Université Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Tom CORNEBIZE

Thèse dirigée par **Arnaud LEGRAND, CNRS**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
et de l'École Doctorale **MSTII**

Calcul haute performance : vers de meilleures prédictions de performance et expériences

High Performance Computing: Towards Better Performance Predictions and Experiments

Thèse soutenue publiquement le **2 juin 2021**,
devant le jury composé de :

Patrick CARRIBAULT
Ingénieur-Chercheur, CEA, France, Rapporteur
Henri CASANOVA
Professeur, University of Hawai'i at Mānoa, États-Unis, Rapporteur
Abhinav BHATELE
Maître de conférences, University of Maryland, États-Unis, Examinateur
Camille COTI
Maîtresse de conférences, Université Sorbonne Paris Nord, France, Examinatrice
Amina GUERMOUCHE
Maîtresse de conférences, Télécom SudParis, France, Examinatrice
Christian PLESSL
Professeur, Universität Paderborn, Allemagne, Examinateur
Jean-François MÉHAUT
Professeur, Université Grenoble Alpes, France, Président
Arnaud LEGRAND
Directeur de recherche, CNRS, France, Directeur de thèse

En essayant continuellement on finit par réussir. Donc : plus ça rate, plus on a de chance que ça marche.

— **Devise Shadok (Jacques Rouxel)**

Acknowledgments

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

I would like to thank all the authors who upload their papers on open archives like HAL or Arxiv. Likewise, thanks a lot to Alexandra ELBAKYAN for the creation of Sci-Hub, which helped me a lot for retrieving articles behind paywalls.

Thank you to the members of the jury for kindly accepting to be part of this committee. In particular, thanks a lot to the two reviewers, Patrick CARRIBAULT and Henri CASANOVA, who made excellent comments on the manuscript which helped to improve the final version.

Thank you to all the Simgrid contributors for creating and maintaining the software on which is based a great part of this thesis. They have been very helpful and friendly in numerous occasions.

Thank you to the Grid'5000 staff, I cannot emphasize enough how much their work is important. This platform is a wonderful tool for computer science experiments.

Thanks a lot to the members of Polamove and Dataris teams who welcomed me for these last years. The days always pass extremely fast with (long) coffee breaks. I am waiting eagerly the end of this pandemic to resume the traditional summer barbecue.

A huge thank you to Arnaud, my advisor. I learned a lot during this thesis and even started to like probabilities and statistics. Above anything else, thank you for your great kindness and empathy. Back in late 2016, when I was looking for a master internship, I read the advice that the advisor should be the most important criteria when planning a PhD, even more so than the thesis subject or the host institution. I followed this advice and never regretted.

Finally, I would like to thank my family, who helped me and supported me a lot. In particular, Maïlys, my wife, with whom I am extremely fortunate and very happy to share my life, and Yaël, my baby, still too young to understand this text, but already a wonderful person.

Abstract / Résumé

Abstract

The scientific community relies more and more on computations, notably for numerical simulation and data processing. While many scientific advances were made possible by the technological progress of computers, additional performance gains are still required for larger scale projects.

The race for performance is addressed with a growing hardware and software complexity, which in turn increases the performance variability. This can make the experimental study of performance extremely challenging, raising concerns of reproducibility of the experiments, akin to the problems already faced by natural sciences.

Our contributions are twofold. First, we present a methodology for predicting the performance of parallel non-trivial applications through simulation. We describe several models for communications and computations, with an increasing complexity. We compare these models through an extensive validation by matching our predictions with real experiments. This validation shows that modeling the spatial and temporal variability of the platform is essential for faithful predictions. As a consequence, predictions require careful sensibility analysis accounting for the uncertainty on the resource models, which we illustrate through several case studies. Second, we present the lessons learned while making the numerous experiments required in the first part and how we improved our methodology. We show that measurements can suffer from multiple experimental biases and we explain how some of these biases can be overcome. We also present how we implemented systematic performance non-regression testing, which allowed us to detect many significant changes of the platform throughout this thesis.

Résumé

La communauté scientifique s'appuie de plus en plus sur les calculs, notamment pour la simulation numérique et le traitement des données. Alors que de nombreuses avancées scientifiques ont été rendues possibles par les progrès technologiques des ordinateurs, des gains de performance supplémentaires sont encore nécessaires pour les projets à plus grande échelle.

La course à la performance est abordée avec une complexité matérielle et logicielle croissante, qui à son tour augmente la variabilité des performances. Cela peut rendre l'étude expérimentale de la performance extrêmement difficile, ce qui soulève des préoccupations quant à la reproductibilité des expériences, de manière similaire aux problèmes déjà rencontrés par les sciences naturelles.

Nos contributions sont doubles. Tout d'abord, nous présentons une méthodologie pour prédire les performances d'applications parallèles non triviales par la simulation. Nous décrivons plusieurs modèles de communications et de calculs, avec une complexité croissante. Nous comparons ces modèles via une validation approfondie en faisant correspondre nos prédictions avec des expériences réelles. Cette validation montre que la modélisation de la variabilité spatiale et temporelle de la plateforme est essentielle pour les prédictions. En conséquence, les prévisions requièrent une analyse de sensibilité minutieuse tenant compte de l'incertitude sur les modèles de ressources, que nous illustrons à travers plusieurs études de cas. Par la suite, nous présentons les leçons apprises lors des nombreuses expériences menées dans la première partie et comment nous avons amélioré notre méthodologie. Nous montrons que les mesures peuvent souffrir de multiples biais expérimentaux et nous expliquons comment certains de ces biais peuvent être surmontés. Nous présentons également comment nous avons mis en œuvre des tests systématiques de non-régression des performances, qui nous ont permis de détecter de nombreux changements significatifs de la plateforme tout au long de cette thèse.

Contents

1 Scientific computing: a story

1.1 First computers, from carbon to silicon

Science has always been tightly associated to computations, hence it is no surprise that the first computers were not machines, but humans. Already in the second century AD, Ptolemy, a scientist living in Alexandria, wrote the Almagest. This book aggregated the state of the art in mathematics and astronomy and remained a reference for centuries. It contained several tables that were computed by the scientist, including a trigonometric table (called *table of chords*).

In 1757, three French astronomers, Clairaut, Lalande and Lepaute, started working on the prediction of the next appearances of Halley's comet [Gri05, Chapter 1]. Using the recent theories of Newton, they had to numerically solve the three-body problem: they computed the orbits of Saturn and Jupiter around the Sun, taking into account the attraction force between the two planets. They carried this computation by splitting the orbits into tiny steps, computing the new planet locations for each step. They used these sequences of coordinates to compute the orbit of Halley's comet around the Sun, by taking into account the effect of the two giant planets on the comet and neglecting the effect of the comet itself on the three bodies. In the end, they were successful in predicting the next appearance of the comet in the beginning of 1759, making an error of only one month. Their work constitutes one of the first recorded division of labor applied to computations, Lalande and Lepaute computed the orbits of the two giant planets while Clairaut computed the orbit of the comet.

Gaspard de Prony, a French civil engineer, went a step further in this endeavor of organized computation [Gri05, Chapter 2]. In 1791, he was named director of the *Bureau du Cadastre*. At the time, the French revolutionary government was preparing reforms for their outdated system of weights and measures, which will eventually result in the creation of the metric system. The reforms proposed to measure angles in grades instead of degrees, dividing a right angle into 100 grades instead of 90 degrees. Prony was tasked to prepare trigonometric tables for this decimal grade system. He organized his staff in a hierarchy with three levels:

- The first class of workers, a handful of renowned scientists like Carnot or Legendre, oversaw the operations. They had to research the appropriate formulas for computing approximations of trigonometric functions with basic arithmetical operations.
- The second class, subsequently named *planning committee*, was a team of eight experienced computers. Their task consisted in translating the trigonometric equations produced by the first class into a sequence of additions and subtractions. They prepared worksheets where all the basic operations were written with a blank space for the result.
- The third class consisted in nearly ninety computers. Many of them were former servants or wigmakers that lost their jobs with the Revolution and did not know any mathematics besides the addition and subtraction. Their job was to compute the results to fill the blank spaces left by the second class of workers.

The idea of constructing a machine capable of doing computations is not recent. Already in 1642, Blaise Pascal, a French mathematician, invented and built a mechanical calculator that could perform the four arithmetical operations. The calculator was not commercially viable at the time, so only twenty machines were built. Much later, in the first half of the nineteenth century, Charles Babbage [Gri05, Chapters 2-3], an English mathematician, invented two very innovative machines. The difference engine, for computing tables of polynomial functions, and the analytical engine, a general purpose computer that would subsequently be qualified as *Turing-complete*. Unfortunately, due to a lack of funding, he was never able to build his inventions. In the same period, a French inventor named Thomas de Colmar designed and manufactured a digital mechanical calculator, called arithmometer. Capable of doing the four arithmetical operations, it was the first machine of its kind to be reliable enough for a practical use. Similarly, Herman Hollerith, an American inventor, created the tabulating machine. Initially built to process the 1890 US Census data, it worked by reading and summarizing information stored in punched cards. It decreased considerably the duration and cost of the whole census organization. These two last inventions [Gri05, Chapter 6] were commercial successes and started an era of mechanical computations that lasted until the second half of the twentieth century.

Gradually, computing became more and more important and recognized as a discipline. The apparition of modern statistics, mainly due to the work of Francis Galton and Karl Pearson in the early twentieth century, led to a growing need for computation power. The First World War itself was an important catalyst, as the

American, French and English governments hired entire computing laboratories to create ballistic tables [Gri05, Chapter 10]. By the time, electromechanical computers were used everywhere, for their efficiency and reliability largely superior to human computers. There was still an important need for human labor, not only for operating these machines, but because some complex operations were still carried by hand.

The first working general purpose programmable computer, named *Z3*, was designed and built more than two decades later in Germany by Konrad Zuse [Cop20]. Completed in 1941, it was an electromechanical machine, using both (mechanical) relays and (electronic) vacuum tubes. Its programs, written on external tapes, could use loops but not conditional branches. In 1944, the British government built the first fully electronic computer, named *Colossus*. Made of vacuum tubes, its primary function was to break the German ciphers during the war. Later, in 1945, the first US electronic computer was unveiled, called *ENIAC*. Both the Colossus and ENIAC computers were programmed by plugboards and switches, instead of reading from a tape like the Z3. Interestingly, the Z3 and Colossus machines used binary arithmetic while the ENIAC used decimal representation.

An important breakthrough came with the notion of stored-program computer, i.e. the idea of storing the program instructions in memory. Although the original ideas can be traced to Turing himself and his 1936 article, the real implementation in electronic computers came several years later. The first stored-program computer was the *Manchester Baby*, built at the University of Manchester in 1948 [Cop20]. Similarly, the successor of the ENIAC, called *EDVAC* was also a stored-program computer. Yet, at the time, computers were still using vacuum tubes. Although this was a great reliability improvement to the mechanical parts used before, the vacuum tubes had a very large electricity consumption which started to become problematic (the ENIAC consumed $150\,\mathrm{kW}$, the EDVAC $56\,\mathrm{kW}$). They also required many human operators for their daily usage.

The invention of the transistor in 1947 by three Bell Labs physicists had a huge impact on the whole electronic world. It achieves the same functionality as a vacuum tube (amplifying and switching an electrical signal), but with a much lower power consumption, much smaller size and easier to mass produce. Hence, it is no surprise if the industry quickly started to use this new technology. The first transistor computer was made in 1953 in the University of Manchester. Later, in 1955, IBM announced the first commercial transistor computer, the IBM 608 [IBM21], made of 3000 transistors. Compared to its predecessor, switching to transistors allowed IBM to reduce the computer physical size by $50\,\%$ and its power consumption by

$90\,\%$ while multiplying its computing speed by 2.5, reaching a performance of 4500 additions per second.

1.2 Exponential growth

In 1965, Gordon Moore, who will later become the CEO and co-founder of Intel, predicted an exponential growth of the number of transistors in a chip [Moo65], based on an extrapolation of the current pace of technological progress. He estimated that the number of transistors was doubling every year:

> The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years.

Ten years later, Moore revised his forecast to a doubling every two year [Moo75]. This prediction, which revealed to be true, is now known as *Moore's law*.

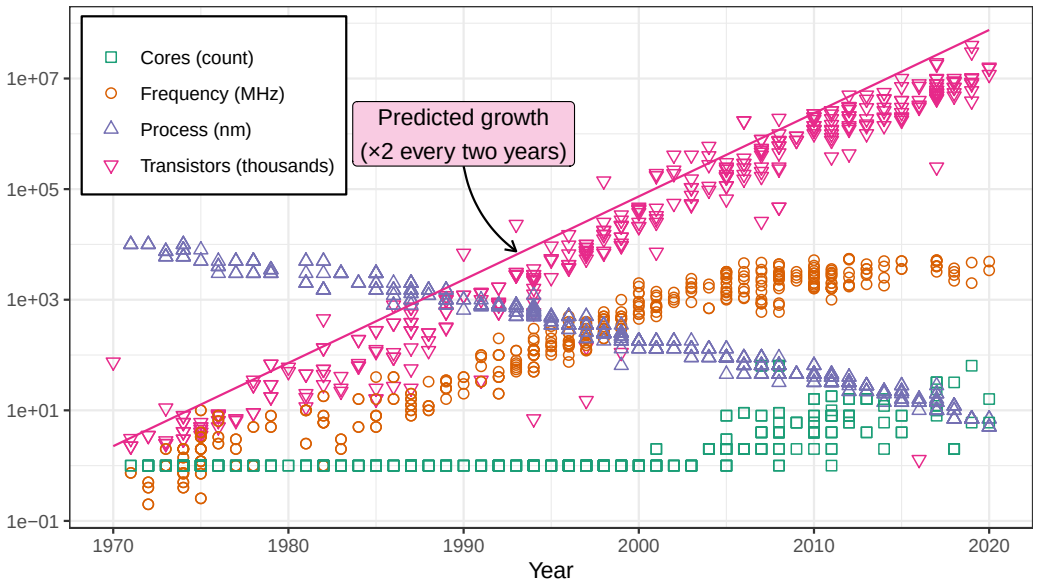


Figure 1.1.: Evolution of the processor characteristics between 1971 and 2020. Plot inspired from the work of Pedro Bruel, generated with data from Wikipedia [Wik21a; Wik21b].

One of the main contributors to this exponential growth of the number of transistors is the exponential decrease of their size, as plotted in Figure 1.1. While the IBM

608 calculator commercialized in the 50s had transistors “no bigger than a paper clip” [IBM21], the latest processors commercialized in 2020 have $5\,\mathrm{nm}$ transistors.

In 1974, Dennard et al. listed a set of rules for scaling simultaneously the transistor density, clock frequency and power dissipation of processors [Den+74], which would eventually be named *Dennard scaling*. The effect of this scaling on the device is summarized in Table 1.1. A scaling of κ will result in a clock frequency multiplied by a factor κ and a number of transistors multiplied by κ^2 while the power density of the chip remains constant, i.e. if the size of the processor does not change, it will have the same power consumption and generate the same amount of heat.

Table 1.1.: Dennard scaling with a factor κ (table reproduced from [Den+74, Table 1]).

Device or Circuit Parameter	Scaling Factor
Device dimension t_{ox}, L, W	$1/\kappa$
Doping concentration N_a	κ
Voltage V	$1/\kappa$
Current I	$1/\kappa$
Capacitance $C = \epsilon A/t$	$1/\kappa$
Delay time/circuit VC/I	$1/\kappa$
Power dissipation/circuit VI	$1/\kappa^2$
Power density VI/A	1

Unfortunately, Dennard scaling came to an end 15 years ago. Indeed, it is no longer possible to scale the operating voltage and the gate oxide thickness [Boh07], as transistors have reached scales where power leakage is no longer negligible. With a voltage that cannot be scaled anymore, the power density cannot remain constant and reaches alarming levels of more than $4\,\mathrm{W/mm^2}$ [HP19; HP18], as illustrated in Figure 1.2.

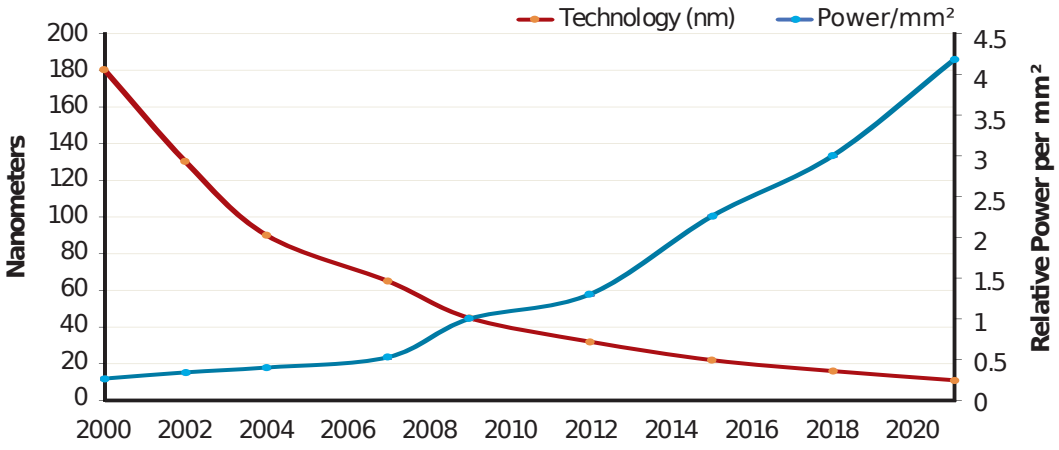


Figure 1.2.: Evolution of the power density in the last 20 years (plot reproduced from [HP19, Figure 3]).

Consequently, it has become impossible to increase further the CPU frequency, which has reached a limit of about $5\,\mathrm{GHz}$ since 2006 after an exponential growth of several decades, as shown by Figure 1.1. One of the responses to keep the race for performance was to design multicore processors. Nowadays, nearly all laptops and smartphones have several cores, while high-end servers can have two or four sockets with up to 64 cores per socket.

Yet, it appears that using several cores was only a short-term help for increasing the CPU performance. With the end of Dennard scaling, the power density increases with each generation of processor. Hence, to stay within a safe thermal design power (TDP), it is no longer possible to power at the nominal voltage all the components of the processors (notion named *dark silicon*). Esmaeilzadeh et al. predict that with the $8\,\mathrm{nm}$ processor generation, more than $50\,\%$ of their transistors might be unpowered at any given time [Esm+11]. With these constraints, Hennessy and Patterson [HP19; HP18] estimate that the single-processor performance is now only growing by a mere $3\,\%$ per year, which is a much slower pace than the $50\,\%$ yearly growth rate that the industry got used to for decades.

1.3 Scientific computing today

1.3.1 High performance computing

Computations are now used in every scientific fields, from small statistical calculations to large numerical simulations. Science as a whole has immensely benefited from this exponential performance improvement. For instance, the price for sequencing an entire human genome has dropped from $\$100{,}000{,}000$ in 2001 to $\$1000$ in 2020 [NIH].

To reach the highest performance, a single processor is not enough, and the benefit of doing computations in parallel was already well known at the time of human computers, as discussed in Section 1.1. For this reason, the largest computations are now performed on supercomputers, large machines made of many processors connected through a fast network. The 500 fastest non-classified supercomputers of the world are ranked biannually in the Top500 list [top500]. As shown by Figure 1.3, they suffered from the same frequency staling as the rest of the industry in 2006, yet their performance kept an exponential increase, in part explained by the exponential increase of their total number of cores.

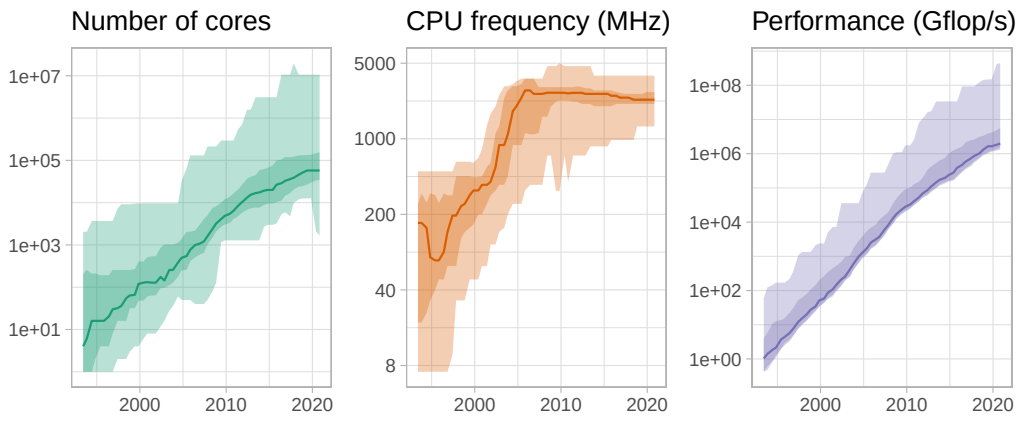


Figure 1.3.: Evolution of the Top500 [top500] supercomputers between 1993 and 2020. The line denotes the median, the inner ribbon contains the $[10\,\%, 90\,\%]$ interval, the outer ribbon contains all the values.
Data compiled by Dan Lenski [Len20] and plotted by ourselves.

The largest supercomputers now have thousands of processors, millions of cores and power consumptions of several megawatts. At these scales, and with the hardware and software complexity required to reach the highest performance, it is extremely difficult to have an accurate picture of the whole platform. Even if all the individual components are deterministic at a microscopic level, a supercomputer observed at a macroscopic level can arguably be seen as a stochastic object. Hence, it is very common for the operators or the users of these machines to stumble on performance anomalies, i.e. the *observed* performance is different from the *expected* performance. For instance, Petrini et al. [PKP03] found a severe but previously undetected performance problem on the supercomputer they were using, responsible for a $100\,\%$ performance variation and caused by operating system perturbations. Likewise, Tuncer et al. [Tun+17, Section 1] list several examples of performance problems observed in operation:

- The amount of variation in application running time can reach $100\,\%$ on real-life systems [Bha+13; SK05].
- Orphan processes left over from previous jobs consuming system resources [Bra+10].
- Firmware bugs, affecting the CPU usage on the server and making the user programs fail [Cisco].
- Memory leaks in applications, eventually leading to job failure [Age+15].
- CPU throttling of the nodes for thermal control [Bra+15].
- Resource contention [Bha+13; Dor+14].

If unnoticed, these kinds of performance anomalies can severely impact the conclusions than an experimenter would draw based on performance measures obtained on the machine.

1.3.2 A reproducibility crisis

For several years there has been an ongoing concern of a reproducibility crisis in science. An online survey done in 2016 found that among the 1576 scientists who responded, $70\,\%$ have failed at least once to reproduce the work of another scientist, $50\,\%$ have even failed to reproduce their own work [Bak16]. This crisis is particularly concerning in medical science, where some researchers estimate that a large part of claimed research findings are false [Ioa05; Fre10]. In this field, the main sources of non-reproducibility are a wrong usage of statistics, poor experimental methods or even frauds [Sch20]. In other disciplines such as computational biology or computational physics, reproducibility issues mainly come from numerical instability and the growing complexity of software environments [Ahn+21].

Computer science is not immune to these concerns of statistical misusage, numerical errors or frauds. A recurrent problem is also the unavailability of the software programs, methods and data on which are based the research articles. For instance, Collberg et al. examined 601 papers from ACM conferences and journals, they were able to personally build the software of only 194 of them while they simply could not access the code of 176 of these papers [CPW15]. Likewise, Papadopoulos et al. propose eight methodological principles for better reproducibility and show that most of the published articles they surveyed did not follow all these principles [Pap+19].

What may be more surprising is that computer science suffers from the same experiment reproducibility problems than the other sciences: experimental biases exist and may even be larger. For instance, Mytkowicz et al. show that simply changing the environment variable size can account for a performance variability of more than $30\,\%$ [Myt+09]. They propose two methods for detecting and avoiding such measurement bias, based on experiment randomization. Similarly, Curtsinger and Berger have implemented a software for repeatedly re-randomizing the layouts of code, stack and heap at runtime, to avoid the bias introduced by a particular binary layout [CB13]. Again, the growing problem of experimental reproducibility we are describing here is tightly coupled to the growing software and hardware complexity.

1.4 This thesis

In this thesis, we argue that a large part of computer science, and in particular high performance computing, is an experimental science. It is commonplace for computer scientists to run experiments, e.g. for comparing the performance of several algorithms. With the kind of variability described in Section 1.3, such experiments could easily lead to false conclusions. We therefore advocate for using similar methods to the natural sciences, which have been faced with the same issues for decades.

Our contribution towards this goal of improving the confidence in the experimental results is multiple:

- In Part I, we propose a methodology for predicting the performance of an application through simulation. Similarly to biology or physics, making an experiment in simulation can be of great help to rule out part of the experimental bias. It can also enable the researchers to test scenarios that would be too costly in reality.
  - We contextualize this work in Chapters 2 and 3.
  - We explain how to improve the efficiency of the simulation in Chapter 4.
  - The models we used for the predictions are presented in Chapter 5.
  - Chapter 6 is a thorough validation of the simulations, where we compare the predicted performance with the observations made on real experiments.
  - We illustrate an important use case of such simulations in Chapter 7 by performing several sensibility studies.
- In Part II, we present additional tools and methods to help experimenters increase the quality of their work.
  - Automation is a great way to improve both reliability and productivity. It helped tremendously throughout the last century for scientific computations, as shown in Section 1.1. Machines make much fewer mistakes than humans (if any) and are much faster. For these reasons, we implemented several programs during this thesis. We automatized the execution of experiments with an experiment engine, presented in Chapter 9. We also implemented several tools, listed in Appendix B, for automating some parts of the analyses as well as cumbersome daily tasks.

– Several times while conducting the research in this thesis, our performance models were not in agreement with the reality we observed. In some instances, our models were too simple, and we needed to add complexity to capture additional real-world phenomena, as described in Chapters 5 and 6. But, most of the time, we suffered from an experimental bias (or the lack thereof). These difficulties are described in depth in Chapter 10.

– With an experimental work that spans several years, like this thesis, it is very likely that the object of the experimental study will evolve. In the case of high performance computing, the machine can have hardware or software upgrades, or even defects, that can all affect significantly the measured performance. In Chapter 11, we present the performance non-regression tests we implemented and used in the second half of this thesis. They helped us to detect numerous platform changes that could have led to wrong conclusions if unnoticed.

Part I

Performance prediction through simulation

The work presented in this part has been published at a conference [CLH19] and has been submitted for publication in a journal [CL21e]. The content of this part contains near-verbatim extracts of these articles, with various additions. This work also directly follows my master thesis [Cor17] whose main contribution is summarized in Chapter 4 for the sake of completeness.

2 Related work

In this chapter, we discuss the different techniques and tools available for predicting the performance of an application. Section 2.1 explains why we need simulation to obtain both efficient and faithful predictions while Section 2.2 gives an overview of the simulator we used for this work.

2.1 Performance prediction

Researchers and engineers often need to make predictions about the performance of a given parallel application on a given platform. The objectives are diverse: to compare several algorithms, to verify if the observed performance is as high as one could hope, or to estimate the gain they could get by upgrading their hardware.

Depending on the exact needs of its user, such a prediction should have several qualities:

Extrapolation on the problem size Most of the applications can take as input problems of various size. The size may denote the number of particles in a physics simulator or the matrix rank in a linear algebra solver. The total duration will usually grow with the problem size. This criterion denotes whether the considered approach can predict the performance of the application for an arbitrary problem size.

Extrapolation on the configuration It is often possible to tune the behavior of the application with some parameters, for instance to select the desired algorithm or the desired granularity. The reason is that the optimal parameter combination may depend on the hardware, the number of nodes, the problem size. This criterion designates if the approach can make predictions for an arbitrary parameter combination.

No full-scale real run Some prediction methods require at least one run of the application at the desired scale in terms of number of nodes. This criterion denotes whether the approach can avoid making such expensive runs (note

that the full-scale runs do not necessarily need to be done on the target platform).

Hypothetical platform This criterion denotes the ability to study *what-if scenarios*, e.g. to predict the performance of the application on a hypothetical cluster with a higher network bandwidth than the ones available.

Efficiency Some approaches have a much higher resource requirement than others. Several metrics can be considered: node-hours, time to solution, energy consumption, memory footprint.

Accuracy While some predictions would only give an approximate trend, some others would be much more accurate, as close as a few percents, even in the presence of perturbations like network contention.

One very common prediction technique is the use of an analytical model. For instance, most of the publications which present a new algorithm will give an estimation of its asymptotic complexity, a.k.a. big-O analysis. These techniques implicitly use an abstraction of the underlying hardware (e.g. the von Neumann model, cache model [Fri+12], PRAM [KR89], LogP [Cul+93], LogGP [Ale+97], BSP [Val90]). There also exists more systematic approaches, like Aspen [SV12], which defines a domain specific language (DSL) for specifying formally the characteristics of small components and combining them to model a whole application. Figure 2.1 illustrates an Aspen model of a Fast Fourier Transform that can be used within a larger application model. The main downside of these analytical techniques is their inability to provide accurate estimates. For instance, complex phenomena like network contention cannot be captured faithfully by these models.

```
kernel localFFT {
    exposes parallelism [n^2]
    requires flops [5 * n * log2(n)] as dp, simd
    requires loads [a * (n*wordSize) * max(1, log(
        n*wordSize)/log(z))] from fftVolume
}
```

Figure 2.1.: Example of model written in Aspen for a local 1D FFT [SV12].

Another approach for estimating the performance of parallel applications implemented using the Message Passing Interface (MPI) standard is statistical modeling of the application as a whole [LD12]. By running the application several times for small- and medium-sized problem (or a few iterations of large problem sizes) and using simple linear regressions, it is possible to predict its execution time for larger sizes with an error of only a few percents and a relatively low cost. Unfortunately, the predictions are limited to the same application configuration and studying the

influence of the number of rows and columns of the virtual grid or of the broadcast algorithms requires a new model and new (costly) runs using the whole target machine. Singh et al. [Sin+07] proposed the reverse approach. They fixed the problem size and sampled random parameter configurations to train a neural network. Again, this allowed them to make accurate predictions at a low cost, but it was not possible to predict the execution time with an arbitrary problem size. Furthermore, such approaches cannot be used to study what-if scenarios (e.g. to evaluate what would happen if the network bandwidth was increased or if node heterogeneity was decreased) that are particularly useful when investigating potential performance improvements.

Simulation provides the level of details and flexibility that is lacking in such black-box modeling approaches. Performance prediction of MPI applications through simulation has been widely studied over the last decades but two approaches can be distinguished in the literature: offline and online simulation.

With the most common approach, *offline simulation*, a trace of the application is first obtained on a real platform. This trace comprises sequences of MPI operations and CPU bursts and is given as an input to a simulator that implements performance models for the CPUs and the network to derive predictions. Researchers interested in finding out how their application reacts to changes to the underlying platform can replay the trace on commodity hardware at will with different platform models. Most HPC simulators available today, notably BigSim [ZKK04], Dimemas [Bad+03], LogGOPSim [HSL10] and CODES [Mub+16], rely on this approach. The main limitation of this approach comes from the trace acquisition requirement. Not only is a large machine required but the compressed trace of a few iterations of an MPI application can quickly reach a few hundred MB, making this approach quickly impractical [Cas+15]. Worse, tracing an application provides only information about its behavior at the time of the run: slight modifications (e.g. to communication patterns) may make the trace inaccurate. The behavior of simple applications (e.g. `stencil`) can be extrapolated from small-scale traces [WM11; CLT13] but this fails if the execution is non-deterministic, e.g. whenever the application relies on non-blocking communication patterns, which is unfortunately often the case.

The second approach discussed in the literature is *online simulation*. Here, the application execution is emulated on top of a simulator that is responsible for simulating the execution of each application process including inter-process communication. This approach allows researchers to study directly the behavior of MPI applications but only a few recent simulators such as SST Macro [Jan+10], SimGrid/SMPI [Cas+14] and the closed-source xSim [Eng14] support it. To the best

```
SUBROUTINE dm_x_bcs
    ...
    CALL MPI_SENDRECV(dm(nx-1, 0:ny+1), ny+2, mpireal, &
        proc_x_max, tag,dm(-1, 0:ny+1), ny+2, mpireal, &
        proc_x_min, tag, comm, status, errcode)
    ...
END SUBROUTINE dm_x_bcs
```

(a) Original Fortran `dm_x_bcs` subroutine

```
void dm_x_bcs(int rank) {
    ...
    mpi->sendrecv(ny+2, sstmac::sw::mpytype::mpi_real,\
    proc_x_max, tag, ny+2, sstmac::sw::mpitype::mpi_real,\
    proc_x_min, tag, world(), stat);
    ...
}
```

(b) Model `dm_x_bcs` subroutine

```
SUBROUTINE remap_x ! remap onto original Eulerian grid
    ...
    DO iy= -1, ny+2
        iym = iy - 1
        DO ix = -1, nx+2
            ixm = ix - 1
            ...
        END DO
    END DO
    ...
END SUBROUTINE remap_x
```

(c) Original Fortran `remap_x` subroutine

```
void remap_x(int rank) {
    ...
    sstmac::timestamp t(remap_x_w*nx*ny);
    compute(t);
    ...
}
```

(d) Model `remap_x` subroutine

Figure 2.2.: Example of skeletonization done by SST [Bir+13].

of our knowledge, only SST Macro and SimGrid/SMPI are mature enough to faithfully emulate an MPI application. This work relies on SimGrid as its performance models and its emulation capabilities are now well established, but the proposed developments would also be possible with SST. Note that the emulation described in Chapter 4 should not be confused with the application skeletonization [Bir+13] commonly used with SST. Skeletons are code extractions of the most important parts of a complex application, as illustrated in Figure 2.2 (e.g. the original Fortran code from Figure 2.2(c) has been translated into the skeleton from Figure 2.2(d)). In our work, we only modify a few dozens of lines of the source code before emulating

it with SMPI. Finally, it is important to understand that the proposed approach is intended to help studies at the level of the whole machine and application, not the influence of microarchitectural details as intended by gem5 [Low+20] or by MUSA [Gra+16].

Some researchers from Intel unaware of our recent work recently applied the same methodology as the one we proposed in [CLH19] to both Intel HPL and OpenHPL in the closed-source CoFluent simulator[Xu+20]. To the best of our knowledge, their work reports two faithful predictions for two large-scale supercomputers but without investigating at all the impact of variability, heterogeneity, nor of communications as we do in this work. Unfortunately, they gave very little details on what allowed them to achieve such impressive results, and they did not publish their code. Given the context, we believe their claims should be interpreted with caution.

2.2 Simgrid/SMPI

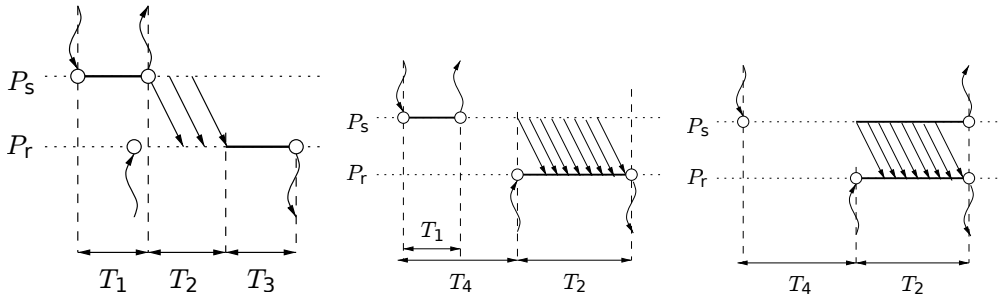
SimGrid [Cas+14] is a flexible and open-source simulation framework that was originally designed in 2000 to study scheduling heuristics tailored to heterogeneous grid computing environments but has later been extended to study cloud and HPC infrastructures. The main development goal for SimGrid has been to provide validated performance models particularly for scenarios making heavy use of the network. Model validation usually consists of comparing simulation predictions with results from real experiments to confirm or debunk network and application models.

SMPI, a simulator based on SimGrid, has been developed and used to simulate unmodified MPI applications written in C/C++ or FORTRAN [Deg+17]. SMPI maps every MPI rank of the application onto a lightweight simulation thread. These threads are then run one at a time, i.e. in mutual exclusion. Every time a thread enters an MPI call, SMPI takes control and the time that was spent computing (isolated from the other threads) since the previous MPI call is injected into the simulator as a virtual delay. This time may be scaled up or down depending on the speed of the simulated machine with respect to the simulation machine. Then, SMPI has to estimate the duration of the communication, based on a network model.

SMPI relies on SimGrid's communication models where each ongoing communication is represented as a whole (as opposed to single packets) by a *flow*. Assuming steady-state, contention between active communications can then be modeled as

a bandwidth sharing problem that accounts for non-trivial phenomena (e.g. cross-traffic interference [Vel+13]). If needed, communications that start or end trigger a re-computation of the bandwidth share. In this fluid model, the time to simulate a message passing through the network is independent of its size, which is advantageous for large-scale applications frequently sending large messages and orders of magnitude faster than packet-level simulation. SimGrid does not model transient phenomena incurred by the network protocol but accounts for network topology and heterogeneity.

SMPI is in charge of modeling the MPI behavior, both in terms of semantic and performance. The complex network optimizations done in real MPI implementations need to be considered when predicting the performance of MPI point-to-point communications. For instance, the behavior of `MPI_Send` depends on the message size, as illustrated by Figure 2.3. The smallest messages are sent *asynchronously* (Figure 2.3(a)), meaning that `MPI_Send` returns immediately and the data is transferred without waiting. The largest messages are sent *synchronously* (Figure 2.3(c)), i.e. the call to `MPI_Send` will be blocking, waiting for the corresponding `MPI_Recv` call to be made. The medium messages are sent in a *detached* way (Figure 2.3(b)), the call to `MPI_Send` returns immediately but the data is only transferred when the corresponding `MPI_Recv` call is made. These different protocols strongly impact the communication performance, it is therefore important to model them accurately.



(a) Asynchronous mode (b) Detached mode (c) Synchronous mode

Figure 2.3.: The "hybrid" network model of SMPI in a nutshell [Deg+17].

With SMPI, the user is responsible for instantiating the model, i.e. they have to provide the aforementioned thresholds where the communication protocols change as well as the bandwidths and latencies for each protocol. An alternative approach was used by Emmanuel et al. [Emm+20] for modeling the Portals 4 protocol, they used a lower abstraction level, at the message passing level instead of the application. Applying the same approach in SMPI would amount to model message communications on the network instead of MPI calls, the MPI calls would then be emulated like the rest of the code. One benefit of this approach is that it

would remove some burden of the user, since it would no longer be required to provide the protocol thresholds and individual bandwidths and latencies, a few bandwidth/latency pairs would be enough. The downside is that the simulation would then depend on the MPI implementation (e.g. OpenMPI, MPICH, Intel MPI) which should be provided either by SMPI or by the user. It would also be impossible to simulate closed-source MPI implementations.

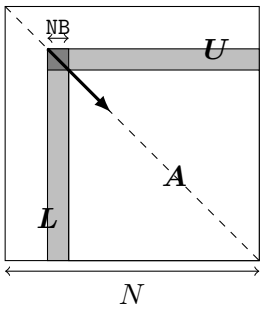
In SMPI, collective communications are simulated by emulation. Simgrid authors have ported in SMPI all the collective communication algorithms of the main MPI implementations, the user simply has to choose the implementation of their choice. This is of little significance in this work, as HPL ships with its own implementation of collective operations, but it can be critical for other applications.

The accuracy of simulations conducted with SMPI has been reported in the literature for several applications. Recent results report consistent performance predictions within a few percent for standard benchmarks on small-scale clusters (up to 12×12 cores [Hei+17] and up to 128×1 cores [Deg+17]). In this thesis, we validate this approach at a much larger scale with HPL.

3 High Performance Linpack

This chapter gives a thorough overview of the application we used as a case study for this work, namely High Performance Linpack (HPL). It presents the different state-of-the-art techniques employed in HPL to reach the highest possible performance, making it more challenging to simulate accurately.

3.1 The benchmark



Algorithm 1: HPL

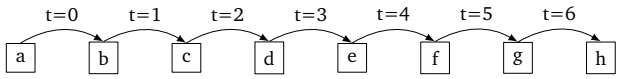
allocate and initialize A
for $k = N$ **to** 0 **step** *NB* **do**
  allocate the panel
  factor the panel
  broadcast the panel
  update the sub-matrix;

Figure 3.1.: Overview of High Performance Linpack.

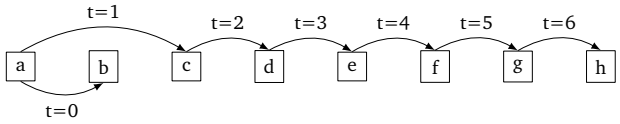
In this work, we use the freely-available reference-implementation of HPL [Pet+] as our main target application. HPL is a parallel application implemented with MPI (Message Passing Interface), an established standard in the industry [For93]. HPL implements a matrix factorization based on a right-looking variant of the LU factorization with row partial pivoting and allows multiple look-ahead depths. The principle of the factorization is depicted in Figure 3.1. It consists of a series of panel factorizations followed by an update of the trailing sub-matrix. HPL uses a two-dimensional block-cyclic data distribution of A and implements several custom MPI collective communication algorithms to efficiently overlap communications with computations. The main parameters of HPL are:

- `N` is the order of the square matrix A.
- `NB` is the *blocking factor*, i.e. the granularity at which HPL operates when panels are distributed or worked on.

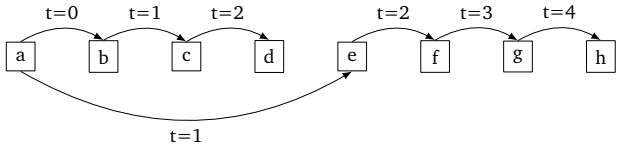
- `P` and `Q` denote the number of process rows and the number of process columns.
- `RFACT` determines the panel factorization algorithm. Possible values are Crout, left- or right-looking.
- `SWAP` specifies the swapping algorithm used while pivoting. Two algorithms are available: one based on *binary exchange* (along a virtual tree topology) and the other one based on a *spread-and-roll* (with a higher number of parallel communications). HPL also provides a panel-size threshold triggering a switch from one variant to the other.
- `DEPTH` controls how many iterations of the outer loop overlap with each other.
- `BCAST` sets the algorithm used to broadcast a panel of columns over the process columns. Legacy versions of the MPI standard only supported non-blocking point-to-point communications, which is why HPL ships with in total six self-implemented variants to overlap the time spent waiting for an incoming panel with updates to the trailing matrix: `ring`, `ring-modified`, `2-ring`, `2-ring-modified`, `long`, and `long-modified`, as illustrated by Figure 3.2 and Figure 3.3.



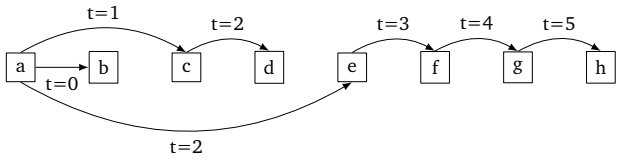
(a) Broadcast `ring`.



(b) Broadcast `ring-modified`.

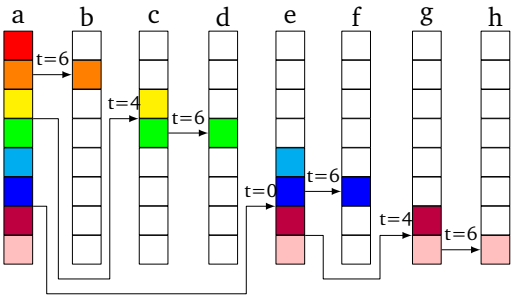


(c) Broadcast `2-ring`.

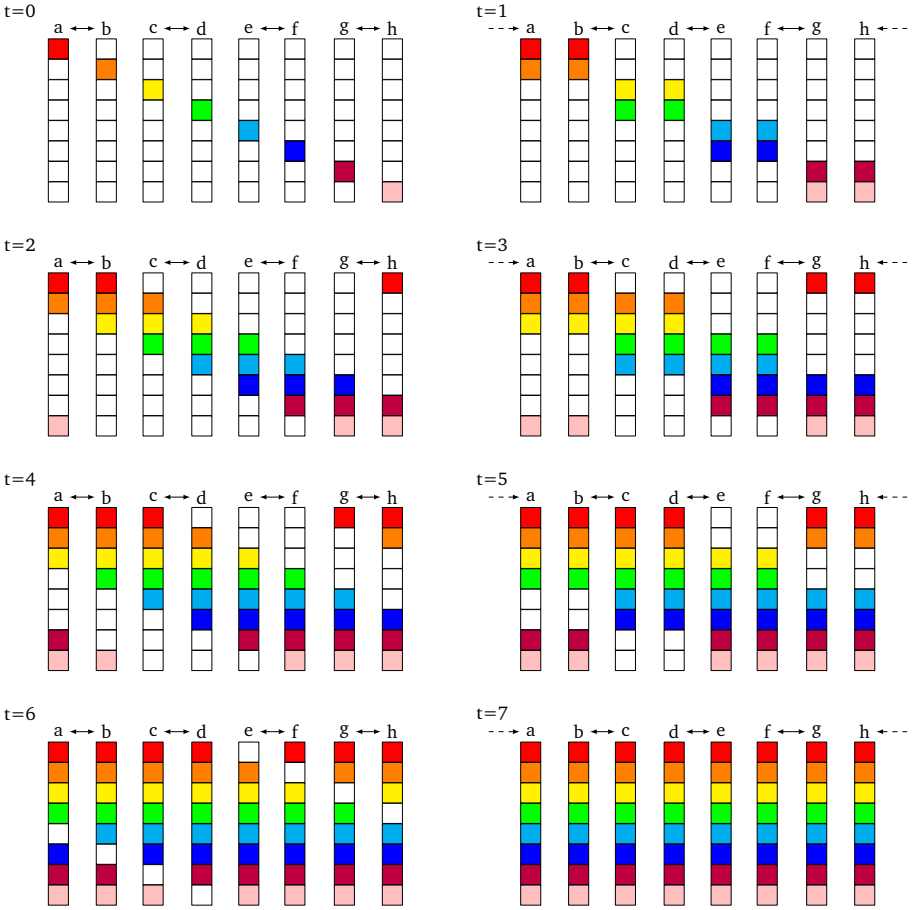


(d) Broadcast `2-ring-modified`.

Figure 3.2.: Illustration of the four HPL ring-broadcast algorithms.



(a) Broadcast `long`, scatter phase.



(b) Broadcast `long`, roll phase.

Figure 3.3.: Illustration of the two phases of the HPL `long` broadcast algorithm. The `long-modified` algorithm (not shown) is very similar, except that process `A` starts by sending the whole message to process `B`, then the `long` broadcast is performed among all processes except process `B` which starts its computations.

