Faithful and Efficient Simulation of High Performance Linpack

Tom Cornebize, Arnaud Legrand, Franz Heinrich

Faithful and Efficient Simulation of High Performance Linpack

Tom Cornebize, Arnaud Legrand, Franz C. Heinrich
Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG
Grenoble, France

ABSTRACT

With a power consumption of several MW per hour on a TOP500 machine, running applications on supercomputers at scale solely to optimize their performance is extremely expensive. Likewise, High-Performance Linpack (HPL), the benchmark used to rank supercomputers in the TOP500, requires a careful tuning of many parameters (problem size, grid arrangement, granularity, collective operation algorithms, etc.) and supports exploration of the most common and fundamental performance issues and their solutions. In this article, we explain how we both extended the SimGrid's SMPI simulator and slightly modified the open-source version of HPL to allow a fast emulation on a single commodity server at the scale of a supercomputer. We explain how to model the different components (network, BLAS, …) and show that a careful modeling of both spatial and temporal node variability allows us to obtain predictions within a few percents of real experiments.

1 INTRODUCTION

Today, machines with 100,000 cores and more are common and several machines beyond the 1,000,000 cores mark are already in production. This high density of computation units requires a diligent optimization of application parameters, such as problem size, process organization or choice of algorithm, as these have a huge impact on load distribution and network utilization. Scientific application developers and users often spend a considerable amount of time and effort running their application at different scales solely to tweak parameters and therefore optimize their performance. This expenditure of time, combined with the power consumption that often reaches several MW for TOP500 machines, makes it financially expensive to test-run applications. Similar difficulties are encountered when (co-)designing supercomputers for specific applications. A large part of this tuning work could be simplified if a generic and faithful performance prediction tool was available. This article presents a decisive step in this direction.

The world's largest and fastest machines are ranked twice a year in the TOP500 [1] list using the High-Performance Linpack (HPL) benchmark. To get near to the peak performance, a sizable input is needed and it typically takes several hours to run HPL on the list's number one system. HPL implements several algorithmic optimizations that should be carefully tuned and it is thus quite representative of many other scientific applications in terms of potential performance issues and their solutions. In the case of HPL, machine vendors know how to estimate the performance by using simple mathematical models. However, these models only give trends but fail to capture the impact of the network congestion and of system noise and are thus not particularly accurate. Whenever actual performance does not match expectations, it can be very difficult to understand whether the mismatch originates from model inaccuracy or machine misconfiguration.

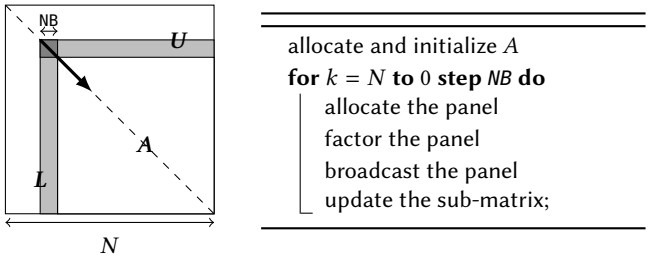


Figure 1: Overview of High Performance Linpack

In this article, we explain how to predict the performance of HPL on a supercomputer with the SimGrid/SMPI [2, 3] simulator. We detail how we obtained faithful models for several key functions (e.g., `dgemm` and `dtrsm`) and managed to reduce the memory consumption from more than a hundred terabytes to several gigabytes. This allowed us to emulate on a single commodity server within 1-2 days a scenario similar to the HPL run that was carried out on the Stampede cluster (hosted at TACC) in 2013 for the TOP500. More importantly, we managed to predict the performance of HPL on a recent cluster (running a thousand MPI ranks) within a few percent of reality.

This article is organized as follows: Section 2 presents the main characteristics of the HPL application and provides details on the run that was conducted at TACC in 2013. Section 3 discusses related work and explains why emulation (or *online simulation*) is the only possible approach when studying an application as complex as HPL. In Section 4, we briefly present the simulator we used for this work, SimGrid/SMPI, followed by an extensive discussion in Section 5 about the optimizations on all levels (i.e., simulator, application, operating system) that were necessary to make a large-scale run tractable. The scalability of our approach is evaluated in Section 5.3. In Section 6, we explain how the different resources of the supercomputer (network, compute kernels) are modeled to predict the performance of HPL faithfully. We particularly discuss the usefulness and relevance of different degrees of modeling complexity. In Section 7, we compare simulation results with real experiments and illustrate the importance of modeling both spatial and temporal variability. Section 8 concludes this article by discussing perspectives and future work.

2 CONTEXT

2.1 High-Performance Linpack

In this work, we use the freely-available reference-implementation of HPL [4], 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 Figure 1. It consists of a series of panel factorizations followed by an update of the trailing sub-matrix. HPL uses a two-dimensional block-cyclic data

distribution of A and implements several custom MPI collective communication algorithms to efficiently overlap communications with computations. The main parameters of HPL are:

