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**

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

Le Titre de la Thèse
The English Title

Thèse soutenue publiquement le **1er janvier 1970**,
devant le jury composé de :

I dedicate this thesis to my grumpy cat.

Elle est où la poulette ?

— **Kadoc De Vannes**

Remerciements

(Acknowledgments)

I would like to thank everyone, except from Dobby the free elf.

Merci public !

Abstract / Résumé

Abstract

The English abstract.

Résumé

Le résumé en français.

Contents

Introduction

Make Zenodo repositories for everything relevant and add the links in the thesis. We should have at least: pytree, peanut, cashew, ratatouille, org_attach (?), calibration_analysis, journal_extract, g5k_test (*2), hpl, platform-calibration

Remove all the vspace and hspace. Most of them come from the puslished articles, they are not needed here.

Scientific Computing: a story

First computers, from carbon to silicon

Science has always been tightly associated to computations, hence this 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 in 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. They were succesful 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 of three levels:

- The first class of workers, a handful of renowed 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 commitee*, 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 substractions. 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 wig makers that lost their jobs with the Revolution, they did not know any mathematics besides the addition and substraction. 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 commercialy 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. Initialy 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.

Add some text on the human computers at NASA, the first IBM computer.

Exponential growth

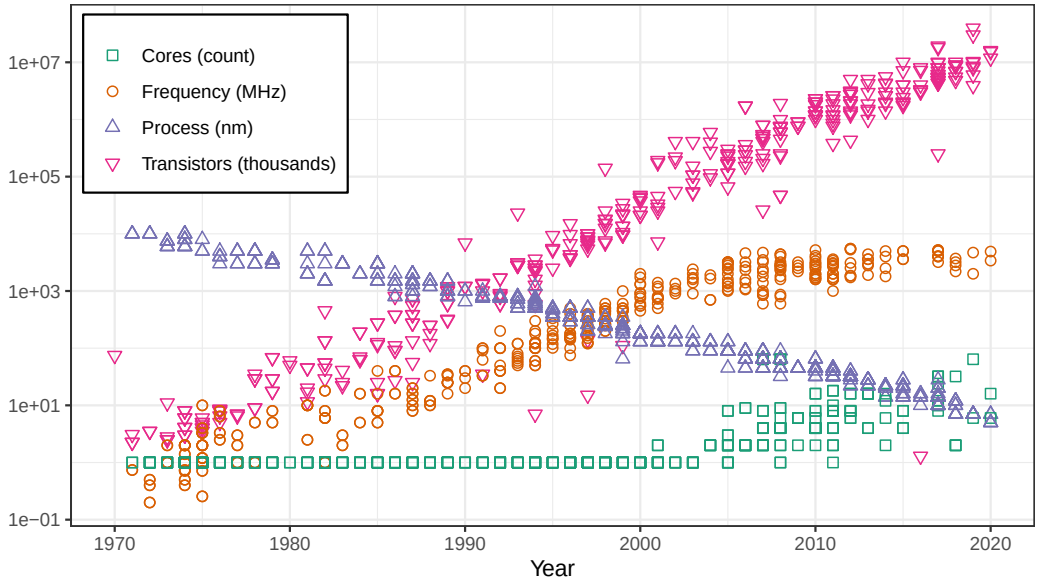


Figure 0.1.: Evolution of the processor characteristics between 1971 and 2020. Plot inspired from the work of Pedro Bruel, generated with data from wikipedia [Wik21a; Wik21b].

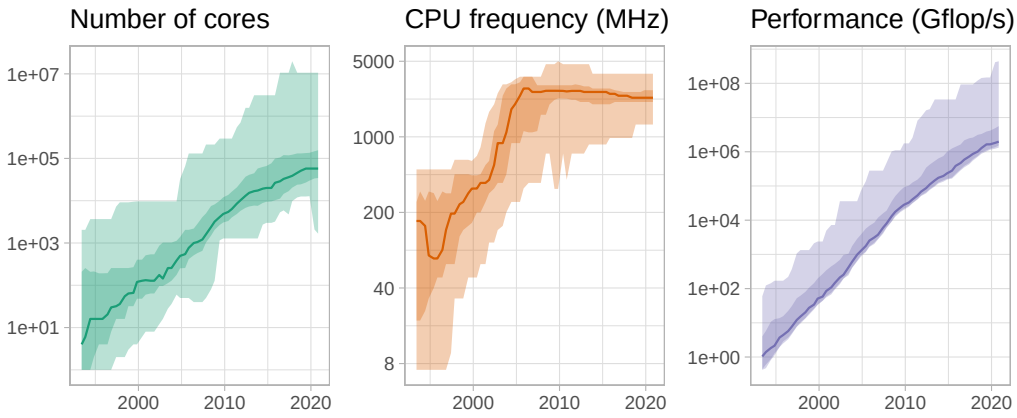


Figure 0.2.: Evolution of the Top500 [top500] supercomputers between 1993 and 2020. The line denotes the median, the inner ribbon contains the $[10\,\%, 90\,\%]$ interval, the outer ribbon contains all the values.
Data compiled by Dan Lenski [Len20] and plotted by ourselves.

Increased complexity

Some horror stories[PKP03]...

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[CL21]. The content of this part is therefore a near-verbatim copy of these articles. This work also directly follows my master thesis[Cor17] whose main contribution is summarized in chapter 3 for the sake of completeness.

1 Related work

1.1 Performance prediction

As discussed in Chapter , researchers and engineers often need to make predictions about the performance of a given parallel application on a given platform. The reasons are diverse, it could be 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 fulfill several of these criterions:

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 creterion denotes whether the considered approach can predict the performance of the application for an unforeseen 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 unforeseen parameter combination.

No full-scale real run Some prediction methods will require at least one run of the application at the desired scale in terms of number of nodes. Note that this does 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 an 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 will only give an approximate trend, some others will be much more accurate, as close as a few percents, even in the presence of perturbations like network contention.

A first approach for estimating the performance of MPI applications is statistical modeling of the application as a whole[LD12]. By running the application several times with small and medium problem (of a few iterations of large problem sizes) and using simple linear regressions, it is possible to predict its makespan 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 makespan with an unforeseen problem size. Furthermore, this kind of approaches inherently prevent 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 details and flexibility missing to such black-box modeling approach. 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] 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 is executed (emulated) on top of a simulator that is responsible to determine when each process is run. 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 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 seemed quite solid but the proposed developments would a priori also be possible with SST. Note that the emulation described in chapter 3 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 whereas 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 MUSA[Gra+16].

The differences between these prediction approaches are summarized in Table 1.1.

Table 1.1.: Summary of the different prediction approaches

Approach	Extrapolation on the problem size	Extrapolation on the configuration	No full-scale real run	Hypothetical platform	Efficiency	Accuracy
Big-O analysis	✓	✓	✓	✓	✓✓	✗✗
[LD12]	✓	✗	✓	✗	✓	✓
[Sin+07]	✗	✓	✗	✗	✓	✓
Ideal off-line sim.	✗	✗	✗	✓	✓	✓
BigSim[ZKK04]						
Dimemas[Bad+03]						
CODES[Mub+16]						
Ideal on-line sim.	✓	✓	✓	✓	✓	✓
xSim[Eng14]						
SST[Jan+10]						
Simgrid[Cas+14]	✓	✓	✓	✓	✓	✓

1.2 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. Such a 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]. The complex network optimizations done in real MPI implementations need to be considered when predicting the performance of MPI applications. For instance, the "eager" and "rendez-vous" protocols are selected based on the message size, with each protocol having its own synchronization semantics, which strongly impact performance. SMPI supports different performance modes through a generalization of the LogGPS model. Another difficult issue is to model network topologies and contention. 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. Special attention to the modeling of collective communication algorithms has also been paid in SMPI, but this is of little significance in this article as HPL ships with its own implementation of collective operations.

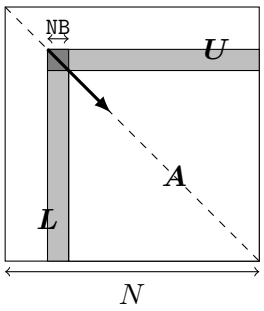
SMPI maps every MPI rank of the application onto a lightweight simulation thread. These threads are then run one at a time, i.e. in mutual exclusion. Every time a thread enters an MPI call, SMPI takes control and the time that was spent computing (isolated from the other threads) since the previous MPI 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.

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, I validate this approach at a much larger scale with HPL, whose emulation 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 tremendous memory consumption and amount of memory accesses need to be drastically reduced.

2 High Performance Linpack

2.1 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 2.1.: Overview of High Performance Linpack

In this work, we use the freely-available reference-implementation of HPL[Pet+], which relies on MPI. 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 Figure2.1. 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, respectively.
- `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.

- `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 6 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`. 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 thereby start working on the panel as soon as possible. The `ring` and `2-ring` versions each broadcast along the corresponding virtual topologies while the `long` version is a *spread and roll* algorithm where messages are chopped into `Q` pieces. This generally leads to better bandwidth exploitation. 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. 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.
- `DEPTH` controls how many iterations of the outer loop can overlap with each other.

The sequential complexity of this 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 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.

2.2 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 Table2.1). In June 2013, the Stampede supercomputer at TACC was ranked 6th in the TOP500 by achieving $5168.1\,\mathrm{TFlop\,s^{-1}}$. In November 2017, the Theta supercomputer at ANL was ranked 18th with a performance of $5884.6\,\mathrm{TFlop\,s^{-1}}$ 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 2.1.: Typical runs of HPL

	Stampede@TACC	Theta@ANL	Dahu@G5K
`Rpeak`	$8520.1\,\mathrm{TFlop\,s^{-1}}$	$9627.2\,\mathrm{TFlop\,s^{-1}}$	$62.26\,\mathrm{TFlop\,s^{-1}}$
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`	$5168.1\,\mathrm{TFlop\,s^{-1}}$	$5884.6\,\mathrm{TFlop\,s^{-1}}$	$24.55\,\mathrm{TFlop\,s^{-1}}$
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 was to simulate systems at the scale of Stampede. Given the scale of this scenario (3,785 steps on 6,006 nodes in two hours), detailed simulations quickly become intractable without significant effort.

3 Emulating HPL at large scale

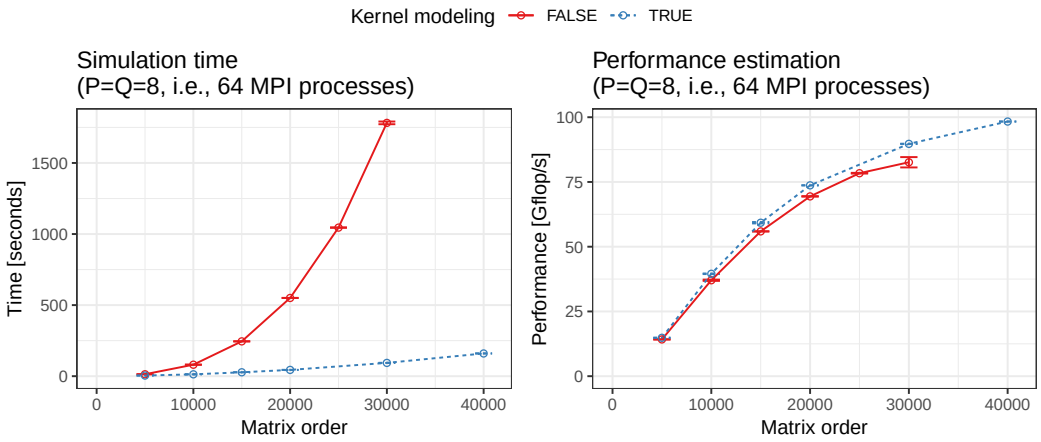
In this chapter, we present the changes to SimGrid and HPL that were required for a scalable simulation. The experiments were done using a single core from nodes of the Nova cluster provided by the Grid'5000 testbed[Bal+13] (32 GB of RAM, two 8-core Intel Xeon E5-2620 v4 CPUs, Debian Stretch OS (Linux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

