Communication-Aware Load Balancing of the LU Factorization over Heterogeneous Clusters

Lucas Nesi, Lucas Mello Schnorr, Arnaud Legrand

Communication-Aware Load Balancing of the LU Factorization over Heterogeneous Clusters

Lucas Leandro Nesi
Institute of Informatics
PPGC/UFRGS
Porto Alegre, Brazil
lucas.nesi@inf.ufrgs.br

Lucas Mello Schnorr
Institute of Informatics
PPGC/UFRGS
Porto Alegre, Brazil
schnorr@inf.ufrgs.br

Arnaud Legrand
Univ. Grenoble Alpes, CNRS,
Inria, Grenoble INP, LIG
F-38000 Grenoble, France
arnaud.legrand@imag.fr

***Abstract*—Large clusters and supercomputers are rapidly evolving and may be subject to regular hardware updates that increase the chances of becoming heterogeneous. Homogeneous clusters may also have variable performance capabilities due to processor manufacturing, or even partitions equipped with different types of accelerators. Data distribution over heterogeneous nodes is very challenging but essential to exploit all resources efficiently. In this article, we build upon task-based runtimes' flexibility to study the interplay between static communication-aware data distribution strategies and dynamic scheduling of the linear algebra LU factorization over heterogeneous sets of hybrid nodes. We propose two techniques derived from the state-of-the-art 1D×1D data distributions. First, to use fewer computing nodes towards the end to better match performance bounds and save computing power. Second, to carefully move a few blocks between nodes to optimize even further the load balancing among nodes. We also demonstrate how 1D×1D data distributions, tailored for heterogeneous nodes, can scale better with homogeneous clusters than classical block-cyclic distributions. Validation is carried out both in real and in simulated environments under homogeneous and heterogeneous platforms, demonstrating compelling performance improvements.**

***Index Terms*—Data Partitioning, LU Factorization, Load Balancing, Task-Based Applications, Heterogeneous Clusters**

I. Introduction

Large supercomputers exhibit a wide variety of setups composed of traditional CPU cores combined with multiple accelerators such as GPU cards. As we can observe in the latest Top500 list [1], most clusters are made of identical hybrid nodes, facilitating large-scale application deployment with an equal work partitioning among the nodes. Nevertheless, these clusters are subject to variable performance capabilities due to processor manufacturing [2]. They sometimes can also have partitions with different configurations due to capacity upgrades along time or to cope with specific application requirements. For example, the Jean Zay cluster[1] (position 79 in Top500) has 1528 CPU-only nodes (2× 20-cores CPUs), and 261 hybrid nodes equipped with 4× V100 NVidia GPUs.

This study was financed in part by the "Coordenação de Aperfeiçoamento de Pessoal de Nível Superior" - Brasil (CAPES) - Finance Code 001, the National Council for Scientific and Technological Development (CNPq), under grant no 141971/2020-7 to the first author, and the projects: FAPERGS (Data Science – 19/711-6, MultiGPU 16/354-8, and GreenCloud – 16/488-9), the CNPq project 447311/2014-0, the CAPES/Brafitec EcoSud 182/15, and the CAPES/Cofecub 899/18.

[1]https://www.top500.org/system/179699

Many strategies for data distribution for homogeneous nodes of a cluster rely on static data partitioning to avoid a significant communication overhead. A similar approach is possible for heterogeneous nodes, but such a task is quite challenging because traditional programming models such as MPI do not handle very well irregular communication patterns. Modern task-based runtimes like DAGuE, ParSEC, and StarPU provide a high-level programming abstraction that can facilitate the use of sophisticated static data distribution strategies across heterogeneous nodes. When in use, all communications are carried out by the runtime transparently.

In this article, we build upon the flexibility of task-based runtimes to study the interplay between static communication-aware data distribution strategies and dynamic scheduling of the linear algebra LU factorization over homogeneous and heterogeneous sets of hybrid nodes. We focus on the LU operation as most other linear algebra operations lead to similar parallelization challenges and require the same kind of strategies. We review the traditional block-cyclic (BC) strategy, and the state of the art 1D×1D distribution [3] that is asymptotically optimal for heterogeneous configurations. After identifying the potential shortcomings of this distribution in a dynamic scheduling context, we propose two possible improvements and evaluate their performance at scale.

The main contributions of this paper are: **(a)** a demonstration of the regular scalability of 1D×1D data distributions even for homogeneous clusters compared to the traditional BC distribution; **(b)** a constrained data distribution whose cleaner structure may save computing power by gradually concentrating final computations in the fastest nodes (1D×1D-C); **(c)** an improvement to this data distribution by carefully shuffling a few blocks to improve the global load balancing and ensure a regular progression of the computation; **(d)** a comprehensive validation both in real and in simulated environments, comparing our techniques against BC and 1D×1D schemes in homogeneous and heterogeneous platforms, demonstrating compelling performance improvements. We analyze all strategies with a critical eye, clearly outline their performance benefits, and explain their potential harm and why they should be employed with care.

The paper is structured as follows. Section II presents some related work on matrix distribution on homogeneous

and heterogeneous platforms, and how modern task-based runtimes facilitates the use of complex distributions. Section III details our methodology and explains how we relied on both real experiments and detailed simulation in our investigation. Section IV presents our first experiments with state of the art partitioning algorithms, pointing out potential improvements. Section V presents our strategy to constrain 1D×1D to use fewer machines gradually as well as the technique to statically adjust the distribution based on load imbalance estimation along with the iterations. Section VI presents our performance evaluation of heterogeneous distributions using large-scale scenarios. Finally, Section VII discusses the limitations of our approach and concludes our work with some perspectives. The companion material of this work is publicly available at https://gitlab.com/lnesi/cluster2020.

II. Related Work

A. Matrix Distribution for Homogeneous Machines

The *de facto* standard library for high-performance dense linear algebra routines over parallel distributed memory machines is ScaLAPACK [4], which provides efficient and scalable implementations for Cholesky, LU, and QR factorization. We focus on the LU operation as most other operations lead to similar parallelization challenges and require the same kind of strategies. Figure 1 sketches the blocked version of the sequential Gaussian elimination algorithm by relying on three LAPACK kernels: `DGTRF`, `DTRSM`, and `DGEMM`. The first remarkable feature of this algorithm in terms of parallelization is that, when N is large, the two inner loops (`m` and `n`) surrounding the $(N-k)^2$ `DGEMM` constitute the major part of computations and are fully parallel (every `A[m][n]` should be updated independently from the others). In contrast, there are dependencies between the different iterations of the outer loop (even though this can mitigated by look-ahead implementations). The second remarkable feature is that the portion of the matrix updated at each iteration of the outer loop gradually decreases. Therefore, one should make sure that every sub-matrix $A[k \dots N][k \dots N]$ is well distributed between the computation nodes and that all nodes can efficiently participate in each update. For homogeneous resources, there is a good understanding of the needs of this load balancing throughout the execution, employing a 2D block-cyclic distribution (BC for short). Popularized by ScaLAPACK [4], BC takes advantage of the gradual decrease of sub-matrices to provide good load balance. The block-cyclic feature allows a regular balance of columns and rows and hence of the sub-matrix $A[k \dots N][k \dots N]$ over the nodes. In contrast, as we will see in the next subsection, the 2D feature allows for a relatively low communication during the update (proportional to the square root of the total number of nodes) compared to a 1D distribution (where it would be proportional to the total number of nodes). Communication avoiding algorithms relying on 3D and 2.5D data distributions [5] reduce even further communication by replicating data, but they also come at the cost of higher memory usage.

```
for (k = 0; k < N; k++)
  DGTRF-NOPIV(RW, A[k][k]);
  for (m = k+1; m < N; m++)
    DTRSM(RW, A[m][k], R, A[k][k]);
    DTRSM(RW, A[k][m], R, A[k][k]);
  for (n = k+1; n < N; n++) // Update
    for (m = k+1; m < N; m++)
      DGEMM(RW, A[m][n], R, A[m][k],
                        R, A[k][n]);
```

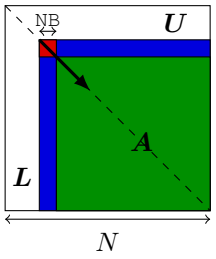

Fig. 1: The LU algorithm (left) without pivoting (for simplification), and the regions of A updated at iteration `k` (right).