The `modified` versions guarantee that the process right after the root (i.e. the process that will become the root in the next iteration) receives data first and does not further participate in the broadcast. This process can therefore start working on the panel as soon as possible. The `ring` and `2-ring` versions each broadcast along the corresponding virtual topologies (see Figure 3.2) while the `long` version is a *spread and roll* algorithm: messages are chopped into `Q` pieces, these `Q` pieces are scattered across the `Q` processes with a binary tree as in Figure 3.3(a), then the pieces are rolled in `Q`-1 steps using mutual exchanges as in Figure 3.3(b). According to HPL authors, the two `long` algorithms generally lead to better bandwidth exploitation, these algorithms send a larger number of messages but the total volume of communication is independent of `Q`: this algorithm is particularly well suited for platforms where the nodes are comparatively much faster than the network.

The `ring` and `2-ring` variants rely on `MPI_Iprobe`, meaning they return control if no message has been fully received yet, hence facilitating partial overlap of communication with computations as illustrated in Algorithm 2.

Algorithm 2: Illustrating the probing mechanism used in HPL broadcasts.

```
try to broadcast
while the broadcast did not succeed do
    perform computations (including calls to dgemm)
    try to broadcast
```

In HPL 2.1 and 2.2, this capability has been deactivated for the `long` and `long-modified` algorithms. A comment in the source code states that some machines apparently get stuck when there are too many ongoing messages.

The *panel* mentionned in Figure 3.1 is an intricate data structure used by multiple functions of the application. It contains both `ints` (accounting for matrix indices, error codes, MPI tags, and pivoting information) and `doubles` (corresponding to a copy of a sub-matrix of `A`). To optimize data transfers, HPL flattens this structure into a single allocation of `doubles`, as in Figure 3.4.

We would like to stress through these examples that although HPL is only a benchmark (as opposed to a real application used in physics simulations for instance), it is already a complex program with elaborated programming techniques to reach the highest available performance. It is possible to get a very rough estimation of HPL's execution time with an analytical model, but more accurate predictions require the

```
typedef struct HPL_S_panel
{
   struct HPL_S_grid   * grid;             /* ptr to the process grid */
   struct HPL_S_palg   * algo;           /* ptr to the algo parameters */
   struct HPL_S_pmat   * pmat;         /* ptr to the local array info */
   double              * A;              /* ptr to trailing part of A */
   double              * WORK;                          /* work space */
   double              * L2;                              /* ptr to L */
   double              * L1;       /* ptr to jb x jb upper block of A */
   double              * DPIV;    /* ptr to replicated jb pivot array */
   double              * DINFO;      /* ptr to replicated scalar info */
   double              * U;                               /* ptr to U */
   int                 * IWORK;     /* integer workspace for swapping */
   void                * * * buffers[2];   /* buffers for panel bcast */
   int                 counts [2];          /* counts for panel bcast */
   MPI_Datatype        dtypes [2];      /* data types for panel bcast */
   MPI_Request         request[1];        /* requests for panel bcast */
   MPI_Status          status [1];          /* status for panel bcast */
   int                 nb;             /* distribution blocking factor */
   int                 jb;                             /* panel width */
   int                 m;   /* global # of rows of trailing part of A */
   int                 n;   /* global # of cols of trailing part of A */
   int                 ia;  /* global row index of trailing part of A */
   int                 ja;  /* global col index of trailing part of A */
   int                 mp;   /* local # of rows of trailing part of A */
   int                 nq;   /* local # of cols of trailing part of A */
   int                 ii;   /* local row index of trailing part of A */
   int                 jj;   /* local col index of trailing part of A */
   int                 lda;           /* local leading dim of array A */
   int                 prow;  /* proc. row owning 1st row of trail. A */
   int                 pcol;  /* proc. col owning 1st col of trail. A */
   int                 msgid;           /* message id for panel bcast */
   int                 ldl2;         /* local leading dim of array L2 */
   int                 len;      /* length of the buffer to broadcast */
   int                 lwork;      /* total length of the WORK buffer */
} HPL_T_panel;
```

(a) Definition of the panel structure.

```
#define    HPL_PTR( ptr_, al_ ) \
    ( ( ( (size_t)(ptr_)+(al_)-1 ) / (al_) ) * (al_) )
// [...]
PANEL->WORK  = (void *)malloc( (size_t)(lwork) * sizeof(double) );
// [...]
PANEL->L2    = (double *)HPL_PTR( PANEL->WORK, dalign );
PANEL->ldl2  = Mmax( 1, ml2 );
PANEL->L1    = PANEL->L2 + ml2 * JB;
PANEL->DPIV  = PANEL->L1   + JB * JB;
PANEL->DINFO = PANEL->DPIV + JB;     *(PANEL->DINFO) = 0.0;
PANEL->U     = ( nprow > 1 ? PANEL->DINFO + 1 : NULL );
```

(b) Sample of the panel structure initialization.

Figure 3.4.: HPL panel data structure.

simulation of the application. The sequential time complexity of HPL's factorization is

$$\text{flop}(N) = \frac{2}{3}N^3 + 2N^2 + \mathcal{O}(N)$$

where N is the order of the matrix to factorize. The parallel time complexity can be approximated by

$$T(N) \approx \frac{\left(\frac{2}{3}N^3 + 2N^2\right)}{P \cdot Q \cdot w} + \Theta((P+Q) \cdot N^2)$$

where w is the flop rate of a single node and the second term corresponds to the communication overhead which is influenced by the network capacity and the previously listed parameters (`RFACT`, `SWAP`, `BCAST`, `DEPTH`, ...) and is very difficult to predict.

3.2 Typical runs on a supercomputer

Although the Top500 reports precise information about the core count, the peak performance and the effective performance, it provides almost no information on how (software versions, HPL parameters, etc.) this performance was achieved. Some colleagues agreed to provide us with the HPL configuration they used and the output they submitted for ranking (see Table3.1). In June 2013, the Stampede supercomputer at TACC was ranked 6th in the Top500 by achieving $5168.1\,\text{TFlop}\,\text{s}^{-1}$. In November 2017, the Theta supercomputer at ANL was ranked 18th with a performance of $5884.6\,\text{TFlop}\,\text{s}^{-1}$ but required a 28-hour run on the whole machine. Finally, we ran HPL ourselves on a Grid'5000 cluster named `dahu` whose software stack could be fully controlled.

Table 3.1.: Typical runs of HPL.

	Stampede@TACC	Theta@ANL	Dahu@G5K
`Rpeak`	$8520.1\,\text{TFlop}\,\text{s}^{-1}$	$9627.2\,\text{TFlop}\,\text{s}^{-1}$	$62.26\,\text{TFlop}\,\text{s}^{-1}$
N	3,875,000	8,360,352	500,000
`NB`	1024	336	128
$P \times Q$	77×78	32×101	32×32
`RFACT`	Crout	Left	Right
`SWAP`	Binary-exch.	Binary-exch.	Binary-exch.
`BCAST`	Long modified	2 Ring modified	2 Ring
`DEPTH`	0	0	1
`Rmax`	$5168.1\,\text{TFlop}\,\text{s}^{-1}$	$5884.6\,\text{TFlop}\,\text{s}^{-1}$	$24.55\,\text{TFlop}\,\text{s}^{-1}$
Duration	2 hours	28 hours	1 hour
Memory	120 TB	559 TB	2 TB
MPI ranks	1/node	1/node	1/core

The performance typically achieved by supercomputers (`Rmax`) needs to be compared to the much larger peak performance (`Rpeak`). This difference can be attributed to the node usage, to the MPI library, to the network topology that may be unable to deal with the intense communication workload, to load imbalance among nodes (e.g. due to a defect, system noise, ...), to the algorithmic structure of HPL, etc. All these factors make it difficult to know precisely what performance to expect without running the application at scale. It is clear that due to the level of complexity of both HPL and the underlying hardware, simple performance models (analytic expressions based on N, P, Q and estimations of platform characteristics) may be able to provide trends but can by no means accurately predict the performance for each configuration (e.g. consider the exact effect of HPL's six different broadcast algorithms on network contention). Additionally, these expressions do not allow engineers to improve the performance through actively identifying performance bottlenecks. For complex optimizations such as partially non-blocking collective communication algorithms intertwined with computations, a very faithful modeling of both the application and the platform is required.

One goal of this thesis is to simulate systems at the scale of Stampede. Given the scale of this scenario (3785 steps on 6006 nodes in two hours), detailed simulations quickly become intractable without significant effort. The simulation of HPL therefore comes with at least two challenges:

- The time-complexity of the algorithm is $\Theta(N^3)$ and $\Theta(N^2)$ communications are performed, with N being very large. The execution on the Stampede cluster took roughly two hours on 6006 compute nodes. Using only a single node, a naive emulation of HPL at the scale of the Stampede run would take about 500 days if perfect scaling was reached.
- The execution of HPL at scale leads to enormous memory consumption. It is therefore not possible to conduct executions at scale on a single or even a few compute nodes.

4 Simulating HPL at large scale

In this chapter, we present the changes to SimGrid and HPL that were required for a scalable simulation. The experiments were done using a single core from nodes of the Nova cluster provided by the Grid'5000 testbed [Bal+13] (32 GB of RAM, two 8-core Intel Xeon E5-2620 v4 CPUs, Debian Stretch OS (Linux 4.9)).

4.1 Speeding Up the Emulation

4.1.1 Compute Kernel Modeling

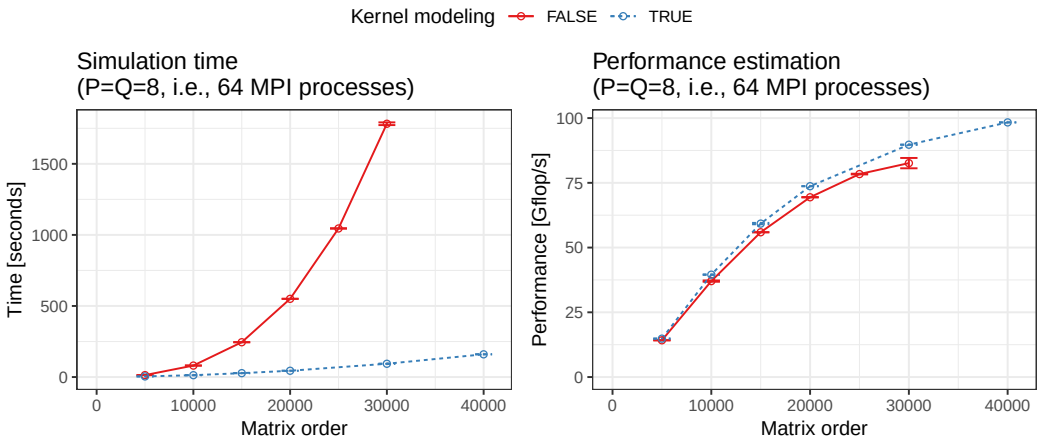
HPL heavily relies on BLAS kernels such as `dgemm` (for matrix-matrix multiplication) or `dtrsm` (for solving an $A \cdot x = b$ equation). The analysis of an HPL execution with 64 processes and a very small matrix of order $30{,}000$ showed that about $96\,\%$ of the time is spent in these two kernels. Since the output of these kernels does not influence the control flow, simulation time can be reduced by substituting `dgemm` and `dtrsm` function calls with a performance model of the respective kernel.

Skipping kernels renders the content of some variables invalid but in simulation, only the performance behavior of the application and not the correctness of computation results are of concern. Figure 4.1(a) shows an example of this macro-based mechanism that allows to keep HPL code modifications to an absolute minimum. The `1.029e-11` value represents the inverse of the flop rate for this compute kernel and was obtained through benchmarking. The estimated time of the kernel is calculated based on the given parameters and passed on to `smpi_execute_benched` which advances the clock of the executing process by this estimate. The effect on the simulation time for a small scenario is depicted in Figure 4.1(b). This modification speeds up the simulation by orders of magnitude. The impact on the accuracy of the simulation will be investigated in more details in the next chapter, but it can already be observed that this simple kernel model leads to a sound, albeit slightly more optimistic, estimation of the execution time.

In addition to the main compute kernels, a profiling of the code made it possible to identify seven other BLAS functions as computationally expensive enough to justify a

```
#define HPL_dgemm(layout, TransA, TransB, M, N, K,      \
        alpha, A, lda, B, ldb, beta, C, ldc) ({         \
    double size = ((double)M)*((double)N)*((double)K);  \
    double expected_time = 1.029e-11*size + 1.981e-12;  \
    smpi_execute_benched(expected_time);                \
})
```

(a) Non-intrusive macro replacement with a very simple computation model.



(b) Gain in terms of simulation time.

Figure 4.1.: Replacing the calls to computationally expensive functions by a model allows to emulate HPL at a larger scale.

specific handling: `dgemv`, `dswap`, `daxpy`, `dscal`, `dtrsv`, `dger` and `idamax`. Similarly, a significant amount of time was spent in fifteen functions implemented in HPL: `HPL_dlaswp*N`, `HPL_dlaswp*T`, `HPL_dlacpy` and `HPL_dlatcpy`. All these functions are called during the LU factorization and hence impact the performance of the HPL execution; however, because of the removal of the `dgemm` and `dtrsm` computations, they all operate on invalid data and hence also produce invalid data. They have been handled similarly to `dgemm` and `dtrsm`, through performance models and macro substitution, which speeds up the simulation by an additional factor of 3 to 4 on small ($N = 30{,}000$) and even more on larger scenarios.

4.1.2 Specific Adjustments

HPL uses pseudo-randomly generated matrices that are set up for every execution. This initialization, just like the factorization correctness verification at the end of the run, is not considered in the reported performance and can therefore be safely skipped. Note that HPL implements an LU factorization with partial pivoting, which requires a special treatment of the `idamax` function that returns the index of the first

element equaling the maximum absolute value. Although the cost of this function was ignored as well, its return value has been set to a random (but controlled) value to make the simulation unbiased (but fully deterministic).

4.2 Scaling Down Memory Consumption

The largest two allocated data structures in HPL are the input matrix `A` (with a size of typically several GB per process) and the `panel` which contains information about the sub-matrix currently being factorized. Unfortunately, when simulating an application with SMPI, all MPI processes are run within the same simulation process on a single node and the memory consumption of the simulation can therefore quickly reach several TB of RAM. Yet, as we no longer operate on real data, storing the whole input matrix A is needless. However, since only a minimal portion of the code was modified, some functions may still read or write some parts of the matrix. It is thus not possible to simply remove the memory allocations of large data structures. SMPI provides the `SMPI_SHARED_MALLOC` (`SMPI_SHARED_FREE`) macro to replace calls to `malloc` (`free`). They indicate that some data structures can safely be shared between processes and that the data they contain is not critical for the execution (e.g. an input matrix) and that it may even be overwritten. `SMPI_SHARED_MALLOC` works as follows (see Figure 4.2): a single block of physical memory (of default size 1 MB) for the whole execution is allocated and shared by all MPI processes. A range of virtual addresses corresponding to a specified size is reserved and cyclically mapped onto the previously obtained physical address. This mechanism allows most applications to obtain a nearly constant memory footprint, regardless of the size of the actual allocations.

Although using the default `SHARED_MALLOC` mechanism works flawlessly for the main matrix `A`, a more careful strategy needs to be used for the `panel`, the complex data structure presented in Figure 3.4. As already discussed, HPL flattens the whole structure into a single allocation (including matrix parts, but also integer indices) to optimize data transfers (illustrated in Figure 4.3(a)). Using a fully shared memory allocation for the `panel` therefore leads to index corruption that results in classic invalid memory accesses. Since `ints` and `doubles` are stored in non-contiguous parts of this flat allocation, it is therefore essential to have a mechanism that preserves the process-specific content. We have thus introduced the `SMPI_PARTIAL_SHARED_MALLOC` macro that allows us to specify which ranges of the allocation should be preserved (i.e. are private to each process) and which ones may be corrupted (i.e. are shared between processes). For a matrix of order $40,000$ and

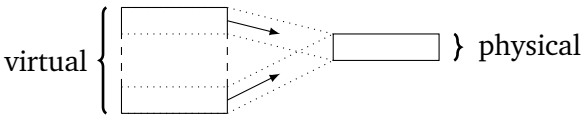
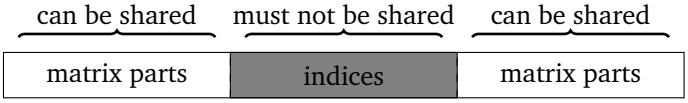
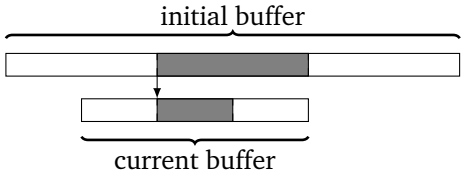


Figure 4.2.: SMPI shared malloc mechanism: large area of virtual memory are mapped onto the same physical pages.



(a) Structure of the panel in HPL.



(b) Reusing panel allocation from an iteration to another.

Figure 4.3.: Panel structure and allocation strategy.

64 MPI processes, memory consumption decreases with this approach from about 13.5 GB to less than 40 MB.

Another HPL specific optimization is related to the systematic allocation and deallocation of panels in each iteration, with the size of the panel strictly decreasing from iteration to iteration. As explained above, the partial sharing of panels requires many calls to `mmap` and introduces an overhead that makes these repeated allocations / frees a bottleneck. Since the very first allocation can fit all subsequent panels, we modified this allocation mechanism so that SMPI can reuse panels as much as possible from an iteration to another (see Figure 4.3(b)). Even for a very small matrix of order 40,000 and 64 MPI processes, the simulation time decreases from 20.5 sec to 16.5 sec. The number of page faults decreased from 2 million to 0.2 million, confirming the devastating effect these allocations/deallocations would have at scale.

The next three optimizations are not specific to HPL. We leveraged the information on which memory area is private, shared or partially shared to improve the overall performance. By making SMPI internally aware of the memory's visibility, it can now avoid calling `memcopy` when large messages containing shared segments are sent from one MPI rank to another. For fully private or partially shared segments, SMPI identifies and copies only those parts that are process-dependent (private) into the corresponding buffers on the receiver side. HPL simulation times and memory consumption were considerably improved in our experiments because the `panel` is

the most frequently transferred data structure but only a small part of it is actually private.

As explained above, SMPI maps MPI processes to threads of a single process, effectively folding them into the same address space. Consequently, global variables in the MPI application are shared between threads unless these variables are *privatized* and the simulated MPI ranks thus isolated from each other. Several technical solutions are possible to handle this issue [Deg+17]. The default strategy in SMPI consists in making a copy of the `data` segment (containing all global variables) per MPI rank at startup and, when context switching to another rank, to remap the `data` segment via `mmap` to the private copy of that rank. SMPI also implements another mechanism relying on the `dlopen` function that allows to load several times the `data` segment in memory and to avoid costly calls to `mmap` (and subsequent cache flush) when context switching. For a matrix of order $80{,}000$ and 32 MPI processes, the number of minor page faults drops from $4{,}412{,}047$ (with `mmap`) to 6880 (with `dlopen`), which results in a reduction of system time from $10.64\,\mathrm{sec}$ (out of $51.47\,\mathrm{sec}$) to $2.12\,\mathrm{sec}$.

Finally, for larger matrix orders (i.e. N larger than a few hundred thousands), the performance of the simulation quickly deteriorates as the memory consumption rises rapidly. Indeed, folding the memory reduces the *physical* memory usage. The *virtual* memory, on the other hand, is still allocated for every process since the allocation calls are still executed. Without a reduction of allocated virtual addresses, the page table rapidly becomes too large for a single node. Thankfully, the x86-64 architecture supports several page sizes, such as the *huge pages* in Linux. Typically, these pages are around $2\,\mathrm{MiB}$ (instead of $4\,\mathrm{KiB}$), which reduces drastically the page table size. For example, for a matrix of order $N = 4,000,000$, it shrinks from $250\,\mathrm{GB}$ to $0.488\,\mathrm{GB}$.

4.3 Scalability Evaluation

The main goal of the previous optimizations is to reduce the complexity from $\Theta(N^3) + \Theta(N^2 \cdot P \cdot Q)$ to something more reasonable. The $\Theta(N^3)$ was removed by skipping most computations. Ideally, since there are N/NB iterations (steps), the complexity of simulating one step should be decreased to something independent of N. SimGrid's fluid network models, used to simulate communications, do not depend on N. Therefore, the time to simulate a step of HPL should mostly depend on P and Q. Yet, some memory operations on the panel that are related to pivoting are intertwined in HPL with collective communications, meaning that it is impossible to get rid of the $\mathcal{O}(N)$ complexity without modifying HPL more extensively.

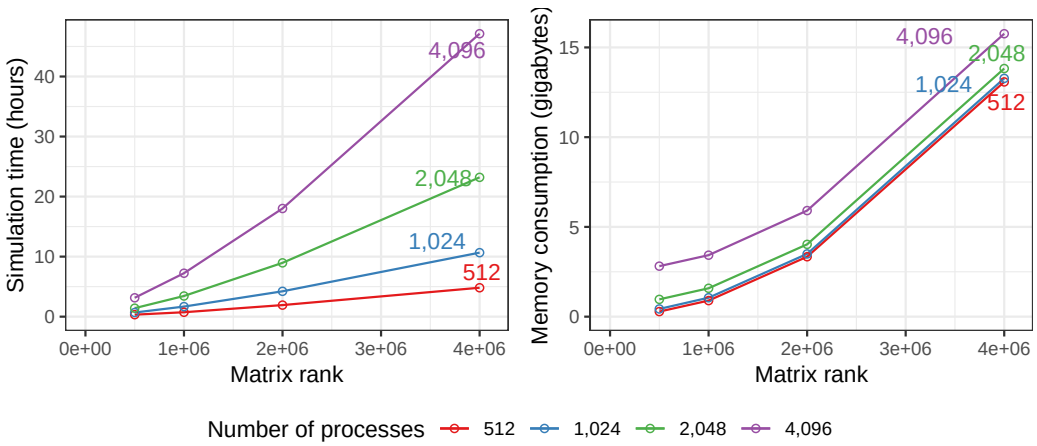


Figure 4.4.: Time complexity and memory consumption are linear in the number of processes but remain mildly quadratic with matrix rank.

To evaluate the efficiency of our proposal, we conduct a first evaluation on a non-existing but Stampede-like platform comprising 4,096 nodes interconnected through a fat-tree topology. We run simulations with 512, 1024, 2048 or 4096 MPI ranks and with matrices of orders 5×10^5, 1×10^6, 2×10^6 or 4×10^6. All other HPL parameters are similar to the ones of the original Stampede scenario. The impact of the matrix order on total execution time and memory is illustrated in Figure 4.4. With all previously described optimizations enabled, the longest simulation took close to 47 hours and consumed $16\,\mathrm{GB}$ of memory whereas the shortest one took 20 minutes and $282\,\mathrm{MB}$ of memory.

5 Modeling HPL kernels and communications

As explained in chapter 4, HPL spends most of its computation time in a dozen specific functions for which a performance model has to be designed. Most compute kernels have several parameters from which a very simple model can generally easily be identified (e.g. proportional to the product of the parameters) but refinements including the individual contribution of each parameter as well as the spatial and temporal variability of the operation are also possible. Likewise, communications between two nodes are mostly linear in message size, but the actual performance can vary wildly depending on the range of the message size as MPI switches from one protocol to another based on message size thresholds. In this chapter we first introduce some notations to describe the complexity of the models we have investigated, then we define these models and explain how we can instantiate them.

5.1 Modeling notations

We denote as T the duration of an operation with parameters M, N, K (in the case of the `dgemm` operation, these parameters describe the geometry of the input matrices). We first consider the three following modeling options:

- Modeling option $\boldsymbol{M-0}$: For simple compute kernels, the duration can be modeled as a constant duration independent of the input parameters, i.e. $T \sim \alpha$, where α is estimated through the sample average of the duration of the operation (or simply 0 if the kernel is negligible).
- Modeling option $\boldsymbol{M-1}$: A simple combination of the parameters (e.g. $S = M.N.K$) may be the primary factor driving the performance of the operation. Then $T \sim \alpha.S\ (+\beta)$ and α and β can be estimated through a classical least-square linear regression.
- Modeling option $\boldsymbol{M-2}$: When the behavior of the operation is complex or requires a faithful modeling over the full range of input parameters, a full

polynomial model is required, i.e. $T \sim \alpha.M.N.K + \beta.M.N + \gamma.N.P + \dots$ Again, the $\alpha, \beta, \gamma, \dots$ can be estimated through a classical least-square linear regression.

There are two situations where more elaborate variations need to be considered:

- Whenever the platform is slightly heterogeneous (*spatial variability*), the previous models should be built for each host individually. This modeling option is denoted $\boldsymbol{M}_H$.

- The behavior of the operation may be mostly linear but only for specific parameter ranges. This is for example the case for networking operations or for computing nodes on Stampede where Intel's Math Kernel Library (MKL) uses the Xeon Phi accelerator only when the input is large enough to compensate for the data transfer. In such situations, the models considered will be piecewise linear,

$$\text{e.g., } T \sim \begin{cases} \text{if } M < \theta_1 & \alpha_1.M + \beta_1 \\ \text{else if } M < \theta_2 & \alpha_2.M + \beta_2 \\ \dots \end{cases},$$

  where the θ, α, β should all be estimated. This kind of model is denoted $\boldsymbol{M'}$.

All previous models can be fit with relatively simple linear regressions or maximum likelihood learning methods. However, an important hypothesis underlying all these methods is the homoscedasticity, i.e. that the variability is independent on the parameters.

The residual (*temporal*) variability may be an important phenomenon to account for, as "system noise" is known to be detrimental to the overall performance of parallel applications like HPL. We thus consider different modeling options for this temporal variability:

- Noise option $\boldsymbol{N}-0$ (no noise): This is the simplest option. It consists in injecting the value predicted by the model

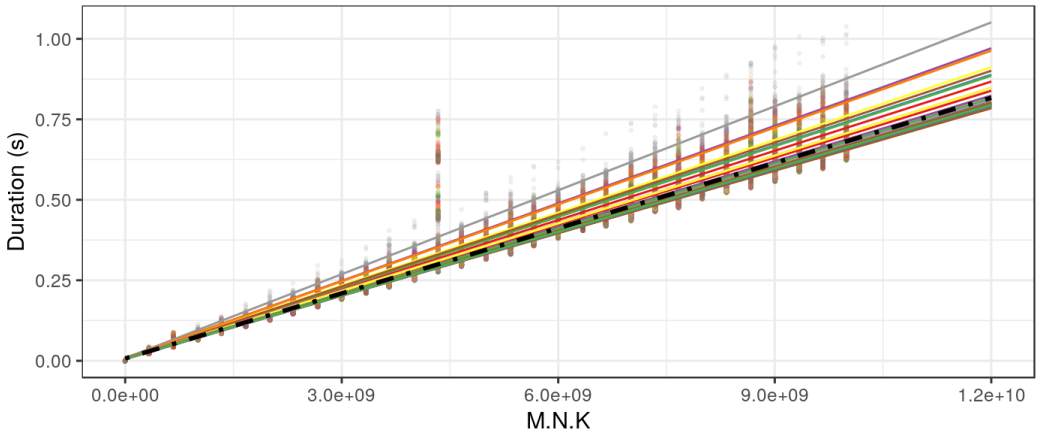
- Noise option $\boldsymbol{N}-1$ (homoscedastic): The simplest probability family to model variability is the normal distribution, hence $T \sim \boldsymbol{M}(M, N, K) + \mathcal{N}(0, \sigma^2)$, where σ^2 is the sample variance of the model residuals.

- Noise option $\boldsymbol{N}-2$ (heteroscedastic): The conditional variance of the residuals (i.e. σ^2 given M, N, K) is modeled by a polynomial function of the input parameters.

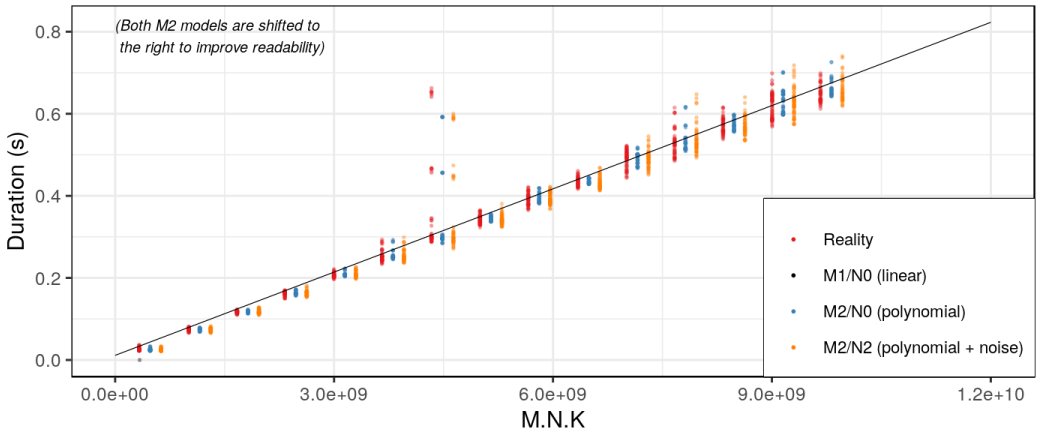
Finally, even the sophisticated normal distribution from $N-2$ may be too simple to describe the noise observed on real platforms. It may be common to have a few operations being one order of magnitude slower than all the other ones even though they had the same parameter set. In this case, a reasonable option consists in modeling noise with a mixture of normal distributions whose parameters $\pi_1, \ldots, \pi_k$ should be estimated. We denote this kind of model as N'. Likewise, the per-host estimations are denoted by N_H.

5.2 Modeling the CPU (i.e. `dgemm` function)

5.2.1 Different linear models



(a) `dgemm` heterogeneity



(b) `dgemm` model

Figure 5.1.: Illustrating the realism of modeling for `dgemm` function.

HPL spends the most time in the `dgemm` kernel, it is therefore of extreme importance to model this function very faithfully. We evaluated the previous modeling alternatives: $\boldsymbol{M}-\{1,2\}$ $\boldsymbol{N}-\{0,1,2\}$ and $\boldsymbol{M}_H-\{1,2\}$ $\boldsymbol{N}_H-\{0,1,2\}$. The $\boldsymbol{M'}$ and $\boldsymbol{N'}$ families were not investigated as nothing in our observations called for such complexity on classical multi-core machines. Figure 5.1 illustrates various models and their respective quality for the `dgemm` function. In these figures, the performance of `dgemm` is evaluated by calling `dgemm` with randomized input sizes over all the cores of each node (to reproduce experimental conditions similar to the one of HPL). The first observation (Figure 5.1(a)) is that a few nodes exhibit quite a different behavior (each color and each regression line under model $\boldsymbol{M}_H-1$ corresponds to a different CPU, whereas the black dotted line corresponds to model $\boldsymbol{M}-1$ over all the nodes). These nodes will systematically be slightly slower than other nodes and accounting for this spatial heterogeneity is likely to be rather important for simulating the execution of HPL. Second, we took care of covering a wide variety of combinations for M, N, and K. It can be observed that $M.N.K$ is not sufficient to describe correctly the performance of `dgemm`, even though the number of operations performed by this function is expected to be proportional to $M.N.K$. Indeed, for $M.N.K \approx 4.5 \times 10^9$ some duration are systematically higher regardless of the node. This happens for some particular (e.g., tall and skinny) matrix geometries, which strongly suggests using the full polynomial model. Figure 5.1(b) depicts the performance (red dots) of a given node as well as the prediction using a simple linear model ($\boldsymbol{M}_H-1$, black line), a full polynomial model ($\boldsymbol{M}_H-2$, blue dots) and a full polynomial model with heteroscedastic noise ($\boldsymbol{M}_H-2$ $\boldsymbol{N}_H-2$, orange dots). A close inspection reveals that all experimental variability is actually very well explained by both the polynomial model (better fit for particular parameter combinations) and some temporal variability.

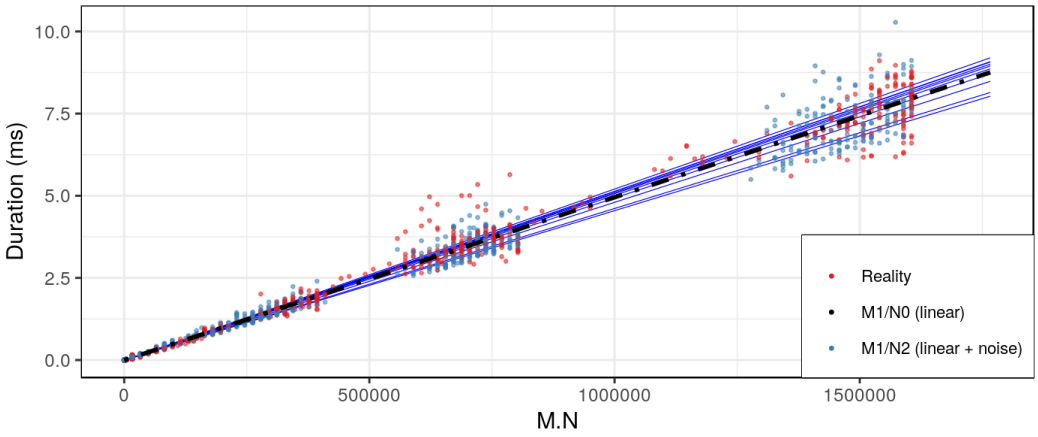


Figure 5.2.: Illustrating the realism of modeling for `HPL_dlatcpy` function.

Four other BLAS kernels and a few other very small HPL compute kernels (often related to memory management) are deeply intertwined with collective operations to allow HPL to be as efficient as possible. Although the total duration of these kernels is minimal compared to the total execution time, they may perturb collective communication by introducing late sends and receives. The behavior of one of these kernels is illustrated in Figure 5.2. This kind of data can only be obtained by running HPL for a small input matrix over each node individually. Again, for all these kernels a single parameter combination explains most of the performance and there is some variability from one node to another (one blue regression line per CPU) but it remains quite limited (black dotted line for the platform as a whole), especially since these kernels are very short and infrequently called compared to `dgemm`. Finally, since variability significantly increases with the value of the input parameters, a $\boldsymbol{N}-2$ model is clearly required. The blue dots in Figure 5.2 represent the outcome of a $\boldsymbol{M}-1$ $\boldsymbol{N}-2$ model and are hardly distinguishable from the real behavior. Similar results can be obtained with this category of model for all other kernels.

5.2.2 Bayesian modeling and generative models

In Chapter 6, we will show that the model $\boldsymbol{M}_H-2$ $\boldsymbol{N}_H-2$ is required for the `dgemm` kernel to make reliable performance predictions for HPL. We recall here the meaning of this notation:

$$\text{For each processor } p,\ \text{dgemm}_p(M,N,K) \sim \mathcal{H}(\mu_p,\sigma_p)$$
$$\begin{cases} \mu_p &= \alpha_p MNK + \beta_p MN + \gamma_p MK + \delta_p NK + \epsilon_p \\ \sigma_p &= \omega_p MNK + \psi_p MN + \phi_p MK + \tau_p NK + \rho_p \end{cases}, \tag{5.1}$$

where $\mathcal{H}(\mu,\sigma)$ denotes a random variable that follows a half-normal distribution with parameters μ,σ accounting for the expectation and the standard deviation. The dependency on p allows accounting for platform heterogeneity (since $\alpha_p,\beta_p,\ldots,\rho_p$ can be specific to each node), i.e. the aforementioned spatial variability. The σ_p parameter allows accounting for (short-term) temporal variability, i.e. to model the fact that the duration of two successive calls to `dgemm` with the same parameters M,N,K are never identical. Finally, we initially decided to use a half-normal distribution instead of a normal distribution to account for the asymmetry of the observed durations (e.g. there are no negative durations), but with hindsight we do not believe this choice to be very important.

As we will show, the performance of nodes exhibits several kinds of variability: i) a spatial variability (between nodes) ii) a "short-term" temporal variability (the one experienced within an HPL run) but also iii) a "long term" temporal variability (from one day to another). As illustrated in Chapter 6, accounting for the first two kinds of variability is essential but during our investigation of the simulation validity, which spanned several months, we also had to deal with the fact that the node performance from one day to another could significantly vary, thereby making our comparisons between a real experiment and the simulation driven by models derived from past measurements sometimes irrelevant.

This section explains how all sources of variability can be accounted for in a single unified model. This will mainly be used in Chapter 7. From our observations, we assume that on a given node p and a given day d, the duration of the `dgemm` kernel can be modeled as follows:

$$\forall M,N,K, \mathsf{dgemm}_{p,d}(M,N,K) \sim \mathcal{H}(\alpha_{p,d}MNK + \beta_{p,d},\ \gamma_{p,d}MNK) \tag{5.2}$$

Compared to the model (5.1), this model includes the daily variability but drops the complexity of a full-fledged polynomial. Such complexity may be important whenever trying to model a particular platform. However, when performing sensibility analysis, a simpler model is preferred, especially as not all terms of the polynomial may be statistically significant. In this model, the short-term temporal variability stems from the $\gamma_{p,d}$ term while the average performance of the node stems from the $\alpha_{p,d}$ and $\beta_{p,d}$ terms, which we gather in a single 3-dimensional vector

$$\mu_{p,d} = (\alpha_{p,d}, \beta_{p,d}, \gamma_{p,d}). \tag{5.3}$$

Now, since every machine is unique it is natural to assume that for each machine:

$$\forall d, \mu_{p,d} \sim \mathcal{N}(\mu_p, \Sigma_T) \tag{5.4}$$

In this model, μ_p accounts for the average performance of the machine p, while Σ_T accounts for its day-to-day variability. From our observation we had no particular reason to assume that this variability was different from one machine to another, hence, Σ_T is not indexed by p but global to all machines. However, the parameters $\alpha_{p,d}, \beta_{p,d}, \gamma_{p,d}$ are generally correlated to each others, hence Σ_T is the full covariance matrix to account for interactions. The choice of a Normal distribution is natural since it is the simpler distribution that accounts for a specific mean and variance, but we will discuss its relevance later in this section.

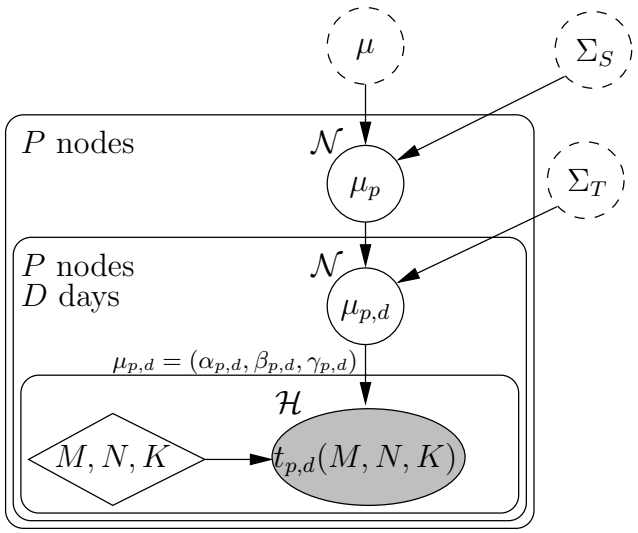


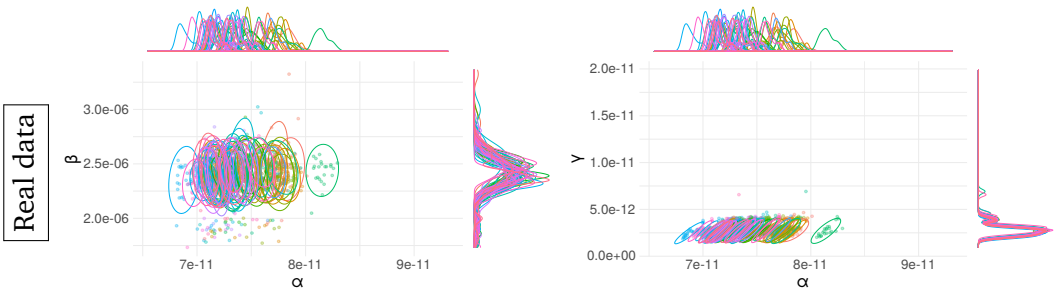
Figure 5.3.: Generative model of kernel duration accounting for the spatial (Σ_S), long-term (Σ_T) and short-term variability ($\gamma_{p,d}$). The shaded node represents observed variables and diamond node represents deterministic variables, while non-shaded nodes represent latent variables. The solid node is the variable which is estimated when conducting (in)validation studies while the dashed ones are useful when conducting sensibility analysis and extrapolating to a hypothetical cluster.

Finally, we need to account for the spatial variability, which we propose to model as follows:

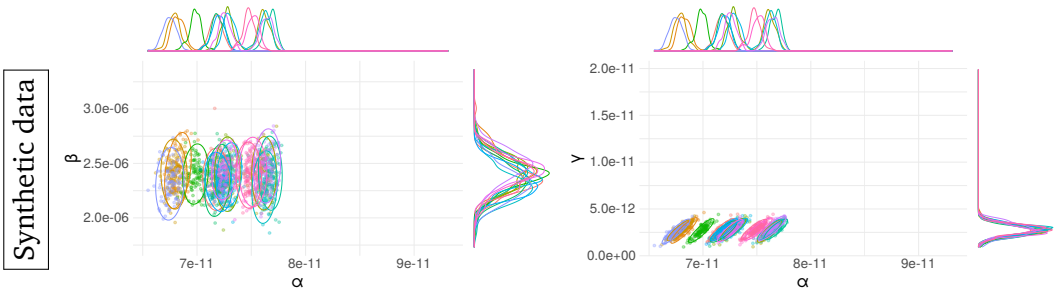
$$\forall p, \mu_p \sim \mathcal{N}(\mu, \Sigma_S) \tag{5.5}$$

Again, in such a model μ accounts for the machines' average performance while Σ_S accounts for the (weak) heterogeneity. This hierarchical model is depicted in Figure 5.3.

The relevance of model (5.2) will be discussed in Chapter 6, but the relevance of models (5.4) and (5.5) requires some attention. Figure 5.4(a) represent the empirical distribution of $\mu_{p,d} = (\alpha_{p,d}, \beta_{p,d}, \gamma_{p,d})$ (the result of the linear regression) for the 32 nodes of the `dahu` cluster on 40 different days from November 2019 to February 2020. The distribution for each node appears approximately normal and passed a Shapiro-Wilk normality test. Although the distribution of the $\beta_{p,d}$ appears slightly skewed toward larger values and one of the nodes (the one with the larger $\alpha_{p,d}$) stands out, there is no good reason for using a more complex distribution than a Gaussian one. Although the correlation between α, β, and γ is very weak, it appeared to be statistically significant (most ellipses are slightly tilted toward the North-East), hence a full variance matrix is needed (at least for Σ_T).

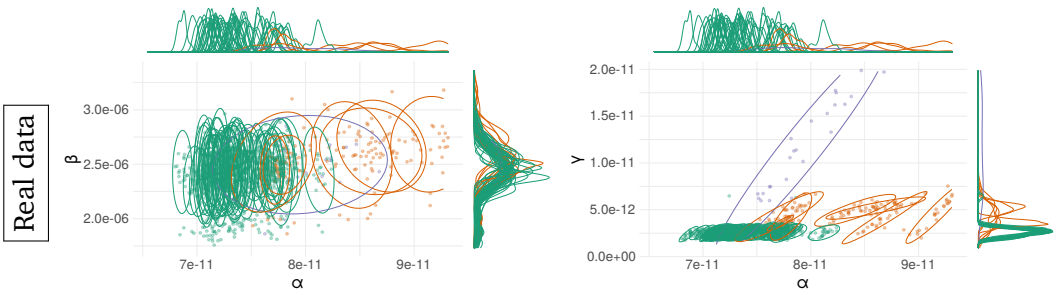


(a) Distribution of α, β, and γ (observations on 2×32 CPUs from November 2019 to February 2020).

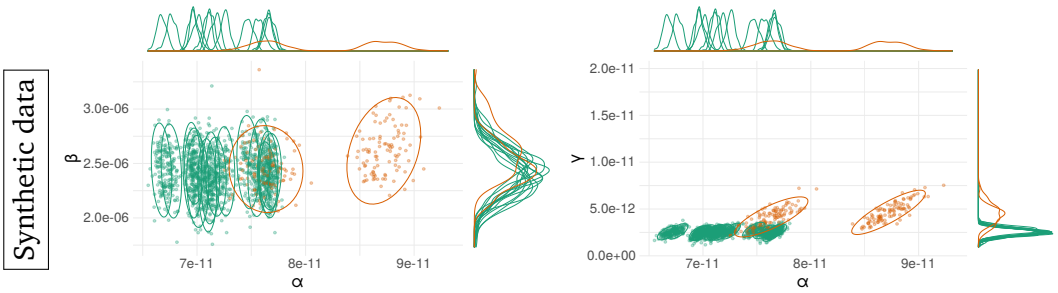


(b) Distribution of α, β, and γ (synthetic data for 16 CPUs).

Figure 5.4.: Distribution of the regression parameters for around 20 `dgemm` calibrations made on each of the 32 nodes. Each color/ellipse corresponds to a different CPU.



(a) Distribution of α, β, and γ (observations on 2×32 CPUs from October to November 2019).



(b) Distribution of α, β, and γ (synthetic data for 16 CPUs).

Figure 5.5.: Distribution of the regression parameters for around 20 `dgemm` calibrations made on each of the 32 nodes. 4 of these nodes had a cooling problem, leading to longer and more variable durations. Each color/ellipse corresponds to a different group of CPUs.

Given the above, it is easy to estimate μ_p and Σ_T by averaging over the $\mu_{p,d}$ of each node, and then to estimate μ and Σ_S by averaging over all the nodes. This moment-matching method is simple and provides very good estimates for μ, Σ_T, and Σ_S because we have enough measurements at our disposal and because it is particularly suited to the Gaussian modeling assumption. Should more complex models (e.g., a mixture to account for "outlier" nodes or a skew normal distribution to account for the distribution's skewness) be used, a general Bayesian sampling framework like STAN [Car+17] would be more adapted. Such frameworks allow to easily specify hierarchical generative models like the one presented in Figure 5.3 and to draw samples from the posterior distribution of μ, Σ_T, and Σ_S, which can be used to generate realistic $\mu_{p,d}$ values for a new hypothetical cluster easily.

Such a process is depicted in Figure 5.4(b) where hypothetical regression parameters for 16 nodes have been generated. Comparing such synthetic data with the original samples from Figures 5.4(a) allows us to evaluate the model's potential weaknesses. Although the orders of magnitude of all parameters and the ellipses are excellent, a few differences are visible. First, the variability of α_p seems a bit overestimated (the spread along the x-axis is larger). This can be explained by the fact that one of the nodes seemed to be significantly slower (with much larger α_p), which artificially increased the spatial variability. Second, as expected from a Gaussian model, the distributions of the $\beta_{p,d}$ are symmetrical whereas there was a slight negative skew in the original samples, but this should be of little significance for our study. The distributions of the $\gamma_{p,d}$ however are particularly realistic.