3.1 Speeding Up the Emulation

3.1.1 Compute Kernel Modeling

```
#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 3.1.: Replacing the calls to computationally expensive functions by a model allows to emulate HPL at a larger scale.

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 simulation 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 behavior of the application and not the correctness of computation results are of concern. Figure3.1(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 calibration. The estimated time of the kernel is calculated based on the given parameters and passed on to `smpi_execute_benched` that advances the clock of the executing rank by this estimate. The effect on the simulation time for a small

scenario is depicted in Figure3.1(b). This modification speeds up the simulation by orders of magnitude. The precision 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 performance.

In addition to the main compute kernels, a profiling of the code allowed to identify seven other BLAS functions as computationally expensive enough to justify a 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 measured by HPL; however, because of the removal of the `dgemm` and `dtrsm` computations, they all operate on bogus data and hence also produce bogus 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 large scenarios.

3.1.2 Specific Adjustments

HPL uses pseudo-randomly generated matrices that are setup every time HPL is executed. 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 was has been set to a random (but controlled) value to make the simulation unbiased (but fully deterministic).

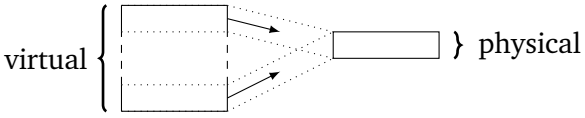
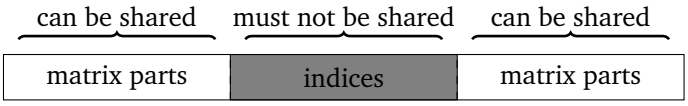
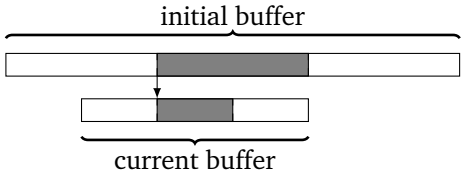


Figure 3.2.: SMPI shared malloc mechanism: large area of virtual memory are mapped onto the same physical pages.



(a) Structure of the panel in HPL.



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

Figure 3.3.: Panel structure and allocation strategy.

3.2 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. This sub-matrix typically occupies a few hundred MB per process. Unfortunately, when emulating 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 Figure3.2): 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 `A`, a more careful strategy needs to be used for the `panel`, which is an intricate data structure with 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` (see Figure3.3(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 64 MPI processes, memory consumption decreases with this approach from about $13.5\,\mathrm{GB}$ to less than $40\,\mathrm{MB}$.

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 an other (see Figure3.3(b)). Even for a very small matrix of order $40,000$ and 64 MPI processes, the simulation time decreases from $20.5\,\mathrm{sec}$ to $16.5\,\mathrm{sec}$. 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 $0.488\,\mathrm{GB}$.

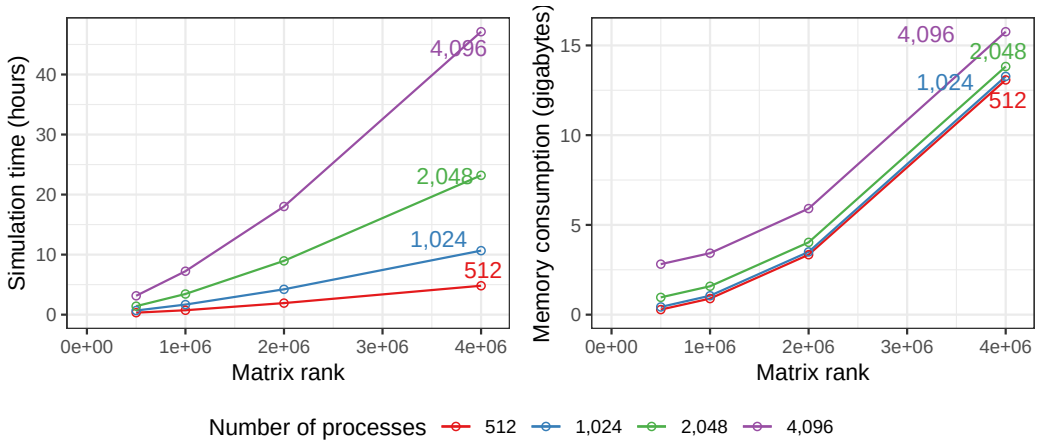


Figure 3.4.: Time complexity and memory consumption are linear in the number of processes but remain mildly quadratic with matrix rank.

3.3 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 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 profoundly.

To evaluate the efficiency of our proposal, we conduct a first evaluation on a non-existing but Stampede resembling 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 makespan and memory is illustrated in Figure3.4. 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.

4 Modeling HPL kernels and communications

As explained in chapter 3, 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 wildly vary depending on the range of the message size as MPI switches from one protocol to another whenever needed. In this chapter we first introduce some notations to describe the complexity of the models we have investigated. We then briefly compare the prediction of these models with individual measurements of both computations and communications to illustrate the importance of the model complexity.

4.1 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 $\mathcal{M}-0$: For simple and stable 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 $\mathcal{M}-1$: 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 $\mathcal{M}-2$: 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 $\mathcal{M}_H$.
- 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 piece-wise 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 $\mathcal{M}'$.

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.

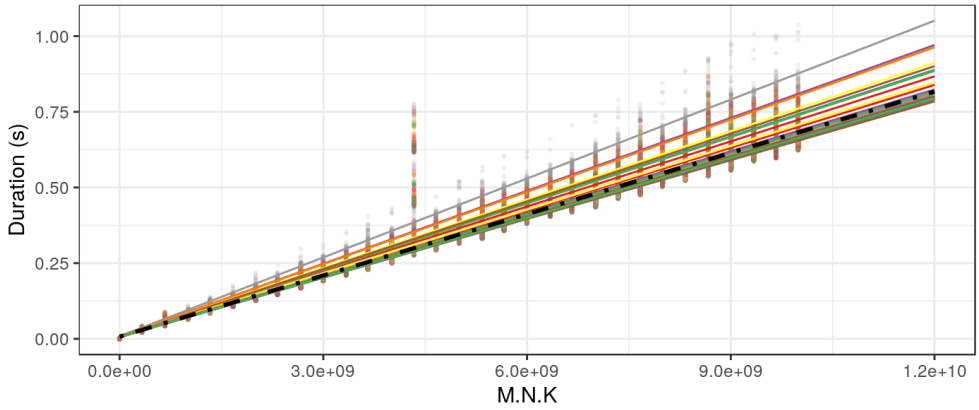
The residual (temporal) variability may be an important phenomenon to account for, as "system noise" is known to be detrimental to the overall performance of parallel applications like HPL. We thus consider different modeling options for this temporal variability:

- Noise option $\mathcal{N}$–0 (no noise): This is the simplest option. It consists in injecting the value predicted by the model
- Noise option $\mathcal{N}-1$ (homoscedastic): The simplest probability family to model variability is the normal distribution, hence $T \sim \mathcal{M}(M,N,K) + N(0,\sigma^2)$, where σ^2 is the sample variance of the model residuals.
- Noise option $\mathcal{N}-2$ (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.

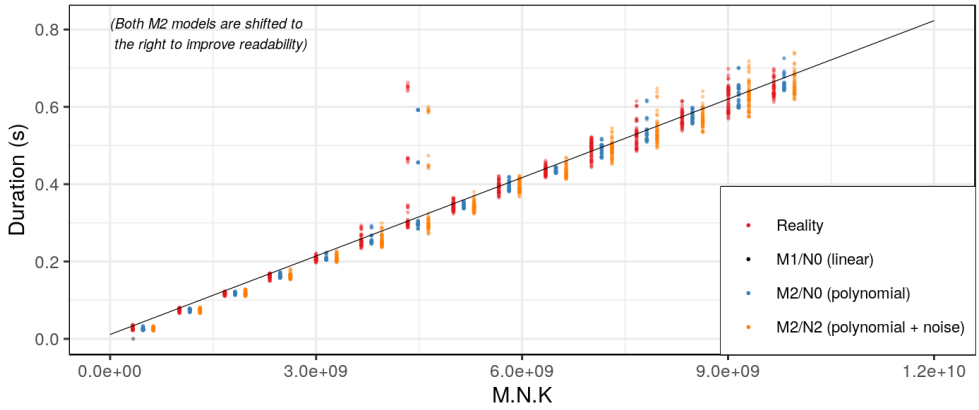
Finally, even the sophisticated normal distribution from $\mathcal{N}-2$ may be too simple to describe the noise observed on real platforms where it may be common for a same parameter set to have a few operations being one order of magnitude slower than all the other ones. In this case, a reasonable option consists of 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 $\mathcal{N}'$. Likewise, the per-host estimations are denoted by $\mathcal{N}_H$.

4.2 Modeling the CPU (i.e. dgemm function)

4.2.1 Different linear models



(a) dgemm heterogeneity



(b) dgemm model

Figure 4.1.: 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: $\mathcal{M}-\{1,2\}$ $\mathcal{N}-\{0,1,2\}$ and $\mathcal{M}_H-\{1,2\}$ $\mathcal{N}_H-\{0,1,2\}$. The $\mathcal{M}'$ and $\mathcal{N}'$ families were not investigated as nothing in our observations called for such complexity on classical multi-core machines. Figure4.1 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 sizes over all the cores of each node (to reproduce experimental conditions similar to the one of HPL). The first observation (Figure4.1(a)) is that a few nodes exhibit quite a different behavior

(each color and each regression line under model $\mathcal{M}_H - 1$ corresponds to a different cpu, whereas the black dotted line corresponds to model $\mathcal{M} - 1$ 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 HPL. Second, we took care of covering a wide variety of combinations for M, N, and K and it can be observed that $M.N.K$ is not sufficient to describe correctly the performance of `dgemm`. 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. Figure4.1(b) depicts the performance (red dots) of a given node as well as the prediction using a simple linear model ($\mathcal{M}_H - 1$, black line), a full polynomial model ($\mathcal{M}_H - 2$, blue dots) and a full polynomial model with heteroscedastic noise ($\mathcal{M}_H - 2\, \mathcal{N}_H - 2$, 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.

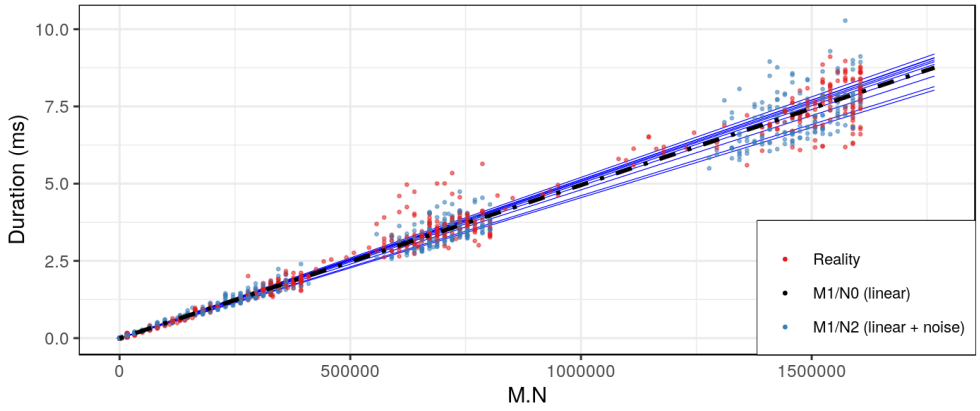


Figure 4.2.: Illustrating the realism of modeling for `HPL_dlatcpy` function

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 extremely small 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 Figure4.2. 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 $\mathcal{N}-2$ model is clearly required. The blue dots in Figure4.2 represent the outcome of a $\mathcal{M}-1\,\mathcal{N}-2$ model and are hardly distinguishable from the real behavior. Similar results can be obtained with this category of model for all other kernels.

4.2.2 Bayesian modeling and generative models

In Chapter 5, we will show that the model $\mathcal{M}_H-2\,\mathcal{N}_H-2$ is required for the `dgemm` kernel to make reliable performance predictions for HPL. We recall here the meaning of this notation:

$$\begin{aligned}&\text{For each processor } p,\ \mathsf{dgemm}_p(M,N,K) \sim \mathcal{H}(\mu_p,\sigma_p)\\ &\begin{cases}\mu_p &= \alpha_p MNK + \beta_p MN + \gamma_p MK + \delta_p NK + \epsilon_p\\ \sigma_p &= \omega_p MNK + \psi_p MN + \phi_p MK + \tau_p NK + \rho_p\end{cases},\end{aligned} \tag{4.1}$$

where $\mathcal{H}(\mu,\sigma)$ denotes a random variable following a half-normal distribution with parameters μ,σ accounting for the expectation and the standard deviation. The dependency on p allows to account 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 to account 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.

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 a day to another). As illustrated in Chapter 5, accounting for the first two kinds of variability is essential but during our investigation of the simulation validity, which spanned over several months, we also had to deal with the fact that the node performance from a day to another could significantly vary, thereby making our comparisons between a real experiment and the simulation driven by model obtained with 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 6. 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:

$$\forall M,N,K, \mathsf{dgemm}_{p,d}(M,N,K) \sim \mathcal{H}(\alpha_{p,d}MNK + \beta_{p,d},\ \gamma_{p,d}MNK) \tag{4.2}$$

Compared to the model (4.1), 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

$$\mu_{p,d} = (\alpha_{p,d}, \beta_{p,d}, \gamma_{p,d}). \tag{4.3}$$

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

$$\forall d, \mu_{p,d} \sim \mathcal{N}(\mu_p, \Sigma_T) \tag{4.4}$$

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 a 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 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.

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

$$\forall p, \mu_p \sim \mathcal{N}(\mu, \Sigma_S) \tag{4.5}$$

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 4.3.

The relevance of model (4.2) will be discussed in Chapter 5, but the relevance of models (4.4) and (4.5) requires some attention. Figure4.4(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,

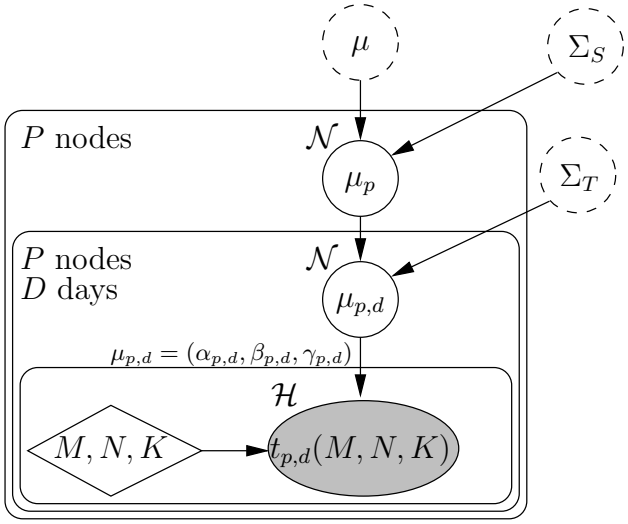
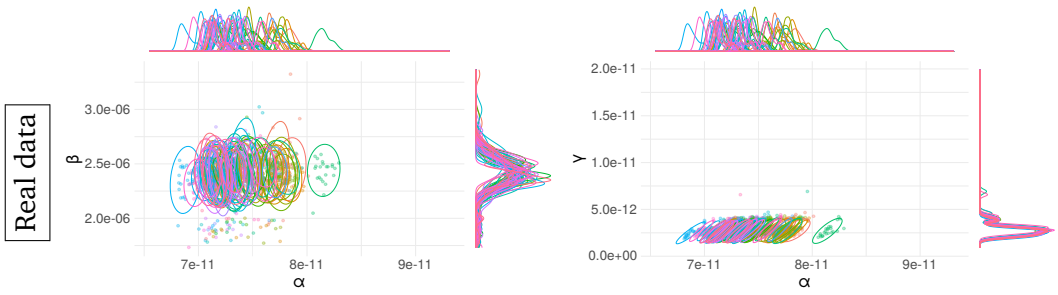


Figure 4.3.: Generative model of kernel duration accounting for the spatial (Σ_S), long-term (Σ_T) and short-term variability ($\gamma_{p,d}$). The shaded node represents observed variables and diamond node represents deterministic variables, while non-shaded nodes represent latent variables. The solid 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 an hypothetical cluster.

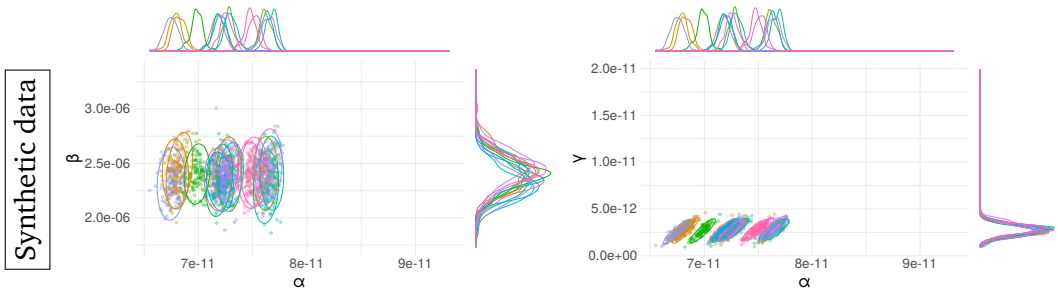
it appeared to be statistically significant (most ellipses are slightly tilted toward North-East), hence a full variance matrix is needed (at least for Σ_T).

It is thus 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 SkewNormal 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 4.3 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 4.4(b) where hypothetical regression parameters for 16 nodes have been generated. Comparing such synthetic data with the original samples from Figures 4.4(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 subtle 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

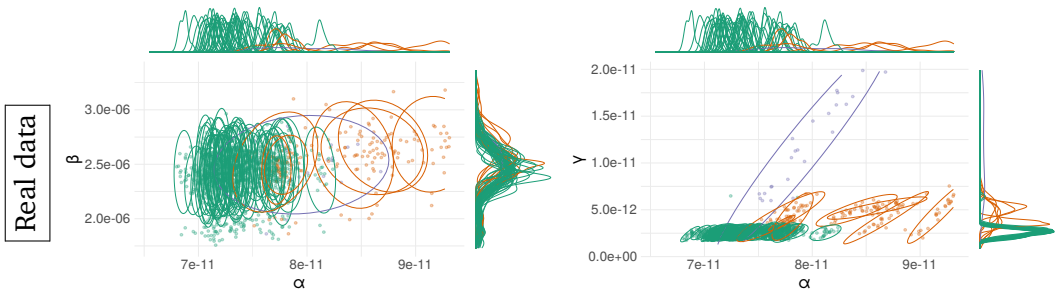


(a) Distribution of α, β, and γ (observations on 2×32 CPUs from November 2019 to February 2020).

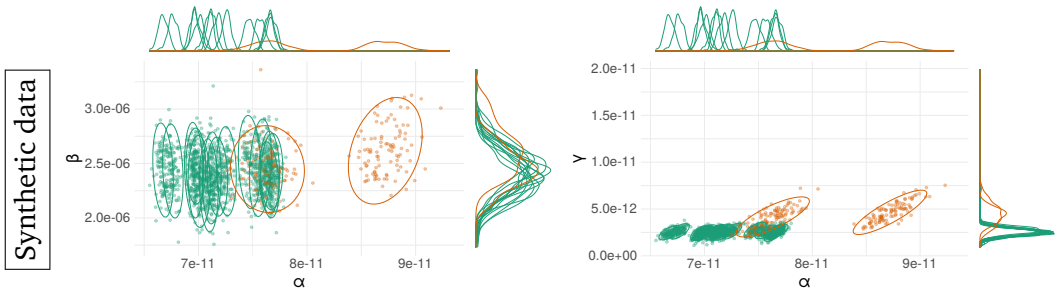


(b) Distribution of α, β, and γ (synthetic data for 16 CPUs).

Figure 4.4.: 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 4.5.: 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.

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 4.5(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 4.3 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 4.5(b) are very similar, although different, 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.

4.2.3 Conclusion

We have described in Section 4.2.1 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 5 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 4.2.2 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 6 for sensibility analysis.

4.3 Modeling the network

4.3.1 Modelisation 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 ($\mathcal{M}'-1$ $\mathcal{N}-0$). Yet, as illustrated in Figure4.6, 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.

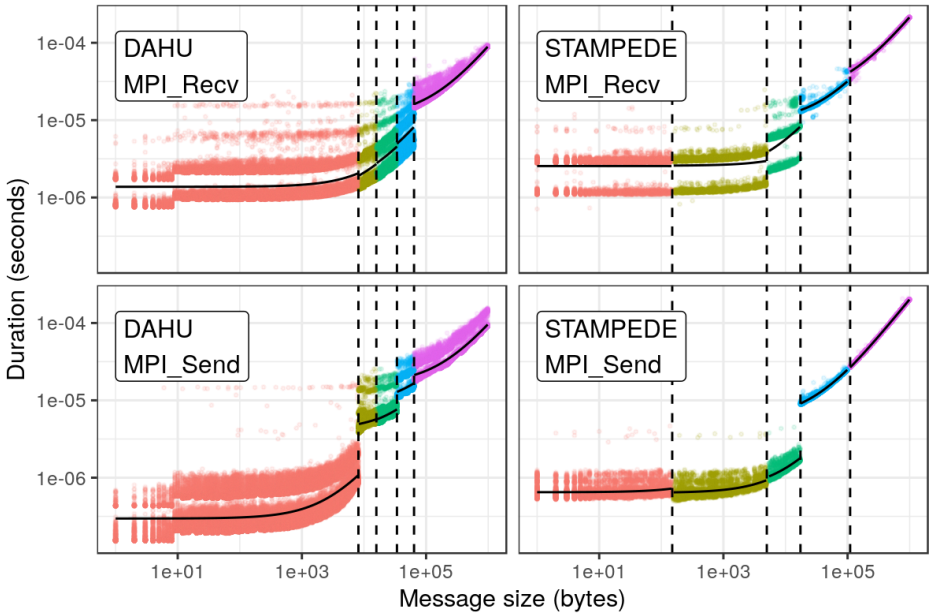


Figure 4.6.: Illustrating piecewise linearity and temporal variability of high-speed communications on two systems.

The ideal model for such observed data would be $\mathcal{M}'-1$ $\mathcal{N}'-1$, 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 bad) performance since they generally get amplified by broadcast and pipelined communication patterns.

Automatically computing the best fit for a complex model like the abovementionned one is way more difficult than for the `dgemm` model proposed in Section 4.3. It should be possible to compute it in one go with a Bayesian framework like STAN,

but it would probably 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, and (2) on each segment, compute the linear regression with a noise mixture.

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 made the choice to implement our own solution, detailed in Section 4.3.2.

2. We tested an existing R package named `flexmix` for computing a linear regression with a Gaussian mixture noise [GL08]. It revealed 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 with a visual inspection and then computing a simple linear regression for each mode.

4.3.2 Learning breakpoints

In this section, we present our solution for automatically computing a piecewise linear regression. This has been implemented as a Python package named `pycewise`[1].

Model

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

$$Y \sim \alpha_i X + \beta_i + \mathcal{N}(0, \gamma_i X + \delta_i) \qquad \text{if } B_i \leq X < B_{i+1} \tag{4.6}$$

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.

[1] `https://github.com/Ezibenroc/pycewise`

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, they could be estimated in a second phase. We will write f_θ the (deterministic) prediction function associated with the solution:

$$f_\theta(X) = \alpha_i X + \beta_i \qquad \text{if } B_i \leq X < B_{i+1} \tag{4.7}$$

Two objective 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 noted $\ell_1 \bullet \ell_2$, hence $\text{BIC}(\ell_1 \bullet \ell_2)$ will denote the BIC of a linear regression without breakpoint ($N = 0$).

Main algorithm

The algorithm is made of 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 simpify the solution. The first step is inspired by the work of Malerba et al. [Mal+04].

In the top-down step (Algorithm 2), we recursively build a tree, where the inner nodes represent a breakpoint and the leaves represent a simple linear regression. 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 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.

Algorithm 2: Top-down step for computing the piecewise linear regression

