UGA
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

To introduce my work, I will write a nice introduction in the following. Citation example for the Top500 website [@top500] and some random paper [Gra69].

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.

Part I

Performance prediction through simulation: the HPL case

The work presented in this part has been published at a conference[CLH19] and has been submitted for publication in a journal. 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.

High Performance Linpack

1.1 The benchmark
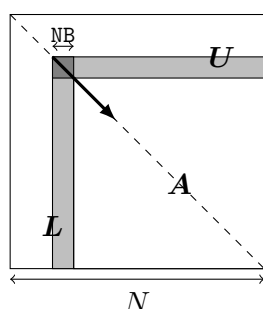

```
allocate and initialize A;
for k = N to 0 step NB do
    allocate the panel;
    factor the panel;
    broadcast the panel;
    update the sub-matrix;
end
```

Figure 1.1.: Overview of High Performance Linpack

In this work, we use the freely-available reference-implementation of HPL[Pet+16], 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 Figure1.1. 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.

1.2 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 Table 1.1). 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 1.1.: 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.

Related work

2

2.1 Performance prediction

A first approach for estimating the performance of applications like HPL is statistical modeling of the application as a whole[LD12]. By running the application several times with small and medium problem sizes (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. Furthermore, this approach does not allow 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 (out of several thousands) of HPL typically reaches 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 the case for HPL.

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 HPL. 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 HPL 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 HPL 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].

2.2 Simgrid/SMPI

SimGrid[Cas+14] is a flexible and open-source simulation framework that was originally designed in 2000 to study scheduling heuristics tailored to heterogeneous grid computing environments but has later been extended to study cloud and HPC infrastructures. The main development goal for SimGrid has been to provide validated performance models particularly for scenarios making heavy use of the network. 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 un-modified 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.

