in the linux kernel `cpufreq` driver.[2] They are purely opportunistic (or even static) and do hence *not* consider other processes in their decision process. In this section, we only present the underlying principles and expected behavior. Experimental results will be presented in Section 9.4.

► **Performance** The cpufreq performance governor is a static governor that uses the fastest frequency, regardless of the current machine load. This is useful when computational tasks should be executed as quickly as possible, regardless of potential power savings.

► **Powersave** The cpufreq powersave governor is a static governor and by default limited to the lowest available frequency. The CPU therefore provides the least computational power but also uses the lowest power at any given instant. When integrating the *total* power consumption for a specific computation, it can turn out that using the lowest frequency on a fully loaded CPU consumes *more* power than using a faster frequency that can finish the job faster. The name "powersave" is therefore deceiving since power is not necessarily saved.

Note that this only refers to the governor from the cpufreq driver; the intel_pstate driver with the same name can change the frequency and hence operates fundamentally different. This is, however, not considered here.

[2] The governors can be found in the kernel's source tree under `source/drivers/cpufreq/cpufreq_*`

```
while (true) {
  if (get_current_load() > freq_up_threshold) {
    set_pstate(fastest_pstate);
  } else {
    new_pstate = get_max_pstate() -
                 get_current_load() * (get_max_pstate() + 1)
    set_pstate(new_pstate)
  }
  sleep(update_interval_length);
}
```

**Listing 2:** The ondemand governor "panics" once a given threshold has been crossed and jumps immediately to the fastest frequency, even if the load spikes only for a short time.

```
while(true) {
  if (get_current_load() > freq_up_threshold) {
    set_pstate(get_next_faster_pstate(current_pstate));
  } else {
    set_pstate(get_next_slower_pstate(current_pstate));
  }
  sleep(update_interval_length);
}
```

**Listing 3:** The conservative governor in- and decrements the used frequency only in single steps.

▶ **Ondemand** The ondemand governor (Listing 2) is a non-static governor that considers the current load of the system. Both the minimum as well as the maximum frequencies can be set via the ondemand governor.

On every update, it compares the load to a configurable threshold (with a default value of `0.8`). If it has been surpassed, the governor sets the speed to the fastest possible pstate. If the load is lesser, then the new pstate is determined linearly by selecting the pstate that is in the right position.

▶ **Conservative** Once a specific load threshold has been crossed, the ondemand governor "panics" and immediately switches to the fastest pstate. The conservative governor (Listing 3), however, reacts more slowly and only switches to the next faster (or slower) pstate. This means that for only short bursts of computations, the conservative governor can remain in a slower state, but it will take longer to reach the fastest state in case of longer computation phases.

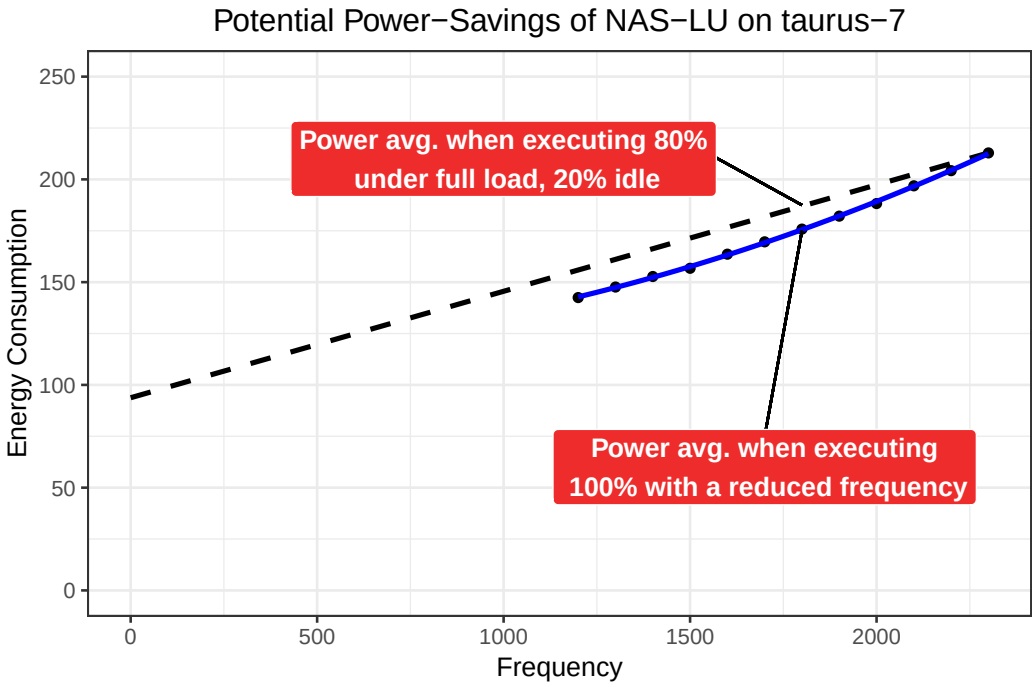


**Figure 9.5:** Alternating between the highest frequency and the idle state (represented by frequency "0") consumes more power than running at a reduced frequency (with the same finishing time). The power data was obtained by actual measurements of NAS-LU.

### Inefficiency of common DVFS governors with HPC applications

The already mentioned system-wide jitter (see page 111) caused by opportunistic DVFS governors can be traced back to the obliviousness regarding the structure of the currently running application. This missing knowledge combined with the two major states of parallel machines (high load or idle) is also reflected by the selected frequency itself, as governors either select the highest frequency due to the high load or the machine is idle (and in this case, the power consumption does not depend on the selected frequency). Figure 9.5 shows the power consumption as a function of the frequency. The solid line shows a possible model for the power consumption we measured for NAS-LU. The dashed line, on the other hand, visualizes the mean power consumption when alternating between fully loaded and idle. The difference between the dashed and the solid line is the power saving potential that can be achieved without slowing the application down.

It is therefore important to provide governors with additional information about the application so that they can select the best frequency quickly and without causing detrimental jitter.

## 9.2 Related Work

### 9.2.1 Adagio (Application Level DVFS)

Adagio [Rou+09] is an application level governor that was developed in 2009 and specifically addresses the weakness of the previously presented governors by combining the exploitation of the regularity of MPI applications and knowledge about the current position within the application. The main idea behind Adagio is easily comprehensible and follows the concept we already presented during the previous discussion of Figure 9.4: A process's idle time between two computations (e.g., caused by a blocking communication call) can be reduced by slowing the preceding computation down. For this, Adagio intercepts all MPI calls and selects the slowest frequency that does not cause a slow-down, i.e., the computation/communication pair always requires in total the same time. As we saw before, this effectively leads to energy savings.

When Adagio takes action, it identifies the current position by inspecting the callstack. It does not require any learning on previously collected data and its decision on the best frequency is therefore only influenced on the current workload. Listing 4 reproduces the pseudo-code as published in the original paper [Rou+09, Figure 2].

Adagio's logic is principally split in two functions that represent the time when they are executed: Before (`PreTask()`) and after (`PostTask()`) a computation.

▶ **`PreTask()`** This function identifies the current location within the application based on the current call stack and sets the frequency for the following computation. If the call comes from a location that has never been reached before, the first execution of the following computation is always executed with the maximum frequency $\hat{f}$. This ensures that no avoidable performance degradation occurs. If the following computation has been executed before, the previously computed best frequency is loaded from the map $Sched$ and activated before the computation is executed.

▶ **`PostTask()`** Once the computation has finished, the `PostTask()` function is called. As a first step, the number of instructions $I$, the time spent computing $t_{comp}$ and the time spent communicating $t_{lib}$ is recorded. The time that any selected frequency must not surpass is called the *target time* ($t_{target}$, Line 21) and consists of the time spent computing and communicating minus a small buffer, called $t_{copy}$,

```
1  PreTask()
2    taskid = hash(stack_pointer_chain)
3    if (isnew(taskid)) {
4      /* First instance of a task: */
5      /* Choose fastest frequency. */
6      f = f̂
7    } else {
8      /* Look up correct frequency. */
9      f = Sched[taskid]
10   }
11   SetFreq(f)
12   InitPerformanceCounters()
13   RunTask(taskid)
14
15 PostTask()
16   /* Generate the schedule for the */
17   /* next execution of this task. */
18   Record I, t_comp, t_lib.
19   Rates[taskid][f] = I/t_comp
20   t = t_comp + t_lib
21   t_target = t − t_copy
22
23   if (isnew(taskid)) {
24     /* First instance of a task: */
25     /* Set slowdown rates to */
26     /* worst-case for each */
27     /* available frequency. */
28     for (f ∈ F) {
29       Rates[taskid][f] = Rates[taskid][f̂] × f̂/f
30     }
31   }
32
33   /* Find slowest frequency that */
34   /* respects the critical path. */
35   /* Default is fastest freq. */
36   Sched[taskid] = f̂
37   for (f from slowest(f̄) to fastest(f̂)) {
38     if (I/Rates[taskid][f] ≤ t_target) {
39       Sched[taskid] = f
40       return;
41     }
42   }
```

**Listing 4:** The original Adagio source code (with slight modifications for presentation purposes) from the publication [Rou+09].

which represents the cost for copying the message and is hence inevitable. Note that the time $t_{lib}$ also includes $t_{copy}$.

Recall that for a first-time execution, the frequency is always fixed to the fastest frequency (Line 6) to ensure that no slowdown is possible. In this case, each other frequency $f$ is initialized by scaling the number of instructions/second with $\hat{f}/f$ (Line 29).

Subsequently, the slowest frequency is searched that can still finish the workload within time $t_{target}$. Starting with the slowest frequency, the first frequency that satisfies this constraint is selected for the next time this computation is executed.

### 9.2.2 Load Balancing with Adaptive MPI (AMPI)

Recall from Section 9.1.1 that the load imbalance of Ondes3D depends on the initial and evolving conditions of the simulated workload. This makes imbalances difficult to model and predict up front and using application-level balancing techniques to prevent the load issues requires forecasting of future iterations. Thankfully, another approach exists that is simpler to use: the domain can be over-decomposed into more cuboids than processing cores are available. If each computation is executed in a migrateable task, an existing load imbalance can be alleviated at runtime by moving a subset of tasks from one processor to another.

This functionality is for instance provided by Adaptive MPI (AMPI) [Acu+14], an extended MPI implementation that provides several new functions to deal with load imbalances in MPI applications, such as `MPI_Migrate()`. Internally, AMPI uses the load balancing features implemented in the thread-based Charm++ [KK93] runtime. Porting an application to AMPI requires several steps, such as the removal of global and static variables (since these would be shared between threads), the implementation of pack/unpack functions that serialize all data structures before moving the thread to a new processor and calling `MPI_Migrate()` in the right place (likely at the end of the outermost loop, i.e., at the end of a time step). Once an application has been ported to AMPI, it can take advantage of the load balancing mechanisms. The resulting performance depends on several factors: First of all, the chosen heuristic. Distributing the load more evenly over all cores is useless if the computation and communication costs of the migration outweigh the saved time. Second, the level of over-decomposition influences the granularity at which the load balancer can make its decisions: with a high over-decomposition level, each thread is assigned a smaller load and can be more easily used to fill-in available computational resources. However, more MPI processes also come with more communication and possibly locality issues. Third, the frequency at which

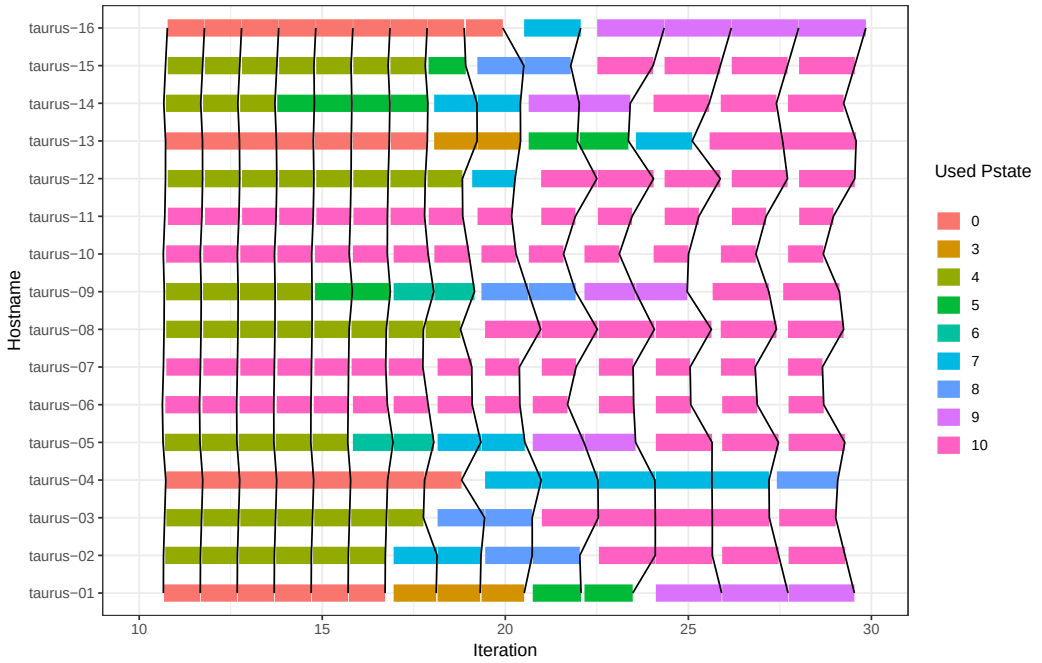


**Figure 9.6:** Using Adagio with Ondes3D works for about 15 iterations, after which even the most loaded host `taurus-16` starts to slow down.