```
Function topdown(ℓ):
    best_BIC = +∞
    foreach (X, Y) ∈ ℓ do
        split ℓ into ℓ1 and ℓ2 such that ∀X' ∈ ℓ1, X' ≤ X and ∀X' ∈ ℓ2, X' > X
        new_BIC = BIC(ℓ1 ⊕ ℓ2)
        if new_BIC < best_BIC then
            best_BIC = new_BIC
            best_solution = (ℓ1, ℓ2)

    if new_BIC < BIC(ℓ) then
        (ℓ1, ℓ2) = best_solution
        lists1 = topdown(ℓ1)
        lists2 = topdown(ℓ2)
        return lists1 ⊕ lists2
    else
        return ℓ
```


In our implementation, each iteration of the `foreach` loop from Algorithm 2 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 to recompute 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 changed the objective function and had to drop this quality of doing an iteration in constant time, the effect on the computation duration will be discussed.

In the bottom-up step (Algorithm 3), 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 minimized the BIC.

Algorithm 3: 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$(\ell_{j+1}))$
    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:

$$\text{RSS}(\ell, \theta) = \sum_{(X,Y)\in\ell} (Y - f_\theta(X))^2 \tag{4.8}$$

The algorithm is an 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:

$$\text{RSS}_{\text{weight}}(\ell, \theta) = \sum_{(X,Y)\in\ell} (w(X)(Y - f_\theta(X)))^2 \tag{4.9}$$

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.

$$\hat{\alpha}_{\text{weight}} = \frac{\sum_{(X,Y)\in\ell} w(X)(X - \hat{X}_{\text{weight}})(Y - \hat{Y}_{\text{weight}})}{\sum_{(X,Y)\in\ell} w(X)(X - \hat{X}_{\text{weight}})^2} \tag{4.10}$$

$$\hat{\beta}_{\text{weight}} = \hat{Y}_{\text{weight}} - \hat{\alpha}_{\text{weight}}\hat{X}_{\text{weight}} \tag{4.11}$$

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

$$\hat{X}_{\text{weight}} = \frac{\sum_{(X,Y)\in\ell} w(X)X}{\sum_{(X,Y)\in\ell} w(X)} \tag{4.12}$$

$$\hat{Y}_{\text{weight}} = \frac{\sum_{(X,Y)\in\ell} w(X)Y}{\sum_{(X,Y)\in\ell} w(X)} \tag{4.13}$$

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

$$\text{RSS}_{\log}(\ell, \theta) = \sum_{(X,Y)\in\ell} (\log Y - \log f_\theta(X))^2 \tag{4.14}$$

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.

Comparison of the solutions

We compared different approaches for computing a piecewise linear regression. We used a typical dataset for our context that has similar characteristics than the network calibration data (see Figure 4.6): the predictor variable is sampled exponentially and the noise is heteroscedastic.

The different approaches are presented in Figure 4.7. 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 a dashed vertical line.

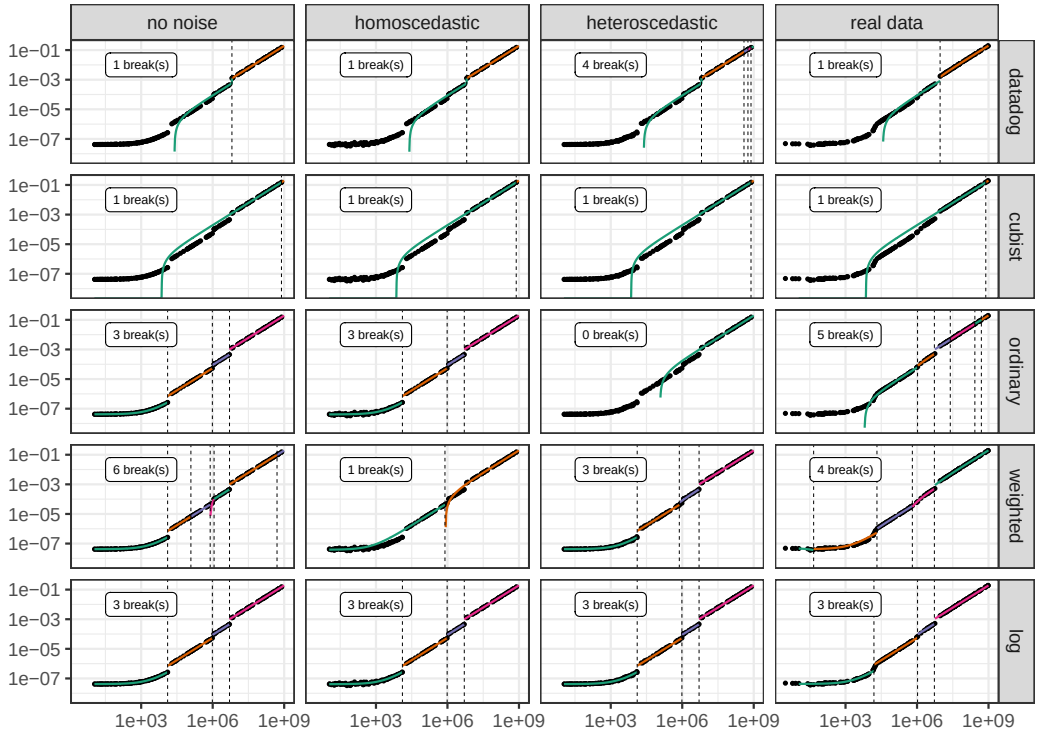


Figure 4.7.: Illustrating different piecewise linear regression approaches on different datasets. Only the logarithmic objective function works well for all datasets.

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 expected 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 4.7, 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 4.8. 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.

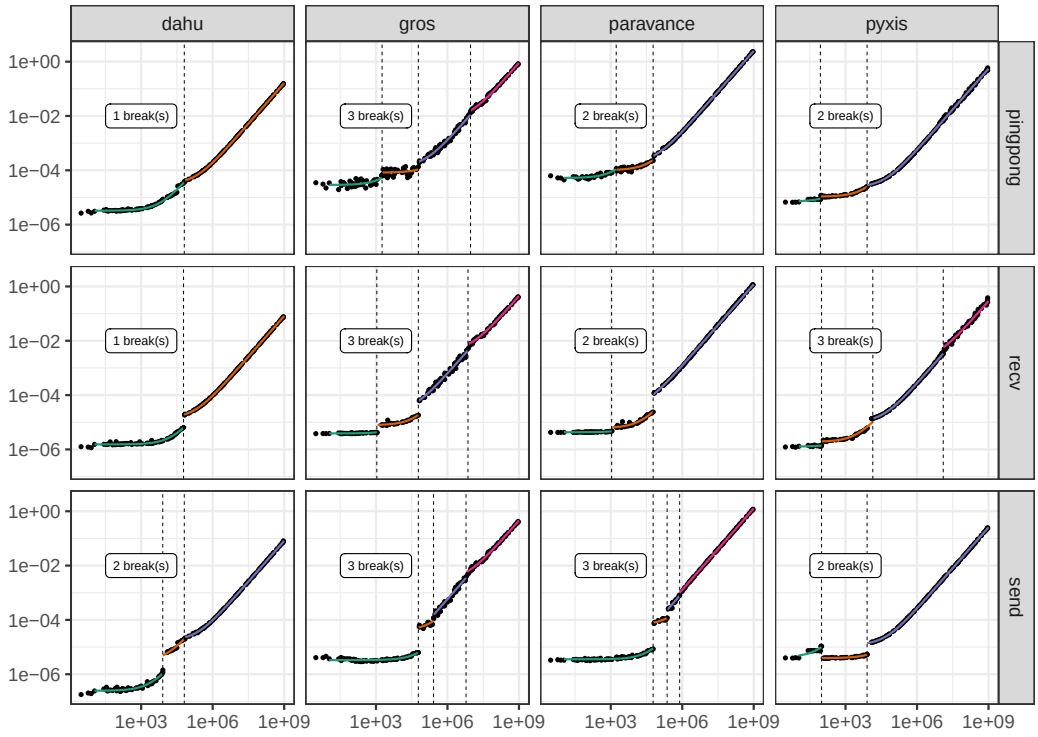


Figure 4.8.: With the logarithmic objective function, pycewise finds acceptable fits for the 12 datasets

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 which minimizes the objective function would be computationally intractable. Therefore, we can only rely on a visual evalution of the regression lines to decide if the fits are satisfying. In Figure 4.8, 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 4.9 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.

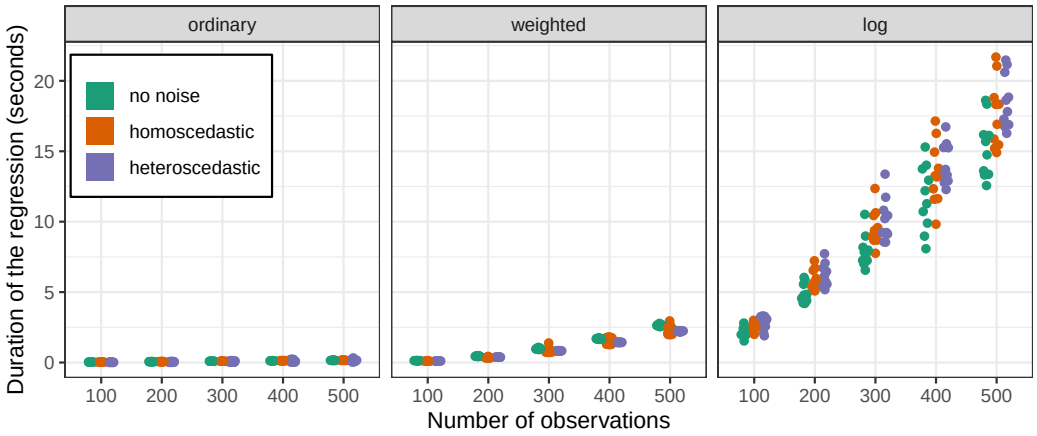


Figure 4.9.: Duration of a complete regression with pycewise for different numbers of observations.

In our context of network calibration, each communication for a given message size is repeated several time. 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.

4.4 Conclusion

We have presented different models for both the network communications and the computation kernels. One diffuclty resides in instanciating 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 paremeters. However, the benefit of these tools is not clear for our context, their computation time was particularly slow and 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 5, we will discuss what level of complexity is required for achieving a sufficiently faithful performance prediction. We will also demonstrate in Chapter 6 how these models can be used for sensibility analysis.

5 Validation

5.1 Experimental setup

To evaluate the soundness of our approach, we compare several real executions of HPL with simulations using the previous models. We used the Dahu cluster from the Grid'5000 testbed. It has 32 nodes connected through a single switch by $100\,\mathrm{Gbit\,s^{-1}}$ Omnipath 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. 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.

5.2 Different problem sizes

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 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 5.1). 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 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 destroy the efficiency of collective operations.

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 Figure5.2). 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: $\mathcal{M}-1$ for the kernels and $\mathcal{M}'-1$ for the network with no noise ($\mathcal{N}-0$) 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 stochastic multi-modal network model ($\mathcal{N}'-1$, the line (b)), does not significantly improve the prediction precision.

Figure 4.1 shows that there is an important heterogeneity in the cluster. For this reason, we started using a $\mathcal{M}_H-1/\mathcal{N}-0$ 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 $\mathcal{M}_H-1$ to $\mathcal{M}_H-2$) 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.

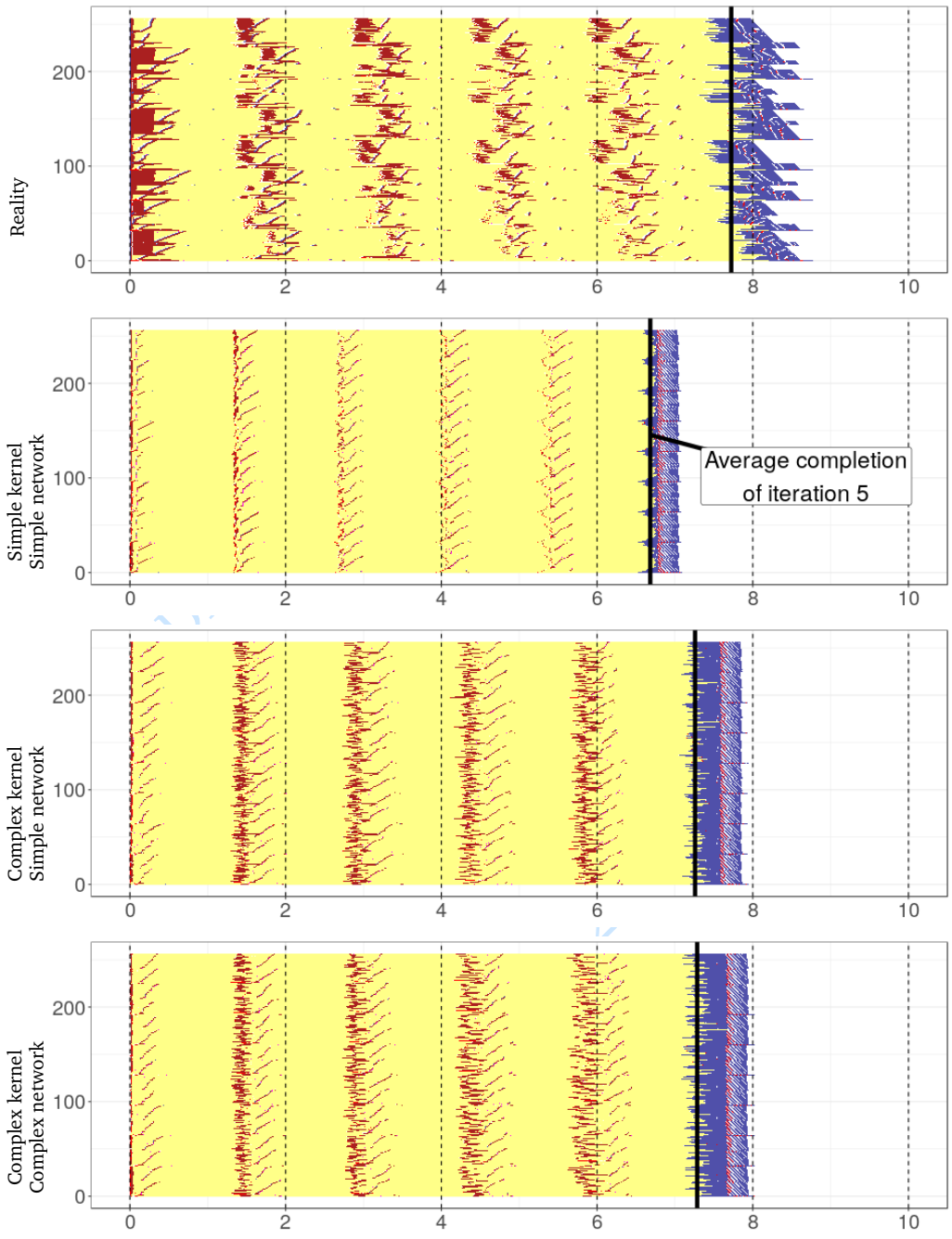


Figure 5.1.: Gantt charts of HPL first iterations in simulation

We found that adding the temporal variability noise ($\mathcal{N}-2$ for all kernels, $\mathcal{N}_H-2$ 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 ($\mathcal{N}'-1$, 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 very important aspect to model for applications that are more *latency-bound*.

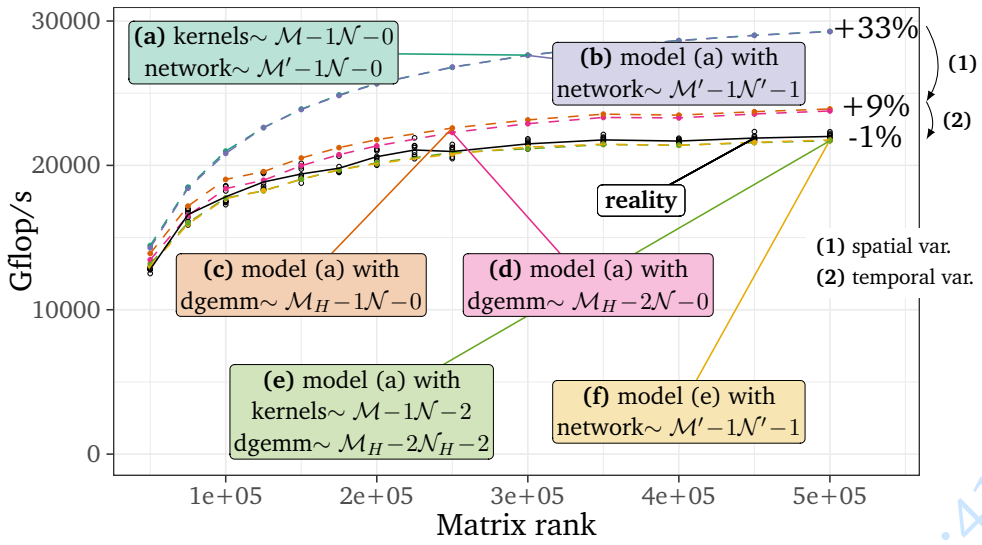


Figure 5.2.: HPL performance: predictions vs. reality for various matrix ranks

As illustrated in Figure 5.2 (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 an other application, only the `dgemm` kernel needs to be carefully modeled with a $\mathcal{M}_H-2$-$\mathcal{N}_H-2$. 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.

5.3 A platform change

Some text...

5.4 Factorial experiment

Some text...

5.5 Different geometries

Some text...

5.6 Conclusion

6 Sensibility analysis

6.1 Influence of temporal variability

Some text...

6.2 Influence of spatial variability

Some text...

6.3 Influence of the physical topology

Some text...

Part II

Experimental control

7 Experimental Testbed and Experiment Engines

7.1 State of the art

7.1.1 Grid'5000

Nearly all the experiments presented in this document have been carried 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.

7.1.2 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 to have 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 unpleasant and unintuitive to write and read. Section 7.2.2 will present some comparisons. 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.

7.2 Yet another experiment engine: `peanut`

7.2.1 Key features

We implemented our own experiment engine, named `peanut`. 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`).

7.2.2 Comparison with Execo

In this section, we use a small example to illustrate some 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 7.1 shows such an experiment using Execo. Run with: `python script.py`