We also illustrate the generality of this model with the data from Figure 5.5(a). These measurements were obtained from October to November 2019 where the cluster was less stable and where some nodes particularly misbehaved. Three nodes (in orange, hence a total of 6 CPUs) are distinguished from the 28 others (in green) and have lower performance (higher values for α, β, and γ) and one node (in blue) is particularly unstable. Although this last node may be considered too abnormal to represent anything, it would be reasonable to assume that a larger cluster would present at least the two kinds of behaviors (green for stable nodes, and orange for slower nodes). The higher layer of the model in Figure 5.3 should then be replaced by a mixture of normal distributions (whose weights would then be sampled from a Dirichlet distribution). Again, hypothetical regression parameters for 16 CPUs have been generated with such a process on Figure 5.5(b) and are very similar to the original measurements.

Overall this model is therefore of excellent quality and can be used to generate large configurations very easily and evaluate the influence of different kinds of variability on the performance of HPL.

5.2.3 Conclusion

We have described in Section 5.2.1 different models with various levels of complexity, from the most basic linear, homogeneous and deterministic model to more elaborated ones with heterogeneity and random noise. The objective of Chapter 6 will be to discuss the required level of complexity and (in)validate the quality of these models for faithful performance predictions of HPL.

Then in Section 5.2.2 we have presented a method for generating new model instances based on observed data. This allows to *extrapolate* a given cluster to a larger one or even to modify some characteristics of the model such as the temporal or spatial variability. This will be used in Chapter 7 for sensibility analysis.

Finally, some of the tests developed in Chapter 11 will use the regression coefficients from Section 5.2.1.

5.3 Modeling the network

5.3.1 Modelization in Simgrid

Prior to this work, the standard way of accounting for protocol changes in SMPI was to estimate breakpoints visually and to conduct a linear regression for each range. The expected duration was then used directly in the simulation with no particular effort with respect to the temporal variability ($\boldsymbol{M'}-1\ \boldsymbol{N}-0$). Yet, as illustrated in Figure 5.6, the variability of high speed networks is quite particular, for some intervals of message sizes the observed durations have several modes. These modes happen homogeneously during the calibration experiment, they are not caused by a transient perturbation of the system.

The ideal model for such observed data would be $\boldsymbol{M'}-1\ \boldsymbol{N'}-1$, several intervals of message sizes with abrupt breakpoints, and a noise distributed as a Gaussian mixture for each range. Such temporal variability could explain some (overall poor) performance since they generally get amplified by broadcast and pipelined communication patterns.

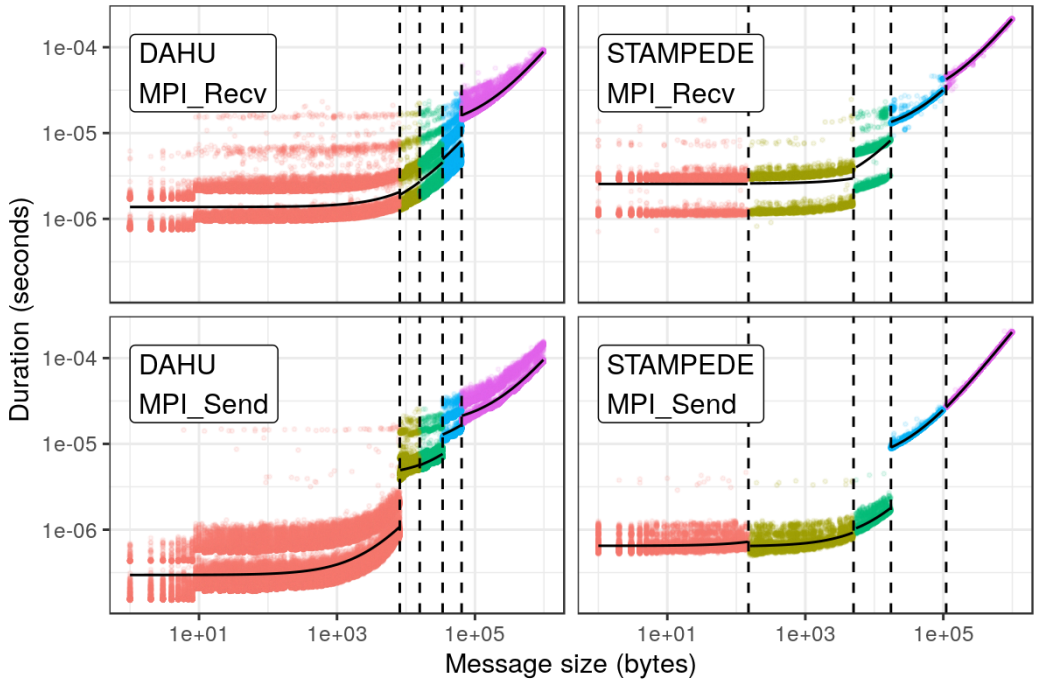


Figure 5.6.: Illustrating piecewise linearity and temporal variability of high-speed communications on two systems.

Automatically computing the best fit for a complex model like the aforementioned one is much more difficult than for the `dgemm` model proposed in Section 5.3. It should be possible to compute it in one go with a Bayesian framework like STAN, but it would likely be computationally prohibitive. A more realistic solution is to perform two steps: (1) average the durations per message size and find the breakpoints using this aggregated data (i.e. fit a $\boldsymbol{M'}-1$ $\boldsymbol{N}-1$ model), and (2) on each segment, compute the linear regression with a noise mixture (i.e. fit a $\boldsymbol{M}-1$ $\boldsymbol{N'}-1$ model).

1. We have tested two different solutions for computing a piecewise linear regression, a Python package named `datadog-piecewise` [Dat], and an R package named `cubist` [KQ20]. Although both of them work perfectly on the simplest generated datasets, they did not produce satisfying results with our real data. The likely reason is both the exponential distribution of the predictor variable and the heteroscedasticity of the noise. We decided to implement our own solution, named `pycewise`, which is detailed in Section 5.3.3 and compared to both `datadog-piecewise` and `cubist`.

2. We tested an existing R package named `flexmix` for computing a linear regression with a Gaussian mixture noise [GL08]. It proved to be fragile, the result being non-deterministic: the number of modes found by the algorithm was

not constant. We ended up clustering the data manually via visual inspection and then computing a simple linear regression for each mode.

5.3.2 Finding semantic breakpoints

In network calibration experiments, i.e. the experiments carried to measure the network performance and instantiate a model, we can observe sudden performance changes, a.k.a. breakpoints (see Figure 5.6). As discussed previously, these breakpoints happen because of a protocol change in MPI implementations. Some of these changes also affect the semantic of the `MPI_Send` operation: we described in Section 2.2 three communication modes, *asynchronous*, *detached* and *asynchronous*. This section presents how to detect automatically such semantic breakpoints. In this section as well as Section 5.3.3, MPI is seen as a black box and we use reverse-engineering techniques to find a suitable model.

This work was done in collaboration with several Simgrid collaborators, we wish to thank them for the ideas and the code prototype.

We define two MPI codes, Algorithm 3 and Algorithm 4. They both take in input a message size `N`. We expect Algorithm 3 to produce a deadlock and not print any message when `N` is large enough (the communication was *synchronous*) while it should print the message "OK" for smaller values of `N`. Likewise, we expect Algorithm 4 to have corrupted data and print the message "CORRUPTED" when `N` is large enough (the communication was not *asynchronous*) while it should print the message "OK" for smaller values of `N`.

Algorithm 3: Prints "OK" if the communication for size N is not synchronous.

Function `deadlocktest(`*N*`)`**:**

```
MPI_Comm_rank(&myself)
other = 1-myself
buff = {0, ..., 0}
MPI_Send (buff, N, other)
MPI_Recv (buff, N, other)
print("OK")
```

We repeatedly call the Algorithms 3 and 4, performing a binary search on the argument `N` (we set a timeout of $3\,\mathrm{s}$ to kill Algorithms 3 in case of deadlock). This allows to find the two semantic breakpoints easily:

- In *asynchronous* mode, we have no deadlock and no corruption.

Algorithm 4: Prints "OK" if the communication for size N is asynchronous.

```
Function corruptiontest(N):
    MPI_Comm_rank(&myself)
    other = 1-myself
    buff = {0, ..., 0}
    if myself = 0 then
        MPI_Isend (buff, N, other)
        for i = 1 to N − 1 do
            buff[i] = 1
    else
        sleep(1)
        MPI_Recv (buff, N, other)
        corrupted=False
        for i = 1 to N − 1 do
            if buff[i] = 1 then
                corrupted=True
        if corrupted then
            print("CORRUPTED")
        else
            print("OK")
```

- In *detached* mode, we have no deadlock, but we have corruption.
- In *synchronous* mode, we have corruption and a deadlock.

Note that in Algorithms 4, the function `MPI_Isend` is used instead of `MPI_Send`. Indeed, the MPI standard specifies that the buffer of function `MPI_Send` can be immediately reused once the function returns, this means that even if the function returns before the communication is actually performed, the data is copied in an auxiliary buffer to prevent any corruption. Since we rely on this data corruption to detect whether we are in *asynchronous* or *detached* mode, we need to use function `MPI_Isend` which does not make such guarantees. Here we suppose that the thresholds used in both functions are the same.

In our tests, we found that the three protocols did not always exist, and that the threshold will depend on the hardware. For instance, on the `dahu` cluster, the semantic breakpoints found by the two binary searches are equal. This means that the *detached* mode does not exist in the MPI implementation installed on this cluster (for a given size `N`, either we have both a deadlock with Algorithms 3 and a corruption with Algorithms 4, or we have none, there is no alternative combination). On clusters `paravance` and `grisou`, the search with Algorithm 4 does not find any breakpoint, the message is always corrupted. This means that the *asynchronous* mode does not exist.

These kinds of inconsistencies show the fragility of the semantic approach for detecting breakpoints. The three behaviors we described, namely *asynchronous*, *detached* and *synchronous*, are not part of the MPI standard. They highly depend on both the MPI implementation and the hardware. For instance, Algorithm 4 depends on the semantic of a concurrent memory read and write, which can produce surprising results on less common architectures. An alternative would be to instrument the memory of the program to have a precise timing of when the memory reads and writes occur.

5.3.3 Learning breakpoints

In this section, we present our solution for automatically computing a piecewise linear regression. The goal is to find performance breakpoints with a statistical approach and not a semantic approach as described in Section 5.3.2. This has been implemented as a Python package named `pycewise` [CL21d].

Model and notations

We define here the base assumptions and notations for our optimization problem. The predictor variables (e.g. the message size) will be denoted X, the response variables (e.g. the communication duration) will be denoted Y. We will denote ℓ the list of tuples (X, Y). We assume that there are N different unknown breakpoints $B_1 < \cdots < B_N$, where N itself is also unknown. For convenience, we will denote $B_0 = -\infty$ and $B_{N+1} = +\infty$. Then, there are $N+1$ tuples $(\alpha_i, \beta_i, \gamma_i, \delta_i)$ such that:

$$Y \sim \alpha_i X + \beta_i + \mathcal{N}(0, \gamma_i X + \delta_i) \qquad \text{if } B_i \leq X < B_{i+1} \tag{5.6}$$

Here, γ_i may be null (homoscedastic data) or non-null (heteroscedastic data). This hypothesis will be discussed later. Likewise, the distribution of the B_i will be discussed when comparing different solutions (no assumption is made yet).

A *solution* θ to the optimization problem consists in the list of breakpoints (B_i) and tuples (α_i, β_i). The values (γ_i, δ_i) are not computed here, but they could be estimated in a second phase. We will write f_θ the (deterministic) prediction function associated with the solution:

$$f_\theta(X) = \alpha_i X + \beta_i \qquad \text{if } B_i \leq X < B_{i+1} \tag{5.7}$$

Two functions are considered:

- $\text{RSS}(\ell, \theta) = \sum_{(X,Y)\in\ell}(Y - f_\theta(X))^2$ is the usual residual sum of squares.
- $\text{BIC}(\ell, \theta) = K\log(S) + S\log(\text{RSS}(\ell, \theta))$ where $S = |\ell|$ is the number of observations, $K = 3(N+1) + N = 4N + 3$ is the number of parameters in the model. This is the usual Bayesian information criterion.

The goal of the optimization problem is to minimize the BIC. The $K\log(S)$ term in this function is intended to penalize the model size and thus prevent overfitting.

By an abuse of notation, we will write $\text{BIC}(\ell)$ the BIC for the observations ℓ with a simple linear regression without breakpoint (i.e. $N = 0$). We will write $\text{BIC}(\ell_1 \oplus \ell_2)$ the BIC for the observations $\ell_1 \cup \ell_2$ with a single breakpoint between the two lists and a simple linear regression for each list (i.e. $N = 1$). The concatenation of lists will be denoted $\ell_1 \bullet \ell_2$, hence $\text{BIC}(\ell_1 \bullet \ell_2)$ will denote the BIC of a linear regression without breakpoint ($N = 0$).

Main algorithm

The algorithm proceeds in two steps: it starts by a top-down step where new breakpoints are greedily added, it finishes by a bottom-up step where some breakpoints are greedily removed to simplify the solution. The first step is inspired by the work of Malerba et al. [Mal+04].

In the top-down step (Algorithm 5), we recursively build a tree, where the inner nodes represent breakpoints and the leaves represent simple linear regressions. The function `topdown` takes as input a list ℓ of tuples (X, Y) sorted by increasing X and returns a list of sublists, each sublist representing an interval for the piecewise regression. On a given call, we iterate on the list to test all the X values as a possible breakpoint: we search for the X that minimizes the objective function. If the objective with the new breakpoint is smaller than the objective without breakpoint, then we recursively call the function on the two sub-lists.

In our implementation, each iteration of the `foreach` loop from Algorithm 5 is executed in a constant time. This is possible because the list ℓ is sorted by increasing X, no copies are made when splitting the list, and both the new linear regressions and the new BIC are computed with online algorithms (i.e. adding or removing one tuple (X, Y) from the sublists ℓ_1 and ℓ_2 does not require recomputing from scratch the new regression coefficients nor the BIC, it is possible to update the values from the previous iterations). In the next sections, we change the objective function and

Algorithm 5: Top-down step for computing the piecewise linear regression.

Function `topdown`(ℓ)**:**
- best_BIC $= +\infty$
- **foreach** $(X, Y) \in \ell$ **do**
  - split ℓ into ℓ_1 and ℓ_2 such that $\forall X' \in \ell_1, X' \leq X$ and $\forall X' \in \ell_2, X' > X$
  - new_BIC $=$ BIC$(\ell_1 \oplus \ell_2)$
  - **if** *new_BIC* $<$ *best_BIC* **then**
    - best_BIC $=$ new_BIC
    - best_solution $= (\ell_1, \ell_2)$
- **if** *new_BIC* $<$ *BIC*(ℓ) **then**
  - $(\ell_1, \ell_2) =$ best_solution
  - lists$_1$ = `topdown`(ℓ_1)
  - lists$_2$ = `topdown`(ℓ_2)
  - **return** *lists*$_1 \oplus$ *lists*$_2$
- **else**
  - **return** ℓ

have to drop this ability of doing an iteration in constant time. The effect on the computation duration will be discussed.

In the bottom-up step (Algorithm 6), adjacent intervals are greedily merged (thereby removing breakpoints), starting by fusions that decrease the least the RSS. All the different iterations of the loop are kept in memory, the merging operation is carried until there is no breakpoint left. In the end, we keep the iteration that minimizes the BIC.

Algorithm 6: Bottom-up step for computing the piecewise linear regression.

Function `bottomup`$(L = (\ell_1, \ldots, \ell_N))$**:**
- $S = []$
- add L to S
- **for** $i = 1$ **to** $N - 1$ **do**
  - best_diff $= +\infty$
  - $n = N - i$
  - **for** $j = 1$ **to** $n - 1$ **do**
    - new_diff $=$ RSS$(\ell_j \bullet \ell_{j+1}) -$ (RSS(ℓ_j) + RSS(ℓ_{j+1}))
    - **if** *new_diff* $<$ *best_diff* **then**
      - best_diff $=$ new_diff
      - $k = j$
  - $L = (\ell_1, \ldots, \ell_k \bullet \ell_{k+1}, \ldots, \ell_n)$
  - add L to S
- let $L \in S$ be the solution with minimal BIC
- **return** L

Objective function

The breakpoint learning process relies entirely on the objective function we defined, based on the residual sum of squares (RSS) and its BIC counterpart that penalizes the model size:

$$\text{RSS}(\ell, \theta) = \sum_{(X,Y)\in\ell} (Y - f_\theta(X))^2 \tag{5.8}$$

The algorithm is a heuristic to minimize the BIC, but the parameters of the linear regressions are also the optimal estimators for this objective function.

In the case of heteroscedastic data, the noise is by definition not constant: it grows linearly with the predictor variable. In this case, the RSS is not the best choice, as it does not penalize errors relatively to the predictor variable. It will give much more weight to the larger noise observed for larger predictor variables, hence a bias towards those large values.

A classical solution for this problem is the use of weighted least square (WLS) instead of the ordinary least square (OLS). A weight function w gives a different weight for each predictor variable. We define the new objective function as such:

$$\text{RSS}_{\text{weight}}(\ell, \theta) = \sum_{(X,Y)\in\ell} (w(X)(Y - f_\theta(X)))^2 \tag{5.9}$$

The $\text{BIC}_{\text{weighted}}$ value is defined like the BIC, replacing the RSS value by $\text{RSS}_{\text{weight}}$. In our case, we used $w(X) = 1/X$, a classical choice for linear regressions in the presence of heteroscedasticity.

As said previously, the optimal values for the regression parameters are not the same with this objective function. A closed formula can be found by taking the derivative of the function, very similarly to the ordinary least square.

$$\hat{\alpha}_{\text{weight}} = \frac{\sum_{(X,Y)\in\ell} w(X)(X - \hat{X}_{\text{weight}})(Y - \hat{Y}_{\text{weight}})}{\sum_{(X,Y)\in\ell} w(X)(X - \hat{X}_{\text{weight}})^2} \tag{5.10}$$

$$\hat{\beta}_{\text{weight}} = \hat{Y}_{\text{weight}} - \hat{\alpha}_{\text{weight}}\hat{X}_{\text{weight}} \tag{5.11}$$

Here, $\hat{X}_{\text{weight}}$ (resp. $\hat{Y}_{\text{weight}}$) denotes the weighted sample mean of the variables X (resp. Y), computed as:

$$\hat{X}_{\text{weight}} = \frac{\sum_{(X,Y)\in\ell} w(X)X}{\sum_{(X,Y)\in\ell} w(X)} \tag{5.12}$$

$$\hat{Y}_{\text{weight}} = \frac{\sum_{(X,Y)\in\ell} w(X)Y}{\sum_{(X,Y)\in\ell} w(X)} \tag{5.13}$$

An alternative objective function is based on the logarithm of the observations and the predictions:

$$\text{RSS}_{\log}(\ell, \theta) = \sum_{(X,Y)\in\ell} (\log Y - \log f_\theta(X))^2 \tag{5.14}$$

Likewise, the value $\text{BIC}_{\log}$ is naturally defined based on $\text{RSS}_{\log}$. The main motivation here is to penalize the prediction error relatively to the predictor variable and not in absolute value.

With such an objective function, there is no closed form for the optimal regression parameters. Instead, we implemented a gradient descent algorithm to find the optimum. The downside of using a gradient descent is that the whole regression is now significantly slower, the extent of this will be discussed later. On the other hand, a nice benefit is the possibility to tune the objective function to better suit our needs. For instance, in the case of network calibrations, the regression parameters α and β represent respectively the inverse of the bandwidth and the latency, it would make no sense for them to be negative. With the gradient descent approach, the search space can easily be restricted to limit the possible solutions to positive values.

As a side note, the $\text{RSS}_{\log}$ objective function is the RSS we would naturally have if we supposed that the noise was following a log-normal distribution instead of a normal distribution. Two justifications could be made to support this hypothesis. First, a random variable following a log-normal distribution is positive, which is what we would expect of duration measures. Second, a log-normal distribution is the product of many independent random variables (by using the central limit theorem in the log domain). It would not be unreasonable to suppose that independent performance perturbations have a multiplicative effect instead of an additive effect. However, these are only *a posteriori* justifications that were not considered at the time of implementing the solution. The next section shall determine if one objective function is better than the others for our needs.

Comparison of the solutions

We compared different approaches for computing a piecewise linear regression. We used a typical dataset for our context that has characteristics similar to that of

the network calibration data (see Figure 5.6): the predictor variable is sampled exponentially and the noise is heteroscedastic.

The different approaches are presented in Figure 5.7. Each row presents one solution, the first two rows are Datadog-piecewise [Dat] and Cubist [KQ20]. The last three rows are our proposed implementation with the three discussed objective functions. The black points represent the observations, the colored lines represent the linear regressions, while the different breakpoints are marked with dashed vertical lines.

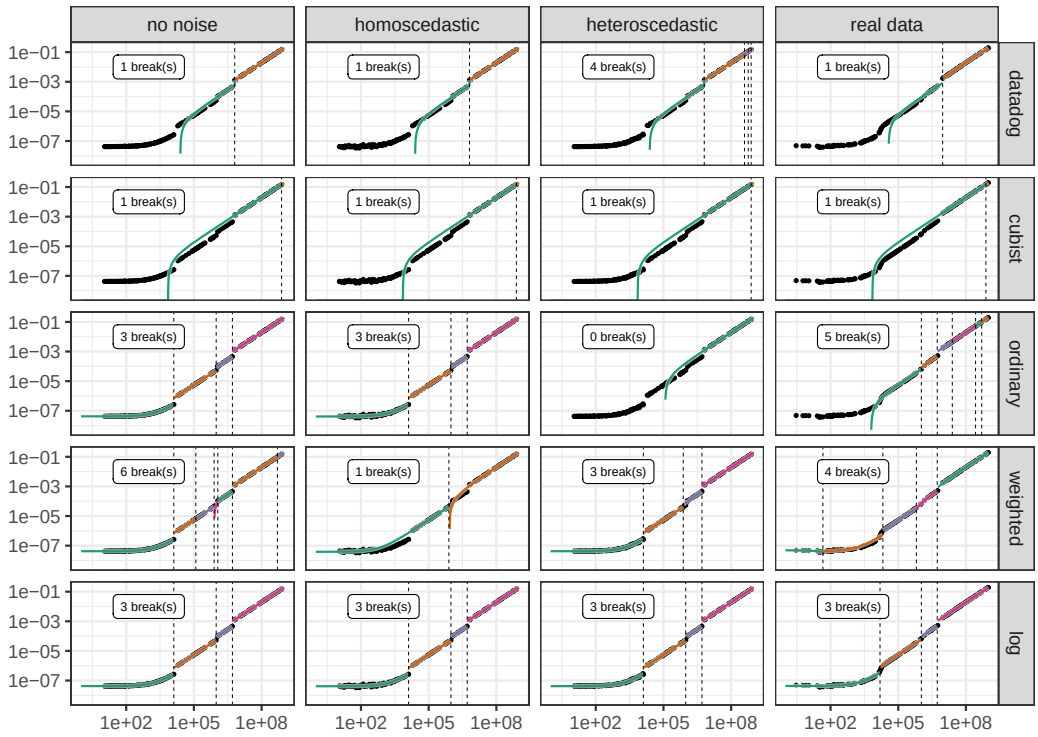


Figure 5.7.: Illustrating different piecewise linear regression approaches on different datasets. Only our implementation (`pycewise`) with the logarithmic objective function works well for all datasets.

Four variations of the dataset are used, one variation per column:

- The real data comes from a real experiment we performed on a node. These are durations of calls to `memcpy` function for various buffer sizes. Each point is the average of several measures with a same size, so we can reasonably expect the noise to be normally distributed.

- The three other variations are generated data. We performed a piecewise linear regression on the real data with three manually selected breakpoints, we sampled a new series of predictor variables, then we computed the ex-

pected durations. The response variables of the *no noise* dataset are entirely deterministic.

- The *homoscedastic* dataset has been generated by adding a normally distributed noise of mean 0 and standard deviation 5×10^{-9} to the *no noise* dataset.
- The *heteroscedastic* dataset has been generated by adding a normally distributed noise of mean 0 and standard deviation $2 \times 10^{-12} \times X$ to the *no noise* dataset, where X denotes the predictor variable.

The goal of using generated data was to understand better the limitations of each approach. From Figure 5.7, it is clear that both Datadog-piecewise and Cubist are unfit for our needs (first and second rows), as they could not even find the three breakpoints with the generated data without noise. The likely reason is the exponential sampling of the predictor variable: the points on the left part of the plots would get compressed into a single point if the data was plotted in linear scale.

As expected, the ordinary least square method (third row) works perfectly in the absence of noise or with homoscedastic noise, as it finds the three breakpoints. Unfortunately, it fails to find any breakpoint with the heteroscedastic noise. With the real data, it finds too many of them for large values but misses an obvious breakpoint for smaller values near 1×10^4.

The weighted least square method (fourth row) works as expected with heteroscedastic data, finding the three breakpoints. It also performs quite well with the real data, finding three obvious breakpoints, but it also detects a spurious one near the smallest values. It also fails spectacularly with the two other datasets, missing obvious breakpoints with the homoscedastic data and finding too many with the no-noise data.

In the end, only the logarithmic least square method (last row) works perfectly for all the datasets.

Another interesting feature of the last method is the positivity constraint. With this approach, it is possible to force the two parameters of each linear regression to be positive, as discussed previously. The three other methods all have an issue for at least one dataset where the intercept of one regression is negative, resulting in negative predictions when the predictor variable is too low. This can be seen when the prediction lines falls very sharply on the left. Note that this positivity constraint is only due to the gradient descent implementation and not to the objective function. We could very well add a similar constraint for the other objective functions if they were also implemented with a gradient descent instead of a simple formula.

The quality of the fit computed by pycewise with the logarithmic objective function is demonstrated in Figure 5.8. We do not show results for the alternative regression methods here because, like for the memcpy experiment, they do not work properly. We executed our MPI calibration program on four clusters of Grid'5000: `dahu`, `gros`, `paravance` and `pyxis`. For each of them, we measured the duration of individual `MPI_Recv`, `MPI_Send` and a whole ping-pong for various message sizes. Once again, we aggregated the data by taking the average duration for each message size. In this figure, one piecewise regression is computed for each cluster/operation pair. We also plotted with a small dotted line the regression we would get without breakpoints, with the logarithmic objective function. This dotted lines shows that without breakpoints, we would make large prediction errors in some cases (for instance, in the `send` experiment for clusters `gros` and `paravance`, the durations of $20\,\mathrm{kB}$ communications would be greatly over-estimated by a simple regression).

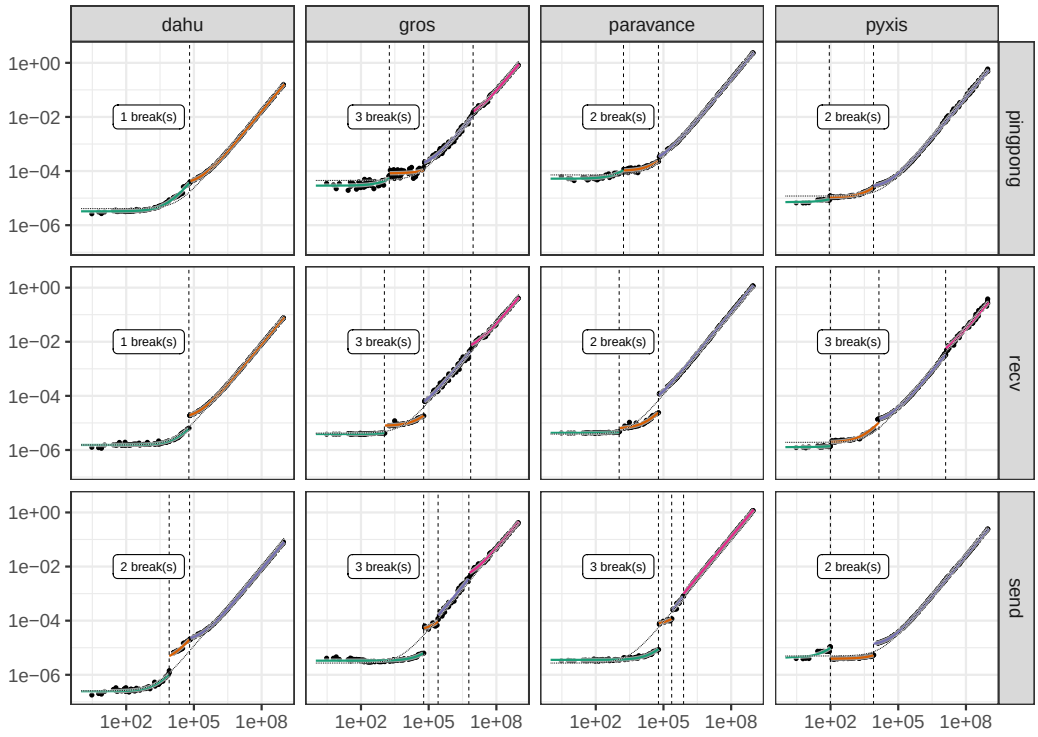


Figure 5.8.: With the logarithmic objective function, `pycewise` finds acceptable fits for the 12 datasets.

Since this is real experimental data, there is no "ground truth" that could serve as reference for evaluating the quality of the fit. Computing the optimal solution that minimizes the objective function would be computationally intractable. Therefore, we can only rely on a visual evaluation of the regression lines to decide if the fits are

satisfying. In Figure 5.8, all the obvious breaks have been found and the lines match the points very closely.

This gradient descent comes with a cost, computing the piecewise linear regression is now significantly slower. Figure 5.9 compares the three different objective functions from our implementation. The logarithmic least square is one order of magnitude slower than the weighted least square, which is itself one order of magnitude slower than the ordinary least square. The nature of the noise does not affect the duration, but we expect that a dataset with more breakpoints would lead to a larger duration.

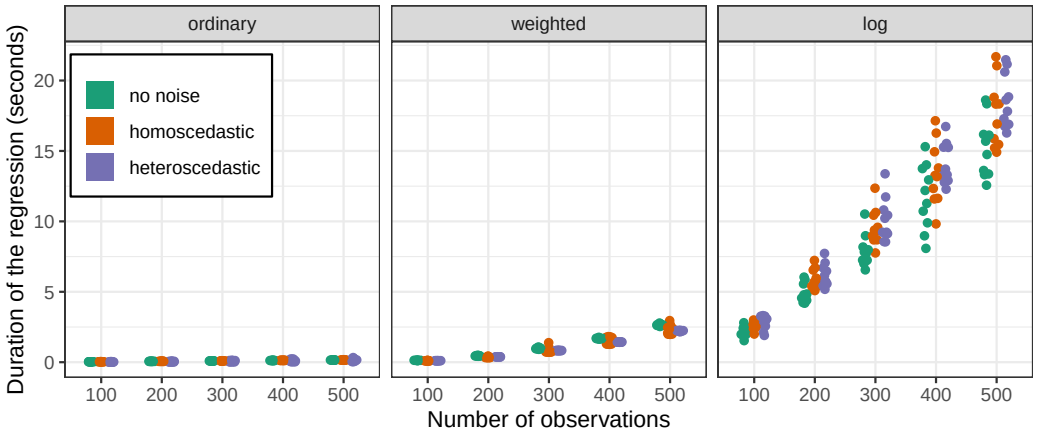


Figure 5.9.: Duration of a complete regression with pycewise for different numbers of observations.

In our context of network calibration, each communication for a given message size is repeated several times. Then, to generate a model, we start by averaging the durations per message size before doing the regression, as discussed previously. We can therefore safely expect to have only a few hundreds of observations: the logarithmic approach is perfectly reasonable.

5.4 Conclusion

We have presented different models for both the network communications and the computation kernels. One difficulty resides in instantiating the models, i.e. computing the best fit for some observations, in the most automated way possible. Bayesian formalism is an interesting candidate for such a task. With Bayesian sampling tools like STAN, we should be able to fit arbitrarily complex models and even have an estimation of the uncertainty of the resulting parameters. However,

the benefit of these tools is not clear for our context as their computation time is particularly long. Furthermore, we had great difficulties in writing down our models in their formalism. In the end, a simpler approach like the method of moments revealed to be enough for our needs.

The model instance, i.e. the fit we compute, will obviously depend on the input data. The experimental protocol for collecting the said data will therefore be as important as the statistical tools. This will be further discussed in Part II.

No model can capture the complexity of the real world perfectly. In Chapter 6, we will discuss what level of complexity is required for achieving a sufficiently faithful performance prediction. We will also demonstrate in Chapter 7 how these models can be used for sensibility analysis.

6 Validation

It is not possible to formally prove the "correctness" of a model like we would do for a theorem. The only reasonable method to evaluate the soundness of our approach is to compare simulation to reality. The goal here is to meticulously try many relevant configurations of the application to possibly unveil any flaw of the models discussed previously.

6.1 Experimental setup

We used the `dahu` cluster from the Grid'5000 testbed. It has 32 nodes connected through a single switch by $100\,\mathrm{Gbit\,s^{-1}}$ Omni-Path links. Each node has two Intel Xeon Gold 6130 CPU with 16 cores per CPU and we disabled hyperthreading. We used HPL version 2.2 compiled with GCC version 6.3.0. We also used the libraries OpenMPI version 2.0.2 and OpenBLAS version 0.3.1. Last, we used one single-threaded MPI rank per core.

Although this machine is much smaller than top supercomputers, faithfully simulating an HPL execution with such settings is already quite challenging.

- We used one rank per core to obtain a higher number (1024) of MPI process. This is more difficult than simulating one rank per node, as (1) this increases the amount of data transferred through MPI and (2) the performance is then subject to memory interference and network heterogeneity (we used a different model for local and remote communications).

- We used a smaller block size than commonly used, which leads to a higher number of iterations and hence more complex communication patterns.

- We used relatively small input matrices, which reduces the makespan and makes good predictions harder to obtain.

6.2 Analyzing the internal behavior of the application

In this section, HPL executions were done using a block size of 128 and a matrix of varying size (from $50{,}000$ to $500{,}000$). We used a look-ahead `depth` of 1, the `increasing-2-ring` broadcast with the `Crout` panel factorization algorithms.

Our first evaluation consists in comparing the traces of the simulations with reality. We instrumented HPL to collect the start and end timestamps of each kernel and MPI call. We limited the execution to 256 ranks and the first five iterations. A first qualitative validation can be done by visually comparing the Gantt charts of the simulations with reality (see Figure 6.1). Calls to `dgemm` are depicted in yellow, `MPI_Send` in red, `MPI_Recv` in blue. Although the structure is similar in all charts, the shape and the duration of the communication phases can be overly optimistic in simulations compared to reality. The major difference are the calls to `MPI_Send` at the start of the execution, they take a much longer duration in reality (large red rectangles) than in simulation. This discrepancy is then attenuated in the rest of the execution. Furthermore, the charts show that, at this scale, using a deterministic or a stochastic model for the network has no noticeable impact on HPL simulation. However, having a more complex model for the kernels leads to much more realistic traces. The variability in the computation durations leads to an increase of the time spent in communications and an overall slightly longer execution. In HPL, computation variability directly translates to late senders/receivers that harm the efficiency of collective operations.

This demonstrates one of the advantages of simulation over other prediction methods. By generating a trace, we can have a very thorough visual inspection of the simulated execution, which makes it possible to identify the culprits of accuracy losses.

6.3 Evaluating the influence of the problem sizes

We now provide a more quantitative comparison using the whole cluster and varying matrix sizes, focusing on the $\mathrm{GFlop\,s^{-1}}$ rate reported by HPL (see Figure 6.2). The real executions are depicted in black, for each matrix size we performed 8 runs of HPL, to illustrate the temporal variability of the performance. The line (a), on the top, is our first attempt to simulate HPL. The simulation was done with a simple model: $M-1$ for the kernels and $M'-1$ for the network with no noise ($N-0$) in both cases. This model overestimates HPL performance by more than $30\,\%$. We initially thought that the network model was too optimistic, however, switching to a

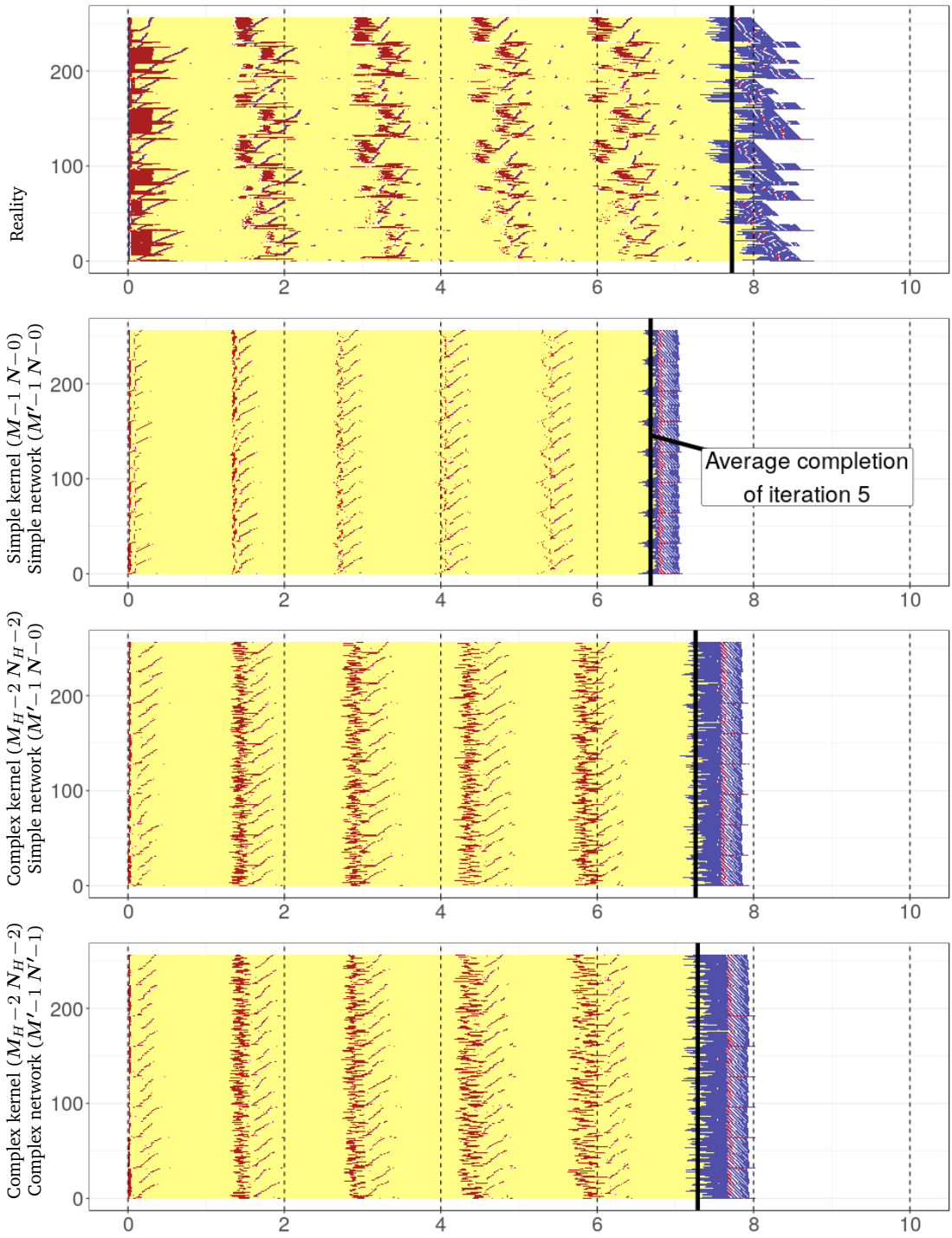


Figure 6.1.: Gantt charts of HPL first iterations in simulation.

stochastic multi-modal network model ($\boldsymbol{N'}-1$, the line (b)), does not significantly improve the prediction precision.

Figure 5.1 shows that there is a significant heterogeneity in the cluster. For this reason, we started using a $\boldsymbol{M}_H-1/\boldsymbol{N}-0$ model for `dgemm` while keeping the models for the other kernels and the network as before. This increases very significantly the realism of the simulation as the performance is now overestimated by only $9\,\%$ (the line (c)). Using a polynomial model for `dgemm` instead of a linear model (thus switching from $\boldsymbol{M}_H-1$ to $\boldsymbol{M}_H-2$) further improves the performance prediction, in particular for smaller matrices. This new model (the line (d)), is very close to reality at the beginning but becomes equivalent to the previous model for larger matrices. We found that adding the temporal variability noise ($\boldsymbol{N}-2$ for all kernels, $\boldsymbol{N}_H-2$ for `dgemm`) is the key ingredient to obtain the last bit of realism. The prediction (the line (e)) is now extremely close to reality as it slightly underestimates the performance by less than $5\,\%$ and even as little as $1\,\%$ for the larger matrices. Adding back temporal variability to the network model ($\boldsymbol{N'}-1$, line (f)) still has no significant effect but this can be explained by the fact that HPL mostly communicates very large amounts of data in bulk. Network temporal variability is however a very important aspect to model applications that are more *latency-bound*.

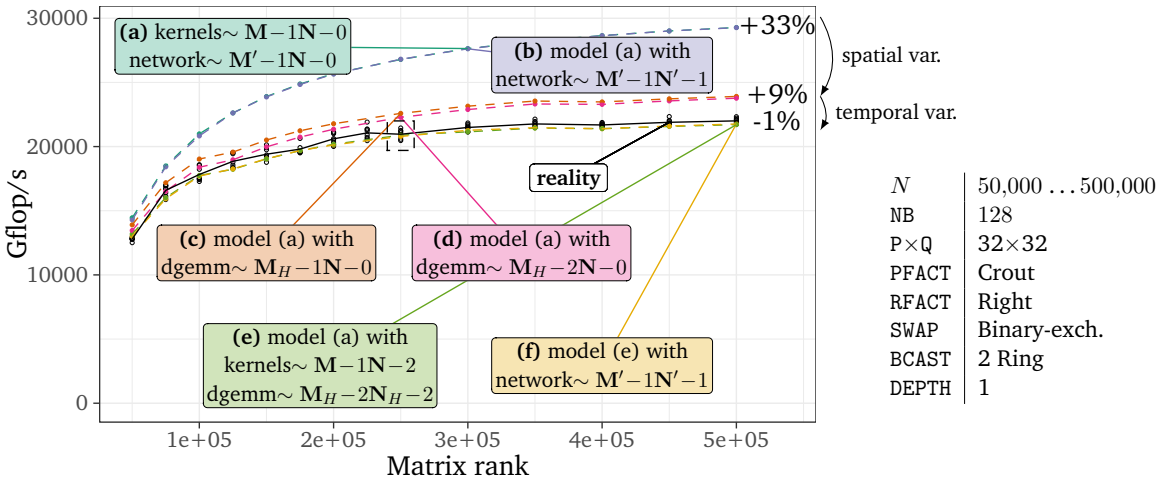


Figure 6.2.: HPL performance: predictions vs. reality for various matrix ranks.

As illustrated in Figure 6.2 (rightmost labels), we would like to stress again that accurate predictions require a careful modeling of both spatial and temporal variability. Overall, although this was not particularly foreseeable and could have been different with another application, only the `dgemm` kernel needs to be carefully modeled with a $\boldsymbol{M}_H-2$-$\boldsymbol{N}_H-2$ model. A detailed modeling of all (BLAS and HPL) kernels is possible but a minimal calibration of the `dgemm` kernel over a representative set of nodes is

thus sufficient to consistently predict the performance of HPL on this machine within a few percent of reality.

6.4 Evaluating the influence of a platform change

This validation study has been carried out over several years (from 2018 to 2020). Despite our efforts to keep the experimental setup stable for the sake of reproducibility, the platform has evolved. The Linux kernel had a minor update, from version 4.9.0-6 to version 4.9.0-13, and the BIOS and firmware of the nodes have been upgraded. During this time frame, the cluster has also suffered from hardware issues, such as a cooling malfunction on four of its nodes. This malfunction had an enormous impact on the performance of HPL, which significantly complicated our validation study but also makes it more meaningful as it has been conducted on a particularly challenging setup.

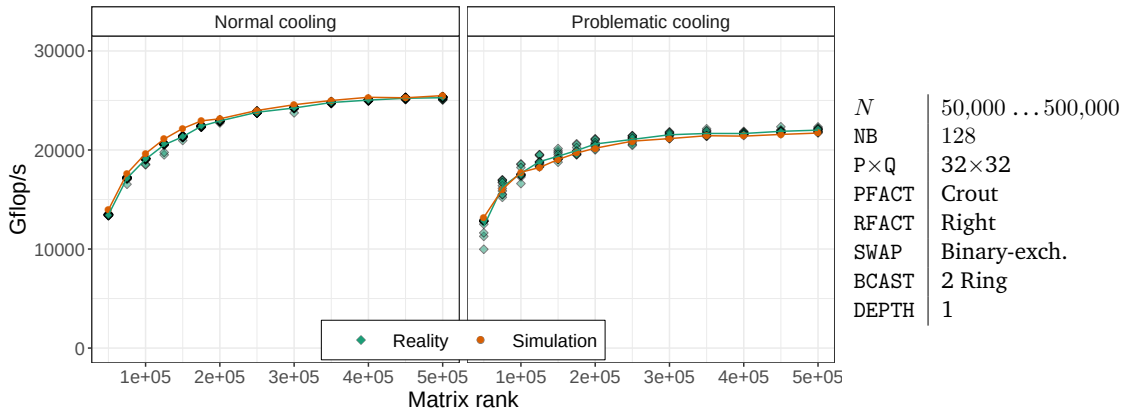


Figure 6.3.: HPL performance: predictions vs. reality (effect of the cooling issue on the nodes `dahu-{13,14,15,16}`).

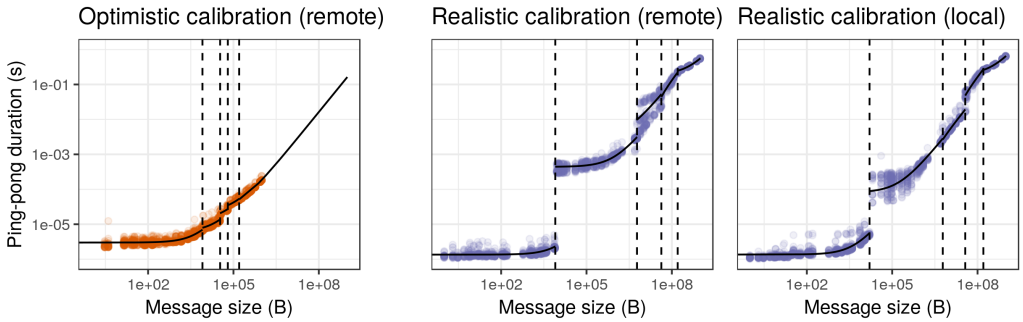
Our simulation approach makes it possible to predict the performance of HPL for a new platform state by merely conducting a new calibration whenever a significant change is detected. This ability to reflect in simulation a platform change is illustrated in Figure 6.3 which, similarly to Figure 6.2 (acquired in March 2019), showcases the influence of matrix size on the performance but at different periods. The left plot represents the *normal* state of the cluster (in September 2020), whereas the right plot has been obtained (in March-April 2019) when 4 of the 32 nodes had a cooling issue which lowered their performance by about $10\,\%$. In all cases, we consistently predict performance within a few percent and performing a new `dgemm` calibration on these four nodes was all that was needed to reflect this platform change in the

simulation. This result illustrates both the faithfulness of our simulations and a potential use case for predictive simulations: a discrepancy between the reality and the predictions can sometimes indicate a real issue on the platform (similar situations have already been reported with SMPI [Deg+17]).

