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

Acknowledgments v

Abstract / Résumé vii

Contents ix

1 Introduction 1

2 Performance prediction through simulation: the HPL case 3
2.1 High Performance Linpack . . . 3
2.1.1 The benchmark . . . 3
2.1.2 Typical runs on a supercomputer . . . 5
2.2 Related work . . . 6
2.2.1 Performance prediction . . . 6
2.2.2 Simgrid/SMPI . . . 7
2.3 Emulating HPL at large scale . . . 9
2.3.1 Speeding Up the Emulation . . . 9
2.3.2 Scaling Down Memory Consumption . . . 11
2.3.3 Scalability Evaluation . . . 13
2.4 Modeling HPL kernels and communications . . . 14
2.5 Validation . . . 15
2.6 Sensibility analysis . . . 15

3 Experimental control 17
3.1 Experimental Testbed and Experiment Engines . . . 17
3.2 Randomizing matters! . . . 17
3.3 Performance non-regression tests . . . 17

4 Conclusion 19

Bibliography A1

List of Figures A3

List of Tables A3

1 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].

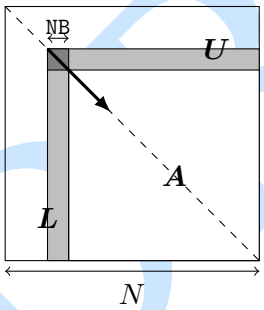
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus 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.

2 Performance prediction through simulation: the HPL case

The work presented in this chapter has been published at a conference[CLH19] and has been submitted for publication in a journal. The content of this chapter is therefore a near-verbatim copy of these articles. This work also directly follows my master thesis[Cor17] whose main contribution is summarized in section2.3 for the sake of completeness.

2.1 High Performance Linpack

2.1.1 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 2.1: 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 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.

2.1.2 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 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.

2.2 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 Section2.3 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].

2.2.2 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.

2.3 Emulating HPL at large scale

In this section, I 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] ($32\,\mathrm{GB}$ of RAM, two 8-core Intel Xeon E5-2620 v4 CPUs, Debian Stretch OS (Linux 4.9)).

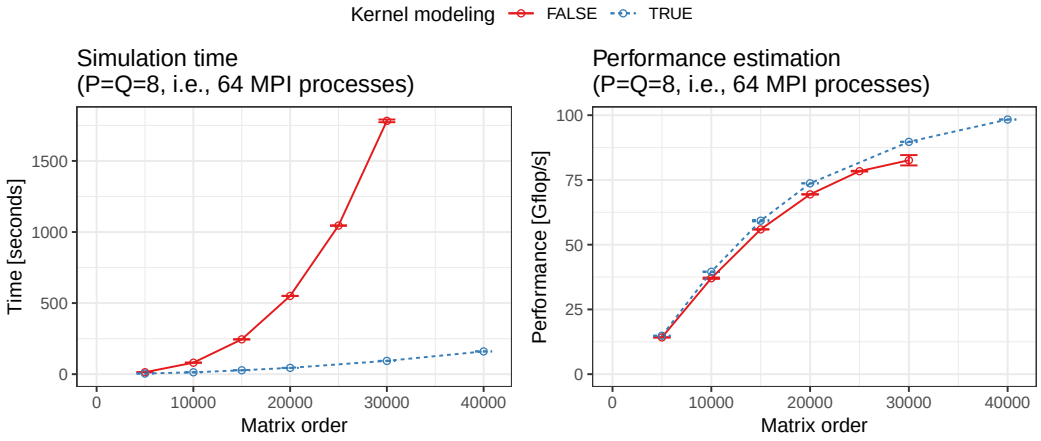
2.3.1 Speeding Up the Emulation

Compute Kernel Modeling

HPL heavily relies on BLAS kernels such as `dgemm` (for matrix-matrix multiplication) or `dtrsm` (for solving an $A \cdot x = b$ equation). The analysis of an HPL 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. Figure2.2(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 Figure2.2(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 sections but it can already be observed that this simple

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

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.

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).

