Asynchronous multi-phase task-based applications: Employing different nodes to design better distributions

Lucas Leandro Nesi, Arnaud Legrand, Lucas Mello Schnorr

Asynchronous multi-phase task-based applications: Employing different nodes to design better distributions

Lucas Leandro Nesi[a,b,*], Arnaud Legrand[b], Lucas Mello Schnorr[a]

[a] *Institute of Informatics PPGC/UFRGS, Porto Alegre, Brazil*
[b] *Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, Grenoble, France*

Abstract

HPC infrastructures often present intra-node (multi-core CPUs and multiple GPUs) and system-level heterogeneity (different nodes arranged into partitions). HPC applications with several phases, each with distinct resource necessities, can exploit the task-based programming paradigm to overlap phases and take advantage of inter-node heterogeneity to improve performance, provided their workload is correctly distributed. We study two applications with these characteristics, a machine learning framework for geostatistics data, ExaGeoStat, and a multivariate data analysis library, Diodon. We show how to both (1) organize the application to improve runtime and scheduling decisions that impact the asynchronous phases overlap with performance gains between 31% and 46% in ExaGeoStat and 29% to 40% in Diodon when running on homogeneous nodes; and (2) create a distribution per phase over heterogeneous nodes considering overlap and reducing redistribution overhead, improving performance up to 69% in ExaGeoStat and 73% in Diodon compared to a block-cyclic distribution, thereby taming the diversity in supercomputers.

Keywords: Task-Based, Scheduling, Partitioning, Load Balancing, Heterogeneous, Distribution

1. Introduction

Intra-node heterogeneity is now a standard feature in High-Performance Computing infrastructures and is vastly utilized [1, 2]. This trend is further shown by TOP500 [3] list of supercomputers where most systems have one type of accelerator. Accelerators enable the performance improvement of specific computationally intensive kernels while keeping the main CPU free to work on other computational loads. However, extra heterogeneity is present in these supercomputer systems when the nodes (machines) are different and usually organized into partitions. This system-level heterogeneity may exist by natural infrastructure upgrades over time, be present since the design of the supercomputer to accommodate different applications, or because of budget constraints.

Heterogeneity is present not only in the hardware but also in the application. Different operations organized into application phases require distinct computing capabilities and resources. While IO is generally well handled by CPUs, linear algebra kernels can use the SIMD model of GPUs to improve performance. This double-side heterogeneity (infrastructure and applications) constitutes an opportunity to enhance distributions and performance. To efficiently use all available resources, applications require certain freedom to execute the phases concurrently; however, programming and algorithmic challenges limit achieving such phase overlap. From the programmer's perspective, phase overlapping and asynchronous executions are usually hard to implement in traditional bulk synchronous parallel applications relying on MPI or MPI+X. Even after the phase overlapping gets implemented, finding an efficient data distribution for the available nodes is challenging. Non-cyclic distributions are also hard to implement in such traditional paradigms.

All these challenges are why many applications lose this opportunity to use system-level heterogeneity in their favor. The necessities of each phase can be better adjusted using the different machines' computational power. However, developing an application to deal with such complexity requires a programming paradigm that provides the necessary flexibility and features. That is the reason this work uses the task-based paradigm with a dynamic runtime. This paradigm represents an application using a Direct Acyclic Graph (DAG), where nodes are tasks and edges are data dependencies. The runtime schedules those tasks over intra-node resources while trying to minimize the total makespan. Although the runtime can perform many optimizations and decisions, the application should be well-defined, with hints and additional information passed to the runtime. Examples of such runtimes are OmpSs [4], ParSEC [5], and StarPU [6]. This last one allows arbitrary data distribution, asynchronous redistribution, and per-node support of different hardware. Hence, it provides excellent flexibility in handling heterogeneous nodes and distributions.

We illustrate how both challenges mentioned above can be addressed on the multi-phase task-based applications ExaGeoStat [7] and Diodon [8, 9]. Both applications use the task-based dense linear algebra library Chameleon [10] and the StarPU runtime. ExaGeoStat is a machine learning framework for computational geostatistics that relies on the Gaussian process frame-

*Corresponding author
Email addresses: lucas.nesi@inf.ufrgs.br (Lucas Leandro Nesi), arnaud.legrand@imag.fr (Arnaud Legrand), schnorr@inf.ufrgs.br (Lucas Mello Schnorr)

work and optimizes the likelihood, enabling the prediction of missing points. This iterative optimization comprises phases, including generating a positive triangular matrix, a Cholesky decomposition, and a triangular solve operation. Diodon is a library for multivariate data analysis of large datasets that can rely on the randomized singular value decomposition (rSVD) method. One of the operations offered by Diodon is multidimensional scaling (MDS), which uses pairwise distances between elements to map them into an abstract Cartesian space. Diodon MDS includes costly operations: data reading, matrix multiplication, QR factorization, and SVD. Both applications' phases have different computational needs and can overlap. The runtime can achieve this overlap if the DAG has asynchronous tasks with the correct information and data distribution.

The main contributions of this paper are strategies for distributing these multi-phase applications over system-level heterogeneous resources. More precisely, (a) improving the task-based asynchronous execution of both applications' phases. In both cases, the same ideas of improving specific DAG structures like priorities, hints, and dependencies lead to better phase overlap and performance gains. (b) A methodology for generating a heterogeneous system-level static distribution that considers the phases' cost and overlap. (c) A distribution algorithm for a particular phase that uses the following phase distribution to maintain the regional balance and reduce redistribution overhead. (d) An extensive and detailed performance evaluation of the contributions in different scenarios.

This paper is an extension of our previous conference paper [11] and is structured as follows. Section 2 presents the background of the task-based programming paradigm and both applications. Section 3 discusses the related work of heterogeneous distributions and redistribution. Section 4 presents the strategies for multi-phase task-based applications' distributions in system-level heterogeneous resources. Section 5 presents the evaluation of these strategies. Section 6 discusses how to apply the proposed methods to other applications. Lastly, Section 7 offers a discussion with a conclusion and future directions.

2. Task-based Applications

Unlike the imperative declaration of where and when to execute in the MPI+X programming paradigm, in the task-based programming one, the developer uses a more declarative approach to formalize computation operations as tasks and express the execution flow using data dependencies among them. The resulting structure is a Directed Acyclic Graph (DAG), where nodes are tasks and edges are data dependencies. A runtime dynamically schedules these tasks over the computing resources while considering possible affinities and managing data movement. The runtime chosen for this work is StarPU, as it powers both used applications, allows arbitrary data distribution and redistribution, and has great performance analysis tools. Although this choice of runtime, there is no known limitation in the proposed methods that would restrict it. Other runtimes with the ability of arbitrary data distributions, asynchronous redistribution, and association of different nodes could use the strategies presented in this work. We consider the evaluation of other runtimes as future work and out-of-scope.

In the case of Starpu, it handles heterogeneous (CPU/GPU) machines and is extendable for multiple nodes using MPI. It schedules tasks dynamically intra-node following a chosen policy [12]. The distribution over multiple nodes requires defining data ownership, and the runtime will place a task on the node that owns the data block it writes to. One feature of StarPU is changing data ownership dynamically (as a DAG task), allowing asynchronous redistribution alongside other operations.

2.1. ExaGeoStat

The Exascale GeoStatistics (ExaGeoStat) is a machine learning task-based application to model GeoStatistics data, enabling the prediction of missing observations. It uses the Chameleon dense linear algebra solver and StarPU. The core operation of ExaGeoStat is the Gaussian process that interpolates spatial data (X, Z), where X are measurement locations (e.g., 2D coordinates) and Z are the measurements, whose smoothness and scale are controlled by a set of parameters θ that requires to be optimized to the data. Therefore the application iteratively optimizes the log-likelihood of θ through Equation 1.

$$l(\theta) = -\frac{n}{2}\log(2\pi) - \frac{1}{2}\log det(\Sigma_\theta) - \frac{1}{2}\mathrm{Z}^T\Sigma_\theta^{-1}\mathrm{Z}, \qquad (1)$$

where Σ_θ is the $n \times n$ covariance matrix constructed from X, built using a covariance function K_θ (i.e., $\Sigma_\theta[i, j] = K_\theta(X_i, X_j)$), holding the measurement similarity between locations. Machine Learning commonly uses the squared exponential (Gaussian) covariance function; however, the Matérn covariance function is more suitable for geostatistics data, which can be rather rough.

Therefore, this optimization requires the computation of the determinant and the inversion of a large dense symmetric and positive definite matrix Σ_θ at each optimization iteration. To this end, this matrix is first decomposed using Cholesky factorization ($\Sigma_\theta = F^T F$) and used through a triangular solve ($F^{-1}Z$) and a dot product to compute the last term of Equation 1. The diagonal blocks of the factorization give the determinant of the matrix. Each optimization iteration has five phases as depicted in Figure 1: (1) Covariance matrix generation by Matérn function with complexity $O(n^2)$; (2) Cholesky decomposition with complexity $O(n^3)$ using the Chameleon library; (3) Matrix determinant with complexity $O(n)$; (4) Triangular solve with complexity $O(n^2)$ also using the Chameleon library; and (5) the dot product of the solve vector with complexity $O(n)$. Figure 1 depicts the DAG corresponding to one iteration. Also, ExaGeoStat authors reported excellent performance and scalability results with homogeneous multi-core systems [7].

Usually, applications focus on their most computationally intensive phase, in this case, the Cholesky factorization, setting up distributions and scheduling priorities based solely on it. However, each phase has a different computational necessity and suitability with accelerators. ExaGeoStat distributes the data for all operations using the traditional block-cyclic distribution for homogeneous nodes of ScaLAPACK [13], the default option in the Chameleon library. Where the matrix $n \times n$ is divided into $nb \times nb$ blocks and cyclically attributed to the nodes.