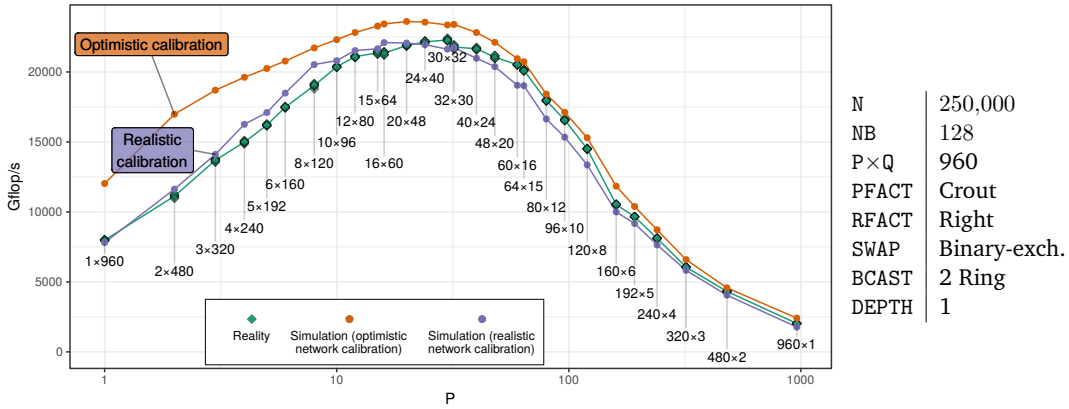
6.5 Evaluating the influence of the geometry

Figure 6.4(b) illustrates the influence on the performance of the geometry of the virtual topology (`P` and `Q`) used in HPL. As expected, geometries that are too distorted lead to degraded performance. All the HPL parameters were fixed (matrix rank is fixed to $250{,}000$ and the other parameters are the same as in Section 6.3) except for the geometry as we evaluate all the pairs (`P`, `Q`) such that $\texttt{P} \times \texttt{Q} = 960$. We used only 30 nodes instead of 32 to cover a larger number of geometries, as 960 has more divisors than 1024.

As in all our previous studies, we report both the predicted performance and the one measured in reality. Like the comparisons presented in the previous section, the simulation was done with the `dgemm` model from Equation (5.1) (stochastic, heterogeneous, and polynomial) and the simplest linear models for the other kernels. In our first simulation attempt that relied on a relatively simple network model (deterministic yet piecewise-linear to account for protocol switch) depicted on the leftmost plot of Figure 6.4(a), we obtained the unsatisfactory orange line on top of Figure 6.4(b) for the prediction. The simulations with the smallest value of `P` had relatively large prediction errors, with a systematic over-estimation that reaches up to +50% for the 1×960 and 2×480 geometries. A qualitative comparison of the execution traces obtained in reality and simulation showed that the broadcast phases' duration was greatly underestimated in simulation. We found out that with such elongated geometries, the message size is significantly larger than what we had used in our calibration, and the performance surprisingly and significantly drops for such size (compare with the rightmost plots of Figure 6.4(a)). This performance drop is explained by poor optimization of the DMA locking mechanism in the Infiniband network layer [Den11]. A similar performance drop also happens for intra-node communications that poorly manage the caches above a given size. Furthermore, the communication patterns generated by HPL during the ring broadcast are significantly impacted by the busy waiting of HPL that intensively calls `MPI_Iprobe` and `dgemm` on small sub-matrices. Our initial procedure for calibrating the network did not capture this phenomenon since we did not inject any additional CPU load. This problem is further discussed in Section 10.7.



(a) Illustrating the effect of the two MPI calibration methods. The optimistic method merely samples message size smaller than 1 MB and extrapolates for larger sizes. Unfortunately, for messages larger than 160 MB, the effective bandwidth significantly drops. The more realistic calibration measures MPI communication duration for messages up to 1 GB while injecting some computation load in the background.



(b) HPL performance: predictions vs. reality (testing all the possible geometries for 960 MPI ranks).

Figure 6.4.: The first (optimistic) network calibration gives poor predictions for very elongated geometries while the improved calibration provides perfect predictions.

We address the above problem by improving our network calibration procedure: (1) we use a distinct model for local and remote calibrations, (2) we sample the message sizes in a larger interval, and (3) we add calls to `dgemm` and `MPI_Iprobe` between each call to `MPI_Send` and `MPI_Recv`. The goal was to make the calibration environment more similar to what happens in HPL. The resulting network model is illustrated in the rightmost plots of Figure 6.4(a). This more realistic network model solved every previous misprediction and allows us to produce very faithful simulations (purple line on Figure 6.4(b)), which are now a few percent of the reality regardless of the geometry. This figure also illustrates the influence of the geometry on overall performance since there is almost a factor of ten between the worst configuration (960×1) and the best one (30×32). Although it is not surprising to see that the geometries which are as square as possible lead to better performance as they minimize the overall amount of data movements, it is interesting to observe

the asymmetric role of `P` and `Q` in the overall performance (smaller values for `P` lead to better performance) and which can be explained by the structure of the collective operations but requires a close inspection of the code.

6.6 Evaluating the influence of the other parameters

Although geometry is among the most important parameters to tune, six other parameters control the behavior of HPL. In Figure 6.5, we compare the performance reported by HPL when fixing the matrix rank to $250{,}000$ and varying the following parameters: block size (128 or 256), depth (0 or 1), broadcast (the six available algorithms), swap (the three available algorithms). The geometry was fixed to $P \times Q = 32 \times 32 = 1024$ as it is best experimentally (the simpler calibration procedure and the network model depicted on the leftmost plot of Figure 6.4(a) were thus used). The parameters `pfact` and `rfact` (panel factorization) were respectively fixed to `Crout` and `Right`, as they had nearly no influence on HPL performance in our early experiments.

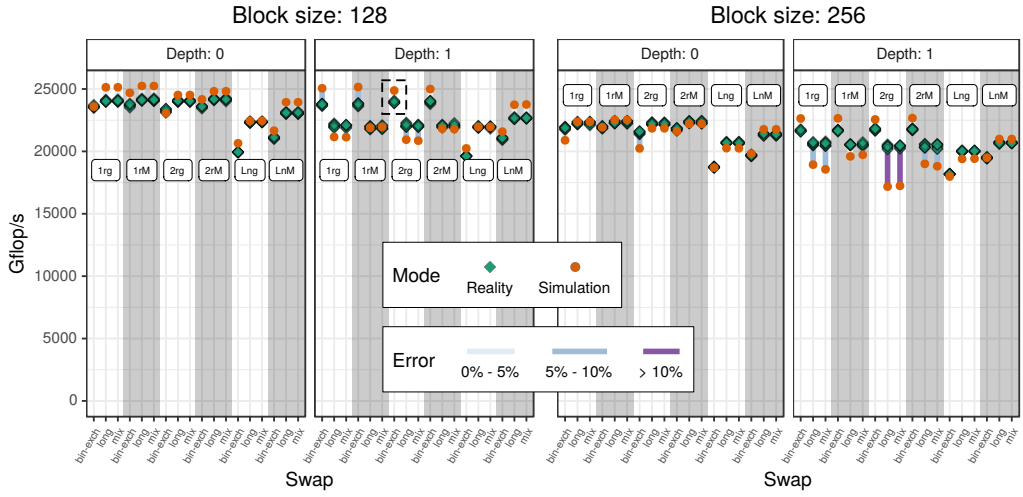


Figure 6.5.: Influence of HPL configuration on the performance (factorial experiment). Parameters have been reorganized based on their influence on performance to improve readability. The boxed configuration corresponds to the one boxed in Figure 6.2.

Figure 6.5 depicts the 72 parameter combinations we tested. These parameters account for up to $30\,\%$ of variability in the performance, which is less important than the geometry but is still quite significant. For 61 of them, the prediction error is lower than $5\,\%$. Only two combinations have shown a large error of approximately $15\,\%$, obtained with a block size of 256, a depth of 1, the `2-ring` broadcast algorithm,

and either the `long` or the `mix` swap algorithm. This demonstrates the soundness of our approach, as our predictions are reasonably accurate most of the time. This experiment confirms that, although the prediction of HPL performance for a given parameter combination has a systematic bias, the error remains within a few percent most of the time. Therefore, this surrogate is good enough for parameter tuning and should be considered when preparing a large-scale run.

While testing all the parameter combinations is the safest method to discover the combination that provides the highest performance, its cost can be prohibitive due to the high number of combinations. An alternative often used in practice is to explore only a small subset of the parameter space and to analyze variance (ANOVA) to identify the parameters with the more substantial effect on performance and then select the appropriate combination. We applied this procedure on samples of both datasets (the one obtained from real runs and the one obtained in simulation). In both cases, the two parameters with the highest effect were the block size `NB` and the `depth`, as shown in Figure 6.5, followed by `bcast` and `swap`. The best combinations selected in both cases were also identical, demonstrating once again the faithfulness of our simulation approach and how it can be used to reduce the experimental cost of parameter tuning.

6.7 Conclusion

Accurately predicting the performance of an application is not a trivial task. Discrepancies between reality and simulation can be multiple: the platform may have changed (e.g. the cooling issue that affected four nodes in Section 6.4), the model could be inaccurate (e.g. the homogeneous and deterministic `dgemm` model is too simple as in Section 6.3) or not correctly calibrated (e.g. the calibration procedure does not cover the appropriate parameter space, or the experimental conditions are too different as in Section 6.5). As expected in any serious investigation of model validity, our validation study is not a mere collection of positive cases. Instead, it is the result of a thorough (we extensively covered the HPL parameter space) attempt to invalidate our model as well as explanations on how we did so. By meticulously overcoming each of these issues, we have demonstrated the ability of our approach to produce very faithful predictions of HPL performance on a given platform.

The difficulties encountered while conducting experiments will be discussed more thoroughly in Part II.

7 Sensibility analysis

We have shown that many typical HPL case studies could be conducted in simulation. However, their conclusions (optimal geometry and parameters) are specific to the cluster we used and they require a precise model of several aspects of the target cluster, which may not be possible at early experimental stages. In particular, only a few cluster nodes may be available at first and the whole cluster model should then be constructed from a limited set of observations and carefully extrapolated. This section shows how typical *what-if* simulation studies should be conducted given such uncertainty. We have presented in Section 5.2.2 a generative model of node performance that can easily be fit from daily measurements and used to produce a similar platform. This model is used to quantify the importance on overall performance of temporal variability of the `dgemm` kernel in Section 7.1 and of spatial variability of nodes in Section 7.2. In particular, we show how to study the efficiency of a simple *slow node eviction* strategy. Finally, we study in Section 7.3 the influence of the physical network topology on overall performance. Most of these studies are particularly difficult to conduct through real experiments because of the difficulty to finely control the platform.

7.1 Influence of `dgemm` temporal variability

In Section 6.3, we were able to highlight the importance of accounting for temporal variability of the `dgemm` kernel to obtain faithful HPL predictions. To the best of our knowledge, HPL developers and experts are often aware of this influence (or at least suspect it). However, they have never fully quantified it since designing and performing real experiments to evaluate it would be quite difficult. Although increasing this variability would be feasible, reducing it would be particularly complicated. This can however easily be done through simulation using the hierarchical model of Section 5.2.2. In our experiments, the order of magnitude of the temporal variability with respect to actual performance (i.e., the ratio between $\gamma_{p,d}$ and $\alpha_{p,d}$ in Equation (5.2)) was around $3\,\%$. This may be a "normal" value or could be considered too high and possibly improved by better controlling thread mapping or Operating System noise. Such a task can be quite tedious and knowing how much performance

gain can be expected beforehand is thus quite useful. In this section, we study the influence of this variability by generating 10 cluster scenarios using the previous model (as in Figure 5.4), comprising 1024 nodes each, but by constraining $\gamma_{p,d}$ to be equal to $\gamma.\alpha_{p,d}$ with $\gamma \in [0, 0.1]$, which represents the coefficient of variation of the `dgemm` kernel. We evaluate the performance of HPL with one multi-threaded MPI rank per node, a block size of 512, a look-ahead `depth` of 1. We used the `increasing-2-ring` broadcast with the `Crout` panel factorization algorithms and $\texttt{P} \times \texttt{Q} = 8 \times 32$, and we tested matrix sizes ranging from 100,000 to 500,000. Let us denote by $T(N, C_i, \gamma)$ the performance of HPL when factorizing a matrix of rank N on cluster C_i with a temporal variability of γ. The overhead for this configuration is the ratio

$$O(N, C_i, \gamma) = \frac{\mathbb{E}[T(N, C_i, \gamma)]}{T(N, C_i, 0)} - 1.$$

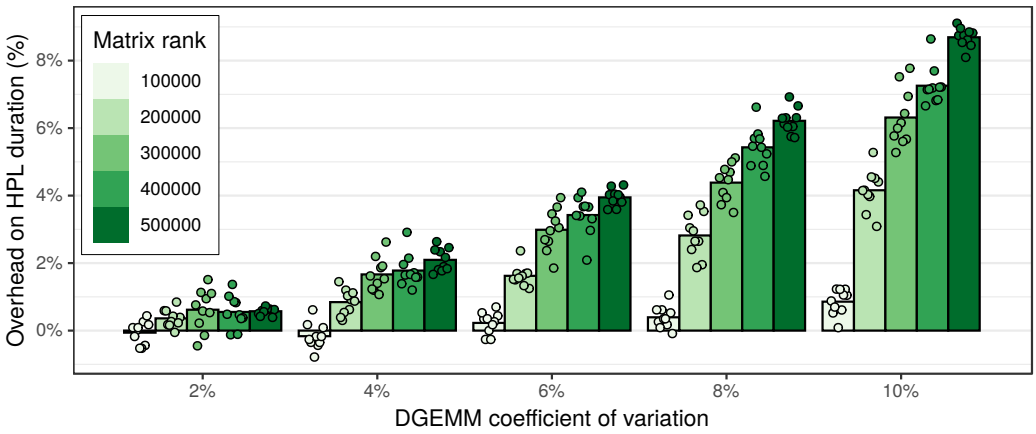


Figure 7.1.: The overhead on HPL duration appears to be linear in `dgemm` temporal variability. Although it is negligible for small matrices, it severely inflates for larger matrices.

Each bubble in Figure 7.1 represents one such overhead. For any γ, this overhead appears to be negligible for small matrices and to increase and flatten when N grows large. In most Top5000 qualification runs, the matrix is made as large as possible and the overhead would thus appear to grow roughly linearly with γ. On a new cluster, a simple statistical evaluation of the nodes' performance using the model of Section 5.2.2 would thus be a good first diagnosis of whether trying to decrease temporal variability is a promising tuning target or not.

7.2 Influence of dgemm spatial variability

Although we showed in Section 6.3 that temporal variability could account for about $9\,\%$ of performance degradation, spatial variability was even more important as it was responsible for $22\,\%$ of overhead compared to a fully homogeneous cluster. In practice, the replacement of a few nodes may be possible but such spatial variability is expected and common [Ina+15] and a workaround would have to be found. A common approach consists in dropping out a few of the slowest nodes. Indeed, since the matrix is evenly divided between the nodes, the computation inevitably progresses at the speed of the slowest node. However, removing the slowest nodes also decreases the overall processing capability and impacts the virtual topology's geometry (the P and Q parameters of HPL). Such adjustment is often done by trial and error and is all the more tricky as temporal variability and uncertainty from real experiments come into play. In this section, we show how such a subtle trade-off can be studied in simulation.

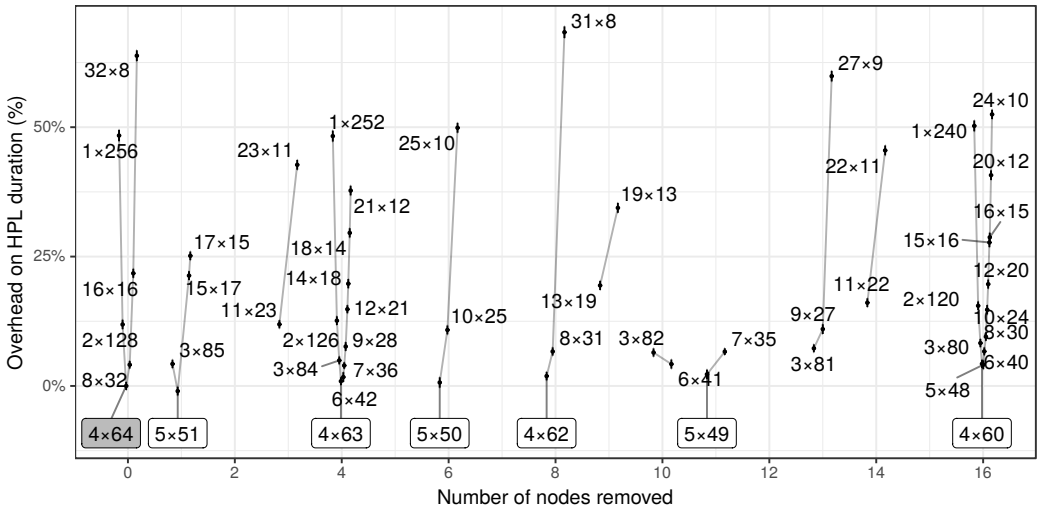


Figure 7.2.: Influence of the number of nodes on the performance of HPL. The geometry of the virtual topology is particularly influential, and it appears that P × Q configurations with a small P perform significantly better than those with a larger P. Each configuration is summarized through the average overhead over the 10 clusters and error bars represent a $95\,\%$ confidence interval.

Using the model from Section 5.2.2, we generate 10 mildly heterogeneous 256 node clusters (i.e., where nodes are similar to the ones of our cluster when operating in the normal state as in Figure 5.4(a)) and we study the performance obtained when removing 1 to 16 of the slowest nodes. When removing nodes, the geometry should be adjusted depending on how the number of remaining nodes decomposes in prime factors. As observed in Figure 6.4(b), having $\mathtt{P} \approx \mathtt{Q}$ is generally a good idea to reduce

the total amount of communication. However, it may be counter-productive for a given broadcast or swap algorithm that serializes communications. Figure 7.2 shows the average (over the 10 clusters) overhead for a matrix of rank $250{,}000$ compared to the best performance obtained using the whole cluster. We group the different $\mathtt{P} \times \mathtt{Q}$ decompositions and order them by increasing `P`. Again, we use the `2-Ring` and `Binary-exch` algorithms, which are among the best configurations according to the study of Section 6.6. It appears that the 4×64 geometry now achieves the best trade-off between the total amount of communications and how well they overlap with each other. The optimal configuration for each number of nodes is boxed in Figure 7.2. It reveals that there is not much to gain, probably because of the mild spatial heterogeneity of our cluster, but that optimizing the virtual topology is particularly important.

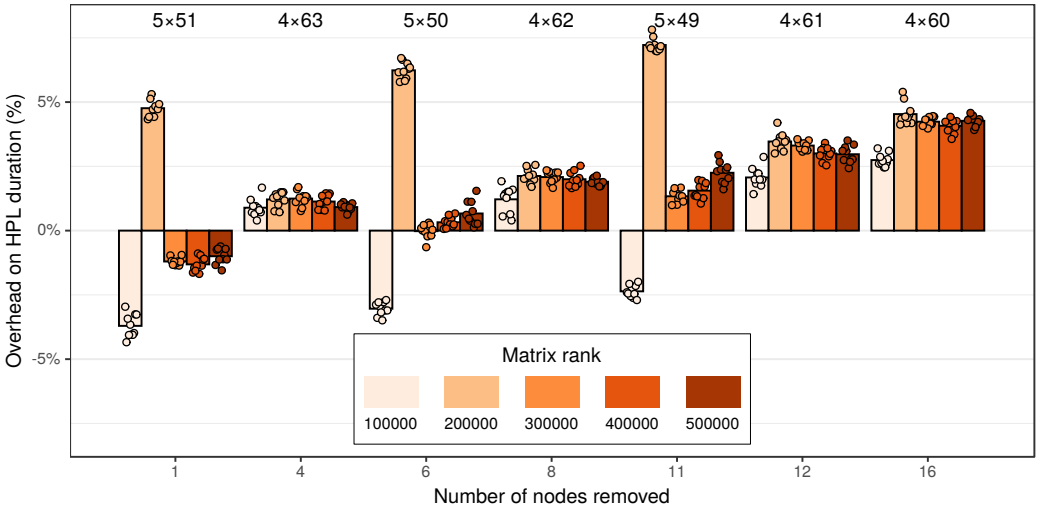


Figure 7.3.: Influence of node removal on performance while taking into account the matrix rank. Due to the mild heterogeneity of these scenarios, evicting nodes brings no benefit.

Figure 7.3 investigates how this overhead for the best geometry and node selection also depends on the matrix rank. It appears that in this scenario, except for very small matrices, removing nodes cannot help improve performance. Note that the overhead for $5 \times \mathtt{Q}$ configurations with a matrix rank 200,000 appears to behave differently from what happens for other matrix sizes. This surprising effect probably arises from a subtle combination of matrix size and virtual topology. We could indeed observe on our cluster that such configurations had a weakly but significantly worse performance than the other configurations. Such interaction also explains why designing a faithful analytical model of HPL is so difficult and why a full simulation of the whole application is generally required. Although absolute performance should be taken with a grain of salt when studying such subtle effects, they are easily

overlooked when conducting real experiments. In this particular small scale mild heterogeneity scenario, there is thus no gain in removing nodes but, as illustrated in Figure 7.4 where we used a multimodal spatial heterogeneity (as in Figure 5.5), this may be a relevant approach. This sensibility analysis shows how, for a given supercomputer, a simple statistical evaluation of the spatial heterogeneity allows evaluating whether spatial variability is a promising tuning target or not.

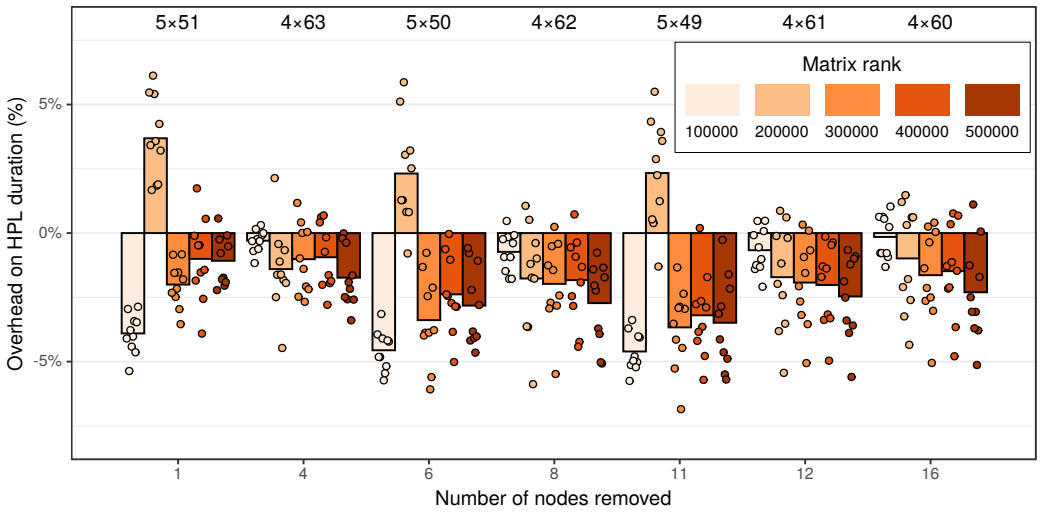


Figure 7.4.: Influence of node removal on performance in a stronger heterogeneity scenario (extrapolation of our test cluster when it had a cooling problem on 4 of its nodes). Removing 6 to 12 nodes our of 256 nodes may bring substantial improvement and such optimization would therefore be worth investigating.

7.3 Influence of the physical topology

Finally, since virtual topology and communications appear to significantly influence the overall performance, one may wonder how much the physical topology influences the performance. Indeed, several recent articles [Leó+16; Taf+19] report that interconnect networks are often oversized compared to the actual need of applications and that turning off some switches could sometimes go completely unnoticed by end-users. In this section we consider ten 256-node clusters with variable node performance (as in Figure 5.4) interconnected by a 2-level fat-tree and quantify by how much performance degrades when the top-tier switches are gradually deactivated. More formally, we use a `(2;32,8;1,N;1,8)` fat-tree with $\texttt{N} \in \{1, 2, 3, 4\}$.

Figure 7.5 depicts this degradation as a function of matrix size. As one could expect, the impact is more significant for smaller matrix sizes (where the execution is more

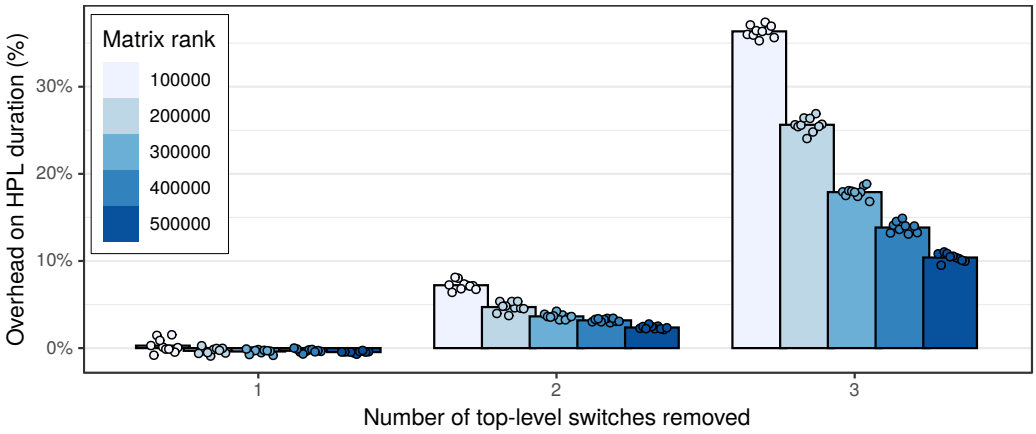


Figure 7.5.: Influence of the physical topology on the overall performance. It is possible to remove up to 2 of the top-level switches without significantly hurting performances for large matrices. Beyond this point, communications become the main performance bottleneck.

network bound). Although removing one switch leads to absolutely no visible performance loss, removing two or three switches can have a dramatic effect. Again, such degradation depends on the broadcast and swap algorithms and may be slightly mitigated. To the best of our knowledge, it is the first time such sensibility analysis is conducted faithfully. Generating random node configurations allows avoiding potential bias, in particular against perfectly homogeneous scenarios. We believe such a tool can be quite useful in the earlier steps of a supercomputer design when performing capacity planning to adjust the network capacity to a given cost and power envelope.

8 Conclusion

HPC application developers implement many elaborate algorithmic strategies whose impact on performance is often dependent on both the input workload and the target platform. This structure makes it very difficult to model and accurately forecast the overall application performance, and many HPC application developers and users are often left with no other option but to study and tune their applications at scale, which can be very time- and resource-consuming. We believe that being capable of precisely predicting an application's performance on a given platform is useful for application developers and users. It will become invaluable in the future as it can, for example, help computing centers with deciding which one of the envisioned technologies for a new machine would work best for a given application, or if an upgrade of the current machine should be considered.

Simulation is an effective approach in this context and SimGrid/SMPI has previously been successfully validated in several small-scale studies with simple HPC benchmarks [Deg+17; Hei+17]. In this work, we have explained how SMPI could be used to efficiently emulate HPL. The proposed approach only requires minimal code modifications and applies to any application whose behavior does not strongly depend on data-dependent intermediate computation results. Although HPL is not a *real* application, it is quite optimized from an algorithmic point of view and its behavior can be controlled through 6 different parameters (granularity, geometry of the virtual topology, broadcast/swapping/factorization algorithm, and the number of concurrent iterations). HPL features classical optimization techniques such as heavily relying on `MPI_Iprobe` to overlap communication with computations, making it particularly challenging both in terms of tuning and simulation.

We present in Chapter 6 an extensive validation study which covers the whole parameter space of HPL. Our study emphasizes the importance of carefully modeling (1) the platform heterogeneity (not all nodes have exactly the same performance), (2) the short-term temporal variability (e.g., system noise) for compute kernels as it may propagate in communication patterns, and (3) the complexity of MPI (performance often wildly differs between small and large messages and between intra-node and extra-node communications). We show that disregarding any of these aspects may lead to wildly inaccurate predictions even on an application as

regular as HPL. By building on a few well-identified micro-benchmarks of the BLAS and MPI, we show that these aspects can be modeled accurately, which allows us to systematically predict the overall performance of HPL within a few percent. Our experimental results span over two years and we report situations (in Sections 6.4 and 6.5) where the simulation helped us to identify performance regression or anomalies incurred by the platform when the prediction did not match the real experiments.

We show in Chapter 6 how this faithful surrogate can be used to evaluate the significance of application parameters and tune them accordingly solely through simulations. We also propose a generative model for the compute nodes' performance that can easily be fit from daily measurements and used to produce synthetic platforms similar to the ones at hand. We demonstrate in Chapter 7 how this model, which allows us to easily control temporal and spatial variability, can feed our simulations to assess the impact of variability on the performance of the application or of mitigation strategies (e.g., the eviction of the slower nodes). Likewise, the simulation makes it possible to assess precisely the influence of the physical network on the overall performance. Most of these *what-if* studies would be particularly difficult to conduct through real experiments because of the difficulty to finely control the platform. This is to the best of our knowledge one of the first sensitivity analyzes of a real HPC code accounting for platform uncertainty.

As future work, building on the effort of SimGrid developers on supporting the emulation of a wide variety of applications with SMPI [SG19], we also intend to conduct similar studies with other HPC benchmarks (e.g. HPCG [DHL15] or HPGMG [F A+14]), real applications (e.g. BigDFT [Gen+08]) and larger infrastructures. As explained in this thesis, a good model of compute kernels and the MPI library is essential. Therefore, the main challenge for systematic use of our simulation technique now lies in the automation of measurements through well-designed experiments and the automatic detection of when the envisioned models miss essential characteristics of the platform (multi-modal behaviors, heteroscedasticity, discontinuities,...). We intend to provide a fully automatic calibration procedure for MPI as well as for every BLAS function, which would allow us to effortlessly predict the performance of many applications by simply linking against a BLAS-replacement library.

This work has required many experiments, notably for measuring the durations of the MPI and BLAS functions and for performing real executions of HPL. In Part II, we will present the difficulties that were encountered while doing these experiments and how they were overcome.

Part II

Experimental control

9 Experimental testbed and experiment engines

9.1 State of the art

9.1.1 Grid'5000

Nearly all the experiments presented in this document have been carried out on the Grid'5000 [Bal+13] testbed. Quoting its official website[1]: "Grid'5000 is a large-scale and flexible testbed for experiment-driven research in all areas of computer science, with a focus on parallel and distributed computing including Cloud, HPC and Big Data and AI." It provides dozens of clusters, each one having between 2 and 124 homogeneous compute nodes. There is a high diversity of hardware, including several generations of Intel processors available, AMD and ARM processors, GPU, persistent memory (PMEM) as well as high-performance networks such as Infiniband or Omni-Path. Another important feature is the ability for the experimenter to get full control on the nodes, as it is possible to deploy a new operating system and therefore to gain superuser access.

9.1.2 Experiment engines

While it is possible to run a complete experiment on a testbed like Grid'5000 by manually issuing commands in an interactive shell, it is not advisable as it quickly becomes extremely tedious and error-prone. Automating the experiment is a necessary condition for reproducible results. A first step toward this goal is to write some ad hoc script. However, two independent experiments might still share many steps that could be refactored in a common layer, e.g. OS deployment, package installation, or even more advanced features like node instrumentation or environment logging.

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

For these reasons, it is a common practice to use an experiment engine. Buchert et al. describe the features of eight different engines [Buc+15]. To the best of our knowledge, only three offer a native support for Grid'5000, namely `expo`, `XPFlow` and `execo`. Unfortunately, `expo` and `XPFlow` are now longer maintained, the last commit in their respective repositories was done on November 2014 and September 2015 For these reasons, the experiment engine `execo` [Imb+13] is often recommended to Grid'5000 newcomers.

Experiments with `execo` are described as a Python script. We believe this is one of its best qualities, as it offers a lot of freedom and flexibility to the experimenter, comparatively to other experiment engines that use custom domain specific languages (DSL). Yet, we made the choice to not use it. The main reason is that a typical `execo` experiment uses many low-level constructs that are really cumbersome and unintuitive to write and read. Section 9.2.2 presents a comparison between the approach we implemented and `execo`. Furthermore, `execo` lacks many important features, like node instrumentation and metadata collection, i.e. we would still have needed to implement many functionalities on top of `execo`.

9.2 Yet another experiment engine: `peanut`

9.2.1 Key features

We implemented our own experiment engine, named `peanut` [Cor21c]. It comes as a Python library that experimenters can use to write their own experiments, also as a Python script.

A new experiment can be defined by inheriting from the class `Job`. Three methods can be overridden: `setup`, `run_exp` and `teardown`.

Once the experiment is written, it can be launched in a single command line. The following steps will happen:

- Implicitly, submit a job with the given characteristics (e.g. cluster, number of nodes, wall time, etc), then deploy the given OS image.
- Implicitly (but optionally) enable or disable some performance functionalities like hyperthreading, turboboost, C-states.

- Implicitly (but optionally) instrument the nodes to collect at a regular interval some system metrics (e.g. core frequencies and temperatures, CPU power consumption, network traffic, memory consumption).
- Implicitly (but optionally) run the `stress` command on all the nodes to warm them up.
- Run the methods `setup`, `run_exp` and `teardown` in that order.
- Produce a `zip` archive containing relevant results and metadata. The experimenter can explicitly add any file to the archive. In addition, the following content is also implicitly archived:
  - Metrics collected with the aforementioned instrumentation.
  - Human-readable log of the commands issued during the experiment.
  - Machine-parsable log of the commands (in JSON format) with their timestamps and output (both `stdout` and `stderr`).
  - Machine-parsable file (in Yaml format) containing relevant information like the exact versions used for `peanut`, `gcc`, `MPI` and the Linux kernel, the command line that was used to launch this experiment, the cluster and the list of nodes, start and end timestamps for each of the three main methods, the list of the git repositories cloned during this experiment with their remote URL and the git hash of the checkout.
  - For each node, the content of the file `/proc/cpuinfo` as well as the output of the commands `env`, `lstopo`, `lspci`, `dmidecode`, `lsmod`, `dmesg`.

In addition, the experiment can be executed interactively in a Python terminal. All the implicit functionalities described previously can also be explicitly called (e.g. there are methods `disable_hyperthreading`, `start_monitoring` and `perform_stress`).

An experiment can be parameterized by two means:

- An installation file. This is a Yaml file that can be used to describe how the setup phase should be done. Typically, it can contain the desired version for different libraries like OpenBLAS or OpenMPI, but also the duration of the warm-up or the frequency of the monitoring.
- An experiment file. These can be of any kind. A typical use case is to provide a CSV file where each line is a particular piece of the experiment (e.g. an individual call to `dgemm`) and the columns represent the parameters for these experiments (e.g. the sizes `M`, `N` and `K` used by `dgemm`).

9.2.2 Comparison with `execo`

In this section, we use a small example to illustrate key differences between `peanut` and `execo`. The goal is to write an experiment that will take several nodes on a given Grid'5000 cluster, compile the `CRoaring` library[2] and run one of its benchmarks.

First, Listing 9.1 shows such an experiment using `execo`. It should be executed as: `python script.py`

```
import execo
import execo_g5k as g5k

site = 'grenoble'
cluster = 'dahu'
nodes = 2
time = '00:20:00'
image = 'debian9-x64-base'

if __name__ == '__main__':
    query = "{cluster in ('%s')}/nodes=%d,walltime=%s" % (cluster, nodes, time)
    [(jobid, site)] = g5k.oarsub([(g5k.OarSubmission(resources=query, job_type='deploy'), site)])
    if jobid:
        print('Created job %d' % jobid)
        g5k.wait_oar_job_start(jobid, site)
        node_list = g5k.get_oar_job_nodes(jobid, site)
        g5k.deploy(g5k.Deployment(node_list, env_name=image), check_timeout=180)
        print('Terminated deployment')
        execo.Remote('apt update -qq', node_list).run()
        execo.Remote('DEBIAN_FRONTEND=noninteractive apt install -qq -y build-essential make git cmake
    ',
                node_list).run()
        execo.Remote('git clone https://github.com/RoaringBitmap/CRoaring.git CRoaring &&\
                cd CRoaring && git checkout v0.2.66', node_list).run()
        execo.Remote('cd CRoaring && mkdir -p build && cd build && cmake .. && make -j', node_list).run()
        print('Terminated installation')
        p = execo.Remote('cd CRoaring && ./build/benchmarks/real_bitmaps_benchmark ./benchmarks/
    realdata/census-income',
                node_list).run()
        for proc in p.processes:
            print(proc.host.address)
            print(proc.stdout)
        print('Terminated experiment')
        g5k.oardel([(jobid, site)])
```

Listing 9.1: Small experiment example using `execo`.

[2] `https://github.com/RoaringBitmap/CRoaring`

This script, albeit fairly small, is already difficult to read in some places. For instance, in lines 11-12, one has to write a complex query as a string as follows:

```
OarSubmission("{cluster in ('dahu')}/nodes=2,walltime=00:20:00")
```

It would be much more pleasant to write it as follows:

```
OarSubmission(cluster="dahu", nodes=2, walltime="00:20:00")
```

Now, Listing 9.2 demonstrates how the same experiment can be rewritten using `peanut` in a much more concise and readable way.

It should be executed as: `peanut script.py run tocornebize \`
`--deploy debian9-x64-base --cluster dahu --nbnodes 2 \`
`--walltime 00:20:00`

```
1  import peanut
2
3  class MyExperiment(peanut.Job):
4      def setup(self):
5          self.apt_install('build-essential', 'make', 'git', 'cmake')
6          self.git_clone('https://github.com/RoaringBitmap/CRoaring.git', 'CRoaring', checkout='v0.2.66')
7          self.nodes.run('mkdir -p build', directory='CRoaring')
8          self.nodes.run('cmake .. && make -j', directory='CRoaring/build')
9
10     def run_exp(self):
11         output = self.nodes.run('./build/benchmarks/real_bitmaps_benchmark ./benchmarks/realdata/census
       -income',
12                                 directory='CRoaring')
13         for node, result in output.items():
14             print(node.host)
15             print(result.stdout)
```

Listing 9.2: Small experiment example using `peanut`.

The script is not only half as long, it is also arguably much easier to read. Furthermore, it accomplishes a lot more than Listing 9.1, as it produces a peanut archive that contains all the metadata related to the experiment.

9.2.3 `EnOSlib`, a promising alternative

A recent experiment engine, called `EnOSlib` [Sim18], appears to be promising. It is a Python library that allows an easy deployment and execution of experiments. The experiment scripts written with `EnOSlib` are concise and readable, thanks to well-designed abstractions. Compared to `peanut` which has been solely implemented and used for the Grid'5000 platform, `EnOSlib` offers multiple backends. One issue is

that it currently does not support the archive building of `peanut` with the automatic collection of experiment data and meta-data.

At the time we started working on `peanut`, we did not know about `EnOSlib`, which was at its early stage of development and not yet advertised. Today, we believe it would be better to join forces and port the main features of `peanut` in `EnOSlib`.

10 On the difficulties of experimentation

Suppose some researcher wants to evaluate the memory bandwidth of their laptop. A first way to answer this question could be to write a small program that allocates a buffer, then write some data on this buffer with the `memset` function while measuring the duration of this operation. The problem is that the time taken to make this memory write may not be representative. The following writes would very probably have different durations due to cache effects. Therefore, it would be better to make several measures that should then be carefully analyzed (maybe simply taking the average, or perhaps there are some *outliers* that should be removed). However, by doing so we only measure the performance of a write for a given size. The effective bandwidth could be very different with a smaller or a larger buffer. The natural solution here is to repeat these sequences of measures for several sizes.

A general advice shared by experimental scientists in such situations is to randomize the experiments. In general, this randomization should happen for:

- The parameter space (in this example, the set of sizes that are evaluated). The goal is to avoid bias, for instance sizes that are a power of two may lead to a different performance. Note that in some occasions, as this chapter will illustrate, it is desirable to bias the experiment towards some particular values.

- The experiment order (in this example, the order of the sizes). The rationale here is to avoid temporal perturbations. In particular, there are often at least two phases, a load build up which converges toward a steady state. There can also be changes that happen once the steady state is reached, e.g. caused by some external source. By randomizing the order of the experiments, it becomes much easier to recognize an eventual temporal perturbation simply by plotting the data.

In this chapter, we will discuss several lessons learned for conducting faithful experiments, most of the time the hard way.

10.1 Experimental setup

All the experiments presented in this chapter share a common setup. They have been repeated on several nodes with `peanut` and follow the same steps:

1. Deploy and install a fresh OS on the node.
2. Run the `stress` command for 10 minutes to warm the node.
3. Start a background process [CM21] to monitor the core frequencies and temperature every second.
4. On each core, run a custom code [SG21] to measure the durations of a given operation (either `dgemm` or several MPI functions, depending on the experiment).

Unless specified otherwise, we used nodes from the `dahu` cluster from Grid'5000[1]. Each of these nodes has two Intel Xeon Gold 6130 CPU, which are 16 core CPU from the Skylake generation. They have a base frequency of $2.1\,\mathrm{GHz}$ and a turbo frequency of up to $3.7\,\mathrm{GHz}$, but their turbo frequency is limited to $2.4\,\mathrm{GHz}$ when their 16 cores are active and in AVX2 mode[2]. We have used OpenBLAS [OpenBLAS] version 0.3.1 and OpenMPI [OpenMPI] version 2.0.2 compiled with GCC version 6.3.0 on a Debian 9 installation with kernel version 4.9.0.

10.2 Defining the parameter space

Two families of experiments are discussed in this chapter: (1) MPI operations such as calls to `MPI_Recv` or `MPI_Send`, their duration is proportional to S, the size of the buffer that is being communicated, and (2) calls to the `dgemm` function, whose duration is proportional to the product of the sizes MNK but also depends on individual interactions of those sizes.

An experiment consists in a sequence of calls to the function of interest, with various sizes as parameters. The goal of this section is to discuss how and why the parameters of these sequences are generated.

[1] `https://www.grid5000.fr/w/Grenoble:Hardware`
[2] `https://en.wikichip.org/wiki/intel/xeon_gold/6130`

10.2.1 MPI communications

If we assume that the duration of a communication is linear in the amount of data being sent, the easiest way to sample the data is to only measure two sizes, one very small buffer (e.g. $1\,\mathrm{B}$) and one very large buffer (e.g. $1\,\mathrm{GB}$). To avoid any bias, we could even try several small and large buffers with slight differences in their respective sizes. However, we saw in Part I that this linearity assumption does not hold, there are large discontinuities. We therefore need to also sample sizes between the two extreme values to (1) make sure that all the breakpoints are visible in our dataset and (2) perform one linear regression in each linear zone.

Given this requirement, the natural sampling method would be a uniform sampling, taking $S \sim \mathcal{U}\left(1, 10^9\right)$. However, in our experiments, we found out that the breakpoints are not uniformly spread, but rather exponentially. For instance, the `MPI_Send` on the `dahu` cluster has four breakpoints: $8.14\,\mathrm{kB}$, $34.0\,\mathrm{kB}$, $63.8\,\mathrm{kB}$ and $285\,\mathrm{MB}$. If the sizes of the experiment were uniformly sampled, we would very probably miss the smaller breakpoints: by sampling uniformly and independently 1000 numbers in the interval $[1, 10^9]$, the probability to have at least one number smaller or equal to 10^5 is a bit less than $10\,\%$. The reason for these exponentially spread breakpoints likely comes from the hardware. Typically, each layer of the memory hierarchy is one order of magnitude larger than the previous layer. For instance, each core of a `dahu` node has $32\,\mathrm{KiB}$ L1 instruction and data caches and one $1\,\mathrm{MiB}$ L2 cache. Then, the 16 cores of a same CPU share a $22\,\mathrm{MiB}$ L3 cache and a $93\,\mathrm{GiB}$ memory.

For these reasons, we made the choice to sample the sizes exponentially in the considered interval. More precisely, each size S is sampled as:

$$S \sim 10^{\mathcal{U}(0,9)}$$

10.2.2 Function `dgemm`

The situation is different with the `dgemm` function. We did not observe any breakpoint in the performance plots as for the MPI communications. Hence, we did not have any reason to use an exponential sampling. A first solution would therefore consist in taking $M = N = K$ and sampling the product MNK uniformly in some interval. This could be sufficient for a simple linear regression with only one parameter, but as discussed in Part I we need several coefficients of the polynomial. For this reason, we have to cover a larger zone of the parameter space.

Two options have been considered. Suppose we would like the three sizes M, N, K to be smaller or equal to some constant Σ and their product MNK to be smaller or equal to some other constant Π.

Independent sizes Each of the three sizes M, N and K is sampled independently and uniformly in the desired interval:

1. $M \sim \mathcal{U}(1, \Sigma)$ and $N \sim \mathcal{U}(1, \Sigma)$ and $K \sim \mathcal{U}(1, \Sigma)$
2. Repeat until $MNK \leq \Pi$
3. Return (M, N, K)

This is the easiest method to implement. With this approach, the product MNK is not uniform, it is heavily skewed towards the small values. Additionally, we use rejection sampling to make sure that the product of the sizes does not get too large.

Uniform product We start by sampling the size product P uniformly in the desired interval, then the sizes M, N and K are sampled randomly to get (approximately) the correct product:

1. $P \sim \mathcal{U}(1, \Pi)$
2. $A \sim \mathcal{U}\left(1, \sqrt[3]{P}\right)$
3. $B \sim \mathcal{U}\left(1, \sqrt{\frac{P}{A}}\right)$
4. $C = \frac{P}{AB}$
5. Repeat steps 2 to 4 until $A \leq \Sigma$ and $B \leq \Sigma$ and $C \leq \Sigma$
6. Return the six possible permutations: $(M, N, K) = (A, B, C), (C, A, B), \ldots$

The goal of returning all the permutations of the sizes is to avoid any bias in the sampling procedure, since they are not generated independently of each other. We also use rejection sampling to make sure that one of the sizes does not get too large.

These two generation methods are illustrated in Figure 10.1. A list of $100{,}000$ size tuples was sampled with each approach.

