Emulating High Performance Linpack on a Commodity Server at the Scale of a Supercomputer

Tom Cornebize, Franz C. Heinrich, Arnaud Legrand
Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG
38000 Grenoble, France
firstname.lastname@inria.fr

Jérôme Vienne
Texas Advanced Computing Center
Austin, Texas, USA
viennej@tacc.utexas.edu

***Abstract*—The Linpack benchmark, in particular the High-Performance Linpack (HPL) implementation, has emerged as the de-facto standard benchmark to rank supercomputers in the TOP500. With a power consumption of several MW per hour on a TOP500 machine, test-running HPL on the whole machine for hours is extremely expensive. With core-counts beyond the 100,000 cores threshold being common and sometimes even ranging into the millions, an optimization of HPL parameters (problem size, grid arrangement, granularity, collective operation algorithms, etc.) specifically suited to the network topology and performance is essential. Such optimization can be particularly time consuming and can hardly be done through simple mathematical performance models. In this article, we explain how we both extended the SimGrid's SMPI simulator and slightly modified HPL to allow a fast emulation of HPL on a single commodity computer at the scale of a supercomputer. More precisely, we take as a motivating use case the large-scale run performed on the Stampede cluster at TACC in 2013, when it got ranked 6th in the TOP500. While this qualification run required the dedication of 6,006 computing nodes of the supercomputer and more than 120 TB of RAM for more than 2 hours, we manage to simulate a similar configuration on a commodity computer with 19 GB of RAM in about 62 hours. Allied to a careful modeling of Stampede, this simulation allows us to evaluate the performance that would have been obtained using the freely available version of HPL. Such performance reveals much lower than what was reported and which was obtained using a closed-source version specifically designed by the Intel engineers. Our simulation allows us to hint where the main algorithmic improvements must have been done in HPL.**

I. Introduction

The world's largest and fastest machines are ranked twice a year in the so-called TOP500 list. Among the benchmarks that are often used to evaluate those machines, the Linpack benchmark, in particular the High-Performance Linpack (HPL) implementation, has emerged as the de-facto standard benchmark, although other benchmarks such as HPCG and HPGMG have recently been proposed to become the new standard. 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 diligent optimization of application parameters, such as problem size, process organization or choice of algorithm, as these have an impact on load distribution and network utilization. Furthermore, to yield best benchmark results, runtimes (such as OpenMPI) and supporting libraries (such as BLAS) need to be fine-tuned and adapted to the underlying platform.

Alas, it takes typically several hours to run HPL on the list's number one system. This duration, combined with the power consumption that often reaches several MW for TOP500 machines, makes it financially infeasible to test-run HPL on the whole machine just to tweak parameters. Yet, performance results of an already deployed, current-generation machine typically also play a role in the funding process for future machines. Results near the optimal performance for the current machine are hence considered critical for HPC centers and vendors. These entities would benefit from being able to tune parameters without actually running the benchmark for hours.

In this article, we explain how to predict the performance of HPL through simulation with the SimGrid/SMPI simulator. We detail how we obtained faithful models for several functions (e.g., `DGEMM` and `DTRSM`) and how we managed to reduce the memory consumption from more than a hundred terabytes to several gigabytes, allowing us to emulate HPL on a commonly available server node. We evaluate the effectiveness of our solution by simulating a scenario similar to the run conducted on the Stampede cluster (TACC) in 2013 for the TOP500 .

This article is organized as follows: Section II presents the main characteristics of the HPL application and provides detail on the run that was conducted at TACC in 2013. Section III discusses existing related work and explains why emulation (or *online simulation*) is the only relevant approach when studying an application as complex as HPL. In Section IV, we briefly present the simulator we used for this work, SimGrid/SMPI, followed by an extensive discussion in Section V about the optimizations on all levels (i.e., simulator, application, system) that were necessary to make a large-scale run tractable. The scalability of our approach is evaluated in Section VI. The modeling of the Stampede platform and the comparison of our simulation with the 2013 execution is detailed in Section VII. Lastly, Section VIII concludes this article by summarizing our contributions.

II. Context

A. High-Performance Linpack

For this work, we use the freely-available reference-implementation of the High-Performance Linpack benchmark [1], HPL, which is used to benchmark systems for the TOP500 [2] list. HPL requires MPI to be available and implements a LU decomposition, i.e., a factorization of a square matrix A as the product of a lower triangular matrix L and an upper triangular matrix U. HPL checks the correctness of this factorization by solving a linear system $A \cdot x = b$, but only the

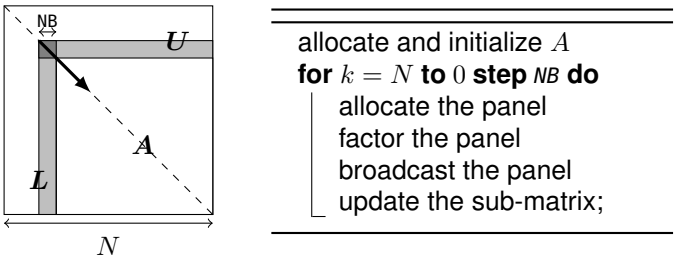


Figure 1. Overview of High Performance Linpack

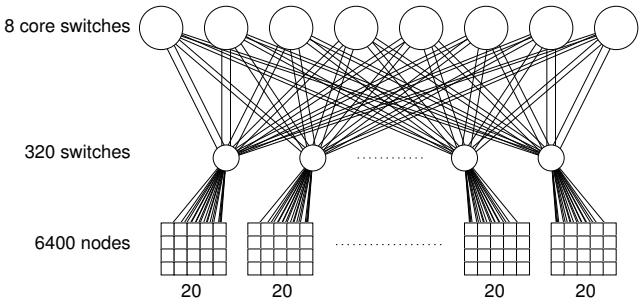


Figure 2. The fat-tree network topology of Stampede.