B. Matrix Distribution for Heterogeneous Machines

We now review some algorithmic work related to the design of matrix distributions, which are efficient when the set of computing nodes is heterogeneous. Figure 1 (right) shows the $A[k \dots N][k \dots N]$ sub-matrix, which is updated with the product of the row and column `k` (computed by the `DTRSM`). When all nodes have different processing speeds, one should, thus, foremost ensure that each node receives a fraction of the sub-matrix, which is proportional to its speed. Nevertheless, since every node requires fragments of the row and column `k` to perform this update, this requirement induces communication with the nodes owning these blocks. One should thus ensure that the total amount of communications is not too large. This problem, known as the *Peri-Sum* problem, has been widely studied in the 2000s.

Peri-Sum: Given P nodes, let s_i denote the relative speed of nodes so that $\sum_i s_i = 1$. Partition the unit square into P rectangles R_i of dimension (h_i, v_i) so that $h_i \times v_i = s_i$ and $\sum_i h_i + v_i$ is minimal.

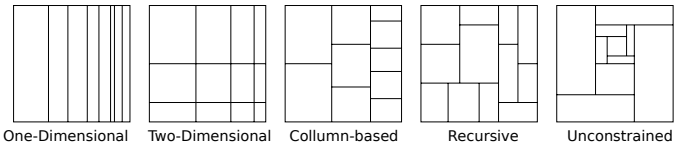

Fig. 2: Taxonomy of unit-square partitions [3].

Kalinov and Lastovetsky [6] first introduced this problem. Although proven to be strong NP-hard in the general case by Beaumont *et al.* [7], Figure 2 illustrates particular classes of partitions for which a solution can be found. Restricting to 1D partitioning is trivial but leads to very tall rectangles, hence to terrible communication costs. Conversely, restricting to 2D partitioning leads to optimal communication costs but rarely allows to reach an optimal load-balancing. Beaumont *et al.* proposed an optimal dynamic programming algorithm for column-based partitions [8], which provides a $7/4$ approximation for the general case. The resulting partition has the good property of grouping the faster nodes together (nodes are sorted by speed before being arranged in columns). Later, Nagamochi and Abe [9] proposed a recursive splitting algorithm based on a Divide-and-Conquer approach, which provides a $5/4$ approximation for the general case.

Algorithm 1: Shuffling a 1D partition into a 1D distribution

$(c_1, \ldots, c_P) = (0, \ldots, 0)$
for $k = N$ **down to** *1* **do**
  $p = \arg\min_{1 \le j \le P}(c_j + 1)/s_j$
  A[k] = p ; $c_p = c_p + 1$
end

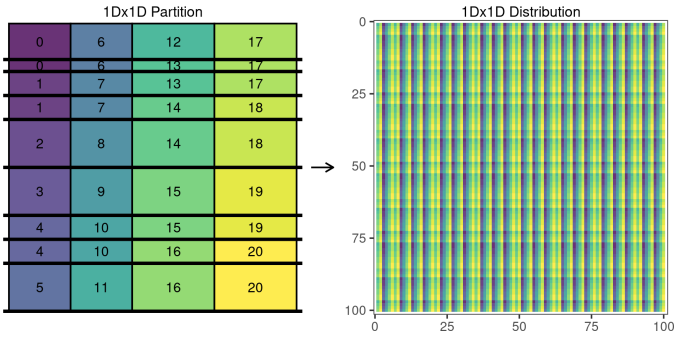

Fig. 3: The 1D×1D partition (left) and the reciprocal 1Dx1D distribution for 14 slow and 7 fast nodes (total of 21 nodes).

Partitions which allow for processing a single Update operation efficiently are thus available. However, a correct shuffling of columns and rows is required to obtain a proper load-balancing throughout the whole execution of the LU algorithm. Beaumont *et al.* [3] proposed a simple shuffling procedure (1D×1D), which is asymptotically optimal, regardless of the initial rectangle partition, and which we now briefly describe as we build upon it in Section V.

Let us first consider a 1D partition. Then the optimal allocation A of columns to nodes can be built through Algorithm 1. It consists of greedily selecting the processor that minimizes the processing time of the sub-matrix $A[k \ldots N][k \ldots N]$. This algorithm produces an allocation that is optimal **for every** $k \in \{1, \ldots, N\}$, and hence optimal overall. An almost optimal distribution can quickly be built from arbitrary partitions by extending processor boundaries (see the left of Figure 3) and applying the previous algorithm for 1D partitions independently to virtual rows and virtual columns. The resulting distribution (see the right of Figure 3) is no more optimal for every $k \in \{1, \ldots, N\}$. However, it is asymptotically optimal, i.e., its execution time is no more than $1 + O(1/N)$ times the one achieved by an ideal distribution.

C. Software Architecture Evolution

Although there have been a few attempts to implement such approaches in frameworks like ScaLAPACK [10], irregular distributions are quite painful to maintain and integrate using MPI, alone especially for more sophisticated factorization algorithms. Furthermore, this static load balancing procedure, which focuses on the main computation kernel (the `DGEMM` of the Update), is never perfect. The small residual imbalance is not satisfying and may even raise more problems than it solves, especially with the growing inner heterogeneity of nodes (multi-GPUs, multi-core).

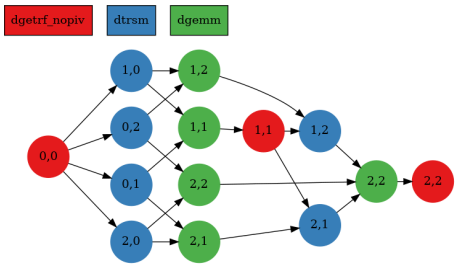

Fig. 4: The DAG of the LU factorization ($N = 3$); each node is a task of a certain type (color) with its block coordinates.

The growing complexity of hardware recently leads to a full reorganization of linear algebra libraries using task-based programming techniques that allow for better synchronization, load-balancing, and internal heterogeneity management. The declarative task-based paradigm is appointed as a path to achieve exascale programs [11]. It works with a description of the application as a graph of tasks (DAG) where edges indicate data dependencies. The code of Figure 1 (left) directly translates through task submission into a DAG, such as the one depicted in Figure 4. Tasks then execute on top of computational processing units, which may be CPU cores or accelerators like GPUs. Sophisticated task scheduling heuristics dynamically define where and when each task will execute [12]. Modern linear algebra libraries, such as PLASMA, MAGMA, and Chameleon, build on task-based runtimes like DAGuE, ParSEC, and StarPU to compute common linear algebra operations on hybrid CPU/GPU nodes. DPLASMA [13] and Chameleon [14] allow for efficient exploitation of clusters with hybrid nodes through MPI.

Nevertheless, despite the programming flexibility brought by this new task-based software architecture, the only available matrix distribution remains the standard BC distribution, which is only adequate for a homogeneous set of nodes. Although there are a few recent works by Eyraud-Dubois and Lambert [15] about communication avoiding algorithms and data distributions for a heterogeneous nodes, they only address the Matrix Multiplication operation inside a hybrid node.