The left plot shows that as expected, the product MNK appears to have a large left-skew with the first approach and to be uniformly distributed with the second approach. The right plot is more surprising, since the parameter M is not uniformly distributed with the first approach. The reason is the rejection sampling, large values

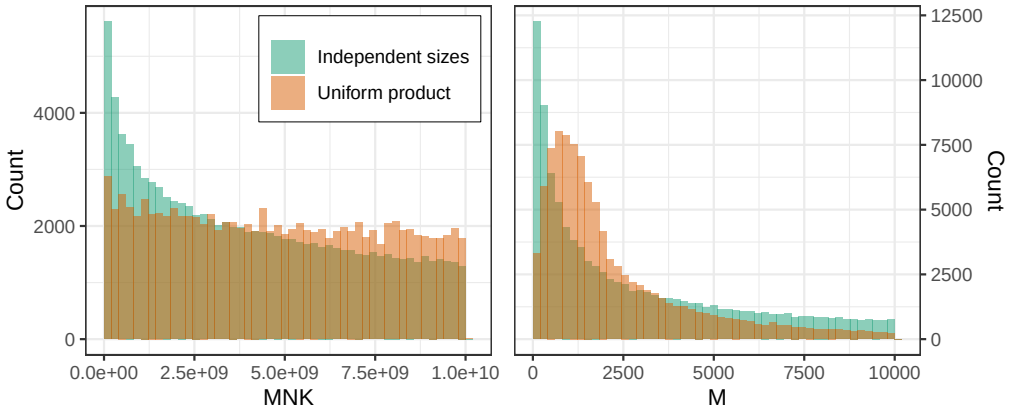


Figure 10.1.: Distribution of the product MNK and the size M with the two generation methods. The maximal product is set to 10^{10} and the maximal size to $10{,}000$.

for M are more likely to give a product MNK above the specified limit and thus to be rejected.

Since the product MNK is the most significant factor for the duration of `dgemm`, it is more natural to use a uniform distribution for this term, we therefore made the choice to use the second approach for generating experiment files for the `dgemm` experiments, with two minor modifications:

- Instead of sampling the product uniformly with $MNK \sim \mathcal{U}(1, S)$, we made the choice to use a quasi-random approach. We first compute γ different values that are uniformly but deterministically spread in the given interval, i.e. we compute the set $\left\{S, S - \frac{S}{\gamma}, S - 2\frac{S}{\gamma}, \dots\right\}$ (typically, $\gamma = 30$). Then we add a random noise independently to each of these sizes. The goal is to ensure a similar (yet still random) distribution of the products MNK each time we generate a new experiment file. This approach is similar to Latin hypercube sampling (LHS).

- In addition of the size tuples randomly generated, we systematically add a few tuples to the list, like $(1, 1, 1)$ or $(2048, 2048, 2048)$. The goal is to ensure that we have a few identical calls to `dgemm` in every experiment, in case we want a fine comparison.

10.3 Randomizing the order

The network model in SMPI needs to be instantiated with a careful calibration of the MPI communication performance, as presented by Degomme et al. [Deg+17]. This was done with an MPI program created by the Simgrid team that performed a sequence of measures with two hosts. Several kinds of measures were implemented: the *recv* (a call to `MPI_Recv` with waiting to avoid late senders), the *isend* (a call to `MPI_Isend`), the *pingpong* (a call to `MPI_Send` followed by a call to `MPI_Recv` to get the round-trip time) as well as several more minor MPI primitives.

---

read the sequence of sizes S_1, typically $|S_1| \approx 1000$;

$S_2 := \underbrace{S_1 \cdot S_1 \cdot \cdots \cdot S_1}_{N \text{ concatenations, typically } N \approx 50}$;

for *each kind of measure (recv, isend, pingpong, etc.)* **do**

  for $s \in S_2$ **do**

    perform the measure $K \approx 10$ times and output each individual duration

---

Although the sequence of sizes S_1 is a random sequence, there are still two obvious biases in this experiment. First, the final sequence S_2 is a concatenation of several instances of S_1, so the same (random) order will be used in these N runs. Then, the different kind of measures are performed one after the other.

In a first step towards a better methodology, we started by shuffling entirely the sequence S_2 after the concatenation. The observed durations for function `MPI_Recv` with both methods are presented in Figure 10.2. There is no obvious difference here, in both cases the duration is piecewise linear in the message size and several modes are present for the small and medium messages.

To compute a network model for SMPI, we need to perform a (segmented) linear regression on this data. One assumption for the simple least-square regression is that the noise should be normally distributed, which is clearly not the case with our dataset, the noise is multi-modal. One simple solution for this is to compute the average duration for each message size, which all have many measures (500 in this figure). With the central limit theorem, assuming that the measures for similar message sizes are independent and identically distributed, their sample average is normally distributed. This means that by averaging the data, we should get a normal noise which would allow us to compute a linear regression.

The aggregated data is presented in Figure 10.3. With the shuffled experiment (right plot), the average durations have a single-mode, as expected. However,

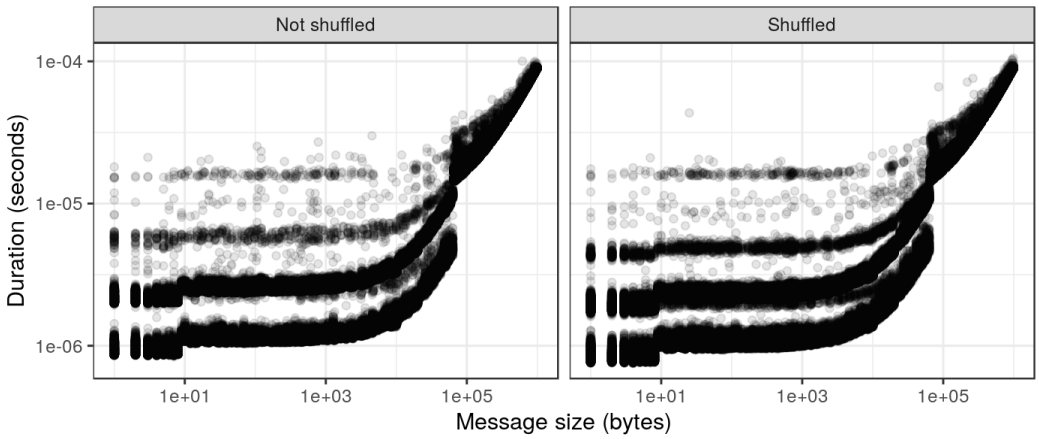


Figure 10.2.: The duration of `MPI_Recv` is piecewise linear, with several modes for small messages.

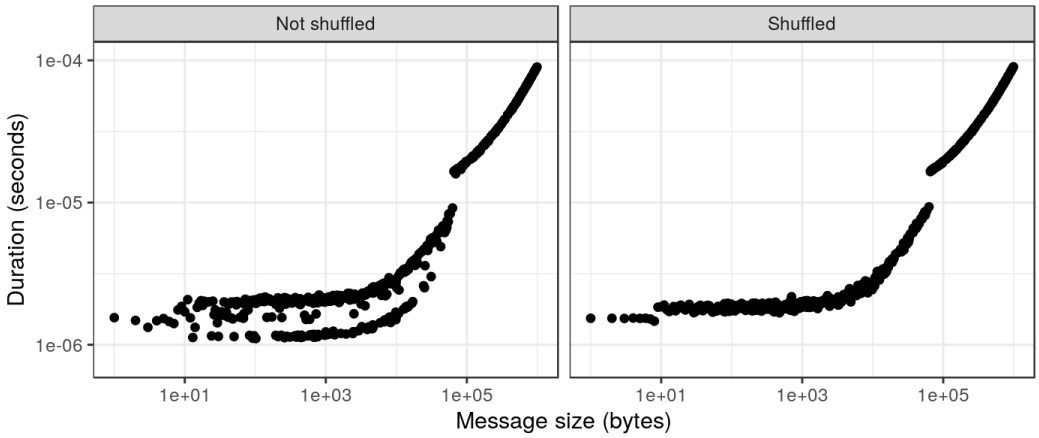


Figure 10.3.: The average durations of `MPI_Recv` in the non-shuffled case still show several modes, which should not happen according to the central limit theorem.

in the non-shuffled case (left plot), at least two modes are clearly present. This contradicts the conclusion of the central limit theorem, thereby threatening the analysis of the experiment. This is confirmed by Figure 10.4, where we zoomed on a few distinct message sizes between 700 B and 800 B. Each point is the duration of an individual call to `MPI_Recv`, the crosses represent the average durations. In the shuffled experiment, on the right, the duration distributions are similar for all message sizes, with two modes clearly identifiable and the average in between. In the non-shuffled case, on the left, the durations of three message sizes are different, namely 703 B, 767 B and 779 B. For these three calls, the distributions have only one mode, so their average durations are significantly shifted. The assumption of identical distribution for these different message sizes is clearly not satisfied here.

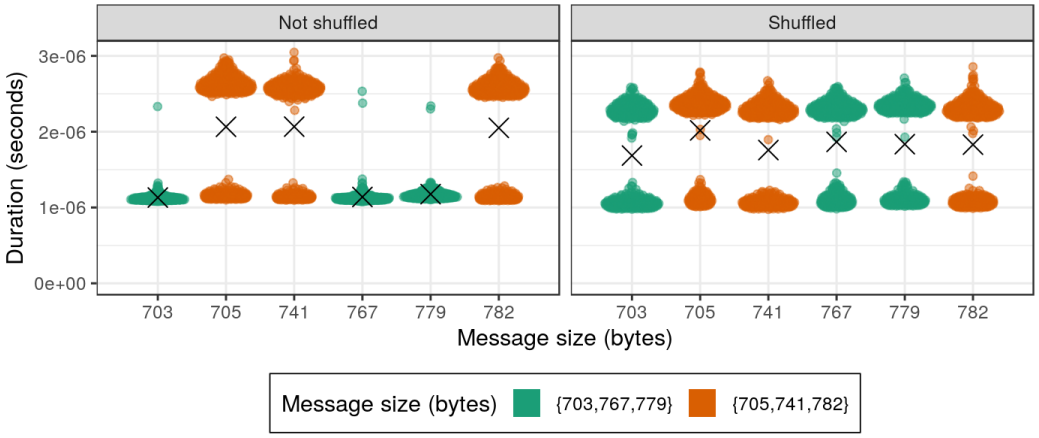


Figure 10.4.: Distribution of the `MPI_Recv` durations for six different message sizes between 700 B and 800 B. The durations are not identically distributed in the non-shuffled case. Durations truncated to 4 µs for a better readability.

Since shuffling correctly the experiments prevents the occurrence of this issue, a possible reason could be that the individual calls to `MPI_Recv` are not truly independent. The sequence before the calls for the sizes like 703 B, 767 B or 779 B would lead to particularly good conditions and thus an excellent performance, which does not systematically happen in the shuffled case because of the proper randomization. However, we could not identify anything suspect regarding the message sizes of the calls made just before these high-performance calls. Some of them had messages of a few bytes, some others had messages of several hundreds kilobytes.

In Figure 10.5, we present the temporal evolution of the durations for the calls to `MPI_Recv` made with the sizes presented in Figure 10.4. In the non-shuffled case, we can identify a temporal pattern. During the first 20 seconds of the experiment, the calls with sizes 705 B, 741 B and 782 B (in orange) have durations above 2 µs for a large fraction of them, only a small part have durations below 1.5 µs. After the

20-second timestamp, this suddenly changes, there are at least two time windows where all these calls have a low duration. Even outside these time windows, a much larger fraction of these calls have low durations. This temporal pattern is not visible in the other cases.

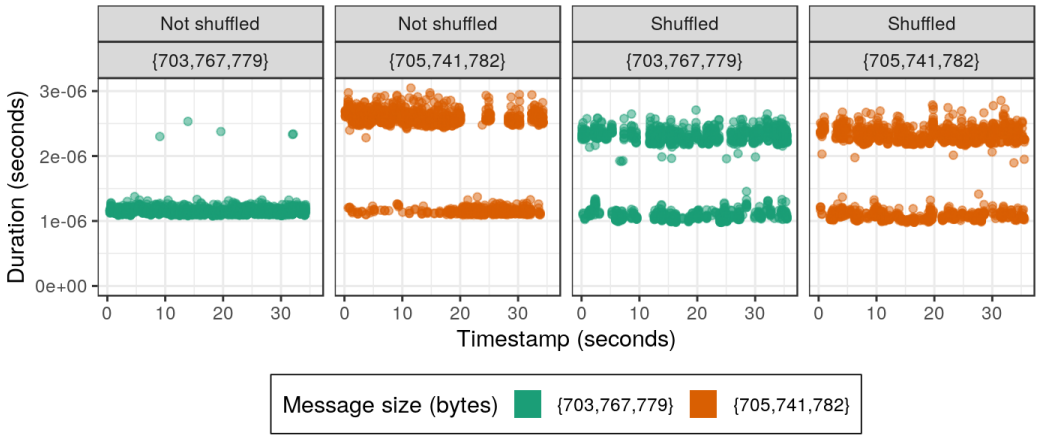


Figure 10.5.: Temporal evolution of the `MPI_Recv` durations for six different message sizes between $700\,\mathrm{B}$ and $800\,\mathrm{B}$. A temporal pattern can be observed. Durations truncated to $4\,\mu\mathrm{s}$ for a better readability.

Another view of the non-shuffled experiment is presented in Figure 10.6. Now all the calls with a size lower than $1\,\mathrm{kB}$ are shown, but only a small fraction of the whole experiment is displayed. The calls to `MPI_Recv` can be divided into two groups depending on their durations, lower (in green) or greater (in orange) than $1.7\,\mu\mathrm{s}$. The rug plot on the top of the figure highlights the position in time of each of these `MPI_Recv` calls. Although there are slow and fast calls uniformly distributed during this time window, there appears to be some clusters where nearly all the calls are of the same kind.

A possible explanation for such a temporal pattern could be an external perturbation that happens at a regular interval. Since we are measuring very small durations, the culprit would be a short but frequent noise (e.g. a system daemon). This is very well explained by Petrini et al. [PKP03]:

> Substantial performance loss occurs when an application resonates with system noise: high-frequency, fine-grained noise affects only fine-grained applications; low-frequency, coarse-grained noise affects only coarse-grained applications.

A similar temporal pattern can be observed in the shuffled experiment. However, since the order of the sizes is completely random, it affects them all equally.

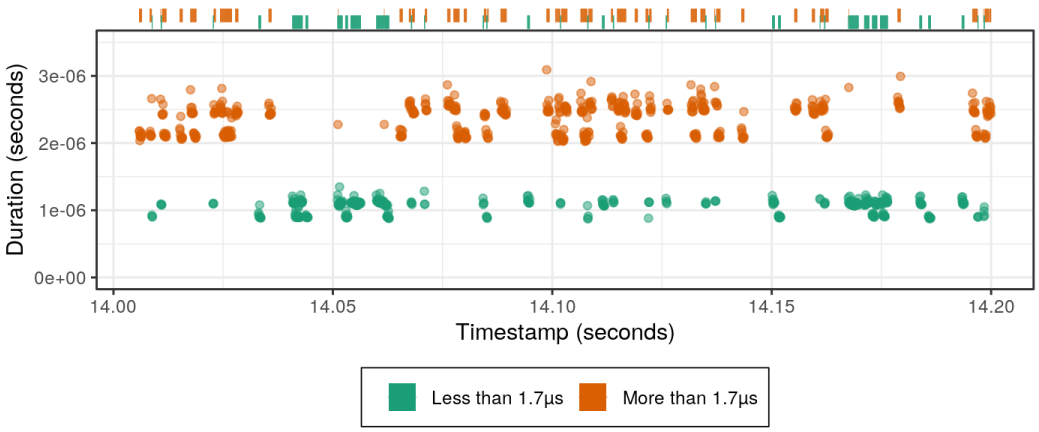


Figure 10.6.: Temporal evolution of the `MPI_Recv` durations for all the sizes between $1\,\mathrm{B}$ and $1\,\mathrm{kB}$ during a $0.2\,\mathrm{s}$ time window of the non-shuffled experiment. Another temporal pattern can be observed. Durations truncated to $4\,\mu\mathrm{s}$ for a better readability.

Later on, we went a step further in improving the methodology of this experiment by also randomizing the outer loop, i.e. the measures are now shuffled, there are *pingpong* measures between *isend* and *recv* measures and vice-versa. This change did not bring any noticeable effect on our observations.

The experiments described in this section were performed in 2018. Two years later, we were unable to replicate this phenomenon, despite using the same MPI implementation and an identical cluster: the averaged data has a single mode, even in the non-shuffled case. We cannot state with certainty the reason this behavior disappeared. It could be due to a change in our calibration program, or on the platform itself. This motivates the implementation of performance non-regression tests, as discussed in Chapter 11.

10.4 Randomizing the sizes

The calibration measures for the `dgemm` function are done with a random sequence of tuples, as discussed in Section 10.2.2. This sequence is properly shuffled, so we eliminated the possible experimental bias discussed in Section 10.3. In this section, we will approach two difficulties that were encountered with the sizes themselves (as opposed to the order of the sequence).

10.4.1 Effect of the experiment file

Through the numerous `dgemm` calibrations that were performed, we eventually realized that the set of sizes used for the experiment had a significant effect on the statistical model obtained with these measures. To demonstrate this effect, we have generated three different experiment files using exactly the same generation method described previously. These three experiments, named `A`, `B` and `C`, were repeated several dozens of times during a weekend in a random order. They have been carried on 8 different nodes of the `dahu` cluster, for a total of 16 different processors, the results are extremely similar for all of them.

The average `dgemm` performance observed in each experiment is reported in Figure 10.7. Some performance variability can be observed, the most efficient runs are approximately $3\,\%$ faster than the least efficient ones. A large fraction of this variability appears to be significantly caused by the experiment file, since all the runs made with file `C` have a higher performance than those made with file `B`, which are themselves more efficient than those made with file `A`. Thanks to the proper randomization of the experiments, we can rule out any temporal bias.

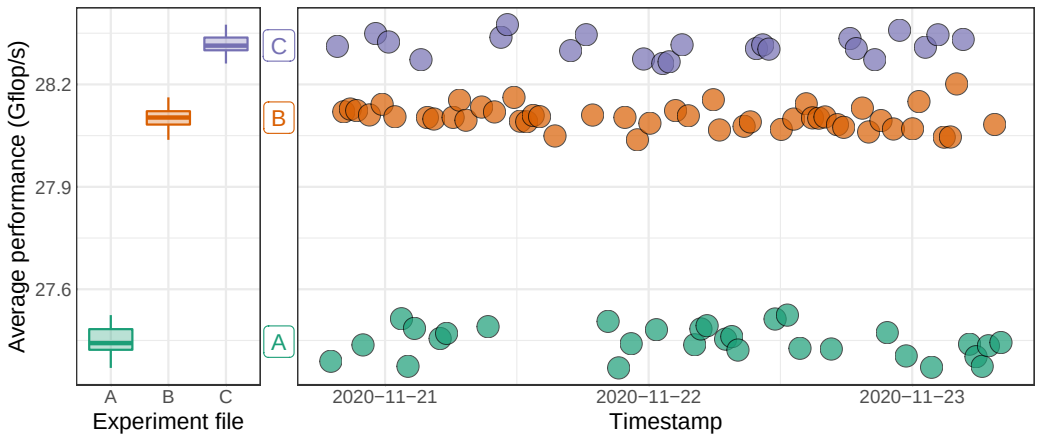


Figure 10.7.: Average performance observed on CPU 1 of `dahu-5`, each point represents one experiment. A significant part of the variability is due to the choice of the experiment file.

The effective performance is not the only aggregated metric affected by the choice of the experiment file. The distributions of two regression coefficients are represented in Figure 10.8, namely the coefficients corresponding to the products MNK and NK (the effect of the experiment file on the coefficients for MK and MN is extremely similar to NK). It appears here that the experiment file causing the highest performance gives the highest cubic coefficient and the lowest quadratic

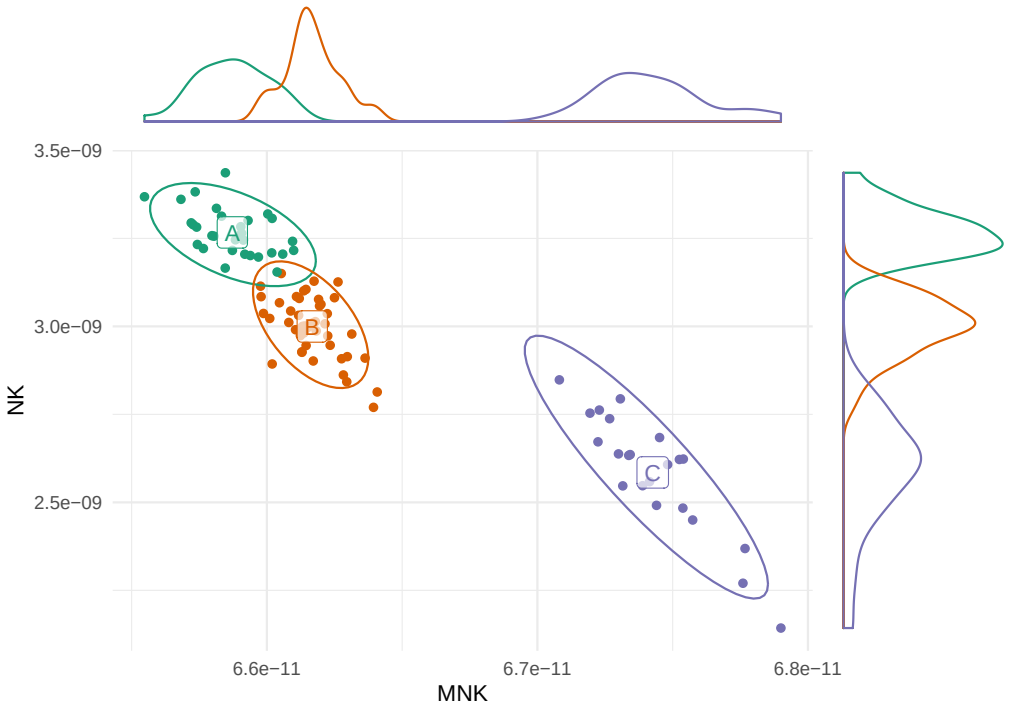


Figure 10.8.: Distribution of two of the regression parameters for CPU 1 of dahu-5, each point represents one experiment. The experiment file has a clear effect on the generated model.

coefficients. In other words, this means that with this experiment file, a larger fraction of the `dgemm` durations is explained by the cubic coefficient.

These observations suggest that experiments `A` and `B` may be less cache-friendly than experiment `C`, since in a matrix product the number of arithmetic operations grows cubically with the size of the input whereas the number of memory accesses grows quadratically.

A non-aggregated view of the data is presented in Figure 10.9, each point represents one individual call to `dgemm`. It appears that most of the calls in the three experiments have extremely similar durations for a given product MNK. However, a small fraction of the `dgemm` calls were significantly slower than the others with experiments `A` and `B`. All these calls have are for a tall and skinny matrix, with $K \geq 3000$, which corroborates the hypothesis of a bad cache utilization.

A final argument towards this hypothesis is presented with Figure 10.10, the average DRAM power consumption during each run is presented. Similarly to Figure 10.7, there is a clear difference between the three experiments that cannot be explained by any temporal perturbation. Experiment `C`, which was the fastest, has the smallest

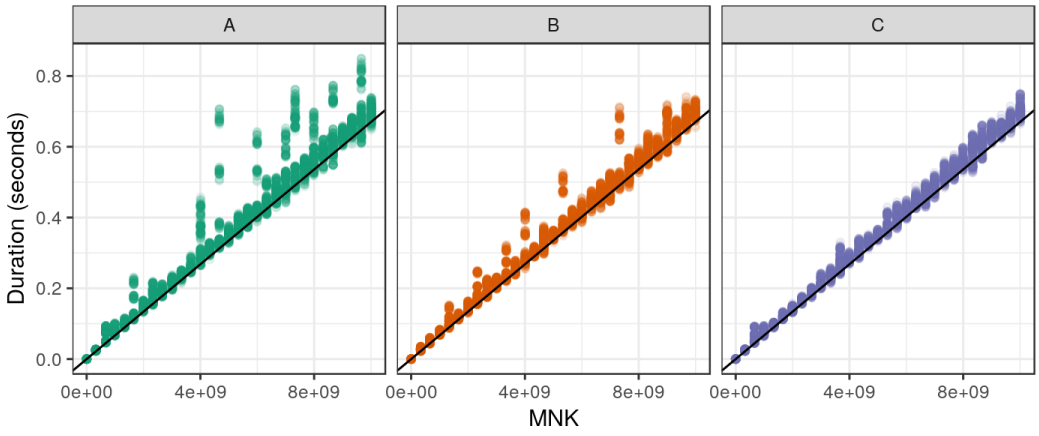


Figure 10.9.: Durations of individual `dgemm` calls for CPU 1 of `dahu-5`. Several calls have significantly longer durations than others. Identical black line on the three plots, with slope 6.7×10^{-11}.

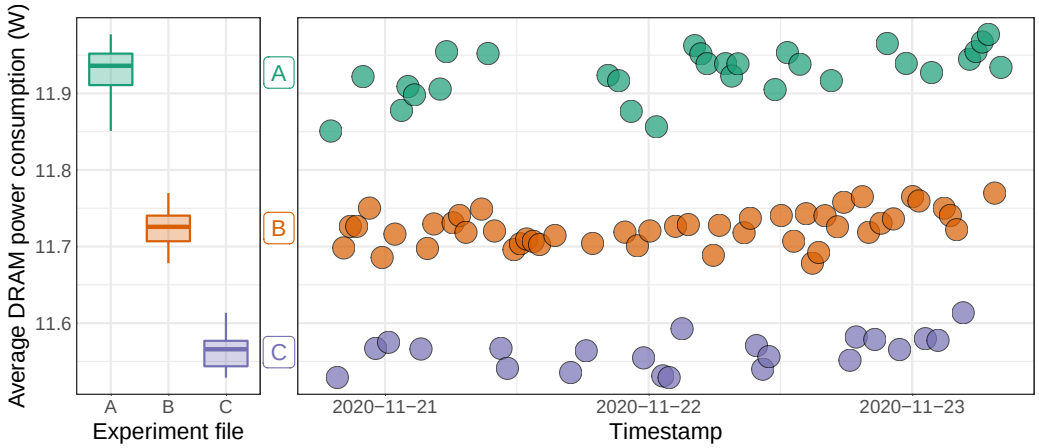


Figure 10.10.: Average DRAM power consumption observed on CPU 1 of `dahu-5`, each point represents one experiment.

DRAM power consumption. This suggests that the memory was used less intensively with this experiment, i.e. there was a better cache utilization.

In this section, we compared several `dgemm` experiments performed with three sets of sizes. These sets have been generated according to the same statistical distribution, yet they lead to significantly different `dgemm` models. We would like to stress that this discrepancy of the resulting models is due to a difference in the experimental conditions and not (only) to a statistical artifact. We proposed the hypothesis of a poor cache utilization, but other possibilities should not be dismissed, since this study was only observational. Our hypothesis would need to be confirmed or refuted with a properly designed experimental study.

10.4.2 Effect of the experiment file generation method

Section 10.4.1 has shown that the experiment file had a significant impact on the experimental conditions which affected the resulting statistical model. We generated three different sequence of sizes using the *uniform product* method and performed several runs with each of these sequences.

Now, we investigate briefly the effect of the generation method itself. We compare the *independent sizes* and the *uniform product* methods described in Section 10.2. For each of these methods, we generated several experiment files and performed one run with each of these files.

Although there is a large variability, which is due to the use of several experiment files, it appears that the two generation methods lead to significantly different experimental conditions, as shown by Figure 10.11. With the *uniform product* method, `dgemm` average performance is higher and the DRAM power consumption is lower.

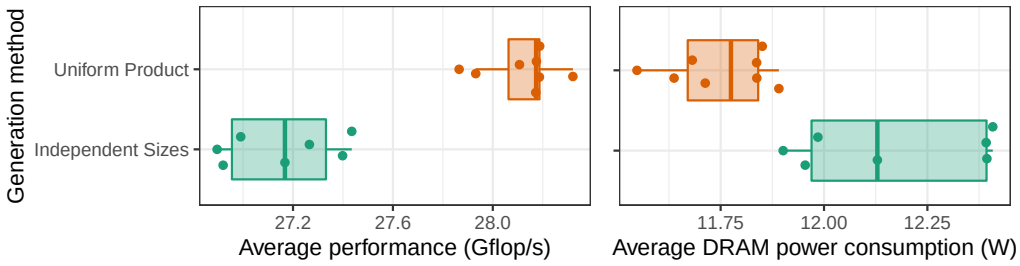


Figure 10.11.: Average `dgemm` performance and power consumption observed on CPU 1 of `dahu-5`, each point represents one experiment.

10.4.3 Effect of calibrating with a fixed size

In the experiments described in Section 10.4.1 and Section 10.4.2, the three `dgemm` parameters M, N and K can take arbitrary values, to avoid any bias. One of the main reasons we make such measures is to generate a statistical model of `dgemm` durations for simulating HPL. For this model to be faithful, the experimental conditions of our measures must be as realistic as possible compared to what happens during HPL execution. However, nearly all the `dgemm` calls performed in HPL use the same value for the parameter K, equal to HPL block size (i.e. the parameter `NB`). For this reason, biasing the calibrations by using a fixed value for K could help to improve

the simulation accuracy. This section investigates the question. We have generated five sets of experiment files:

Random This is the usual *uniform product* generation procedure already discussed in previous sections.

Fixed K We modified the *uniform product* procedure to have a constant value for K. We generated three sets of files, with $K = 128$, $K = 256$ and $K = 512$.

Several fixed K We modified the *uniform product* procedure to have the value of K chosen randomly in {128,256,512}. This is equivalent to concatenating three files generated with the *fixed K* method and then shuffling the resulting file.

For each of the five experiment kinds, we have generated several dozens of experiment files. Then, we performed one experiment with each of these files in a random order during a weekend on two nodes of the `dahu` cluster for a total of four different processors. Again the results are similar for all of them, so we will focus on a single processor.

Figure 10.12 presents the observed `dgemm` performance with the five experiments. We can observe that again, the generation method for the size sequence has an enormous effect on the experiment. First, using a fixed value for K reduces very significantly the inter-run performance variability. The average performance is also greatly affected, it is the highest with K fixed to 128, the lowest with K fixed to 256 or 512 or with the random generation, and it is intermediate with K randomly sampled in $\{128, 256, 512\}$.

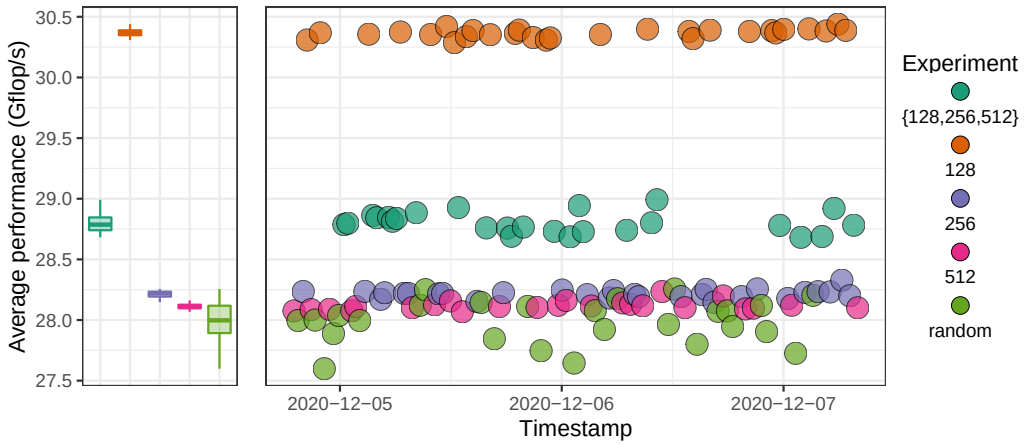


Figure 10.12.: Average `dgemm` performance observed on CPU 1 of `dahu-5`, each point represents one experiment.

The monitoring data collected during the experiments also reveals interesting differences. The average CPU frequency is reported in Figure 10.13. It appears that the frequency is the highest with $K = 128$ and with the random experiment. It is significantly lower with K chosen randomly among the three sizes, and even lower with $K = 256$ and $K = 512$. Interestingly, there is a positive correlation between the frequency and `dgemm` performance, but the random experiment is a clear exception as it leads to a relatively low performance and high frequency.

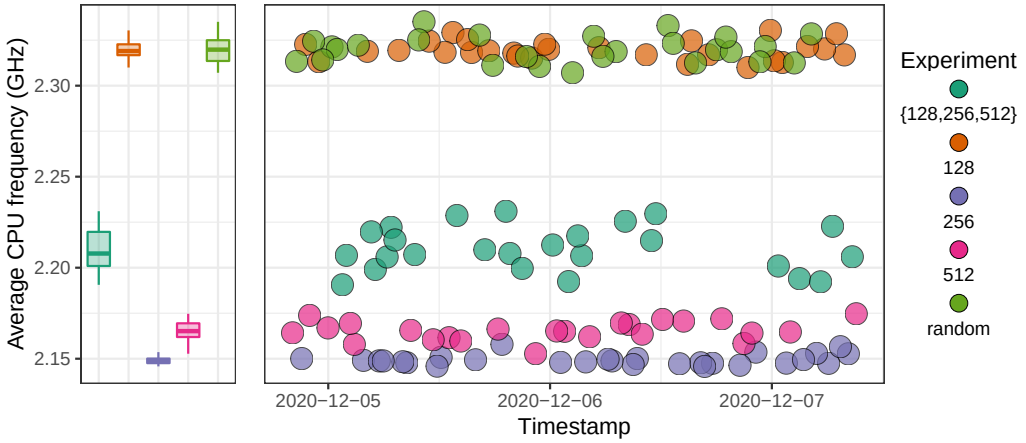


Figure 10.13.: Average CPU frequency, observed on CPU 1 of `dahu-5`, each point represents one experiment.

The average CPU power consumption, presented in Figure 10.14, is particularly notable, and peculiar. The random experiment has a power consumption significantly lower and more variable than the four other experiments that are all extremely stable, with nearly no inter-run variability. This observation is very counter-intuitive, since the CPU power consumption is in general proportional to the CPU frequency [Hei+17]. This only happens on the CPU 1 of the two nodes we tested, the power consumption of the CPU 0 is extremely stable and similar for the five experiment kinds.

The five experiments exhibit very different average DRAM power consumption, as depicted in Figure 10.15. The experiment with $K = 128$ is the most energy-hungry, followed by the experiment with $K \in \{128, 256, 512\}$, then the experiments with a random K, $K = 256$ and $K = 512$. It is interesting to note that the experiment with the highest DRAM power consumption is also the one with the highest average `dgemm` performance, which is the opposite of what was observed in Section 10.4.1.

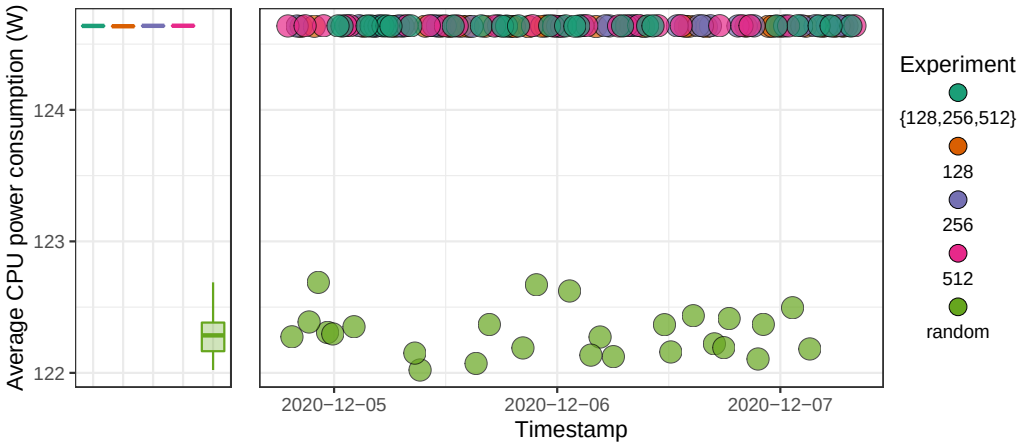


Figure 10.14.: Average CPU power consumption, observed on CPU 1 of `dahu-5`, each point represents one experiment.

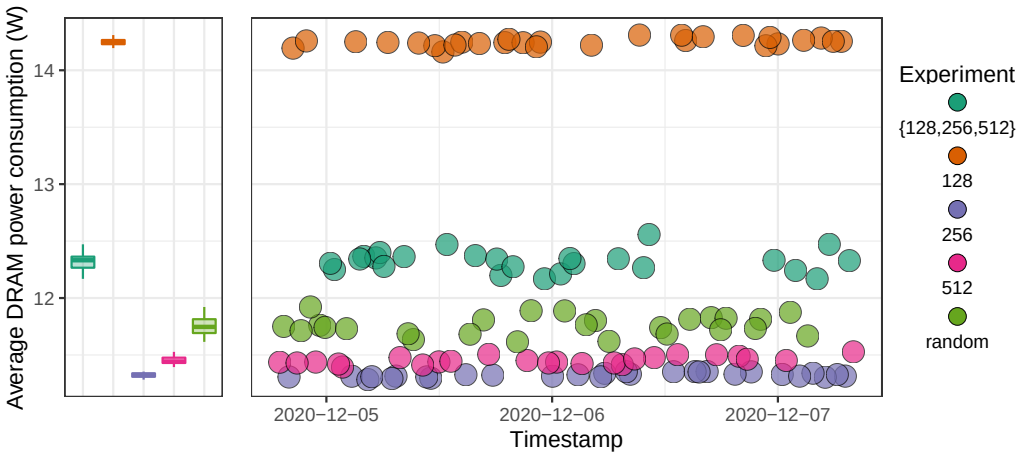


Figure 10.15.: Average DRAM power consumption, observed on CPU 1 of `dahu-5`, each point represents one experiment.

Finally, the performance of individual `dgemm` calls are presented in Figure 10.16. We made the observation earlier that there was much less inter-run variability when the value of K was fixed. This plot shows that there is also significantly less intra-run variability. Furthermore, we can compare the performance of `dgemm` for a given value of K. With $K = 128$, the performance is higher in the experiment where all the calls are done with $K = 128$ than in the experiment with $K \in \{128, 256, 512\}$. With the two other values, $K = 256$ and $K = 512$, this is the opposite, the performance is higher in the mixed experiment than in the experiment with only one K value. This shows that the durations of individual `dgemm` calls are not independent, one call can be faster or slower depending on the calls previously made.

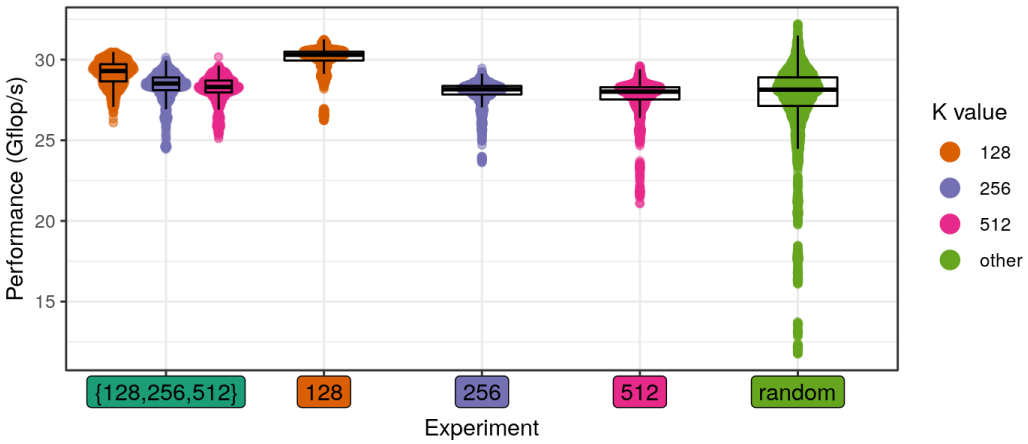


Figure 10.16.: Durations of individual `dgemm` calls for CPU 1 of `dahu-5`.

In this section, we demonstrated once again that the sampling method has an important effect on the measured performance. This will affect any statistical model that relies on the measured data, not because of a statistical bias, but because of an experimental bias. Such a bias might be desirable, for instance it should help improve the prediction accuracy of our HPL simulations. It comes at a price though, since we need to perform a new `dgemm` calibration if we want to simulate HPL with another block size.

10.5 Randomizing the data

The work presented in this section has been published as a technical report [CL19]. The content of this section is therefore a near-verbatim copy of this report.

This experiment comes, yet again, from an unfortunate phenomenon we stumbled upon when calibrating the platform for our simulations. Our predictions were

wrong, so we investigated further and we noticed a significant mismatch between the durations measured with our calibration code and the durations observed in HPL. We found out that the performance of the `dgemm` function depends on the content of the matrix, which was unexpected. All the experiments described in this section were done with matrices of fixed size 2048×2048, but the phenomenon we observed is reproducible with any matrix size.

10.5.1 Randomization of the matrix initialization

The three matrices are allocated once at the start of the program as a buffer of size N^2 with $N = 2,048$. Then, their content is initialized in three different ways, depending on the experiment:

1. All the elements of the matrices are equal to some constant. We have tested with three different values: 0, 0.987 and 1.
2. The elements of the matrices are an increasing sequence in the interval $[0, 1]$. More precisely, `mat[i] = i/(N^2-1)` for i in $[0, N^2 - 1]$.
3. Each element of the matrix is randomly and uniformly sampled in the interval $[0, 1]$.

Figure 10.17 shows the evolution of the `dgemm` durations during the experiment. A clear temporal pattern can be distinguished, the performance is oscillating. Furthermore, several layers can be seen, the durations of `dgemm` are the highest when the matrices are initialized randomly and the lowest when they are initialized with a constant value. The sequential initialization is in between.

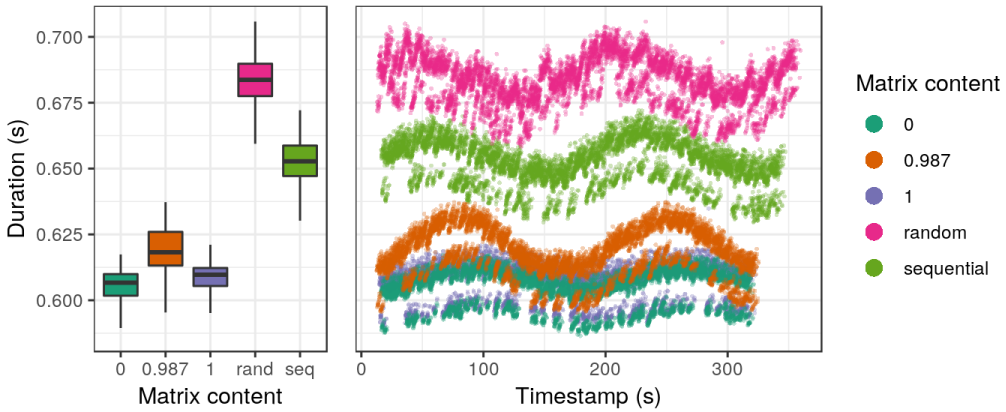


Figure 10.17.: Durations of individual `dgemm` calls are lower with constant values in the matrices.

Such an observation was unforeseen. The function `dgemm` implements the usual matrix product with cubic complexity. The control flow of the function does not depend on the matrix content, so we did not expect its duration to be data-dependent.

The observations we have made on `dgemm` performance can be explained by Figure 10.18 which shows the evolution and the distribution of the core frequencies during the experiment. There is a clear correlation between the frequencies and `dgemm` performance: the random initialization produces lower frequencies whereas the constant initialization gives higher frequencies. A similar temporal pattern can also be distinguished with clear oscillations.

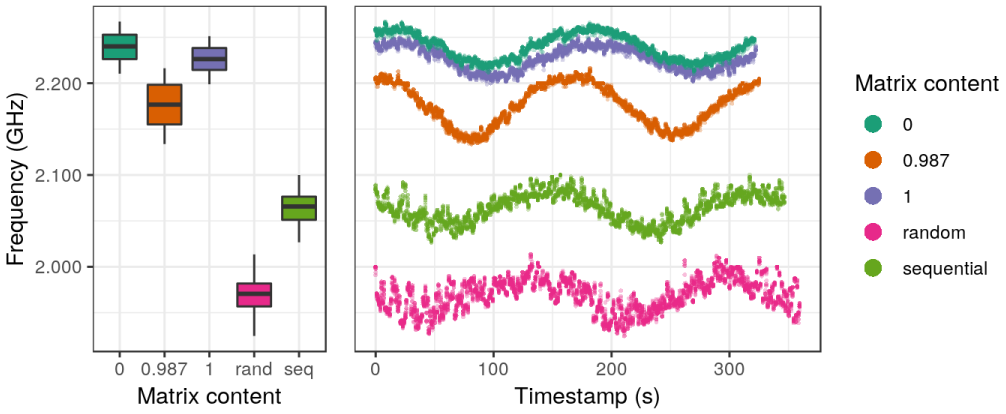


Figure 10.18.: Core frequencies are higher with constant values in the matrices.

This experiment has been repeated on other Grid'5000 clusters, each time on at least four distinct nodes. Table 10.1 gives a summary of our observations. Five other clusters show a similar behavior, the performance of `dgemm` is higher when the matrices are generated with a constant value. However, for five other clusters, this phenomenon could not be observed, the matrix content had no impact on the performance.

10.5.2 Hypotheses

Several hypotheses were discussed to explain this unexpected phenomenon.

There could be a small cache on the floating-point unit of the cores to memorize the results of frequent operations. This could explain why the durations were higher when the matrices were initialized randomly, but this does not explain why the sequential initialization is in between.

Table 10.1.: Observation of the performance anomaly on Grid'5000 clusters.

Cluster	CPU	Generation	Release date	Anomaly
nova	Intel Xeon E5-2620 v4	Broadwell	Q1'12	no
taurus	Intel Xeon E5-2630	Sandy Bridge	Q1'12	no
ecotype	Intel Xeon E5-2630L v4	Broadwell	Q1'12	yes
paranoia	Intel Xeon E5-2660 v2	Ivy Bridge	Q3'13	no
parasilo	Intel Xeon E5-2630 v3	Haswell	Q3'14	yes
chiclet	AMD EPYC 7301	-	Q2'17	no
dahu	Intel Xeon Gold 6130	Skylake	Q3'17	yes
yeti	Intel Xeon Gold 6130	Skylake	Q3'17	yes
pyxis	ARM ThunderX2 99xx	-	Q2'18	no
gros	Intel Xeon Gold 5220	Cascade Lake	Q2'19	yes
troll	Intel Xeon Gold 5218	Cascade Lake	Q2'19	yes

This could be due to kernel same page merging (KSM), a mechanism that allows the kernel to share identical memory pages between different processes. Again, this would explain the difference between the random initialization and the constant one, but not why the sequential initialization gives intermediate performance.