`MPI_Migrate()` is called determines how fast load imbalances can be identified and resolved. Alas, this also means that the possibly costly migration needs to be executed more often. `MPI_Migrate()` acts as a global barrier and therefore has the potential to deteriorate overall performance.

The performance boost an application can get from load balancing therefore depends on tuning these parameters. In the case of Ondes3D, which has already been ported to AMPI, an up to $28.35\,\%$ faster execution on a 8-node / $64$ core machine was the consequence of using AMPI. Determining whether porting an application to AMPI is worth it and which parameters should be used is an interesting question that can be answered with the help of simulators. SimGrid already supports load balancing through its SAMPI API [Tes+18, Chapter 4], an extension of the SMPI API (Section 4.3) that supports the `MPI_Migrate()` function and can estimate the cost of migrating a task by tracking the allocated memory through overwriting `malloc()` (and consorts) and `free()` functions.

Emulating the entire application for each combination of parameters is inefficient, as the load balancing algorithm never changes the sizes of workloads but only re-assigns them to a new host. Time independent traces (discussed in Section 4.3.3) capture the behavior of the application and can be replayed much quicker than a full emulation. Using a trace also comes with the guarantee that all experiments are carried out on exactly the same workload (as it is represented by the traces), i.e.,

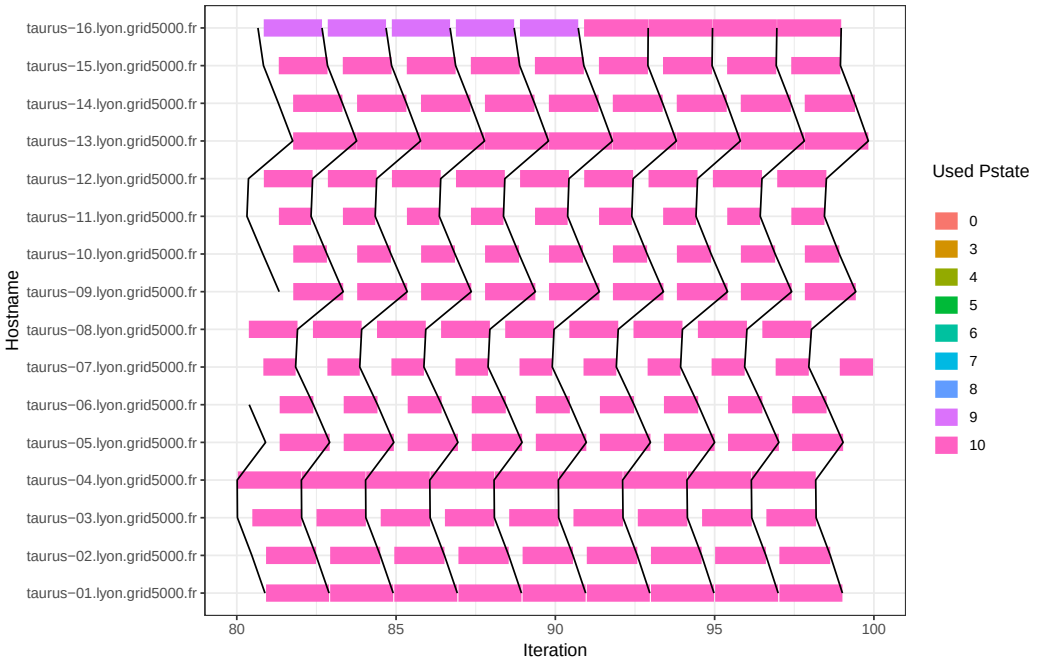


**Figure 9.7:** Around the 85th iteration of the Chuetsu earthquake scenario (with 16 processes) even the last (and most loaded) node `taurus-16` enters the lowest pstate. The vertical lines help to distinguish the start and end of each iteration.

the reported outcome of a specific configuration (e.g., load balancing parameters, chosen DVFS governor) cannot be influenced by temporary effects such OS noise.

### 9.2.3 Residual Load Imbalance

As aforementioned, the granularity of the over-decomposition impacts the computational workload assigned to each task and therefore the load balancer's possible choices when it comes to selecting the tasks for migration. It is therefore common that even after the load balancer has finished the migration period, the load is still not fully uniformly distributed. In this situation, called the *residual load imbalance*, secondary options such as DVFS can (when correctly used) reduce the frequency of underloaded processors to increase powersavings [Pad+14].

## 9.3 Contributions

In this section, we start by analyzing the limitations of existing governors. Then, building on this analysis, we propose two new governors.

Adagio assumes that the runtime of an identified task will not differ significantly between two runs [Rou+09, p. 463]. This condition is normally fulfilled for highly

regular applications but inaccurate for applications that suffer from temporal load imbalances. Applications such as Ondes3D can indeed not benefit from Adagio since the code as presented in Listing 4 can only slow-down, but never speed-up. We found that Adagio does work for a few iterations with Ondes3D (Figure 9.6), but even the most loaded machine started to slow down after around 15 (out of 300) iterations. This caused other machines to further slow down and in the end every single node selected the lowest frequency (Figure 9.7).

This can be seen as follows: We have $t_{target} = t_{comp} + t_{lib} - t_{copy}$ (Line 21) with $t_{lib} \geq t_{copy} \geq 0$ (because $t_{lib}$ includes $t_{copy}$ [Rou+09, Table 2]) and hence $t_{target} \geq t_{comp}$. Note that $t_{comp}$ is a measured value and thus depends on the current frequency. This also implies the same for $t_{target}$ (because it contains by definition $t_{comp}$) while $t_{copy}$ is independent of the frequency. This means that once the for-loop in Line 37 has reached the *current* frequency (and hence rejected all slower frequencies), the condition it tests becomes (with the definition in Line 19) $I/Rates[taskid][f] \overset{Def}{=} t_{comp} \leq t_{target}$. As we just saw, this is always true, and hence the frequency is always re-accepted. Any *faster* frequency is therefore never tested and a speed-up cannot occur.

### 9.3.1 DVFS Governor: AdagioImproved

Each time a node wants to determine whether it needs to scale the frequency up or down, it needs to first find out whether it is waiting for others (and hence a slowdown is reasonable) or if others are waiting for it (and it needs to run faster). There are several ways to answer this question.

► **Global Communication** A first solution is to use an all-to-all global communication call. Each node sends its own communication time and receives all others, to which it can compare its own time. Although this solution is very easy and can be implemented quickly, it is highly undesireable, because it introduces significant overhead through the additional communication call which can deteriorate performance.

► **Transfer Model** A second proposal is to use a precise model of transfer time (that includes congestion and latency). With this model, a node can calculate the transfer time for a message and then subtract it from the overall communication time. This yields the time spent waiting. With almost no waiting time, i.e., when it is close to $0$, a node can safely assume that others are waiting and hence speed up. This is essentially how Adagio works, even though Adagio does not include congestion and latency but merely the copy time.

```
1 if (t_lib > t_min · factor) { // we're waiting.
2   Adagio() ; // should decrease t_lib
3 } else { // we're too slow
4   i = get_current_pstate_id()
5   f = f_{i/2} // Avoid jumping to the fastest pstate
6           // bisect the pstates instead
7   t_min = t_lib
8 }
```

**Listing 5:** Minor modifications allow Adagio to speed-up if needed.

► **Statistics** When these two approaches are not usable, for instance, because no such model exists, a decision can be made solely on the data that was collected so far. This means that the decision process is based on a statistic from previous *iterations* (but not application runs).

We found that very simple changes (see Listing 5) to Adagio are sufficient to make it work with Ondes3D, but we believe that other applications that suffer from temporal load imbalances can use this approach, too. First, each node saves the minimum time spent communicating (over the last iterations) in a variable called $t_{min}$. Adagio's missing speedup condition can (under the premise that message size remains constant) then be implemented as follows: When the communication time $t_{lib}$ has increased compared to the previously observed minimum $t_{min}$ (times a specific factor to counter inevitable variation caused by noise), a slower frequency is selected by the original Adagio algorithm because the waiting time has increased. In the other case, i.e., when the communication time is either the same or when the node spent less time than previously, the other nodes seem to be already waiting. The node then sets the frequency to the pstate that lies in the middle of the current and the fastest pstate. Avoiding to jump immediately to the fastest pstate helps to avoid constant changes of the frequency, as the frequency would likely be reduced in the following step. This is similar to many adaptive schemes (e.g., the *additive increase, multiplicative decrease* approach used in TCP [Jac88]) but whether such "natural" scheme would be efficient in this particular context remains to be proven.

### 9.3.2 DVFS Governor: Lagrange

#### Mathematical, continuous formulation

A common approach to minimize a convex function $F$ is to use a gradient-based algorithm. For a stepsize $\epsilon > 0$ and iteration $t$, we can find the next value by walking into the direction of the gradient:

$$x(t+1) = x(t) - \epsilon(t) \cdot F'(x(t))$$

Determining the right value for $\epsilon(t)$ is cumbersome. To avoid oscillations, $\epsilon$ is often set as $\epsilon(t) = \frac{1}{t}$, however, the convergence is slow when large values of $t$ are attained and if $F$ evolves with $t$, then the learning process is likely to become more and more inefficient. Furthermore, if $x$ is distributed over different computing ressources, paying particular attention to the computation of $F'(x(t))$ will be required.

Although this approach is simple and easy to understand, it is difficult to use when existing constraints restrict the possible solutions. A common formulation for such situation is the following:

$$\min F(x,z) \text{ s. t. } Ax + By = c$$

with $x \in \mathbb{R}^n, z \in \mathbb{R}^m, c \in \mathbb{R}^p$ and $A \in \mathbb{R}^{p \times n}, B \in \mathbb{R}^{p \times m}$. To fall back to an unconstrained optimization problem, these constraints can be incorporated into the optimization objective, for example by introducing a new dual variable, $\lambda$, which is associated to the constraint $Ax + By = c$ and which can be interpreted as an allowed error.

This optimization problem can be reformulated using the *augmented Lagrangian* [Boy10, Chapter 3]:

$$L_\rho(x,z,\lambda) = F(x,z) + \frac{\rho}{2} \cdot ||Ax + Bz - c - \lambda||_2^2$$

where a *penalty parameter* $\rho > 0$ controls the impact of the term $Ax + Bz - c - \lambda$, which accounts for violations of the constraints. Indeed, $L_\rho(x,z,\lambda) = F(x,z)$ if and only if $\lambda = Ax + Bz - c$, and hence is consistent with our interpretation of $\lambda$ as a measure of how much the constraints are violated. One can prove that

$$\min_{\substack{x,z \\ Ax+Bz=c}} F(x,z) = \max_{\rho>0} \min_{x,z} L_\rho(x,z,\lambda)$$

The following algorithm, called *Alternating Direction Method of Multipliers* solves the previous optimization problem [Boy10, Chapter 3].

$$x^{t+1} = \arg\min_x(L(x, z^t, \lambda^t)) \tag{9.1}$$

$$z^{t+1} = \arg\min_z(L(x^{t+1}, z, \lambda^t)) \tag{9.2}$$

$$\begin{aligned}\lambda^{t+1} &= \arg\min_\lambda(L(x^{t+1}, z^{t+1}, \lambda)) \\ &= \lambda^t + (Ax^{t+1} + Bz^{t+1} - c)\end{aligned} \tag{9.3}$$

with $x \in \mathbb{R}^n, z \in \mathbb{R}^m$, $\lambda, c \in \mathbb{R}^p$ and $A \in \mathbb{R}^{p\times n}, B \in \mathbb{R}^{p\times m}$. Note that this algorithm does not require a step size and that it can be used in a distributed manner when $F$ is separable, i.e., $F = \sum_i F_i$.

### Application in our context

For a node $i$, we are mainly interested in two values: $f_i$, the **inverse** of the node's frequency (in flop/s), and which we still denote $f$ to simplify notations, and $W_i$ which denotes its (constant) workload. The time it takes to compute $W_i$ on node $i$ at "frequency" $1/f_i$ is $f_i \cdot W_i$.