```
1 import execo
2 import execo_g5k as g5k
3
4 site = 'grenoble'
5 cluster = 'dahu'
6 nodes = 2
7 time = '00:20:00'
8 image = 'debian9-x64-base'
```

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

```
9
10 if __name__ == '__main__':
11     query = "{cluster in ('%s')}/nodes=%d,walltime=%s" % (cluster, nodes, time)
12     [(jobid, site)] = g5k.oarsub([(g5k.OarSubmission(resources=query, job_type='deploy'), site)])
13     if jobid:
14         print('Created job %d' % jobid)
15         g5k.wait_oar_job_start(jobid, site)
16         node_list = g5k.get_oar_job_nodes(jobid, site)
17         g5k.deploy(g5k.Deployment(node_list, env_name=image), check_timeout=180)
18         print('Terminated deployment')
19         execo.Remote('apt update -qq', node_list).run()
20         execo.Remote('DEBIAN_FRONTEND=noninteractive apt install -qq -y build-essential make git cmake
       ',
21                      node_list).run()
22         execo.Remote('git clone https://github.com/RoaringBitmap/CRoaring.git CRoaring &&\
23                      cd CRoaring && git checkout v0.2.66', node_list).run()
24         execo.Remote('cd CRoaring && mkdir -p build && cd build && cmake .. && make -j', node_list).run()
25         print('Terminated installation')
26         p = execo.Remote('cd CRoaring && ./build/benchmarks/real_bitmaps_benchmark ./benchmarks/
       realdata/census-income',
27                      node_list).run()
28         for proc in p.processes:
29             print(proc.host.address)
30             print(proc.stdout)
31         print('Terminated experiment')
32         g5k.oardel([(jobid, site)])
```

Listing 7.1: Small experiment example Execo

This script, albeit fairly small, is already difficult to read in some places. For instance, 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 7.2 demonstrates how the same experiment can be rewritten using Peanut in a much more concise and readable way.

Run with: `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 7.2: Small experiment example using Peanut

The script is not only twice shorter, it is also much more elegant. Furthermore, it accomplishes much more than Listing 7.1, as it produces a peanut archive with all the metadata related to the experiment.

8 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 rational 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.

8.1 Experimental setup

All the experiments presented in this chapter share a common setup. They have been repeated on several nodes 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[1] to monitor the core frequencies and temperature every second.
4. On each core, run a custom code[2] 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[3]. 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[4]. We have used OpenBLAS[5] version 0.3.1 and OpenMPI[6] version 2.0.2 compiled with GCC version 6.3.0 on a Debian 9 installation with kernel version 4.9.0.

[1] `https://github.com/Ezibenroc/ratatouille`
[2] `https://github.com/Ezibenroc/platform-calibration/src/calibration`
[3] `https://www.grid5000.fr/w/Grenoble:Hardware`
[4] `https://en.wikichip.org/wiki/intel/xeon_gold/6130`
[5] `https://github.com/xianyi/OpenBLAS`
[6] `https://github.com/open-mpi/ompi`

8.2 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.

8.2.1 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:

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

8.2.2 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 8.1. A list of $100{,}000$ size tuples was sampled with each approach.

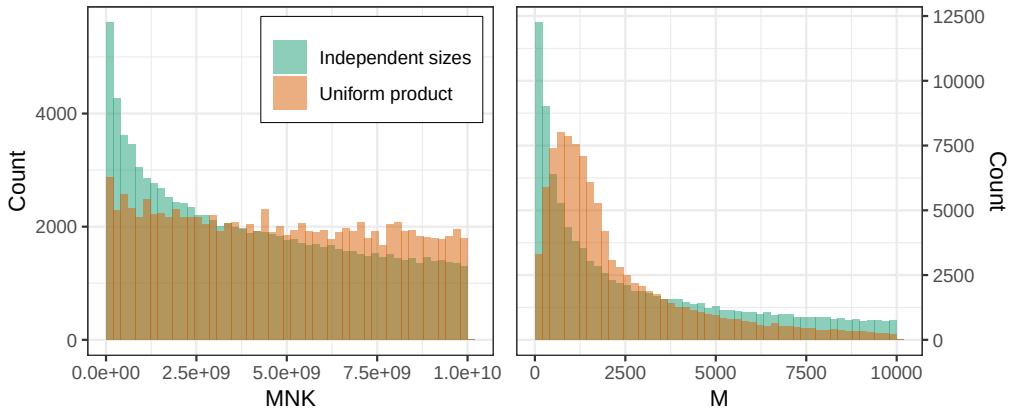


Figure 8.1.: Distribution of the product MNK and the size M with the two generation methods. The maximal product is set to 10^{10} and the maximal size to $10{,}000$.

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 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 slightly more deterministic 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.

8.3 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 a 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 8.2. 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 8.3. With the shuffled experiment (right plot), the average durations have a single-mode, as expected. However, in the

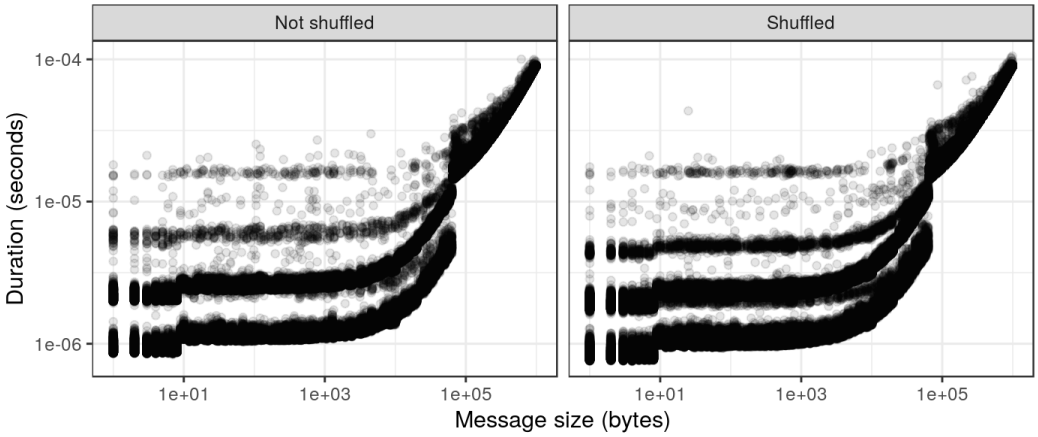


Figure 8.2.: The duration of `MPI_Recv` is piecewise linear, with several modes for small messages.

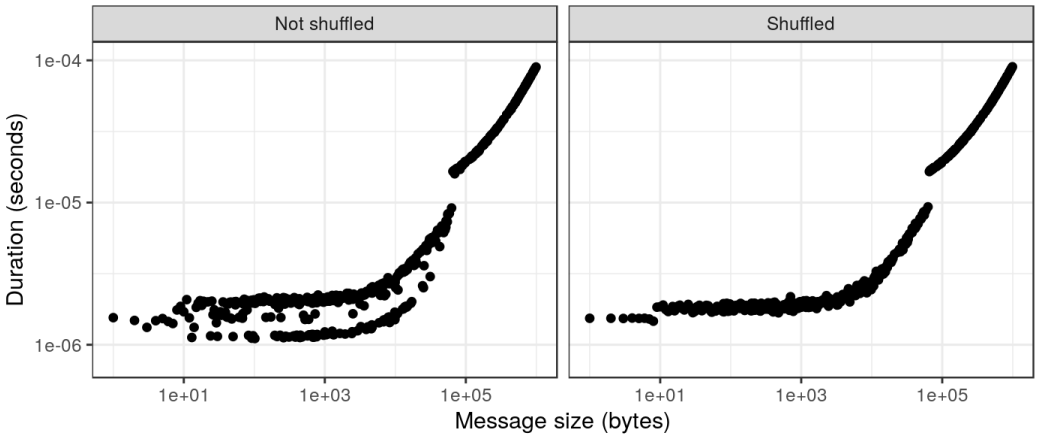


Figure 8.3.: The average durations of `MPI_Recv` in the non-shuffled case still show several modes, which should not happen according to the central limit theorem.

non-shuffled case (left plot), at least two modes are clearly present. This contradicts the conclusion of the central limit theorem, thereby proving that its hypotheses do not hold for this experiment. This is confirmed by Figure 8.4, 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.

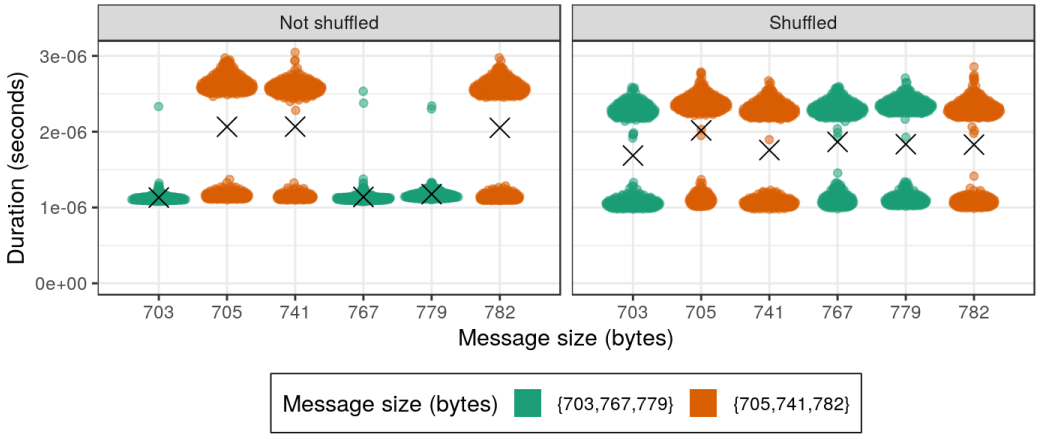


Figure 8.4.: Distribution of the `MPI_Recv` durations for six different message sizes between 700 B and 800 B. The durations are not identically distributed in the non-shuffled case. Durations truncated to 4 µs for a better readability.

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 8.5, we present the temporal evolution of the durations for the calls to `MPI_Recv` made with the sizes presented in Figure 8.4. 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 green) 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.

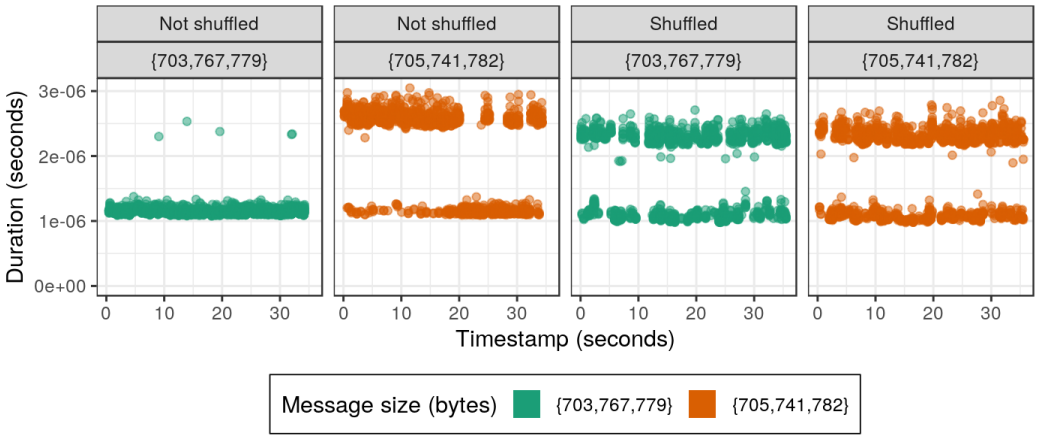


Figure 8.5.: Temporal evolution of the `MPI_Recv` durations for six different message sizes between 700 B and 800 B. A temporal pattern can be observed. Durations truncated to 4 μs for a better readability.

Another view of the non-shuffled experiment is presented in Figure 8.6. Now all the calls with a size lower than 1 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 μs. 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.

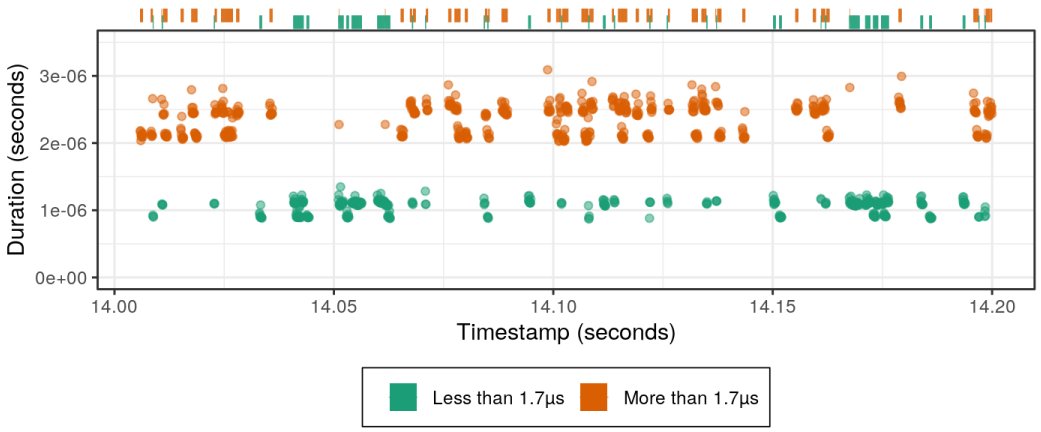


Figure 8.6.: Temporal evolution of the `MPI_Recv` durations for all the sizes between $1\,\mathrm{B}$ and $1\,\mathrm{kB}$ during a $0.2\,\mathrm{s}$ time window of the non-shuffled experiment. Another temporal pattern can be observed. Durations truncated to $4\,\mu\mathrm{s}$ for a better readability.

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 9.

8.4 Randomizing the sizes

The calibration measures for the `dgemm` function are done with a random sequence of tuples, as discussed in Section 8.2.2. This sequence is properly shuffled, so we eliminated the possible experimental bias discussed in Section 8.3. In this section, we will approach two difficulties that were encountered with the sizes themselves (as opposed to the order of the sequence).

8.4.1 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, 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 dozen of times during a week-end 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 8.7. 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.