Although the principal operation of the Cholesky factorization, the general matrix multiplication kernel `gemm`, performs well in GPUs, the generation's main task, the Matérn covariance function, is only available through a costly CPU implementation at the moment. Therefore, the generation phase makespan can be longer than the Cholesky factorization in small and medium cases [14], even with one complexity order of difference. ExaGeoStat official public repository[1] version presents two options for execution: (1) Synchronous, with a synchronization point between each phase, and (2) Asynchronous, where the synchronizations barriers between factorization and determinant, and solve with the dot product disappear. Yet, this second option is not a fully asynchronous version.

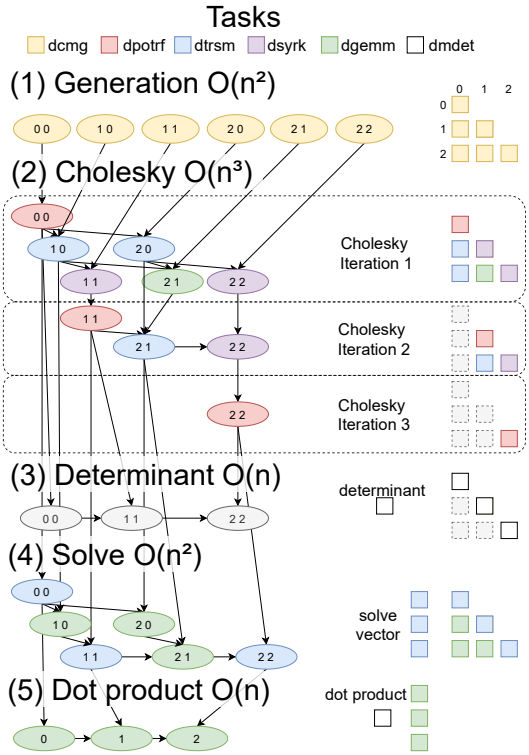

Figure 1: ExaGeoStat iteration DAG for $nb = 3$.

2.2. Diodon

Diodon is a C++ library to compute multivariate data analysis of large datasets. It provides the methods of Principal Component Analysis (PCA), Multidimensional Scaling (MDS), and Correspondence Analysis (CoA). These methods rely on the Singular Value Decomposition (SVD), which unfortunately does not scale well with matrix dimension and is quite hard to distribute and parallelize, but it can be accelerated using the Randomized Singular Value Decomposition (rSVD) approximation. This work focuses on the Multidimensional Scaling (MDS) operation that takes as an input the matrix of dissimilarities D between n elements and finds n vectors $x_i \in \mathbb{R}^m$ such that $\forall i, j \in 1..n\ D_{i,j} \approx \left\|x_i - x_y\right\|$. With m being much smaller than the dimension of the original elements (e.g., images or sequences of DNA fragments), this abstract mapping allows for more straightforward data visualization.

The approximation of the MDS operation using the rSVD is presented in Figure 2, depicting the Diodon DAG and respective phases. The matrix D is structured into $nb \times nb$ blocks of size $bs \times bs$. The first phase is the read of D through `hdf5` files. This step may assume a different distribution because the `hdf5` files may have another grid than the one used by the linear algebra operations. If the `hdf5` file has n columns, each read task will read a complete set of rows (each set with bs rows). Otherwise, there will be a task for each group of rows on each file, totaling rb read blocks per bs rows. In the next step, `copypaneltotile` tasks will map this grid system to the final one, generating $nb \times nb$ blocks into the matrix D. Diodon will then apply the Gram operation, which requires computing the sum of squares of each block, aggregating them for each row (and columns for other operations other than MDS), and finally, for the whole matrix. Then with these values, it is possible to compute the Gramian of each block. While this happens, Diodon can generate the random Gaussian Ω (of dimension $n \times r$) matrix used for the rSVD. Finally, a matrix multiplication operation is performed with D and Ω, followed by a QR factorization and a synchronization. Subsequently, it conducts another matrix multiplication and QR factorization, and Diodon can apply the SVD in this matrix (which is much smaller than the original D). Since the SVD is not yet available in the task-based paradigm, it makes a call to the traditional BLAS library. After the SVD, post-processing operations extract and adjust the eigenvalues and eigenvectors.

The read phase of Diodon uses the `hdf5` library [15]; hence, only CPU workers perform the read tasks. The original version of Diodon performs IO with only one read task at a time, as the library `hdf5` is not thread-safe. We enabled multiple read tasks simultaneously by instantiating `hdf5` with `dlopen` several times, so each worker has a unique and isolated library copy. The ideal number of parallel reads may vary per node and storage resource, as multiple simultaneous tasks will create contention. In our experiments, computational nodes with SSD reach a plateau in performance improvements after five or six concurrent reads, which means that the contention generated when adding a new worker slightly degraded performance. In the rest of the paper, the experiments use five read workers for all SSD and distributed file-system machines.

A remark on Diodon's DAG structure is that there is an algorithmic synchronization point when the sum of squares is computed for the whole matrix (task SO in Figure 2). All the Gram operation tasks and following phases (except for the low-cost Ω generation) must wait for the SO task. Another problem is that this task requires a complete read of matrix A. This synchronization limits resource usage because all tasks before SO do not use GPUs, effectively making all GPUs idle until the entire and long matrix read. Another remark is that in the original version of Diodon, the QR operation was synchronous because of the temporary memory workspace allocation for this operation. Anyway, the synchronization point in the previous Gram operation already limits the overlap, so any optimization

[1] `https://github.com/ecrc/exageostat`

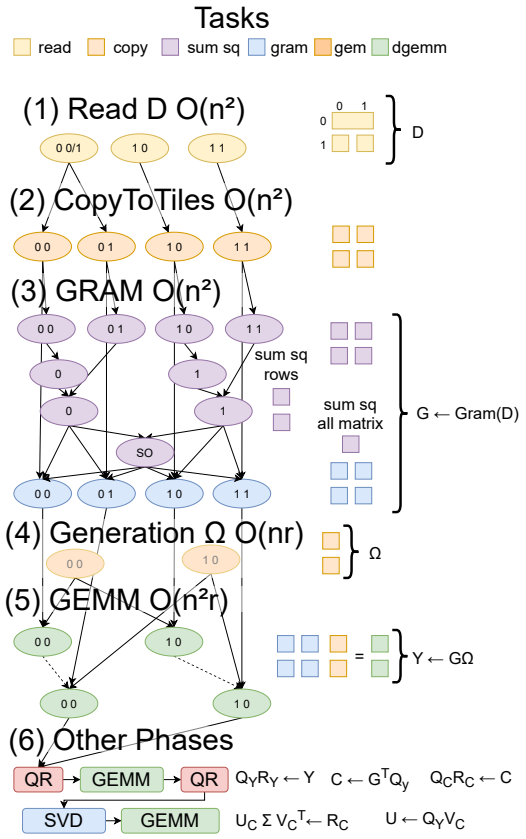

Figure 2: Diodon MDS DAG for $nb = 2$.

in the QR synchronization would have limited improvement. This work focuses on the overlap and the performance of the first five phases plus the first QR factorization when it ends with a synchronization barrier.

Moreover, the BLAS libraries perform the SVD operation directly and are not handled by the Task-Based Chameleon, requiring an entire synchronization point. Finally, although MDS uses a symmetric matrix, we study the functions that use the complete matrix computation to consider some behavior of the other methods, including PCA and CoA. Diodon authors focused on achieving performance with homogeneous nodes with large matrices in double and single precision. We aim to further improve performance by exploiting system-level heterogeneity.

2.3. The opportunity for using Heterogeneous Nodes

The distinct phases raise severe load-balancing issues when considering different hybrid nodes (CPU+GPU). Indeed, some operations better exploit GPUs while others better utilize CPUs. While in ExaGeoStat, the Matérn covariance function has a different need from the Cholesky Factorization, in Diodon, the IO `hdf5` tasks also have another type of demand than the Matrix Multiplication. Ideally, when balancing the load across phases, each would have its distribution and not necessarily use the traditional block-cyclic one. Different distributions require redistribution along the asynchronous overlap of the phases. Nevertheless, the diverse tasks' demands can exploit the nodes' heterogeneity while adequating resources and load better.

3. Related Work

The `col-peri-sum` algorithm for data distribution. Most linear algebra factorization algorithms update a matrix region that diminishes along the iterations of the factorization. In homogeneous platforms, comprising nodes with identical hardware configurations, a 2D block-cyclic distribution [13] is the more widely used strategy to obtain load balance while minimizing communications among nodes. When system-level heterogeneity is present, with nodes with different hardware configurations, distributions must respect the computational capabilities of each node [16]. An alternative 2D block-cyclic distribution is achievable by (1) doing a matrix partitioning considering predefined areas according to nodes' computational capacities while minimizing communication (a classical algorithm is the `col-peri-sum` algorithm [17]), as depicted in the left of Figure 3, and (2) shuffling rows and columns to obtain a smoother iteration progressing by employing, for instance, the 1D-1D algorithm [18], as shown in the right of Figure 3. All matrix distributions obtained with these two steps are asymptotically optimal. Chameleon can use this distribution [19], extending it to other applications that rely on the library, including ExaGeoStat and Diodon. As a consequence, it provides a solid basis for this work.