In this work, we revisit the 1D×1D shuffling [3] of column-based partitions [8], in a distributed doubly heterogeneous setup (inside and between nodes) and propose communication-aware load-balancing improvements. This work is possible by taking advantage of the programming flexibility and communication abstraction present on modern runtimes.

III. Experimental Methodology

We adopt StarPU [16] as it provides means for smoothly running the application in multiple nodes, has extensive tracing capabilities, and a simulation mode over Simgrid [17].

The StarPU runtime is a task-based runtime for heterogeneous multi-core and multi-node platforms. It enables an interface for an application to submit tasks to the runtime. StarPU uses the Sequential Task Flow (STF) paradigm [18] for task submission, where the application sequentially submits

the tasks to the runtime that is responsible for scheduling then. In StarPU, each resource (CPU Core, GPU) has an associated entity called worker. The scheduling heuristic in use assign tasks to workers, dynamically. Each task can have different implementations for different resources. For example, a `DGEMM` task can have two implementations, one for CPU and another for GPU. During the execution, StarPU determines the best implementation at a given time. The multi-node execution is possible by using the StarPU-MPI extension [19]. The main difference for the application is that data partitioning among nodes occurs in data declaration, statically. Then each node unrolls the DAG, handling tasks that write on data they own.

We relied on the LU factorization as implemented in Chameleon [14] using a block size of 960×960 for all experiments. To facilitate the evaluation of several distribution strategies, we performed a minimal modification in Chameleon so that distributions can be defined from a simple file instead of the default BC algorithm. This minimal change enables the testing of arbitrary partitions quickly since StarPU will seamlessly handle all the communications among different nodes and heterogeneous resources.

Real executions were conducted on the Grid5000 platform using the `Chifflet` and `Chetemi` clusters. `Chifflet` comprises eight nodes, each one with two Intel Xeon E5-2680 v4 (14 cores/CPU) and two NVIDIA GTX 1080Ti while `Chetemi` comprises 15 nodes, each one equipped with two Intel Xeon E5-2630 v4 (10 cores/CPU). A 10Gbps Ethernet network connects both clusters. We used the StarPU developer branch from April 2020 and the 0.9.2 Chameleon version, modified to accept our custom distributions. Our experiments required some tuning of the StarPU-MPI configuration to obtain better performance. We relied on the MPI low-latency implementation from the NewMadeleine suite [20] in its development version of April 2020 because of our experiments with OpenMPI [21] did exhibit significant idle times incurred by abnormal delays in point-to-point operations. We took care of dedicating a core/thread responsible for task submissions and one core per node to the MPI thread responsible for communications among nodes and bound it to the first NUMA node, where the network card resides. We also took care of binding each GPU worker thread to the NUMA node that is physically attached to the corresponding GPU, considering PCI proximity. We configure the internal StarPU's GPU pipeline to have four stages. We also make the scheduler assume the RAM-GPU-RAM data transfer cost to be ten times larger than calibrated values. This configuration avoids an optimistic view of the PCI bus, especially in the NUMA node attached to the Ethernet card, improving performance. We relied on the NUMA support for StarPU to consider RAM locality when scheduling. The binding of all CPU workers reduces thread migrations achieving more stable results.

To evaluate larger configurations than the one offered by Grid5000, we resorted to simulations using the SimGrid framework [22] version 3.24 combined with the StarPU-Simgrid module [17]. These simulations allow for a full-fledged emulation of the StarPU runtime while simulating the consumption on CPU, GPU, PCI bus, and Ethernet resources. Simulations require performance models for each machine, which come from the real executions at a smaller scale on the `Chifflet` and `Chetemi` machines. The machines that do not have a real counterpart assume the performance models of the last one real machine.

Finally, all the detailed performance analysis of both real and simulated executions use the enriched traces provided by StarPU and the StarVZ visualization framework [23] [24]. We follow the reproducible research and open science principles by making all the code and data available in our companion at https://gitlab.com/lnesi/cluster2020.

IV. Experimenting with State of Art Partitioning

We first present an in-depth analysis of the performance of Chameleon/StarPU-MPI over both homogeneous and heterogeneous clusters using the state of the art BC and 1D×1D distributions. We build upon these experiments to hint potential performance improvements and to show that simulation is sufficiently faithful to conduct such data distribution/scheduling studies as they capture essential details of load imbalance.

A. Homogeneous Cluster with the BC Distribution

Figure 5 depicts the behavior of two runs considering the 15-node `Chetemi` cluster, first in reality (left part), and then in simulation (right). We employ four panels temporally aligned (in the X-axis) for this comparison. The Application Workers panel is a space/time view similar to a Gantt chart, which depicts each CPU worker's behavior among the 15 nodes. This panel includes each node's ABE (Area Bound Estimation, i.e., the optimal runtime when ignoring all dependencies) as a short red line. A red dashed line spanning all the nodes indicates the global ABE, which is thus an absolute lower bound on the makespan whose value lies on the panel's right. Finally, this panel has a rectangle for each node on the far right (they are so thin for these traces that they appear as black lines), which indicates when the node started being fully idle until the completion of the factorization. The next two panels indicate the number of submitted and ready tasks, broken down per node, along time. Finally, the upper panel (Iteration) is a representation of the progression of the computation across iterations. The iteration of the outer loop is represented on the Y-axis while time remains on the X-axis. Each tiny green rectangle indicates that one node is working on a particular iteration task during a particular time interval. The three black lines indicate when the first/median/last task of each iteration is processed, hence providing an intuition of how long each iteration is active and how they overlap. For example, if the runtime did synchronize at each iteration, the horizontal green lines would not overlap. Fortunately, we can see here that StarPU does an excellent depth-first progression on the DAG, enabling a swift release of work on every node.

We can see that the BC distribution keeps all nodes active until the very end of the execution. This excessive activity generates many small communications at the end, resulting in generalized idle time. Using fewer nodes toward the end may