factorization step is benchmarked. The factorization is based on a right-looking variant of the LU factorization with row partial pivoting and allows multiple look-ahead depths. The working principle of the factorization is depicted in Figure 1 and 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 collective communication algorithms to efficiently overlap communication with computation. The main parameters of HPL are listed subsequently:

- 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 the panel of columns to the other process columns. Legacy versions of the MPI standard only supported non-blocking point-to-point communications but did not support non-blocking collective communications, which is why HPL ships with in total 6 self-implemented variants to efficiently 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 participate further in the broadcast. This process can thereby start working on the panel as soon as possible. The `ring` and `2-ring` versions correspond to the name-giving two 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 and hence facilitate partial overlapping of communication with computations. In HPL 2.2 and 2.1, 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 $\mathrm{flop}(N) = \frac{2}{3}N^3 + 2N^2 + \mathcal{O}(N)$ 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 by the previously listed parameters (`RFACT`, `SWAP`, `BCAST`, `DEPTH`, ...). After each run, HPL reports the overall flop rate $\mathrm{flop}(N)/T(N)$ (expressed in $\mathrm{GFlop\,s^{-1}}$) for the given configuration. See Figure 3 for a (shortened) example output.

A large-scale execution of HPL on a real machine in order to submit to the TOP500 can therefore be quite time consuming as all the BLAS kernels, the MPI runtime, and HPL's numerous parameters need to be tuned carefully in order to reach optimal performance.

B. A Typical Run on a Supercomputer

In June 2013, the Stampede supercomputer at TACC was ranked 6th in the TOP500 by achieving $5168.1\,\mathrm{TFlop\,s^{-1}}$ and was still ranked 20th in June 2017. In 2017, this machine got upgraded and renamed Stampede2. The Stampede platform consisted of 6400 Sandy Bridge nodes, each with two 8-core Xeon E5-2680 and one Intel Xeon Phi KNC MIC coprocessor. The nodes were interconnected through a $56\,\mathrm{Gbit\,s^{-1}}$ FDR InfiniBand 2-level Clos fat-tree topology built on Mellanox switches. As can be seen in Figure 2, the 6400 nodes are divided into groups of 20, with each group being connected to one of the 320 36-port switches ($4\,\mathrm{Tbit\,s^{-1}}$ capacity), which are themselves connected to 8 648-port "core switches" (each with a capacity of $73\,\mathrm{Tbit\,s^{-1}}$). The peak performance of the 2 Xeon CPUs per node was approximately $346\,\mathrm{GFlop\,s^{-1}}$, while the peak performance of the KNC co-processor was about $1\,\mathrm{TFlop\,s^{-1}}$. The theoretical peak performance of the platform was therefore $8614\,\mathrm{TFlop\,s^{-1}}$. However, in the TOP500, Stampede was ranked with $5168\,\mathrm{TFlop\,s^{-1}}$. According to the log submitted to the TOP500 (see Figure 3) that was provided to us, this execution took roughly two hours and used $77 \times 78 = 6,006$ processes. The matrix of order $N = 3,875,000$ occupied approximately $120\,\mathrm{TB}$ of memory, i.e., $20\,\mathrm{GB}$ per

```
================================================================================
HPLinpack 2.1  --  High-Performance Linpack benchmark  --   October 26, 2012
Written by A. Petitet and R. Clint Whaley,  Innovative Computing Laboratory, UTK
Modified by Piotr Luszczek, Innovative Computing Laboratory, UTK
Modified by Julien Langou, University of Colorado Denver
================================================================================

The following parameter values will be used:

N        : 3875000
NB       :   1024
PMAP     : Column-major process mapping
P        :      77
Q        :      78
PFACT    :   Right
NBMIN    :       4
NDIV     :       2
RFACT    :   Crout
BCAST    :  BlongM
DEPTH    :       0
SWAP     : Binary-exchange
L1       : no-transposed form
U        : no-transposed form
EQUIL    : no
ALIGN    :    8 double precision words

--------------------------------------------------------------------------------


[...]


Peak Performance = 5172687.23 GFlops /   861.25 GFlops per node
================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WC05C2R4     3875000  1024    77    78            7505.72             5.16811e+06
HPL_pdgesv() start time Sun Jun  2 13:04:59 2013

HPL_pdgesv() end time   Sun Jun  2 15:10:04 2013

--------------------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)=        0.0007822 ..... PASSED
```

Figure 3. HPL output submitted in June 2013 for the ranking of Stampede in the TOP500.

node. One MPI process per node was used and each node's computational resources (the 16 CPU-cores and the Xeon Phi) must have been controlled by OpenMP and/or Intel's MKL.

C. Performance Evaluation Challenges

The performance achieved by Stampede, $5168\,\mathrm{TFlop\,s^{-1}}$, needs to be compared to the peak performance of the 6,006 nodes, i.e., $8084\,\mathrm{TFlop\,s^{-1}}$. This difference may be attributed to the node usage (e.g., the MKL), to the MPI library, to the network topology that may be unable to deal with the very intensive communication workload, to load imbalance among nodes because some node happens to be slower for some reason (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 II-A) may be able to provide trends but can by no means predict the performance for each configuration (i.e., consider the exact effect of HPL's 6 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, very faithful modeling of both the application and the platform is required. Given the scale of this scenario (3,785 steps on 6,006 nodes in two hours), detailed simulations quickly become intractable without significant effort.