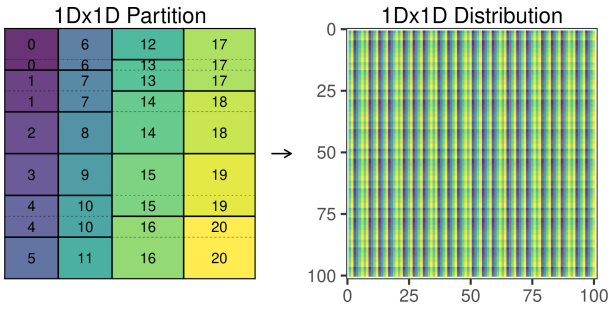

Figure 3: Left: the 1D-1D column-based partition; Right: after shuffling.

Methods for data redistribution. Carrying out data redistribution has been previously tackled since Prylli et al. [20] proposed the first algorithms. The common objective is to keep the cost of scheduling redistribution minimal when moving from one block-cyclic distribution to another. The changes can consider different block sizes, each more adequate for distinct application phases. Typically, data redistribution occurs synchronously and inefficiently between the phases of an MPI application. Using synchronous communication has become obsolete for modern large-scale supercomputers. As an alternative, the state-of-the-art comprises data flow algorithms with fewer synchronizations [1]. Linear algebra solvers have embraced asynchronous features provided by modern runtimes to overlap tasks and iterations and increase parallelism [10, 21]. Targeting applications containing multiple phases is a challenge that involves finding the best distribution for each operation, simultaneously minimizing the redistribution communication cost. Herrmann et al [22] indicates that the redistribution may consist of many permutations, for example, with nodes with similar computational capacity. In such scenarios, they argue that find-

ing the ideal distribution from the many possible permutations while reducing the computation and communication cost is NP-hard. Finally, they present performance models for the redistribution in a synchronous execution, using modern runtimes that can overlap communication and computation.

Comparison with previous publication. This paper is an extension of a previous conference paper [11]. The main contributions revolve around expanding and generalizing the methods for other applications. The overlap optimizations in Section 4.2 adds three categories that have the potential to limit overlap and impair performance. The strategies for distributing asynchronous multi-phase applications on heterogeneous resources are generalized for applications with two major cluster phases (when the phases of interest fall into two categories). Another significant addition is the new equations to extract the correct powers from the linear program, presented at the end of Section 4.3. Section 4.4 now contains a new version of distribution for the computational intensive phases, based on the idea of constraining the end of the distribution on some resources [19]. The Diodon application has been an addition in illustrating all these points. Finally, we significantly expanded the experimental Section with at least 18 scenarios per application when evaluating the distribution techniques, including new workloads sizes, machines' scenarios, distribution versions, and the SDumont Supercomputer. The expanded experiments' diversity with two applications strongly demonstrates the paper's contributions.

4. Multi-phase partitioning in heterogeneous partitions

This Section presents the strategies to correctly partition the load of multi-phase applications into system-level heterogeneous resources. Before aiming for heterogeneous resources, we had to improve the asynchronous execution of the phases. Section 4.1 details ExaGeoStat and Diodon behavior, highlighting problems in execution. Section 4.2 presents a categorization of phase overlap optimizations that can be applied to task-based applications, following a set of detailed improvements made to both applications improving phases overlap. With such optimizations, Section 4.3 presents the strategies to compute the relative node power per phase, as it is the input for typical distribution algorithms, considering all major operations, their overlap, and heterogeneous partitions. Finally, Section 4.4 shows how to compute a phase distribution while minimizing communication, using the following distribution as a foundation.

4.1. Characterizing the Phases

The first step to understanding the opportunities for improvement is to characterize the application's behavior. To this end, we use visualizations from the performance analysis tool StarVZ [23, 24]. Figure 4 presents three panels for one iteration of the synchronous version of ExaGeoStat. In all panels, the X-axis is the time in milliseconds. The middle panel is a Gantt Chart showing the aggregated utilization per resource type per node (for example, CPU 0 is the aggregated utilization in % of all CPUs in node 0). The makespan is the number position on the right end. The upper panel shows the computational intensity of each iteration of the Cholesky factorization on the Y-axis, with generation tasks mapped to 0 and post-Factorization mapped to the last subsequent iteration. Two supplementary lines show each iteration's beginning (left) and end (right). This plot depicts how the factorization unfolds and gives a signature to the execution. The last panel shows the memory utilization per memory node: each node's RAM and each GPU memory.

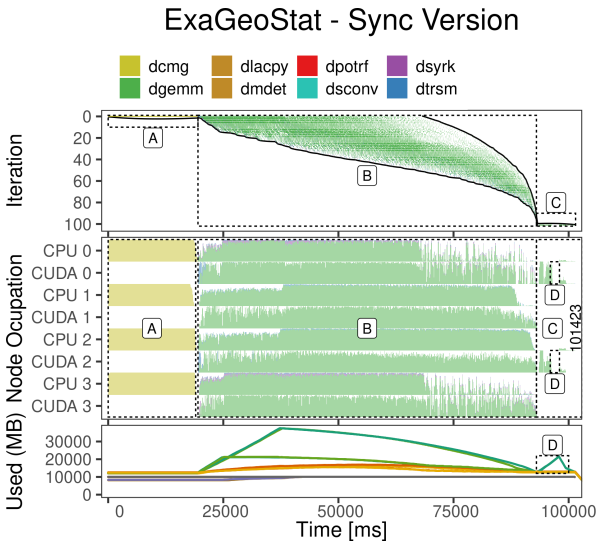

Figure 4: Iteration, Node occupation, and Memory panels for the synchronous version of the ExaGeoStat iteration.

In the Figure, there is a distinction between the main three phases of ExaGeoStat. In the beginning, the yellow area (A) reflects the `dcmg` tasks of the *generation* phase that only run on CPUs because the Matérn covariance function relies on the modified Bessel function whose implementation is complex and which is not available for GPUs. The (B) mostly green area represents the *Cholesky factorization* with its predominant `dgemm` tasks. Finally, the (C) area is the *post-factorization* operations (determinant, solve, and dot product, whose dominant operations are the `dgemm` tasks (in green) from the solve. Considering the whole execution, the resource usage can be improved, especially at the beginning (where only the CPU cores work because of the generation constraint) and toward the end (where there is not enough work for all nodes). Yet, the DAG should enable further parallelism that could shift a lot of green tasks to the beginning of the execution and improve the execution time.

Diodon's behavior shares similarities with ExaGeoStat. Figure 5 presents the StarVZ visualization (similar to the ExaGeoStat one) of this application behavior using the same four homogeneous nodes. The figure also clearly shows all Diodon's phases: the read phase with the yellow IO tasks and the orange `copypaneltotile` ones (A annotation); the `sum of squares` (purple) and the `gram` tasks (blue) (B); the first and second pair of matrix multiplication (`dgemm`) in green and QR factorization tasks in red (C and D); the external untraced SVD call (E); and the post-processing operations (F). The resource utilization is small in the read phase, as the number of reading workers is limited for IO reasons. Because there is a synchronization

point between all phases, GPUs remain completely idle until the matrix multiplication phase starts very late in the application. Finally, the SVD and post-processing operations for this particular workload size are minimal.

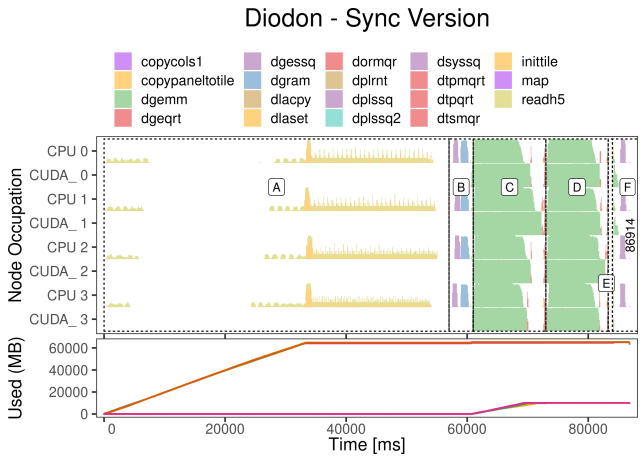

Figure 5: Node occupation and Memory panels for the synchronous version of the Diodon complete execution.

In both applications, the lack of overlap between phases makes the application miss an enormous performance opportunity: powerful resources (GPU) could work more. The following subsection highlights problems and proposes strategies to improve phase overlapping.

4.2. Improving Application's Phases Overlap

Applications with synchronous phases lose an opportunity to adapt their workload and increase resource usage. In ExaGeoStat, the factorization could start as soon as the matrix upper part is generated and exploit the GPUs while the generation proceeds. A similar situation happens in Diodon since the Gram and matrix multiplication phases can proceed with only a portion of the data. This way, our first optimization removes the synchronization points between all internal operations and makes a **fully asynchronous** execution. This optimization allows StarPU to control the tasks' order and flow. However, a correct phase overlap requires extra information passed to the runtime (like priorities) and the leverage of non-computational operations, including communication and allocation costs.

Although these hints and collateral effects may vary depending on the algorithm and application structure, they share the same goals and principles. We globally organize these optimizations into the following categories. **[I] Algorithm asynchronism optimizations** to improve the number of parallel operations and remove possible synchronization points in the program. **[II] Memory optimizations** to improve memory allocation and utilization between phases, and when is the best time to do those operations. **[III] Communication optimizations** to reduce data transfers between phases and enable the earlier start and correct overlap. **[IV] Scheduling hints and artifacts optimizations** to tweak and give information to the scheduler (like task priorities considering the whole application) to maximize phases' overlap and induce the earliest execution of the critical path. The rest of the Section presents each one of the optimizations on each application, indicating their categories.