- N is the order of the square matrix A.
- `NB` is the "blocking factor", i.e., the granularity at which HPL operates when panels are distributed or worked on.
- P and Q denote the number of process rows and the number of process columns, respectively.
- `RFACT` determines the panel factorization algorithm. Possible values are Crout, left- or right-looking.
- `SWAP` specifies the swapping algorithm used while pivoting. Two algorithms are available: one based on *binary exchange* (along a virtual tree topology) and the other one based on a *spread-and-roll* (with a higher number of parallel communications). HPL also provides a panel-size threshold triggering a switch from one variant to the other.
- `BCAST` sets the algorithm used to broadcast a panel of columns over the process columns. Legacy versions of the MPI standard only supported non-blocking point-to-point communications, which is why HPL ships with in total 6 self-implemented variants to overlap the time spent waiting for an incoming panel with updates to the trailing matrix: `ring`, `ring-modified`, `2-ring`, `2-ring-modified`, `long`, and `long-modified`. The `modified` versions guarantee that the process right after the root (i.e., the process that will become the root in the next iteration) receives data first and does not further participate in the broadcast. This process can thereby start working on the panel as soon as possible. The `ring` and `2-ring` versions each broadcast along the corresponding virtual topologies while the `long` version is a *spread and roll* algorithm where messages are chopped into Q pieces. This generally leads to better bandwidth exploitation. The `ring` and `2-ring` variants rely on `MPI_Iprobe`, meaning they return control if no message has been fully received yet, hence facilitating partial overlap of communication with computations. In HPL 2.1 and 2.2, this capability has been deactivated for the `long` and `long-modified` algorithms. A comment in the source code states that some machines apparently get stuck when there are too many ongoing messages.
- `DEPTH` controls how many iterations of the outer loop can overlap with each other.

The sequential complexity of this factorization is $\text{flop}(N) = \frac{2}{3}N^3 + 2N^2 + \mathcal{O}(N)$ where N is the order of the matrix to factorize. The time complexity can be approximated by

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

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

2.2 Typical Runs on a Supercomputer

Although the TOP500 reports precise information about the core count, the peak performance and the effective performance, it provides almost no information on how (software versions, HPL parameters, etc.) this performance was achieved. Some colleagues agreed to provide us with the HPL configuration they used and the output they submitted for ranking. In June 2013, the Stampede supercomputer at TACC was ranked 6th in the TOP500 by achieving 5168.1 TFlop s^{-1}. In November 2017, the Theta supercomputer at ANL was ranked 18th with a performance of 5884.6 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 that we could fully control. Table 1 shows the parameters that were used for each of these runs.

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 as presented in Section 2.1) 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. Our goal in this article is to simulate systems at the scale of Stampede. Given the scale of this scenario (3,785 steps on 6,006 nodes in two hours), detailed simulations quickly become intractable without significant effort.

3 RELATED WORK

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

Table 1: Typical runs of HPL

	Stampede@TACC	Theta@ANL	Dahu@G5K
`Rpeak`	8520.1 TFlop s^{-1}	9627.2 TFlop s^{-1}	62.26 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 TFlop s^{-1}	5884.6 TFlop s^{-1}	24.55 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 underlying platform can replay the trace on commodity hardware at will with different platform models. Most HPC simulators available today, notably BigSim [5], Dimemas [6] and CODES [7], 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 [8]. Even worse, tracing an application provides only information about its behavior at the time of the run. Even slight modifications (e.g., to communication patterns) may make the trace inaccurate. It is possible for simple applications (e.g., `stencil`) to extrapolate behavior from small-scale traces [9, 10] but this fails if the execution is non-deterministic, i.e., 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 [11], SimGrid/SMPI [2] and the closed-source xSim [12] support it. To the best of our knowledge, only SST Macro and SimGrid/SMPI are mature enough to faithfully emulate HPL. In this work, we decided to rely on SimGrid as its performance models and its emulation capabilities seemed quite solid but the developments we propose would a priori also be possible with SST.

4 SIMGRID/SMPI IN A NUTSHELL

SimGrid [2] 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 [3]. 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 or network heterogeneity [13]). If needed, communications that start or end trigger a re-computation of the bandwidth share. In this 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. 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 [14] and up to 128×1 cores [3]). In this article, we 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 6,006 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.

5 EMULATING HPL AT LARGE SCALE

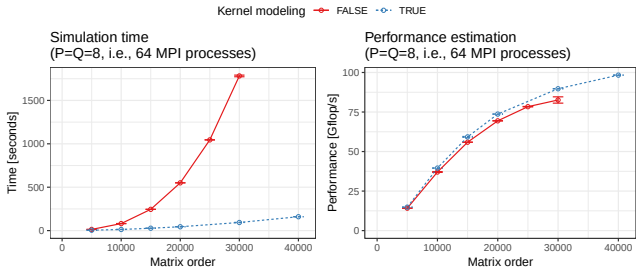
We now present the changes to SimGrid and HPL that were required for a scalable simulation. We provide only a brief presentation of our modifications and refer the reader interested in details to [15] and [16]. For our experiments in this section, we used a single core from nodes of the Nova cluster provided by the Grid'5000 testbed [17] with 32 GB of RAM, two 8-core Intel Xeon E5-2620 v4 CPUs processors and a Debian Stretch OS (Linux 4.9).