III. Related Work

Performance prediction of MPI application through simulation has been widely studied over the last decades, with today's literature distinguishing mainly between two approaches: offline and online simulation.

With the most common approach, *offline simulation*, a time-independent trace of the application is first obtained on a real platform. This trace comprises sequences of MPI optimizations and CPU bursts and can be given as an input to a simulator that implements performance models for the CPUs and the network to derive timings. 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 [3], Dimemas [4] and CODES [5], rely on this approach.

The main limitation of this approach comes from the trace acquisition requirement. Additionally, tracing an application provides only information about its behavior at the time of the run. Even light modifications (e.g., to communication patterns) may make the trace inaccurate. For simple applications (e.g., `stencil`) it is sometimes possible to extrapolate behavior from small-scale traces [6], [7] but the execution is non-deterministic whenever the application relies on non-blocking communication patterns, which is unfortunately the case for HPL.

The second approach discussed in literature is *online simulation*. Here, the application is executed (emulated) on top of a simulator that is responsible for determining 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 [8], SimGrid/SMPI [9] and the closed-source extreme-scale simulator xSim [10] support it. To the best of our knowledge, only SST Macro and SimGrid/SMPI are not only mature enough to faithfully emulate HPL but also free software. For our work, we relied on SimGrid as we have an excellent knowledge of its internals although the developments we propose would a priori also be possible with SST Macro. Emulation of HPL comes with at least two challenges:

- Firstly, the time-complexity of the algorithm is $\Theta(N^3)$. Furthermore, $\Theta(N^2)$ 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 is reached. Although the emulation could be done in parallel, we want to use as little computing resources as possible.
- Secondly, the tremendous memory consumption and consequent high number of RAM accesses for read/write operations need to be dealt with.

IV. SimGrid/SMPI in a Nutshell

SimGrid [9] is a flexible and open-source simulation framework that was originally designed in 2000 to study scheduling heuristics tailored to heterogeneous grid computing environments. Since then, SimGrid has also been used to study peer-to-peer systems with up to two million peers [11] just as cloud and HPC infrastructures. To this end, SMPI, a simulator

based on SimGrid, has been developed and used to faithfully simulate unmodified MPI applications written in C/C++ or FORTRAN [12]. A main development goal for SimGrid has been to provide validated performance models particularly for scenarios leveraging the network. Such a validation normally consists of comparing simulation predictions with results from real experiments to confirm or debunk network and application models. In [13], we have for instance validated SimGrid's energy module by accurately and consistently predicting within a few percent the performance and the energy consumption of HPL and some other benchmarks on small-scale clusters (up to 12×12 cores in [13] and up to 128×1 cores in [12]).

In this article, we aim to validate our approach through much larger experiments. This scale, however, comes at the cost of a much less controlled scenario for real-life experiments since the Stampede run of HPL was done in 2013 and we only have very limited information about the setup (e.g., software versions).

A. MPI Communication Modeling

The complex network optimizations done in real MPI implementations need to be considered when predicting performance of MPI applications. For instance, message size not only influences the network's latency and bandwidth factors but also the protocol used, such as "eager" or "rendez-vous", as they are selected based on the message size, with each protocol having its own synchronization semantics. To deal with this, SMPI relies on a generalization of the LogGPS model [12] and supports specifying synchronization and performance modes. This model needs to be instantiated once per platform through a carefully controlled series of messages (`MPI_Send` and `MPI_Recv`) between two nodes and through a set of piece-wise linear regressions. Modeling network topologies and contention is also difficult. 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 be modeled as a bandwidth sharing problem that accounts for non-trivial phenomena (e.g., RTT-unfairness of TCP, cross-traffic interference or network heterogeneity [14]). Communications that start or end trigger re-computation of the bandwidth sharing if needed. 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.

Finally, collective operations are also challenging, particularly since these operations often play a key factor to an application's performance. Consequently, performance optimization of these operations has been studied intensively. As a result, MPI implementations now commonly have several alternatives for each collective operation and select one at runtime, depending on message size and communicator geometry. SMPI implements collective communication algorithms and the selection logic from several MPI implementations (e.g., Open MPI, MPICH), which helps to ensure that simulations are as close as possible to real executions. Although SMPI supports these facilities, they are not required in the case of HPL as it ships with its own implementation of collective operations.

B. Application Behavior Modeling

In Section III we explained that SMPI relies on the *online* simulation approach. Since SimGrid is a sequential simulator, SMPI maps every MPI process 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 can be injected into the simulator as a virtual delay.

Mapping MPI processes to threads of a single process effectively folds 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 [12]. The default strategy in SMPI consists of 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 saves calls to `mmap` when context switching.

This causes online simulation to be expensive in terms of both simulation time and memory since the whole parallel application is executed on a single node. To deal with this, SMPI provides two simple annotation mechanisms:

- **Kernel sampling**: Control flow is in many cases independent of the computation results. This allows computation-intensive kernels (e.g., BLAS kernels for HPL) to be skipped during the simulation. For this purpose, SMPI supports annotation of regular kernels through several macros such as `SMPI_SAMPLE_LOCAL` and `SMPI_SAMPLE_GLOBAL`. The regularity allows SMPI to execute these kernels a few times, estimate their cost and skip the kernel in the future by deriving its cost from these samples, hence cutting simulation time significantly. 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.
- **Memory folding**: 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: a single block of physical memory (of default size 1 MB) for the whole execution is allocated and shared by all

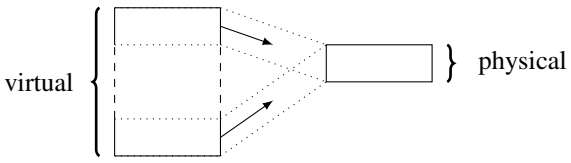


Figure 4. SMPI shared malloc mechanism: large area of virtual memory are cyclically mapped onto the same physical pages.