4.2.1. ExaGeoStat

Local solve algorithm [I, III]: The behavior depicted in Figure 4 (D annotation) shows an increased memory consumption with associated idle times. Further investigation revealed that a collateral effect of the solve phase caused it. The original Chameleon's solve algorithm performs its `dgemm` operation on the node that owns the solution vector (the right-hand vector) that follows the distribution of the main matrix. Consequently, many matrix blocks are moved between nodes to complete a simple `dgemv` operation, demanding extra allocation and communication between nodes. The number of allocations and communication, especially in a non-optimal order, will cause idle time, which can be considerable depending on the machines' configuration and matrix size. The solution was to replace the Chameleon solve algorithm with a version (Algorithm 1) that improves locality by performing all possible `dgemv` operations locally (that the node already has the data). This is done by accumulating its outputs in a local node vector **G** and communicating only it to perform a reduction on **Z**.

Algorithm 1: Local solve algorithm.

for $k = 0$ **up to** nb **do**
  dtrsv($M_{(k,k)}$, Z_k)
  for $i = k + 1$ **up to** nb **do**
    dgemv($M_{(i,k)}$, Z_k, $G_{(i,\text{node of } M_{i,k})}$)
  end
  foreach *updated* $G_{(k+1,j)}$ **do**
    dgeadd($G_{(k+1,j)}$, Z_{k+1})
  end
end

Memory optimizations [II]: In an asynchronous environment, the maximum memory consumption will not be the individual phases but a mixture of all. This mixture can cause problems if forgotten. We perform four optimizations to mitigate memory management issues by changing runtime functionality, application DAG, and tasks. The first optimization is to transfer the RAM allocation from the task submission function and perform it as other allocations, as a management task in the DAG. The second optimization is to enable cache for the StarPU's chunk memory system when using RAM, causing chunk reuse between phases and iterations. The third optimization is related to a slow allocation of the CUDA API, which GPU workers can perform. To maximize GPU throughput, we disable such operations on GPU workers. The last optimization is pre-allocating some memory chunks during application initialization for the first iteration use it as a cache.

New priority equations [IV]: One of the possible hints the application can pass to the runtime is priorities, which will play a role in the scheduler decisions. The original Chameleon implementation defined priorities for the Cholesky Factorization tasks from $2nb$ to $-nb$ with an order following roughly the anti-diagonal. Other operations did not specify priorities, which

StarPU interprets as 0. That means that for non-defined priorities, procedures like the generation or solve had effectively priority 0, all preferably executing in the middle of the factorization. For this reason, we propose new priorities for all tasks considering all phases to guarantee a smoother transition between them all. Such priorities follow the natural DAG critical path with a unit execution cost (i.e., starting from the last tasks and going backward to the first generation's tasks). The new priorities equations for each task are in Equation 2, which considers a task that writes on a block of coordinate (i, j) on iteration k. Because the Cholesky DAG is essentially the foundation, the generation priorities align with the first factorization iteration ($k = 0$), and to force an earlier execution, the negative component is divided by 2. The other operations (determinant and dot product) receive a 0 priority as they are DAG leaves and can be interchanged without any consequence.

$$
\begin{aligned}
\text{[Generation] } \texttt{dcmg} &= 3nb - \tfrac{i+j}{2} \\
\text{[Cholesky] } \texttt{dpotrf} &= 3\,(nb-k) \\
\text{[Cholesky] } \texttt{dtrsm} &= 3\,(nb-k) - (i-k) \\
\text{[Cholesky] } \texttt{dsyrk} &= 3\,(nb-k) - 2\,(j-k) \\
\text{[Cholesky] } \texttt{dgemm} &= 3\,(nb-k) - (j-k) - (i-k) \\
\text{[Solve] } \texttt{dtrsm} &= 2\,(nb-k) \\
\text{[Solve] } \texttt{dgemm} &= 2\,(nb-k) - i \\
\text{[Solve] } \texttt{dgeadd} &= 2\,(nb-k) \\
\text{[Determinant] } \texttt{dmdet} &= 0 \\
\text{[Dot] } \texttt{dgemm} &= 0
\end{aligned}
\tag{2}
$$

Submission order [IV]: Although the scheduler will try to follow the priorities order, in practice, an artifact can occur and lead to a low-priority task executing before a higher one. Because schedulers can not foresee the future, a low-priority task will immediately start if there are idle resources. In this way, if the following task submitted has a higher priority, it will have to wait for the previous one to finish. This problem can be easily overcome by submitting tasks following their priorities. Because this artifact mainly occurs in the generation phase, only this one had the order modified.

Oversubscribe [IV]: Another possible runtime artifact is the delay of high-priority tasks because of longer ones. If all workers have longer tasks, and a high-priority one becomes ready, it may have to wait for an available resource (without preemption procedures). This situation may be a problem if the task is part of the critical path and is responsible for releasing a lot of other tasks. This is the case of `dpotrf`, which can only execute on CPUs and may have to wait because generation tasks (`dcmg`) are much longer. Also, the `dpotrf` tasks are the ones that release many `dgemm` tasks in the Cholesky and can use powerful resources like GPUs. We use the idea of oversubscribing a CPU core with a dedicated worker for critical tasks and the core StarPU dedicated to task submission. Although while tasks are being submitted, the computation of they will suffer from contention, the collateral impact is small relative to the possible gains of advancing the critical path faster.

4.2.2. Diodon

Memory optimizations [II]: Data allocation management was also a problem at Diodon. Similar optimizations conducted in ExaGeoStat also apply here, including not making all allocations at submission time and disabling slow allocations on GPU workers and critical workers (the workers that perform the read). Another optimization was to pre-allocate the blocks for reading panels. In the default behavior, StarPU would make all allocations in nodes with CUDA GPUs employing the CUDA API to pin the memory to accelerate transfer. However, these allocations are slow. The read panels will never be transferred to GPU directly, as Diodon will first break those panels into blocks and redistribute them for the computational-intensive phase. This redistribution into other memory regions means that those panels are not required to be pinned.

Asynchronous Gram and Matrix multiplication workflow [I, III]: One of the original operations performed in the Diodon MDS is the Gram one. Diodon makes this operation by first computing the sum of squares to each row and the whole matrix and then applying the `gram` centralizing kernel. However, this situation implies a global synchronization point during the Gram operation as all `gram` tasks require the whole matrix sum of squares. Because the order of operations starts with the read followed by the Gram, and both do not use GPUs, these powerful resources would continue to idle until Diodon computes the whole matrix sum of squares, which requires reading the entire matrix. This synchronization implies a waste of GPU power during matrix reading. To overcome this problem, we propose a new Gram and Matrix multiplication workflow to improve parallelism and start the subsequent phases that can use GPU during matrix reading. We use associativity properties to break and modify the Gram and first matrix multiplication operations. The original Gram operation computes the Gramian block G_{ij} of the original matrix block D_{ij} using c (a constant coefficient), m (number of rows), n (number of columns), SR, SC, and SO, which are the sum of squares of the row, column, and the overall matrix respectively. Then, with a random generation matrix for the rSVD Ω, it performs the matrix multiplication $Y = G\Omega$. The following equations describe this operation in an expanded form for each block of the final Y matrix.

$$
G_{ij} = c(D_{ij}^2 - \frac{SR_i^2}{n} - \frac{SC_j^2}{m} + \frac{SO^2}{nm}) \tag{3}
$$

$$
\begin{aligned}
Y_{ij} &= \sum_k (G_{ik}\Omega_{kj}) \\
&= \sum_k (c(D_{ik}^2 - \frac{SR_i^2}{n} - \frac{SC_k^2}{m} + \frac{SO^2}{nm})\Omega_{kj}) \\
&= \sum_k (c(D_{ik}^2)\Omega_{kj}) + \\
&\quad c(-\frac{SR_i^2}{n} + \frac{SO^2}{nm}) \sum_k (\Omega_{kj}) - c(\sum_k (\frac{SC_k^2}{m}\Omega_{kj}))
\end{aligned}
\tag{4}
$$

The goal here is to perform the costlier ($O(n^2 r)$) operation $Y = G\Omega$ earlier. In this way, we reorganize the equations so

that the first task only computes the matrix powering square G', followed by multiplying such matrix to omega $Y' = G'\Omega$. The difference of Y for each column j is $\frac{SR_i^2}{n}$ and $\frac{SO^2}{nm}$ multiplied by the j column sum of the matrix Ω and $\sum_k(\frac{SC_k^2}{m}\Omega_{kj})$. This way, after submitting the generation tasks for the matrix Ω, we submit tasks that will compute the sum of each column of Ω and store it in A_j. Because the generation of Ω and these summations are independent of D, such tasks can run during D's read while the system load is relatively low. We also submit tasks to compute the $\sum_k(\frac{SC_k^2}{m}\Omega_{kj})$ component for each column j and store it in B_j. Those tasks will gradually execute as SC becomes available. Finally, we submit two new tasks: `gram_gemm_post` to correct the value of Y' (and become the original Y) by using A and B, and `gram_post` that will fix the value of G' to G.

$$G'_{ij} = c(D_{ij}^2) \tag{5}$$

$$Y'_{ij} = \sum_k (G'_{ik}\Omega_{kj}) \tag{6}$$

$$= \sum_k (c(D_{ik}^2)\Omega_{kj})$$

$$A_j = \sum_k \Omega_{kj} \tag{7}$$

$$B_j = \sum_k \frac{SC_k^2}{m}\Omega_{kj} \tag{8}$$

$$Y_{ij} = Y'_{ij} + c(-\frac{SR_i^2}{n} + \frac{SO^2}{nm})A_j - cB_j \tag{9}$$

$$G_{ij} = G'_{ij} + c(-\frac{SR_i^2}{n} - \frac{SC_j^2}{m} + \frac{SO^2}{nm}) \tag{10}$$

Commutable tasks [I, IV]: Another interesting scheduler artifact is task dependencies that establish a strict task sequence that, in reality, can occur in any order. One example of such behavior is matrix multiplication. A final result block of coordinate (i, j) will be computed by the sum of the $M(i, k)N(k, j)$ blocks. This sum is commutable and can happen in any order. However, during submission time, the default behavior of StarPU is to create a dependency between a later submitted task that uses a block with `RW` (read and write) permission to any previous task that has used that block. When Chameleon submits the `dgemm` tasks from $k = 1$ to nb, the `dgemm` task of $k = 2$ will depend on the $k = 1$ task, when in reality, they can commute. The commutability notion is known and possible to StarPU; however, it requires that such tasks be submitted with the permission `RWC`. Such optimization a priori may appear small. However, in our case, without the optimization, the `dgemm` tasks of local M and Ns (already in the local memory of the node) with a larger k would need to wait for the communication of other M and Ns of lower k, reducing possible ready tasks and then resource usage. Thus, such optimization removes this unnecessary dependency path, increasing the number of possible ready tasks.