Let $E^i(f)$ denote the energy node $i$ consumes with frequency $\frac{1}{f}$ and $E^i_{idle}$ its idle power consumption. We claim that minimizing the total energy consumption is equivalent to minimizing the following function:

$$\sum_i \left( f_i \cdot W_i \cdot (E^i(f_i) - E^i_{idle}) + \left( \max_j (f_j \cdot W_j) \right) \cdot E^i_{idle} \right)$$

This can be interpreted as follows: The slowest node (= maximum execution time) determines how much idle time is spent on all nodes when waiting for the synchronization to finish. This corresponds to the last term with the max. When node $i$ computes, which takes $f_i \cdot W_i$ time, it consumes $(E^i(f) - E^i_{idle})$ watts more than when idling. This yields the total power consumption of all nodes.

To make this optimization amenable to distribution, we now introduce for each node $i$ a helper variable $t_i$ that can be thought of as the computation time estimate

of node $i$. We furthermore add another variable, $s$, which can be interpreted as the maximum of all $t_i$. This allows us to rewrite the above as

$$\begin{aligned}&\text{MINIMIZE } \sum_i \left(f_i \cdot W_i \cdot (E^i(f_i) - E^i_{idle}) + s \cdot (E^i_{idle}) \cdot \infty_{\{t_i \leq s\}}\right), \\ &\text{UNDER THE CONSTRAINTS} \\ &\begin{cases} f_i \cdot W_i & = t_i \\ t_i & \leq s \end{cases}\end{aligned} \tag{9.4}$$

As we explained during the discussion of the ADMM algorithm, the first constraint is enforced by $\lambda^i$. The expression $\infty_{\{t_i \leq s\}}$ is 1 if $t_i \leq s$ and $\infty$ otherwise. This forces us to only accept values for the random variables $t_i$ that fulfill the requirement $t_i \leq s$ because otherwise, the function evaluates to $\infty$ which is not the minimum.

This optimization problem can now be formulated with the Lagrangian formulation by setting the variables as follows:

$$x = (f_1, \ldots, f_N), z = (t_1, \ldots, t_N)$$

and $A, B$ are

$$A = diag(W_1, \ldots, W_N), B = diag(-1, \ldots, -1), c = (0, \ldots, 0)$$

With these values, the error term $||Ax + Bz - c - \lambda||^2$ can be written as the following sum, making it separable

$$\sum_i (f_i \cdot W_i - t_i - \lambda_i)^2 \tag{9.5}$$

### Discrete implementation

Lagrangian optimization requires a *continuous* domain, however, the possible frequencies are discrete. We therefore had to modify the algorithm for our implementation.

► **Step 1: Adjusting the frequencies (ADMM step (9.1))** In iteration $t + 1$, node $i$ first determines the frequency that is used for the next iteration. The separability of the objective function in (9.4) allows us to write it as $\sum_i g_i(f_i)$ with

$$g_i(f) = f \cdot W_i \cdot E^i(f) + \min(s, f \cdot W_i) \cdot E^i_{idle}$$

Each node should just locally minimize $g_i(f) + \frac{\rho}{2}(f \cdot W_i - t^t_i - \lambda^t_i)^2$, which depends solely on the frequency $f$. The function $E^i(f)$ can be modeled with a polynom and

therefore the optimal value $f_{opt}$ can be determined by deriving the function $g_i$. Unfortunately, the target frequency corresponding to $f_{opt}$ is not necessarily supported by node $i$. In the case where $f_{opt}$ is in between of two available frequencies $f^1$ and $f^2$, frequency $f^2$ could be used first and with the help of a timer later switched to $f^1$ to obtain on average the frequency $f_{opt}$.

To keep our implementation as simple as possible, we have not pursued this approach. Instead, we evaluate the target function for each available frequency and switch to the most well suited frequency, $f_{min}$.

As discussed above, the variable $s$ should normally be set to $\max(t_i)$, however, the $t_i$ are node-specific and are hence not known globally. However, the duration of the iteration can be measured locally and we fixed the part of the duration that can be used for computations as a configuration option. This allows us to avoid global communicaton by substituting the $s$ in the previous formula with

$$s = overlayable\_iteration\_time \cdot iteration\_duration_i$$

In our implementation, we arbitrarily set $overlayable\_iteration\_time$ to $0.95$.

► **Step 2: Updating the estimated execution times (ADMM step (9.2))** In our case, the error term can be written as shown in (9.5). In the best case, this term vanishes completely, i.e., it evaluates to zero. As every element of the sum is squared (and hence $> 0$), an error introduced by element $i$ cannot be offset by element $j$. We can therefore set the $t_i$ independently of all others. Setting

$$t_i = f_i \cdot W_i - \lambda_i$$

is desireable, but not always possible, since the constraints in (9.4) require $t_i \leq s$. This implies that we have to consider the case $f_i \cdot W_i - \lambda_i > s$ as well. Indeed, we can then only subtract at most $s$, as the constraint requires and set

$$t_i^{t+1} = \min(f_i^{t+1} \cdot W_i - \lambda_i, s)$$

► **Step 3: Updating the error-variables $\lambda_i$ (ADMM step (9.3))** Recall that $\lambda$ was introduced to enforce the condition $t_i \leq s$. We now update $\lambda_i$:

$$\lambda_i^{t+1} = \lambda_i^t + (f_i^{t+1} \cdot W_i - t_i^{t+1})$$

## 9.4 Performance Evaluation/Effectiveness

The energy model we presented in Chapter 8 allowed us to implement and study DVFS governors in SimGrid.

We implemented Adagio, our proposed improvement AdagioImproved, and our own contribution Lagrange. We have furthermore implemented the four governors we discussed in Section 9.1.2: performance, powersave, ondemand and conservative. The implementation of the linux governors is basic and only uses their essential logic. Although some governors offer several configuration options, we only implemented the ones we needed: the sampling frequency, allowed subset of pstates and speed-up / slow-down thresholds.

Recall from Section 9.2.2 that AMPI can be used to load balance MPI applications and that Ondes3D has already been ported to AMPI. SimGrid also supports AMPI through its own SAMPI API, and we implemented and carefully tested the GreedyLB heuristic to obtain realistic predictions [Tes+18].

We furthermore discussed in Section 4.2.2 that by using trace replay, one can capture the application behavior once and then test different configurations with the exact same trace. Besides being able to guarantee that all tests used the exact same experimental basis, the simulation also benefits from a speed-up as no instructions from the application's binary are executed.

To evaluate the aforementioned approaches, we obtained two traces (immediately one after the other) for 300 iterations of the *Chuetsu* earthquake scenario executed on the `taurus` cluster in Lyon. For the first trace, we used 16 processes (1 per node) and replayed it on top of SimGrid with different DVFS governors activated. The second trace, on the other hand, uses 64 processes (4 per node), effectively allowing us to ensure over-decomposition (which is required for the load balancer) when a single-core system is used. We replayed this trace on top of SimGrid with the GreedyLB load balancing heuristic and the frequency fixed to the maximum.

### 9.4.1 DVFS/Adagio/AdagioImproved/Lagrange

Recall from Section 9.1.2 that in order to save energy with DVFS, idle time must be present. This is the case on almost all nodes when simulating the Chuetsu scenario with Ondes3D, as can be seen in Table 9.2, which shows the idle times when fixing the frequency to the maximum (i.e., for the performance governor). The most charged node, `taurus-16`, experiences almost no idle time whereas

| Host | Total idle time | % Idle |
|---|---|---|
| taurus-01 | 13.5998 | 4.00 |
| taurus-02 | 85.4122 | 25.14 |
| taurus-03 | 86.1985 | 25.37 |
| taurus-04 | 5.59119 | 1.65 |
| taurus-05 | 84.1905 | 24.78 |
| taurus-06 | 176.189 | 51.86 |
| taurus-07 | 177.232 | 52.17 |
| taurus-08 | 87.0137 | 25.61 |
| taurus-09 | 84.982 | 25.01 |
| taurus-10 | 177.136 | 52.14 |
| taurus-11 | 176.603 | 51.98 |
| taurus-12 | 86.9111 | 25.58 |
| taurus-13 | 9.99497 | 2.94 |
| taurus-14 | 85.9065 | 25.29 |
| taurus-15 | 86.5902 | 25.49 |
| taurus-16 | 2.24836 | 0.66 |

**Table 9.2:** Total idle time of all nodes when using the performance governor (Chuetsu, 300 iterations, 16 processes). Total runtime was 339.74 s

| Governor | Makespan (s) | Energy (J) | Energy Savings/Loss (in percent) |
|---|---|---|---|
| Performance | 339.747 | 989016 | 0. |
| Powersave | 677.152 | 1460435 | 47.665 |
| Ondemand | 339.757 | 986237 | -0.281 |
| Conservative | 339.755 | 984329 | -0.474 |
| Adagio | 667.189 | 1445656 | 46.171 |
| AdagioImproved | 340.129 | 966994 | -2.227 |
| Lagrange | 340.146 | 963964 | -2.533 |

**Table 9.3:** Makespan and energy predictions we obtained when using several governors with the Chuetsu scenario (300 iterations, 16 processes).

others (e.g., `taurus-06`) are idle just over 50 % of their total execution time. It is therefore reasonable to expect that DVFS can save power. This is indeed the case, as shown in Table 9.3. The Lagrange (with $\rho = 10$) and AdagioImproved governors managed to save up to 2.5 % over the 300 iterations, whereas the linux kernel governors `ondemand` and `conservative` save less than 0.5 %. Powersave, which fixes the frequency to the lowest speed, and the original Adagio implementation (which cannot speed up and hence ends up using the lowest frequency as well, see Section 9.3)) even worsen the situation by around 47 %, which is expected.

It is interesting to study the reduction of idle time for the different governors, as a remaining high idle time can indicate that additional savings are possible by slowing down even more, yet the frequencies cannot be further reduced due to hardware limitations. In the case of the Lagrange (Table 9.4) and AdagioImproved

| Host | Total idle time | % Idle |
|---|---|---|
| taurus-01 | 12.0648 | 3.55 |
| taurus-02 | 21.1016 | 6.20 |
| taurus-03 | 20.7913 | 6.11 |
| taurus-04 | 5.98996 | 1.76 |
| taurus-05 | 20.5558 | 6.04 |
| taurus-06 | 68.1604 | 20.04 |
| taurus-07 | 69.837 | 20.53 |
| taurus-08 | 21.6866 | 6.38 |
| taurus-09 | 20.9417 | 6.16 |
| taurus-10 | 69.8326 | 20.53 |
| taurus-11 | 68.788 | 20.22 |
| taurus-12 | 21.4463 | 6.31 |
| taurus-13 | 10.3937 | 3.06 |
| taurus-14 | 22.6129 | 6.65 |
| taurus-15 | 22.7786 | 6.70 |
| taurus-16 | 2.64713 | 0.78 |

**Table 9.4:** Total idle time of all nodes when using our proposed Lagrange governor. Total runtime: 340.14 s. (Chuetsu, 300 iterations, 16 processes)

governors (Table 9.5), idle times are significantly reduced but still remain present, particularly for the hardly loaded nodes `taurus-6,7,10,11`. For Lagrange, these nodes continue to idle just over a minute whereas this is only true for AdagioImproved for `taurus-11`. The other nodes have a reduced yet still high idle time. Note that these nodes switched to the lowest frequency during the first iterations, as shown in Figure 9.6 for the Lagrange governor. The ondemand (Table 9.6) and conservative (Table 9.7) governors on the other hand fail to reduce the idle time for almost all of their nodes, and they are hence unable to reduce energy notably.

### 9.4.2 Efficiency Comparison DVFS / Load Balancer

DVFS only controls the frequency of a node and, once the lowest frequency is reached, cannot reduce energy consumption of this node further. A load balancer, on the other hand, can move computations to underloaded nodes. This allows the nodes to be very energy efficient as they can work at full speed most of the time. Table 9.8 compares an execution at maximum frequency with executions that benefit from load balancing after a pre-determined number of iterations (denoted as an index of the heuristic name). For the Chuetsu scenario, we determined experimentally that calling the GreedyLB algorithm every 5 iteration is the optimum migration frequency. Doubling this 10 iterations results in just over one minute (21%) of reduced execution time and 11.1 % of energy savings. However, even when load balancing is employed, idle time does not fully disappear (Table 9.9), even though less than with the Lagrange and Performance governors remains.

| Host | Total idle time | % Idle |
|---|---|---|
| taurus-01 | 10.4653 | 3.08 |
| taurus-02 | 23.9697 | 7.05 |
| taurus-03 | 26.3388 | 7.74 |
| taurus-04 | 5.91632 | 1.74 |
| taurus-05 | 20.9864 | 6.17 |
| taurus-06 | 55.1024 | 16.20 |
| taurus-07 | 45.8586 | 13.48 |
| taurus-08 | 27.1594 | 7.99 |
| taurus-09 | 27.8492 | 8.19 |
| taurus-10 | 36.5526 | 10.75 |
| taurus-11 | 66.0996 | 19.43 |
| taurus-12 | 30.7309 | 9.04 |
| taurus-13 | 8.73714 | 2.57 |
| taurus-14 | 24.6275 | 7.24 |
| taurus-15 | 39.3532 | 11.57 |
| taurus-16 | 2.62996 | 0.77 |

**Table 9.5:** Our improved version of Adagio manages to reduce idle time significantly. Total runtime: 340.12 s. (Chuetsu, 300 iterations, 16 processes)

A coordinated DVFS governor / load balancer approach would be able to ameliorate the situation by reducing this residual imbalance [Pad+14].