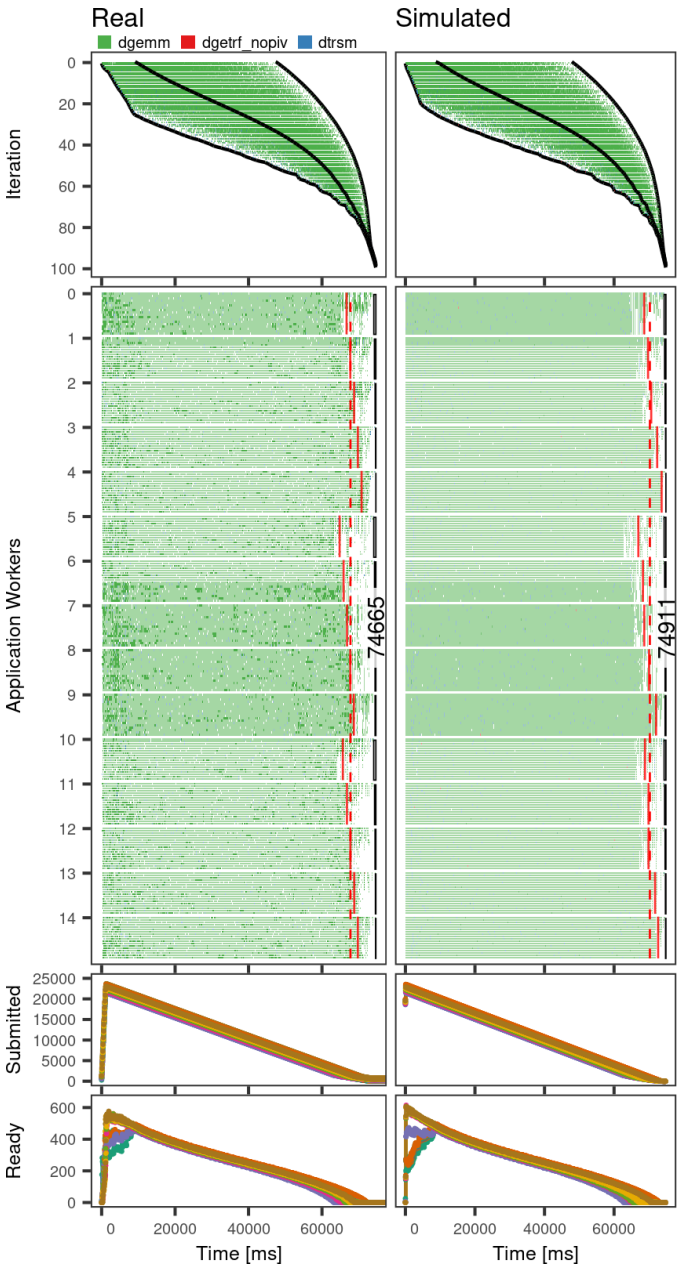

Fig. 5: The 3×5 BC partitioning using 15 identical CPU-only nodes to compare the behavior of a real execution (left) against the simulation (right): the simulation captures the load imbalance coming from the BC distribution.

thus be beneficial and result in both a slightly better makespan and resource usage, as communicating less and stopping some nodes sooner to put them in a deep sleep mode could save energy. Furthermore, the per-node ABE indicates that although the BC distribution leads to a fair load balance, it is not perfect because of the rounding induced by the rectangular partition that operates at the granularity of rows and columns of blocks.

We can also observe that the simulation also matches the real execution well both in terms of general metrics and the specific behavior of application workers. The first global indicator of a faithful simulation is the makespan, which is within the bounds of the natural variability of real executions. Although simulation traces are a bit more "clean", if we look at each node's specific behavior, we can see that there is an almost identical amount of work and per-node ABE estimation in reality and simulation. Note that minor ABE differences may also appear from the natural day to day performance variability of nodes that may not entirely correspond to the model built from a previous experiment. Global metrics such as submitted and ready tasks and the upper iteration plots also look very similar along time, which indicates an identical progression throughout the DAG.

B. Heterogeneous Cluster with the 1D×1D Distribution

Figure 6 depicts the behavior of two runs considering a (7+14)-node platform (7 fast `Chifflet` nodes comprising GPUs and 14 slow `Chetemi` nodes) in a real configuration (left part) and in simulation (right). We use the same panels as already described for Figure 5. The difference is that the last seven machines (from id 14 to 20) have 2× GPUs each.

The ABE indicates that the 1D×1D algorithm manages to handle the heterogeneity of the machine well by giving each node an amount of work proportional to its capacity. Unfortunately, the nodes from 0 to 13 without GPUs end up with an ABE that is larger than the one from the nodes with GPUs. The distribution also appears sub-optimal because of the long idle periods toward the end of the run, especially in the nodes equipped with GPUs. Similarly to the BC distribution, we can also imagine that using fewer nodes toward the end may bring performance improvements by concentrating all the last blocks in the more powerful nodes.

The comparison between real and simulated runs indicates that the simulation is once again sufficiently accurate to capture general scheduling trends as well as details. The ABEs and the completion time structure are very similar. The general metrics (submitted and ready tasks) also have similar shapes, except for task submission, which is faster in simulation on nodes comprising GPUs. Indeed, in StarPU-SimGrid, the submission overhead is uniform over nodes, but we decided not to tune this for the heterogeneous context finely. This choice results in a slight difference in node progression visible on the upper panel (with more iterations overlapping in simulation) and a slightly more optimistic ($\approx 2-3\%$) makespan prediction. Nevertheless, our other comparisons in various scenarios lead us to consider this effect to be of sufficiently little significance.

C. Discussion and Motivation

Our experiments with the state-of-the-art algorithms in homogeneous and heterogeneous setups show that StarPU-SimGrid is capable of delivering very faithful simulations with multiple nodes. While previously published results only considered inner-node behavior [17], we now have reliable multi-node estimations both for the general metrics and the ABE that reflects the load imbalance. This combination allows us to rely on simulation to run large-scale experiments in a reproducible way with a good level of confidence.

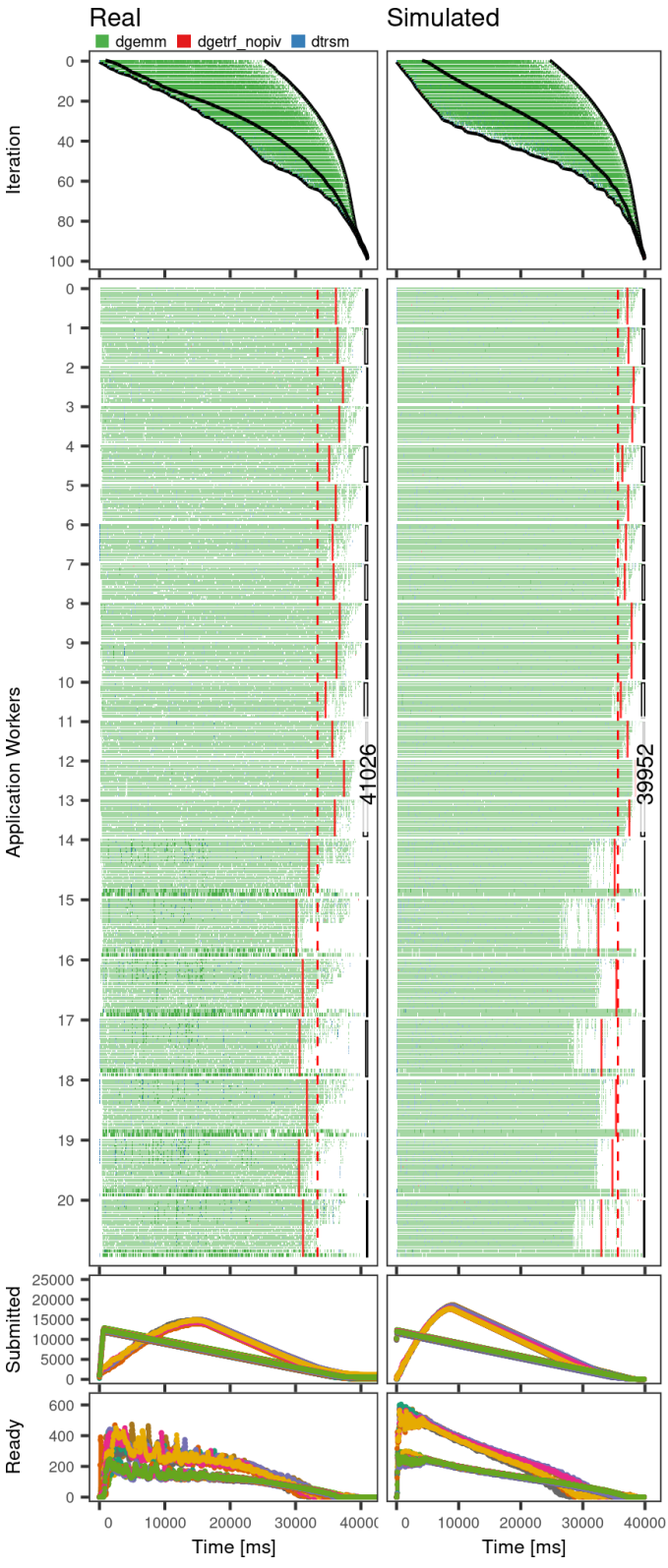

Fig. 6: The 1D×1D partitioning using a configuration with 14 CPU-only nodes plus 7 GPU-equipped nodes to compare the behavior of a real execution (left) against the simulation (right): the simulation is optimistic regarding the makespan but captures the general trend of the load imbalance coming from the 1D×1D algorithm.