5.1 Speeding Up the Emulation

5.1.1 Compute Kernel Modeling. HPL heavily relies on BLAS kernels such as `dgemm` (for matrix-matrix multiplication) or `dtrsm` (for solving an $A \cdot x = b$ equation). The analysis of an HPL simulation with 64 processes and a very small matrix of order 30,000 showed that $\approx 96\,\%$ 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. Figure 2(a) shows an example of this macro-based mechanism that allows us 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

```
#define HPL_dgemm(layout, TransA, TransB,                          \
        M, N, K, alpha, A, lda, B, ldb, beta, C, ldc) ({           \
    double expected_time = (1.029e-11)*((double)M)*                \
                    ((double)N)*((double)K) + 1.981e-12;           \
    if(expected_time > 0)  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: Replacing the calls to computationally expensive functions by a model allows to emulate HPL at a larger scale.

scenario is depicted in Figure 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 kernel model leads to a sound, albeit slightly more optimistic, estimation of the performance.

In addition to the main compute kernels, we identified seven other BLAS functions through profiling 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. To study their impact on the overall performance, we handled them 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.

5.1.2 Specific Adjustments of HPL. 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 we ignored the cost of this function as well, we set its return value to a random (but controlled) value to make the simulation unbiased (but fully deterministic). We confirmed that this modification was harmless in terms of performance prediction.

5.2 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): a single block of physical memory (of default size 1 MB) for the whole execution is allocated and shared by all MPI processes. A range of virtual addresses corresponding to a specified size is reserved and cyclically mapped onto the previously obtained physical address. This mechanism allows most applications to obtain a nearly constant memory footprint, regardless of the size of the actual allocations.

Although using the default `SHARED_MALLOC` mechanism works flawlessly for `A`, a more careful strategy needs to be used for the `panel`, which is an intricate data structure with both `ints` (accounting for matrix indices, error codes, MPI tags, and pivoting information) and `doubles` (corresponding to a copy of a sub-matrix of `A`). To optimize data transfers, HPL flattens this structure into a single allocation of `doubles` (see Figure 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 GB to less than 40 MB.

Another HPL specific optimization is related to the systematic allocation and deallocation of panels in each iteration, with the size of the panel strictly decreasing from iteration to iteration. As we 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 to allocate only the first panels and reuse them as much as possible for subsequent iterations (see Figure 4(b)). Even for a very small matrix of order 40,000 and 64 MPI processes, the simulation time decreases

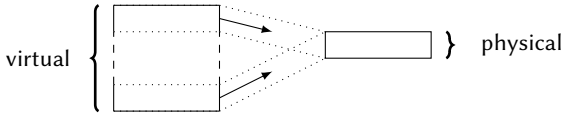
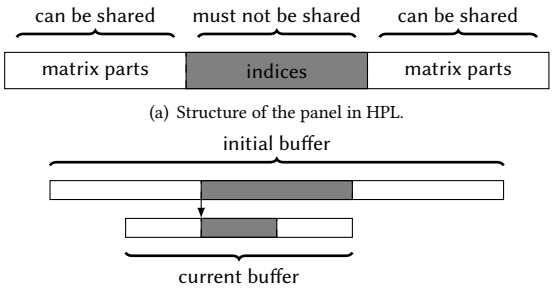


Figure 3: SMPI shared malloc mechanism: large area of virtual memory are mapped onto the same physical pages.



(a) Structure of the panel in HPL.

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

Figure 4: Panel structure and allocation strategy.

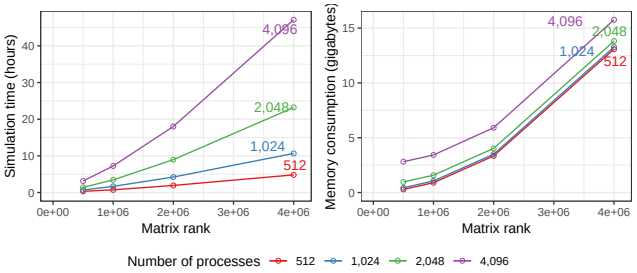


Figure 5: Time complexity and memory consumption are linear in the number of processes but remain mildly quadratic with matrix rank.

from 20.5 sec to 16.5 sec. The number of page faults decreased from 2 million to 0.2 million, confirming the devastating effect these allocations/deallocations would have at scale.

The last three optimizations we describe 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 [3]. 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 sec (out of 51.47 sec) to 2.12 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 MiB (instead of 4 KiB), which reduces drastically the page table size. For example, for a matrix of order $N = 4,000,000$, it shrinks from 250 GB to 0.488 GB.

5.3 Scalability Evaluation

The main goal of the previous optimizations is to reduce the complexity from $\Theta(N^3) + \Theta(N^2 \cdot P \cdot Q)$ to something more reasonable. The $\Theta(N^3)$ was removed by skipping most computations. Ideally, since there are N/NB iterations (steps), the complexity of simulating one step should be decreased to something independent of N. SimGrid's fluid models, used to simulate communications, do not depend on N. Therefore, the time to simulate a step of HPL should mostly depend on P and Q. Yet, some memory operations on the panel that are related to pivoting are intertwined in HPL with collective communications, meaning that it is impossible to get rid of the $O(N)$ 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 5. With all previously described optimizations enabled, the longest simulation took close to 47 hours and consumed 16 GB of memory whereas the shortest one took 20 minutes and 282 MB of memory.

6 MODELING HPL KERNELS AND COMMUNICATIONS

As explained in Section 5, 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 (see Figure 6). In this section 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. We finally show in Section 7 that using simpler modeling options leads to a serious inaccuracy of the simulation.