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] ($32\,\text{GB}$ 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

```c
#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.
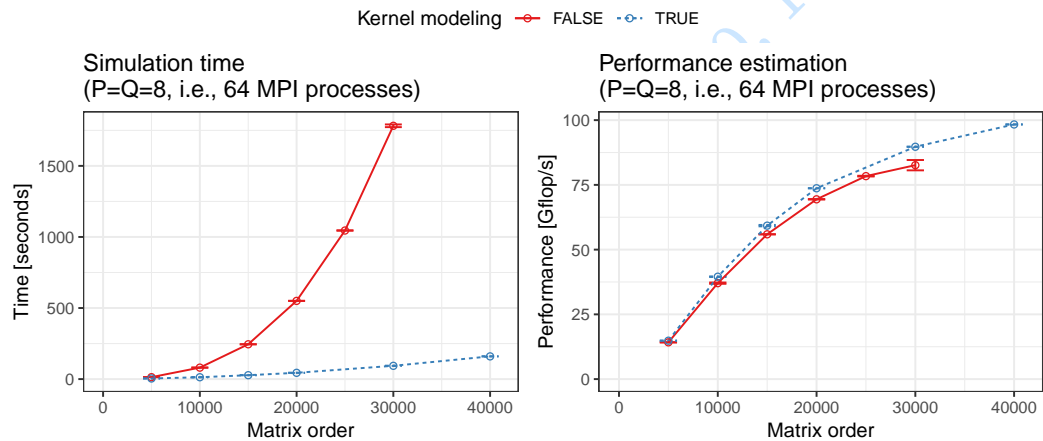
(b) Gain in terms of simulation time.

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).

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 Figure 3.2): 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 `int`s (accounting for matrix indices, error codes, MPI tags, and pivoting information) and `double`s (corresponding to a copy of a sub-matrix of A). To optimize data transfers, HPL flattens this structure into a single allocation of `double`s (see Figure 3.3(a)). Using a fully shared memory allocation for the `panel` therefore leads to index corruption that results in classic invalid memory accesses. Since `int`s and `double`s 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).
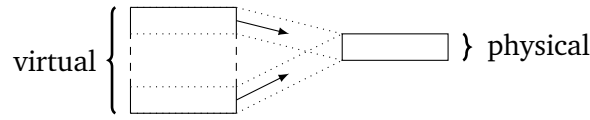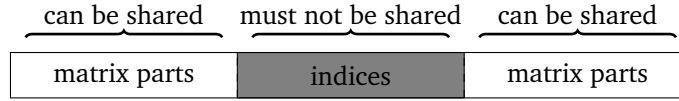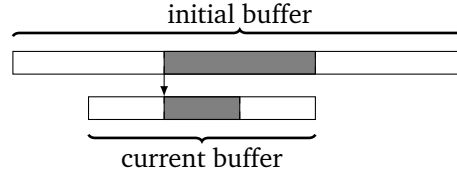
Figure 3.2.: SMPI shared malloc mechanism: large area of virtual memory are mapped onto the same physical pages.

(a) Structure of the panel in HPL.

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

Figure 3.3.: Panel structure and allocation strategy.

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 $0.488\,\mathrm{GB}$.

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

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

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 Figure 3.4. 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.

Modeling HPL kernels and communications

<div align="right">

4

</div>

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. $T \sim \alpha.M.N.K + \beta.M.N + \gamma.N.P + \ldots$ Again, the $\alpha, \beta, \gamma, \ldots$ 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,

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

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 $\mathcal{N}_H$.
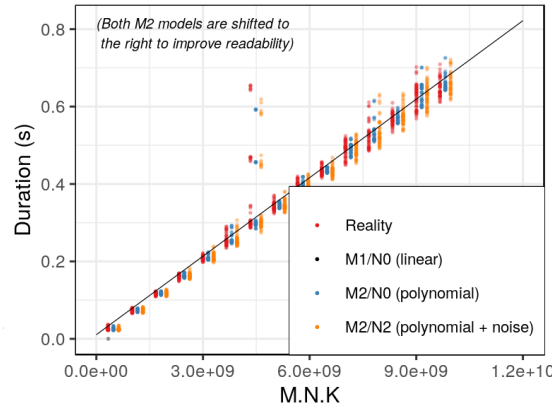
4.2 Modeling the CPU (i.e. `dgemm` function)

4.2.1 Different linear models

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. Figure 4.1 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 (Figure 4.1(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. Figure 4.1(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

(a) `dgemm` heterogeneity

(b) `dgemm` model

Figure 4.1.: Illustrating the realism of modeling for `dgemm` function

well explained by both the polynomial model (better fit for particular parameter combinations) and some temporal variability.

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 Figure 4.2. 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

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

`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 Figure 4.2 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

Some text...

4.2.3 Conclusion

Some text...

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 Figure 4.3, the variability of high speed networks is quite particular. We therefore

diligently estimated all the parameters of $\mathcal{M}'-1\,\mathcal{N}'-1$, where each message size range is automatically estimated with `pytree`, as well as the 2 to 4 modes of the 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.
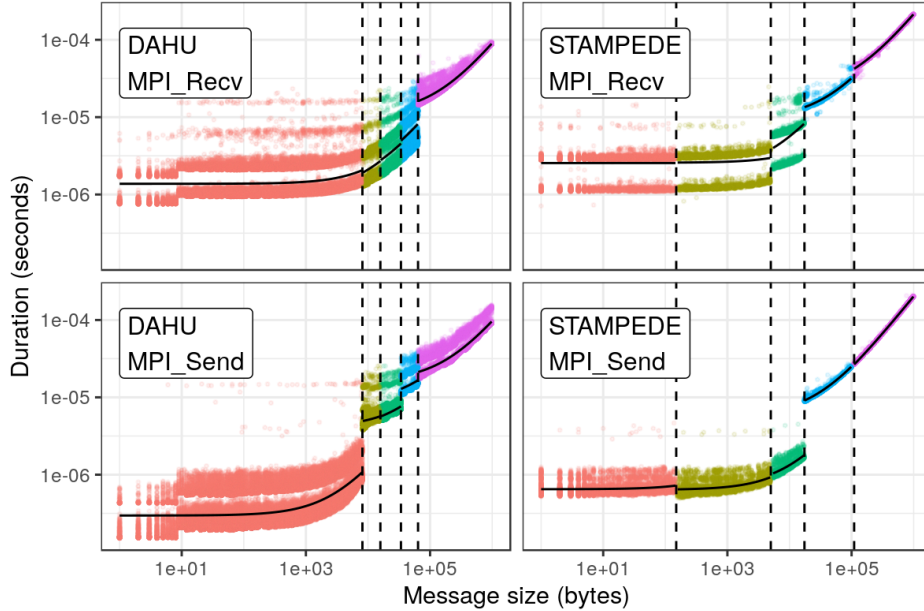

Figure 4.3.: Illustrating piecewise linearity and temporal variability of high-speed communications on two systems.

4.3.2 Learning breakpoints

Some text...

4.4 Conclusion

Some text...

5 Validation

5.1 Different problem sizes

Some text. . .

5.2 A platform change

Some text. . .

5.3 Factorial experiment

Some text. . .

5.4 Different geometries

Some text. . .

5.5 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

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 dozen 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 extremly 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 a lot of 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 softwares [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 a lot of low-level constructs that are really unpleasant and unintuitive to write and read. Section 7.2.2 will present some comparisons. Furthermore, Execo lacks a lot of important features, like node instrumentation and metadata collection, i.e. we would still have needed to implement a lot functionnalities 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 `peanut.Job`. Three methods can be overriden, `setup`, `run_exp` and `teardown`.

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

- Implicitely, submit a job with the given characteristics (e.g. cluster, number of nodes, walltime, etc), then deploy the given OS image.

- Implicitely (but optionnaly) enable or disable some performance functionnalities like hyperthreading, turboboost, C-states.

- Implicitely (but optionnaly) 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).

- Implicitely (but optionnaly) 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 explicitely add any file to the archive. In addition, the following content is also implicitely 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 `stdin` 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 functionnalities described previously can also be explicitely called (e.g. there are methods `disable_hyperthreading`, `start_monitoring` and `perform_stress`).

An experiment can be parametrized by two means:

- An install 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 softwares like OpenBLAS or OpenMPI, but also the duration of the warmup 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`