MPI processes. A range of virtual addresses corresponding to a specified size is reserved and cyclically mapped onto the previously obtained physical address, as illustrated by Figure 4. This mechanism allows applications to obtain a nearly constant memory footprint, regardless of the size of the actual allocations.

V. Improving SMPI Emulation Mechanisms and Preparing HPL

We now present our changes to SimGrid and HPL that were required for a scalable and faithful simulation. We provide only a brief evaluation of our modifications and refer the reader interested in details to [15] and our laboratory notebook[1]. For our experiments in this section, we used a single core from nodes of the Nova cluster provided by the Grid'5000 testbed [16] with 32 GB RAM, two 8-core Intel Xeon E5-2620 v4 CPUs processors with 2.1 GHz and Debian Stretch (kernel 4.9).

A. Kernel modeling

As explained in Section II-C, faithful prediction of HPL necessitates emulation, i.e., to execute the code. HPL relies heavily on BLAS kernels such as `dgemm` (for matrix-matrix multiplication) or `dtrsm` (for solving an equation of the form $Ax = b$). An analysis of an HPL simulation with 64 processes and a very small matrix of order $30,000$ showed that roughly $96\,\%$ of the time is spent in these two very regular kernels. For larger matrices, these kernels will consume an even bigger percentage of the computation time. Since these kernels do not influence the control flow, simulation time can be reduced by substituting `dgemm` and `dtrsm` function calls with a performance model for the respective kernel. Figure 5(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 computation kernel and was obtained through calibration. The estimated time for the real kernel is calculated based on the parameters and eventually passed on to `smpi_execute_benched` that advances the clock of the executing rank by this estimate by entering a sleep state. The effect on simulation time for a small scenario is depicted in Figure 5(b). On the one hand, this modification speeds up the simulation by orders of magnitude, especially when the matrix order grows. On the other hand, this kernel model leads to an optimistic estimation of the floprate. This may be caused by inaccuracies in our model as well as by the fact that the initial emulation is generally more sensitive to pre-emptions, e.g., by the operating system, and therefore more likely to be pessimistic compared to a real execution.

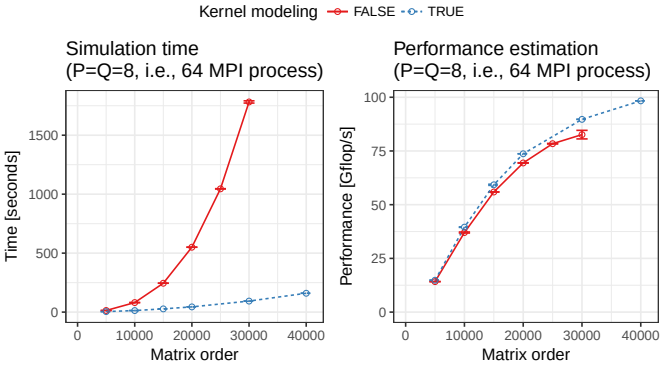
B. Adjusting the behavior of HPL

HPL uses pseudo-randomly generated matrices that need to be setup every time HPL is executed. The time spent on this just as the validation of the computed result is not considered in the reported $\mathrm{GFlop\,s^{-1}}$ performance. We skip all the computations since we replaced them by a kernel model and therefore, result validation is meaningless. Since both phases do not have an impact on the reported performance, we can safely skip them.