We also observe that these algorithms are not optimal across nodes, as pointed out by the per-node ABE estimations. One could expect that per-node ABEs, computed after the load partitioning, would be very similar. Although it is relatively good, there is room for improvements as indicated by the uneven distribution of red lines on the application workers panels of Figures 5 (for BC) and 6 (for 1D×1D).

V. Balancing Work and Using Fewer Nodes

We explain the fact per-node ABE imbalance by the fact that, first, the 1D×1D distribution is only asymptotically optimal, and only the `DGEMM` computation kernels are taken into account when computing the column-based partition. Second, all the fastest nodes (those with GPUs) are not only slightly less loaded but mostly idle toward the end of the execution. Using all 21 nodes toward the end incurs many synchronizations and communications. The computation would end sooner if only the faster nodes were working. Furthermore, using slow nodes toward the end may also negatively impact the progression along the critical path.

A. Constraining the 1D×1D distribution

We propose to modify the 1D×1D algorithm to enforce the use of only the faster nodes toward the end of the execution. The allocation is built incrementally similarly to the 1D×1D algorithm, by picking the best virtual row and column of processors for each k from N down to 1, but by restricting to *sections* of virtual rows and columns.

We start with a section made of a single virtual column (right-most column on the partition, with the fastest nodes) and a single virtual row (the largest one from the fastest node). Let us denote by $W_p(k)$ the total work (in FLOPS) induced by processing sub-matrix $A[k \dots N][k \dots N]$ on node p. By dividing this work by the speed of the node, we obtain $ABE_p(k)$, the minimum time required to process iteration k on node p and $ABE(k) = \max_p ABE_p(k)$, the minimum time required to process iteration N for a given distribution. Since we do not redistribute the matrix during the factorization, this global $ABE(k)$ is always larger than the absolute $ABE^*(k)$ defined as the total work of iteration k divided by the total speed of all the active nodes of the section. We also compute the critical path bound $CPB(k)$ of this iteration, i.e., the duration of the largest sequence of tasks in the DAG (considering communications). The metric would be the optimal processing time of iteration k if an infinite number of resources were available on each node. Since initially (when $k \approx N$) there is very little work in the sub-matrix $A[k \dots N][k \dots N]$, we start with $ABE(k) = ABE^*(k) < CPB(k)$. When $ABE^*(k)$ becomes larger than $CPB(k)$, some work of k will surely have to wait for resources, and we should add new nodes.

We then create a new section by adding one new column of processors (from the right to the left of the partition, as column-based partitions of [8] sort nodes by their processing speed) and several virtual rows sorted by decreasing height and involving the fastest possible nodes. As the partition has many more virtual rows than columns, we benefit from releasing

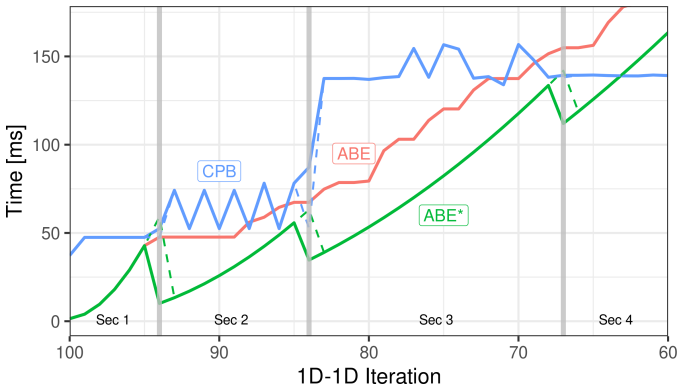

Fig. 7: The 1D×1D-C metrics (CPB, ABE^* and ABE) for 7+14 machines from iteration 100 to 60 with four sections.

them faster than columns to obtain a better load balance. In Figure 3, we first release node 20, then nodes 14 – 16, 18, 19; then nodes 6 – 11, 12, 13, 17; and finally all the nodes.

After adding the new virtual columns and lines, the algorithm keeps allocating rows and columns of the matrix using the 1D×1D procedure while updating $ABE(k)$ and $CPB(k)$. The progression of these metrics is showed on Figure 7, where the blue line is the CPB metric, the green line is the $ABE^*(k)$ per iteration considering all available resources, the red line is the $ABE(k)$, and the gray vertical lines point whenever $ABE^*(k)$ would become larger than $CPB(k)$ to create a new section. The $ABE^*(k)$ keeps increasing quadratically within a section, proportionally to the sub-matrix, until it exceeds the $CPB(k)$, as indicated by dashed lines. Therefore, we always have $ABE^*(k) \leq CPB(k)$ unless we run out processors and cannot create a new section.

In our example of Figure 7, we allocate a 100×100 matrix on 14 (slow) `Chetemi` and 7 (fast) `Chifflet`, which leads to four sections. The first section corresponds to iterations 100-96 and contains only one `Chifflet`. Then the section from iterations 95-85 contains six `Chifflet` nodes. Then the section from iterations 84-68 contains seven `Chifflet` and eight `Chetemi` nodes. Finally, the section from iterations 67-1 contains all the nodes. In Figure 7, one may note that the CPB varies from an iteration to another, which may seem surprising since the longest path in the sub-DAG corresponding to iteration `k` remains the same throughout iterations. The MPI communications incurred by using several nodes justify the first increase of CPB in the second section. The second (and much larger) increase of CPB in the third section appears because of the addition of slow nodes (`Chetemi`). Small differences in the tasks' execution times between machines cause remaining fluctuations.

The upper left part of Figure 8 depicts the final resulting data distribution, called 1D×1D-C, which is way less regular than the original 1D×1D distribution (see Figure 3) and favors the use of the faster nodes toward the end. In contrast, the distribution of the upper parts of the matrix remains relatively similar. The bottom left part of Figure 8 depicts the execution obtained with this constrained distribution. A first notable difference with the execution of a 1D×1D distribution (see Figure 6) is that now, as indicated by the rightmost white rectangles on each node (A), the slower processors stop working almost instantaneously instead of vainly contributing to the end of the computation. Unfortunately, even though slower nodes finish their work much sooner, the load is still very imperfectly imbalanced (as illustrated by the red lines). The faster nodes still exhibit critical idle periods (B), indicating there is still room for improvement.

B. *Shuffling Blocks*

When using the 1D×1D, one often noticed some idle times in the middle of execution, relatively early. We have identified that the work imbalance among nodes in the earlier iterations is the cause of these idle times. Remember that with a runtime like StarPU, there is no synchronization between each iteration. Nevertheless, if some nodes progress faster across iterations than others, they may quickly run out of work. Let us consider the cumulative ABE per node up to iteration k, i.e. $CABE_p(k) = \sum_{j=1}^{k} ABE_p(j)$. As illustrated on the left of Figure 9, where the group of lines represents the $CABE$ per node, and the red bottom line represents the difference of the $\max_p CABE_p(k)$ and $\min_p CABE_p(k)$. For some iterations, there is a significant imbalance, so the nodes that have less work have to wait for the most loaded ones. The waiting times appear because 1D×1D-C tries to balance $ABE(k)$ for k starting from N downward 1, which corresponds to the reverse order from what the LU algorithm does.