Read phase distribution [III]: The original Diodon reads the matrix panels in a sequential node order, which means that for x nodes, the first one will receive a continuous $\frac{n}{x}$ area to read. Although this may benefit IO read purposes on specific disks, it can generate too many data transfers when reorganizing the blocks' distributions for the computationally intensive phases. At this point, this optimization only matches the read phase distribution to the computation phase. However, in the next Section, when discussing heterogeneous nodes and phases, this optimization is replaced with our distribution calculated from the computation-intensive phases, which considers an ideal read distribution while reducing communications to the next phase.

New priority equations [IV]: In the same situation as ExaGeoStat, we have to improve the phases overlap by hinting to the scheduler the tasks that should execute first with priorities. This is important as the submission order differs from the ideal execution order. Because there will be an inevitable synchronization point before the `gram_post` operation, the priorities for this optimization are only defined for the following tasks: `read`, `copypaneltotile`, `gram_pre`, `dgemm`, and tasks related to the sum of squares. Because the generation reads the whole line of each `hdf5` file and the matrix multiplication operation requires a complete line of D to calculate each final block, we follow the order of the lines in all phases, i.e., the block index i of most operations. A high priority task `dlaset` for initializing the block is set to the highest priority n (number of columns). The priority for the matrix multiplication operation follows its intra-loop k to follow the left to right order of the matrix D for the available blocks. Equation 11 shows the new priorities.

$$\begin{aligned}
\text{[Generation] } \texttt{inittile} &= nb - i \\
\text{[Generation] } \texttt{read} &= nb - i \\
\text{[Generation] } \texttt{copypaneltotile} &= nb - i \\
\text{[Computation] } \texttt{dlaset} &= nb \\
\text{[Computation] } \texttt{syssq} &= nb - i \\
\text{[Computation] } \texttt{gessq} &= nb - i \\
\text{[Computation] } \texttt{plssq} &= nb - i \\
\text{[Computation] } \texttt{gram_pre} &= nb - i \\
\text{[Computation] } \texttt{dgemm} &= nb - k
\end{aligned} \tag{11}$$

The correct overlap of the phases in a smooth way is a requirement for further load distribution. These optimizations will help improve performance locally, and when using non-standard distributions, computational-intensive late DAG tasks with a higher priority can run earlier.

4.3. Load Balancing across Application Phases

The distribution of asynchronous multi-phase applications into multiple node partitions must consider the phases' overlap. This situation happens because essential tasks from a further phase could and should start earlier and use resources idle from previous phases. The standard literature procedure to compute a distribution is to use the relative power of the machines involved, as shown in Section 3. In this context, the overlap length should be considered when computing the relative machine powers for a phase. Consider ExaGeoStat, for example;

while the two main phases *generation* and *factorization* are relatively long, the generation cannot use GPUs. If the computation of the relative machines' powers for the factorization only considers itself, the actual distribution for GPU machines would be undersized, as *factorization* tasks can start already during *generation*. This means CPU-only machines would be working all the time, but GPU machines can receive more load than the exact relative machines' power. And as powerful as the GPU is, more load it can receive, alleviating work from CPU-only machines. A similar situation happens in Diodon, as the read phase is IO and performed in the CPUs, while the next matrix multiplication and Gram can occur asynchronously (the computational intensive phases). However, in Diodon, there will be an inevitable synchronization at the end of the `gram_post` tasks, as the whole matrix is necessary to finalize the operation. These Section strategies are then applied between the read phase, the Gram operation, and the first matrix multiplication.

This paper presents a linear program to correctly estimate the nodes groups' powers in task-based applications considering the interaction of two phases. Since phases overlap and depend on each other, the central concept is to divide the *phases* into virtual *steps* and relate the duration of each virtual phase step to dependencies and resource usage. The linear programming model uses the following notations. The task *type* t corresponds to the different tasks in the application (`dgemm`, `dcmg`, `dgram`). A virtual step s represents a set of approximately independent tasks. In ExaGeoStat, the first phase is the generation, and we decide that each generation step will correspond to an anti-diagonal in the matrix (all the blocks of coordinates i and j such as $s = \frac{i+j}{2}$), which corresponds to the priorities in Equations of the previous Section. For the second phase, a factorization task of step s was created by a generation task of the same step s. In Diodon, we consider the matrix's natural read order, and each step is composed of the total number of parallel read tasks (based on the maximum of workers that can perform them, in the case of this paper, is five) and the computationally intensive tasks they release (Gram and matrix multiplication ones). In this way, considering that each row has rb read blocks, and bs rows are grouped into nb blocks, a read block of coordinate i ($1 \leq i \leq nb$) and j ($1 \leq j \leq rb$) is on step s such as $s = \frac{i \times rb + j}{5}$, as five is the number of parallel read workers used in this paper. Because the generation of the matrix Ω is low cost and can be done during the first read step, we do not add it to the model.

The linear programming model takes as an input $N_{s,t}$, the total number of tasks of type t at step s, and their respective processing duration. A resource group r corresponds to, for example, all CPUs of a homogeneous set of nodes. We denote by $w_{t,r}$ the duration that a task of type t takes on resource group r (tasks that cannot execute on a given resource r have their cost set to infinity $w_{t,r} = \infty$). Finally, we will denote possible resources, steps, and task types by $\mathcal{R}$, $\mathcal{S}$, and $\mathcal{T}$. The sets $\mathcal{P}_1$ and $\mathcal{P}_2$ denote the task types of each phase and are disjoint.

The fractional number of tasks $\alpha_{s,t,r}$ of type t of step s placed on resource group r is the main output of the program. Dependencies among phase are accounted by introducing the variables $\tau_s^{(1)}$ and $\tau_s^{(2)}$, which represent the ending times of step s at the first and second phase respectively. The application makespan is given by the last second phase (computationally intensive) ending time $\tau_{nb}^{(2)}$. The following linear program approximates ExaGeoStat and Diodon behavior, where $\alpha_{s,t,r}$, $\tau_s^{(1)}$, and $\tau_s^{(2)}$ are all positive variables:

$$\text{Minimize} \sum_{s \in \mathcal{S}} (\tau_s^{(1)} + \tau_s^{(2)}) \; s.t. : \tag{12}$$

$$\forall t \in \mathcal{T}, \forall s \in \mathcal{S} : \sum_{r \in \mathcal{R}} \alpha_{s,t,r} = N_{s,t} \tag{12a}$$

$$\forall s > 1, \forall r \in \mathcal{R} : \tau_{s-1}^{(1)} + \sum_{t \in \mathcal{P}_1} \alpha_{s,t,r} w_{t,r} \leq \tau_s^{(1)} \tag{12b}$$

$$\forall s \in \mathcal{S}, \forall r \in \mathcal{R} : \tau_s^{(1)} + \sum_{t \in \mathcal{P}_2} \alpha_{s,t,r} w_{t,r} \leq \tau_s^{(2)} \tag{12c}$$

$$\forall s > 1, \forall r \in \mathcal{R} : \tau_{s-1}^{(2)} + \sum_{t \in \mathcal{P}_2} \alpha_{s,t,r} w_{t,r} \leq \tau_s^{(2)} \tag{12d}$$

$$\forall r \in \mathcal{R}, \forall s \in \mathcal{S} : \sum_{z \leq s, t \in \mathcal{T}} \alpha_{z,t,r} w_{t,r} \leq \tau_s^{(2)} \tag{12e}$$

$$\min_{r \in \mathcal{R}, t \in \mathcal{P}_1} (w_{t,r}) \leq \tau_1^{(1)} \tag{12f}$$

While the general goal of these optimizations is to reduce application execution time, the objective function for the linear program in Equation 12 is more complicated. If the LP used a simple loose objective function like $\tau_{nb}^{(2)}$, the ending of the previous phases' steps $\tau_s^{(2)}$ for $s < nb$ could appear as late as possible when the first phase is the bottleneck, which is undesirable. Instead, the objective function minimizes the sum of all $\tau_s^{(1)}$ and $\tau_s^{(2)}$ inducing a simultaneous minimization of all steps endings. In our experiments, giving more weight to $\tau_s^{(2)}$ or adopting a recursive minimization fail to provide any practical improvement compared to this simple sum.