[1]See `journal.org` at https://github.com/Ezibenroc/simulating_mpi_applications_at_scale/

```
#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.



(b) Gain in term of simulation time.

Figure 5. Replacing the calls to computationally expensive functions by a model allows to significantly reduce simulation time.

In addition to the main computation kernels `dgemm` and `dtrsm`, 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`.

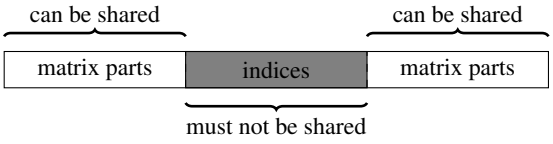
All of these functions are called during the LU factorization and hence impact the performance measured by HPL; however, because of the removal of the `dgemm` and `dtrsm` computations, they all operate on bogus data and hence also produce bogus data. We also determined through experiments that their impact on the performance prediction is minimal and hence modeled them for the sake of simplicity as being instantaneous.

Note that HPL implements an LU factorization with partial pivoting and 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 an arbitrary value to make the simulation fully deterministic. We confirmed that this modification is harmless in terms of performance prediction while it speeds up the simulation by an additional factor of ≈ 3 to 4 on small ($N = 30,000$) and even more on large scenarios.

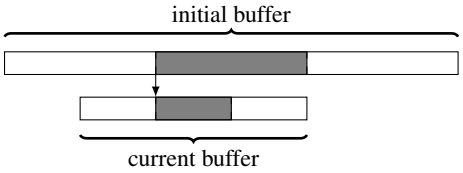
C. Memory folding

As explained in Section IV, when emulating an application with SMPI, all MPI processes are run within the same simulation process on a single node. 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



(a) Structure of the panel in HPL.



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

Figure 6. Panel structure and allocation strategy when simulating.

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 altogether. Instead, SMPI's `SHARED_MALLOC` mechanism can be used to share unimportant data structures between all ranks, minimizing the memory footprint.

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. Although using the default `SHARED_MALLOC` mechanism works flawlessly for `A`, a more careful strategy needs to be used for the `panel`. Indeed, the `panel` 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 6(a)). Using a fully shared memory allocation for the `panel` therefore leads to index corruption that results in classic invalid memory accesses as well as communication deadlocks, as processes may not send to or receive from the correct process. 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 macro `SMPI_PARTIAL_SHARED_MALLOC` that works as follows: `mem = SMPI_PARTIAL_SHARED_MALLOC(500, {27,42 , 100,200}, 2)`. In this example, 500 bytes are allocated in `mem` with the elements `mem[27]`, ..., `mem[41]` and `mem[100]`, ..., `mem[199]` being shared between processes (they are therefore generally completely corrupted) while all other elements remain private. To apply this to HPL's `panel` data-structure and partially share it between processes, we only had to modify a few lines.

Designating memory explicitly as private, shared or partially shared helps with both memory management and overall performance. As SMPI is internally aware of the memory's visibility, it can 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 were considerably improved in our experiments because the `panel` as the most frequently transferred datastructure is partially shared with only a small part being private. The additional error introduced by this technique was negligible (below $1\,\%$) while the memory consumption was lowered significantly: for a matrix of order $40,000$ and 64 MPI processes, the memory consumption decreased from about $13.5\,\mathrm{GB}$ to less than $40\,\mathrm{MB}$.

D. Panel reuse

HPL `mallocs`/`frees` 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 become a bottleneck. Since the very first allocation can fit all subsequent panels, we modified HPL to allocate only the first panel and reuse it for subsequent iterations (see Figure 6(b)).

We consider this optimization harmless with respect to simulation accuracy as the maximum additional error that we observed was always less than $1\,\%$. Simulation time is reduced significantly, albeit the reached speed-up is less impressive than for previous optimizations: For a very small matrix of order $40,000$ and 64 MPI processes, the simulation time decreases by four seconds, from $20.5\,\mathrm{sec}$ to $16.5\,\mathrm{sec}$. Responsible for this is a reduction of system time, namely from $5.9\,\mathrm{sec}$ to $1.7\,\mathrm{sec}$. The number of page faults decreased from 2 million to 0.2 million, confirming the devastating effect these allocations/deallocations would have at scale.

E. MPI process representation (mmap vs. dlopen)

We already explained in Section IV-B that SMPI supports two mechanisms to keep local static and global variables private to each rank, even though they run in the same process. In this section, we discuss the impact of the choice.

- **mmap** When `mmap` is used, SMPI copies the `data` segment on startup for each rank into the heap. When control is transferred from one rank to another, the `data` segment is `mmap`'ed to the location of the other rank's copy on the heap. All ranks have hence the same addresses in the virtual address space at their disposition although `mmap` ensures they point to different physical addresses. This also means inevitably that caches must be flushed to ensure that no data of one rank leaks into the other rank, making `mmap` a rather expensive operation.