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 8.8, 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 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

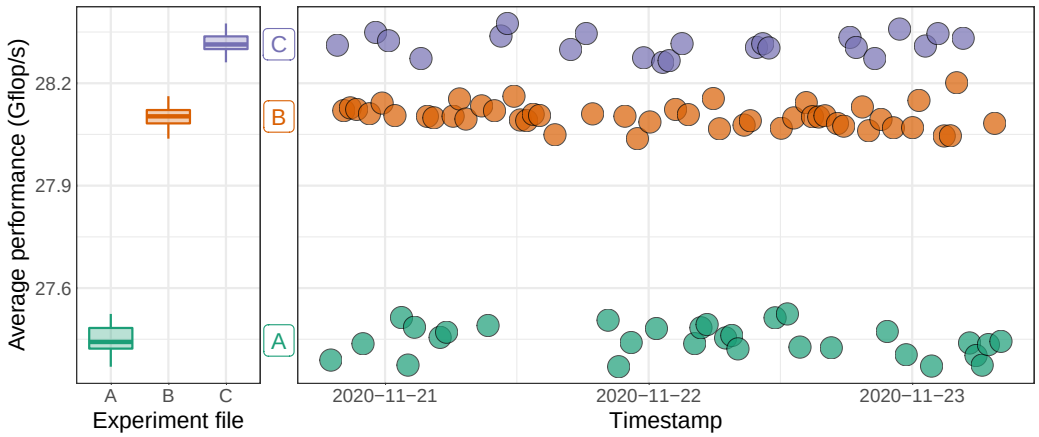


Figure 8.7.: Average performance observed on CPU 1 of dahu-5, each point represents one experiment. A significant part of the variability is due to the choice of the experiment file.

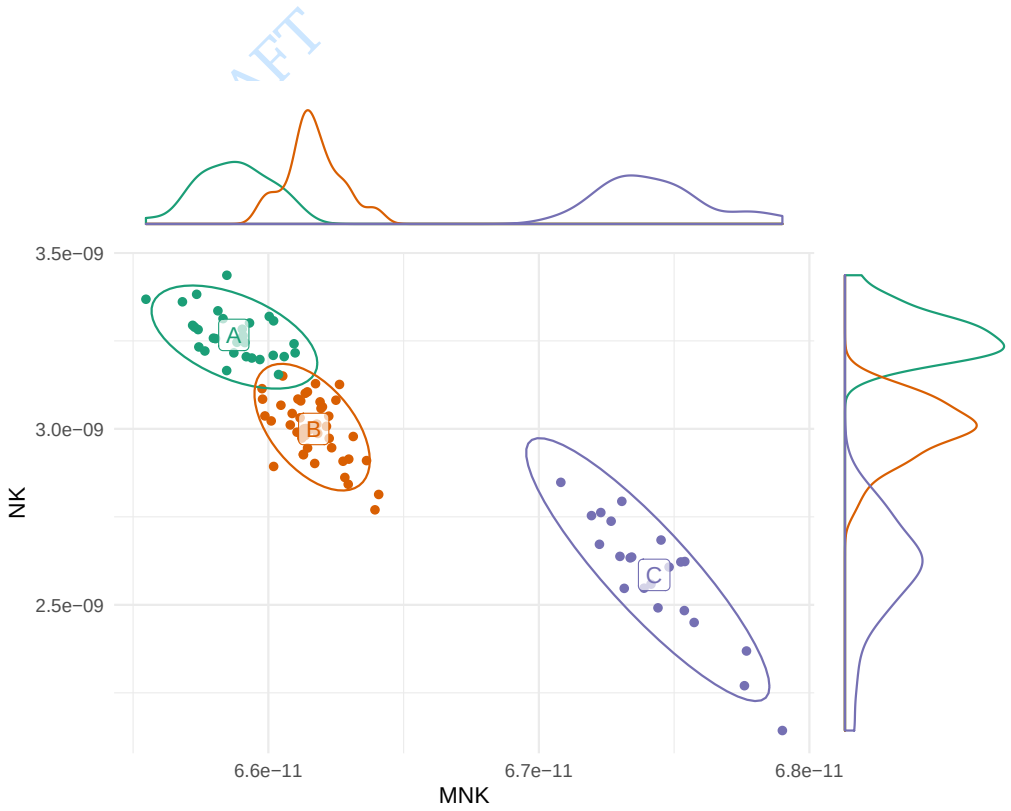


Figure 8.8.: Distribution of two of the regression parameters for CPU 1 of dahu-5, each point represents one experiment. The experiment file has a clear effect on the generated model

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 8.9, 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 been made with a tall and skinny matrix, with $K \geq 3000$, which corroborates the hypothesis of a bad cache utilization.

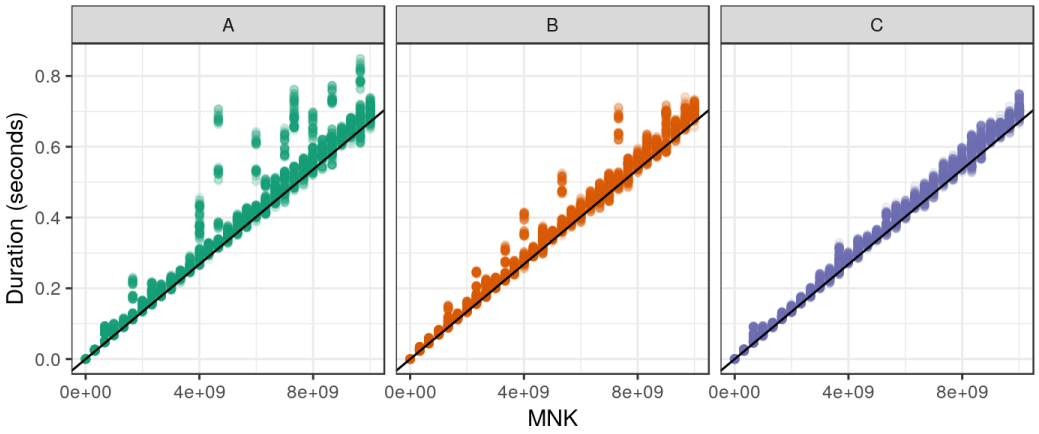


Figure 8.9.: Durations of individual `dgemm` calls for CPU 1 of dahu-5. Several calls have significantly longer durations than others. Identical black line on the three plots, with slope 6.7×10^{-11}.

A final argument towards this hypothesis is presented with Figure 8.10, the average DRAM power consumption during each run is presented. Similarly to Figure 8.7, 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 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.

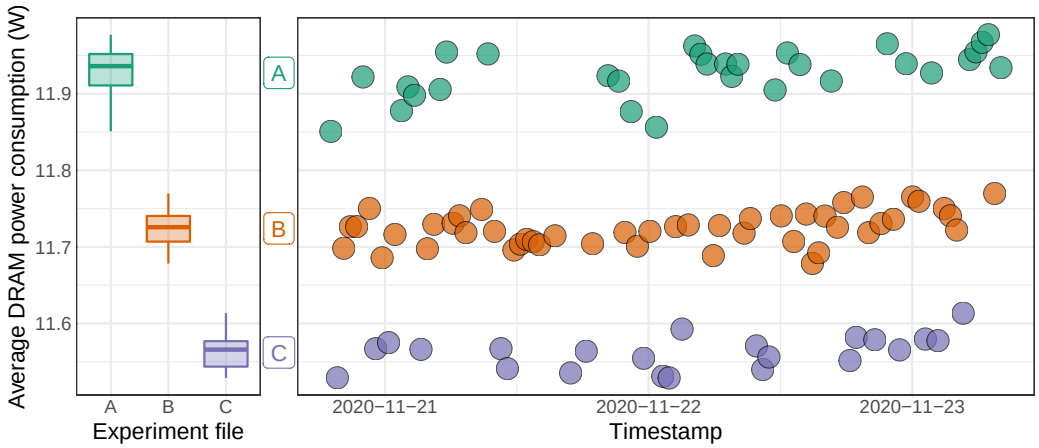


Figure 8.10.: Average DRAM power consumption observed on CPU 1 of dahu-5, each point represents one experiment.

8.4.2 Effect of the experiment file generation method

Section 8.4.1 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 8.2. 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 8.11. With the *uniform product* method, `dgemm` average performance is higher and the DRAM power consumption is lower.

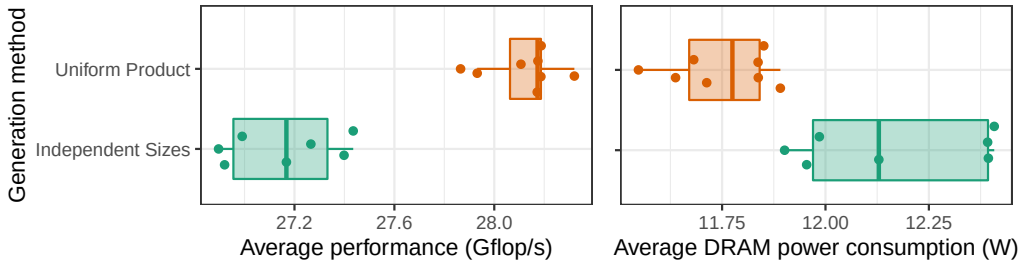


Figure 8.11.: Average `dgemm` performance and power consumption observed on CPU 1 of dahu-5, each point represents one experiment.

8.4.3 Effect of calibrating with a fixed size

In the experiments described in Section 8.4.1 and Section 8.4.2, 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 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 dozen of experiment files. Then, we performed one experiment with each of these files in a random order during a week-end 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 8.12 presents the observed `dgemm` performance with the five experiments. We can observe that again, the generation method for the size sequence has a huge 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 $\{128, 256, 512\}$.

The monitoring data collected during the experiments also reveal interesting differences. The average CPU frequency is reported in Figure 8.13. 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

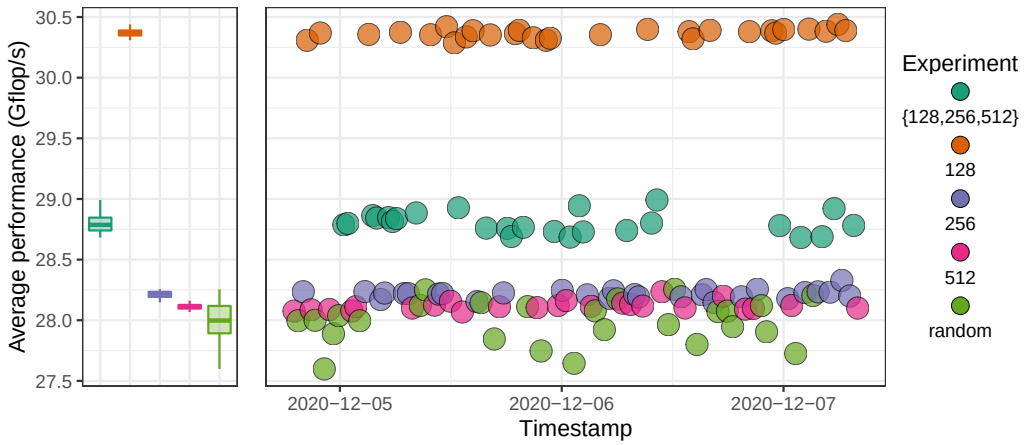


Figure 8.12.: Average `dgemm` performance observed on CPU 1 of dahu-5, each point represents one experiment.

with $K = 256$ and $K = 512$. It is interesting to note that 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.

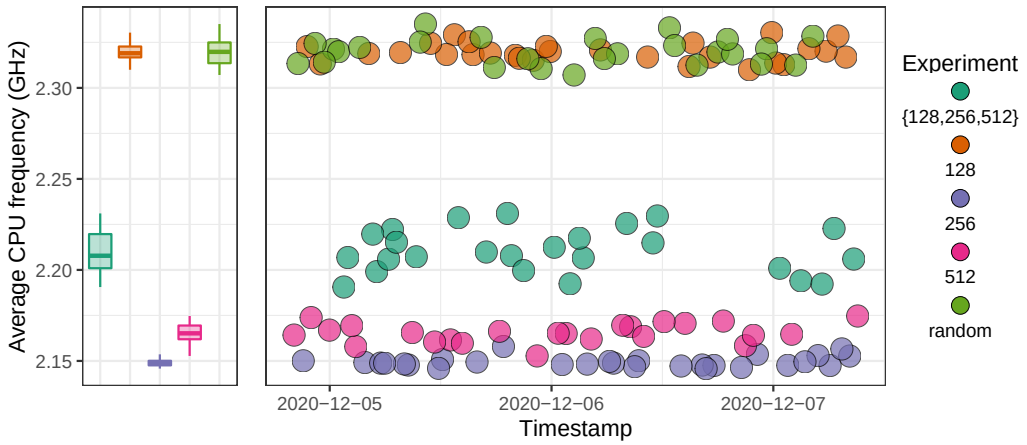


Figure 8.13.: Average CPU frequency, observed on CPU 1 of dahu-5, each point represents one experiment.

The average CPU power consumption, presented in Figure 8.14, is particularly 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.

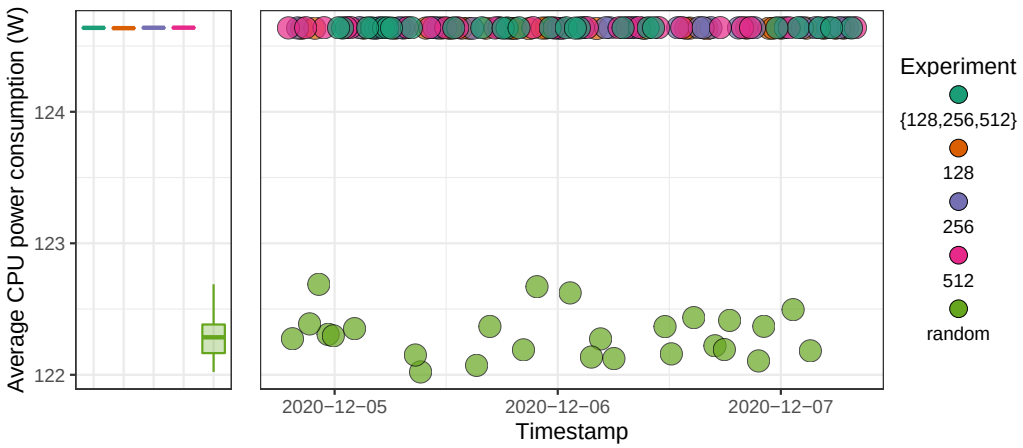


Figure 8.14.: Average CPU power consumption, observed on CPU 1 of dahu-5, each point represents one experiment.

The five experiments exhibit very different average DRAM power consumption, as depicted in Figure 8.15. 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 8.4.1.

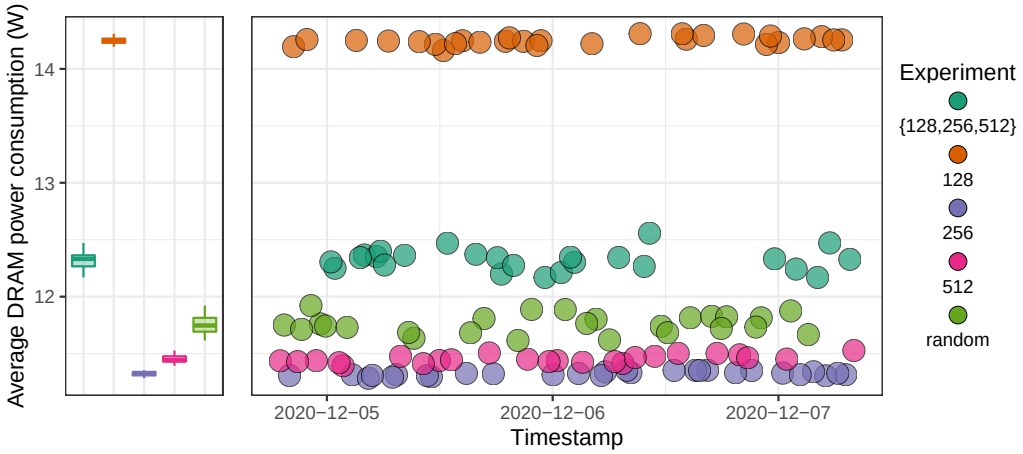


Figure 8.15.: Average DRAM power consumption, observed on CPU 1 of dahu-5, each point represents one experiment.

Finally, the performance of individual `dgemm` calls are presented in Figure 8.16. 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.

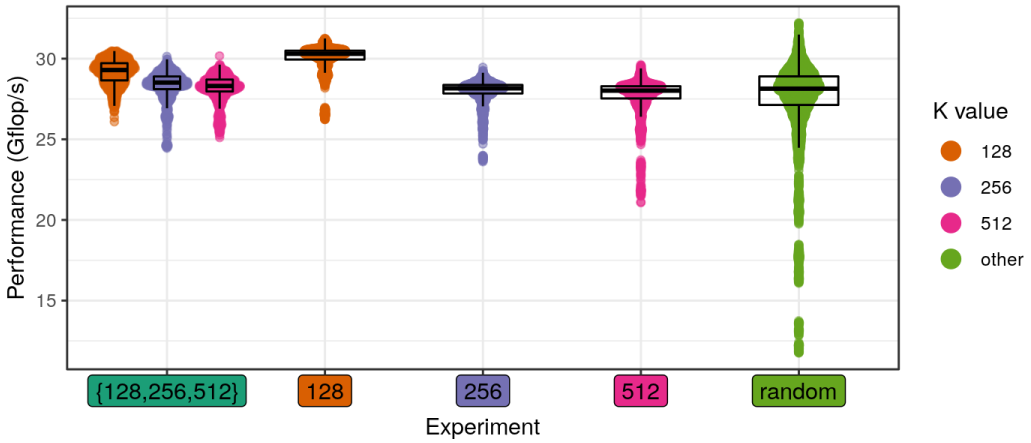


Figure 8.16.: Durations of individual `dgemm` calls for CPU 1 of dahu-5.

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 we could build using 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.

8.5 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 a bit 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.

8.5.1 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 made of 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 8.17 shows the evolution of the `dgemm` durations during the experiment. A clear temporal patterns 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.

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 8.18 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

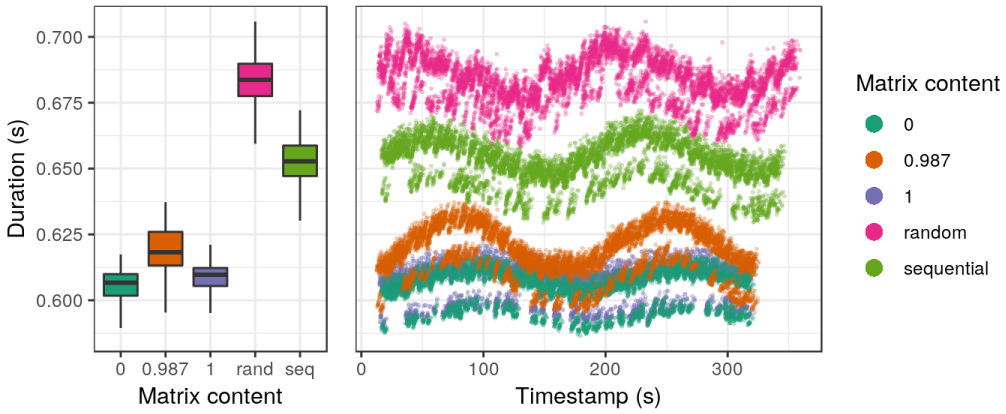


Figure 8.17.: `dgemm` durations are lower with constant values in the matrices

the constant initialization gives higher frequencies. A similar temporal patterns can also be distinguished with clear oscillations.

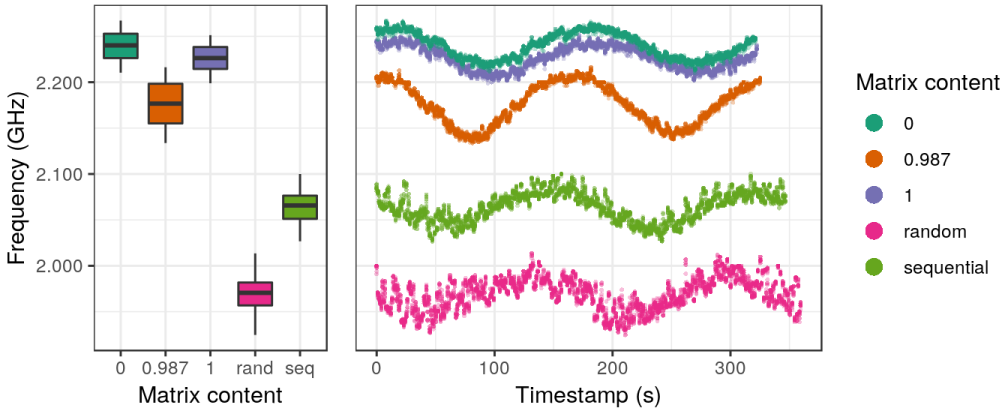


Figure 8.18.: Core frequencies are higher with constant values in the matrices

This experiment has been repeated on other Grid'5000 clusters, each time on at least four distinct nodes. Table 8.1 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.

8.5.2 Hypotheses

Several hypotheses were discussed to explain this unexpected phenomenon.

Table 8.1.: 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

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.

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 causes less bit flips than the random initialization but more bit flips than the constant initialization and thus an intermediate performance.

8.5.3 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 8.19, 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[7]. 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).

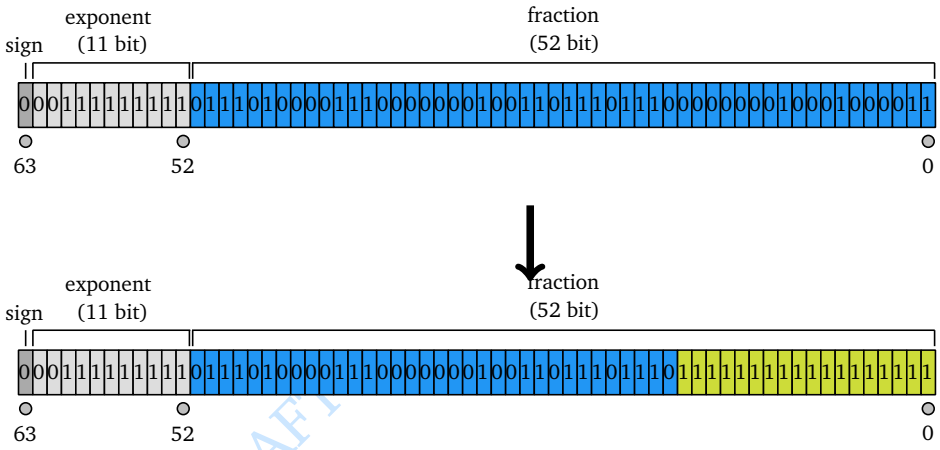


Figure 8.19.: Illustrating the effect of applying a mask on the random part of the matrix elements

The evolution and the distribution of the `dgemm` durations is plotted in Figure 8.20. 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.

This correlation with the mask size can also be seen with the frequencies in Figure 8.21: larger masks lead to higher frequencies.

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.

8.5.4 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

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

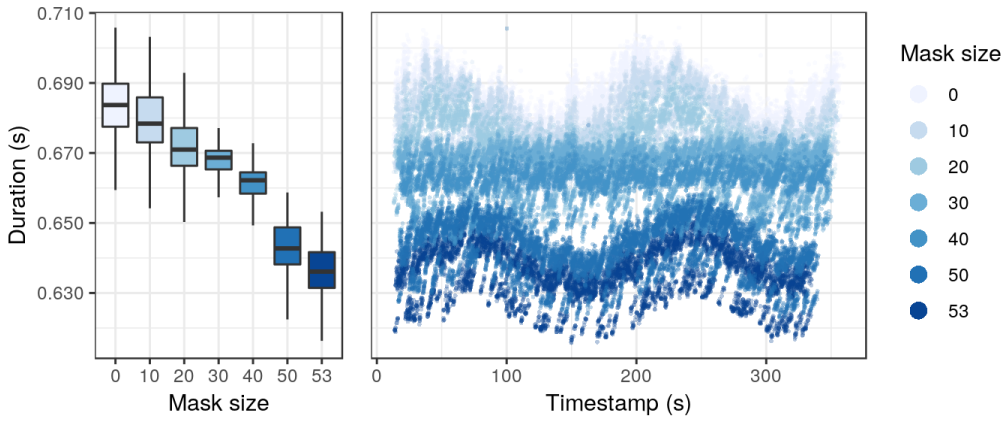


Figure 8.20.: `dgemm` durations are lower with larger bit masks

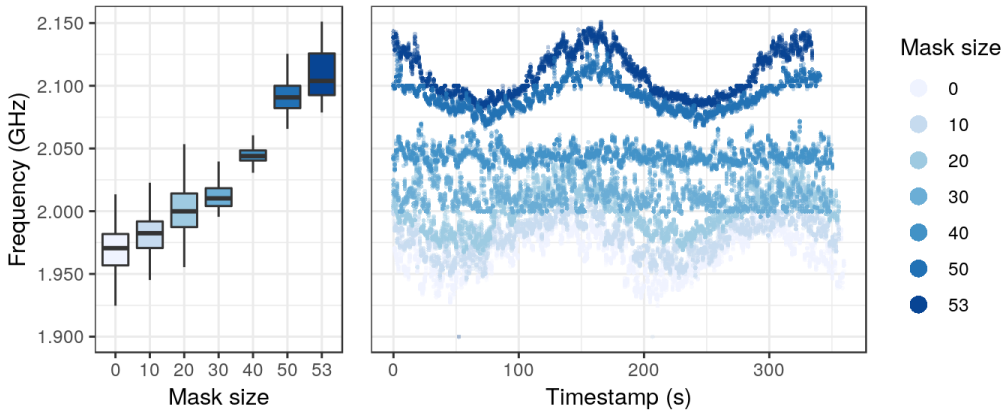


Figure 8.21.: Core frequencies are higher with larger bit masks

caused by bit flips inside the processor. To respect its energy budget, the CPU has to throttle more aggressively its frequency when the matrices content is 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 that helped us to 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.

8.5.5 Related work