To alleviate this load imbalance, we propose a Shuffling mechanism that moves blocks around (therefore potentially inducing more communications since node alignments may become broken). Shuffling occurs within a section, say iteration k_1 to k_2, and aims at balancing the total work of this section (remember 1D×1D is only asymptotically optimal), $SABE_p = \sum_{j=k_1}^{k_2} ABE_p(j)$. The fastest node gets the right-bottom most block of the slower node once we determine the least loaded and the most loaded node. This process repeats until one of the three possible conditions becomes true. (i) The difference between the $SABE$ of the nodes falls under a threshold (in our experiments, we use 20ms). (ii) There is no more block to move from the fastest node (the $SABE$ may be particularly high because of blocks from previous sections). (iii) Moving a block would result in trespassing the average $SABE_p$ on the least loaded node. The right of Figure 9 shows how cumulative $CABE$ evolves after two shuffling operations are applied, where the red line represents the difference between the maximum and minimum $CABE(k)$. The top right of Figure 8 depicts the resulting irregular distribution, called 1D×1D-C+S, with the runtime behavior.

We first apply this shuffling to the most significant section. Usually, if the work is sufficient for the total amount of blocks, it will be the first section containing all nodes. It may be the case that after a shuffling, the cumulative load imbalance ($\max_p CABE_p(k) - \min_p CABE_p(k)$) remains high, in which case we incur an additional shuffling at this

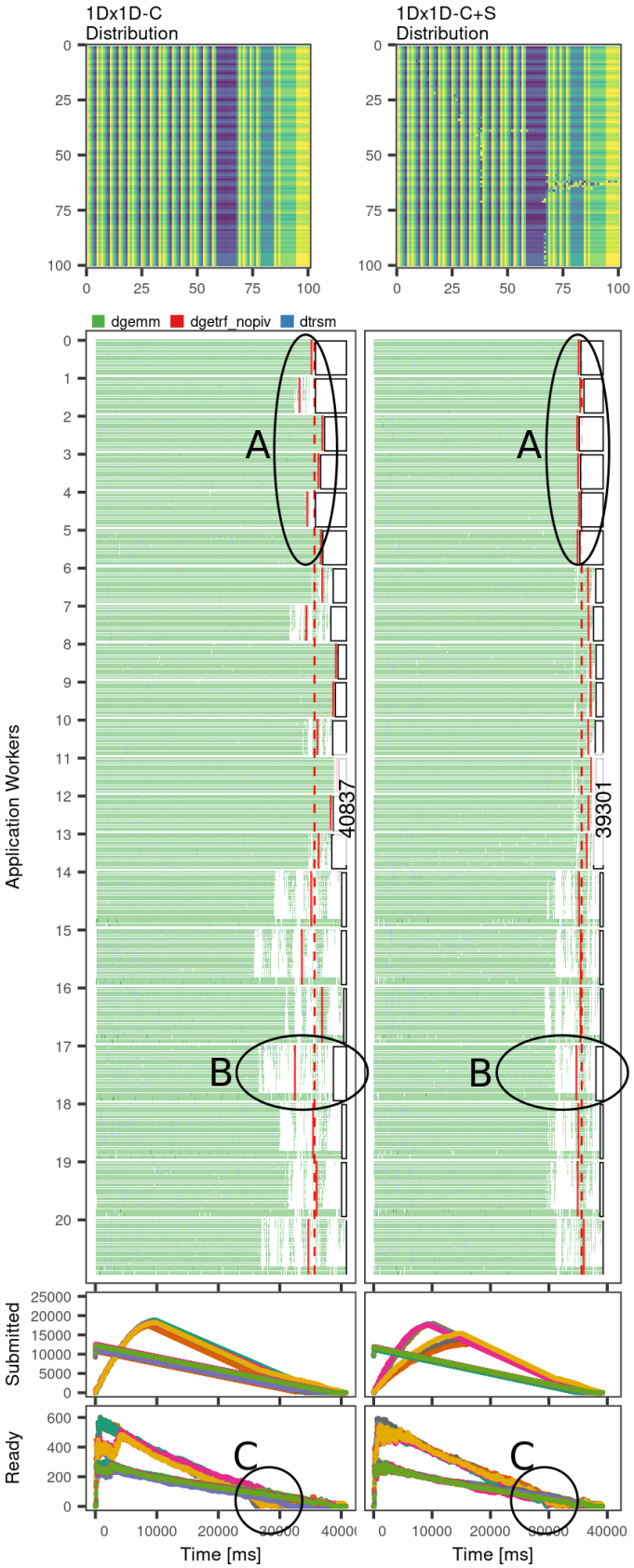

Fig. 8: When using 14 CPU-only nodes plus 7 GPU-equipped nodes, we illustrate the distribution (top) and behavior (bottom) of the 1D×1D-C (left) and the 1D×1D-C+S (right) runs.

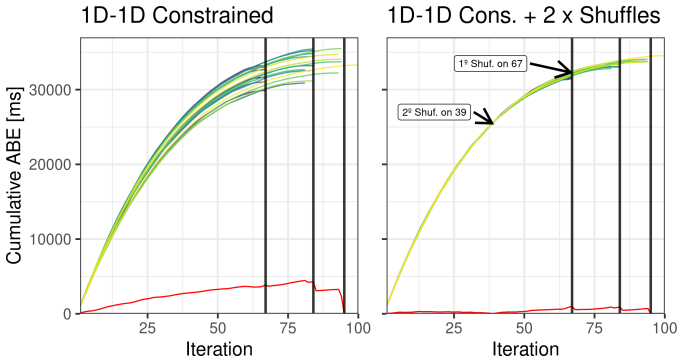

Fig. 9: Cumulative ABE prior and after two shufflings.

spike. When applied carefully, this shuffling operation can allow a smooth progression of the runtime across iterations and reduce every visible idle period.

The upper right part of Figure 8 depicts the block shuffled distribution, where small glitches can be noticed and correspond to the few blocks allotted to faster nodes. A noteworthy aspect of the corresponding execution, depicted in the bottom right of Figure 8, is that now the ABE of each node is almost perfectly balanced (see for example the comparison of ellipses A for the first six nodes). In this particular example, the makespan improvement is not significant: 39.30 seconds compared to 39.94 for the original 1D×1D distribution. However, the activity period of all slow nodes now matches their ABE, which may allow to use them for another computation or merely to put them into a deep sleep mode to save energy. Idle periods (see ellipses B) on the faster nodes are significantly reduced compared to the distribution without block shuffling. Idling corresponds to the small load imbalance, which is maximum for iteration 84 and visible in Figure 9. They appear because of the lack of ready tasks (see ellipses C). Our experiences to balance this load further indicate that it is particularly difficult (there is very little work) and that shuffling should be applied with much care as it involves balancing the total load from the beginning with the remaining one.

VI. Performance Evaluation

We evaluate the efficiency of BC, 1D×1D, 1D×1D-C, and 1D×1D-C+S distributions in three different scenarios. We start in Section VI-B by a quick illustration of the gain of 1D×1D distributions over BC distributions in a classical homogeneous setting with a strong-scaling scenario with a fixed matrix size and where we gradually increase the number of nodes. Then, we show the gains brought by 1D×1D-C and 1D×1D-C+S when strong scaling on a heterogeneous platform. Finally, we present in Section VI-C how distributions compare on a heterogeneous 46 nodes cluster in terms of performance when growing the matrix size. Note that we now employ all available cores in our simulations since the performance loss caused by the absence of dedicated cores in real-life experiments is absent in the simulation.

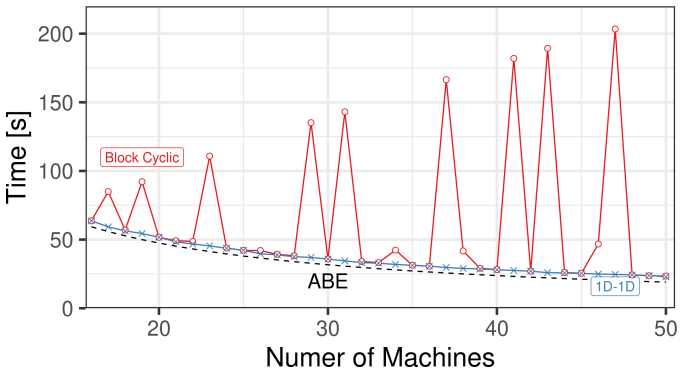

Fig. 10: Strong scaling for a 100×100 matrix in a homogeneous 50-node cluster of `Chetemis`; while 1D×1D gracefully scales, BC handles very severely the prime number of machines that incur lots of communications.