- **dlopen** With `dlopen`, copies of the global variables are still made but they are stored inside the `data` segment as opposed to the heap. When switching from one rank to another, the starting virtual address for the storage is readjusted rather than the target of the addresses. This means that each rank has distinct addresses for global variables. The main advantage of this approach is that caches do not need to be flushed as is the case for the `mmap` approach, because data consistency can always be guaranteed.

Impact of choice of mmap/dlopen The choice of `mmap` or `dlopen` influences the simulation time indirectly through its

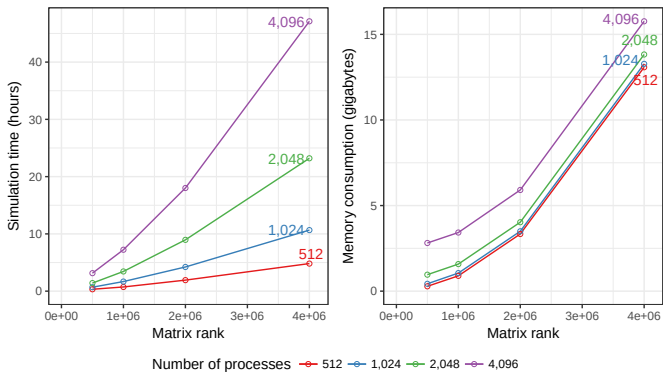


Figure 7. Time complexity and memory consumption are linear in the number of processes but remain mildly quadratic with matrix rank.

impact on system/user time and page faults, e.g., 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`). This results in a reduction of system time from $10.64\,\text{sec}$ (out of $51.47\,\text{sec}$ in total) to $2.12\,\text{sec}$. Obviously, the larger the matrix and the number of processes, the larger the number of context switch during the simulation, and thus the higher the gain.

F. Huge pages

For larger matrix orders (i.e., N larger than a few hundred thousand), the performance of the simulation quickly deteriorates as the memory consumption rises rapidly.

We explained already how we fold the memory in order to reduce 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 to fit in a single node. More precisely, the size of the page table containing pages of size $4\,\text{KiB}$ can be computed as:

$$PT_{size}(N) = \frac{N^2 \cdot \texttt{sizeof(double)}}{4,096} \cdot \texttt{sizeof(pointer)}$$

This means that the addresses in the page table for a matrix of order $N = 4,000,000$ consume $PT_{size}(4,000,000) = 2.5 \times 10^{11}$ bytes, i.e., $250\,\text{GB}$ on a system where double-precision floating-point numbers and addresses take 8 bytes. Thankfully, the x86-64 architecture supports several page sizes, known as "huge pages" in Linux. Typically, these pages are around $2\,\text{MiB}$ (instead of $4\,\text{KiB}$), although other sizes ($2\,\text{MiB}$ to $256\,\text{MiB}$) are possible as well. Changing the page size requires administrator (root) privileges as the Linux kernel support for *hugepages* needs to be activated and a `hugetlbfs` file system must be mounted. After at least one huge page has been allocated, the path of the allocated file system can then be passed on to SimGrid. Setting the page size to $2\,\text{MiB}$ reduces drastically the page table size. For example, for a matrix of order $N = 4,000,000$, it shrinks from $250\,\text{GB}$ to $0.488\,\text{GB}$.

VI. Scalability Evaluation

In Section V we explained the problems we encountered when trying to run a large-scale simulation on a single node and how we solved them. For the most part, we identified and eliminated bottlenecks one after another while simultaneously making sure that the accuracy of our performance prediction was not impacted. Certainly, the main goal was to reduce the complexity from $\mathcal{O}(N^3) + \mathcal{O}(N^2 \cdot P \cdot Q)$ to something more reasonable. The $\mathcal{O}(N^3)$ was removed through 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 completely get rid of the $\mathcal{O}(N)$ complexity without modifying HPL more profoundly.

Although our goal was to model and simulate HPL on the Stampede platform, we decided to conduct a first evaluation on a similar, albeit non-existing, platform comprising 4,096 8-core nodes interconnected through a $\langle 2; 16, 32; 1, 16; 1, 1\rangle$ fat-tree topology built on ideal network links with a bandwidth of $50\,\text{GB/sec}$ and a latency of $5\,\mu\text{sec}$. We ran simulations with 512; $1,024$; $2,048$ or $4,096$ MPI processes and with matrices of orders 5×10^5, 1×10^6, 2×10^6 or 4×10^6. The impact of the matrix order on total makespan and memory is illustrated in Figure 7. With all previously described optimizations enabled, the simulation with the largest matrix took close to 47 hours and consumed $16\,\text{GB}$ of memory whereas the smallest one took 20 minutes and $282\,\text{MB}$ of memory. One can also see that, when the matrix order (N) is increased, memory consumption and simulation time both grow slightly quadratic as the amount of matrix elements is N^2 and the number of steps of the algorithm also linearly.

Moreover, all the simulations spend less than $10\,\%$ of their execution time in kernel mode, which means the number of system calls is reasonably low.

VII. Modeling Stampede and Simulating HPL

A. Modeling Stampede

1) Computations: Each node of the Stampede cluster comprises two 8-core Intel Xeon E5-2680 8C $2.7\,\text{GHz}$ CPUs and one 61-core Intel Xeon Phi SE10P (KNC) $1.1\,\text{GHz}$ accelerator that is roughly three times more powerful than the two CPUs and can be used in two ways: either as a classical accelerator, i.e., for offloading expensive computations from the CPU, or by compiling binaries specifically for and executing them directly on the Xeon Phi. While the accelerator's $8\,\text{GiB}$ of RAM are rather small, the main advantage of the second approach is that data does not need to be transferred back and forth between the node's CPUs and the accelerator via the x16 PCIe bus.

The HPL output submitted to the TOP500 (Figure 3) does not indicate how the KNC was used. However, because of the values assigned to P and Q, we are certain that only a single MPI process per node was run. For this reason, it is likely that the KNC used as an accelerator. With Intel's Math Kernel Library (MKL), this is effortless as the MKL comes with support for automatic offloading **for** selected BLAS functions.

	CPU (CPU)		KNC (PHI)	
	Coefficient [sec /Flop]	Intercept [sec]	Coefficient [sec /Flop]	Intercept [sec]
DGEMM	1.029×10^{-11}	2.737×10^{-2}	1.981×10^{-12}	6.316×10^{-1}
DTRSM	9.882×10^{-12}	4.329×10^{-2}	1.954×10^{-12}	5.222×10^{-1}