## 9.5 Limitation and Future Work

### 9.5.1 Unavailable Frequencies and More Complex Architectures

When we discussed how we implemented the first step of the ADMM algorithm (Section 9.3.2), we already mentioned that using interrupts to lower the frequency during a compute is a possibility to use (on average) a frequency that is optimal yet (due to the discrete nature of frequencies) not available on a host. This approach was also proposed for Adagio [Rou+09]. Implementing this feature is interesting and may not be too difficult to do, however, Ondes3D's computations are generally very small (less than one second) and the expected gain is minimal or even negative, as switching the frequency also comes with a cost (e.g., the interrupt, latency for switching). The possible energy savings are indeed not primarily limited by the number of available frequencies but rather by the possibility to lower the frequency far enough to reduce the idle time to a bare minimum and hence to increase the *energy proportionality*, i.e., the ratio of consumed energy and useful work done by the system.

| Host | Total idle time | % Idle |
|---|---|---|
| taurus-01 | 11.1322 | 3.28 |
| taurus-02 | 72.1573 | 21.24 |
| taurus-03 | 72.89 | 21.45 |
| taurus-04 | 5.26874 | 1.55 |
| taurus-05 | 70.6237 | 20.79 |
| taurus-06 | 162.996 | 47.97 |
| taurus-07 | 163.843 | 48.22 |
| taurus-08 | 73.4741 | 21.63 |
| taurus-09 | 71.621 | 21.08 |
| taurus-10 | 163.661 | 48.17 |
| taurus-11 | 162.987 | 47.97 |
| taurus-12 | 73.3483 | 21.59 |
| taurus-13 | 8.36255 | 2.46 |
| taurus-14 | 72.4478 | 21.32 |
| taurus-15 | 73.077 | 21.51 |
| taurus-16 | 2.2585 | 0.66 |

**Table 9.6:** The ondemand governor does not significantly reduce idle times. Total runtime: 339.75 s. (Chuetsu, 300 iterations, 16 processes)

Processors that are heterogeneous at the core level, i.e., that consist of both weak (but power efficient) and powerful (but energy hungry) cores, are already available (e.g., big.LITTLE). When DVFS slows the processor down, the computation could potentially move from the powerful to the weak core. DVFS could therefore yield larger energy savings on complicated architectures than on standard CPUs, however, the performance prediction is also more complicated as an application that uses features specific to the powerful core will be slowed more significantly than a generic application.

### 9.5.2 Current Limitations of Our Implementation

Ondes3D's main loop consists of three kernels and intertwined communication calls (see Listing 1 (page 109)). Adagio can select a frequency based on the location of the MPI call, as it has access to the call stack. In our implementation of the Lagrange governor (and hence not in the algorithm itself), the frequency is updated essentially twice: once after `Stress()` and once after `Velocity()`, however, our implementation lacks knowledge about the current location and hence cannot adapt the frequency on a per-kernel level.

We performed our evaluation as if the machines had only a single core, as we assume that `Stress()` and `Velocity()` use OpenMP to exploit all cores. Further optimization of the residual load imbalance [Pad+14] is difficult in a multi-core setting, and integrating our Lagrange governor with AMPI for a coordinated DVFS+Load

| Host | Total idle time | % Idle |
|---|---|---|
| taurus-01 | 13.3612 | 3.93 |
| taurus-02 | 79.7384 | 23.47 |
| taurus-03 | 80.5178 | 23.70 |
| taurus-04 | 5.5975 | 1.65 |
| taurus-05 | 78.7834 | 23.19 |
| taurus-06 | 154.92 | 45.60 |
| taurus-07 | 155.961 | 45.90 |
| taurus-08 | 81.03 | 23.85 |
| taurus-09 | 79.3348 | 23.35 |
| taurus-10 | 155.992 | 45.91 |
| taurus-11 | 155.553 | 45.78 |
| taurus-12 | 80.9117 | 23.81 |
| taurus-13 | 9.84613 | 2.90 |
| taurus-14 | 80.0478 | 23.56 |
| taurus-15 | 80.5845 | 23.72 |
| taurus-16 | 2.2568 | 0.66 |

**Table 9.7:** The conservative governor does not significantly reduce idle times and yields for almost all nodes worse results than the ondemand governor, with the exception of `taurus-6,7,10,11`. Total runtime: $339.75\,\mathrm{s}$. (Chuetsu, 300 iterations, 16 processes)

Balancer approach is difficult as it would require to manage ressources at the core level (and not just the node level) because Lagrange is task-based rather than load-based.

## 9.6 Conclusions

Increasing the energy savings through DVFS as a passive technique is a good idea but difficult in an HPC context due to the data dependencies between processes. Indeed, looking only at the load was not sufficient for sizeable energy savings when using Ondes3D. We implemented and tested in total $5$ dynamic governors, but only through an improvement of Adagio and our own rather complicated "Lagrange" governor could we succeed to obtain a $2.5\,\%$ saving.

This needs to be contrasted with the almost $10\,\%$ energy savings and $20\,\%$ smaller makespan when using over-decomposition in combination with a greedy load balancing heuristic.

On current hardware, load balancing is likely to be always strictly superior to DVFS, but this should change once more energy proportionally efficient hardware becomes available. Such a study will be very easy to conduct with SimGrid, as all required facilities (load balancers, governors) are now already available.

| Governor | Makespan | Energy | Energy Savings/Loss (%) |
|---|---|---|---|
| Performance | 339.747 | 989016.128355 | 0.000 |
| Performance + GreedyLB$_5$ | 266.107 | 876433.238571 | -11.383 |
| Performance + GreedyLB$_{10}$ | 267.979 | 879287.679396 | -11.095 |
| Performance + GreedyLB$_{15}$ | 270.227 | 882735.980103 | -10.746 |
| Performance + GreedyLB$_{20}$ | 271.618 | 884867.878757 | -10.530 |
| Performance + GreedyLB$_{25}$ | 273.538 | 887809.278575 | -10.233 |
| Performance + GreedyLB$_{30}$ | 275.039 | 890115.523342 | -10.000 |

**Table 9.8:** Comparison of makespan and energy estimates when load balancing is used with the performance governor. GreedyLB$_k$ means that every $k$ iterations, GreedyLB is called. (Chuetsu, 300 iterations, 64 processes)

| Host | Total idle time | % Idle |
|---|---|---|
| taurus-01 | 3.12365 | 1.166 |
| taurus-02 | 8.45286 | 3.155 |
| taurus-03 | 15.9001 | 5.934 |
| taurus-04 | 17.8564 | 6.664 |
| taurus-05 | 15.2849 | 5.704 |
| taurus-06 | 16.6086 | 6.198 |
| taurus-07 | 14.4727 | 5.401 |
| taurus-08 | 14.8928 | 5.558 |
| taurus-09 | 17.7397 | 6.621 |
| taurus-10 | 29.6589 | 11.069 |
| taurus-11 | 29.6381 | 11.061 |
| taurus-12 | 33.0626 | 12.339 |
| taurus-13 | 18.0793 | 6.747 |
| taurus-14 | 31.0929 | 11.604 |
| taurus-15 | 2.29086 | 0.855 |
| taurus-16 | 7.87874 | 2.940 |

**Table 9.9:** Idle times when using the GreedyLB heuristic every 10 iterations. Total runtime was 267.95 s. (Chuetsu, 300 iterations, 16 processes)

# 10 Conclusion and Future Work

## 10.1 Thesis Summary

MPI-based applications are the most common type of applications that run on current High-Performance Computing systems. In order to deliver more computational power to users, machines have to grow in size (e.g., number of nodes, cores) which will further increase their already significant power consumption, making power consumption one of the most important cost factors for supercomputing centers. The HPC community has for this reason identified energy as the principal optimization goal for applications. In this thesis, we showed how to model, predict and optimize energy consumption of MPI applications running on common multi-core systems by using the SimGrid simulation framework.

In a first step, we proposed a deliberately simple model that computes the power consumption of a particular application, running on a given node with a fixed frequency solely based on the load of the CPU and justified why the application needs to be considered in our model as well. It is possible to instantiate our model using only a single node when all nodes are fully homogeneous, which is often not the case. In fact, even supposedly homogeneous machines can have different power consumptions and in this case, the calibration may have to be executed on a representative sample of nodes.

In a second step, we showed that predicting the energy consumption is tightly associated with accurate runtime predictions. Previous work had mostly worked with SimGrid using single-core hosts only, while we needed actual support for multi-core systems. We identified two main issues for obtained mispredictions: An oversimplified multi-core model in SimGrid as well as no consideration of fast loopback links for intra-node communications. We showed that several issues that come with the use of online simulation, such as cache issues, can be responsible for inaccurate predictions and we provided a calibration strategy that can speed-up or slow-down certain parts of the application during an online simulation.

We furthermore contributed a calibration method for local communications, which proved to be of importance in the case of HPL only when a single node was used.

Locality-aware applications, however, will require a thorough calibration even when multiple nodes are involved.

We validated this approach through real experiments with up to 12 nodes (144 cores) and systematically compared real-life outcomes with predictions. We noted that *executing* the real application is not difficult whereas *controlling* the environment of the experiment is. In order to obtain trustworthy results, we spent considerable time to setup a proper experimental environment. Our performance predictions were eventually within only a few percent of the real experiments and allowed us to validate the accuracy of our energy model by comparing power predictions with data measured on real systems.

Last, this thesis contributed to the optimization of the power consumption of MPI applications. We implemented in SimGrid several DVFS governors from the linux kernel (ondemand, conservative) but also a userspace governor called Adagio [Rou+09] from literature, which has been specifically designed to exploit the regularity of MPI applications. Unfortunately, Adagio and other classical governors did not lead to any significant improvements for the application of our choice, the earthquake simulator Ondes3D, which can be seen as a representative of many other legacy MPI applications. We therefore proposed an improved version of Adagio and our own DVFS governor, based on lagrangian optimization methods. First experiments have shown that the DVFS governors are only able to save very little energy (less than 5 %) whereas computationally more expensive load balancing achieved significant savings.

## 10.2 Limitations

The proposed models, the experimental environments and the applications we studied during the validation are limited to specific use-cases.

### 10.2.1 Model Limitations

Recall from Section 8.5 that our energy model for multi-core CPUs only supports applications with (almost) constant energy consumption. We identified three main cases that are not supported: First, applications that can be divided into phases, each with a distinct power consumption. Second, applications such as *in-situ* applications that run inherently different codes on the same machine. Third, unpredictable combination of kernels, for example by dynamic runtimes, and therefore memory / cache usage that depends on the currently executed codes. In this last case, all combinations of kernels would need to be calibrated. Measuring the impact of

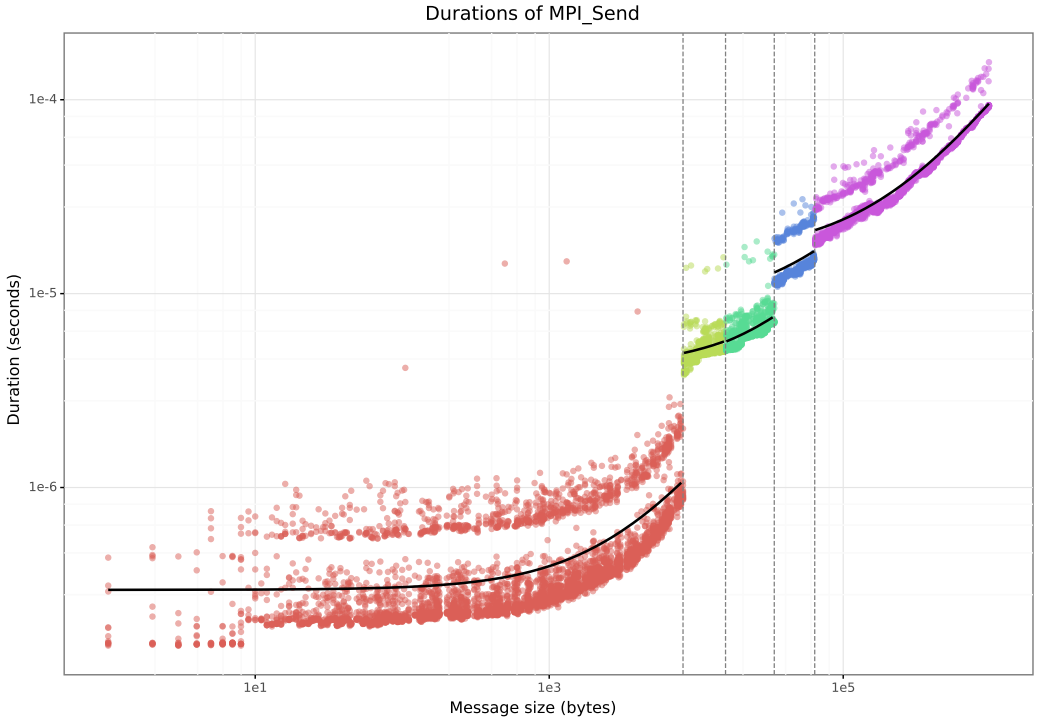


**Figure 10.1:** On the Stampede supercomputer, one "slow" and one "fast" mode seem to exist for `MPI_Recv` and `MPI_Send` (depicted) calls, making faithful simulation very difficult. Figure obtained from Tom Cornebize.

running several kernels on the same CPU requires also more precise power samples than a single value per second as provided by the wattmeter we used.