M. Laß, C. Plessl and R. Schade (Paderborn Center for Parallel Computing) made independently 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. This experiment was done using another `dgemm` implementation than us (Intel MKL) on an Intel Xeon Gold 6148 (Skylake-SP family) from a noctua node[8].
- 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[9] to perform multiplications with more or less random data, using only the registers of the CPU. They did not observe

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

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.

8.6 Beware of extrapolations

Should we remove this section? It seems pretty obvious and uninteresting...

In 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 8.2. 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 outside of its parameter space.

8.7 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 8.22.

- 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 can appear oddly twisted. 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 8.22. 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

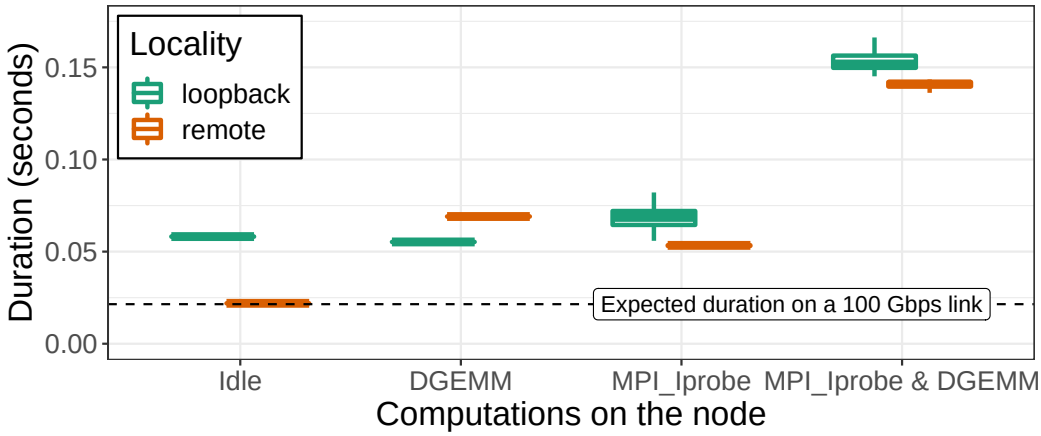


Figure 8.22.: Durations of a call to `MPI_Recv` with a message of 256 MiB and different background activities.

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 s.

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.

8.8 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 phenomenons that were only revealed through the 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, begin 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 artifacts, it can become more difficult to trust any experimental results, especially in the presence of unexpected phenomenons like the ones presented here. This demonstrates the great importance of gaining back this trust by (1) having an experiment engine to automatize everything (thereby limiting greatly the human errors) and collect relevant information and (2) making regular tests on the platform to detect eventual changes.

9 Performance non-regression tests

When working on the simulations described in Part I, we occasionally encountered inconsistent performance. 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 and explaining the statistics we based our work on in Section **??**. Then, in Section 9.2 we describe how we implemented the 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 Grid'5000 platform. Finally, in Section 9.3 we propose several ways to improve this work.

We should discuss the normality assumption somewhere, I am not sure of the best location though.

9.1 State of the art

Testing is a core activity of software development. It is usually taught as soon 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 correction.

- 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 *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.

9.1.1 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

[1] https://github.com/xianyi/OpenBLAS/tree/f917c26e/benchmark

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.

9.1.2 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:

[2] https://open-mpi.github.io/mtt/
[3] `https://github.com/simdjson/simdjson/issues/812`
[4] https://github.com/google/benchmark/blob/8f5e6ae0/tools/compare.py

- 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 unfortunate in our use case where we measure longer function calls that take at least several milliseconds.

- There is no support for randomization, 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.

- 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

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 measure their performance, both in reality and in simulation. The goal is to detect any

performance regression eventually introduced by a change in the software. Figure 9.1 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 an extremely 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, they allow ruling out the false conclusions that would be made by looking only at the real runs.

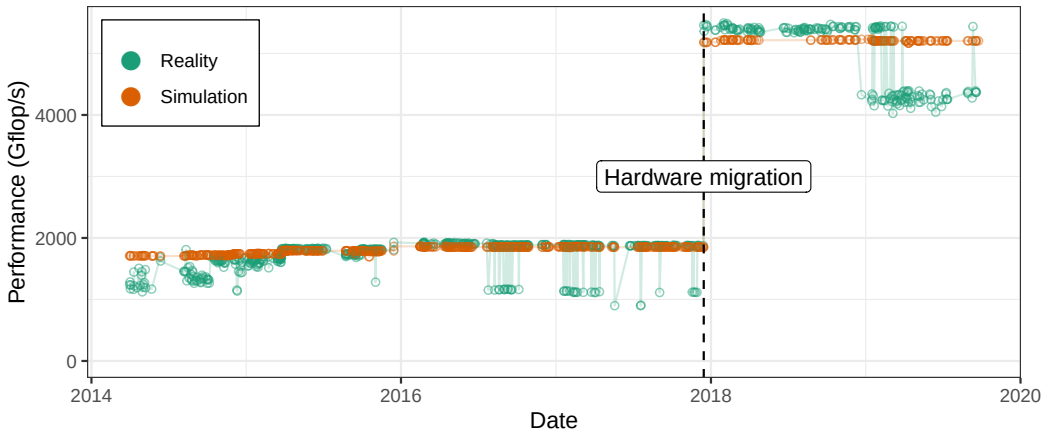


Figure 9.1.: Evolution of the performance of a StarPU implementation of `spotrf` function, both in reality and in simulation [TSL20].

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, this 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 inspection very tedious.

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

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+] 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 *correction* 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.

9.1.3 Statistical test

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 paper from Chew [Che66, Section 4.4].[6]

[6] This publication, dating from 1966, is the oldest paper 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

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:

$$\frac{nr}{n+r}(\boldsymbol{m}^{(r)} - \overline{\boldsymbol{x}})^T \boldsymbol{S}^{-1}(\boldsymbol{m}^{(r)} - \overline{\boldsymbol{x}}) = \frac{(n-1)p}{n-p} Q_F(1-\gamma, p, n-p) \quad (9.1)$$

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:

$$t = \frac{nr(n-p)}{(n+r)(n-1)p}(\boldsymbol{m}^{(r)} - \overline{\boldsymbol{x}})^T \boldsymbol{S}^{-1}(\boldsymbol{m}^{(r)} - \overline{\boldsymbol{x}}) \quad (9.2)$$

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 9.2. 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

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.

[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`

they both fall within the $99.5\,\%$ interval. Now, when we look at both dimensions simultaneously, the green point becomes much more likely to be observed, it is within the blue ellipse while the orange point is outside.

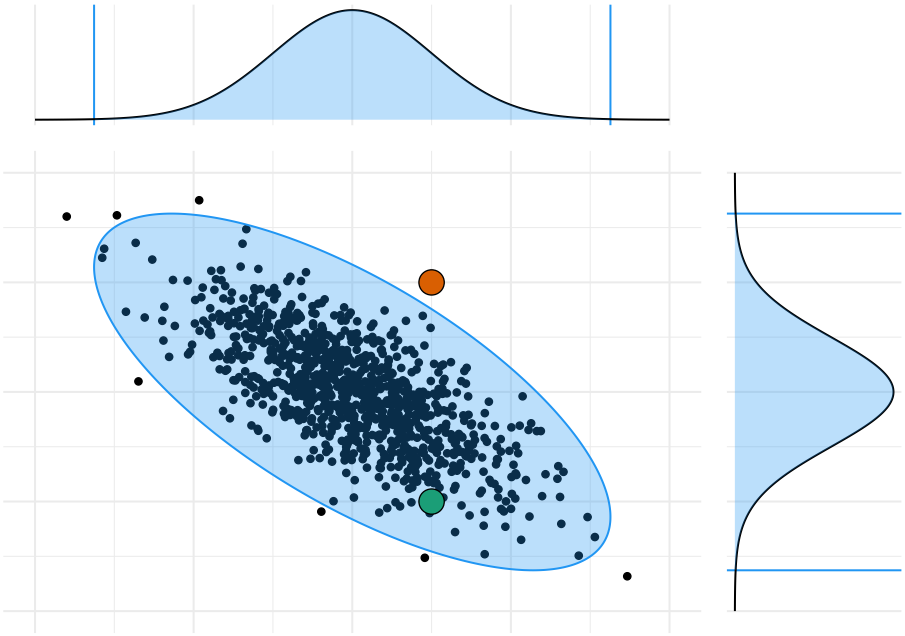


Figure 9.2.: Illustrating the proposed test with two sets of one observation ($r = 1$) and a bivariate normal distribution ($p = 2$). The blue zones represent the $99.5\,\%$ prediction regions.

In Figure 9.3, 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.

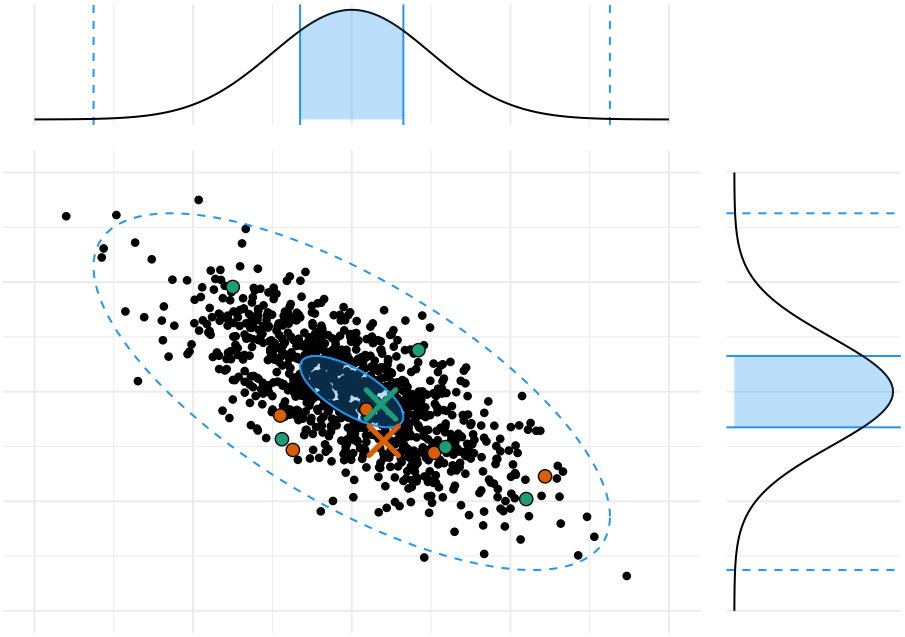


Figure 9.3.: Illustrating the proposed test with two sets of five observations ($r = 5$) and a bivariate normal distribution ($p = 2$). The blue dashed lines represent the $99.5\,\%$ prediction regions for single observations, the blue zones represent the $99.5\,\%$ prediction regions for the averages of five new observations, represented by crosses.

9.2 Implementation of the test

9.2.1 Workflow

This section describes the different steps and tools involved to perform this performance test, 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 consists in three main steps:

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.

Experiment execution

The experiment script is written in Python and uses the Peanut experiment engine[9] (see Section 7.2). 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 or 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

[9] `https://github.com/Ezibenroc/peanut`