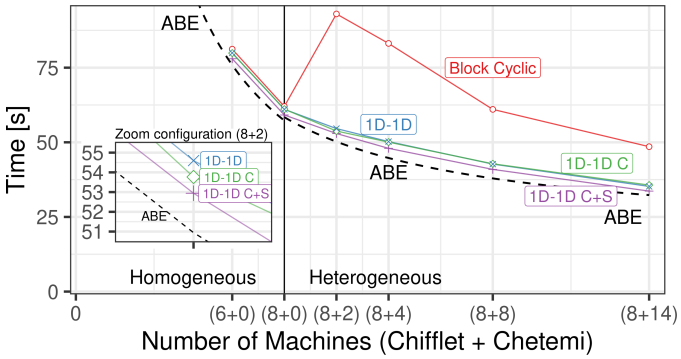

Fig. 11: Execution time (Y-axis) as a function of combinations of number of machines (X-axis) and distributions (lines).

A. Strong Scaling in a Homogeneous Context

We adopt a fixed matrix size of 100×100 blocks of 960×960, and we gradually increase the number of `Chetemi` computing nodes from 16 to 50. The BC strategy is common for homogeneous platforms. Unfortunately, it is known that when the number of machines does not nicely decompose in prime numbers, the communication overhead may lead to significant performance degradation. Indeed, for any prime number of machines, one ends up with a pure one-dimensional distribution. Instead, as illustrated in Figure 10, the 1D×1D distribution handles such situations gracefully and scales perfectly. Although 1D×1D distributions originated 20 years ago, to the best of our knowledge, it is the first time they appear in a real solver stack. Interestingly, the distance between makespans of the 1D×1D distribution and the (very optimistic) ABE bound of all resources is constant, which corresponds to the very end of the execution that cannot efficiency exploit all computing nodes. However, in such a regular and homogeneous context, constraining and shuffling are of little interest and may even be harmful as the load balance is optimal.

B. Strong Scaling in a Heterogeneous Context

We now consider the scenario where we first use `Chifflet` nodes as they comprise GPUs and gradually add the `Chetemi` nodes for a heterogeneous platform at the end. Figure 11 presents the performance of every distribution in terms of execution time (on the Y-axis) as a function of the size of the platform (on the X-axis). As expected, as soon as slower nodes participate, the time required by the BC distribution dramatically increases since now the whole execution progress at the speed of the slower nodes. In contrast, 1D×1D distributions gracefully handle the addition of new nodes and are all very close to the optimal (the ABE with all resources). Although the gain is relatively small (1 to 2 seconds), 1D×1D-C and 1D×1D-C+S improve systematically upon 1D×1D. The inset zoom with the (8+2) node configuration clearly shows differences between strategies. For larger configurations, the difference between 1D×1D and 1D×1D-C decreases while 1D×1D-C+S maintains a constant gain of about 2 seconds over 1D×1D. This constant gain appears because 1D×1D-C+S manages to ensure smooth progress throughout iterations and to optimize the end of the execution.

A consequence of constraining the 1D×1D-C distribution to use fewer nodes toward the end of the computation allows to release nodes earlier (see Section V). Figure 12 provides, for every configuration, the total activity time of the nodes from the first to the last computation task each node executes (we do not include BC as it is terrible compared to the 1D×1D variants). This metric is of interest as it relates to the total energy consumption provided nodes can be put in a deep sleep state when they are fully idle. As expected, the 1D×1D-C and 1D×1D-C+S significantly improve upon 1D×1D.

Finally, it is also interesting to evaluate how much communication is incurred by these distributions. Indeed, although they are all derived from the column-based *Peri-Sum* partition, the constraining and the shuffling trade-off some communications for a better load balancing throughout the whole execution. When factorizing a 100×100 matrix with 8+14 nodes, a pure BC requires transferring $55\,329$ blocks, while the 1D×1D reduces this down to $35\,417$ blocks, and 1D×1D-C+S requires the transfer of $42\,632$ blocks. It is thus important to understand that although the 1D×1D-C+S allows for better exploitation of computing resources, some communication overhead costs may become problematic at a larger scale.

C. Performance Gain over a Large Heterogeneous Cluster

We now use a 46 node cluster made of 16 fast `Chifflet` nodes and 30 slow `Chetemis` nodes, and we gradually increase the matrix size. The goal is thus to evaluate how quickly we reach the maximum performance (in GFlops). Figure 13 presents the performance of each distribution for matrix sizes ranging from 25×25 blocks (which is very small since each node gets less than 14 blocks in average for a BC distribution) to 150×150 blocks. Similarly to the previous evaluation, we depict the maximum achievable performance computed from the ABE (considering all resources) and the CPB (without any communication). As expected, the best

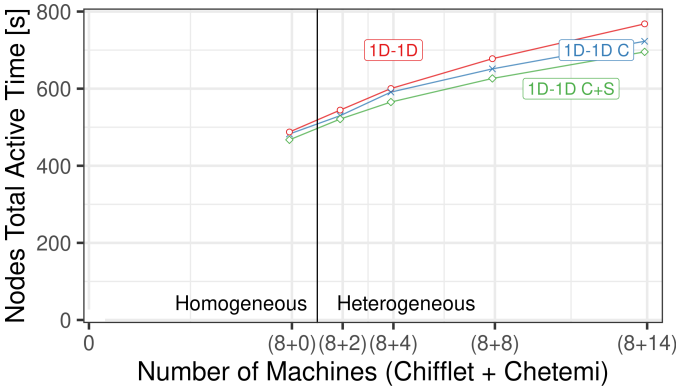

Fig. 12: Total machine utilization time (Y-axis) for different number of machines (X-axis) and data distributions (lines).

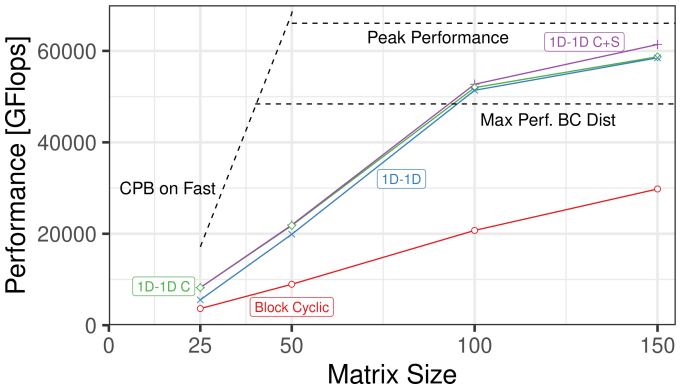

Fig. 13: GFlops Performance (Y-axis) for different matrix sizes (X-axis) and distributions (lines) for a case with 16+30 nodes.

performance achieved by a BC distribution is far smaller than the peak performance, and the BC distribution is still far from it even for the 150×150 matrix (about 40% less). The rather unfavorable geometry for BC distributions ($46 = 2 \times 23$) has many communications, explaining these results. Variants of the 1D×1D distribution remain undisturbed by the prime decomposition of the number of nodes, getting quickly closer to the peak performance (within 6% for the 150×150 matrix).