```python
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'
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")
```

---

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

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`

```python
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.

Randomizing matters!

<div style="text-align: right">8</div>

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 carefuly 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.

8.2 Randomizing the sizes

Some text. . .

8.3 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

---

[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`

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.

8.3.1 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 8.1 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.
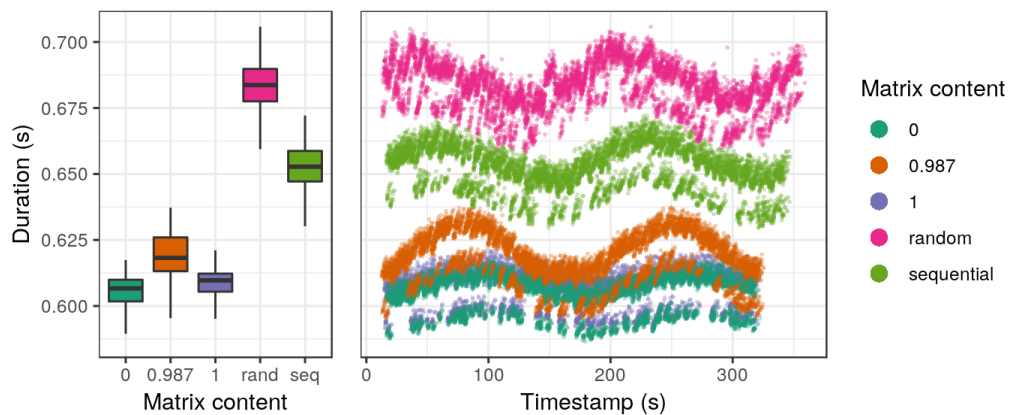

Figure 8.1.: DGEMM durations are lower with constant values in the matrices

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

The observations we have made on `dgemm` performance can be explained by Figure 8.2 which shows the evolution and the distribution of the core frequencies during the experiment. There is a clear correlation between the frequencies and `dgemm` performance: the random initialization produces lower frequencies whereas the constant initialization gives higher frequencies. A similar temporal patterns can also be distinguished with clear oscillations.
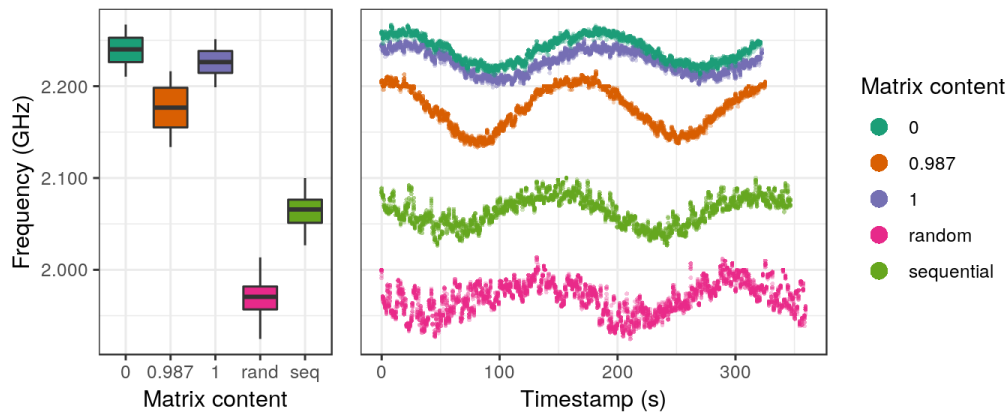