6.1 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.
- 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:

- The platform may be slightly heterogeneous (spatial variability) and the previous models should then be built for each host individually. We denote this modeling option as $\mathcal{M}_H$.
- 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, 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. We denote this kind of model as $\mathcal{M}'$.

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 simply 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 $\mathcal{N}_H$.

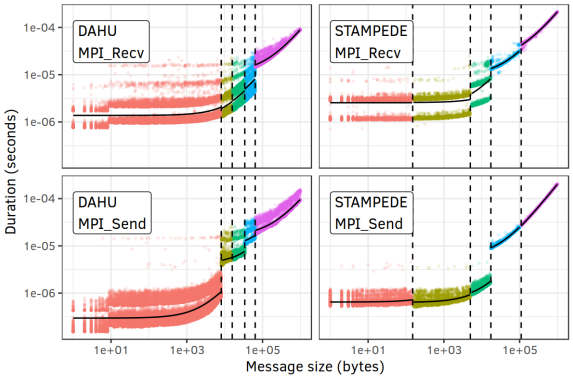


Figure 6: Illustrating piecewise linearity and temporal variability of high-speed communications on two systems.

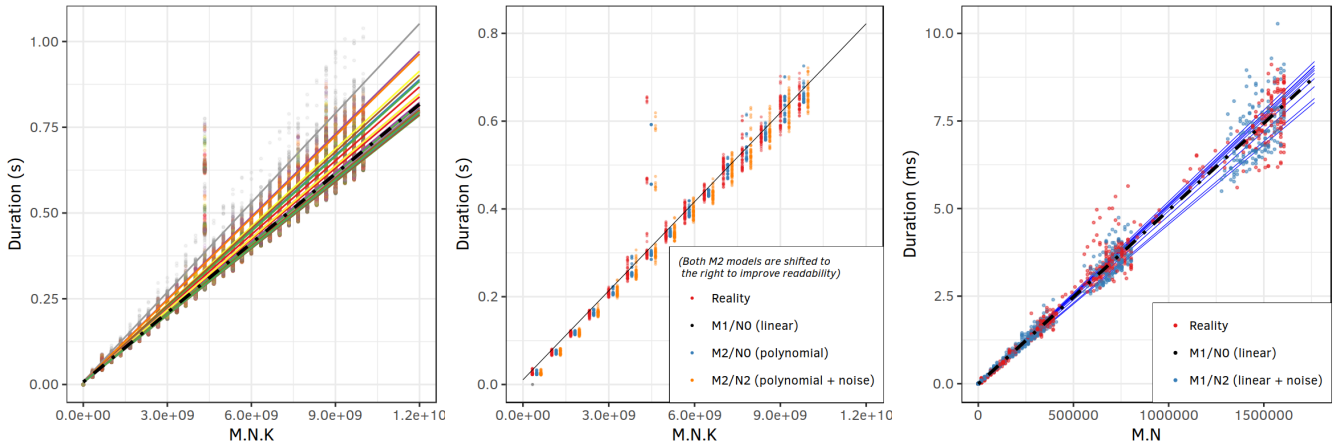
Linear regression is a standard tool in R [18] or in python/statsmodels [19] and models $\mathcal{M}_H-0, 1, 2$ are thus easy to fit assuming $\mathcal{N}-1$. Model $\mathcal{M}'-0$ (piece-wise constant) assuming $\mathcal{N}-1$ can easily be fit using the `cubist` [20] library. We could not find any implementation allowing to fit $\mathcal{M}'-1$ (piece-wise linear, possibly discontinuous) assuming $\mathcal{N}-1$ so we implemented a method based on model trees [21, Chapter 9] [22] in a python library (`pytree`). When the noise is heteroscedastic ($\mathcal{N}-2$), it is sometimes possible to fall back to the previous methods if a careful sampling method is employed (by sampling more on highly variable areas and fitting the average). Finally, for even more complex noise modeling options ($\mathcal{N}'-$), some custom Expectation Maximization algorithms are available (e.g., `flexmix` [23] for $\mathcal{M}-1$ $\mathcal{N}'-1$).

6.2 Modeling MPI communications

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 6, 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.

6.3 Modeling `dgemm`

HPL spends the most time in the `dgemm` kernel. We therefore 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 7 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



(a) `dgemm` heterogeneity (b) `dgemm` model (c) `HPL_dlatcpy` model

Figure 7: Illustrating the realism of modeling for BLAS and HPL functions.

experimental conditions similar to the one of HPL). The first observation (Figure 7(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 7(b) depicts the performance (red dots) of a given node as well as the prediction using a simple linear model ($\mathcal{M}_H - 1$, black line), a full polynomial model ($\mathcal{M}_H - 2$, blue dots) and a full polynomial model with heteroscedastic noise ($\mathcal{M}_H - 2\ \mathcal{N}_H - 2$, orange dots). A close inspection reveals that all experimental variability is actually very well explained by both the polynomial model (better fit for particular parameter combinations) and some temporal variability.

6.4 Modeling Other BLAS and HPL Kernels

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 7(c). This kind of data can only be obtained by running HPL for a small input matrix over each node individually. Again, for all these kernels a single parameter combination explains most of the performance and there is some variability from one node to another (one blue regression line per CPU) but it remains quite limited (black dotted line for the platform as a whole), especially since these kernels are very short and infrequently called compared to `dgemm`. Finally, since variability significantly increases with the value of the input parameters, a $\mathcal{N} - 2$ model is clearly required. The blue dots in Figure 7(c) 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.

7 VALIDATION AT SCALE

7.1 Experimental Setup

To evaluate the soundness of our approach, we compare several real executions of HPL with simulations using the previous models. We used the Dahu cluster from the Grid'5000 testbed. It has 32 nodes connected through a single switch by 100 Gbit s^{-1} Omnipath links. Each node has two Intel Xeon Gold 6130 CPU with 16 cores per CPU. We disabled hyperthreading and fixed the frequency to 3.7 GHz. We used HPL version 2.2 compiled with GCC version 6.3.0. We also used the libraries OpenMPI version 2.0.2 and OpenBLAS version 0.3.1. HPL executions were done using a block size of 128 and a matrix of varying size (from 50,000 to 500,000). We used one single-threaded MPI rank per core and a look-ahead `depth` of 1. Finally, we used the `increasing-2-ring` broadcast with the `Crout` panel factorization algorithms.

Although this machine is much smaller than top supercomputers, we think that faithfully simulating an HPL execution with such settings is as challenging as a typical TOP500 qualification run.

- We simulated one rank per core to obtain a higher number (1024) of MPI process. This is more difficult than simulating one rank per node, as this (1) increases the amount of data transferred through MPI and (2) the performance is then subject to memory interference and network heterogeneity (which we modeled through different models for local and remote communications).
- We used a smaller block size than commonly used, which leads to a higher number of iterations and hence more complex communication patterns.
- We used a smaller input matrix and the makespan is therefore reduced. This is generally harder to predict with a good precision.

7.2 Comparing Simulations with Real Executions