The Equations 12a to 12f refer to the constraints of the linear program, and their individual goals follow. Equation 12a guarantees that the exact number of tasks are used in the resources. The dependencies by consecutive steps inside the same phase are approximated by Equation 12b, which states that one phase will end after the previous one ends, plus its own tasks. The next constraint refers to the dependencies between phases. Equation 12c guarantees that a step from phase two cannot end earlier than the same step from phase one and its own tasks. Equation 12d is similar to Equation 12b and enforces that the end of phase two will be after the end of the previous phase plus its tasks. This rule is stricter in the model than in reality because of the completely asynchronous execution; many iterations of the algorithm (that share multiple steps) can be executed in parallel. However, because they essentially share the same cost, it ensures the correct progression between phases without penalizing it too much. To guarantee that two tasks do not overlap in the resources, Equation 12e states that for each resource r, phase two ends at a step s be at least the sum of all previous tasks on that resource. The last constraint, Equation 12f, improves the approximation in the earlier stages of the execution. As the linear program adopts rational variables, the model allows the "split" of tasks between resources. In ExaGeoStat, the duration of the best implementation of the first

phase tasks will be minimal time for the first step of phase one. In Diodon, because of the step definition and the guarantee that each read step will execute at maximum x parallel tasks, as we have only x parallel workers, this rule can be strict to $\forall s \in \mathcal{S} : \tau_s^{(1)} + \min_{r\in\mathcal{R},t\in\mathcal{P}_1}(w_{t,r}) \leq \tau_{s+1}^{(1)}$.

Although the linear program has many constraints, it is fast solved in all our cases. The final duration result remains an excellent approximation and a lower bound even if some rules do not entirely represent reality (Equation 12d strictness, for example). In addition, the α variable is an excellent indicator of the number of tasks each resource group should execute and can be used to compute the relative power per machine per phase.

The relative node power for each phase is an input for most distribution algorithms. The computation of power p_x of node x uses as input α from the LP, the number of tasks in the problem $\sum_{s\in\mathcal{S}} N_{s,t}$, and a task type weight (h_t) to adjust the most significant tasks. Equations 13 shows how to compute p_x.

$$c_t = \frac{1}{\sum_{r\in\mathcal{R}} \frac{1}{w_{t,r}}} \sum_{s\in\mathcal{S}} N_{s,t} \tag{13a}$$

$$h_t = \frac{c_t}{\sum_{z\in\mathcal{T}} c_z} \tag{13b}$$

$$p_x = \sum_{r\in\mathcal{R}^{(x)},t\in\mathcal{T},s\in\mathcal{S}} h_t \frac{\alpha_{s,t,r}}{N_{s,t}} \tag{13c}$$

The weight h_t uses c_t, the time to complete all tasks of type t using all resources. The component $\frac{1}{\sum_{r\in\mathcal{R}} \frac{1}{w_{t,r}}}$ express the duration needed to complete one task of type t if it could be hypothetically split in all resources. The weight h_t is then the normalization of c_t, and the normalized values are greater for the task type that requires more time to fully complete. Finally, the power is computed by summing the normalized amount of tasks placed ($\frac{\alpha_{s,t,r}}{N_{s,t}}$) on any resource of that node ($\mathcal{R}^{(x)}$) for each task type, multiplied by the weight of that type.

4.4. Multi-Partitioning for distinct phases

Ideally, each phase has a distinct distribution, as each has a different computational need and resource affinities. In ExaGeoStat and Diodon, while the main tasks of the computationally intensive phase are the `dgemm` that GPUs can accelerate, the main tasks in the first phase (data generation) only execute on CPUs. The linear program output can derive the ideal computation load for each phase. Consider a simple situation with ExaGeoStat with four nodes with the same CPU, but two have GPUs. When considering the problem individually per phase, while the generation distribution will be equally divided, the factorization would mainly use the two faster nodes. Figure 6 shows a possible data distribution for the generation in the left (a simple 2D block-cyclic distribution) and for the factorization in the middle (a 1D-1D distribution).

In ExaGeoStat, the 1D-1D distribution obtained by the shuffling procedure mentioned in Section 3 ensures a well-balanced factorization with minimal communication. In Diodon, the problem can be relaxed to a 1D distribution. However, because of the ratio of work in Diodon, assigning the last rows of the

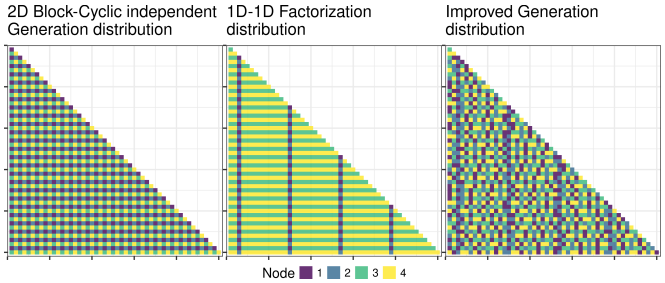

Figure 6: Generation and Factorization distributions for two nodes (1, 2) without and two (3, 4) with GPUs.

matrix to CPU-only machines during the computational intensive phase would create a substantial critical path, as $\frac{r}{bs}$ tasks would need to execute sequentially with the traditional algorithm (other algorithms would require extra memory and a reduction operation). For this reason, we used the idea of a constrained distribution [19] where we only consider assigning a row to a node if the global amount of work divided by the resources (expected application duration time) is larger than the Critical Path Bound (CPB) of assigning that row to that node. Considering the placement of the x row, starting from the last one, the CPB is easily computed as $\frac{r}{bs}$ times the best implementation of the `dgemm` task on that node plus the LP result for the read phase x. At the same time, the amount of work divided by the resources can be estimated by the LP result of the computationally intensive phase at the last step $\tau_{nb}^{(2)}$. We always consider nodes for assignment with the lowest CPB plus a tolerance of two times the number of blocks in milliseconds. This heuristic handles cases with very small workloads and cases where the CPB is marginally different for the nodes. This idea of constraining the end of the distribution can also be applied to ExaGeoStat. We restrict the placement of the last rows of the Cholesky distribution to GPU-only nodes while the CPB of an iteration is greater than the Area-Bound Estimator (ABE) [19].

If the phases' distributions (data generation and computationally intensive) are built independently, most of the assigned nodes on each block will probably be different, requiring more communication during the redistribution. For example, consider ExaGeoStat with a 50×50 matrix and the scenario of Figure 6, using the optimal independent partitions results in the communication of 890 blocks between the generation and the factorization phase, i.e., 70% of the total number of blocks.

However, each node should receive roughly the same number of blocks, as the CPU is the same. When considering the phases overlap and possible interference, the linear program states an ideal of [318, 319, 319, 319] blocks per node in the generation and [60, 60, 565, 590] for the factorization. These differences mean that the first two nodes should transfer 517 blocks to the latter ones. This redistribution with 517 communications would be the minimum possible, i.e., 373 (≈42%) fewer transfers than when distributions are independent. However, not only are the quantities important, but even for the generation, the ideal distribution would maintain it "cyclic", just like the 1D-1D distributions, ensuring that the first generation blocks are spread and processed in parallel over the nodes. This

cyclicality helps increase earlier computational intensive tasks ready, as in the factorization, this order matter.

Redistributing the two distributions in both applications can incur extra communication overhead. To minimize it, we propose Algorithm 2, which receives a computationally intensive distribution (for example, the 1D-1D distribution for ExaGeoStat factorization or a 1D heterogeneous cyclic distribution for Diodon, both with or without the constrained rule) and an ideal number of generation blocks per node. The output is a distribution for the generation that respects the factorization cyclicality, minimizing the redistribution communication cost.

Algorithm 2: Generation of a target (data generation) distribution ($dist_{(2)}$) from a source (computational intensive) distribution ($dist_{(1)}$) with p nodes.

Input: $dist_{(1)}[1...mb][1...nb]$
$db_{(2)}[1...p]$ *Desired number of blocks per node*
Output: $dist_{(2)}[1...mb][1...nb]$
$db_{(1)}[1...p] \leftarrow$ Number of blocks per node in $dist_{(1)}$
$diff \leftarrow db_{(1)}$ - $db_{(2)}$
$ratio, base \leftarrow \frac{nb_{(1)}}{diff}$
$count[1...p] \leftarrow (0, ..., 0)$
$dist_{(2)} \leftarrow dist_{(1)}$
foreach *(i,j) in (*$1...mb$, $1...nb$*)* **do**
  $node \leftarrow dist_{(1)}[i, j]$
  if *diff[node] > 0* **then**
    $count[node] \leftarrow count[node] + 1$
    if *count[node]* $\geq$ *ratio[node]* **then**
      $neediest \leftarrow$ which.min($diff$)
      $dist_{(2)}[i, j] \leftarrow neediest$
      $diff[neediest]$++
      $diff[node]$−−
      $ratio[node] \leftarrow ratio[node] + base$[node]
      if *diff[neediest] > 1* **then return**
    end
  end
end

The algorithm's general idea is to iterate over the computationally intensive phase distribution computing the current ratio of blocks each node should give or receive by the total number of blocks it is in the moment. The owner of the block changes if a node has more blocks than it should, giving it to the neediest node. If a node has twice as many blocks as it should have, its base ratio is two, and at every two blocks that the algorithms pass through that owner, one block moves to the neediest node. Since the computational intensive distribution in both cases is uniformly spread over the nodes, this cyclic update also ensures a uniform node spread of the generation but respects processing speeds. Figure 6 (right) presents the final distribution yield. This distribution minimizes the communications while aiming to reach the ideal number of blocks per node. It is possible to see similarities between this distribution and the factorization one, particularly the vertical stripes for nodes 1 and 2 and the horizontal stripes for nodes 3 and 4.