Figure 8.2.: Core frequencies are higher with constant values in the matrices

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. Four other clusters show a similar behavior, the performance of `dgemm` is higher when the matrices are generated with a constant value. However, for four other clusters, this phenomenon could not be observed, the matrix content had no impact on the performance.

8.3.2 Hypotheses

Several hypotheses were discussed to explain this unexpected phenomenon.

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

Table 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

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. 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.

8.3.3 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 8.3, 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

---

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

mask sizes have been tested, from 0 (the elements are left unchanged) to 53 (the mantissa becomes completely deterministic, all the elements are equal).
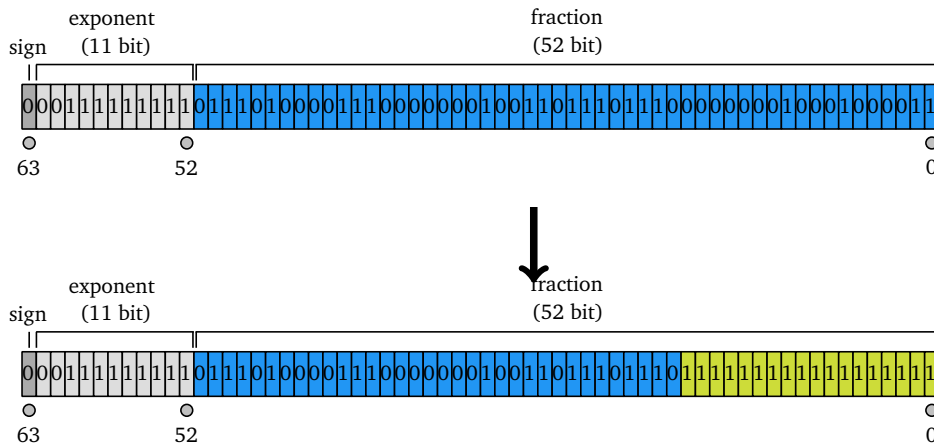

Figure 8.3.: Illustrating the effect of applying a mask on the random part of the matrix elements

The evolution and the distribution of the `dgemm` durations is plotted in Figure 8.4. Their is a very clear correlation between the mask size and the performance: the larger the mask, the lower the duration. Similarly to the previous experiment, some temporal patterns can also be distinguished.
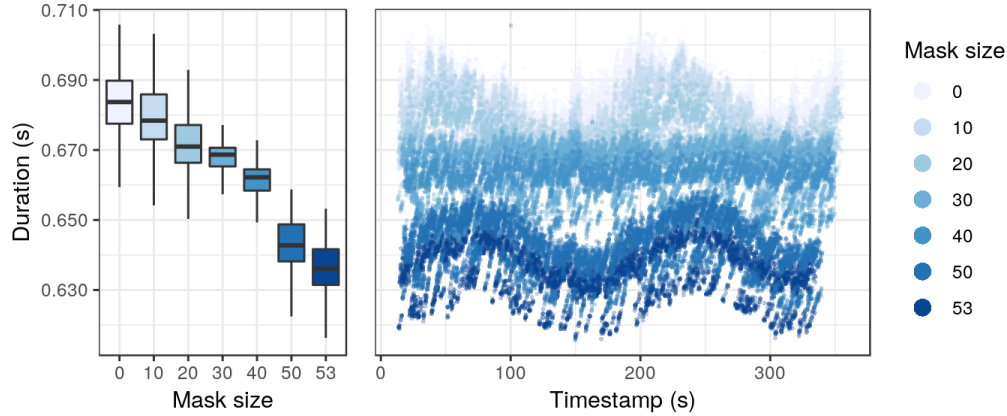