Our first evaluation consists in comparing the traces of the simulations with reality. We instrumented HPL to collect the start and end

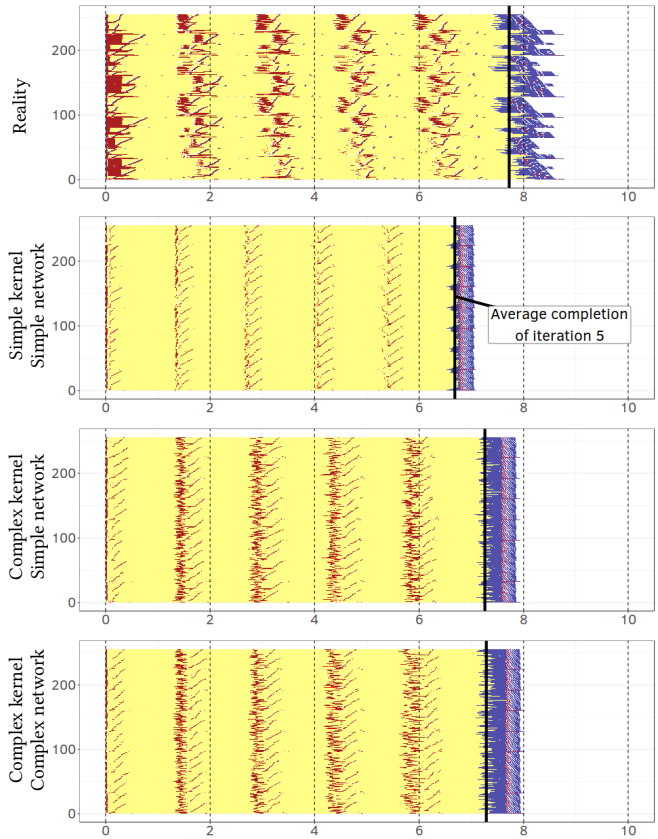


Figure 8: Gantt charts of HPL first iterations in simulation

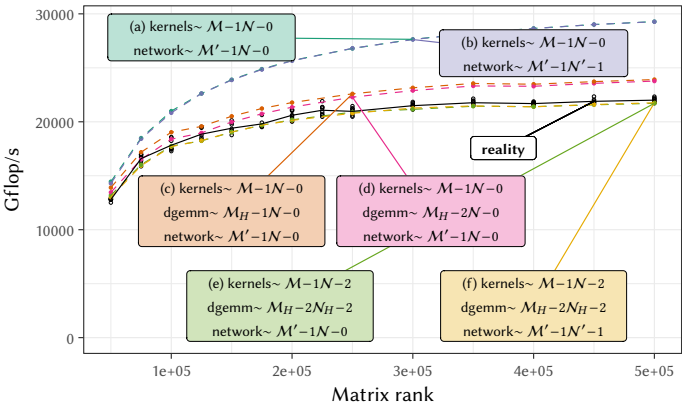


Figure 9: HPL performance: predictions vs. reality

timestamps of each kernel and MPI call. We limited the execution to 256 ranks and the first five iterations. A first qualitative validation can be done by visually comparing the Gantt charts of the simulations with reality (see Figure 8). Calls to `dgemm` are depicted in yellow, `MPI_Send` in red, `MPI_Recv` in blue. Although the structure is similar in all charts, the shape and the duration of the communication phases can be overly optimistic in simulations compared to reality. The charts show that, at this scale, using a deterministic or a stochastic model for the network has no noticeable impact on HPL simulation. However, having a more complex model for the kernels leads to much more realistic traces. The variability in the computation durations leads to an increase of the time spent in communications and an overall slightly longer execution. In HPL, computation variability directly translates to late senders/receivers that destroy the efficiency of collective operations.

We now provide a more quantitative comparison using the whole cluster and varying matrix sizes, focusing on the GFlop s^{-1} rate reported by HPL (see Figure 9). The real executions are depicted in black, for each matrix size we performed 8 runs of HPL, to illustrate the temporal variability of the performance. The line (a), on the top, is our first attempt to simulate HPL. The simulation was done with a simple model: $\mathcal{M}-1$ for the kernels and $\mathcal{M}'-1$ for the network with no noise ($\mathcal{N}-0$) in both cases. This model overestimates HPL performance by more than 30 %. We initially thought that the network model was too optimistic, however, switching to a stochastic multi-modal network model ($\mathcal{N}'-1$, the line (b)), does not significantly improve the prediction precision.

Figure 7 shows that there is an important heterogeneity in the cluster. For this reason, we started using a $\mathcal{M}_H-1/\mathcal{N}-0$ model for `dgemm` while keeping the models for the other kernels and the network as before. This increases very significantly the realism of the simulation as the performance is now overestimated by only 9 % (the line (c)). Using a polynomial model for `dgemm` instead of a linear model (thus switching from $\mathcal{M}_H-1$ to $\mathcal{M}_H-2$) further improves the performance prediction, in particular for smaller matrices. This new model (the line (d)), is very close to reality at the beginning but becomes equivalent to the previous model for larger matrices. We found that adding the temporal variability noise ($\mathcal{N}-2$ for all kernels, $\mathcal{N}_H-2$ for `dgemm`) is the key ingredient to obtain the last bit of realism. The prediction (the line (e)) is now extremely close to reality as it slightly underestimates the performance by less than 5 % and even as little as 2 % for the larger matrices. Adding back temporal variability to the network model ($\mathcal{N}'-1$, line (f)) still has no significant effect but this can be explained by the fact that HPL mostly communicates very large amounts of data in bulk. Network temporal variability may be a very important aspect to model for applications that are more *latency-bound*.

It is interesting to mention that we obtained early access to the Dahu cluster just after it was installed. It experienced several maintenance sessions and over the duration of our investigations (a few months), the performance of HPL has surprisingly decreased by about 10 %. Although we had full software control over the cluster, we noticed that the performance of a few nodes changed significantly. This increased the system heterogeneity, leading to the current performance depicted in Figure 9. By replacing the few currently slow nodes by "average" nodes in our model, we were able to re-estimate the overall performance of HPL and to obtain a prediction matching the performance obtained a few months ago.

We have shown it is possible to predict consistently the performance of HPL within a few percent of reality provided a faithful model of each individual resource is used. We would like to stress again that modeling both spatial and temporal variability is crucial.

7.3 Comparing With the Stampede Qualification Run

Each node of the Stampede cluster comprises two 8-core Intel Xeon E5-2680 8C 2.7 GHz CPUs and one 61-core Intel Xeon Phi SE10P

(KNC) 1.1 GHz accelerator that is roughly three times more powerful than the two CPUs. The HPL output submitted to the TOP500 (Table 1) does not indicate how the KNC was used. However, because of the values assigned to P and Q, we are certain that there were only one MPI rank per node. For this reason, it is likely that the KNC was used as an accelerator, which is effortless with Intel's Math Kernel Library (MKL) as it supports automatic offloading for selected BLAS functions. While we cannot know precisely which MKL version was used in 2013, we measured the default version (version 11.1.1) that was used on Stampede in the beginning of 2017. The MKL documentation states that, depending on the matrix geometry, the computation will run on either all the cores of the CPU or exclusively on the KNC. In the case of `dgemm`, the computation of $A = \alpha \cdot A + \beta \cdot B \times C$ with A, B, C of dimensions $M \times K$, $K \times N$ and $M \times N$, respectively, is offloaded onto the KNC whenever M and N are both larger than 1280 while K is simultaneously larger than 256, which calls for a $\mathcal{M}'$ model. Similarly, offloading for `dtrsm` is used when both M and N are larger than 512, which results in a better throughput but incurs a higher latency. The measured performance was close to the peak performance: e.g., `dgemm` on the Phi reached 1 TFlop s^{-1}. Since the block size used in HPL is 1024, all calls (except for maybe the very last iteration) are offloaded to the KNC. We therefore decided as a first step to use very simple ($\mathcal{M}'-1$ $\mathcal{N}-0$) models for `dgemm` and `dtrsm` and to simply skip all other functions, which is a very optimistic modeling. Likewise, although the performance of MPI measured in 2017 (see Figure 6) calls for a $\mathcal{M}'-1$ $\mathcal{N}'-1$ model, we used a simple optimistic $\mathcal{M}'-1$ $\mathcal{N}-0$ model. All HPL input parameters were set to exactly the values used in the TOP500 qualification run and were coherent with the submitted output. Figure 10 compares two simulation scenarios with the original result from 2013. The solid red line represents the HPL performance prediction as obtained with SMPI with the highly optimistic Stampede model we just described. Surprisingly, the prediction was much lower than the TOP500 result. We verified that no part of HPL was left unmodeled and decided to investigate whether a potential flaw in our network model that could for example cause too much congestion could explain the difference. Alas, switching to a congestion-free network model resulted in only minor improvements. In our experiments to model `DGEMM` and `DTRSM`, either the CPU or the KNC seemed to be used at a time and a specifically optimized version of the MKL may have been used in 2013. Removing the offloading latency and modeling each node as a single 1.2 TFlop s^{-1} node also fails (dashed blue line in Figure 10) to explain the huge difference between our prediction results and reality.

To identify the origin of this mismatch, we decided to simulate the first iterations of HPL to get an idea of what could be improved (the trace for the first five iterations can be obtained in about 10 minutes on a commodity computer and is about 175 MB large once compressed). Figure 11 illustrates the very synchronous and phase-based nature of the first iterations with the HPL configuration of Table 1: one can identify first a factorization of the panel, then a broadcast to all the nodes, and finally an update of the trailing matrix. More than one fifth of each iteration is spent communicating (although the first iterations are the ones with the lowest communication to computation ratio), which prevents HPL from reaching

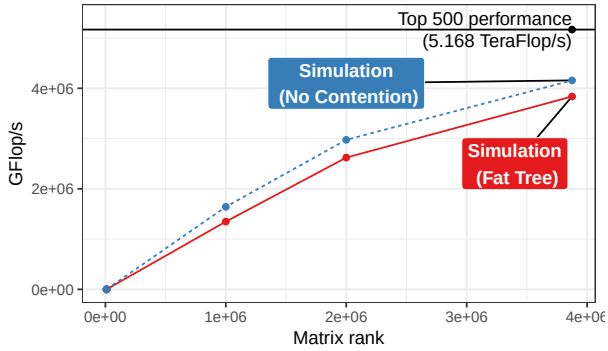


Figure 10: HPL on Stampede: predictions vs. reality

the TOP500 performance. Overlapping these heavy communication phases with computation would significantly improve overall performance. The fact that this is almost not happening can be explained by the look-ahead `DEPTH` parameter that was allegedly set to `0`. This is quite surprising as even the tuning section of the HPL documentation indicates that a `depth` of `1` is supposed to yield the best results, even though a large problem size could be needed to see some performance gain. We discussed this surprising behavior with the Stampede-team and were informed that, although the HPL output mentions `HPLinpack 2.1` from `October 26, 2012` by Petitet *et al.*, the run in 2013 was actually executed with Intel's binary version of HPL and which was probably specifically modified for Stampede. The submitted HPL output provides information about the progress of HPL throughout iterations and statistics for the panel-owning process about the time spent in the most important parts. According to these statistics, the total time spent in the `Update` section was 9390 sec whereas the total execution time was 7505 sec, which is impossible unless iterations have overlapped.

It is thus almost certain that some configuration values have been hardcoded to enforce an overlap of iterations with others and that the values HPL reported in its output should not be trusted. The broadcast and swapping algorithms use very heavy communication patterns. This is not a surprise, since for a matrix of this order, several hundred megabytes need to be broadcast. The output states that the `long-modified` broadcast algorithm was used, however, it is likely that another algorithm was used instead. We tried the six other broadcast algorithms provided by HPL but could not obtain

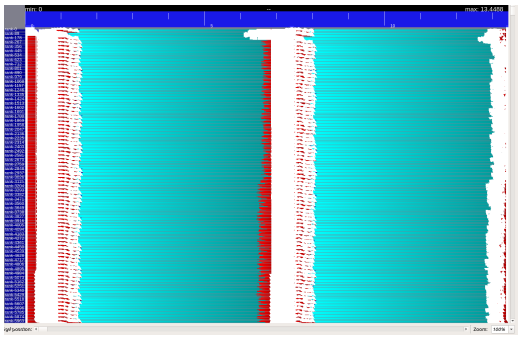


Figure 11: Gantt chart of the first two iterations of HPL. Communication states are in red while computations are in cyan. Each communication is represented with a white arrow, which results in very cluttered white areas.

significantly better overall performance. An analysis of the symbols in the Intel binary revealed that another broadcast algorithm named `HPL_bcast_bpush` was available. Unlike the others, this new algorithm relies on non-blocking sends, which could contribute to the performance obtained in 2013. Likewise, the swapping algorithm that was used (`SWAP=Binary-exchange`) involves communications that are rather long and organized in trees, which is surprising as the `spread-roll` algorithm is recommended for large matrices.