The computation of the ideal number of blocks per node for the first phase comes from the LP result. In ExaGeoStat, the number of tasks is straightforward from the α values. In Diodon, because we use single dimensions distributions, we get α, compute the relative power, and apply a 1D distribution again (with the number of blocks as the number of lines, even in situations where there will be more tasks because of the reading grid) and count the number of tasks (rows) per node.

5. Performance Evaluation

This Section presents the proposed strategies' performance evaluation. It includes the infrastructure description, the phase overlap optimizations evaluation, and the strategies to distribute multi-phase applications over heterogeneous systems.

5.1. Hardware and Software Settings

The experiments utilize the Grid5000 platform (Lille site) and the SDumont supercomputer (Base). Table 1 summarize all the machines' descriptions. In Grid5000, Chifflet and Chifflot have the GTX 1080 and Tesla P100 GPUs, respectively, while Chetemi and Chiclet are CPU-only machines. The operating system for all is Debian 10 with Linux kernel 4.19. The Chetemi and Chifflet network is a 10Gb/s Ethernet, while for Chifflot and Chiclet is a 25Gb/s Ethernet. The experiments used the following configurations to control the environment: (a) Intel hyper-threading off; (b) Frequency governor set to performance; (c) NVIDIA GPUs with maximum clocks and set to persistence mode when possible; (d) network cards with an MTU of 9000; and (e) Network interruptions only to cores of the same network card's NUMA node. In the SDumont site, the only difference between B710 and B715 is the presence of the GPU K40 on the latter. All nodes are interconnected with an Infiniband FDR (56Gb/s) and use RedHat Linux 7.6.

The commits for the software stack used are the following. ExaGeoStat main branch commit `9518886` containing HiCMA, Chameleon, and Stars-H in the fixed submodules. With StarPU developer branch commit `015357bd`, and the communication layer NewMadeleine [25] main branch commit `51d3bf40`. We added in ExaGeoStat and Chameleon some functions to load custom distributions. All the phase overlap modifications are dynamically enabled or disabled during execution time. ExaGeoStat authors made available a list of workloads[2], for which we selected three synthetic ones identified by numbers 8, 9, and 10 with $n = 57600$, $n = 96600$ and $n = 122500$, and a subset of workload 27 with $n = 172800$. These present the best workload ratio to our computational power. Using 960 as block size, we obtain a matrix size of 60×60, 101×101, 128×128, and 180×180 blocks. We use these numbers (60, 101, 128, and 180) to identify each workload.

Diodon experiments use the official main branch version commit `4e6027e2`. NewMadeleine with main branch commit `51d3bf40` while StarPU is `0fb603d8` and Chameleon `6f185f1`. The hdf5 is the 1.10.7 version. Diodon and Chameleon have

[2] `https://ecrc.github.io/exageostat/md_docs_examples.html`

Table 1: Compute nodes available for our experiments.

Site	Machine	CPU	Memory	GPU	Network
Grid5000	Chetemi (Che)	2× Intel Xeon E5-2630 v4	256 GiB	–	10Gb/s Ethernet
Grid5000	Chiclet (Cle)	2× AMD EPYC 7301	128 GiB	–	25Gb/s Ethernet
Grid5000	Chifflet (Chi)	2× Intel Xeon E5-2680 v4	768 GiB	2× Nvidia GTX 1080	10Gb/s Ethernet
Grid5000	Chifflot (Cho)	2× Intel Xeon Gold 6126	192 GiB	2× Nvidia P100	25Gb/s Ethernet
SDumont	B710 (SDC)	2× Intel Xeon E5-2695v2	64GiB	–	Infiniband FDR
SDumont	B715 (SDG)	2× Intel Xeon E5-2695v2	64GiB	2× Nvidia K40	Infiniband FDR

modifications for the proposed optimizations and accept heterogeneous distributions. Finally, the workload consists of synthetic datasets generated by computing the distance of n dimension random points with sizes 48k, 64k, and 128k. The execution environment uses a block size of 640, resulting in workloads of 75×75, 100×100, and 200×200 blocks.

In both cases, the StarPU uses the `dmdas` scheduler with two reserved CPU cores: one for the MPI thread and the other for the application thread responsible for task submissions. We bound the MPI and GPU workers' threads to the cores belonging to the NUMA nodes to which the hardware resource (NIC or GPU) is attached. There is one StarPU process per node.

5.2. *Improving ExaGeoStat Phases Overlap*

The phase overlap strategies in ExaGeoStat were evaluated using two workloads (60 and 101), two sets of machines (four and six Chifflets), and ten repetitions. Figure 7 depicts the results. The X-axis is the cumulative optimizations enabled (from left to right), while Y-axis is time in seconds with a 99% confidence interval. The percentage values in the far right represent the total gain with all optimizations compacted to the synchronous version. The results indicate that the first three strategies (full asynchronous, new solve algorithm, memory optimizations) were the ones that brought more improvements. The priority and submission optimizations caused minor improvements in the 101 workload or no improvement in the 60 workload. However, ≈10% improvements were further seen in heterogeneous scenarios. The last optimization, over-subscription, brought a slight yet consistent improvement in all cases. All these optimizations represent a total gain from 31% in the 101 workload with four machines up to 46% in the 60 workload with six machines when compared to the synchronous version. The rest of the Section presents a comprehensive analysis of how the optimizations improved the behavior and performance.

The execution visualization of three different optimization cases is present in Figure 8 for the four machine case with the 101 workload. In the left panel (`Async`), the first case is the **full asynchronous**. The second one in the middle panel (`+ New Solve + Memory`) is the one with two extra optimizations concerning the first one, the **new solve algorithm** and the **memory optimizations** strategies. The last one in the right panel (`All optimizations`) contains all the optimizations. When comparing the simple asynchronous execution with the earlier Figure 4 that is the synchronous one, the synchronizations barriers disappear, meaning that factorization tasks (green) are executed

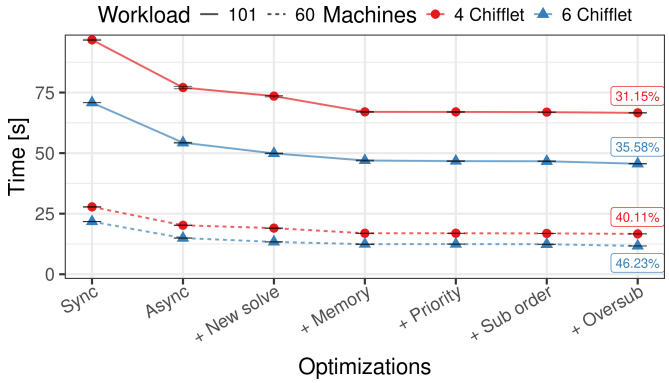

Figure 7: Performance comparison of our phase overlap improvement strategies against the synchronous version of ExaGeoStat.

in GPUs alongside the generation ones (yellow) in the CPU. Yet, the simple asynchronous case on the left still has improvement potential, as idle time is present at the beginning of the execution and during the solve phase at the end. As discussed earlier, those problems may be related to communication and memory utilization, even in the asynchronous case. This assumption is confirmed when applying the first optimizations (the new solve algorithm and memory optimizations) present in the center panel of Figure 8. There is no gradual memory consumption (allocations are expensive) when comparing annotation A.2 to A.1. Also, resource utilization is almost at 100%, as shown by comparing annotation B.2 to B.1. The total amount of communication, computed from traces, is reduced from 11044MB in the asynchronous version to 8886MB with the New Solve optimization.

The right execution of Figure 8 also shows the differences in the behavior of the last three optimizations (Priorities, Submission order, and Over-subscription) when compared to the middle one, though with minor performance gains. The first difference is that the factorization iteration parallelism is higher, as seen in annotations C.3 to C.2, indicating a faster start, advancing the critical path sooner and releasing more tasks. This situation is further confirmed by comparing annotations D.3 to D.2, which shows the slow start's corrections and the resources' full utilization much earlier. The second main difference is that generation tasks run at the beginning of the execution, without misplaced tasks in the middle, as shown in annotations E.3 with E.2, a result mainly from the defined priorities. A third difference is the start of the late phases. As seen in F.1 to F.2, the

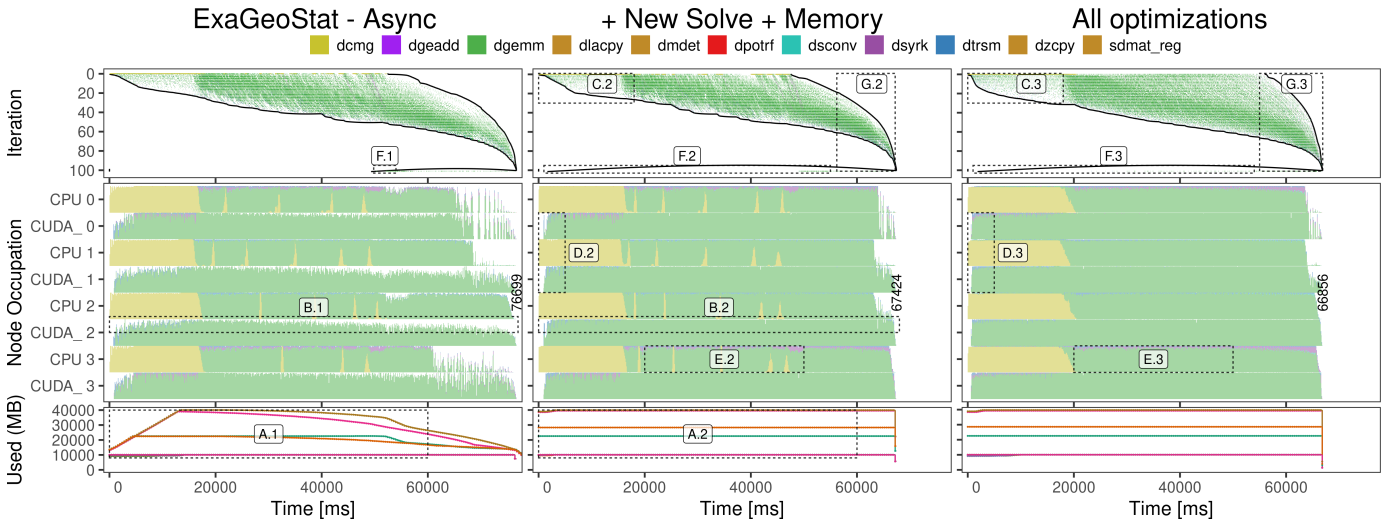

Figure 8: Cholesky Iteration, Node occupation, and Memory utilization panels using 4 Chifflet for one ExaGeoStat iteration in three cases: Asynchronous, Async + New solve + Memory optimizations, All optimizations.

solve can start when it's ready; however, most solve tasks in F.2 occur after the middle of the execution. With the priorities, most of the F.3 region tasks occur gradually during the execution. Finally, annotations G.2 and G.3 present the behavior at the execution's end. While in G.2, there is a more gradual ending, in G.3, there is an abrupt closure, indicating the preference to delay some tasks in the first iterations.