All of our models (inter- and intra-network communications, energy, multi-core) rely on a calibration procedure that generates the right configuration for use with SimGrid. However, these procedures generally still require manual intervention and execution (e.g., setting the breakpoints when calibrating the network) and the quality of the results is therefore dependent on the expertise of the user. Automation of these procedures is desireable, however, not always trivial: Figure 10.1 illustrates results we obtained when calibrating `MPI_Send` operations on the Stampede supercomputer in a joint-work with Tom Cornebize. On this particular machine, we observed that one "slow" and one "fast" mode exist. The "slow" mode is almost twice as slow and we are still unsure what causes this phenomenon [Cor+17, Section VII]. Clearly, modeling these modes through a single linear regression is questionable. To calibrate such a machine correctly and automatically, we would need to recognize each mode automatically and generate an individual model for each mode. In order to select the right mode during simulation, we would need to find out what this phenomenon depends on and account for it within SimGrid.

We have already stated that machines in the HPC world are becoming increasingly complex and in many cases, they are equipped with GPUs, but also sometimes

with FPGAs and accelerators such as the previously popular Xeon Phi. Using any of these accelerators is currently not supported by our methods and their usage must be investigated separately: modern GPUs, for example, support the execution of several different kernels at the same time and their performance and energy consumption should vary based on the workload. Furthermore, DVFS on GPUs is available in recent GPUs but may induce a different behavior than CPU-based DVFS.

### 10.2.2 Application Limitations

The applications we used ranged from extremely simple toy-benchmarks (NAS-EP) to commonly used benchmarks (HPL) to actual applications (Ondes3D). Running real applications with SimGrid is often difficult, as technical limitations often exist. For example, when an application requires OpenMP (or any paradigm other than MPI), it cannot be simulated as SimGrid has no support for OpenMP. Other applications may require a specific subset of MPI functions that are not yet implemented in SMPI, such as asynchronous collective operations or MPI I/O. Validation of our models with more realistic applications may therefore require to overcome non-trivial technical difficulties.

### 10.2.3 Experimental Limitations

In our experiments, we tried to control the environment as much as possible. On a per-node level, we disabled turbomode and hyperthreading to obtain reproducible and, in the case of our multi-core calibration, comparable results. Recall that this calibration computes speed-up and slow-down factors for each code region based on the aggregated time spent execution each region. These values may not be comparable when turbomode is activated, as SimGrid runs on a single core and turbomode is therefore much more likely to boost the simulation (and hence, the emulated code) than it is to boost the real experiment that runs on all cores simultaneously and that serves as a reference. Furthermore, the energy model would have to be adapted to the frequencies that are available with turbo mode, but not all frequencies are available with each load.

On the network level, we reserved all clusters connected to the used switch and therefore isolated the application. Alas, this is very rarely the case for real applications and cross-traffic is likely to impact the performance and even decisions taken for example by load balancing algorithms.

## 10.3 Future Work

The limited timeframe of this thesis has only allowed us to build the foundation for future energy-focused application analyses. Many interesting questions therefore still remain untouched but are now feasible.

### 10.3.1 Extending the Work of this Thesis

#### Energy Calibration

There are already applications in use that use one or more programming models besides MPI, and they will become even more common in the future [Don+11, p. 10]. As we pointed out in Section 8.5, our energy model may not work in certain scenarios. An application that uses an accelerator only in specific phases, for instance, will see significant changes in energy consumption over time as a GPU can consume up to 300 W under full load. Implementing support for phases is not too difficult (e.g., by using the delimiting MPI calls (possibly through hashing the call stack) as an identifier) and would allow a much larger class of applications to be simulated as well. The difficulty lies in the power measurement tools, which provided at the time of study only a single sample per second but were recently upgraded to 50 samples per second, which may be sufficient.

Our approach also required only a single node of a fully homogeneous machine. In this context, it is interesting to note that future machines may not be fully homogeneous and therefore may require the application to run on each of the node-configurations. A proper, automatic energy calibration will therefore be necessary to make our energy model easy to use and a re-calibration after significant changes simple enough to be adopted even in more complicated cases.

#### DVFS

In Section 9.6, we concluded that load balancing is *currently* the only viable way to reduce energy consumption significantly as DVFS only gained a few percent. This is, however, potentially no longer true once machines with more complicated architectures become widely available. This effect could be studied by modeling for example big.LITTLE processors, where weak (but energy efficient) and powerful (but energy hungry) cores are combined on the same die and the core a computation uses is determined by the experienced load. These processors can already be

modeled in SimGrid by declaring pstates with different meanings: The first $n$ pstates could represent the computational power of the weak cores. Similarly, the next $m$ pstates could represent the powerful cores. The logic that decides which core is used could be implemented similarly to the DVFS governors that already exist in SimGrid.

In our case, we studied only a single load balancer and several DVFS governors. Implementing more load balancing algorithms is not difficult and could be used to study which algorithm is the best approach for a specific class of applications. In this context, particularly energy-aware algorithms are of interest, especially those that do not only move computations to other hosts but also determine the speed at which these computations are executed.

Load balancing will also be important on future machines, however, the cost for synchronizing and moving data around rises significantly with a growing number of processes. A large-scale study could therefore already show under which circumstances current load balancing strategies can still be used and how their effectiveness changes when used on platforms consisting of tens of thousands of nodes.

In our case, we ran Ondes3D only on a few nodes as a proof-of-concept. A real study should use different earthquake scenarios, as they determine the load imbalance, and use a varying number of cores and maybe even various network topologies to test how the regular communication patterns impact the load balance when the underlying topologies do not correspond to the communication calls.

Other applications that are seemingly regular but suffer from load imbalances are also interesting to study. Implementing support for load balancing in simulation is as simple as adding a call to a single function to the code; SimGrid can then be used to predict if implementation of full AMPI support for real executions is worth the effort. Demonstrating significant energy savings and/or performance gains could lead to a faster adoption of load balancing techniques by application developers.

### 10.3.2 SimGrid

#### Placement Policies / Task Mapping

Placement Policies / Task Mapping has been investigated for a long time through theoretical measures. It would be interesting to study the impact of a changed placement on common or complex topologies, such as torus, DragonFly [Kim+08]

or SlimFly [BH14], through simulation that can account for congestion. SimGrid has almost everything in place to support such a study but implementing a new routing algorithm and describing a large platform (with thousands of nodes) is currently rather cumbersome. We implemented a first prototype for Lua-bindings, but this solution was not convincing since the XML API and Lua were separated. A better approach would be to use python-based bindings and use one of the already existing python XML parsers to replace the SimGrid XML parser. Lua is very fast and can be used rather easily with C/C++, but application developers are more likely to already have python bindings, hence effectively lowering the entry barrier. The SimGrid project has just recently started to adopt this approach and we believe that being able to implement the routing (for fast testing) and the platform description programmatically in python will help with large-scale studies.

### Towards Exascale

We have already shown that SimGrid can simulate HPL at large scale [Cor+17]. Our simulation of the supercomputer Stampede required simulation times of approximately $50\,\mathrm{h}$, which can be still faster than to wait for a full machine-reservation. However, simulations of applications using an exascale machine with up to one million nodes [Don+11, p. 62] (and $10^8$ to $10^9$ cores) will become prohibitive, if no further changes are made to how we simulate nodes (and cores). Several approaches are possible and may even be combined: First, the simulation could be run in parallel, spanning several nodes [Mub+12]. This can speed up the simulation but can also help with memory limitations. Unfortunately, SimGrid is a sequential simulator and although there has already been significant effort in this direction [QRT11], this approach is unlikely to scale well. Instead, a time parallel simulation seems promising for HPC applications that tend to synchronize regularly [Fuj15]. Second, the simulation could not run every process individually but rather simulate one process as a representative for a class of processes with (approximatively) identical behavior, similar to what is called "fast simulation" in the mean field framework [BMM07]. This would require either advance knowledge of the application or specific techniques to identify these process classes. Last, the simulation could be reduced in time, space or space and time together. A reduction in time means that the simulation could start at a specific time by loading a previously acquired snapshot of the simulation variables and run only for a specified length or until a specific event occurs. A reduction in space, on the other hand, means that only a subset of the domain is simulated. The communication between simulated and non-simulated processes would need to be substituted, and messages entering the simulated domain would need to be meaningful.

Another interesting aspect is resilience of MPI applications regarding node and link failures. Resilience has already been studied with SimGrid's SimDag module [DCN18], but we are unaware of publications that use emulation to study a real MPI application including the load-imbalances that occur when nodes are delayed through a roll-back [Don16, Section 1.2]. SimGrid already supports mechanisms to change the state of nodes and links (on/off) during a simulation as well as load balancing. It remains to implement a proper resilience algorithm or to identify an application that has resilience built-in.

### 10.3.3 Joint Work with the SimGrid Userbase

A significant number of studies that are conducted with SimGrid either use benchmarks similar to the ones used in this thesis (HPL, NAS-LU and NAS-EP) or rely on small proxy-applications. Studies using real-world applications are still relatively scarce, even though some high-profile applications (such as BigDFT) exist that closely integrate with SimGrid to predict their performance.

Extending the userbase by working closely with select developers of such applications would have several benefits: First and foremost, technical and scientific issues that were previously not considered almost certainly need to be resolved, but the exact type depends on the simulated application and the target platform. One such issue could be the fast and faithful simulation of BLAS routines, as these functions often make up a large part of the simulation. In our study of HPL at large scale [Cor+17], we replaced several BLAS routines with linear models. Building a complete *libsimblas* for use with SimGrid would be interesting but analyzing and modeling all functions provided by the BLAS API is non-trivial. Second, real-life applications that can be predicted accurately with SimGrid can also be used to validate new scientific concepts introduced to SimGrid. A validation using real applications can be seen as more trustworthy. A third benefit of a growing userbase could lie in the contributions back to the SimGrid project, for instance through bug reports, patches (code and documentation), tutorials at workshops or even new features. Finally, funding for continuous development of SimGrid could be easier to procur when more reputable applications rely on SimGrid's predictions.

# Bibliography


[Acu+14] Bilge Acun, Abhishek Gupta, Nikhil Jain, et al. "Parallel Programming with Migratable Objects: Charm++ in Practice". In: *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Nov. 2014 (cit. on pp. 61, 117).

[All87] David W. Allan. "Time and Frequency (Time-Domain) Characterization, Estimation, and Prediction of Precision Clocks and Oscillators". In: *IEEE Trans. on Ultrasonics, Ferroelectrics, and Frequency Control* 34.6 (Nov. 1987) (cit. on p. 65).

[Ans+14] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, et al. "OpenTuner". In: *Proceedings of the 23rd international conference on Parallel architectures and compilation - PACT '14*. ACM Press, 2014 (cit. on p. 31).

[Aug+10] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures". In: *Concurrency and Computation: Practice and Experience* 23.2 (Nov. 2010), pp. 187–198 (cit. on pp. 30, 58, 61).

[Bad+03] Rosa M. Badia, Jesús Labarta, Judit Giménez, and Francesc Escalé. "Dimemas: Predicting MPI Applications Behaviour in Grid Environments". In: *Proc. of the Workshop on Grid Applications and Programming Tools*. June 2003 (cit. on p. 34).

[Bal+13] Daniel Balouek, Alexandra Carpen-Amarie, Ghislain Charrier, et al. "Adding Virtualization Capabilities to the Grid'5000 Testbed". In: *Cloud Computing and Services Science*. Ed. by IvanI. Ivanov, Marten Sinderen, Frank Leymann, and Tony Shan. Vol. 367. Communications in Computer and Information Science. Springer International Publishing, 2013 (cit. on p. 63).

[Bed+13] Paul Bedaride, Augustin Degomme, Stéphane Genaud, et al. "Toward Better Simulation of MPI Applications on Ethernet/TCP Networks". In: *Proc. of the 4th Intl. Workshop on Performance Modeling, Benchmarking and Simulation*. Vol. 8551. LNCS. Denver, CO: Springer, Nov. 2013 (cit. on p. 50).