A last hypothesis is the power consumption of the cores. Each state change of the electronic gates of the CPU costs an energy overhead. In the case of the constant initialization, the registers will change less often during the execution of `dgemm`, in comparison with the random initialization. Thus, with the constant initialization, the processor cores would be able to maintain a higher frequency while respecting the thermal design power (TDP), with the random initialization the frequency would be throttled more aggressively and thus the performance would be lower. As for the sequential initialization, we can imagine that we have a locality effect: nearby elements of the matrices will have more bits in common, this would cause fewer bit flips than the random initialization but more bit flips than the constant initialization and thus an intermediate performance.

10.5.3 Testing the bit-flip hypothesis

To test the hypothesis that the lower frequencies are caused by more frequent bit flips in the processor, the matrix initialization has been changed. Now, each element of the matrix is randomly and uniformly sampled in the interval $[0, 1]$. Then a bit mask is applied on the lower order bits of their mantissa. As a result, all the elements of the matrices have some bits in common. This method is illustrated in Figure 10.19, the mantissa of the matrix elements (in blue) is at first completely random, then we

apply a mask so that the right-most bits (in green) become deterministic[3]. Several mask sizes have been tested, from 0 (the elements are left unchanged) to 53 (the mantissa becomes completely deterministic, all the elements are equal).

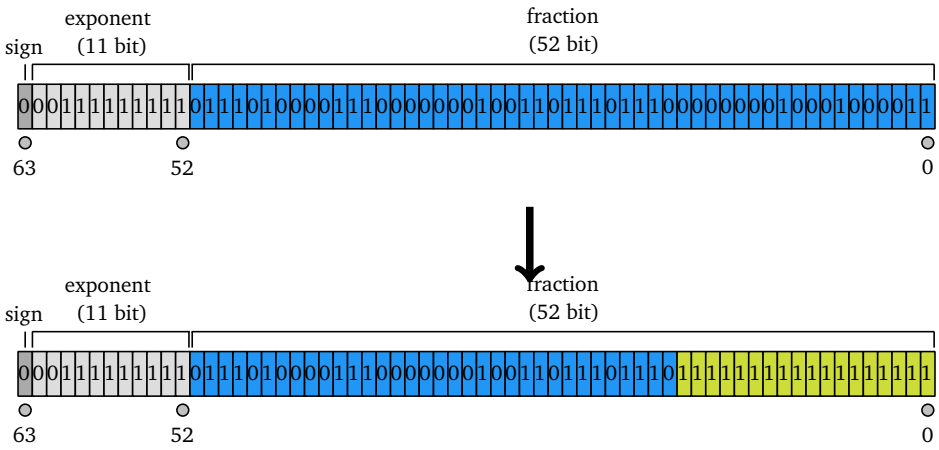


Figure 10.19.: Illustrating the effect of applying a mask on the random part of the matrix elements.

The evolution and the distribution of the `dgemm` durations is plotted in Figure 10.20. There is a very clear correlation between the mask size and the performance: the larger the mask, the lower the duration. Similarly to the previous experiment, some temporal patterns can also be distinguished.

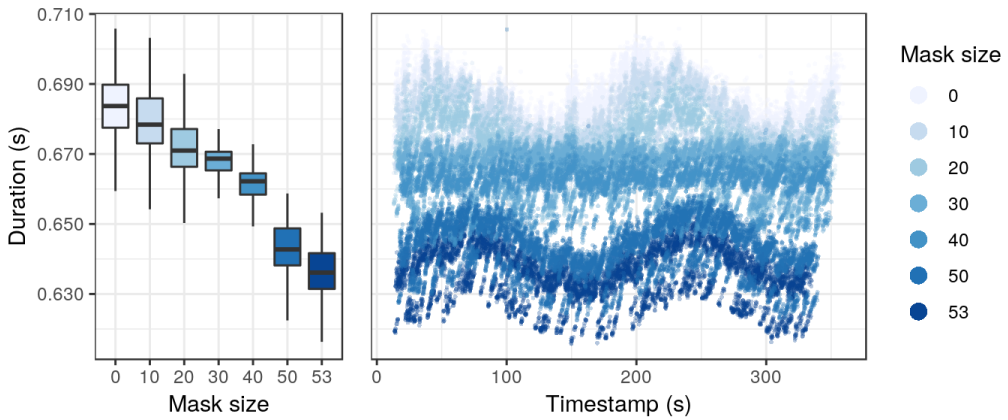


Figure 10.20.: Durations of individual `dgemm` calls are lower with larger bit masks.

This correlation with the mask size can also be seen with the frequencies in Figure 10.21: larger masks lead to higher frequencies.

[3]Image adapted from `https://en.wikipedia.org/wiki/Double-precision_floating-point_format`

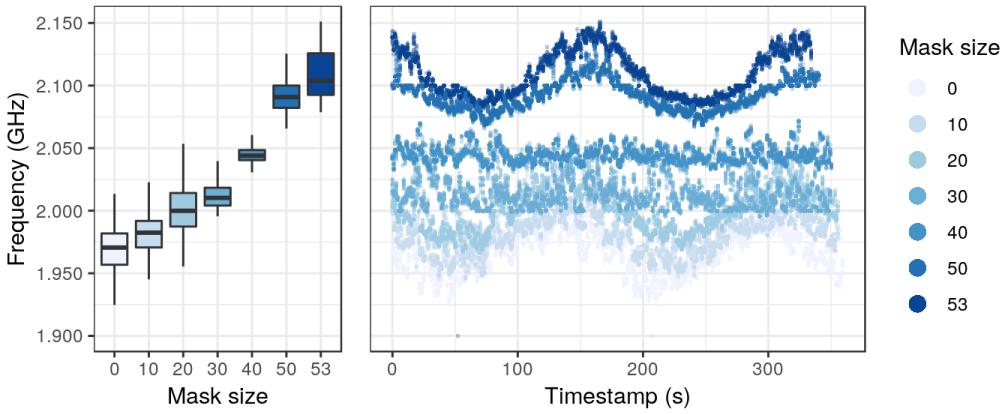


Figure 10.21.: Core frequencies are higher with larger bit masks.

This experiment has been repeated on two other Grid'5000 clusters, `ecotype` and `gros`. For both of them, the same observations could be made, a clear correlation between the mask size, the frequencies and the performance.

10.5.4 Conclusion

We have shown that the performance of the `dgemm` function is data-dependent. The best explanation we have for this counter-intuitive fact is an energy cost overhead caused by bit flips inside the processor. To respect its energy budget, the CPU has to throttle more aggressively its frequency when matrix elements are more diverse and thus more energy consuming. This theory has been corroborated by a controlled experiment where the elements of the matrices are initialized semi-randomly: they all share an identical bit suffix.

To strengthen this claim further, the next steps will be to perform a similar experiment with another compiler, another BLAS library and/or another computation kernel. We also need to understand why some processors are subject to this phenomenon and some others are not.

We warmly thank our colleagues who helped us find hypotheses for this performance anomaly. In particular, Guillaume Huard suggested that the performance anomaly may be caused by the bit flips in the processor.

10.5.5 Related work

M. Laß, C. Plessl and R. Schade (Paderborn Center for Parallel Computing) independently made similar observations (personal communications on 2020/09/24 and 2020/11/11). Their findings are summarized here:

- They observed that `dgemm` performance depended on the content of the matrix. They also made the hypothesis that it was caused by bit-flips. To test this hypothesis, they used the same approach, they filled the matrices with random values with a mask applied to the lower-order bits. They observed a correlation between the mask size and `dgemm` performance, which is an argument in favor of this hypothesis. Their experiment was done using another `dgemm` implementation than the one used in this work (Intel MKL) on an Intel Xeon Gold 6148 (Skylake-SP family) from a noctua node[4].

- They reproduced the same experiment on an FPGA (more precisely, a Bittware 520N PCIe accelerator card, equipped with an Intel Stratix 10 GX 2800 FPGA). This time, the way the matrix was filled had an effect on the power consumption and not the performance. This was expected since there is no DVFS on FPGA.

- To see whether the effect was caused by the CPU arithmetic units or the cache, they adapted a micro-benchmark[5] to perform multiplications with more or less random data, using only the registers of the CPU. They did not observe any difference of performance caused by the data, but they did observe a higher power consumption of $5\,\%$ with random data. This power consumption remained under the TDP even after the increase, so the CPU frequency did not get throttled.

Schöne et al. [Sch+19] observed a data-dependent power consumption with a Skylake-SP processor using AVX instructions. Note that in their experiment, the core frequency and the instruction rate were constant.

André et al. [And+20] observed in one of their experiments that limiting the uncore frequency (i.e. the frequency of the L3 cache and the memory controller) can increase the performance of HPL by about $1.5\,\%$. The reason is that HPL power consumption reaches TDP, so lowering the uncore frequency allows a higher power consumption of the cores and thus a higher frequency.

[4] `https://pc2.uni-paderborn.de/hpc-services/available-systems/noctua/`
[5] `https://github.com/pc2/Flops`

10.6 Beware of extrapolations

On several occasions when working on Part I, we found that our predictions were inaccurate because of a wrong extrapolation. The measures made for instantiating the model were made with parameters that were very different to the ones needed when using these models in simulation. This happened at least twice, with the `dgemm` function and with MPI communications.

Function `dgemm` The model used to be instantiated using the durations of `dgemm` calls made with random arguments M, N and K, as described in Section 10.2. The maximum value of these three parameters was $15{,}000$. However, when HPL is executed with very elongated geometries (e.g. a very small `P` or a very small `Q`), it performs `dgemm` calls with very elongated matrices. We observed in some experiments that M or N could take values as large as $150{,}000$ while the two other parameters remained rather small. In these conditions, the durations of individual `dgemm` calls are much more variable, while the average performance is slightly lower.

Functions `MPI_Recv` and `MPI_Send` In the legacy script for calibrating MPI communications for Simgrid, the message size was sampled randomly, with a maximum size of $1\,\mathrm{MB}$. However, similarly to the `dgemm` calibration issue we had, some communications in HPL are much larger, up to $1\,\mathrm{GB}$ in our experiments. When increasing the maximal size, we found out that the network performance drops very significantly at some point.

These two examples can seem obvious after the fact. Yet, they show the difficulty of anticipating when the usage of a model will reach its limits. In both cases, we had to instrument the HPL code to generate a trace and find that the parameters space used in the calibration was very different from the parameter space used during the execution. One way to limit the risk of missing such extrapolations would be to define explicitly in the model files the parameter ranges that were used for calibrating them. Then, during a simulation, we could raise a warning whenever a model is used too often outside its parameter space.

10.7 Beware of experimental conditions

We already have demonstrated numerous times in previous sections that the experimental conditions are of utmost importance in this work, any change can have a

significant effect on the measures, as harmless as it may seem. In this section, we give some details on an interaction between computations and MPI communications we observed while working on Part I.

We compare four different scenarios for sending a message of $256\,\mathrm{MiB}$ with MPI. For each scenario, we performed two experiments, one where this communication happens locally and another one where it is done remotely. The result is presented in Figure 10.22.

- Scenario `idle`. Two MPI ranks perform a simple ping-pong with the aforementioned message. Each MPI rank is bound to one core (either two cores of the same node for the loopback communication, or two cores of distinct nodes in the remote case). All the other cores are kept idle.

- Scenario `dgemm`. This one is identical to the scenario `idle`, except that all the cores not involved in the communication are performing `dgemm` calls.

- Scenario `MPI_Iprobe`. The communication pattern is more elaborated in this scenario. It uses two nodes, for a total of 64 cores. It repeats several times a sequence of 64 steps, where at step i the MPI rank i performs a ping-pong with the MPI rank $(i+1) \mod 64$. This is similar to a ring broadcast, except that the ranks perform a two-way exchange instead of one-way. All the ranks waiting for an incoming communication are performing a busy waiting, looping on the result of the function `MPI_Iprobe`.

- Scenario `MPI_Iprobe` & `dgemm`. This scenario is identical to the `MPI_Iprobe` scenario, except that every rank performs a call to function `dgemm` between any call to function `MPI_Iprobe`.

The two last scenarios may seem contrived. The goal was, again, to implement a micro-benchmark that is as close as possible to what happens in the real application we tried to model, HPL in this case.

The background noise we introduced had a large effect on the performance of the ping-pong. The durations of individual calls to `MPI_Recv` are presented in Figure 10.22. Making simple calls to `dgemm` in the background increases the remote duration by nearly a factor 3, without affecting the local durations. The ring communication pattern with `MPI_Iprobe` busy waiting increases both the local and remote durations, although the effect remains small for the local communication. Finally, adding `dgemm` calls to the busy waiting increases a lot the durations of all communications, by adding an overhead of about $0.1\,\mathrm{s}$.

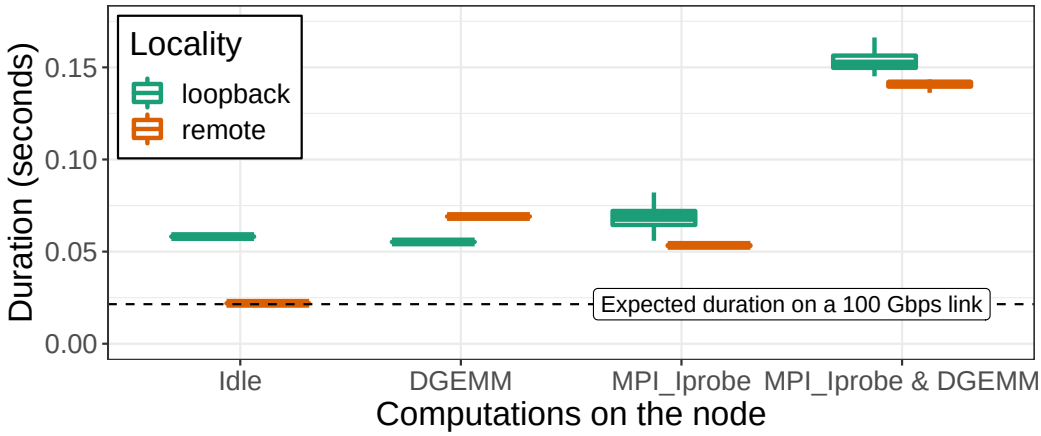


Figure 10.22.: Durations of a call to `MPI_Recv` with a message of 256 MiB and different background activities.

Although we do not have a definitive explanation of the root cause of this phenomenon, it is extremely important to reproduce it in the calibration benchmark if we wish to have a faithful model.

10.8 Conclusion

A common theme to several sections of this chapter was the importance of the randomization. It is of great help to avoid or at least limit any experimental bias. We have presented several counter-intuitive phenomena that were only revealed through randomization (or the lack thereof). However, in some cases, we needed our experiments to be biased, to have more realistic experimental conditions. When we execute a micro-benchmark that supposedly mimic a real application, the main objective is for this micro-benchmark to be the best possible surrogate for the real application, being unbiased is only a secondary goal.

During our work, we have been confronted multiple times to external factors that unknowingly affected our experiments (e.g. cooling issues, BIOS upgrades). With this kind of experimental artifacts, it can become more difficult to trust any experimental results, especially in the presence of unexpected phenomena like the ones presented here. This demonstrates the great importance of gaining back this trust by (1) having an experiment engine to automate the entire experimental process (thereby greatly reducing the human errors) and collect relevant information and (2) making regular tests on the platform to detect possible changes.

11 Performance non-regression tests

When working on the simulations described in Part I, we occasionally encountered inconsistent performance across different time periods. The reasons are multiple, ranging from the hardware to the software. This motivated the implementation of performance non-regression tests, to help us detect any noteworthy change on the platform. In this chapter, we start by giving an overview of the existing state of the art in Section 11.1. We then describe our contribution in Section 11.2. We start by detailing the statistics on which rely our tests. Then, we describe how we implemented these tests, from the experiment execution to the final test result. We show and explain the graphical representation of our test results, and we describe the various events we were able to detect on the Grid'5000 platform. Finally, in Section 11.3 we propose several ways to improve this work.

11.1 State of the art

Testing is a core activity of software development. It is usually taught as early as the first programming courses, students need to verify that their programs work as expected. More experienced software developers generally use a wide variety of techniques and methodologies to limit the presence of bugs.

Performance testing is less common, the main reason being that it is arguably much more difficult than testing for correctness:

- For a performance value to be meaningful, a lot of care needs to be taken when running the test for controlling the experimental environment.
- The result of a benchmark is a raw performance value (e.g. a duration, an amount of memory or an amount of energy). Ultimately, the test should return a boolean, stating whether the test was successful or failed. It is difficult to define properly such a *ground truth*. One could define an interval for the expected value, but this will most of the time be an arbitrary choice. A

complementary solution, to ensure that the performance values remain stable throughout the life cycle of the software, would be to use statistical tests.

11.1.1 Lack of tests

This section lists several examples of well-established software projects with limited or even nonexistent performance testing.

Simgrid [Cas+14], the simulation framework used in Part I, is a well-established software. In the last twenty years, several dozens of contributors have helped to improve or extend it. Simgrid has also supported the research for several hundreds of articles, demonstrating a wide user base (relatively to its niche area). The main developers of Simgrid have spent countless hours in optimizing the speed of its core components, like the linear maxmin solver. Despite these efforts, there are currently no performance test in place to prevent eventual performance regressions.

To the best of our knowledge, even HPC libraries like OpenBLAS [OpenBLAS] and OpenMPI [OpenMPI] do not use automated performance tests. OpenBLAS has several benchmarks implemented[1], but they rely on human intervention to run the tests and interpret the performance results. OpenMPI has a software, named MTT[2], to automatically deploy a middleware on an infrastructure and run correction tests as well as benchmarks. Yet again, the result of these benchmarks has to be interpreted by a human.

SimdJSON [LL19] is a state-of-the-art C++ library to parse JSON documents extremely efficiently. It can process documents at about $2.5\,\mathrm{GB/s}$, more than twice faster than the concurrent JSON parsers. It is widely used, with more than twelve thousand stars on GitHub. Yet, in April 2020, one of the main contributors submitted an issue[3] to report a previously unnoticed $50\,\%$ performance drop when the library was compiled with Visual Studio 2019 instead of G++ 7.5.0. This shows that even high performance libraries do not always have the methodology and tools to avoid performance regressions.

[1] `https://github.com/xianyi/OpenBLAS/tree/f917c26e/benchmark`
[2] `https://open-mpi.github.io/mtt/`
[3] `https://github.com/simdjson/simdjson/issues/812`

11.1.2 Existing work

Benchmarking software: testing code snippets

Several libraries already exist for benchmarking C++ software. Catch2 [Catch2] is a widely used unit testing framework for C++. It also provides basic functionalities for benchmarking code snippets. Several alternatives exist, developed specifically for C++ benchmarking [Celero; Hayai; Nonius], but seemingly no longer maintained and with a smaller user base.

Google Benchmark [GBench] is a C++ library to benchmark code snippets. It is well-established with several dozens of contributors and about 5000 stars on GitHub. It greatly facilitates the creation of parametric micro-benchmarks by adding only a few lines to an existing code. The result can be pretty-printed in the terminal or written in a file in a CSV or JSON format. It is even possible to automatically compute the asymptotic complexity. Finally, they provide a script[4] to compare the results of two executions of the same benchmark, it will print the difference with a raw value as well as a percentage. If there is a large enough number of repetitions of these benchmarks, the script can also perform a Mann–Whitney U test to test if the performance for these two series of runs is statistically identical.

All these libraries suffer from at least one of the following limitations:

- For a given code snippet and a given input, they will perform several iterations and report the average duration. The user has no control on this number of iterations which is not even deterministic, as the code snippet is executed in a loop for a fixed period of time.
- We do not have any information on the distribution of these durations, only the average is reported, the standard deviation is not displayed, and we do not have any confidence interval for the estimations of the mean. Note that this is understandable, these libraries typically start a timer, then perform the iterations, they never measure the duration of an individual iteration. It makes sense when measuring an extremely short code snippet that can take a few nanoseconds, but this is more problematic in our use case where we measure longer function calls that take at least several milliseconds.
- There is no support for randomization as the code snippets are executed in a deterministic order. The calls to a given snippet with different parameters are also executed in a deterministic order.

[4] `https://github.com/google/benchmark/blob/8f5e6ae0/tools/compare.py`

- The statistical tests implemented (if any) are quite basic. There is also no graphical visualization of the results, which would greatly help the user to (1) better understand the outcome of the test (not everyone knows what is a *P-value*) and (2) visually detect eventual differences not captured by the test (the Anscombe's quartet is the classical example).

- There is no support for comparing long-term performance evolution. With these libraries, it is possible to make performance measures for a given state of the codebase on a given machine. It is even possible to make point-to-point comparison with Google Benchmark, i.e. to compare two states of the codebase or two machines. However, we still miss the full picture, the variation of the performance on a historical timeframe.

Benchmarking software: long-term performance evolution of an application

Oliveira et al. show how it is possible to implement performance non-regression tests as part of a software development practice, akin to the usual unit tests and integration tests for verifying the correctness of the software [Oli+17]. Their goal is to avoid running the full benchmark whenever a new commit is made. To this end, they perform static analysis on the binary and have a list of several indicators to trigger a new execution of the benchmark (e.g. if the number of functions added or deleted by the commit is larger than some threshold). Their article also presents particularly well the state of the art, notably regression benchmarking, continuous integration systems, regression test selection and performance bug detection in the field.

Another interesting work is related to the StarPU project [Aug+11]. StarPU is a state of the art C/C++ task programming library that handles task dependencies and optimized usage of resources (heterogeneous scheduling and data transfers). It is also possible to accurately simulate a StarPU application, using the Simgrid simulation framework [Sta+15]. As part of their development methodology, the StarPU team performs nightly runs of a few selected StarPU applications and measures their performance, both in reality and in simulation. The goal is to detect any performance regression possibly introduced by a change in the software. Figure 11.1 presents the temporal evolution of the StarPU implementation of `spotrf`, which computes a Cholesky factorization of a given matrix. The green line is the observed performance in reality, the orange line is the estimation made in simulation. The large jump on 15/12/2017 is due to a hardware upgrade. The real runs exhibit a large variability, which are mainly experiment artifacts: some experiments were

performed on a badly configured node which does not report the correct number of cores, so StarPU cannot use appropriately the machine. Since the StarPU team is interested in detecting regressions in StarPU itself and not the platform, these simulations are extremely valuable to them, allowing them to rule out the false conclusions that would be made by looking only at the real runs.

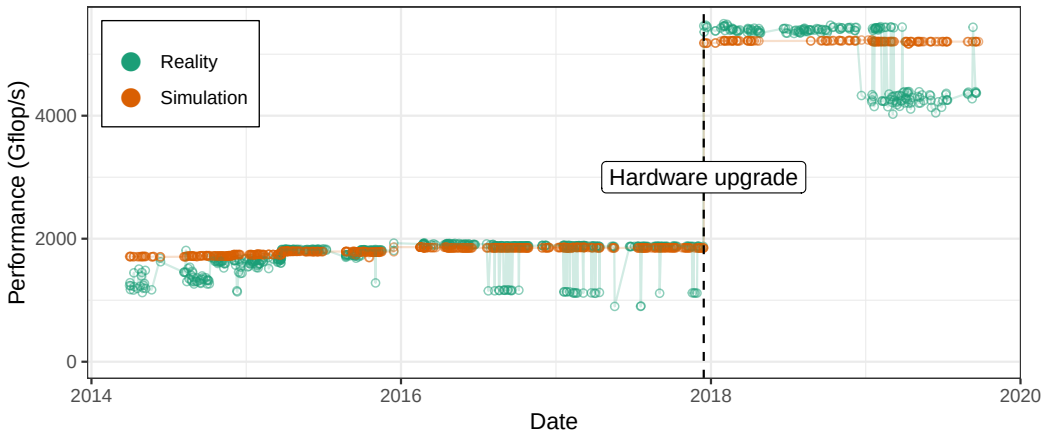


Figure 11.1.: Evolution of the performance of a StarPU implementation of `spotrf` function, both in reality and in simulation [TSL20].

The approach of the StarPU team is particularly interesting, because they executed their application on a controlled environment on a near-daily basis for several years. An important issue however is that they did not perform any statistics. They only relied on a visual inspection of the plot to decide whether the last modifications of their codebases had any significant effect on performance. While this can be satisfying to some extent, it raises two concerns:

- They could miss a subtle change in the performance or detect it much later, as it can be extremely difficult and error-prone. [5]
- This does not scale well, adding more applications or running the test on more machines would make this visual inspection very tedious.

Benchmarking systems

Other tools exist for measuring the performance of a platform under some workload. Here, we are not interested in the execution of a given software, but rather the whole platform, i.e. the combination of the hardware, the operating system and eventual programs running concurrently on the system. High Performance Linpack [Pet+]

[5] Emery Berger uses the term “eyeball statistics”: `https://youtu.be/r-TLSBdHe1A`

is such a tool. It performs a dense linear algebra operation using one or several computers and reports the observed flop rate.

Another system benchmarking tool is stress-ng [Stress-ng]. It runs a selected workload for a given duration and reports various aggregated performance. Over 240 different stress tests are implemented and can be combined (e.g. it is possible to launch three processes that perform many I/O operations and five processes that are CPU-intensive with vectorized floating-point operations). The reported metrics include the duration, but also the number of instructions, the number of cache reads and writes, the number of page faults, etc. Several dozens of bugs have been found in the Linux kernel thanks to the use of this benchmark.

These libraries suffer from the same limitations previously described: there is no randomization when several configurations are combined, and no statistical tests are available.

The Grid'5000 platform already has tests in place. First, a script called `g5k-checks` verifies each time a node boots that its characteristics match the reference information (e.g. the amount of memory, the BIOS version). Additionally, more extensive tests are executed on a regular basis on all the nodes [Nus17]. These are mostly *correctness* tests. They perform several operations (e.g. node deployment, network reconfiguration) that can either succeed or fail. Two *performance* tests are implemented, one for the network bandwidth, the other for the disk bandwidth. However, they use threshold values to define what is an acceptable performance, hence they will detect a severe regression but will miss more subtle changes.

11.2 Performance non-regression statistical testing

11.2.1 Statistical test basis

In this section, we describe how to compute a prediction region for multivariate normal variables. This will then be used to implement a statistical test. This section is mostly based on the work by Chew [Che66, Section 4.4].[6]

[6] This publication, dating from 1966, is one of the oldest papers used in this thesis. It has been written by an employee of RCA Service Co., a contractor of the US Air Force. The author of the paper describes the typical use case: computing the coordinates of a prediction ellipse for the splash point of a future missile shot. I believe this is a poor usage of statistics.

Suppose that we have already observed n vectors of dimension p: $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n} \in \mathbb{R}^p$. We define the random variable $\boldsymbol{m}^{(r)}$ to be the sample mean of the next (unknown) r observations: $\boldsymbol{m}^{(r)} = \frac{\boldsymbol{x_{n+1}} + \cdots + \boldsymbol{x_{n+r}}}{r}$. We assume here that all the $\boldsymbol{x_i}$ are independent and identically distributed according to a multivariate normal distribution of unknown mean and covariance matrix.

Then, the prediction region of $\boldsymbol{m}^{(r)}$ with probability γ is:

$$\frac{nr}{n+r}(\boldsymbol{m}^{(r)} - \overline{\boldsymbol{x}})^T \boldsymbol{S}^{-1} (\boldsymbol{m}^{(r)} - \overline{\boldsymbol{x}}) = \frac{(n-1)p}{n-p} Q_F(1-\gamma, p, n-p) \quad (11.1)$$

Where $\overline{\boldsymbol{x}}$ and $\boldsymbol{S}$ are respectively the sample mean and sample covariance matrix of the n observed $\boldsymbol{x_i}$, and $Q_F(\alpha, v_1, v_2)$ denotes the upper α quantile of $F(v_1, v_2)$, the F-distribution with v_1 and v_2 degrees of freedom.

We therefore propose the following statistical test for $\boldsymbol{m}^{(r)}$ with confidence γ. First, compute the value:

$$t = \frac{nr(n-p)}{(n+r)(n-1)p} (\boldsymbol{m}^{(r)} - \overline{\boldsymbol{x}})^T \boldsymbol{S}^{-1} (\boldsymbol{m}^{(r)} - \overline{\boldsymbol{x}}) \quad (11.2)$$

Then, raise an error if $t \geq Q_F(\gamma, p, n-p)$. Note that there is no closed form for the value Q_F, but it can be obtained (numerically) in R with the function `qf`[7] and in Python with the function `scipy.stats.f.ppf`[8].

The proposed test is illustrated in Figure 11.2. Numerous points, in black, have been generated according to a known bivariate normal distribution. Their abscissa (resp. ordinate) are distributed according to a normal distribution of mean μ_x and standard deviation σ_x (resp. μ_y and σ_y). The abscissa and ordinates of the points are not independent, they have a correlation coefficient of -0.7. The $99.5\,\%$ prediction regions are shown in blue. Two additional points are shown in the scatter plot, representing new observations. The orange point has coordinates $(\mu_x + 2\sigma_x, \mu_y + 2\sigma_y)$ and the green point has coordinates $(\mu_x + 2\sigma_x, \mu_y - 2\sigma_y)$.

If each dimension was considered independently, we would conclude that the probabilities to observe the orange point or the green point are equal. Indeed, these two points have equivalent positions in the one-dimension density plots and they both fall within the $99.5\,\%$ interval. Now, when we look at both dimensions

[7] `https://stat.ethz.ch/R-manual/R-devel/library/stats/html/Fdist.html`
[8] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.f.html`

simultaneously, the green point becomes much more likely to be observed as it is within the blue ellipse while the orange point is outside.

In Figure 11.3, two sets of five additional observations are presented as colored dots. Individually, they were all likely to be observed, they are within the dashed ellipse representing the $99.5\,\%$ prediction region for a single point (i.e. $r = 1$). However, if we consider them together, the prediction region for their averages shrinks drastically. Now it becomes clear that the set of green points was more likely to be observed than the set of orange points: the green average (marked by a cross) is within the filled ellipse representing the $99.5\,\%$ region for a five-point average (i.e. $r = 5$) whereas the orange average is outside.

11.2.2 Implementation workflow

This section describes the different steps and tools involved to perform the performance test described in Section 11.2.1, from the execution of the experiment on the target machines to the generation of the final plots. It discusses several technical choices that have been made and gives some insights to their advantages and limitations.

The workflow, illustrated in Figure 11.4, consists of three main steps. The first one is implemented with `peanut` while the two others are implemented with `cashew` [CL21b], a Python library implemented specifically for this task.

Experiment execution The `dgemm` calibration program is executed on the desired nodes. We obtain a `zip` archive containing the experiment data and metadata.

Archive extraction and aggregation The data is extracted from the archive and appended to two HDF5 files (one for the `dgemm` duration, one for the monitoring data). Then, some statistics are computed with this new data on a per-CPU and per-experiment basis (e.g. average `dgemm` performance and average CPU temperature), these aggregated values are added to two CSV files.

Statistical test computation Several statistical tests are done on the sequence of aggregated data. The result is presented as several Jupyter notebooks containing plots.

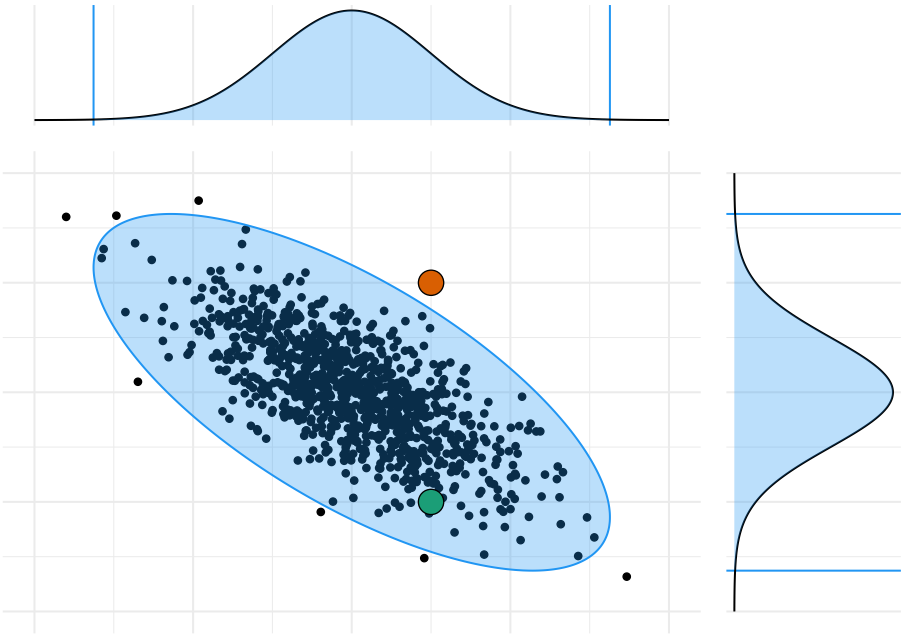


Figure 11.2.: Illustrating the proposed test with two sets of one observation ($r = 1$) and a bivariate normal distribution ($p = 2$). The blue zones represent the $99.5\,\%$ prediction regions.

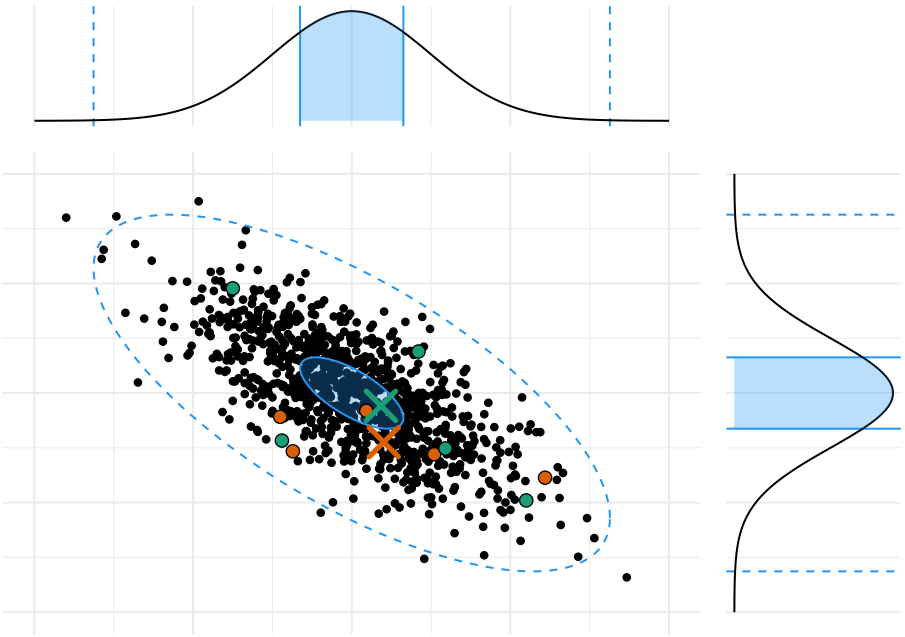


Figure 11.3.: Illustrating the proposed test with two sets of five observations ($r = 5$) and a bivariate normal distribution ($p = 2$). The blue dashed lines represent the $99.5\,\%$ prediction regions for single observations, the blue zones represent the $99.5\,\%$ prediction regions for the averages of five new observations, represented by crosses.

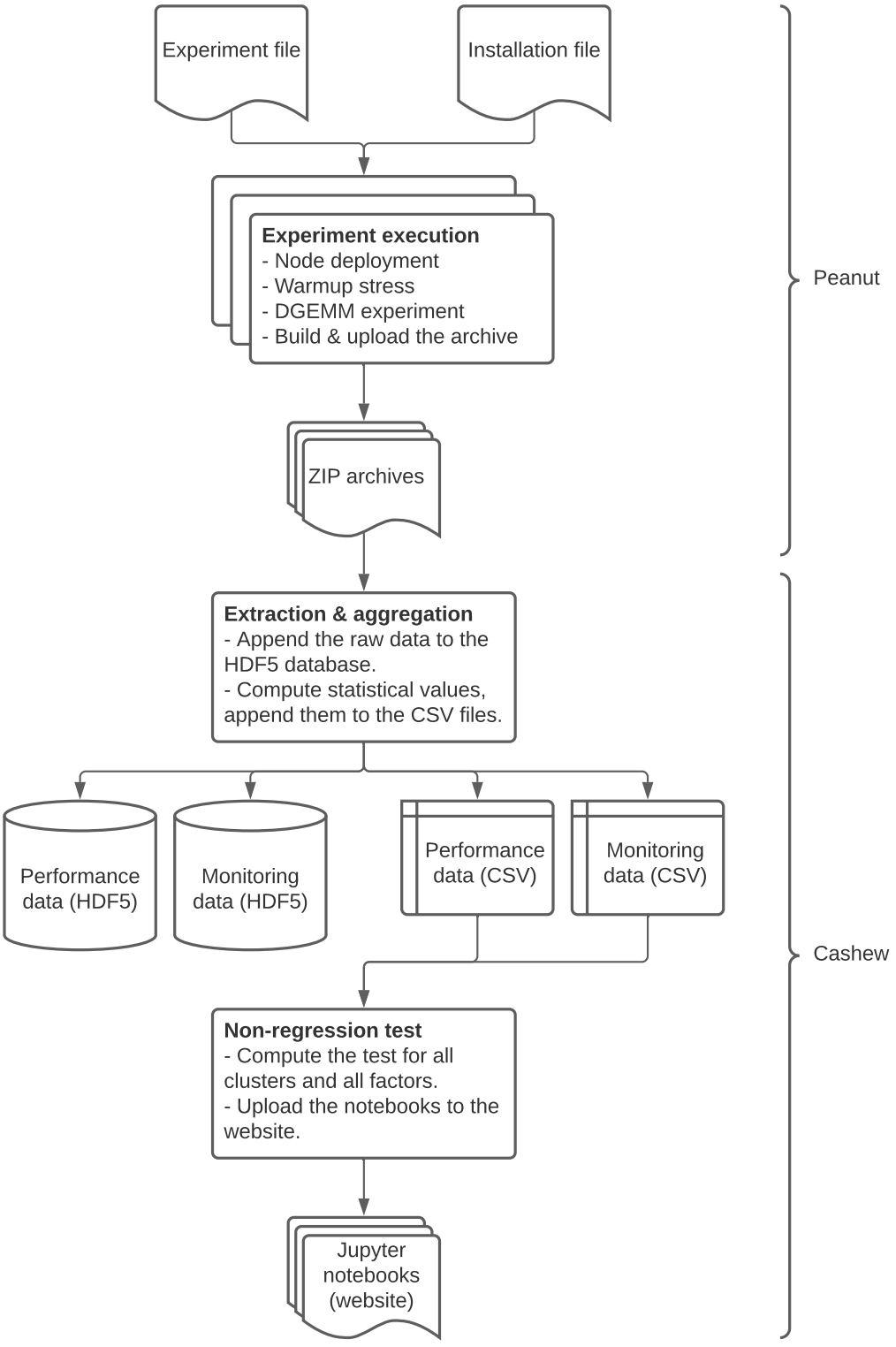


Figure 11.4.: Full workflow implemented by our tests.

Experiment execution

The experiment script is written in Python and uses the `peanut` experiment engine [Cor21c] (see Section 9.2). This is also the script used to perform the `dgemm` calibrations for HPL simulation (see Part I).

On a regular basis, typically three to four times a week, we use this script to submit new experiment jobs to Grid'5000's scheduler. This regular submission could have been easily automatized, e.g. with `cron` or `systemd`, but we made the choice to keep this step manual. By inspecting the Grid'5000 Gantt chart, we can check whether the clusters are currently heavily used or not and decide if we should postpone our experiment to avoid disturbing too much the other users. A particularly good timeframe for our experiments is usually between 7 a.m. and 9 a.m., when the (longer) nightly experiments have terminated and the (shorter and interactive) daily experiments have not started yet. We generally submit one job per cluster, requesting all the nodes, but we sometimes have to split the experiments and make several smaller jobs when only a subset of the nodes is free immediately. The alternative would be to submit the full-scale job anyway, but it would get scheduled later in the day, which might disturb other users that need to work interactively on some nodes. Again, this kind of decisions is the reason why this step is not trivial to automatize.

When scheduled, the experiment script performs the following steps:

1. Deploy a fresh Debian image on all the nodes of the job and install the required dependencies.

2. Apply the desired setup to the nodes (e.g. enable turboboost, disable hyperthreading, use the lowest C-state and P-state).

3. Start the node monitoring script in the background.

4. Perform a stress of $10\,\mathrm{min}$ on the nodes as a warm-up.

5. Run the `dgemm` calls with the desired experiment file.

6. Collect the resulting data (two CSV files, one for `dgemm` durations and one for monitoring values) and meta-data (system files, software versions, etc.) and build an archive.

7. Push the archive to the git repository.

The choice to use a git repository for storing the experiment data can seem peculiar. Git is a very good version control system for small text files but is not the best choice to store gigabytes of data (although we mitigated this by using git LFS). It would have been undoubtedly more efficient to set up a proper database server for this task, but probably longer to implement.

The `dgemm` CSV file contains one row for each individual `dgemm` call. The columns are the hostname of the node, the core ID, the exact time at which the call was made, the duration of the call and the `dgemm` parameters (including the matrix sizes M, N and K).

The monitoring CSV file contains one row per sample, which typically happens every $5\,\mathrm{s}$. The columns are the hostname of the node, the exact time at which this sample was taken, then one column for each available metric (e.g. temperature of the CPU n°1, frequency of the core n°6, power consumption of the CPU n°0).

Archive extraction and aggregation

Once the experiments of the day have terminated, we submit an *extraction* job on Grid'5000. In the first versions of this work, this step used to be implemented with GitLab's continuous-integration, but this quickly revealed to be too demanding for the runner of our GitLab instance.

The extraction job is a small script that uses `cashew`. It performs the following steps:

1. Clone the git repository.
2. Iterate on all the new archives to extract the performance and monitoring data. Append it to the two HDF5 data files.
3. Process the new additions in the data files to produce aggregated values, append these aggregated values to the CSV files.
4. Push the changes to the git repository.

In the extraction step, we use the archive metadata to add some information to the tables stored in the two CSV files. For instance, we add the ID of the job and its scheduled date, the cluster of the node and a hash of the experiment file. We also reshape the monitoring table to switch from its wide format to a long format (i.e. each row now has a set of identifier variables and a single measure variable). These two tables are then appended to the two `hdf5` files.

We compared several alternative file formats to store this data: a simple CSV file, a CSV file compressed with `zip`, SQLite [SQLite], Apache Arrow [Arrow] and HDF5 [HDF5] with various options. The Apache Arrow library had the fastest read and write operations as well as the smallest file. HDF5 with the `table` format and the `zlib` compression algorithm was the second best, the other alternatives were largely inferior in terms of I/O speed and disk footprint. Two important features of HDF5 made it preferable to Apache Arrow for our use case: (1) The possibility to append data to a file. With Apache Arrow, for each new experiment we would have to read the whole file in memory to add the new data. (2) The ability to load only a subset of the data based on a logical predicate. For instance, it is possible to load the measures made on a given node between two given dates without having to read the whole file.

In the aggregation step, we summarize the data from a given experiment into a single row for each CPU of each node. We keep the `dgemm` data and the monitoring data separated. Both these files have identifier columns: cluster name, node ID, CPU ID, job ID, experiment start time and experiment file hash. Their measure columns are listed below:

- In the `dgemm` file, each row has the observed `dgemm` performance, computed as the total number of flops (equal to $2\sum_i M_i N_i K_i$) divided by the total duration. Each row also has the coefficients of the linear regression using the full polynomial model, as discussed in Part I.

- In the monitoring file, each row has the observed mean temperature, frequency, CPU power consumption and DRAM power consumption. To represent the steady state of the experiment, these averages are computed on a subset of the available values, a window starting $2\,\mathrm{min}$ after the start of the `dgemm` calibration program and ending $2\,\mathrm{min}$ later.

Statistical test computation

When the extraction and aggregation of the new data is terminated, we compute the statistical test. This is usually done in the same Grid'5000 job as the previous step, but it can be done independently. The test does not need the full repository, instead it automatically downloads the required files.

The implementation directly follows the formula described in Section 11.2.1. It relies heavily on NumPy and Pandas for a better efficiency.

Several factors are available. They are all tested independently, as a one-dimensional variable (i.e. $p = 1$). For each of them, two tests are realized, one with a window of one job (i.e. $r = 1$) and one with a window of five jobs (i.e. $r = 5$). The different factors are:

Performance Average `dgemm` performance, including all calls. We recall that it is computed as the total number of flops (equal to $2 \sum_i M_i N_i K_i$) divided by the total duration. This is not equal to the arithmetic mean of the individual performance values.

Performance$_{2048}$ Average `dgemm` performance, restricted to the calls with matrices of size 2048×2048.

Frequency Average CPU frequency during the `dgemm` calls.

Power$_{CPU}$ Average CPU power consumption during the `dgemm` calls.

Power$_{DRAM}$ Average DRAM power consumption during the `dgemm` calls.

Temperature Average CPU temperature during the `dgemm` calls.

Model All the parameters of the linear regression for the `dgemm` durations, i.e. the intercept and the coefficients for the variables MNK, MN, MK, NK, M, N and K. The regression parameters of the linear regression for `dgemm` variability are also available.

A multi-dimensional test is also performed on the eight parameters of the linear regression for `dgemm` durations (i.e. $p = 8$). An overview of the graphical presentation of these test results is presented in section 11.2.4.

As for the previous step, the test is implemented with `cashew` and uses a Jupyter notebook. This notebook is instantiated and executed several times, once per cluster and per factor. All these copies are then converted to HTML and deployed to a website.

11.2.3 On the normality assumption

The statistical test described in Section 11.2.1 assumes that the data follows a normal distribution. In this section, we argue that all the variables described previously satisfy this hypothesis.

- The factors Frequency, Temperature, $\text{Power}_{\text{CPU}}$ and $\text{Power}_{\text{DRAM}}$ are all an arithmetic mean of several dozens of values. It comes directly by the central limit theorem that these averages follow a normal distribution.

- The Model factors (e.g. MNK) are coefficients of an ordinary least-square linear regression (OLS). The input data are the averaged `dgemm` durations, for each distinct tuple M, N, K, we compute the arithmetical mean of the durations observed on all the cores of a given CPU. Thus, by the central limit theorem, the input data has a normally distributed noise. Now, if we note $\boldsymbol{X}$ the $n \times 8$ matrix of regressors (each of the n rows is an observation, the 8 columns are the products MNK, MN, MK, NK, M, N and K that were used for this observation) and we note $\boldsymbol{y}$ the vector of observed durations, the estimator of regression coefficients $\hat{\boldsymbol{\beta}}$ can be obtained by computing $\hat{\boldsymbol{\beta}} = \left(\boldsymbol{X}^\intercal \boldsymbol{X}\right)^{-1} \boldsymbol{X}^\intercal \boldsymbol{y}$. By doing this regression, we assume that $\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}$ where $\boldsymbol{\beta}$ is the true (unknown) vector of coefficients and $\boldsymbol{\varepsilon}$ is the normally distributed noise. It follows that the estimator $\hat{\boldsymbol{\beta}}$ is also normally distributed.