We have not reverse engineered the Intel HPL code. We can, however, already draw two conclusions from our analysis: (1) the parameters printed by HPL are not the ones used in the real execution, probably because these values were hardcoded and the configuration output file was not updated accordingly and (2) it is apparent that the communication part has been optimized with respect to asynchronous communications and to the broadcast algorithm.

8 CONCLUSIONS

Studying HPC applications at scale can be very time- and resource-consuming. We believe that being capable of precisely predicting an application's performance on a given platform will become invaluable in the future as it can for example aid compute centers with the decision of whether the envisioned technology of a new machine works best for a given application or if an upgrade of the current machine should be considered. Simulation is often an effective approach in this context and SMPI has previously been successfully validated in several small-scale studies with simple HPC benchmarks [3, 14]. In this article, we proposed and evaluated extensions to the SimGrid/SMPI framework that allowed us to emulate HPL at the scale of a supercomputer. Our application of choice, HPL, is particularly challenging in terms of simulation as it implements its own set of non-blocking collective operations that rely on `MPI_Iprobe` in order to facilitate overlapping with computations.

More specifically, we tried to reproduce the execution of HPL on the Stampede supercomputer conducted in 2013 for the TOP500, which involved a 120 TB matrix and took two hours on 6,006 nodes. Our emulation of this specific configuration runs on a single machine for about 62 hours and requires less than 19 GB of RAM. This emulation employed several non-trivial operating-system level optimizations (memory mapping, dynamic library loading, huge pages) that have since been integrated into the last version of SimGrid/SMPI.

The downside of scaling this high is a less well-controlled scenario. We explained how to model the performance of compute kernels and of the MPI library and showed that a careful modeling allows to predict performance within a few percents of real experiments on a controlled cluster of Grid'5000. Interestingly, spatial and temporal variability are key ingredients of faithful predictions. Unfortunately, the reference run of HPL on Stampede was done several years ago. We only have very limited information about the setup (e.g., software versions and configuration) and a reservation and re-execution on the whole machine was impossible. We nevertheless modeled Stampede carefully, which allowed us to predict the performance that would have been obtained using an unmodified, freely available version of HPL. Despite all our efforts, the predicted performance was much lower than what was reported in 2013. We determined that this discrepancy comes from the fact that a modified, closed-source version of HPL supplied by Intel was used in 2013 and misreports the true configuration. An analysis of the optimized HPL binary confirmed that the availability of new algorithms is likely the reason for the performance discrepancy. We claim that the modifications we made in HPL are minor and are applicable to Intel's optimized version as well. In fact, while HPL comprises 16K lines of ANSI C over 149 files, our modifications only changed 14 files with 286 line insertions and 18 deletions.

As we explained, a careful modeling of networking and computing resources is crucial but requires a combination of statistical libraries. We intend to automate our benchmarking and modeling approach as much as possible using a generic and unified Bayesian estimation framework like STAN [24]. The MCMC sampling technique used in STAN would be particularly helpful to inject uncertainty (in particular the platform heterogeneity which should be estimated from only a few sample nodes) in the simulation. This integration effort is underway. As another future work, building on the effort of SimGrid developers on supporting the emulation of a wide variety of applications with SMPI [25], we also intend to conduct similar studies with other HPC benchmarks (e.g., HPCG [26] or HPGMG [27]), and real applications (e.g., BigDFT [28]) and other TOP500 machines. In this context, we believe that improving reproducibility through a faithful and public reporting of the experimental conditions (compiler options, library versions, input parameters, HPL output, etc.) is invaluable and would allow researchers to better understand how to correctly model these platforms.

Acknowledgments

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

We warmly thank our TACC colleagues for their support in this study and providing us with as much information as they could. We thank our Cray colleagues for providing us with information about the TOP500 run of Theta, a resource of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

Last, we warmly thank the SimGrid developers for their help in integrating our contributions as well as for their early feedback on this work before submission.

REFERENCES

[1] H. W. Meuer, E. Strohmaier, J. Dongarra, and H. D. Simon, *The TOP500: History, Trends, and Future Directions in High Performance Computing*, 1st ed. Chapman & Hall/CRC, 2014.

[2] H. Casanova, A. Giersch, A. Legrand, M. Quinson, and F. Suter, "Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2899–2917, 2014.