[Bes+18] Maciej Besta, Syed Minhaj Hassan, Sudhakar Yalamanchili, et al. "Slim NoC". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '18*. ACM Press, 2018 (cit. on p. 30).

[BH14] Maciej Besta and Torsten Hoefler. "Slim Fly: A Cost Effective Low-Diameter Network Topology". In: *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Nov. 2014 (cit. on pp. 29, 139).

[Bie+15] Mario Bielert, Florina M. Ciorba, Kim Feldhoff, Thomas Ilsche, and Wolfgang E. Nagel. "HAEC-SIM: A Simulation Framework for Highly Adaptive Energy-efficient Computing Platforms". In: *Proceedings of the 8th International Conference on Simulation Tools and Techniques (SIMUTools)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2015, pp. 129–138 (cit. on p. 34).

[Bla+13] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. "Post-failure recovery of MPI communication capability". In: *The International Journal of High Performance Computing Applications* 27.3 (June 2013), pp. 244–254 (cit. on p. 22).

[BMM07] Jean-Yves Le Boudec, David McDonald, and Jochen Mundinger. "A Generic Mean Field Convergence Result for Systems of Interacting Objects". In: *Fourth International Conference on the Quantitative Evaluation of Systems (QEST 2007)*. IEEE, Sept. 2007 (cit. on p. 139).

[Bob+12] Laurent Bobelin, Arnaud Legrand, David Alejandro González Márquez, et al. "Scalable Multi-Purpose Network Representation for Large Scale Distributed System Simulation". In: *Proc. of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. Ottawa, Canada, 2012 (cit. on p. 86).

[Bos+13] George Bosilca, Aurelien Bouteiller, Anthony Danalis, et al. "PaRSEC: Exploiting Heterogeneity to Enhance Scalability". In: *Computing in Science & Engineering* 15.6 (Nov. 2013), pp. 36–45 (cit. on p. 30).

[Boy10] Stephen Boyd. "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers". In: *Foundations and Trends® in Machine Learning* 3.1 (2010), pp. 1–122 (cit. on pp. 122, 123).

[Cal+11] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, Cesar A. F. De Rose, and Rajkumar Buyya. "CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms". In: *Software: Practice and Experience* 41.1 (Jan. 2011) (cit. on p. 33).

[Cap+14] Franck Cappello, Al Geist, William Gropp, et al. "Toward Exascale Resilience: 2014 update". In: *Supercomputing Frontiers and Innovations* 1.1 (2014) (cit. on pp. 21, 22).

[Cas+14] Henri Casanova, Arnaud Giersch, Arnaud Legrand, Martin Quinson, and Frédéric Suter. "Versatile, scalable, and accurate simulation of distributed applications and platforms". In: *Journal of Parallel and Distributed Computing* 74.10 (Oct. 2014), pp. 2899–2917 (cit. on pp. 38, 39).

[CLR08] Henri Casanova, Arnaud Legrand, and Yves Robert. *Parallel Algorithms*. CRC Press, 2008 (cit. on p. 14).

[Cor+17] Tom Cornebize, Franz C Heinrich, Arnaud Legrand, and Jérôme Vienne. "Emulating High Performance Linpack on a Commodity Server at the Scale of a Supercomputer". working paper or preprint. Dec. 2017 (cit. on pp. 102, 106, 135, 139, 140).

[CPJ11] Patrick Carribault, Marc Pérache, and Hervé Jourdren. "Thread-Local Storage Extension to Support Thread-Based MPI/OpenMP Applications". In: *OpenMP in the Petascale Era*. Springer Berlin Heidelberg, 2011, pp. 80–93 (cit. on p. 50).

[Cza+12] Tomasz S. Czajkowski, Utku Aydonat, Dmitry Denisenko, et al. “From opencl to high-performance hardware on FPGAS”. In: *22nd International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Aug. 2012 (cit. on p. 28).

[Dav+12] J. D. Davis, S. Rivoire, M. Goldszmidt, and E. K. Ardestani. “Including Variability in Large-Scale Cluster Power Models”. In: *IEEE Computer Architecture Letters* 11.2 (2012) (cit. on p. 65).

[Day+09] M S Day, J B Bell, R K Cheng, et al. “Cellular burning in lean premixed turbulent hydrogen-air flames: Coupling experimental and computational analysis at the laboratory scale”. In: *Journal of Physics: Conference Series* 180 (July 2009), p. 012031 (cit. on p. 21).

[DCN18] Kiril Dichev, Kirk Cameron, and Dimitrios S. Nikolopoulos. “Energy-efficient localised rollback via data flow analysis and frequency scaling”. In: *Proceedings of the 25th European MPI Users’ Group Meeting on - EuroMPI’18*. ACM Press, 2018 (cit. on p. 140).

[Deg+17] Augustin Degomme, Arnaud Legrand, George S. Markomanolis, et al. “Simulating MPI Applications: The SMPI Approach”. In: *IEEE Transactions on Parallel and Distributed Systems* 28.8 (Aug. 2017), pp. 2387–2400 (cit. on pp. 37, 46, 53, 54, 72).

[Don+11] Jack Dongarra, Pete Beckman, Terry Moore, et al. “The International Exascale Software Project roadmap”. In: *The International Journal of High Performance Computing Applications* 25.1 (Jan. 2011), pp. 3–60 (cit. on pp. 2, 19, 20, 23, 30, 60, 61, 137, 139).

[Don+17] Jack Dongarra, Stanimire Tomov, Piotr Luszczek, et al. “With Extreme Computing, the Rules Have Changed”. In: *Computing in Science & Engineering* 19.3 (May 2017), pp. 52–62 (cit. on pp. 26, 31).

[Don16] Jack Dongarra. “With Extreme Scale Computing the Rules Have Changed”. In: *Mathematical Software – ICMS 2016*. Springer International Publishing, 2016, pp. 3–6 (cit. on pp. 19, 30, 140).

[Dut+16a] Pierre-François Dutot, Michael Mercier, Millian Poquet, and Olivier Richard. “Batsim: a Realistic Language-Independent Resources and Jobs Management Systems Simulator”. In: *20th Workshop on Job Scheduling Strategies for Parallel Processing*. Chicago, United States, May 2016 (cit. on pp. 54, 60).

[Dut+16b] Pierre-François Dutot, Michael Mercier, Millian Poquet, and Olivier Richard. “Batsim: a Realistic Language-Independent Resources and Jobs Management Systems Simulator”. In: *20th Workshop on Job Scheduling Strategies for Parallel Processing*. May 2016 (cit. on p. 106).

[DWF16] Miyuru Dayarathna, Yonggang Wen, and Rui Fan. “Data Center Energy Consumption Modeling: A Survey”. In: *IEEE Communications Surveys & Tutorials* 18.1 (2016), pp. 732–794 (cit. on p. 90).

[Eng14] Christian Engelmann. “Scaling To A Million Cores And Beyond: Using Light-Weight Simulation to Understand The Challenges Ahead On The Road To Exascale”. In: *Future Generation Computer Systems* 30 (Jan. 2014) (cit. on p. 34).

[Fuj15] Richard Fujimoto. “Parallel and distributed simulation”. In: *2015 Winter Simulation Conference (WSC)*. IEEE, Dec. 2015 (cit. on p. 139).

[Gab+04] Edgar Gabriel, Graham E. Fagg, George Bosilca, et al. “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer Berlin Heidelberg, 2004, pp. 97–104 (cit. on pp. 15, 48).

[Gen+08] Luigi Genovese, Alexey Neelov, Stefan Goedecker, et al. “Daubechies wavelets as a basis set for density functional pseudopotential calculations”. In: *The Journal of Chemical Physics* 129.1 (July 2008), p. 014109 (cit. on p. 74).

[Geo+15] Yiannis Georgiou, David Glesser, Krzysztof Rzadca, and Denis Trystram. “A Scheduler-Level Incentive Mechanism for Energy Efficiency in HPC”. In: *Proc. of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. Shenzhen, China, May 2015 (cit. on p. 106).

[Gro02] William Gropp. “MPICH2: A New Start for MPI Implementations”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer Berlin Heidelberg, 2002, pp. 7–7 (cit. on pp. 15, 48).

[Gué+13] Tom Guérout, Thierry Monteil, Georges Da Costa, et al. “Energy-aware simulation with DVFS”. In: *Simulation Modelling Practice and Theory* 39 (Dec. 2013) (cit. on p. 33).

[Gue+19] Loic Guegan, Betsegaw Lemma Amersho, Anne-Cécile Orgerie, and Martin Quinson. “A Large-Scale Wired Network Energy Model for Flow-Level Simulations”. In: *AINA 2019 - 33rd International Conference on Advanced Information Networking and Applications*. Matsue, Japan, Mar. 2019, pp. 1–12 (cit. on p. 105).

[Hac+13] Daniel Hackenberg, Roland Oldenburg, Daniel Molka, and Robert Schone. “Introducing FIRESTARTER: A processor stress test utility”. In: *2013 International Green Computing Conference Proceedings*. IEEE, June 2013 (cit. on p. 68).

[Hei+17a] Franz C. Heinrich, Alexandra Carpen-Amarie, Augustin Degomme, et al. “Predicting the Performance and the Power Consumption of MPI Applications With SimGrid”. working paper or preprint. Jan. 2017 (cit. on pp. 63, 89).

[Hei+17b] Franz Christian Heinrich, Tom Cornebize, Augustin Degomme, et al. “Predicting the Energy-Consumption of MPI Applications at Scale Using Only a Single Node”. In: *2017 IEEE International Conference on Cluster Computing, CLUSTER 2017, Honolulu, HI, USA, September 5-8, 2017*. IEEE Computer Society, 2017, pp. 92–102 (cit. on pp. 33, 73, 83, 89).

[Heu+11] Martin Heusse, Sears A. Merritt, Timothy X. Brown, and Andrzej Duda. “Two-way TCP connections”. In: *ACM SIGCOMM Computer Communication Review* 41.2 (Apr. 2011), p. 5 (cit. on p. 43).

[HSL10] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. “LogGOPSim: Simulating Large-scale Applications in the LogGOPS Model”. In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. Chicago, Illinois: ACM, 2010, pp. 597–604 (cit. on p. 34).

[Ina+15] Yuichi Inadomi, Tapasya Patki, Koji Inoue, et al. “Analyzing and Mitigating the Impact of Manufacturing Variability in Power-constrained Supercomputing”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’15. Austin, Texas: ACM, 2015 (cit. on pp. 65, 95).

[Jac88] V. Jacobson. “Congestion avoidance and control”. In: *Symposium proceedings on Communications architectures and protocols - SIGCOMM ’88*. ACM Press, 1988 (cit. on p. 121).

[Jan+10] Curtis L. Janssen, Helgi Adalsteinsson, Scott Cranford, et al. “A Simulator for Large-scale Parallel Architectures”. In: *International Journal of Parallel and Distributed Systems* 1.2 (2010) (cit. on pp. 34, 80).

[KBK12] Dzmitry Kliazovich, Pascal Bouvry, and Samee U. Khan. “A packet-level simulator of energy-aware cloud comuting data centers”. In: *Journal of Supercomputing* 62.3 (2012) (cit. on p. 33).

[Kel18] Rafael Keller Tesser. “A Simulation Workflow to Evaluate the Performance of Dynamic Load Balancing with Over-decomposition for Iterative Parallel Applications”. Theses. Universidade Federal Do Rio Grande Do Sul, Apr. 2018 (cit. on p. 61).

[Kim+08] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. “Technology-Driven, Highly-Scalable Dragonfly Topology”. In: *2008 International Symposium on Computer Architecture*. IEEE, June 2008 (cit. on pp. 29, 138).

[KK93] Laxmikant V. Kale and Sanjeev Krishnan. “CHARM++”. In: *ACM SIGPLAN Notices* 28.10 (Oct. 1993), pp. 91–108 (cit. on pp. 30, 117).

[Koi+12] Michihiro Koibuchi, Hiroki Matsutani, Hideharu Amano, D. Frank Hsu, and Henri Casanova. “A case for random shortcut topologies for HPC interconnects”. In: *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, June 2012 (cit. on p. 29).

[Koi+13] M. Koibuchi, I. Fujiwara, H. Matsutani, and H. Casanova. “Layout-conscious random topologies for HPC off-chip interconnects”. In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Feb. 2013 (cit. on p. 29).

[Leb+15] Adrien Lebre, Arnaud Legrand, Frederic Suter, and Pierre Veyre. “Adding Storage Simulation Capacities to the SimGrid Toolkit: Concepts, Models, and API”. In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, May 2015 (cit. on p. 37).

[Loz+16] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, et al. “The Linux scheduler”. In: *Proceedings of the Eleventh European Conference on Computer Systems - EuroSys ’16*. ACM Press, 2016 (cit. on p. 68).