2.3.2 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 Figure2.3): 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

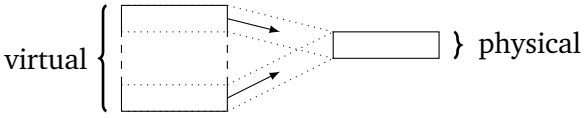
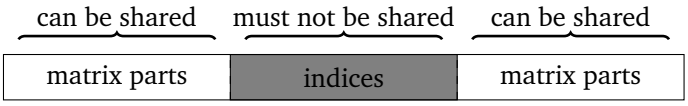
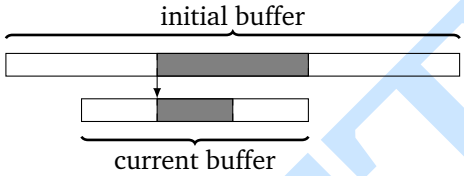


Figure 2.3: SMPI shared malloc mechanism: large area of virtual memory are mapped onto the same physical pages.



(a) Structure of the panel in HPL.



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

Figure 2.4: Panel structure and allocation strategy.

`doubles` (see Figure2.4(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 Figure2.4(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 $0.488\,\mathrm{GB}$.

2.3.3 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

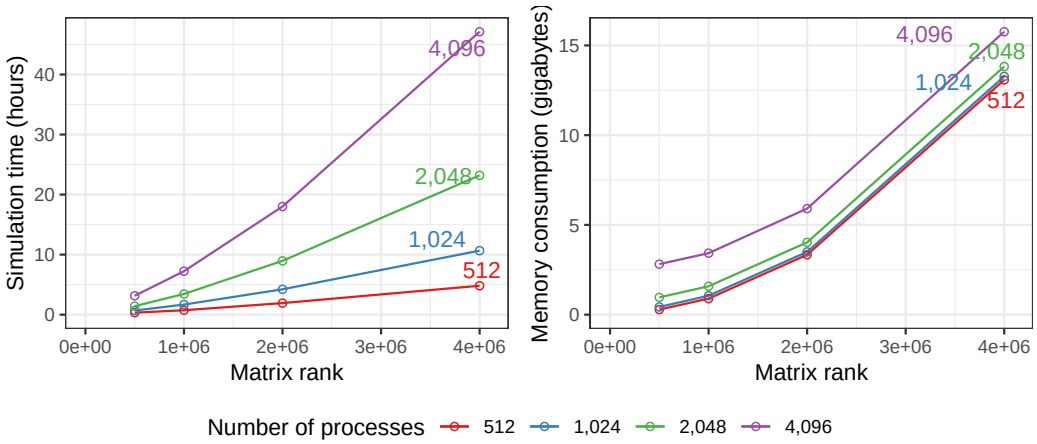


Figure 2.5: Time complexity and memory consumption are linear in the number of processes but remain mildly quadratic with matrix rank.

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 Figure2.5. 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.

2.4 Modeling HPL kernels and communications

some text...

2.5 Validation

some text...

2.6 Sensibility analysis

some text...

3 Experimental control

3.1 Experimental Testbed and Experiment Engines

some text...

3.2 Randomizing matters!

some text...

3.3 Performance non-regression tests

some text...

4 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

[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. 7

[Bal+13] Daniel Balouek, Alexandra Carpen-Amarie, Ghislain Charrier, et al. "Adding Virtualization Capabilities to the Grid'5000 Testbed". In: *Cloud Computing and Services Science*. Ed. by IvanI. Ivanov, Marten Sinderen, Frank Leymann, and Tony Shan. Vol. 367. Communications in Computer and Information Science. Springer International Publishing, 2013. cit. on p. 9

[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. 7

[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. 7

[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. 7

[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. 3

[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. 7

[Cor17] Tom Cornebize. "Capacity Planning of Supercomputers: Simulating MPI Applications at Scale". MA thesis. Grenoble INP ; Université Grenoble - Alpes, June 2017. cit. on p. 3

[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. 8, 13

[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. 7

[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. 7

[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. 8

[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. 7

[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. 6

[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. 7

[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. 3

[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. 8

[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. 7

[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. 7

List of Figures

2.1 Overview of High Performance Linpack 3
2.2 Replacing the calls to computationally expensive functions by a model allows to emulate HPL at a larger scale. 10
2.3 SMPI shared malloc mechanism: large area of virtual memory are mapped onto the same physical pages. 12
2.4 Panel structure and allocation strategy. 12
2.5 Time complexity and memory consumption are linear in the number of processes but remain mildly quadratic with matrix rank. 14

List of Tables

2.1 Typical runs of HPL 5

Abstract

The English abstract.

Résumé

Le résumé en français.