[3] A. Degomme, A. Legrand, G. Markomanolis, M. Quinson, M. S. Stillwell, and F. Suter, "Simulating MPI applications: the SMPI approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 8, pp. 2387–2400, Feb. 2017.

[4] A. Petitet, C. Whaley, J. Dongarra, A. Cleary, and P. Luszczek, "HPL - a portable implementation of the High-Performance Linpack benchmark for distributed-memory computers," http://www.netlib.org/benchmark/hpl, February 2016, version 2.2.

[5] G. Zheng, G. Kakulapati, and L. Kale, "BigSim: A Parallel Simulator for Performance Prediction of Extremely Large Parallel Machines," in *Proc. of the 18th IPDPS*, 2004.

[6] R. M. Badia, J. Labarta, J. Giménez, and F. Escalé, "Dimemas: Predicting MPI Applications Behaviour in Grid Environments," in *Proc. of the Workshop on Grid Applications and Programming Tools*, Jun. 2003.

[7] M. Mubarak, C. D. Carothers, R. B. Ross, and P. H. Carns, "Enabling parallel simulation of large-scale HPC network systems," *IEEE Transactions on Parallel and Distributed Systems*, 2016.

[8] H. Casanova, F. Desprez, G. S. Markomanolis, and F. Suter, "Simulation of MPI applications with time-independent traces," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 5, p. 24, Apr. 2015. [Online]. Available: https://hal.inria.fr/hal-01232776

[9] X. Wu and F. 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.

[10] L. Carrington, M. Laurenzano, and A. Tiwari, "Inferring large-scale computation behavior via trace extrapolation," in *Proc. of the Workshop on Large-Scale Parallel Processing*, 2013.

[11] C. L. Janssen, H. Adalsteinsson, S. Cranford, J. P. Kenny, A. Pinar, D. A. Evensky, and J. Mayo, "A simulator for large-scale parallel architectures," *International Journal of Parallel and Distributed Systems*, vol. 1, no. 2, pp. 57–73, 2010, http://dx.doi.org/10.4018/jdst.2010040104.

[12] C. Engelmann, "Scaling To A Million Cores And Beyond: Using Light-Weight Simulation to Understand The Challenges Ahead On The Road To Exascale," *FGCS*, vol. 30, pp. 59–65, Jan. 2014.

[13] P. Velho, L. Schnorr, H. Casanova, and A. Legrand, "On the Validity of Flow-level TCP Network Models for Grid and Cloud Simulations," *ACM Transactions on Modeling and Computer Simulation*, vol. 23, no. 4, p. 23, Oct. 2013.

[14] F. C. Heinrich, T. Cornebize, A. Degomme, A. Legrand, A. Carpen-Amarie, S. Hunold, A.-C. Orgerie, and M. Quinson, "Predicting the Energy Consumption of MPI Applications at Scale Using a Single Node," in *Proc. of the 19th IEEE Cluster Conference*, 2017. [Online]. Available: https://hal.inria.fr/hal-01523608

[15] T. Cornebize, "Capacity Planning of Supercomputers: Simulating MPI Applications at Scale," Master's thesis, Grenoble INP ; Université Grenoble - Alpes, Jun. 2017. [Online]. Available: https://hal.inria.fr/hal-01544827

[16] ——, "Laboratory Notebook hosted on GitHub. Capacity Planning of Supercomputers: Simulating MPI Applications at Scale," https://github.com/Ezibenroc/m2_internship_journal, 2017.

[17] D. Balouek, A. Carpen-Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, ser. Communications in Computer and Information Science, I. Ivanov, M. Sinderen, F. Leymann, and T. Shan, Eds. Springer International Publishing, 2013, vol. 367.

[18] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2018. [Online]. Available: https://www.R-project.org/

[19] S. Seabold and J. Perktold, "Statsmodels: Econometric and statistical modeling with python," in *9th Python in Science Conference*, 2010.

[20] M. Kuhn and R. Quinlan, *Cubist: Rule- And Instance-Based Regression Modeling*, 2018, r package version 0.2.2. [Online]. Available: https://CRAN.R-project.org/package=Cubist

[21] T. Hastie, R. Tibshirani, and J. H. Friedman, *The elements of statistical learning: data mining, inference, and prediction, 2nd Edition*, ser. Springer series in statistics. Springer, 2009. [Online]. Available: http://www.worldcat.org/oclc/300478243

[22] D. Malerba, F. Esposito, M. Ceci, and A. Appice, "Top-down induction of model trees with regression and splitting nodes," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 5, pp. 612–625, may 2004. [Online]. Available: https://doi.org/10.1109%2Ftpami.2004.1273937

[23] B. Grün and F. Leisch, "FlexMix version 2: Finite mixtures with concomitant variables and varying and constant parameters," *Journal of Statistical Software*, vol. 28, no. 4, pp. 1–35, 2008. [Online]. Available: http://www.jstatsoft.org/v28/i04/

[24] B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, and A. Riddell, "Stan: A probabilistic programming language," *Journal of Statistical Software*, vol. 76, no. 1, 2017. [Online]. Available: https://doi.org/10.18637%2Fjss.v076.i01

[25] S. Developers, "Continuous integration of ECP Proxy Apps, CORAL and Trinity-Nersc benchmarks with SimGrid/SMPI," https://github.com/simgrid/SMPI-proxy-apps/, 2019.

[26] J. J. Dongarra, M. A. Heroux, and P. Luszczek, "HPCG benchmark : a new metric for ranking high performance computing systems," Technion Israel Institute of Technology, Tech. Rep. UT-EECS-15-736, 2015.

[27] M. F. Adams, J. Brown, J. Shalf, B. Straalen, E. Strohmaier, and S. Williams, "Hpgmg 1.0: A benchmark for ranking high performance computing systems," LBNL, Tech. Rep., 05 2014.

[28] L. Genovese, A. Neelov, S. Goedecker, T. Deutsch, S. A. Ghasemi, A. Willand, D. Caliste, O. Zilberberg, M. Rayson, A. Bergman, and R. Schneider, "Daubechies wavelets as a basis set for density functional pseudopotential calculations," *Journal of Chemical Physics*, vol. 129, 2008.