[LWP04] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K. Panda. “High Performance RDMA-Based MPI Implementation over InfiniBand”. In: *International Journal of Parallel Programming* 32.3 (June 2004), pp. 167–198 (cit. on pp. 15, 48).

[Mar+07] Gustavo Marfia, Claudio Palazzi, Giovanni Pau, et al. “TCP Libra: Exploring RTT-Fairness for TCP”. In: *NETWORKING 2007. Ad Hoc and Sensor Networks, Wireless Networks, Next Generation Internet*. Springer Berlin Heidelberg, 2007, pp. 1005–1013 (cit. on p. 43).

[Mar14] Georgios Markomanolis. “Performance Evaluation and Prediction of Parallel Applications”. Theses. Ecole normale supérieure de lyon - ENS LYON, Jan. 2014 (cit. on p. 82).

[MDV11] V. Moureau, P. Domingo, and L. Vervisch. “From Large-Eddy Simulation to Direct Numerical Simulation of a lean premixed swirl flame: Filtered laminar flame-PDF modeling”. In: *Combustion and Flame* 158.7 (July 2011), pp. 1340–1357 (cit. on p. 20).

[MN15] Shinobu Miwa and Hiroshi Nakamura. “Profile-based power shifting in interconnection networks with on/off links”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015*. Ed. by Jackie Kern and Jeffrey S. Vetter. ACM, 2015, 37:1–37:11 (cit. on p. 70).

[Mub+12] Misbah Mubarak, Christopher D. Carothers, Robert Ross, and Philip Carns. “Modeling a Million-Node Dragonfly Network Using Massively Parallel Discrete-Event Simulation”. In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, Nov. 2012 (cit. on p. 139).

[Mub+17] M. Mubarak, C. D. Carothers, Robert B. Ross, and Philip H. Carns. “Enabling Parallel Simulation of Large-Scale HPC Network Systems”. In: *IEEE Transactions on Parallel and Distributed Systems* 28.1 (Jan. 2017), pp. 87–100 (cit. on p. 34).

[Myt+09] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter 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 2009, Washington, DC, USA, March 7-11, 2009*. Ed. by Mary Lou Soffa and Mary Jane Irwin. ACM, 2009, pp. 265–276 (cit. on pp. 65, 67).

[Noe+09] Michael Noeth, Prasun Ratn, Frank Mueller, Martin Schulz, and Bronis R. de Supinski. “ScalaTrace: Scalable compression and replay of communication traces for high-performance computing”. In: *Journal of Parallel and Distributed Computing* 69.8 (Aug. 2009), pp. 696–710 (cit. on p. 82).

[Now+15] T. Nowatzki, J. Menon, C. H. Ho, and K. Sankaralingam. “Architectural Simulators Considered Harmful”. In: *IEEE Micro* 35.6 (Nov. 2015) (cit. on p. 33).

[ODL14] Anne-Cécile Orgerie, Marcos Dias de Assunção, and Laurent Lefèvre. “A Survey on Techniques for Improving the Energy Efficiency of Large-Scale Distributed Systems”. In: *ACM Computing Surveys (CSUR)* 46.4 (2014) (cit. on pp. 89–91).

[OPF10] Simon Ostermann, Radu Prodan, and Thomas Fahringer. “Dynamic Cloud Provisioning for Scientific Grid Workflows”. In: *Proc. of the 11th ACM/IEEE Intl. Conf. on Grid Computing (Grid)*. Oct. 2010 (cit. on p. 33).

[Pad+14] Edson L. Padoin, Marcio Castro, Laercio L. Pilla, Philippe O. A. Navaux, and Jean-Francois Mehaut. “Saving energy by exploiting residual imbalances on iterative applications”. In: *2014 21st International Conference on High Performance Computing (HiPC)*. IEEE, Dec. 2014 (cit. on pp. 119, 129, 130).

[QRT11] Martin Quinson, Cristian Rosa, and Christophe Thiery. “Parallel Simulation of Peer-to-Peer Systems”. In: *CCGrid 2012 – The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. CCGRID ’12 Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. Ottawa, Canada: IEEE, May 2011, pp. 668–675 (cit. on p. 139).

[R C16] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2016 (cit. on pp. 71, 72).

[Raj+16] Nikola Rajovic, Alejandro Rico, Filippo Mantovani, et al. "The Mont-Blanc Prototype: An Alternative Approach for HPC Systems". In: *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, Nov. 2016 (cit. on p. 25).

[Ren12] Paul Renaud-Goud. "Energy-aware scheduling : complexity and algorithms". Theses. Ecole normale supérieure de lyon - ENS LYON, July 2012 (cit. on p. 89).

[Rou+09] Barry Rountree, David K. Lownenthal, Bronis R. de Supinski, et al. "Adagio". In: *Proceedings of the 23rd international conference on Conference on Supercomputing - ICS '09*. ACM Press, 2009 (cit. on pp. 2, 115, 116, 119, 120, 129, 134).

[Sch+12] Eric Schulte, Dan Davison, Thomas Dye, and Carsten Dominik. "A Multi-Language Computing Environment for Literate Programming and Reproducible Research". In: *Journal of Statistical Software* 46.3 (2012) (cit. on p. 71).

[Sho+17] Hayk Shoukourian, Torsten Wilde, Detlef Labrenz, and Arndt Bode. "Using Machine Learning for Data Center Cooling Infrastructure Efficiency Prediction". In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, May 2017 (cit. on pp. 1, 3, 33, 89).

[Sna+02] Allan Snavely, Laura Carrington, Nicole Wolter, et al. "A Framework for Performance Modeling and Prediction". In: *Proc. of the ACM/IEEE Conference on Supercomputing*. Baltimore, MA, Nov. 2002 (cit. on p. 34).

[Sta15] Luka Stanisic. "A Reproducible Research Methodology for Designing and Conducting Faithful Simulations of Dynamic HPC Applications". 2015GREAM035. PhD thesis. 2015 (cit. on p. 58).

[Ter+10] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. "Collecting Performance Data with PAPI-C". In: *Tools for High Performance Computing 2009*. Springer Berlin Heidelberg, 2010, pp. 157–173 (cit. on p. 59).

[Tes+18] Rafael Keller Tesser, Lucas Mello Schnorr, Arnaud Legrand, et al. "Performance modeling of a geophysics application to accelerate over-decomposition parameter tuning through simulation". In: *Concurrency and Computation: Practice and Experience* (Oct. 2018), e5012 (cit. on pp. 61, 108, 109, 118, 126).

[Tig+12] Michael Tighe, Gaston Keller, Michael Bauer, and Hanan Lutfiyya. "DCSim: a data centre simulation tool for evaluating dynamic virtualized resource management". In: *Int. Conf. on Network and Service Management*. 2012 (cit. on p. 33).

[TOR+17] Sunao TORII, Hitoshi ISHIKAWA, Yasuyuki KIMURA, and Motoaki SAITOH. "Technologies and Future Prospects of Green Supercomputer ZettaScaler". Japanese. In: *C* 100.11 (2017). Document in Japanese, pp. 537–544 (cit. on pp. 26, 27).

[Val90] Leslie G. Valiant. "A bridging model for parallel computation". In: *Communications of the ACM* 33.8 (Aug. 1990), pp. 103–111 (cit. on p. 107).

[Vel+13a] Pedro Velho, Lucas Mello Schnorr, Henri Casanova, and Arnaud Legrand. "On the Validity of Flow-level TCP Network Models for Grid and Cloud Simulations". In: *ACM Trans. Model. Comput. Simul.* 23.4 (Dec. 2013) (cit. on p. 33).

[Vel+13b] Pedro Velho, Lucas Schnorr, Henri Casanova, and Arnaud Legrand. "On the Validity of Flow-level TCP Network Models for Grid and Cloud Simulations". In: *ACM Transactions on Modeling and Computer Simulation* 23.4 (Oct. 2013) (cit. on pp. 43, 53).

[Vie10] Jérôme Vienne. "Prédiction de performances d'applications de calcul haute performance sur réseau Infiniband". Theses. Université de Grenoble, July 2010 (cit. on p. 53).

[Wil18] Torsten Wilde. "Assessing the Energy Efficiency of High Performance Computing (HPC) Data Centers". PhD thesis. Technical University Munich, Germany, 2018 (cit. on p. 33).

[WPD01] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. "Automated empirical optimizations of software and the ATLAS project". In: *Parallel Computing* 27.1-2 (Jan. 2001), pp. 3–35 (cit. on p. 31).

[Yas+19] Ryota Yasudo, Michihiro Koibuchi, Koji Nakano, Hiroki Matsutani, and Hideharu Amano. "Designing High-Performance Interconnection Networks with Host-Switch Graphs". In: *IEEE Transactions on Parallel and Distributed Systems* 30.2 (Feb. 2019), pp. 315–330 (cit. on p. 37).

[ZKK04] Gengbin Zheng, Gunavardhan Kakulapati, and Laxmikant Kale. "BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines". In: *Proc. of the 18th IPDPS*. 2004 (cit. on p. 34).

## Webpages

[DEL] DELL. *Dell OpenManage Deployment Toolkit Version 4.4 Command Line Interface Reference Guide*. URL: https://www.dell.com/support/manuals/fr/fr/frbsdt1/dell-opnmang-dplymnt-toolkit-v4.4/dtk_cli-v3/-memopmode-memoperatingmode (visited on Mar. 16, 2019) (cit. on p. 66).

[Ead16] Doug Eadline. *Network Co-design as a Gateway to Exascale*. Sept. 2016. URL: https://insidehpc.com/2016/09/network-co-design-as-a-gateway-to-exascale (visited on Jan. 25, 2019) (cit. on p. 23).

[For] MPI Forum. *MPI Forum*. URL: https://www.mpi-forum.org/ (visited on Apr. 2, 2019) (cit. on p. 15).

[Huf18] Jennifer Huffstetler. *Intel Processors and FPGAs - Better Together*. May 2018. URL: https://itpeernetwork.intel.com/intel-processors-fpga-better-together/ (visited on Mar. 16, 2019) (cit. on p. 28).

[Inc] The Khronos Group Inc. *OpenCL Overview - The Khronos Group Inc*. URL: https://www.khronos.org/opencl/ (visited on Apr. 2, 2019) (cit. on p. 11).

[Kal17] Kalray. *Kalray announces the release of its third-generation MPPA® processor "Coolidge"*. May 2017. URL: https://www.kalray.eu/release-of-third-generation-mppa-processor-coolidge/ (visited on Mar. 16, 2019) (cit. on p. 27).

[Kid08] Taylor Kidd. Mar. 2008. URL: https://software.intel.com/en-us/blogs/2008/03/27/update-c-states-c-states-and-even-more-c-states/ (visited on Jan. 30, 2019) (cit. on p. 91).

[NVI] NVIDIA. *CUDA Zone | NVIDIA Developer*. URL: https://developer.nvidia.com/cuda-zone (visited on Apr. 2, 2019) (cit. on p. 11).

[Off17] German Federal Statistic Office. 2017. URL: https://www.destatis.de/DE/ZahlenFakten/GesamtwirtschaftUmwelt/Umwelt/MaterialEnergiefluesse/Tabellen/StromverbrauchHaushalte.html (visited on Mar. 16, 2019) (cit. on p. 23).

[Rad] Netherlands Institute for Radio Astronomy. *UniBoard I and II*. URL: https://www.astron.nl/r-d-laboratory/uniboard/uniboard-i-and-ii (visited on Mar. 16, 2019) (cit. on p. 27).

[Tea] The SimGrid Team. *Publications*. URL: https://simgrid.org/Publications.html (visited on Apr. 1, 2019) (cit. on pp. 37, 38).

[Tea12] The Grid'5000 Team. *File:Lyon net.png - Grid5000*. Nov. 2012. URL: https://www.grid5000.fr/w/File:Lyon_net.png (visited on Apr. 1, 2019) (cit. on p. 66).

[Tea19] The SimGrid Team. *SimGrid [Jenkins]*. Feb. 2019. URL: https://ci.inria.fr/simgrid/job/SimGrid/ (visited on Feb. 8, 2019) (cit. on p. 41).