Figure 8.4.: DGEMM durations are lower with larger bit masks

This correlation with the mask size can also be seen with the frequencies in Figure 8.5: 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.
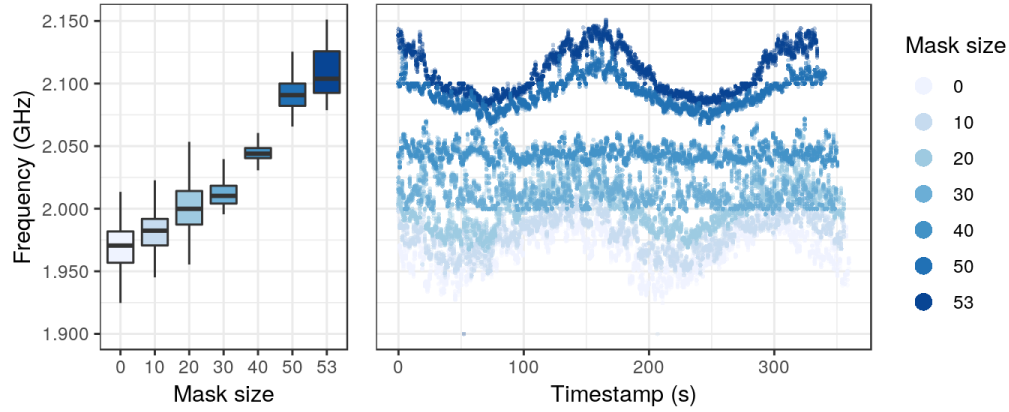
Figure 8.5.: Core frequencies are higher with larger bit masks

8.3.4 Conclusion

We have shown that the performance of the `dgemm` function is data-dependent. The best explanation we have for this counter-intuitive fact is an energy cost overhead caused by bit flips inside the processor. To respect its energy budget, the CPU has to throttle more aggressively its frequency when the matrices content is more diverse and thus more energy consumming. 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.

8.3.5 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 is 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 anoter `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 a 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 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 controler) 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.

8.4 Beware of extrapolations

Some text. . .

---

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

8.5 Beware of experimental conditions

Some text. . .

8.6 Conclusion

Some text. . .

9 Performance non-regression tests

9.1 State of the art

Some text...

9.2 Performance measure and information collection

Some text...

9.3 Statistical test

Some text...

9.4 Conclusion

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

[@top500]   *TOP500 Website*. URL: https://www.top500.org/ (visited on Sept. 7, 2020).
cit. on p. 1

[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. 46

[Bad+03]   Rosa M. Badia, Jesús Labarta, Judit Giménez, and Francesc Escalé. "Dimemas: Predicting MPI Applications Behaviour in Grid Environments". In: *Proc. of the Workshop on Grid Applications and Programming Tools*. June 2003.   cit. on p. 11

[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. 15, 33

[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. 12

[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. 34

[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 p. 12

[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. 11

[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. 40

[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. 5

[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. 12

[Cor17]   Tom Cornebize. "Capacity Planning of Supercomputers: Simulating MPI Applications at Scale". MA thesis. Grenoble INP ; Université Grenoble - Alpes, June 2017.   cit. on p. 5

[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. 12, 13, 19

[Eng14]   Christian Engelmann. "Scaling To A Million Cores And Beyond: Using Light-Weight Simulation to Understand The Challenges Ahead On The Road To Exascale". In: *FGCS* 30 (Jan. 2014), pp. 59–65.   cit. on p. 12

[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. 12

[Gra69]   Ronald L. Graham. "Bounds on multiprocessing timing anomalies". In: *SIAM journal on Applied Mathematics* 17.2 (1969), pp. 416–429.   cit. on p. 1

[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 p. 13

[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. 34

[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 p. 12

[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 p. 11

[Mub+16]   M. Mubarak, C. D. Carothers, Robert B. Ross, and Philip H. Carns. "Enabling Parallel Simulation of Large-Scale HPC Network Systems". In: *IEEE Transactions on Parallel and Distributed Systems* (2016).   cit. on p. 11

[Pet+16]   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*. http://www.netlib.org/benchmark/hpl. Version 2.2. Feb. 2016.   cit. on p. 7

[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. 46

[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. 13

[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. 12

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

         cit. on p. 11

DRAFT

12th November 2020, 13:01:54

List of Figures

List of Tables

Abstract

The English abstract.

Résumé

Le résumé en français.