```
#define HPL_dtrsm(layout, Side, Uplo, TransA, Diag, M, N, alpha, A, lda, B, ldb) ({ \
    double expected_time;                                                          \
    double coefficient, intercept;                                                 \
    if((M) > 512 && (N) > 512) {                                                   \
        coefficient = (double)SMPI_DTRSM_PHI_COEFFICIENT;                          \
        intercept = (double)SMPI_DTRSM_PHI_INTERCEPT;                              \
    } else {                                                                       \
        coefficient = (double)SMPI_DTRSM_CPU_COEFFICIENT;                          \
        intercept = (double)SMPI_DTRSM_CPU_INTERCEPT;                              \
    }                                                                              \
    if((Side) == HplLeft) {                                                        \
        expected_time = coefficient*((double)(M))*((double)(M))*((double)(N));     \
    } else {                                                                       \
        expected_time = coefficient*((double)(M))*((double)(N))*((double)(N));     \
    }                                                                              \
    expected_time += intercept                                                     \
    if(expected_time > 0)                                                          \
        smpi_execute_benched(expected_time);                                       \
})
```

Figure 8. Modeling automatic offloading on KNC in MKL BLAS kernels.

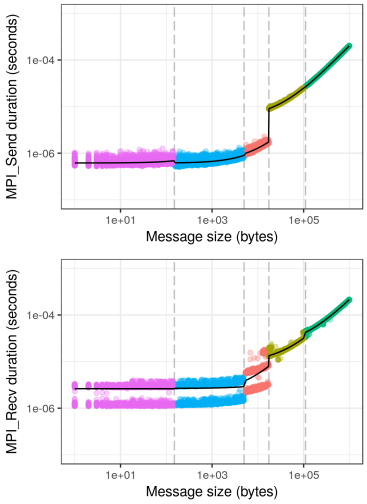


Figure 9. Modeling communication time on stampede. Each color is manually adjusted and corresponds to a different synchronization mode (eager, rendez-vous,...).

Unfortunately, we do not know which MKL version was used in 2013 and therefore decided to use the default version used on Stampede in the beginning of 2017, i.e., version 11.1.1. 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. 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 complexity for DGEMM is always of the order of $M \cdot N \cdot K$ ($M \cdot N^2$ for DTRSM) but the model that describes the time it takes to run DGEMM (DTRSM) is very different for small and large matrices. The table in Figure 8 indicates the parameters of the linear regression for the four scenarios (DGEMM or DTRSM and CPU or Phi). The measured performance was close to the peak performance: e.g., for DGEMM on the Phi reached $2/1.981 \times 10^{-12} = 1.009\,\mathrm{TFlop\,s^{-1}}$. Since the granularity used in HPL (see Figure 3) is 1024, all calls (except for maybe the very last iteration) are offloaded to the KNC. In any case, this behavior can easily be accounted for by replacing the macro in Figure 5(a) by the one in Figure 8.

2) Communications: We unfortunately do not know for sure which version of Intel MPI was used in 2013, so we decided to use the default one on Stampede in May 2017, i.e., version 3.1.4. As explained in Section IV, SMPI's communication model is a hybrid model between the LogP family and a fluid model. For each message, the send mode (e.g., fully asynchronous, detached or eager) is determined solely by the message size. It is hence possible to model the resulting performance of communication operations through a piece-wise linear model, as depicted in Figure 9. For a thorough discussion of the calibration techniques used to obtain this model, see [12]. As illustrated, the results for MPI_Send are quite stable and piece-wise regular, but the behavior of MPI_Recv is surprising: for small messages with a size of less than 17 420 B (represented by purple, blue and red dots), one can observe two modes, namely "slow" and "fast" communications. "Slow" operations take twice longer and are much more common than the "fast" ones. We observed this behavior in several experiments even though both MPI processes that were used in the calibration were connected through the same local switch. When observed, this "perturbation" was present throughout the execution of that calibration. Having taken into consideration that small messages are scarce in HPL, we eventually decided to ignore this phenomenon and opted to use the more favorable scenario (fast communications) for small messages. We believe that the impact of our choice on the simulation accuracy is minimal as primarily large, bulk messages are sent that make use of the *rendez-vous* mode (depicted in dark green).

Furthermore, we configured SMPI to use Stampede's network topology, i.e., Mellanox FDR InfiniBand technology with $56\,\mathrm{Gbit\,s^{-1}}$, setup in a fat-tree topology (see Figure 2). We assumed the routing was done through D-mod-K [17] as it is commonly used on this topology.

3) Summary of modeling uncertainties: For the compiler, Intel MPI and MKL, we were unable to determine which version was used in 2013, but decided to go for rather optimistic choices. The models for the MKL and for Intel MPI are close to the peak performance. It is plausible that the compiler managed to optimize computations in HPL. While it is true that most of these computations are executed in our simulations, they are not accounted for. This allows us to obtain fully deterministic simulations without harming the outcome of the simulation as these parts only represent a tiny fraction of the total execution time of HPL. A few HPL compilation flags (e.g., HPL_NO_MPI_DATATYPE and HPL_COPY_L that control whether MPI datatypes should be used and how, respectively) could not be deduced from HPL's original output on Stampede but we believe their impact to be minimal. Finally, the HPL output reports the use of HPL v2.1 but the main difference between v2.1 and v2.2 is the option to continuously report factorization

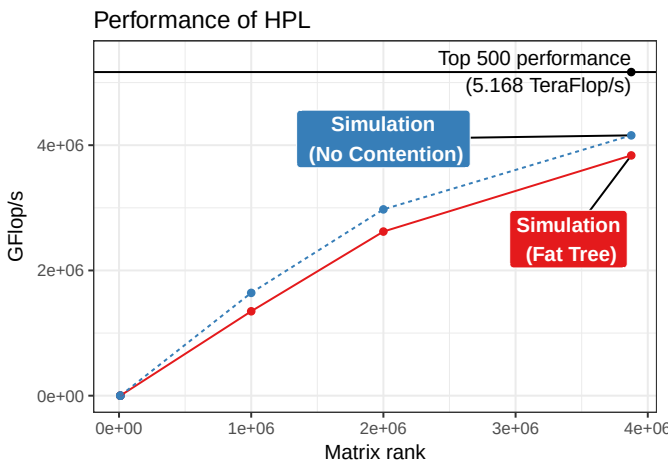


Figure 10. Performance prediction of HPL on Stampede using SimGrid.

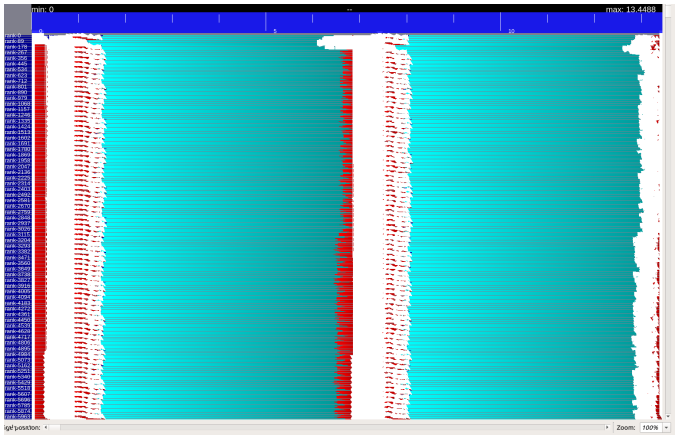


Figure 11. Gantt chart of the first two iterations of HPL. Communication states are in red while computations are in cyan. Each communication between two process is represented with a white arrow, which results in very cluttered white areas.

progress. We hence decided to apply our modifications to the later version of HPL.

With all these modifications in place, we expected the prediction of our simulations to be optimistic yet close to results obtained by a real life execution.