# List of Figures


1.1 The energy bill for data centers has drastically increased since the early 2000s, making energy savings an important goal for hardware vendors and application developers. This statistic shows the increase of the Germany-based Leibniz-Rechenzentrum [Sho+17, Figure 1]. . . . . 3

2.1 A domain decomposition of rocky ground used for an earthquake simulation with Ondes3D. The physical domain is cut into cuboids which are evenly assigned to processors. . . . . 6

2.2 A visual representation of the cache hierarchy and sizes of a workstation laptop as obtained through hwloc-ls. . . . . 9

2.3 A visual representation of the cache hierarchy and sizes of a node as obtained through hwloc-ls. . . . . 10

2.4 Several classic network topologies. Shown are: (a) fully connected network, (b) ring, (c) mesh, (d) torus, (e) hypercube and (f) fat-tree [CLR08, Figure 3.1]. . . . . 14

2.5 An illustration of (simulated) cellular flames where temperatures differ locally due to differences in burning velocity. The right picture highlights that parts of the flame can have highly irregular shapes and can be detached from the rest [Day+09]. . . . . 21

2.6 Development of the Top500 list over time and extrapolation for the near future. From `www.top500.org` . . . . . 24

2.7 Visualization of nodes allocated to a job running on BlueWaters 3D-torus topology. Obtained through private communication from Greg Bauer. . . . . 29

4.1 Impact of the SimGrid project per year, measured by the amount of citations (red), papers detailing new components/modules of the SimGrid project (blue) and scientific work that uses SimGrid as a tool for research (green) [Tea]. . . . . 38

4.2 Overview over legacy SimGrid components. . . . . 39

4.3 Highlevel overview over SimGrid 4. Only the new S4U layer can communicate with the kernel. . . . . 39

4.4 Overview of used platforms and essential configurations for building and testing SimGrid as used on 2019-02-08 [Tea19]. . . . . 41

4.5 SimGrid's models simplify real-life network usage by ignoring the bandwidth adjustment phase in the beginning and when a new message enters the network, it reduces the bandwidth immediately, without going through another adjustment phase. . . . . 44
4.6 A simple platform, taken from SimGrid's collection of examples, called `onelink.xml`. Two hosts with 1 Gflop/s and one link with 25 ms latency and 1 GB/s are declared. This link is furthermore used to connect both hosts. . . . . 45
4.7 Faithful prediction of MPI applications requires to account for message-size dependent protocol and mode changes. By calibrating our cluster through a series of experiments [Deg+17], we determined that on this particular cluster and MPI implementation five main modes (each colored differently) with sometimes significant differences exist. . . . . 54
4.8 The (shortened) calibration output obtained for the Grid5000 taurus cluster that serves as input to SimGrid. Note that the last value of `smpi/or` and `smpi/os` shows values that are 0: These are correct, since these messages will be sent synchronously (as defined `smpi/send-is-detached-thres`) and the entire cost is hence already accounted for. . . . . 55
4.9 An example of a Time Independent Trace for a single rank. The first column denotes the rank id and the second column the action, each with their own parameters (e.g., flop for the `compute` action or receiver, tag, message size and data type for the `isend` operation) . . . . . 55

5.1 Factors that can have an impact on the performance of an HPC application. . . . . 64
5.2 In Lyon, several cluster are connected to the same switch, which is why we reserved all of them except for the service networks and machines (due to lack of permissions) [Tea12]. . . . . 66
5.3 Different load and different core configurations can change the energy consumption significantly. In the beginning, 12 cores experience 80 % load, the second plateau represents 100 % load. For the third measurement, the CPU's turbo mode was enabled and for the last measurement, hyperthread was enabled as well. Note that the drops to around 100 W are short breaks in between the measurements and the system was idle. These tests were executed with FIRESTARTER [Hac+13] on 2018/08/13 from 11:15:00am to 11:20:00am on `taurus-6`. . . . . 68

6.1 Excerpt of the NAS LU-PB (`exchange_1.f`) highlighting code regions between any two MPI calls. . . . . 77
6.2 Trace merging process used for the NAS-LU benchmark to compute region-based speedup/slowdown factors and correct the simulation. . 78

6.3 Comparison of calibrated (blue) and uncalibrated (green) runs of LU with real experiments (red) and ideal scaling (grey). . . . . 79

7.1 Intra-node communications can rely on different protocols than inter-node communications. To calibrate the loopback link, we executed the same calibration procedure as for inter-node communications (see Figure 4.7 on page 54 ). As can be seen, there is signifcantly less jitter for local communications than for inter-node communications. We furthermore found that small messages were sent *faster* over network links than shared memory since the send operation was executed asynchronously for remote destinations. For large messages, the loopback was almost an order of magnitude faster which is expected due to the much faster bandwidth. . . . . 84

7.2 HPL does not exploit locality and therefore, only the single-node execution is largely overestimated when the loopback link remains configured with the same speed as the more than four times slower network. . . . . 86

7.3 When executed on only one node, HPL's performance is influenced by loopback bandwidth: A thousandfold increase to $5.12\,\mathrm{TB/s}$ reduces the runtime by almost $5\,\mathrm{s}$ (from $76.64\,\mathrm{s}$ to $71.61\,\mathrm{s}$) whereas lowering the bandwidth to around two-third ($3.2\,\mathrm{GB/s}$) causes the runtime to be overestimated (with a total of $77.8\,\mathrm{s}$). When executed on more than one node, local communication becomes less important and hence the impact of these settings is not noticeable. . . . . 87

8.1 We measured idle power on every machine (here: `taurus-1`) for ten minutes per frequency and active cores. These measurements were repeated three times and no-earlier than 3 weeks after the previous measurement. Note that the y-axis begins at $85\,\mathrm{W}$. The variation per core-count is therefore minimal and on the order of about $1\,\mathrm{W}$. . . . . 92

8.2 Changing the frequency while keeping the load constant causes the consumed energy to grow quadratically. The energy that is required to keep an idling node on (despite all energy-efficiency measures, such as C-states), is additionally shown here as $P_{idle}$. . . . . 93

8.3 Varying the number of cores while keeping the workload constant (NAS-EP, class C) reveals a linear connection between load and power consumption. Note that not all frequencies are shown in this figure to reduce overplotting. The energy that is required to keep an idling node on (despite all energy-efficiency measures, such as C-states), is shown here as $P_{idle}$. . . . . 94

8.4 Power consumption over time when running NAS-EP, NAS-LU, HPL or idling (with 12 active cores and the frequency set to $2300\,\mathrm{MHz}$). . . . 95

8.5 Comparison of two idle power measurements. The first one was executed in 2014 by Alexandra Carpen-Amarie and Sascha Hunoldt, the one in 2016 by me. The frequency was fixed at 2300 MHz for the used nodes. In this plot, the y-axis starts at around 90 W s to highlight the differences, especially for taurus-12. . . . . 96

8.6 Power consumption for a single node when polling via MPI_Iprobe. The distinct growing phases represent one frequency, ranging from 2300 MHz to 1300 MHz, and 1 to 12 cores. When the second CPU gets activated, a jump is recognizable, especially for the highest frequency. . 97

8.7 A sample configuration for the `taurus-8` node. The configured workload here is the EP benchmark and corresponds to the linear regression of Figure 8.3 . For each frequency, three values are provided: the idle power $P_{idle}$, the power consumption when the workload is executed on a single core and the power consumption when the workload is executed on all cores. SimGrid allows the user to configure the energy that the node consumes when it is turned off through the "watt_off" option. . . . . 98

8.8 The validity of this model was tested with three popular benchmarks: NAS-EP, NAS-LU, and HPL. The taurus cluster was used with up to 12 nodes and 12 processes per node. . . . . 100

8.9 Comparison of predicted energy usage of HPL with and without accounting for the additional energy consumption of MPI_Iprobe calls. . 101

8.10 Comparison of predicted energy usage of NAS-EP when using only one or an individual power model for all simulated nodes. . . . . 101

8.11 Time- and energy-to-solution extrapolated for two different matrix sizes with up to $256 \times 12 = 3,072$ MPI processes, interconnected by a fat-tree topology. Once a threshold is reached, adding more nodes does not yield faster performance but only increased energy consumption. . 103

9.1 Illustration of the spatial load imbalance encountered during the first iteration of a run with $16 \times 16 = 256$ processes on the Ligurian workload. The imbalance consists of high load on the border, as more conditions have to be checked, and weaker but varying load (visualized by blue shades) in the interior that depends on the rock geology. The load imbalance is therefore workload dependent [Tes+18, Figure 2 (a)]. . . 108

9.2 The evolution of the temporal load imbalance for $8 \times 8 = 64$ processes on the Ligurian workload [Tes+18, Figure 2 (b)]. . . . . 109

9.3 Reducing the frequency can save energy when power during idle times is counted as well. This is even the case when the finishing time is pushed back (here: running at reduced frequency takes 1.9166s as opposed to the 1.8s including idle time). . . . . 110

9.4 It is possible to save energy without introducing a delay by overlapping idle time with computations. . . . . 111
9.5 Alternating between the highest frequency and the idle state (represented by frequency "0") consumes more power than running at a reduced frequency (with the same finishing time). The power data was obtained by actual measurements of NAS-LU. . . . . 114
9.6 Using Adagio with Ondes3D works for about 15 iterations, after which even the most loaded host `taurus-16` starts to slow down. . . . . 118
9.7 Around the 85th iteration of the Chuetsu earthquake scenario (with 16 processes) even the last (and most loaded) node `taurus-16` enters the lowest pstate. The vertical lines help to distinguish the start and end of each iteration. . . . . 119

10.1 On the Stampede supercomputer, one "slow" and one "fast" mode seem to exist for `MPI_Recv` and `MPI_Send` (depicted) calls, making faithful simulation very difficult. Figure obtained from Tom Cornebize. 135

# List of Tables


4.1 Exhaustive list of all currently by SimGrid supported contexts, i.e., mechanisms to virtualize user code . . . . . 46

4.2 SMPI implements several algorithms for each collective operations to allow users to simulate their MPI runtime more closely. . . . . 47

5.1 Even though all machines were provisioned with identical hardware, their bios settings proved to be different. This table illustrates the different results we obtained when testing for memory performance through the `mbw` benchmark on 2016-08-12. . . . . 66

5.2 This (arbitrary) excerpt from `/proc/interrupts` shows how interrupts were not correctly re-balanced when disabling all cores (except for CPU0) and then immediately re-activating them. As can be seen, CPU0 handles almost all interrupts exclusively (the interrupts handled by other cores were handled before the cores were disabled and re-activated) continued to almost exclusively handle all interrupts, resulting in a performance degradation. . . . . 69

8.1 An entire execution of NAS-LU (class C) on a single node with 12 cores, broken down by the time spent with each possible load factor and the percentage relative to the total execution time of 81.336 s. . . . . 105

9.1 Power consumption for the CPU-bound toy benchmark NAS-EP with one process per core (i.e., the machine operates under full load) as measured on `taurus-7` during our energy calibration. The scaled values are always scaled with regards to the fastest frequency (2300 MHz). On this machine, choosing lower frequencies on a fully-loaded machine is highly inefficient and results in significant power loss. . . . . 110

9.2 Total idle time of all nodes when using the performance governor (Chuetsu, 300 iterations, 16 processes). Total runtime was 339.74 s . . . 127

9.3 Makespan and energy predictions we obtained when using several governors with the Chuetsu scenario (300 iterations, 16 processes). . . 127

9.4 Total idle time of all nodes when using our proposed Lagrange governor. Total runtime: 340.14 s. (Chuetsu, 300 iterations, 16 processes) . . . 128

9.5 Our improved version of Adagio manages to reduce idle time significantly. Total runtime: 340.12 s. (Chuetsu, 300 iterations, 16 processes) . 129

9.6 The ondemand governor does not significantly reduce idle times. Total runtime: 339.75 s. (Chuetsu, 300 iterations, 16 processes) . . . . . 130
9.7 The conservative governor does not significantly reduce idle times and yields for almost all nodes worse results than the ondemand governor, with the exception of `taurus-6,7,10,11`. Total runtime: 339.75 s. (Chuetsu, 300 iterations, 16 processes) . . . . . 131
9.8 Comparison of makespan and energy estimates when load balancing is used with the performance governor. GreedyLB$_k$ means that every $k$ iterations, GreedyLB is called. (Chuetsu, 300 iterations, 64 processes) 132
9.9 Idle times when using the GreedyLB heuristic every 10 iterations. Total runtime was 267.95 s. (Chuetsu, 300 iterations, 16 processes) . . . . . 132

# List of Abbreviations

ABFT Algorithm-Based Fault Tolerance, page 23

AMPI Adaptive MPI, page 119

AVX Advanced Vector Extensions, page 9

BRGM French Geological Survey, page 109

CC-State Core C-State, page 93

CTS Clear To Send, page 52

DVFS Dynamic Voltage Frequency Scaling, page 92

DVFS Dynamic Voltage Frequency Scaling, page 109

FDM Finite Differences Method, page 109

HPC High-Performance Computing, page 1

HPC High-Performance Computing, page 5

ILP Instruction Level Parallelism, page 9

LC-State Logical C-State, page 93

LMM Linear MaxMin Model, page 39

MKL Math Kernel Library, page 17

MPI Messager Passing Interface, page 16

NoC Network On Chip, page 30

OpenCL Open Compute Language, page 17

OpenMP Open Multi-Processing, page 17

PC-State Processor C-State, page 93

RL Real-Life, page 77

RTS Ready To Send, page 52

TI / TIT Time Independent (Trace), page 53

ULFM User Level Fault Mitigation, page 22