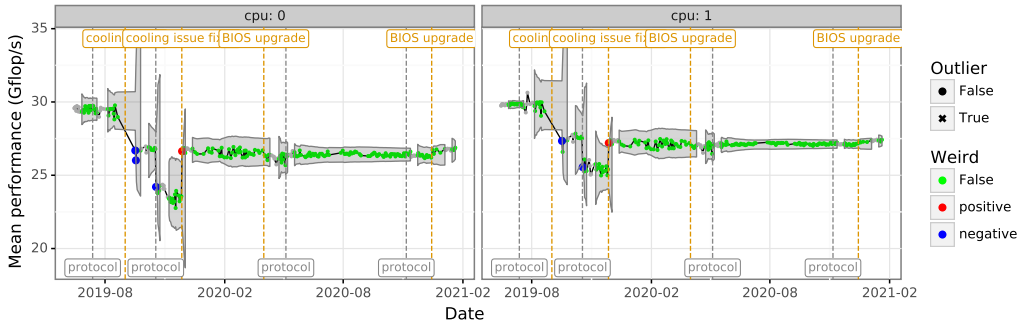
- The factors Performance and $\text{Performance}_{2048}$ are defined as a ratio of two values, the total number of operations divided by the total duration. The number of operations is constant, only the duration is a random variable. Since it is itself the sum of individual durations, it follows a normal distribution, by the central limit theorem. If we note the ratio as $R = \frac{F}{T+\varepsilon}$ where F and T are constant and ε is a random normal noise, we have $R = \left(\frac{F}{T}\right)\left(\frac{1}{1+\varepsilon/T}\right) \approx \frac{F}{T}\left(1 - \frac{\varepsilon}{T}\right)$. The last approximation is obtained by the Taylor expansion, it holds because $T \gg \varepsilon$, i.e. the variability of the total duration is small compared to its average. Since F and T are constant and ε is normally distributed, it follows that R is also approximately normally distributed.

11.2.4 Presentation of the test results

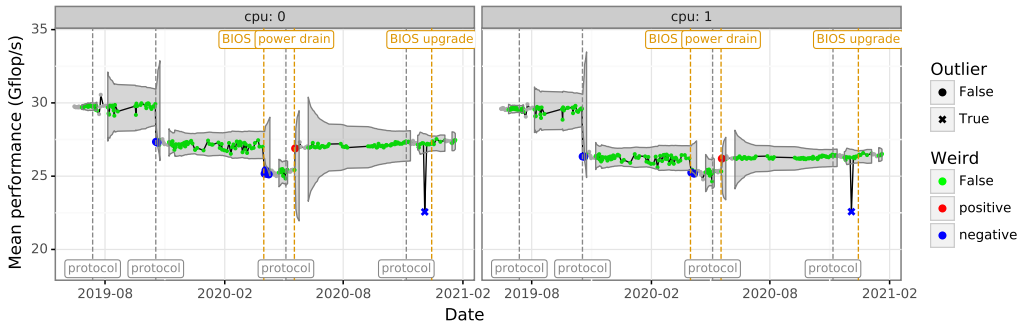
Evolution plot

The historical evolution of the mean performance of two nodes is presented in Figure 11.5. Each point represents the observed performance on a given experiment. The vertical dashed lines denote changes in the platform that had a significant effect on at least one of the observed factors. The gray lines (with a label on bottom) are protocol changes, we modified the experiment in a way that affected the results. The orange lines (with a label on top) are events that happened on the platform regardless of our will. The gray ribbon is the *fluctuation interval*, we expect all

the observations to fall within this interval with a given confidence ($99.99\,\%$ in this figure). The *reference set* of observations used to compute this interval consists of past observations that were made after the last change. In other words, (1) we do not consider the observations that were made in the future and (2) each time we recognize a change and add a vertical line, the fluctuation interval gets reset.



(a) Performance evolution of node `dahu-14`.



(b) Performance evolution of node `dahu-26`.

Figure 11.5.: Evolution of the mean performance of the two processors of two nodes from cluster `dahu` (the fluctuation interval has a confidence of $99.99\,\%$).

The fluctuation interval for a new value $\boldsymbol{m}^{(r)}$ (which is the sample mean of r new observations $x_{n+1}, \ldots, x_{n+r}$, note that $r = 1$ in Figure 11.5) follows a rewriting of Equation 11.2 with a single dimension ($p = 1$). Noting $\overline{\boldsymbol{x}}$ (resp. $\boldsymbol{s}^2$) the sample mean (resp. sample variance) of the reference set, the fluctuation interval is defined as:

$$I = \overline{\boldsymbol{x}} \pm s \left(\sqrt{\frac{n+r}{nr} Q_F(\gamma, 1, n-1)} \right) \tag{11.3}$$

Whenever an observation falls outside the fluctuation interval, it is detected as an anomaly. It is classified as either a positive anomaly and colored in red if it is larger than the sample mean, or it is classified as a negative anomaly and colored in blue if it is lower than the sample mean.

It occasionally happens that a single observation is way outside the fluctuation interval (e.g. on Figure 11.5(b), one experiment had a performance of approximately $23\,\mathrm{GFlop/s}$ near the end of 2020). When this happens, we manually label this point as an outlier and remove it from the reference set: it will not be taken into account for computing the reference interval afterwards.

Overview plot

We are running regular tests for several factors on hundreds of nodes, therefore it would be very long and tedious to review each individual plot. For this reason, we implemented overview plots, as presented in Figure 11.6. In this presentation, each processor of each node occupies one row. Individual experiments are still presented as points and the vertical gray or orange lines represent the same platform changes. We added intermediate levels in the colors to display how unlikely it was to make a given observation. In the one-dimension case (i.e. $p = 1$), this represents the distance between the observation and the sample mean of the reference set.

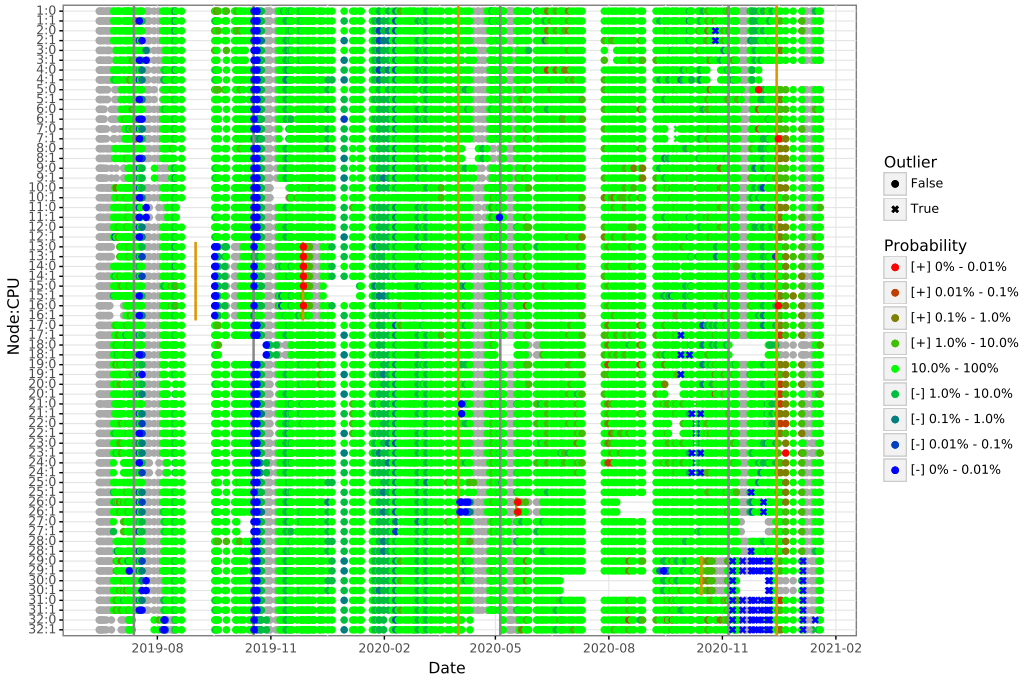


Figure 11.6.: Overview of the test result for the mean performance on cluster `dahu`.

More formally, this likelihood is the probability to make an observation at least as extreme. To compute it, we use the value t defined in Equation 11.2 and define the likelihood $\mathcal{L}$ as follows:

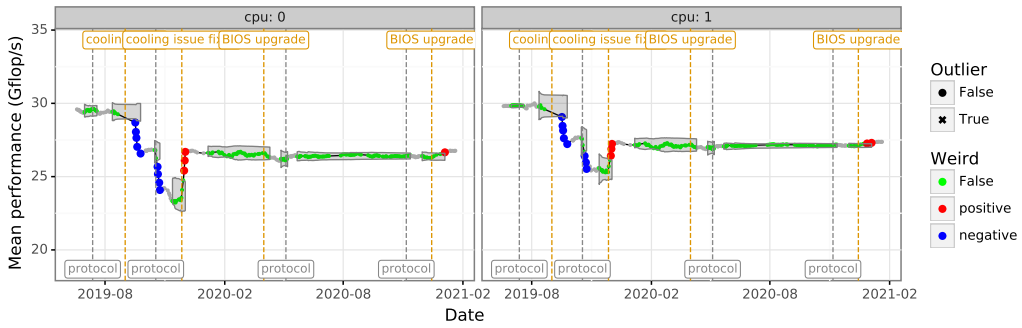
$$\mathcal{L} = 1 - F_{F(p,n-p)}(t) \tag{11.4}$$

Here, $F_{F(v_1,v_2)}$ denotes the cumulative distribution function of the F-distribution with v_1 and v_2 degrees of freedom. An implementation is available in R with the function `pf`[7] and in Python with the function `scipy.stats.f.cdf`[8].

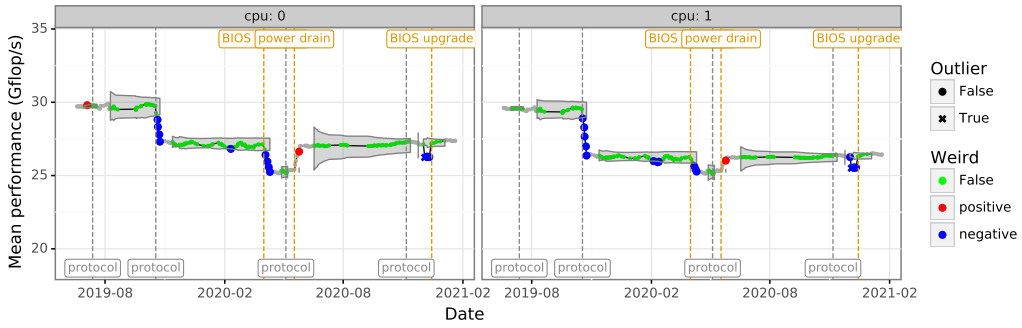
Windowed test

The test presented in Figure 11.6 and Figure 11.5 was done with a single factor ($p = 1$), for comparing a single new observation to the reference set ($r = 1$). A similar test is implemented to compare five new observations to the reference set, still with a single factor ($p = 1$ and $r = 5$).

Figure 11.7 shows the performance evolution of two nodes of the cluster `dahu` with the five experiment window. Taking the average of several runs reduces the noise. This allows reducing considerably the width of the fluctuation interval, thereby permitting to detect much more subtle changes. The downside is that it introduces a lag, meaning that the change might not be detected immediately.



(a) Performance evolution of node `dahu-14`.



(b) Performance evolution of node `dahu-26`.

Figure 11.7.: Evolution of the mean performance of the two processors of two nodes from cluster `dahu` using a window of five experiments (the fluctuation interval has a confidence of $99.99\,\%$).

The overview plot with the five experiment window is displayed in Figure 11.8. The vertical orange line on 2020/12/15 marks a platform change we noticed. On this plot, it is very clear that the change has affected significantly a large fraction of the nodes by increasing their performance (positive anomaly). This was slightly visible on the non-windowed plot from Figure 11.6, but much more tenuous. This demonstrates well the complementarity of both tests: the non-windowed test is better for detecting quickly any large change, whereas the windowed test is best at detecting more subtle changes but with some lag.

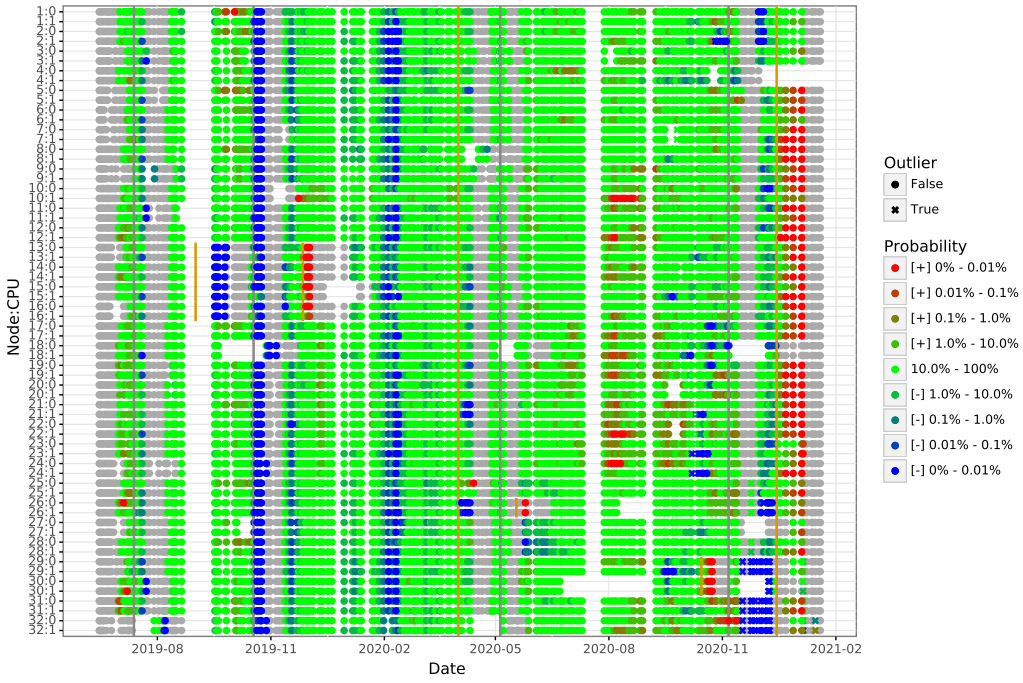


Figure 11.8.: Overview of the test result for the mean performance on cluster `dahu` using a window of five experiments.

Multi-factor test

We have also implemented the multidimensional test. Figure 11.9 presents the overview plot for this test made on the eight regression parameters (i.e. $p = 8$, the parameters are MNK, MN, MK, NK, M, N, K and the intercept). With several dimensions, there is no notion of *negative* or *positive* anomaly, hence all the detected anomalies are colored in red. Likewise, we cannot represent graphically the temporal evolution of these eight factors like we did in the single-dimension case.

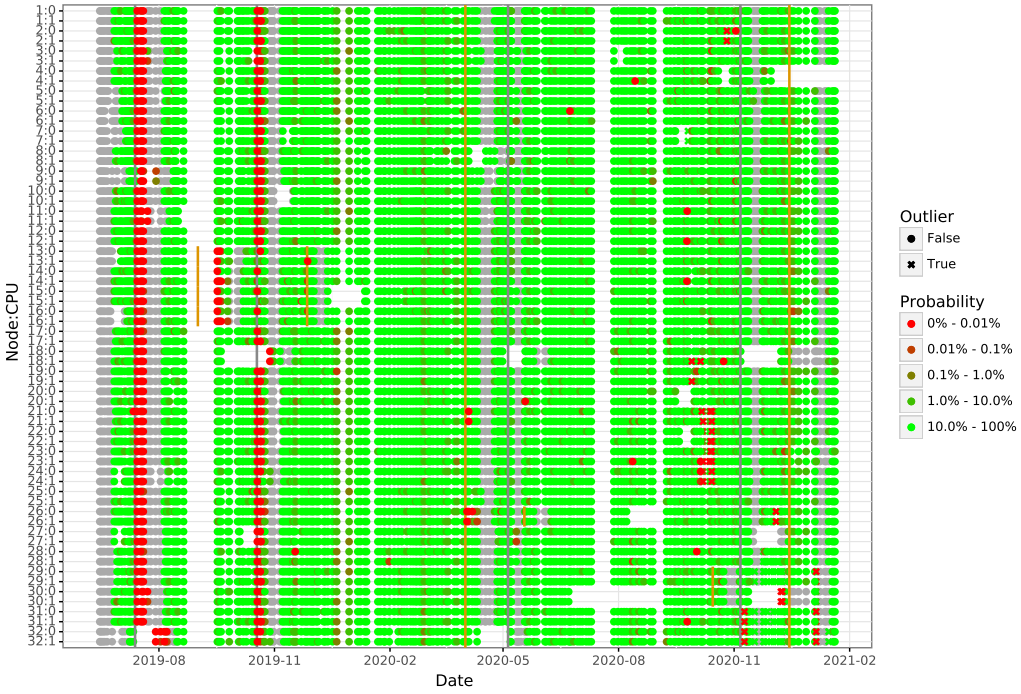


Figure 11.9.: Overview of the test result for the eight regression parameters on cluster `dahu`.

This figure should be compared to Figure 11.6, since both of them aim at detecting changes in `dgemm` durations. We can notice two interesting differences:

- The first change, which happened on 2019/07/13, was detected as very significant on all the nodes with the multi-factor test. On the other hand, the single-factor test only marked a subset of the nodes.

- Two events have gone completely unnoticed by the multi-factor test whereas they were reported by the single-factor test: the positive anomaly on `dahu-26` from 2020/05/18, and the series of negative anomalies (that we later decided to be outliers) between 2020/11 and 2020/12 on nodes `dahu-29`, `dahu-30`, `dahu-31` and `dahu-32`. The reason they went unnoticed is that a change was registered shortly before, so the reference set had a very low number of points. The radius of our fluctuation region is proportional to $Q_F(\gamma, p, n-p)$, it is obviously not defined when $n \leq p$. When n is larger than p, the value $Q_F(\gamma, p, n-p)$ starts extremely high, then quickly decreases. Hence, although it is defined, the test is not useful yet, we need to have a few more observations as reference. The same problem obviously exists when $p = 1$, but we have to wait longer with a larger number of parameters.

One way to limit this issue is to limit the number of factors used in the test. It appears that only three of the regression parameters are really significant: MNK, MK and NK. By doing so, the multi-factor test is able to detect the anomaly from 2020/05/18 on `dahu-26`, but it still misses the outliers from 2020/11-2020-12.

Website presentation

A screenshot of the landing page[9] of the Grid'5000 performance tests we implemented is presented in Figure 11.10. Each cluster is represented by a row, each factor by a column. Clicking on a button will open the test notebook for the desired cluster and factor. The last column is a drop-down menu showing all the coefficients for the linear regression, as well as a multi-dimensional test made on the first eight coefficients together.

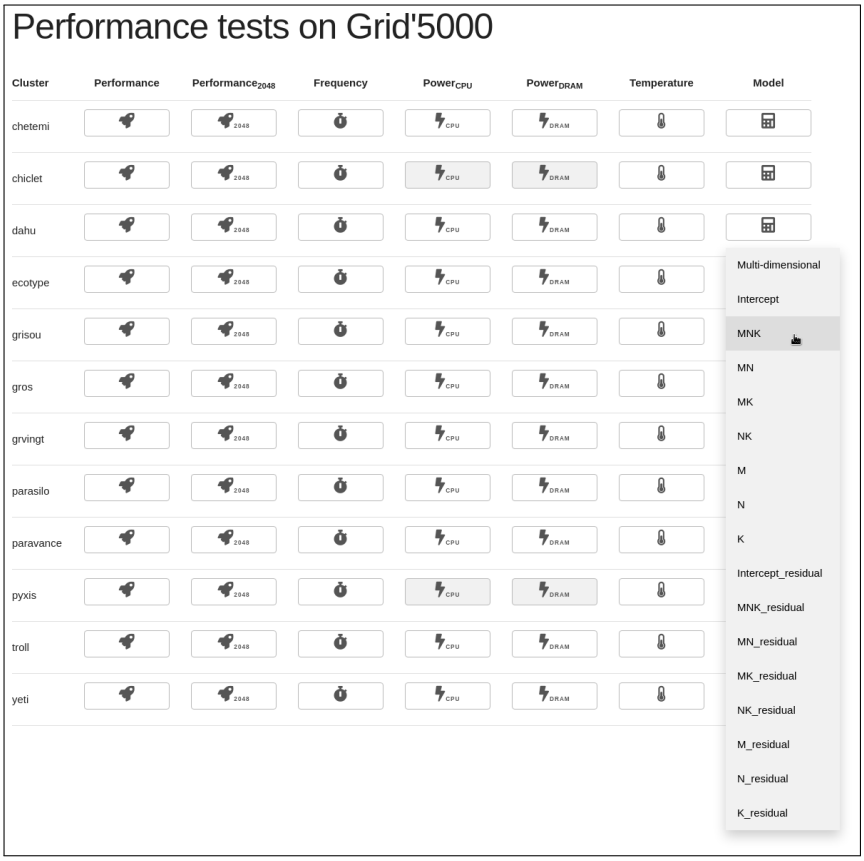


Figure 11.10.: Landing page of the test website.

[9]The website presented here is publicly available at `https://cornebize.net/g5k_test/` and permanently archived on Zenodo [CL21c].

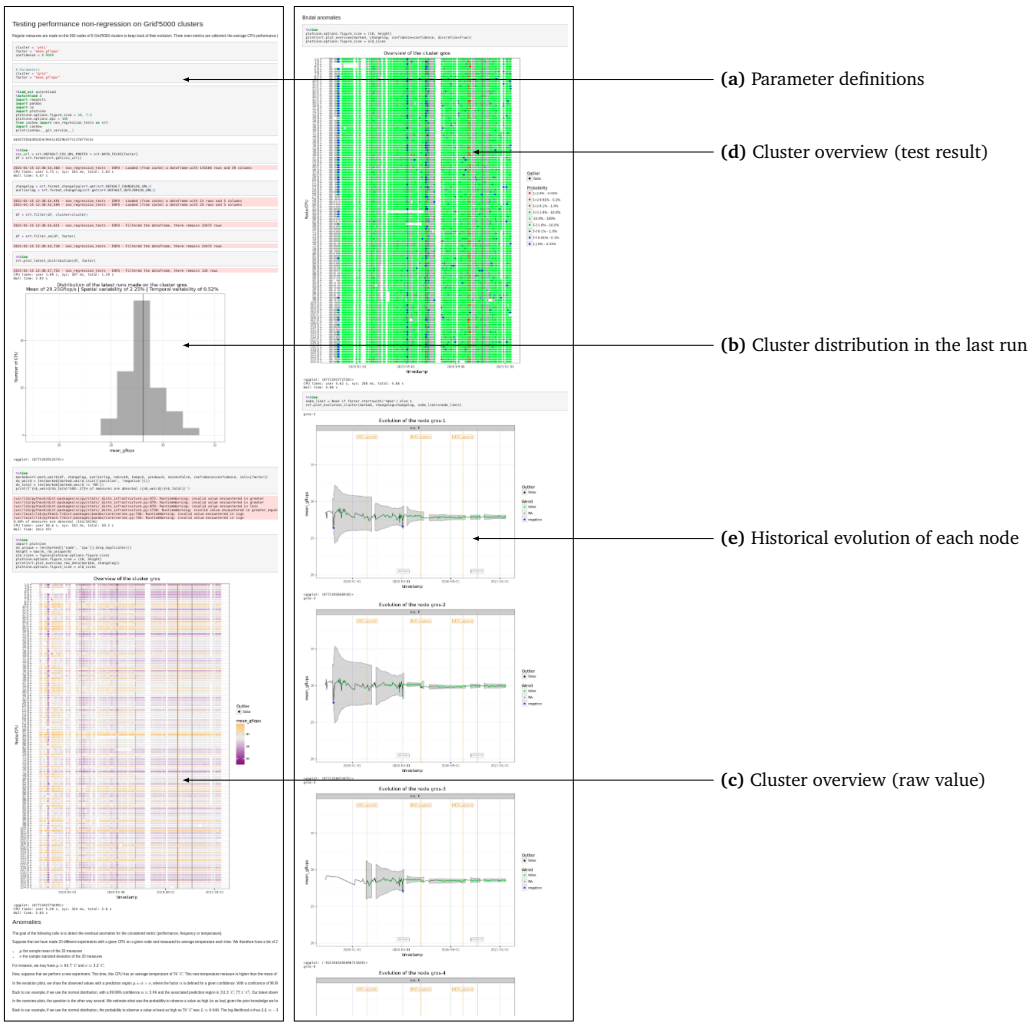


Figure 11.11.: Overview of a notebook (cluster `gros`, average performance).

An extract of a single-factor notebook is presented in Figure 11.11. The different parts of the notebooks are:

a This part is intended to import the required libraries and set up a few options, but most importantly to define three variables: the desired cluster, factor(s) and confidence.

b This is a histogram of the values obtained in the last run among the different processors of the cluster for the desired factor. The average value as well as the spatial variability coefficient are also displayed.

c This is an overview plot of the cluster, similar to the overview plots previously presented. The main difference is that the colors do not encode a test result, but the raw value. Like the histogram, it helps to visualize the heterogeneity

of the cluster. It also demonstrates once again the utility of statistical tests: only the most abrupt changes are visible in this plot, the more subtle ones get completely unnoticed to the naked eye.

d This is the non-windowed overview plot, as presented previously.

e These are the non-windowed evolution plots, as presented previously.

This notebook extract is obviously incomplete, only three evolution plots are shown. It also contains in a second part the windowed versions of the overview plot and evolution plots.

11.2.5 Detected events

Our non-regression tests have been used on a regular basis on twelve Grid'5000 clusters during a period of several months (from July 2019 to March 2021 for `dahu`, the first cluster we started to monitor). In total, we have tested 454 nodes (792 processors).

The list and basic characteristics of these clusters is described in table 11.1. It has been curated from Grid'5000 documentation[10].

Table 11.1.: List of the Grid'5000 clusters covered by our tests.

Cluster	Nodes	CPU	Cores	Memory
`chetemi`	15	2× Intel Xeon E5-2630 v4	2 × 10	256 GiB
`chiclet`	8	2× AMD EPYC 7301	2 × 16	128 GiB
`dahu`	32	2× Intel Xeon Gold 6130	2 × 16	192 GiB
`ecotype`	48	2× Intel Xeon E5-2630L v4	2 × 10	128 GiB
`grisou`	51	2× Intel Xeon E5-2630 v3	2 × 8	128 GiB
`gros`	124	1× Xeon Gold 5220	1 × 18	96 GiB
`grvingt`	64	2× Intel Xeon Gold 6130	2 × 16	192 GiB
`parasilo`	28	2× Intel Xeon E5-2630 v3	2 × 8	128 GiB
`paravance`	72	2× Intel Xeon E5-2630 v3	2 × 8	128 GiB
`pyxis`	4	2× ThunderX2 99xx	2 × 32	256 GiB
`troll`	4	2× Intel Xeon Gold 5218	2 × 16	384 GiB
`yeti`	4	4× Intel Xeon Gold 6130	4 × 16	768 GiB

We give in this section an exhaustive list of the events detected by running our tests.

[10] `https://www.grid5000.fr/w/Hardware`

BIOS upgrade

The Grid'5000 technical team occasionally upgrades the BIOS and the firmware of all the nodes of a cluster. Most of the time, it does not affect the nodes noticeably, but we did measure a significant change on several occasions.

- On 2020/02/06, cluster `gros`. The temperature of all the nodes increased by several degrees. In a more subtle way, but still significant, the frequency and the `dgemm` performance have decreased on a large fraction of the nodes (T-test with a $95\,\%$ confidence).
- On 2020/04/01, cluster `dahu`. The upgrade caused a performance drop of $5\,\%$ on node `dahu-26`, as well as a decrease of its frequency and temperature. There was also a statistically significant (albeit very small) change on the other nodes of the cluster (T-test with a $95\,\%$ confidence). This issue on `dahu-26` has later been resolved by an administrator doing a power drain of the node. Note that this return to normal was also detected by our tests.
- On 2020/06/10, cluster `gros`. All the nodes of the cluster had a performance drop of about $1\,\%$. The frequency has also dropped noticeably.
- On 2020/09/29, cluster `troll`. A very large temperature drop followed the upgrade. The largest effect was on CPU n°1 of node `troll-2`, where the temperature decreased from $86\,°\mathrm{C}$ to $60\,°\mathrm{C}$. Not all the processors of the cluster encountered this temperature anomaly. In a more subtle but still significant way, the `dgemm` performance of several processors has increased. The frequency was not affected.
- On 2020/10/01, cluster `gros`. The average `dgemm` performance of nearly all nodes had a small but significant increase. The frequency has also slightly increased, whereas the temperature was not affected.
- On 2020/12/15, cluster `dahu`. A large fraction of the nodes had a performance increase of $1\,\%$. The frequency also slightly increased and the temperature decreased.

Cooling issue

Four nodes from cluster `dahu` encountered some issues in two occasions, namely `dahu-13`, `dahu-14`, `dahu-15` and `dahu-16`. Their performance and frequency was lower by $10\,\%$ whereas their temperature was much higher, especially on their CPU n°0 where it went above $90\,°\mathrm{C}$ instead of the usual $70\,°\mathrm{C}$. Our hypothesis is that

for some unknown reason the cooling system of the nodes started to malfunction. It is interesting to note that the four nodes are located in the same chassis, which probably explain why they were all affected at the same time.

This problem appeared a first time between October 2018 and March 2019. We did not perform regular measures on the platform at the moment, so we do not have a more precise timeframe. The issue was solved on 2019/05/15 by changing the node chassis.

A few months later, the same issue arose again on the same four nodes, their performance and frequency had a very large drop and their temperature rose dramatically. We do not have an accurate date for this event, it could have happened between 2019/08/20 and 2019/09/16. The problem was solved on 2019/11/27 as a side effect of some work done in the server room. A cluster was installed, so some computer racks were moved and some cable management was done.

Faulty memory

Occasionally, some nodes became extremely slow. The performance difference was so large that our test program did not even have the time to terminate in the usual timeframe we use for the jobs. Thus, we did not detect these events as a non-green point in the plots, but simply because our jobs were failing.

We were able to reproduce the issue with a much simpler program that was stressing the memory, by calling the function `memset` on all the cores of the node simultaneously. Each time, it was located on a single processor of the node, the other processor was functioning normally.

This issue was noticed on the following instances:

- Node `yeti-3` on 2019/07/05.
- Nodes `dahu-20` and `dahu-24` on 2019/08/14.
- Nodes `dahu-20` and `dahu-22` on 2019/11/03.
- Node `dahu-7` on 2020/09/17 and on 2020/09/28.
- Node `dahu-14` on 2020/09/28.

The Grid'5000 team was able to solve this problem, sometimes by swapping two memory sticks, sometimes by replacing one memory stick, sometimes by simply performing a power drain of the nodes.

Power instability

The CPU power consumption on nodes from cluster dahu is very stable when the dgemm calls are performed. On a normal experiment, each individual processor has an average consumption between 124.63 W and 124.65 W. Yet, sometimes one node has a large drop of the CPU power consumption on both its processors. We identified at least 68 events of one node having the consumption of one of its processors below 124 W, which is already very significant given the extreme stability that we usually observe. On half of these events, the power consumption was below 119.4 W. The largest power drop happened on node dahu-26 on 2020/12/04, where its CPU n°0 consumed only 97.5 W. For reasons we ignore, such power drops happened particularly frequently on the four nodes dahu-29, dahu-30, dahu-31 and dahu-32 between November 2020 and February 2021. Since the four nodes are part of the same chassis and therefore share their power supply, our first hypothesis was that there could be an issue with the power supply itself.

Whenever these power drops happens, both the frequency and the dgemm performance also have a huge drop, up to 10 %, whereas the temperature is unaffected.

Every time, this was a temporary anomaly, the affected nodes were back to normal on the following experiments. For this reason, we did not add a new platform change for these events, instead we marked them as outliers in the plots.

After some investigations with Grid'5000 staff, we realized that most of these power drops started to happen when a new cluster, called drac, was installed in the same machine room. It also appears that the jobs where we observed a power drop had been scheduled when most of the drac nodes were used, on the other hand we did not observe any drop when no job was scheduled on the drac nodes. Submitting an idle job on the whole drac cluster (i.e. all the nodes were booted, but idle) was enough to cause a power drop on some dahu nodes.

This drac cluster is made of 12 nodes, each node has two Power8 CPUs and four Nvidia Tesla P100 GPU. When booted but idle, a single node has a power consumption between 600 W and 800 W, this consumption can rise to more than 2000 W when the nodes are under heavy load. This is a large consumption compared to other Grid'5000 nodes (the dahu nodes consume about 350 W at most).

These later findings tend to confirm our initial hypothesis: the power supply cannot sustain enough power, which results in a power drop, a frequency drop and a performance drop on some nodes of dahu cluster.

System upgrade

On March 2021, we decided to upgrade both the operating system and the BLAS library used to perform these tests. We upgraded Debian from version 9 to version 10 (thereby upgrading GCC from version 6.3.0 to version 8.3.0 and the Linux kernel from version 4.9.0 to version 4.19.0). We also upgraded OpenBLAS from version 0.3.1 to version 0.3.13. OpenBLAS had several changes between these two versions, including the AVX512 support for `dgemm` (until now, they only used AVX2).

This resulted in huge changes on a lot of clusters. For instance, on cluster `dahu`:

- The average performance increased from $27\,\mathrm{GFlop/s}$ per core to $45\,\mathrm{GFlop/s}$.
- The average frequency decreased from $2.2\,\mathrm{GHz}$ to $1.8\,\mathrm{GHz}$.
- The mean power consumption of the DRAM increased from $12\,\mathrm{W}$ to $15\,\mathrm{W}$.
- The mean power consumption of the CPU and the temperature were not affected significantly.

The same observations could be made on nearly all clusters. We also observed significant drops of CPU power consumption on some nodes of a few clusters (e.g. on `troll` or `gros` clusters).

The performance increase is understandable, `dgemm` is a compute-intensive kernel for dense linear algebra, it was expected that wider vector instructions would be more efficient. It is also expected that the maximal turbo frequency of processors depends on the workload, for instance Intel Xeon Gold 6130 CPU (the processor of the `dahu` nodes) has a maximal frequency of $2.8\,\mathrm{GHz}$ when all cores are active with normal instructions, $2.4\,\mathrm{GHz}$ when all cores are active with AVX2 instructions and $1.9\,\mathrm{GHz}$ when all cores are active with AVX512 instructions[11].

Other issues

Several other significant and durable changes have been detected on individual nodes. To this day, we were unable to determine their root cause. On seven nodes of four different clusters, the temperature has inexplicably dropped by several degrees, from $5\,^{\circ}\mathrm{C}$ to $15\,^{\circ}\mathrm{C}$ depending on the nodes. Some of them were back to normal a few weeks later, but others remained in this state. Table 11.2 summarizes those unexplained changes. Most of the time, the frequency and `dgemm` performance were also slightly affected, but to a lower extent.

[11] `https://en.wikichip.org/wiki/intel/xeon_gold/6130`

Table 11.2.: Unexplained changes detected on Grid'5000 nodes.

Node	Date	Back to normal	Temperature drop
ecotype-24	2020/05/21	2020/07/02	15 °C
ecotype-47	2020/07/13	NA	15 °C
grisou-12	2020/08/06	2021/01/23	15 °C
parasilo-1	2020/09/13	NA	10 °C
parasilo-11	2020/09/19	NA	15 °C
dahu-29	2020/10/15	2020/12/15	5 °C
dahu-30	2020/10/15	2020/12/15	5 °C

The two nodes parasilo-1 and parasilo-11 also had a temperature increase of 5 °C a few weeks later, on 2020/10/06. This new change was not enough to revert the effect of the first change.

The two nodes dahu-29 and dahu-30 had a temperature increase a few weeks later and went back to normal, this change coincided with the BIOS upgrade that happened on the whole cluster.

11.3 Conclusion and future work

We have implemented statistical tests for detecting performance regressions on computers. The novelty of our work does not reside on the statistics, our approach is entirely based on a 55-year-old paper. However, to the best of our knowledge, we are the first to apply these statistics for testing computer performance.

We have monitored 454 nodes from Grid'5000 testbed for more than one year, running new tests several times a week. This allowed us to detect multiple events that had a significant effect on the nodes, from subtle performance changes of 1 % to much more severe degradations of more than 10 %, or even nodes that were literally unusable when their memory was under heavy load. These events went unnoticed by both Grid'5000 technical team and Grid'5000 users, yet they could greatly harm the reproducibility of experiments and lead to wrong scientific conclusions. We therefore believe that our approach could greatly benefit to the HPC community.

There remains some engineering work before targeting a broader adoption. Some parts of our workflow should be re-implemented with other more suitable technologies, for instance the data should be stored with a proper database management system. There also remains some automation to implement, in particular the scheduling of new experiments, with the constraint that it should not bother other users too much.

Our test currently relies solely on performance measures for the `dgemm` function. This is a CPU intensive workload that makes a heavy use of vector floating-point operations and, to a lesser extent, also stresses the memory. For a broader coverage, it would be interesting to implement new tests that stress other parts of the platform, e.g. with workloads that perform many memory operations, disk operations, or even network communications. To this end, the stress-ng [Stress-ng] benchmark would be a great source of inspiration.

12 Discussion

12.1 Contribution

This thesis has contributed to the improvement of experimental reproducibility in high performance computing.

In Part I, we described a method for predicting the performance of MPI applications through simulation. Using Simgrid/SMPI simulator, we managed to emulate the High Performance Linpack benchmark at scale by applying only a few modifications to its source code. We compared several computation and communication models and showed the importance of modeling both the temporal and spatial variability of the platform. In a thorough comparison of the simulations with real executions, we show that the prediction error remains very low, only a few percent, thereby demonstrating the faithfulness of this approach. Several sensibility analyses are then performed to quantify the effect of platform variability and showcase an important use case of simulation.

The lessons learned during this work are then presented in Part II. We start by describing the experiment engine we developed and that was used throughout this thesis. Then, we present an in-depth report of the many experimental biases we faced, including very unsettling phenomenons we did not anticipate. While some of these biases can be desirable if they are also occurring in the simulated application, most of them had to be suppressed through randomization. Finally, we showcase the performance non-regression test we implemented. While not a statistical novelty, they allowed us to detect many changes on Grid'5000 platform that affected significantly the performance and could harm experiments if gone unnoticed. We believe the HPC community could greatly benefit of such tests.

12.2 Trusting our predictions

Unlike mathematical theorems or algorithms, the correctness of a model like those of Part I cannot be formally proven. The only sound method for validating its

faithfulness is to thoroughly try to break it, by comparing predictions to reality while methodically changing the configurations. As presented in Chapter 6, we did cover an extensive range of configurations in our validation. Before managing to systematically obtain accurate predictions, we stumbled on many problems which caused our predictions to be unrealistic. We had to investigate, understand and overcome these multiple obstacles, as reported in Part II.

A few weeks before the defense of this thesis, we decided to repeat the whole simulation study from scratch on another Grid'5000 cluster named `gros`, using 60 of its nodes. This cluster has different nodes (one Intel Xeon Gold 5220 processor per node with $96\,\mathrm{GB}$ of memory), a different network (with $25\,\mathrm{Gbit/s}$ Ethernet) and we used a more recent version of OpenBLAS (resulting in the use of AVX512 instructions by the `dgemm` function instead of AVX2). Over the course of a weekend, we calibrated the platform by measuring the `dgemm` and MPI durations. Then, we performed real and simulated HPL executions. At first, the predicted performance was too low by one order of magnitude. This was due to a mistake we made: we computed the linear regression of the `dgemm` model using all the terms of the polynomial, even those of degree one (i.e. M, N and K) which were not statistically significant. This resulted in an overfitted model with spurious predictions. After fixing this issue by only considering the significant terms, we obtained extremely accurate predictions of HPL performance for various matrix sizes (see Figure 12.1).

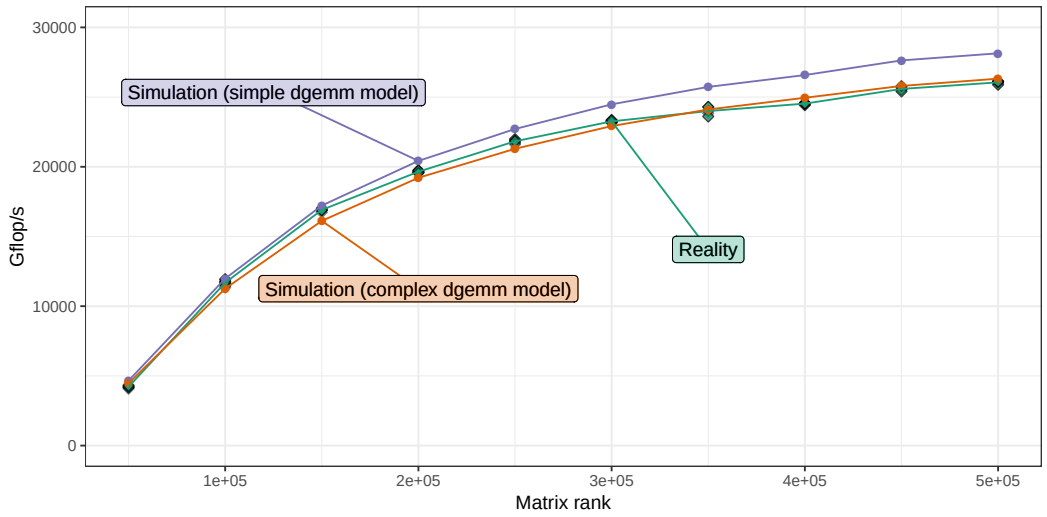


Figure 12.1.: HPL performance: prediction vs. reality for various matrix ranks, using 60 nodes of the `gros` cluster.

Similarly to what we observed in Chapter 6, the predictions are more accurate with a complex `dgemm` model (i.e. heterogeneous, stochastic and polynomial) than with a simpler model (i.e. homogeneous, deterministic and linear). However, the simpler

model still makes reasonably low prediction errors here as the `gros` cluster has less variability than the `dahu` cluster.

This small additional study demonstrates clearly the level of trust we can now have in our model, but also its fragility, as seemingly inoffensive changes in the approach can lead to widely inaccurate predictions.

12.3 Future work

Besides the next steps already discussed in Chapter 8 and Section 11.3, there also remains a unification work. By going a step further in the automation, we could use the data produced by the non-regression tests to generate new model instances for Simgrid. It would then be possible to make new simulations with a platform model that reflects the latest changes of the real platform. Then, by implementing the same statistical test on the performance predicted by the simulation, we should be able to detect whether a platform change has affected the application performance and quantify this effect. A minor performance drop of the computation kernels could be amplified by the synchronization phases of the application and become very concerning at a larger scale. Conversely, it could also be attenuated by the global noise and go completely unnoticed.

We believe that our predictions could be extremely valuable to the whole life cycle of supercomputers:

Design Using simulations, manufacturers could apply co-design techniques to construct the most performant machines for a given set of target applications and within a given budget. This could help achieve more faithful results than the current techniques relying on less accurate simulations or even guesswork.

Development Simulations could also largely benefit software developers. Both debugging and tuning the application are more convenient and less expensive in simulation than in reality, especially if large scale runs are required.

Maintenance Whenever the platform employees need to perform some maintenance, there is a non-negligible risk of affecting the machine performance, as we illustrated in Chapter 11. To verify that the performance did not change, the usual method is to perform large-scale runs of a benchmark such as HPL, which can largely lengthen the maintenance duration. A more convenient alternative would be to (1) carry small but carefully designed performance tests as those described in Chapter 11 to check if there is anything obviously

wrong, and (2) perform large-scale simulations with the updated model and compare the new predictions with the previous ones.

A Using a laboratory journal for better reproducibility

Throughout the several years of this thesis, we used meticulously a laboratory journal, inspired by the lectures from Legrand et al. [PLH20; LV20]. Since we firmly believe that this was a great help for improving the reproducibility of this work, this chapter succinctly describes our methodology.

The journal is written with Org-mode [ORG]. This is a mode from the Emacs text editor for editing, formatting and organizing text documents, using a lightweight markup language. It offers multiple functionalities such as hierarchy within a file, TODO lists, tags, hyperlinks, file attachments and even code execution. We use git for versioning and collaborating.

The journal follows a chronological order and hierarchy, as illustrated by Figure A.1. There is one main part for each year, then subparts for months and days. Within the part of a given day, we create additional subparts, called *journal entries*, related to the topic that should be recorded. Each part can be folded and unfolded to keep a reasonable amount of displayed text. For instance, in Figure A.1, the first journal entry of 2021 is simply a link to a video presentation.

Each journal entry can have one or several tags, displayed on the right in upper-case and delimited by colons. These keywords are used for organizing the different entries by semantic. For instance, we can easily list all the entries related to experiments made on the `dahu` cluster by making a search for the tag `DAHU`.

We used three kinds of journal entries:

Experiment analysis All the experiments carried during this thesis have been executed by the `peanut` experiment engine (see Part II), resulting in an archive containing data and meta-data for the experiment. Then, we collected, analyzed and made plots for these archives using literate programming, with Jupyter notebooks [IPYNB]. Although this could be done directly within Org-mode, thanks to its code execution capability, we simply preferred the Jupyter ecosystem. When the analysis was done, we compiled the notebook to HTML, created a new entry in the journal and attached the notebook. This process

has been automatized by a Python script we made, called `org_attach` [CB21]. This new entry contains at least two parts, one to summarize the experiment (what, why and how) and one to list the various observations we made in the analysis.

Paper reading Whenever we find an interesting paper, we create a new journal entry, named after the paper title. The entry is marked with a keyword `READ` or `UNREAD` if we have already read the paper or not. The entry also contains at least three parts, namely the paper abstract, any notes we may have taken while reading the paper, and the bibtex. Finally, the PDF file of the paper is attached to the entry. Again, the process of creating the new entry and attaching the paper is automatized by `org_attach`.

Free text We also write in the journal any relevant material. This can be a discussion with colleagues, a seminar we attended, some unorganized thoughts, a short piece of code to test a new idea or demonstrate a new tool, etc. Again, the use of proper tags is important to be able to find these entries after a while.

In the end, the text file of the journal has now more than one thousand entries. The text file has a size of more than $2.7\,\mathrm{MB}$, while the attachments totalize more than $1.1\,\mathrm{GB}$.