Again, the 1D×1D-C+S distribution obtains systematic gains over both 1D×1D-C and 1D×1D distributions. It is interesting to note that for small 25×25 matrices, 1D×1D-C+S and 1D×1D-C distributions are equivalent (there are not enough blocks to shuffle) and significantly improve upon the 1D×1D distribution (32.5% gain). For large 150×150 matrices, 1D×1D-C and 1D×1D-C obtain a similar performance which reflects the asymptotically optimal, but imperfect, load balance. On the other way round, the 1D×1D-C+S significantly improves the overall load balancing. Last, regarding communications, the factorization of a 150x150 matrix requires the communication of 129 164 blocks for 1D×1D. Constraining the execution on fewer nodes towards the end allows us to decrease this amount to 123 780 blocks, but shuffling blocks to achieve a perfect load balancing increases this amount up to 149 474 blocks.

VII. Discussion and Conclusion

We present an in-depth study of static data distribution techniques for both homogeneous and heterogeneous sets of nodes in the context of dynamic task-based runtimes. By using the linear algebra LU factorization, we demonstrate how the 1D×1D distributions can be beneficial even for a homogeneous set of hybrid nodes when compared against BC, presenting much more stable scalability. In experiments with a heterogeneous set of hybrid machines, for which the 1D×1D is also near-optimal, we propose two modifications. The first is to use fewer and fewer nodes towards the end of the factorization, gradually intensifying the computation in the more powerful machines. The second is to apply additional shuffling of blocks targeting the area bound estimation. The constraining technique allows for optimizing the end of the run, while block shuffling allows the reach of a quasi-optimal load balance across iterations and ensures a smooth progression. A careful combination of these techniques provides compelling performance improvements compared to the original 1D×1D, both in load balancing and execution time.

Our improvements for 1D×1D should also work for Cholesky or other factorization operations, although this should be verified. Note that the optimal distribution may be different, which raises more generally the question of designing heterogeneous distributions which are efficient throughout the execution of a complex sequence of different linear algebra operations, without having to move data between each of them.

Another takeaway message from our study is that 1D×1D distributions are an excellent starting point, which is not so easy to improve. In practice, it is not clear that the systematic use of 1D×1D-C and 1D×1D-C+S are beneficial as it requires a good performance model and may induce communication overhead. The uncertainty on node performance leads us to believe that online adjustments may be necessary [25]. Understanding the potential and shortcomings of static scheduling was thus necessary. This first study allowed us to identify essential quantities for performing such re-balancing and the consequences of having irregular distributions on the overall execution. Online data distribution adjustments would have to be done with great care, as they may interfere with the task submission mechanism. Any development in this direction should thus involve runtime developers to tackle the submission overhead on huge matrices properly.

Finally, although we provided insights on the total amount of communication induced by these load distributions, StarPU-MPI does relatively few optimizations regarding their organization than, for example, HPL, which may be harmful at massive scale. The recent work of Denis *et al.* [26] on dynamic broadcasts should significantly improve this and allow us to investigate really large supercomputers.

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).

REFERENCES

[1] J. J. Dongarra, H. W. Meuer, E. Strohmaier *et al.*, "Top500 supercomputer sites," *Supercomputer*, vol. 13, pp. 89–111, 1997.

[2] Y. Inadomi, T. Patki, K. Inoue, M. Aoyagi, B. Rountree, M. Schulz, D. Lowenthal, Y. Wada, K. Fukazawa, M. Ueda, M. Kondo, and I. Miyoshi, "Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2807591.2807638

[3] O. Beaumont, A. Legrand, F. Rastello, and Y. Robert, "Static LU decomposition on heterogeneous platforms," *Int. Journal of High Performance Computing Applications*, vol. 15, pp. 310–323, 2001.

[4] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley, and J. J. Dongarra, *ScaLAPACK User's Guide*. USA: Society for Industrial and Applied Mathematics, 1997.

[5] L. Grigori, J. W. Demmel, and H. Xiang, "Communication avoiding gaussian elimination," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008, pp. 1–12.

[6] A. Kalinov and A. Lastovetsky, "Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers," *J. of Par. and Distr. Comp.*, vol. 61, no. 4, p. 520, 2001.

[7] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert, "Partitioning a square into rectangles: NP-completeness and approximation algorithms," *Algorithmica*, vol. 34, pp. 217–239, 2002.

[8] ——, "Matrix multiplication on heterogeneous platforms," *IEEE Trans. Parallel Distributed Systems*, vol. 12, no. 10, pp. 1033–1051, 2001.

[9] H. Nagamochi and Y. Abe, "An approximation algorithm for dissecting a rectangle into rectangles with specified areas," *Discrete Applied Mathematics*, vol. 155, no. 4, pp. 523–537, 2007.

[10] O. Beaumont, V. Boudet, A. Petitet, F. Rastello, and Y. Robert, "A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers)," *IEEE Trans. Computers*, vol. 50, no. 10, pp. 1052–1070, 2001.

[11] J. Dongarra, S. Tomov, P. Luszczek, J. Kurzak, M. Gates, I. Yamazaki, H. Anzt, A. Haidar, and A. Abdelfattah, "With extreme computing, the rules have changed," *Comp. in Sci. Eng.*, vol. 19, no. 3, p. 52, 2017.

[12] S. Thibault, "On runtime systems for task-based programming on heterogeneous platforms," Hab. à diriger des rech., U. Bordeaux, 2018.

[13] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra, "Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 2011, pp. 1432–1441.

[14] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov, "Faster, cheaper, better – a hybridization methodology to develop linear algebra software for GPUs," in *GPU Computing Gems*, W. mei W. Hwu, Ed. Morgan Kaufmann, Sep. 2010, vol. 2.

[15] L. Eyraud-Dubois and T. Lambert, "Using static allocation algorithms for matrix matrix multiplication on multicores and GPUs," in *ICPP 2018 - 47th International Conference on Parallel Processing*, Eugene, OR, United States, Aug. 2018.

[16] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Conc. Comp.: Pract. Exp., SI:EuroPar'09*, vol. 23, 2011.

[17] L. Stanisic, S. Thibault, A. Legrand, B. Videau, and J.-F. Méhaut, "Faithful performance prediction of a dynamic task-based runtime system for heterogeneous multi-core architectures," *Concurrency and Computation: Practice and Experience*, p. 16, May 2015.

[18] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, "Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems," *ACM Tr. Math. Softw.*, vol. 43, no. 2, 2016.

[19] C. Augonnet, O. Aumage, N. Furmento, R. Namyst, and S. Thibault, "StarPU-MPI: Task programming over clusters of machines enhanced with accelerators," in *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 298–299.

[20] O. Aumage, E. Brunet, N. Furmento, and R. Namyst, "New madeleine: A fast communication scheduling engine for high performance networks," in *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, pp. 1–8.

[21] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2004, pp. 97–104.

[22] 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, Jun. 2014.

[23] V. G. Pinto, L. M. Schnorr, L. Stanisic, A. Legrand, S. Thibault, and V. Danjean, "A visual performance analysis framework for task based parallel applications running on hybrid clusters," *Concurrency and Computation: Practice and Experience*, 2018.

[24] L. Nesi, S. Thibault, L. Stanisic, and L. Schnorr, "Visual performance analysis of memory behavior in a task-based runtime on hybrid platforms," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2019.

[25] Y. Pei, G. Bosilca, I. Yamazaki, A. Ida, and J. Dongarra, "Evaluation of programming models to address load imbalance on distributed multicore CPUs: A case study with block low-rank factorization," in *2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*, 2019, pp. 25–36.

[26] A. Denis, E. Jeannot, P. Swartvagher, and S. Thibault, "Using dynamic broadcasts to improve task-based runtime performances," in *Euro-Par 2020: 26th International European Conference on Parallel and Distributed Computing*, Aug. 2020.