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 will perform 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 hyper-threading, 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 setup 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[10], a Python library implemented specifically for this task. 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

[10] `https://github.com/Ezibenroc/cashew`

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, it automatically downloads the required files.

The implementation directly follows the formula described in Section 9.1.3. 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 realised, 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.

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

Frequency Average CPU frequency during the `dgemm` calls.

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

$\text{Power}_{\text{DRAM}}$ 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 9.2.3.

As for the previous step, the test is implemented in 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.

9.2.2 On the normality assumption

The statistical test described in Section 9.1.3 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 $\hat{\beta}$ can be obtained by computing $\hat{\beta} = \left(\boldsymbol{X}^\mathsf{T}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^\mathsf{T}\boldsymbol{y}$. By doing this regression, we assume that $\boldsymbol{y} = \boldsymbol{X}\beta + \varepsilon$ where β is the true (unknown) vector of coefficients and ε is the normally distributed noise. It follows that the estimator $\hat{\beta}$ is also normally distributed.

- The factors Performance and 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 normally distributed.

9.2.3 Presentation of the test results

Evolution plot

The historical evolution of the mean performance of two nodes is presented in Figure 9.4. 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 reseted.

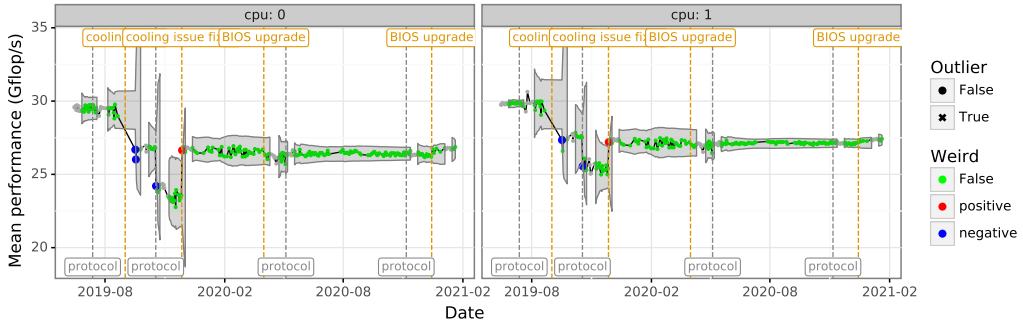
The fluctuation interval for a new value $\boldsymbol{m}^{(r)}$ (which is the sample mean of r new observations $x_{n+1}, \ldots, x_{n+r}$, note that $r = 1$ in Figure 9.4) follows a rewriting of Equation 9.2 with a single dimension ($p = 1$). Noting $\overline{\boldsymbol{x}}$ (resp. $\boldsymbol{s}^2$) the sample mean (resp. sample variance) of the reference set, the fluctuation interval is defined as:

$$I = \overline{\boldsymbol{x}} \pm s\left(\sqrt{\frac{n+r}{nr} Q_F(\gamma, 1, n-1)}\right) \tag{9.3}$$

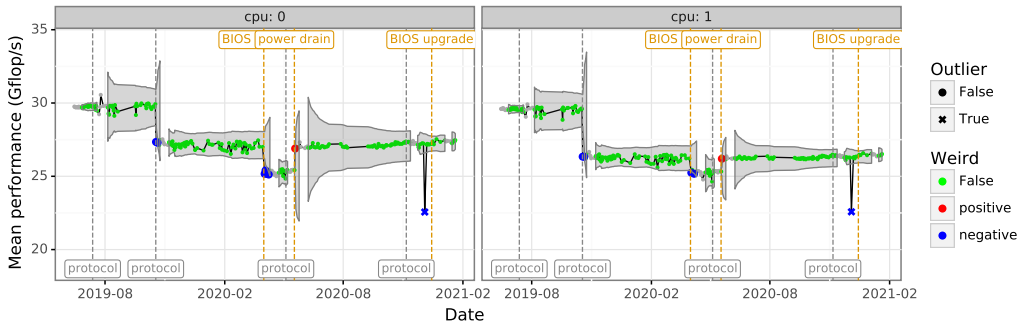
Whenever an observation falls outside the fluctuation interval, it is detected as an anomaly. It is classified as either a positive anomaly and colored in red if it is larger

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 9.4(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.



(a) Performance evolution of node *dahu-14*



(b) Performance evolution of node *dahu-26*

Figure 9.4.: Evolution of the mean performance of the two processors of two nodes from cluster *dahu* (the fluctuation interval has a confidence of $99.99\,\%$)

Overview plot

We are running regular tests for several factors on hundreds of nodes, it would be very long and tedious to review each individual plot. For this reason, we implemented overview plots, as presented in Figure 9.5. 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.

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 9.2 and define the likelihood $\mathcal{L}$ as follows:

$$\mathcal{L} = 1 - F_{F(p,n-p)}(t) \tag{9.4}$$

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].

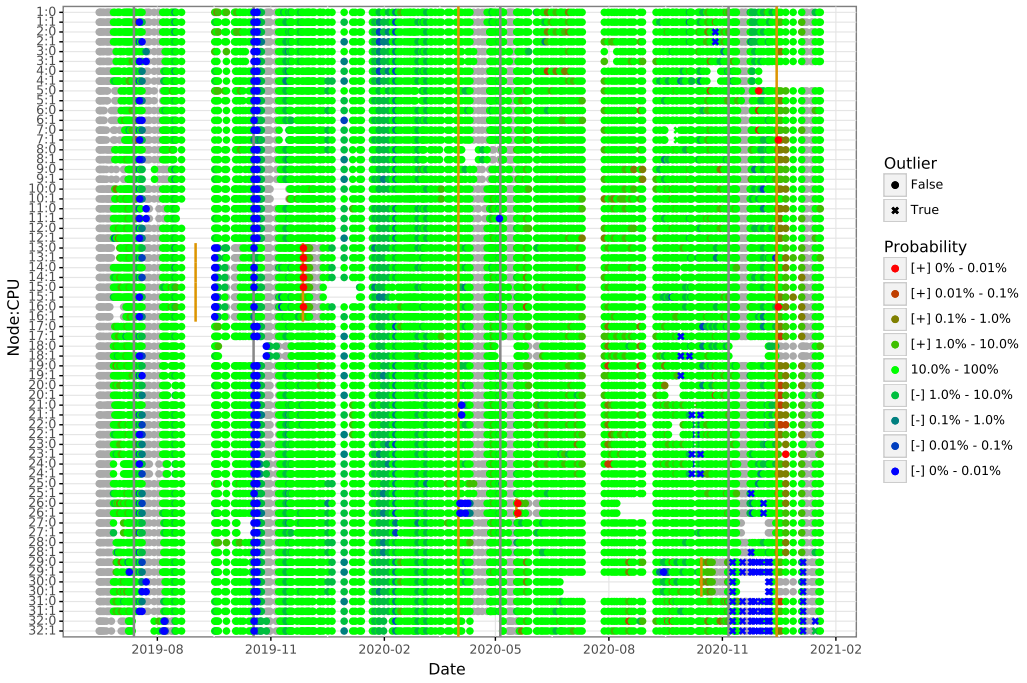


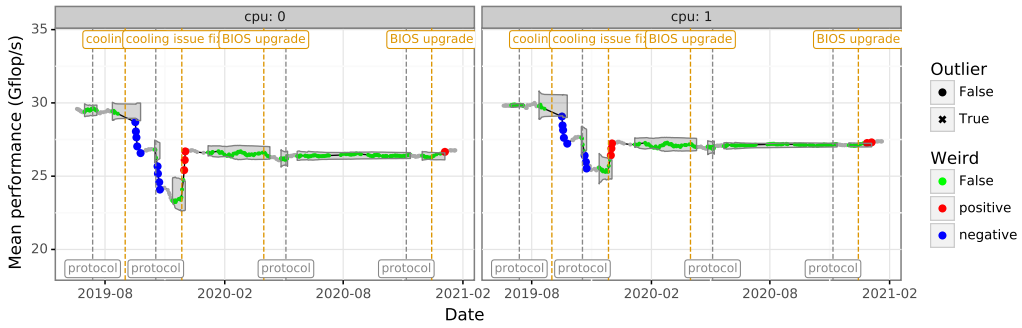
Figure 9.5.: Overview of the test result for the mean performance on cluster *dahu*.

Windowed test

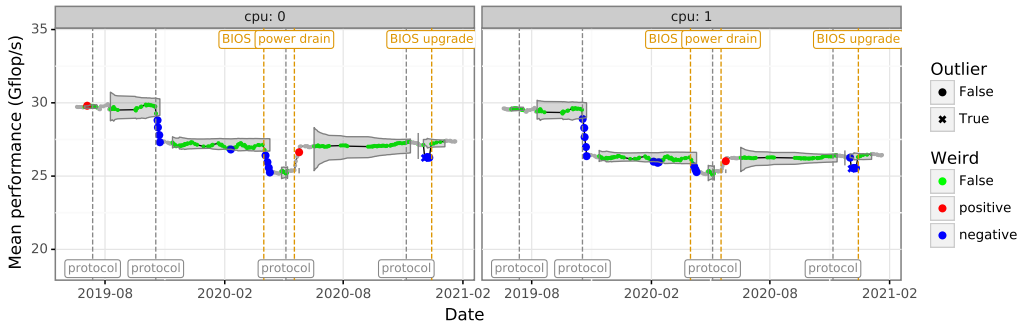
The test presented in Figure 9.5 and Figure 9.4 was done with a single factor ($p = 1$), for comparing a single new observation to the reference set ($r = 1$). A similar test is implemented to compare five new observations to the reference set, still with a single factor ($p = 1$ and $r = 5$).

Figure 9.6 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, the change might not be detected immediately.



(a) Performance evolution of node *dahu-14*



(b) Performance evolution of node *dahu-26*

Figure 9.6.: 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 9.7. 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 9.5, 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.

Multi-factor test

We have also implemented the multidimensional test. Figure 9.8 presents the overview plot for this test made on the eight regression parameters (i.e. $p = 8$, the parameters areMNK, MN, MK, NK, M, N, K and the intercept). With several

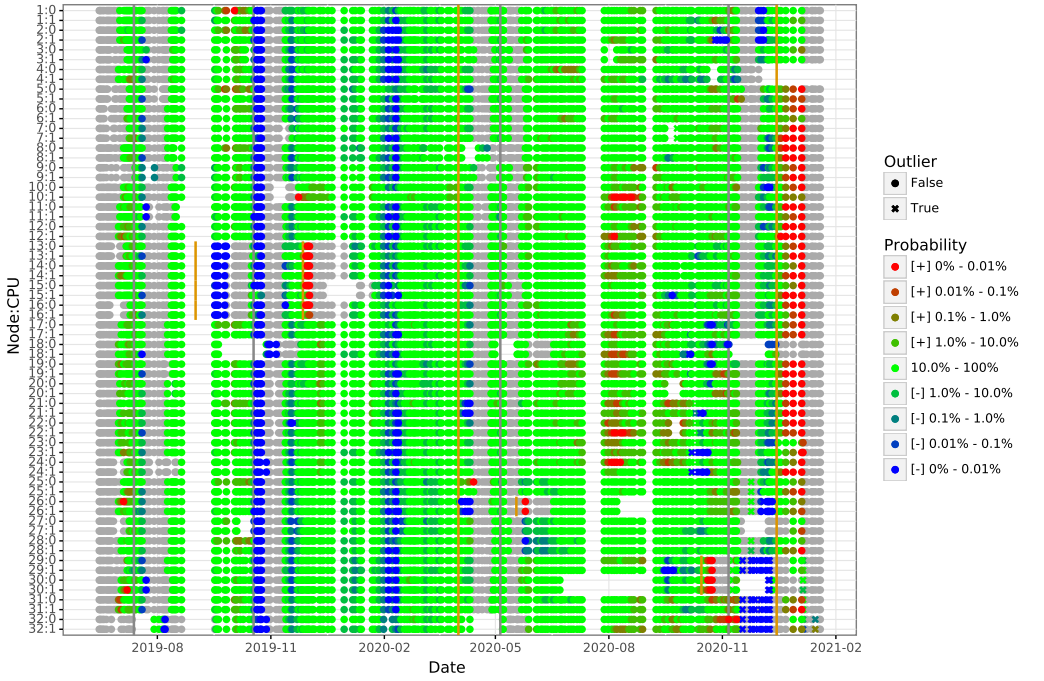


Figure 9.7.: Overview of the test result for the mean performance on cluster *dahu* using a window of five experiments

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.

This figure should be compared to Figure 9.5, 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.

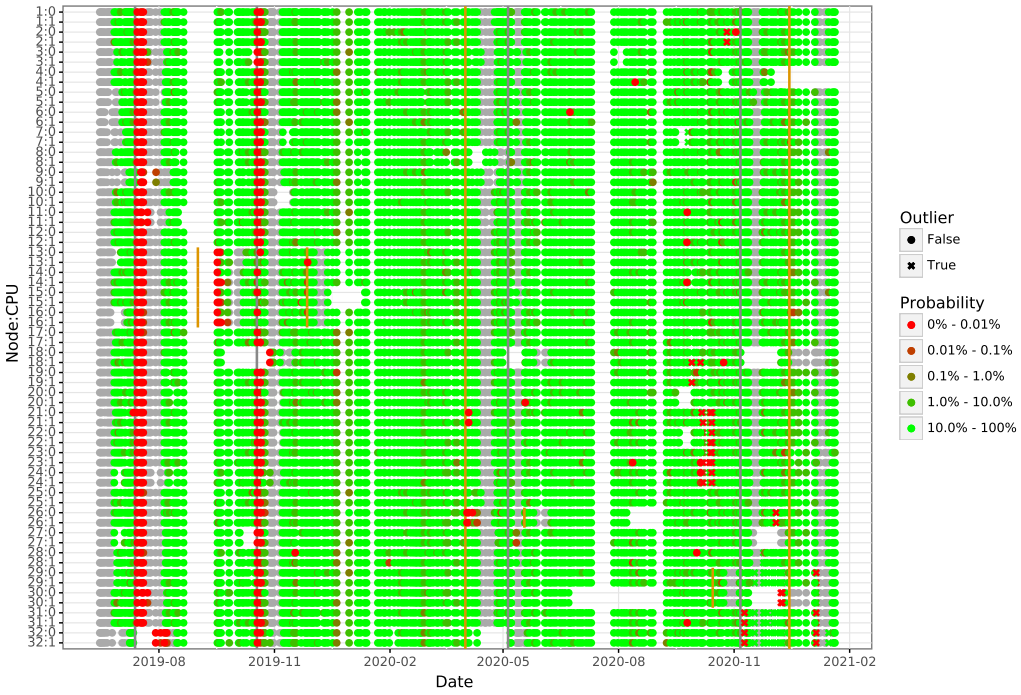


Figure 9.8.: Overview of the test result for the eight regression parameters on cluster *dahu*.

Website presentation

A screenshot of the landing page[11] of the Grid'5000 performance tests we implemented is presented in Figure 9.9. 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.

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

Add a Zenodo link

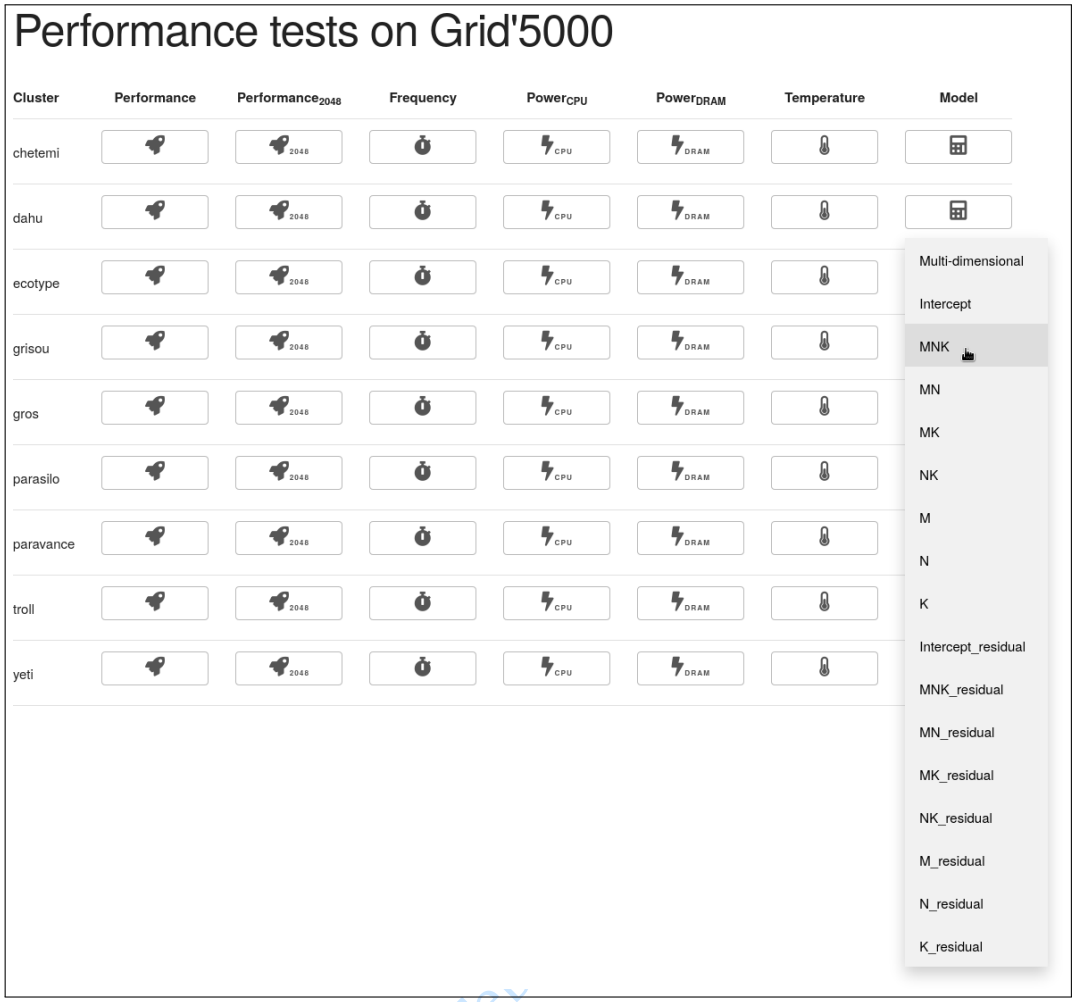


Figure 9.9.: Landing page of the test website.

An extract of a single-factor notebook is presented in Figure 9.10. The different parts of the notebooks are:

a This part is intended to import the required libraries and setup 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 brutal 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.

9.2.4 Detected events

Our non-regression tests have been used on a regular basis on nine Grid'5000 clusters during a period of several months (from July 2019 to February 2021 for *dahu*, the first cluster we started to monitor). In total, we have tested 386 nodes (656 processors).

The list and basic characteristics of these clusters is described in table 9.1. It has been curated from Grid'5000 documentation[12].

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

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

Figure 9.10.: Overview of a notebook (cluster *gros*, average performance).

Table 9.1.: 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
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
parasilo	28	2× Intel Xeon E5-2630 v3	2 × 8	128 GiB
paravance	72	2× Intel Xeon E5-2630 v3	2 × 8	128 GiB
troll	4	2× Intel Xeon Gold 5218	2 × 16	384 GiB
yeti	4	4× Intel Xeon Gold 6130	4 × 16	768 GiB