B. Simulating HPL

1) Performance Prediction: 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 Stampede model that we described in the previous section. Although we expected SMPI to be optimistic, the prediction was surprisingly much lower than the TOP500 result. We verified that no part of HPL was left unmodeled and decided to investigate whether a flaw in our network model that would result in too much congestion could explain the performance. Alas, even a congestion-free network model (represented by the dashed blue line in Figure 10) only results in minor improvements. In our experiments to model `DGEMM` and `DTRSM`, either the CPU or the KNC seemed to be used at one 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 $1.2\,\mathrm{TFlop\,s^{-1}}$ node does not sufficiently explain the divide between our results and reality.

2) Performance Gap Investigation: In this section, we explain our investigation and give possible reasons for the aforementioned mismatch (apparent in Figure 10). With SMPI, it is simple to trace 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 609 seconds on a commodity computer and is compressed about 175 MB large). Figure 11 illustrates the very synchronous and iterative nature of the first iterations: One can identify first a factorization of the panel, then a broadcast to all the nodes, and finally an update of 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 the Top500 performance. Overlapping of these heavy communication phases with computation would improve performance significantly. The fact that this is almost not happening can be explained by the look-ahead `DEPTH` parameter that was supposedly set to `0` (see Figure 3). 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 the run in 2013 was executed with an HPL binary provided by Intel and probably specifically modified for Stampede. We believe that some configuration values have been hardcoded to enforce an overlap of iterations with others. Indeed, the shortened part (marked "[...]") in Figure 3 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.

The broadcast and swapping algorithms use very heavy communication patterns. This is not at all surprising since for a matrix of this order, several hundred megabytes need to be broadcast. Although the output states that the `blongM` algorithm was used it could be the case that another algorithm had been used. We tried the other of the 6 broadcast algorithms HPL comes with but did not achieve 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 do not aim to reverse engineer the Intel HPL code. We can, however, already draw two conclusions from our simple analysis: 1) it is apparent that many optimizations have been done on the communication side and 2) it is very likely that the reported parameters are not the ones used in the real execution, probably because these values were hardcoded and the configuration output file was not updated accordingly.

VIII. Conclusions

Studying HPC applications at scale can be very time- and resource-consuming. Simulation is often an effective approach in this context and SMPI has previously been successfully validated in several small-scale studies with standard HPC applications [12], [13]. 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 a similar configuration ran on a single machine for about 62 hours and required 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. The reference run of HPL on Stampede was done several years ago and we only have very limited information about the setup (e.g., software versions and configuration), but a reservation and re-execution on the whole machine was impossible for us. 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. Unfortunately, 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. We believe that some of the HPL configuration parameters were hardcoded and therefore misreported in the output. A quick analysis of the optimized HPL binary confirmed that algorithmic differences were likely to be the reason for the performance differences.

We conclude that a large-scale (in)validation is unfortunately not possible due to the modified source code being unavailable to us. We claim that the modifications we made are minor and are applicable to that optimized version. 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.

We believe being capable of precisely predicting an application's performance on a given platform will become invaluable in the future to aid compute centers with the decision of whether a new machine (and what technology) will work best for a given application or if an upgrade of the current machine should be considered. As a future work, we intend to conduct similar studies with other HPC benchmarks (e.g., HPCG or HPGMG) and with other top500 machines. From our experience, we believe that a faithful and public reporting of the experimental conditions (compiling options, library versions, HPL output, etc.) would be invaluable and allow a better understanding of how such platforms actually behave.

IX. Acknowledgements

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 our TACC colleagues for their support in this study and providing us with as much information as they could.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] C. L. Janssen, H. Adalsteinsson, S. Cranford *et al.*, "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.

[9] 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.

[10] 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.

[11] M. Quinson, C. Rosa, and C. Thiéry, "Parallel simulation of peer-to-peer systems," in *Proc. of the 12th IEEE/ACM Intl. Symposium on Cluster, Cloud and Grid Computing*, Ottawa, Canada, 2012.

[12] 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.

[13] F. C. Heinrich, T. Cornebize, A. 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. [Online]. Available: https://hal.inria.fr/hal-01523608

[14] 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.

[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] D. Balouek, A. Carpen-Amarie, G. Charrier *et al.*, "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.

[17] E. Zahavi, "D-mod-k routing providing non-blocking traffic for shift permutations on real life fat trees," Technion Israel Institute of Technology, Tech. Rep., 2010.