Such a rigorous method can appear as a waste of time, even overwhelming. We argue that despite the steep learning curve of org-mode, this time investment was worth it. We were able to come back to some early experiments several years later, understand precisely what these experiments were doing, and know with certitude the software versions that were used. Furthermore, we could also read the thought process we had at the time, the hypotheses we made that were later confirmed or rebutted. This is obviously of great help for reproducing these experiments or writing this thesis.

```
◉ G5K bugs reported...
◉ 2018...
◉ 2019...
◉ 2020...
◉ 2021
  ○ 2021-01 January
    ✸ 2021-01-04 Monday
      ✿ Video presentation of stress-ng            :STATE_OF_THE_ART:THESIS:
        Link: https://youtu.be/8QaXStKfq3A
    ✸ 2021-01-05 Tuesday
      ✿ READ The Grey Hoodie Project: Big Tobacco, Big Tech, and the threat on academic integrity :PAPER:ATTACH:
        :PROPERTIES:
        :DOI:
        :URL:
        :AUTHORS: Mohamed Abdalla, Moustafa Abdalla
        :Attachments: The_Grey_Hoodie_Project:_Big_Tobacco,_Big_Tech,_and_the_threat_on_academic_integrity.pdf
        :ID: 977917fc68206cf0b98e701f50e7e7570be1728c30da28719d53fa98f646a3ea07834100295a4b8d8487858389ccaaba67eeac329e2
        :END:
        ◉ Summary...
        ◉ Notes...
        ◉ Open Questions [0/2]...
        ◉ BibTeX
          #+BEGIN_SRC bib :tangle bibliography.bib
          @misc{abdalla2021grey,
              author = "Abdalla, Mohamed and Abdalla, Moustafa",
              title = "The Grey Hoodie Project: Big Tobacco, Big Tech, and the threat on academic integrity",
              year = "2021",
              eprint = "2009.13676",
              archivePrefix = "arXiv",
              primaryClass = "cs.CY"
          }
          #+END_SRC
      ✿ Performance Monitoring in the Linux Kernel  :STATE_OF_THE_ART:THESIS:PAPER:ATTACH:...
    ✸ 2021-01-07 Thursday...
    ✸ 2021-01-08 Friday...
    ✸ 2021-01-11 Monday
      ✿ Discussion (tests de non régression)                :ARNAUD:MEETING:
        ◉ État de l'art
          - Mieux détailler la différence entre ce qui est fait par gbench et starpu
            (micro-bench vs full app, comparaison de deux runs vs série de tests à long
            terme). Faire un paragraphe spécifique pour StarPU.
        ◉ Tests G5K...
      ✿ Should we adapt the g5k-test for factors with extremely low variability? :G5K:DAHU:STATS:NOTEBOOK:PYTHON:ATTACH:
        :PROPERTIES:
        :LANGUAGE: python
        :VERSION: 3.7.3
        :Attachments: g5k_test.html
        :ID: a60c009de2b60350187ab1b88c8a12967aef108d45b1a8667f8a46ba90662b7d4da5b88042eaabd6850b617657b39019fa0e727840c
        :END:
        ◉ Summary
          In attachment is a (modified) g5k-test notebook for the CPU power consumption of
          dahu. The original can be found here.

          On nearly all the g5k clusters, the CPU power consumption is extremely stable. I
          investigate whether we should modify our non-regression tests for such factors.
        ◉ Notes
          In the 88 experiments I made on dahu-1 since June, the minimal power consumption
          I observed is 124.635767 W while the maximal is 124.639291 W (a variation of
          0.002% between the min and the max).

          It is still interesting to make a test on this factor, since on some nodes we
          sometimes have much larger differences, as can bee seen on cell 9:
          - the node dahu-26 had a power consumption of about 100W on a single day,
          - the nodes dahu-{29..32} had low and variable power consumption during several
            weeks.

          Something unfortunate is that the anomalies on those nodes are completely hidden
          by other tiny "anomalies" in the overview plot from cell 10.
```

Figure A.1.: Screenshot of the laboratory journal used throughout this work.

List of software and data

B

As this thesis was above all experimental, we implemented several programs to assist us in our work. Furthermore, the many experiments that we performed generated large amounts of data. For the sake of reproducibility, both software and data are permanently archived on Zenodo. The full list is described in Table B.1.

Table B.1.: Software and data produced during this thesis.

[Cor21c]	`peanut` The experiment engine described in Section 9.2, written in Python. It allows to easily write scripts for deploying and running experiments, producing an archive with the experiment data and meta-data.
[CL21b]	`cashew` The Python library described in Section 11.2.2 used for extracting the relevant data and meta-data from `peanut` archives, summarizing the data, computing non-regression tests on the aggregated data and generating the notebooks.
[CB21]	`org_attach` A small Python script described in Appendix A to add an entry in an Org-mode laboratory journal with an attachment (either a paper or a Jupyter notebook).
[CL21a]	`calibration_analysis` Nearly all the experiment data produced during this thesis (i.e. `peanut` archives) and the jupyter notebooks to analyze it. This includes MPI calibrations, `dgemm` calibrations, real and simulated runs of HPL, etc.
[CL21c]	`g5k_test` The data and notebooks of the performance non-regression tests made on Grid'5000. This includes all the `peanut` archives, the HDF5 files containing the raw data, the CSV files containing the aggregated data and a full copy of the website presenting the test results.
[Pet+21]	`hpl` The source code of High Performance Linpack, with the modifications we had to make for simulating its execution on top of SMPI.
[SG21]	`platform-calibration` The micro-benchmarks used to calibrate the models for the durations of `dgemm` and MPI communications.
[CL21d]	`pycewise` The Python library described in Section 5.3.3 for automatically computing piecewise linear regressions.
[CM21]	`ratatouille` A Python script launched as a background process in experiments for monitoring system metrics such as the CPU temperature, core frequencies, CPU and DRAM power consumption. These metrics are collected at a fixed pace (e.g. every five seconds) and written in a CSV file.

Additional material has also been archived once the defense had been carried out, the full list is described in Table B.2.

Table B.2.: Additional material archived after this thesis.

[Cor21a]	`defense` The video of this thesis defense, which was live-streamed on YouTube.
[Cor21b]	`manuscript` The repository used for writing this manuscript and the slides of the defense. Besides the text and images, it also contains the code and a copy of the data used for generating the figures.

Unfortunately, the laboratory journal described in Appendix A could not be published, as it contained numerous copyrighted articles as well as confidential information.

Carbon footprint of this thesis

C

This thesis was written between November 2020 and March 2021, in the midst of the COVID-19 pandemic. At the time of the writing, more than two millions persons had already passed away and several millions will probably have long-term medical issues. Yet, this world disaster is arguably much less threatening than global warming. It is now established that the average temperature on Earth is increasing, mainly caused by anthropogenic emissions of greenhouse gases. It is difficult to estimate the damage that will be caused by global warming. The World Health Organization (WHO) writes that it is already causing $150{,}000$ excess deaths per year [WHOb] and that this figure is likely to rise to $250{,}000$ per year between 2030 and 2050 [WHOa], mainly caused by heat exposure, diarrhea, malaria and childhood undernutrition. For the year 2018, according to the French government, the average carbon footprint of a French person is estimated to $11\,\mathrm{t}$ of CO2eq. This is more than five times larger than the yearly $2\,\mathrm{t}$ target to limit the warming to $2\,^{\circ}\mathrm{C}$ [MTE].

For this reason, we felt it was important to compute the direct effect of this thesis on global warming. This chapter tries to estimate the greenhouse gases emission we generated, in carbon dioxide equivalent. We account for the two principal sources of CO2 emissions, i.e. the business trips we made and the computing time we used on Grid'5000. The goal is not to compute an exact figure, which would be extremely tedious, but rather to make a rough estimate and compare it to the $2\,\mathrm{t}$ target.

Business trips

Several business trips done during this thesis required to take a plane. In the following, we will use the value of $195\,\mathrm{g/km}$ of CO2eq for long haul flight and $254\,\mathrm{g/km}$ for short haul flights [BBC]. We will neglect the emissions caused by the train travels, as this transportation mean is much less problematic (a French high speed train emits about $2\,\mathrm{g/km}$ for each passenger).

- On April 2017, I attended a Simgrid meeting in Bordeaux. I had to go by plane from Lyon ($435\,\mathrm{km}$) as the train lines are very Paris-centered in France. This trip emitted about $0.2\,\mathrm{t}$ of CO2eq.

- From October to December 2017, I visited Argonne National Laboratory in Chicago, USA. The outward flights went from Lyon to Frankfurt ($562\,\mathrm{km}$) then to Chicago ($6967\,\mathrm{km}$). The return flight went directly from Chicago to Paris ($6653\,\mathrm{km}$). In total, this trip has emitted approximately $2.8\,\mathrm{t}$ of CO2eq.

- In November 2017, I attended the Supercomputing conference in Denver, USA. I went there with a direct flight from Chicago ($1475\,\mathrm{km}$), emitting about $0.7\,\mathrm{t}$ of CO2eq.

- In September 2019, I presented a paper at the Cluster conference, in Albuquerque, USA. The outward flights went from Paris to Los Angeles ($9085\,\mathrm{km}$) then to Albuquerque ($1067\,\mathrm{km}$). The return flights passed by Dallas ($586\,\mathrm{km}$), then New York City ($2206\,\mathrm{km}$) and finally Paris ($5837\,\mathrm{km}$). This trip emitted approximately $3.9\,\mathrm{t}$ of CO2eq.

Hence, the total amount of greenhouse gas emissions of this thesis due to airplane transportation is $7.6\,\mathrm{t}$ of CO2eq.

Computing time

This thesis had a very important experimental component, it is therefore natural to expect a large amount of computing time usage. Figure C.1 summarizes the cumulated computing time spent on Grid'5000 clusters during these years, grouped by usage type, for a total of $2{,}112{,}014$ core hours. The performance tests described in Chapter 11 represent slightly more than half of this total time, which is very large. However, this should be contrasted by the consumption of the official Grid'5000 tests: the `ajenkins` user, a bot responsible for periodically verifying the integrity of the platform, spent $3{,}272{,}044$ core hours in the year 2020 alone.[1]

Berthoud et al. estimate the greenhouse gas emissions caused by GRICAD another French computing center [Ber+20]. They tried to account for several parameters, including the electrical consumption of the hardware (computing nodes, storage nodes, network, cooling system), the manufacturing of this hardware, but also the emissions caused by the employees in charge of operating this computing center in their daily commute to work. They estimate that one hour of computation on one core is responsible for the emission of $5\,\mathrm{g}$ of CO2eq. The distribution of these emissions is depicted in Figure C.2, slightly more than half of these $5\,\mathrm{g}$ is caused by the electrical consumption of the machine. It should be noted that French electricity

[1] `https://intranet.grid5000.fr/stats/users.html`

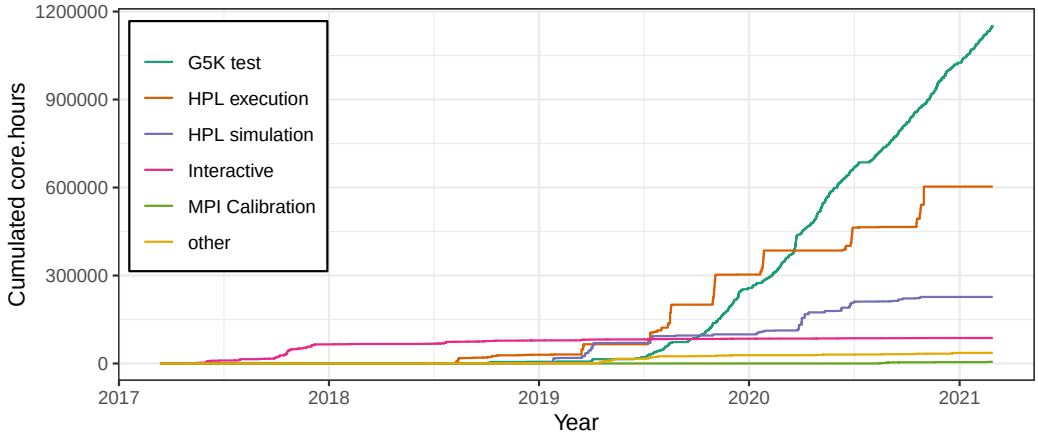


Figure C.1.: Core hours used throughout this thesis for each experiment kind. In total, more than $2{,}112{,}014$ core hours have been used.

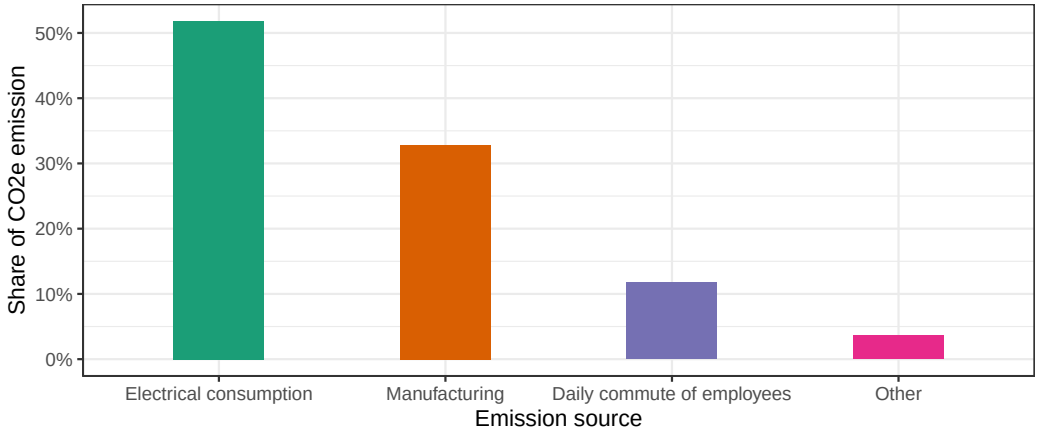


Figure C.2.: Source of greenhouse gases emission in a French scientific computing center, one core hour emits approximately $5\,\mathrm{g}$ of CO2eq [Ber+20].

is particularly low-carbon, meaning that this figure might be significantly larger in other countries that rely more on fuel and coal.

In the end, we estimate that the total amount of greenhouse gas emissions of this thesis due to computing is $10.6\,\mathrm{t}$ of CO2eq.

Conclusion

In this chapter, we tried to estimate the carbon footprint of this thesis. We do not claim to have an accurate figure to present (these back-of-the-envelope calculations are themselves based on rough approximations) but we should have the correct order of magnitude.

We estimate that this thesis was responsible for the emission of $18.2\,\mathrm{t}$ of CO2eq. This is significantly larger than the average yearly emission of a French person ($11\,\mathrm{t}$ of CO2eq) and six times larger than the yearly target emission to limit the warming to $2\,^{\circ}\mathrm{C}$ ($2\,\mathrm{t}$ of CO2eq). Maybe counter-intuitively, this thesis emitted more greenhouse gases with computations than with airplane transportation, despite four transatlantic flights.

Bibliography

[Age+15] Anthony Agelastos, Benjamin Allan, Jim Brandt, et al. "Toward Rapid Understanding of Production HPC Applications and Systems". In: *2015 IEEE International Conference on Cluster Computing*. IEEE, Sept. 2015. DOI: 10.1109/cluster.2015.71. cit. on p. 7

[Ahn+21] Dong H. Ahn, Allison H. Baker, Michael Bentley, et al. "Keeping Science on Keel When Software Moves". In: *Commun. ACM* 64.2 (Jan. 2021), pp. 66–74. DOI: 10.1145/3382037. cit. on p. 8

[Ale+97] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. "LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation". In: *Journal of Parallel and Distributed Computing* 44.1 (1997), pp. 71–79. DOI: 10.1006/jpdc.1997.1346. cit. on p. 16

[And+20] Etienne André, Remi Dulong, Amina Guermouche, and François Trahay. "DUF : Dynamic Uncore Frequency scaling to reduce power consumption". working paper or preprint. Feb. 2020. URL: https://hal.archives-ouvertes.fr/hal-02401796. cit. on p. 110

[Arrow] [SW], *Apache Arrow*, LIC: Apache. URL: https://arrow.apache.org/, VCS: https://github.com/apache/arrow. cit. on p. 127

[Aug+11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures". In: *CCPE - Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009* 23 (2 Feb. 2011), pp. 187–198. DOI: 10.1002/cpe.1631. URL: http://hal.inria.fr/inria-00550877. cit. on p. 118

[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. 17

[Bak16] Monya Baker. *1,500 scientists lift the lid on reproducibility*. 2016. URL: https://www.nature.com/news/1.19970 (visited on Mar. 15, 2021). cit. on p. 8

[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. DOI: 10.1007/978-3-319-04519-1_1. cit. on pp. 31, 81

[BBC] BBC. *Climate change: Should you fly, drive or take the train?* 2019. URL: https://www.bbc.com/news/science-environment-49349566 (visited on Feb. 27, 2021). cit. on p. A7

[Ber+20] Francoise Berthoud, Bruno Bzeznik, Nicolas Gibelin, et al. *Estimation de l'empreinte carbone d'une heure.coeur de calcul*. Research Report. UGA - Université Grenoble Alpes ; CNRS ; INP Grenoble ; INRIA, Apr. 2020. URL: https://hal.archives-ouvertes.fr/hal-02549565. cit. on pp. A8, A9

[Bha+13] Abhinav Bhatele, Kathryn Mohror, Steven H. Langer, and Katherine E. Isaacs. "There goes the neighborhood". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, Nov. 2013. DOI: 10.1145/2503210.2503247. cit. on p. 7

[Bir+13] R. F. Bird, S. A. Wright, D. A. Beckingsale, and S. A. Jarvis. "Performance Modelling of Magnetohydrodynamics Codes". In: *Computer Performance Engineering*. Ed. by Mirco Tribastone and Stephen Gilmore. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 197–209. cit. on p. 18

[Boh07] Mark Bohr. "A 30 Year Retrospective on Dennard's MOSFET Scaling Paper". In: *IEEE Solid-State Circuits Newsletter* 12.1 (2007), pp. 11–13. DOI: 10.1109/n-ssc.2007.4785534. cit. on p. 5

[Bra+10] James Brandt, Frank Chen, Vincent De Sapio, et al. "Quantifying effectiveness of failure prediction and response in HPC systems: Methodology and example". In: *2010 International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, June 2010. DOI: 10.1109/dsnw.2010.5542629. cit. on p. 7

[Bra+15] J. Brandt, D. Debonis, A. Gentile, et al. *Enabling Advanced Operational Analysis Through Multi-subsystem Data Integration on Trinity.* 2015. cit. on p. 7

[Buc+15] Tomasz Buchert, Cristian Ruiz, Lucas Nussbaum, and Olivier Richard. "A survey of general-purpose experiment management tools for distributed systems". In: *Future Generation Computer Systems* 45 (2015), pp. 1–12. DOI: 10.1016/j.future.2014.10.007. URL: https://hal.inria.fr/hal-01087519. cit. on p. 82

[Car+17] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, et al. "Stan: A Probabilistic Programming Language". In: *Journal of Statistical Software* 76.1 (2017). DOI: 10.18637/jss.v076.i01. cit. on p. 45

[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 (2014), pp. 2899–2917. DOI: 10.1016/j.jpdc.2014.06.008. URL: https://hal.inria.fr/hal-01017319. cit. on pp. 17, 19, 116

[Cas+15] Henri Casanova, Frédéric Desprez, George S. Markomanolis, and Frédéric Suter. “Simulation of MPI applications with time-independent traces”. In: *Concurrency and Computation: Practice and Experience* 27.5 (Apr. 2015), p. 24. DOI: 10.1002/cpe.3278. URL: https://hal.inria.fr/hal-01232776. cit. on p. 17

[Catch2] [SW], *Catch2*, LIC: BSL. VCS: https://github.com/catchorg/Catch2. cit. on p. 117

[CB13] Charlie Curtsinger and Emery D. Berger. “STABILIZER: Statistically Sound Performance Evaluation”. In: *SIGARCH Comput. Archit. News* 41.1 (Mar. 2013), pp. 219–228. DOI: 10.1145/2490301.2451141. cit. on p. 8

[CB21] [SW] Tom Cornebize and Pedro Bruel, *org_attach*, Mar. 2021. LIC: MIT. DOI: 10.5281/zenodo.4625618, VCS: https://github.com/Ezibenroc/org_attach. cit. on pp. A2, A5

[Celero] [SW], *Celero*, LIC: Apache. VCS: https://github.com/DigitalInBlue/Celero. cit. on p. 117

[Che66] Victor Chew. “Confidence, Prediction, and Tolerance Regions for the Multivariate Normal Distribution”. In: *Journal of the American Statistical Association* 61.315 (1966), pp. 605–617. DOI: 10.2307/2282774. cit. on p. 120

[Cisco] Cisco. *Cisco Bug: CSCtf52095 - Manually Flushing OS Cache during load impacts server*. 2017. URL: https://quickview.cloudapps.cisco.com/quickview/bug/CSCtf52095 (visited on Mar. 14, 2021). cit. on p. 7

[CL19] Tom Cornebize and Arnaud Legrand. *DGEMM performance is data-dependent*. Research Report RR-9310. Université Grenoble Alpes ; Inria ; CNRS, Dec. 2019. URL: https://hal.inria.fr/hal-02401760. cit. on p. 104

[CL21a] [SW] Tom Cornebize and Arnaud Legrand, *Calibration data and analyses*, Mar. 2021. LIC: MIT. DOI: 10.5281/zenodo.4628248. cit. on p. A5

[CL21b] [SW] Tom Cornebize and Arnaud Legrand, *cashew*, Mar. 2021. LIC: MIT. DOI: 10.5281/zenodo.4625736, VCS: https://github.com/Ezibenroc/cashew. cit. on pp. 122, A5

[CL21c] [SW] Tom Cornebize and Arnaud Legrand, *Grid’5000 performance data*, Mar. 2021. LIC: ODbL v1.0. DOI: 10.5281/zenodo.4624877. cit. on pp. 135, A5

[CL21d] [SW] Tom Cornebize and Arnaud Legrand, *pycewise*, Mar. 2021. LIC: MIT. DOI: 10.5281/zenodo.4581362, VCS: https://github.com/Ezibenroc/pycewise. cit. on pp. 50, A5

[CL21e] Tom Cornebize and Arnaud Legrand. “Simulation-based Optimization and Sensibility Analysis of MPI Applications: Variability Matters”. working paper or preprint. Feb. 2021. URL: https://hal.inria.fr/hal-03141988. cit. on p. 13

[CLH19] Tom Cornebize, Arnaud Legrand, and Franz C Heinrich. “Fast and Faithful Performance Prediction of MPI Applications: the HPL Case Study”. In: *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. Albuquerque, United States, Sept. 2019. DOI: 10.1109/CLUSTER.2019.8891011. URL: https://hal.inria.fr/hal-02096571. cit. on pp. 13, 19

[CLT13] Laura Carrington, Michael Laurenzano, and Ananta Tiwari. “Inferring Large-scale Computation Behavior via Trace Extrapolation”. In: *Proc. of the Workshop on Large-Scale Parallel Processing*. 2013. cit. on p. 17

[CM21] [SW] Tom Cornebize and Clément Mommessin, *ratatouille*, Mar. 2021. LIC: MIT. DOI: 10.5281/zenodo.4576010, VCS: https://github.com/Ezibenroc/ratatouille. cit. on pp. 88, A5

[Cop20] B. Jack Copeland. “The Modern History of Computing”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Winter 2020. Metaphysics Research Lab, Stanford University, 2020. URL: https://plato.stanford.edu/archives/win2020/entries/computing-history/. cit. on p. 3

[Cor17] Tom Cornebize. “Capacity Planning of Supercomputers: Simulating MPI Applications at Scale”. MA thesis. Grenoble INP ; Université Grenoble - Alpes, June 2017. URL: https://hal.inria.fr/hal-01544827. cit. on p. 13

[Cor21a] Tom Cornebize. *High Performance Computing: Towards Better Performance Predictions and Experiments - Defense video*. June 2021. URL: https://youtu.be/J3N1qS5gcGI. cit. on p. A6
DOI: 10.5281/zenodo.4946269

[Cor21b] Tom Cornebize. *High Performance Computing: Towards Better Performance Predictions and Experiments - Thesis repository*. June 2021. URL: https://github.com/Ezibenroc/thesis. cit. on p. A6
DOI: 10.5281/zenodo.5017099

[Cor21c] [SW] Tom Cornebize, *peanut*, Mar. 2021. LIC: MIT. DOI: 10.5281/zenodo.4625666, VCS: https://github.com/Ezibenroc/peanut.
cit. on pp. 82, 125, A5

[CPW15] Christian Collberg, Todd Proebsting, and Alex M. Warren. *Repeatability and Benefaction in Computer Systems Research*. Tech. rep. 2015. URL: http://reproducibility.cs.arizona.edu/v2/RepeatabilityTR.pdf. cit. on p. 8

[Cul+93] David Culler, Richard Karp, David Patterson, et al. “LogP: towards a realistic model of parallel computation”. In: *ACM SIGPLAN Notices* 28.7 (July 1993), pp. 1–12. DOI: 10.1145/173284.155333. cit. on p. 16

[Dat] [SW] Datadog, *Piecewise*, LIC: BSD. URL: https://www.datadoghq.com/blog/engineering/piecewise-regression/, VCS: https://github.com/DataDog/piecewise. cit. on pp. 47, 55

[Deg+17] Augustin Degomme, Arnaud Legrand, Georges Markomanolis, et al. “Simulating MPI applications: the SMPI approach”. In: *IEEE Transactions on Parallel and Distributed Systems* 28.8 (Feb. 2017), p. 14. DOI: 10.1109/TPDS.2017.2669305. URL: https://hal.inria.fr/hal-01415484/.
cit. on pp. 19, 20, 21, 35, 66, 77, 92

[Den+74] R.H. Dennard, F.H. Gaensslen, V.L. Rideout, E. Bassous, and A.R. LeBlanc. “Design of ion-implanted MOSFET’s with very small physical dimensions”. In: *Solid-State Circuits, IEEE Journal of* 9.5 (Oct. 1974), pp. 256–268. DOI: 10.1023/A:1008373903657. URL: https://web.ece.ucsb.edu/courses/ECE225/225_W07Banerjee/reference/Dennard.pdf.
cit. on p. 5

[Den11] Alexandre Denis. “A High-Performance Superpipeline Protocol for InfiniBand”. In: *Euro-Par 2011*. Ed. by E. Jeannot, R. Namyst, and J. Roman. Vol. 6853. Lecture Notes in Computer Science. Bordeaux, France: Springer, Aug. 2011, pp. 276–287. URL: https://hal.inria.fr/inria-00586015.
cit. on p. 66

[DHL15] Jack J. Dongarra, Michael A. Heroux, and Piotr Luszczek. *HPCG Benchmark : a New Metric for Ranking High Performance Computing Systems*. Tech. rep. UT-EECS-15-736. Technion Israel Institute of Technology, 2015.
cit. on p. 78

[Dor+14] Matthieu Dorier, Gabriel Antoniu, Robert Ross, Dries Kimpe, and Shadi Ibrahim. “CALCioM: Mitigating I/O Interference in HPC Systems through Cross-Application Coordination”. In: *IPDPS - International Parallel and Distributed Processing Symposium*. Phoenix, United States, May 2014. DOI: 10.1109/IPDPS.2014.27. URL: https://hal.inria.fr/hal-00916091.
cit. on p. 7

[Emm+20] Julien Emmanuel, Matthieu Moy, Ludovic Henrio, and Gregoire Pichon. “Simulation of the Portals 4 protocol, and case study on the BXI interconnect”. In: *HPCS 2020 - International Conference on High Performance Computing & Simulation*. Barcelona, Spain, Dec. 2020, pp. 1–8. URL: https://hal.archives-ouvertes.fr/hal-02972297.
cit. on p. 20

[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: *FGCS* 30 (Jan. 2014), pp. 59–65.
cit. on p. 17

[Esm+11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. “Dark silicon and the end of multicore scaling”. In: *ACM SIGARCH Computer Architecture News* 39.3 (July 2011), p. 365. DOI: 10.1145/2024723.2000108.
cit. on p. 6

[F A+14] Mark F. Adams, Jed Brown, John Shalf, et al. *HPGMG 1.0: A Benchmark for Ranking High Performance Computing Systems*. Tech. rep. LBNL, May 2014.
cit. on p. 78

[For93] The MPI Forum. “MPI: A Message Passing Interface”. In: *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. Supercomputing ’93. Portland, Oregon, USA: Association for Computing Machinery, 1993, pp. 878–883. DOI: 10.1145/169627.169855.
cit. on p. 23

[Fre10] David H. Freedman. *Lies, Damned Lies, and Medical Science*. Nov. 2010. URL: https://www.theatlantic.com/magazine/archive/2010/11/lies-damned-lies-and-medical-science/308269/. cit. on p. 8

[Fri+12] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. "Cache-Oblivious Algorithms". In: *ACM Trans. Algorithms* 8.1 (Jan. 2012). DOI: 10.1145/2071379.2071383. cit. on p. 16

[GBench] [SW], *Google Benchmark*, LIC: Apache. URL: https://github.com/google/benchmark, VCS: https://github.com/google/benchmark. cit. on p. 117

[Gen+08] Luigi Genovese, Alexey Neelov, Stefan Goedecker, et al. "Daubechies wavelets as a basis set for density functional pseudopotential calculations". In: *Journal of Chemical Physics* 129 (2008). cit. on p. 78

[GL08] Bettina Grün and Friedrich Leisch. "FlexMix Version 2: Finite Mixtures with Concomitant Variables and Varying and Constant Parameters". In: *Journal of Statistical Software* 28.4 (2008), pp. 1–35. DOI: 10.18637/jss.v028.i04. URL: https://www.jstatsoft.org/v28/i04/. cit. on p. 47

[Gra+16] Thomas Grass, César Allande, Adrià Armejach, et al. "MUSA: A Multi-level Simulation Approach for Next-generation HPC Machines". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '16. Salt Lake City, Utah: IEEE Press, 2016, 45:1–45:12. DOI: 10.1109/SC.2016.44. cit. on p. 19

[Gri05] David Alan Grier. *When Computers Were Human*. Princeton University Press, 2005. URL: http://www.jstor.org/stable/j.ctt4cgc82. cit. on pp. 1, 2, 3

[Hayai] [SW], *Hayai*, LIC: Apache. URL: https://bruun.co/2012/02/07/easy-cpp-benchmarking, VCS: https://github.com/nickbruun/hayai. cit. on p. 117

[HDF5] [SW], *HDF5*, LIC: BSD. URL: https://www.hdfgroup.org/, VCS: https://github.com/HDFGroup/hdf5. cit. on p. 127

[Hei+17] Franz C. Heinrich, Tom Cornebize, Augustin Degomme, et al. "Predicting the Energy Consumption of MPI Applications at Scale Using a Single Node". In: *Proc. of the 19th IEEE Cluster Conference*. 2017. URL: https://hal.inria.fr/hal-01523608. cit. on pp. 21, 77, 102

[HP18] John L. Hennessy and David A. Patterson. *A new golden age for computer architecture*. 2018. URL: https://youtu.be/3LVeEjsn8Ts. cit. on pp. 5, 6

[HP19] John L. Hennessy and David A. Patterson. "A new golden age for computer architecture". In: *Communications of the ACM* 62.2 (Jan. 2019), pp. 48–60. DOI: 10.1145/3282307. cit. on pp. 5, 6

[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. DOI: 10.1145/1851476.1851564. cit. on p. 17

[IBM21] IBM. *IBM 608 calculator*. 2021. URL: https://www.ibm.com/ibm/history/exhibits/vintage/vintage_4506VV2214.html (visited on Mar. 9, 2021). cit. on pp. 3, 5

[Imb+13] Matthieu Imbert, Laurent Pouilloux, Jonathan Rouzaud-Cornabas, Adrien Lebre, and Takahiro Hirofuchi. "Using the EXECO Toolkit to Perform Automatic and Reproducible Cloud Experiments". In: *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*. IEEE, Dec. 2013. DOI: 10.1109/cloudcom.2013.119. cit. on p. 82

[Ina+15] Y. Inadomi, T. Patki, K. Inoue, et al. "Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing". In: *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2015, pp. 1–12. DOI: 10.1145/2807591.2807638. cit. on p. 73

[Ioa05] John P. A. Ioannidis. "Why Most Published Research Findings Are False". In: *PLoS Medicine* 2.8 (Aug. 2005), e124. DOI: 10.1371/journal.pmed.0020124. cit. on p. 8

[IPYNB] [SW], *Jupyter*, LIC: BSD. URL: https://jupyter.org/. cit. on p. A1

[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), pp. 57–73. DOI: 10.4018/jdst.2010040104. cit. on p. 17

[KQ20] Max Kuhn and Ross Quinlan. *Cubist: Rule- And Instance-Based Regression Modeling*. R package version 0.2.3. 2020. URL: https://CRAN.R-project.org/package=Cubist. cit. on pp. 47, 55

[KR89] R M Karp and V Ramachandran. "A survey of parallel algorithms for shared-memory machines". In: (Jan. 1989). cit. on p. 16

[LD12] Piotr Luszczek and Jack Dongarra. "Reducing the Time to Tune Parallel Dense Linear Algebra Routines with Partial Execution and Performance Modeling". In: *Parallel Processing and Applied Mathematics*. Springer Berlin Heidelberg, 2012, pp. 730–739. DOI: 10.1007/978-3-642-31464-3_74. cit. on p. 16

[Len20] Dan Lenski. *TOP500 data*. 2020. URL: https://github.com/dlenski/top500/raw/b6c4ddf1777447479757b5bda86ae7228227e331/TOP500_history.csv (visited on Nov. 17, 2020). cit. on p. 7

[Leó+16] E. A. León, I. Karlin, A. Bhatele, et al. “Characterizing Parallel Scientific Applications on Commodity Clusters: An Empirical Study of a Tapered Fat-Tree”. In: *SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2016, pp. 909–920. DOI: 10.1109/SC.2016.77. cit. on p. 75

[LL19] Geoff Langdale and Daniel Lemire. “Parsing Gigabytes of JSON per Second”. In: *CoRR* abs/1902.08318 (2019). arXiv: 1902.08318. URL: http://arxiv.org/abs/1902.08318. cit. on p. 116

[Low+20] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, et al. *The gem5 Simulator: Version 20.0+*. 2020. arXiv: 2007.03152 [cs.AR]. cit. on p. 19

[LV20] Arnaud Legrand and Jean-Marc Vincent. *Scientific Methodology and Performance Evaluation for Computer Scientists*. 2020. URL: https://github.com/alegrand/SMPE (visited on Feb. 27, 2021). cit. on p. A1

[Mal+04] D. Malerba, F. Esposito, M. Ceci, and A. Appice. “Top-down induction of model trees with regression and splitting nodes”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.5 (May 2004), pp. 612–625. DOI: 10.1109/tpami.2004.1273937. cit. on p. 51

[Moo65] Gordon E. Moore. “Cramming more components onto integrated circuits”. In: *Electronics* 38.8 (Apr. 1965). URL: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/05/moores-law-electronics.pdf. cit. on p. 4

[Moo75] Gordon E. Moore. “Progress In Digital Integrated Electronics”. In: *Internaional Electron Devices Meeting* (1975). URL: http://www.eng.auburn.edu/~agrawvd/COURSE/E7770_Spr07/READ/Gordon_Moore_1975_Speech.pdf. cit. on p. 4

[MTE] Ministère de la transition écologique. *L’empreinte carbone des Français reste stable*. 2020. URL: https://www.statistiques.developpement-durable.gouv.fr/lempreinte-carbone-des-francais-reste-stable (visited on Feb. 27, 2021). cit. on p. A7

[Mub+16] 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* (2016). cit. on p. 17

[Myt+09] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. “Producing wrong data without doing anything obviously wrong!” In: *ACM SIGPLAN Notices* 44.3 (Feb. 2009), pp. 265–276. DOI: 10.1145/1508284.1508275. cit. on p. 8

[NIH] National Human Genome Research Institute. *The Cost of Sequencing a Human Genome*. 2020. URL: https://www.genome.gov/about-genomics/fact-sheets/Sequencing-Human-Genome-cost (visited on Mar. 14, 2021). cit. on p. 6

[Nonius] [SW], *Nonius*, LIC: Apache. URL: https://nonius.io/, VCS: https://github.com/libnonius/nonius. cit. on p. 117

[Nus17] Lucas Nussbaum. “Towards Trustworthy Testbeds thanks to Throughout Testing”. In: *REPPAR - 4th International Workshop on Reproducibility in Parallel Computing (with IPDPS'2017)*. Orlando, United States, June 2017, p. 9. URL: https://hal.inria.fr/hal-01538682. cit. on p. 120

[Oli+17] Augusto Born De Oliveira, Sebastian Fischmeister, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. “Perphecy: Performance Regression Test Selection Made Simple but Effective”. In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Mar. 2017. DOI: 10.1109/icst.2017.17. cit. on p. 118

[OpenBLAS] [SW], *OpenBLAS*, LIC: BSD. URL: http://www.openblas.net/, VCS: https://github.com/xianyi/OpenBLAs. cit. on pp. 88, 116

[OpenMPI] [SW], *OpenMPI*, LIC: BSD. URL: https://www.open-mpi.org/, VCS: https://github.com/open-mpi/ompi. cit. on pp. 88, 116

[ORG] [SW], *Org-Mode*, LIC: GPL 3.0. URL: https://orgmode.org/fr/index.html. cit. on p. A1

[Pap+19] Alessandro Vittorio Papadopoulos, Alexandru Iosup, Laurens Versluis, et al. “Methodological Principles for Reproducible Performance Evaluation in Cloud Computing”. In: *IEEE Transactions on Software Engineering* (2019), pp. 1–1. DOI: 10.1109/tse.2019.2927908. cit. on p. 8

[Pet+] [SW] Antoine Petitet, Clint Whaley, Jack Dongarra, Andy Cleary, and Piotr Luszczek, *HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers*, LIC: BSD. URL: http://www.netlib.org/benchmark/hpl. cit. on pp. 23, 119

[Pet+21] [SW] Antoine Petitet, Clint Whaley, Jack Dongarra, et al., *HPL (modified for Simgrid/SMPI)*, Mar. 2021. LIC: BSD. DOI: 10.5281/zenodo.4628245, VCS: https://github.com/Ezibenroc/hpl/. cit. on p. A5

[PKP03] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. “The Case of the Missing Supercomputer Performance”. In: *Proceedings of the 2003 ACM/IEEE conference on Supercomputing - SC '03*. ACM Press, 2003. DOI: 10.1145/1048935.1050204. cit. on pp. 7, 95

[PLH20] Christophe Pouzat, Arnaud Legrand, and Konrad Hinsen. *Recherche reproductible : principes méthodologiques pour une science transparente*. 2020. URL: https://www.fun-mooc.fr/courses/course-v1:inria+41016+self-paced/about (visited on Feb. 27, 2021). cit. on p. A1

[Sch+19] Robert Schöne, Thomas Ilsche, Mario Bielert, Andreas Gocht, and Daniel Hackenberg. “Energy Efficiency Features of the Intel Skylake-SP Processor and Their Impact on Performance”. In: *CoRR* abs/1905.12468 (2019). DOI: 10.1109/HPCS48598.2019.9188239. arXiv: 1905.12468. URL: http://arxiv.org/abs/1905.12468. cit. on p. 110

[Sch20] Leonid Schneider. *Science misconduct*. July 2020. URL: https://forbetterscience.com/2020/07/07/science-misconduct/ (visited on Mar. 25, 2021). cit. on p. 8

[SG19] The Simgrid Team. *Continuous integration of ECP Proxy Apps, CORAL and Trinity-Nersc benchmarks with SimGrid/SMPI*. 2019. URL: https://framagit.org/simgrid/SMPI-proxy-apps. cit. on p. 78

[SG21] [SW] The Simgrid Team, *Platform calibration for Simgrid/SMPI*, Mar. 2021. LIC: MIT. DOI: 10.5281/zenodo.4628246, VCS: https://framagit.org/simgrid/platform-calibration. cit. on pp. 88, A5

[Sim18] [SW] Matthieu Simonin, *EnOSlib*, 2018. LIC: GPL 3.0. URL: https://msimonin.gitlabpages.inria.fr/enoslib/, VCS: https://gitlab.inria.fr/discovery/enoslib. cit. on p. 85

[Sin+07] Karan Singh, Engin İpek, Sally A. McKee, et al. "Predicting parallel application performance via machine learning approaches". In: *Concurrency and Computation: Practice and Experience* 19.17 (2007), pp. 2219–2235. DOI: 10.1002/cpe.1171. cit. on p. 17

[SK05] D. Skinner and W. Kramer. "Understanding the causes of performance variability in HPC workloads". In: *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.* IEEE, 2005. DOI: 10.1109/iiswc.2005.1526010. cit. on p. 7

[SQLite] [SW], *SQLite*, LIC: Public domain. URL: https://www.sqlite.org/, VCS: https://github.com/sqlite/sqlite. cit. on p. 127

[Sta+15] Luka Stanisic, Samuel Thibault, Arnaud Legrand, Brice Videau, and Jean-François Méhaut. "Faithful Performance Prediction of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures". In: *Concurrency and Computation: Practice and Experience* (May 2015), p. 16. DOI: 10.1002/cpe.3555. URL: https://hal.inria.fr/hal-01147997. cit. on p. 118

[Stress-ng] [SW], *Stress-ng*, LIC: GPL 2.0. URL: https://kernel.ubuntu.com/~cking/stress-ng/, VCS: https://github.com/ColinIanKing/stress-ng. cit. on pp. 120, 143

[SV12] K. L. Spafford and J. S. Vetter. "Aspen: A domain specific language for performance modeling". In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Nov. 2012, pp. 1–11. DOI: 10.1109/SC.2012.20. cit. on p. 16

[Taf+19] P. Taffet, S. Rao, E. León, and I. Karlin. "Testing the Limits of Tapered Fat Tree Networks". In: *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 2019, pp. 47–52. DOI: 10.1109/PMBS49563.2019.00011. cit. on p. 75

[top500] *TOP500 Website*. URL: https://www.top500.org/ (visited on Sept. 7, 2020). cit. on pp. 6, 7

[TSL20] Samuel Thibault, Luka Stanisic, and Arnaud Legrand. "Faithful Performance Prediction of a Dynamic Task-based Runtime System, an Opportunity for Task Graph Scheduling". In: *SIAM PP 2020 - SIAM Conference on Parallel Processing for Scientific Computing*. Seattle, United States, Feb. 2020. URL: https://hal.inria.fr/hal-02943753. cit. on p. 119

[Tun+17] Ozan Tuncer, Emre Ates, Yijia Zhang, et al. “Diagnosing Performance Variations in HPC Applications Using Machine Learning”. In: *Lecture Notes in Computer Science*. Springer International Publishing, 2017, pp. 355–373. DOI: 10.1007/978-3-319-58667-0_19. cit. on p. 7

[Val90] Leslie G. Valiant. “A Bridging Model for Parallel Computation”. In: *Commun. ACM* 33.8 (Aug. 1990), pp. 103–111. DOI: 10.1145/79173.79181. cit. on p. 16

[Vel+13] 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), p. 23. DOI: 10.1145/2517448. URL: https://hal.inria.fr/hal-00872476. cit. on p. 20

[WHOa] World Health Organization. *Climate change and health*. 2018. URL: https://www.who.int/news-room/fact-sheets/detail/climate-change-and-health (visited on Feb. 27, 2021). cit. on p. A7

[WHOb] World Health Organization. *Climate change*. 2021. URL: https://www.who.int/heli/risks/climate/climatechange/en/ (visited on Feb. 27, 2021). cit. on p. A7

[Wik21a] Wikipedia. *Microprocessor Chronology*. 2021. URL: https://en.wikipedia.org/wiki/Microprocessor_chronology (visited on Feb. 7, 2021). cit. on p. 4

[Wik21b] Wikipedia. *Transistor count*. 2021. URL: https://en.wikipedia.org/wiki/Transistor_count (visited on Feb. 7, 2021). cit. on p. 4

[WM11] Xing Wu and Frank Mueller. “ScalaExtrap: Trace-Based Communication Extrapolation for SPMD Programs”. In: *Proc. of the 16th ACM Symp. on Principles and Practice of Parallel Programming*. 2011, pp. 113–122. cit. on p. 17

[Xu+20] G. Xu, H. Ibeid, X. Jiang, V. Svilan, and Z. Bian. “Simulation-Based Performance Prediction of HPC Applications: A Case Study of HPL”. In: *2020 IEEE/ACM International Workshop on HPC User Support Tools (HUST) and Workshop on Programming and Performance Visualization Tools (ProTools)*. 2020, pp. 81–88. DOI: 10.1109/HUSTProtools51951.2020.00016. cit. on p. 19

[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. 17

Abstract

The scientific community relies more and more on computations, notably for numerical simulation and data processing. While many scientific advances were made possible by the technological progress of computers, additional performance gains are still required for larger scale projects.

The race for performance is addressed with a growing hardware and software complexity, which in turn increases the performance variability. This can make the experimental study of performance extremely challenging, raising concerns of reproducibility of the experiments, akin to the problems already faced by natural sciences.

Our contributions are twofold. First, we present a methodology for predicting the performance of parallel non-trivial applications through simulation. We describe several models for communications and computations, with an increasing complexity. We compare these models through an extensive validation by matching our predictions with real experiments. This validation shows that modeling the spatial and temporal variability of the platform is essential for faithful predictions. As a consequence, predictions require careful sensibility analysis accounting for the uncertainty on the resource models, which we illustrate through several case studies. Second, we present the lessons learned while making the numerous experiments required in the first part and how we improved our methodology. We show that measurements can suffer from multiple experimental biases and we explain how some of these biases can be overcome. We also present how we implemented systematic performance non-regression testing, which allowed us to detect many significant changes of the platform throughout this thesis.

Résumé

La communauté scientifique s'appuie de plus en plus sur les calculs, notamment pour la simulation numérique et le traitement des données. Alors que de nombreuses avancées scientifiques ont été rendues possibles par les progrès technologiques des ordinateurs, des gains de performance supplémentaires sont encore nécessaires pour les projets à plus grande échelle.

La course à la performance est abordée avec une complexité matérielle et logicielle croissante, qui à son tour augmente la variabilité des performances. Cela peut rendre l'étude expérimentale de la performance extrêmement difficile, ce qui soulève des préoccupations quant à la reproductibilité des expériences, de manière similaire aux problèmes déjà rencontrés par les sciences naturelles.

Nos contributions sont doubles. Tout d'abord, nous présentons une méthodologie pour prédire les performances d'applications parallèles non triviales par la simulation. Nous décrivons plusieurs modèles de communications et de calculs, avec une complexité croissante. Nous comparons ces modèles via une validation approfondie en faisant correspondre nos prédictions avec des expériences réelles. Cette validation montre que la modélisation de la variabilité spatiale et temporelle de la plateforme est essentielle pour les prédictions. En conséquence, les prévisions requièrent une analyse de sensibilité minutieuse tenant compte de l'incertitude sur les modèles de ressources, que nous illustrons à travers plusieurs études de cas. Par la suite, nous présentons les leçons apprises lors des nombreuses expériences menées dans la première partie et comment nous avons amélioré notre méthodologie. Nous montrons que les mesures peuvent souffrir de multiples biais expérimentaux et nous expliquons comment certains de ces biais peuvent être surmontés. Nous présentons également comment nous avons mis en œuvre des tests systématiques de non-régression des performances, qui nous ont permis de détecter de nombreux changements significatifs de la plateforme tout au long de cette thèse.