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 of temperature 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 $20\,°\mathrm{C}$. 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 happened 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 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 inverting two memory sticks, sometimes by changing one memory stick for a new one, 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 $124.63\,\mathrm{W}$ and $124.65\,\mathrm{W}$. 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 $124\,\mathrm{W}$, which is already very significant given the extreme stability that we usually observe. On half of these events, the power consumption was below $119.4\,\mathrm{W}$. The largest power drop happened on node *dahu-26* on 2020/12/04, where its CPU n°0 consumed only $97.5\,\mathrm{W}$. 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 January 2021. Since the four nodes are part of the same chassis and therefore share their power supply, our hypothesis is 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.

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 9.2 summarizes those unexplained changes. Most of the time, the frequency and `dgemm` performance were also slightly affected, but to a lower extent.

The two nodes *parasilo-1* and *parasilo-11* also had a temperature increase of $5\,^{\circ}\mathrm{C}$ 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.

Table 9.2.: 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

9.3 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 performance.

We have monitored 386 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.

Their 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.

Conclusion

Your beautiful conclusion. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Bibliography

[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. cit. on p. 98

[Arrow] [SW], *Apache Arrow*, LIC: Apache. VCS: `https://github.com/apache/arrow`. cit. on p. 114

[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. cit. on p. 106

[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 pp. 14, 16

[Bal+13] Daniel Balouek, Alexandra Carpen-Amarie, Ghislain Charrier, et al. “Adding Virtualization Capabilities to the Grid’5000 Testbed”. In: *Cloud Computing and Services Science*. Ed. by IvanI. Ivanov, Marten Sinderen, Frank Leymann, and Tony Shan. Vol. 367. Communications in Computer and Information Science. Springer International Publishing, 2013. cit. on pp. 23, 67

[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. 15

[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. cit. on p. 68

[Car+17] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, et al. “Stan: A Probabilistic Programming Language”. In: *Journal of Statistical Software* 76.1 (2017). cit. on p. 38

[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. cit. on pp. 15, 16, 17, 104

[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. cit. on p. 14

[Catch2] [SW], *Catch2*, LIC: BSL. VCS: `https://github.com/catchorg/Catch2`. cit. on p. 105

[Celero] [SW], *Celero*, LIC: Apache. VCS: `https://github.com/DigitalInBlue/Celero`. cit. on p. 105

[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. cit. on p. 108

[CL19] Tom Cornebize and Arnaud Legrand. *DGEMM performance is data-dependent*. Research Report RR-9310. Université Grenoble Alpes ; Inria ; CNRS, Dec. 2019. cit. on p. 92

[CL21] Tom Cornebize and Arnaud Legrand. “Simulation-based Optimization and Sensibility Analysis of MPI Applications: Variability Matters”. working paper or preprint. Feb. 2021. cit. on p. 11

[CLH19] Tom Cornebize, Arnaud Legrand, and Franz C Heinrich. “Fast and Faithful Performance Prediction of MPI Applications: the HPL Case Study”. In: *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. 2019 IEEE International Conference on Cluster Computing (CLUSTER). Albuquerque, United States, Sept. 2019. cit. on p. 11

[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. 15

[Cor17] Tom Cornebize. “Capacity Planning of Supercomputers: Simulating MPI Applications at Scale”. MA thesis. Grenoble INP ; Université Grenoble - Alpes, June 2017. cit. on p. 11

[Dat] [SW] Datadog, *Piecewise*, LIC: BSD. VCS: `https://github.com/DataDog/piecewise`. cit. on pp. 42, 47

[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. cit. on pp. 17, 18, 28, 79

[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 pp. 15, 16

[GBench] [SW], *Google Benchmark*, LIC: Apache. VCS: `https://github.com/google/benchmark`. cit. on p. 105

[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. cit. on p. 42

[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. cit. on p. 15

[Gri05] David Alan Grier. *When Computers Were Human*. Princeton University Press, 2005. cit. on pp. 3, 4

[Hayai] [SW], *Hayai*, LIC: Apache. VCS: `https://github.com/nickbruun/hayai`. cit. on p. 105

[HDF5] [SW], *HDF5*, LIC: BSD. VCS: `https://github.com/HDFGroup/hdf5`. cit. on p. 114

[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. cit. on pp. 18, 89

[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. cit. on p. 68

[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). `http://dx.doi.org/10.4018/jdst.2010040104`, pp. 57–73. cit. on pp. 15, 16

[KQ20] Max Kuhn and Ross Quinlan. *Cubist: Rule- And Instance-Based Regression Modeling*. R package version 0.2.3. 2020. cit. on pp. 42, 47

[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. cit. on pp. 14, 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. 6

[LL19] Geoff Langdale and Daniel Lemire. “Parsing Gigabytes of JSON per Second”. In: *CoRR* abs/1902.08318 (2019). arXiv: `1902.08318`. cit. on p. 105

[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. cit. on p. 43

[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 pp. 14, 16

[Nonius] [SW], *Nonius*, LIC: Apache. VCS: https://github.com/libnonius/nonius. cit. on p. 105

[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. cit. on p. 108

[OpenBLAS] [SW], *OpenBLAS*, LIC: BSD. VCS: https://github.com/xianyi/OpenBLAs. cit. on p. 104

[OpenMPI] [SW], *OpenMPI*, LIC: BSD. VCS: https://github.com/open-mpi/ompi. cit. on p. 104

[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. 19, 108

[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. cit. on pp. 7, 82

[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). arXiv: 1905.12468. cit. on p. 98

[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. cit. on pp. 14, 16

[SQLite] [SW], *SQLite*, LIC: Public domain. VCS: https://github.com/sqlite/sqlite. cit. on p. 114

[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. cit. on p. 106

[Stress-ng] [SW], *Stress-ng*, LIC: GPL 2.0. VCS: https://github.com/ColinIanKing/stress-ng. cit. on pp. 108, 130

[top500] *TOP500 Website*. URL: https://www.top500.org/ (visited on Sept. 7, 2020). cit. on p. 6

[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. cit. on p. 107

[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. cit. on p. 17

[Wik21a] Wikipedia. *Microprocessor Chronology*. 2021. URL: `https://en.wikipedia.org/wiki/Microprocessor_chronology` (visited on Feb. 7, 2021). cit. on p. 5

[Wik21b] Wikipedia. *Transistor count*. 2021. URL: `https://en.wikipedia.org/wiki/Transistor_count` (visited on Feb. 7, 2021). cit. on p. 5

[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. 15

[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 pp. 14, 16

List of Figures

List of Tables

Abstract

The English abstract.

Résumé

Le résumé en français.