Another metric that indicates the strategies improvements was resource utilization. This metric is the application's complete tasks time divided by the working time of all resources. Application tasks time does not include runtime overhead. The total resource utilization for each of the three executions in Figure 8 is 87.74%, 95.78%, and 96.83%, respectively. Also, when only considering the first 95% of the iteration, the metric is 93.01%, 99.55%, and 99.55%. This result indicates that the last possible improvements are at the application's end when parallelism diminishes and working in the critical path becomes more important [19]. All the results here indicate that the strategies improved phase overlap in ExaGeoStat and increased resource utilization. These improvements are essential for the multi-phase distribution strategies over heterogeneous systems.

5.3. Improving Diodon Phases Overlap

In the Diodon case, Figure 9 presents the evaluation of the optimizations, and it consists of using three different workloads sizes for the matrix A (facets) and two different sizes for the matrix Ω (Line shape). The optimizations are present on the x-axis and the makespan until the SVD is on the vertical axis. Two groups of machines were used, four (red) and six chifflets (blue). The percentage in the right part presents the final improvement compared to the original sync version. The optimizations improved the performance from 29% up to 40%.

Moreover, the following behaviors of individual optimizations highlight. An asynchronous flow without consideration impaired the performance in most cases. While the allocation, split Gram, and commute optimizations improved performance by the same amount, a considerable performance gain appears only when the read order optimization is enabled. We stress that this is caused not only by this optimization alone but also by combining the split Gram, commute, and reading order. Finally, the priorities harvest the last final gains and are more visible in large Ω sizes.

The optimizations modify the behavior of the execution. Figure 10 presents the workers' utilization of three different optimizations over four nodes' execution. The left case presents the original asynchronous execution. In this situation, the read phase had problems because of allocation (A.1), and it is possible to observe the hard algorithm synchronization point in the `gram` (blue tasks in B.1). The center case used the allocation and split Gram optimizations. The read tasks are now smooth (A.2), but the `dgemm` did not start earlier than possible (B.2). This behavior results from a poor overlap because of the commute problem, extra communication, and lack of priorities. The right and final case presents the behavior using all optimizations, giving a clear execution and overlap of read and `dgemm` tasks (B.3). This last case shows a makespan of 18.6s compared to the asynchronous naive case of 29.9s. The beginning of the execution still presents some idle time caused by allocations for the matrix multiplication matrices (A.3). However, even if the `dgemm` tasks could start earlier, the improvement would be limited, as there will be an inevitable algorithm synchronization in the `gram_post` tasks before the QR factorization. The distance between those tasks and the last read tasks is small.

5.4. ExaGeoStat phases partitioning in heterogeneous clusters

The experiments use workloads 101, 128, and 180 over 18 different sets of machines combining the available ones on each system, as depicted by the panels of Figure 11. These sets of machines demonstrate different system-level heterogeneity levels. Each panel shows the makespan in seconds (Y-axis) as a function of the distribution strategy (X-axis and colored bars). The first three bars of each panel are our baselines: (1) the homogeneous block-cyclic distribution using all the resources (*BC*

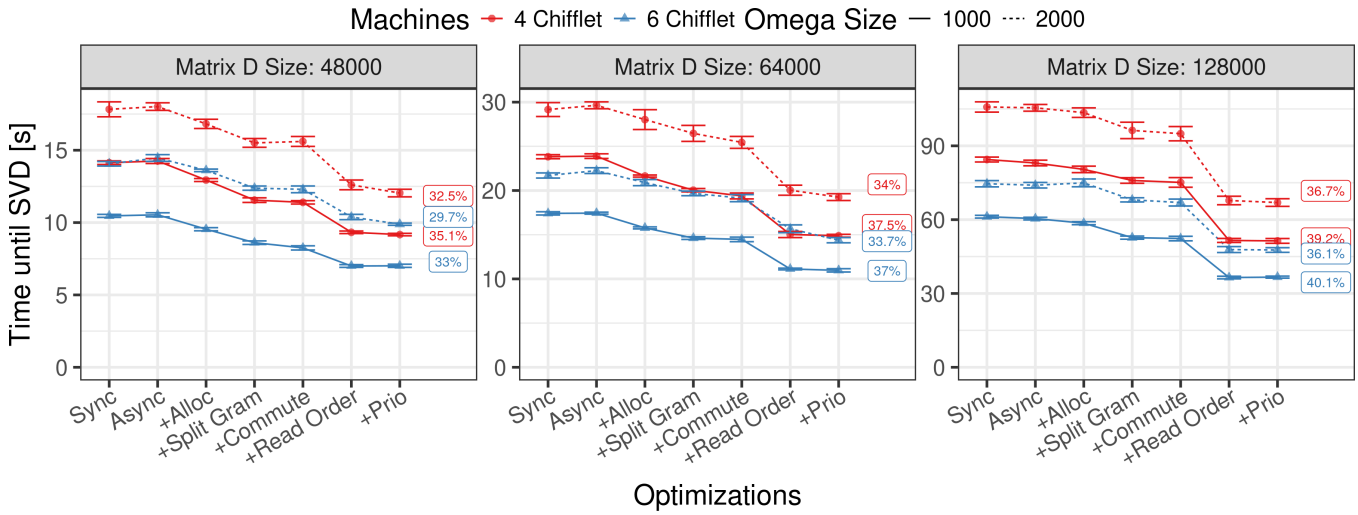

Figure 9: Performance comparison of our phase overlap improvement strategies against the Diodon synchronous version.

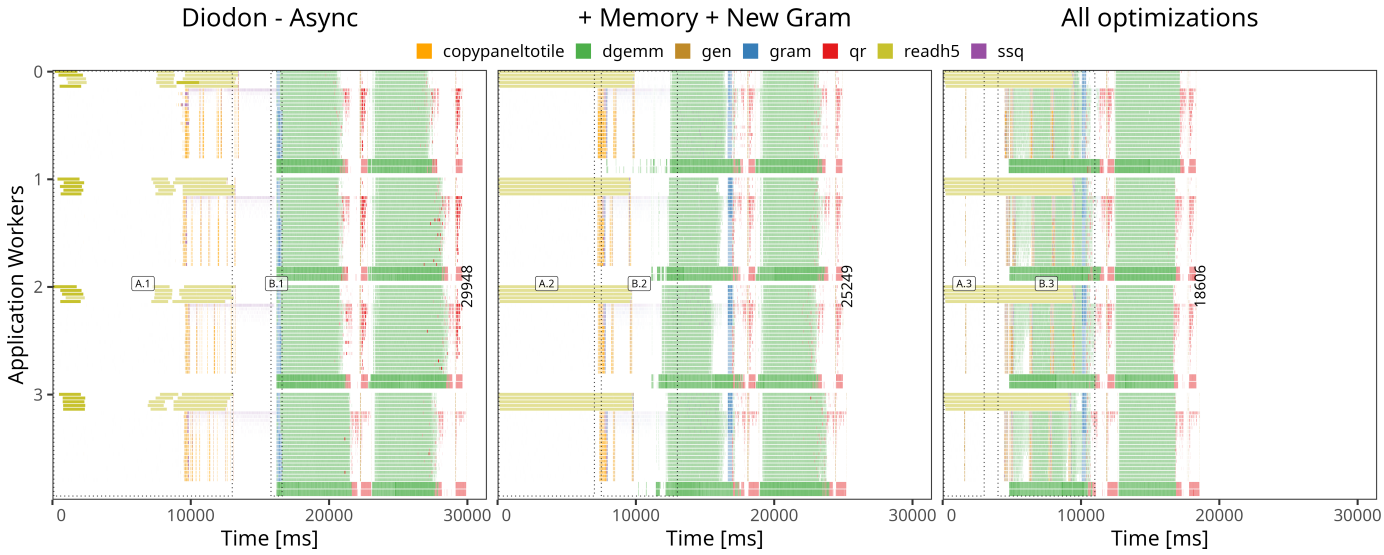

Figure 10: Node occupation using 4 Chifflet for Diodon: Asynchronous, Async + Allocation + Split Gram, All optimizations.

on All in red); (2) the homogeneous block-cyclic distribution for the fastest homogeneous subset of nodes (*BC on fast* in blue); and (3) the heterogeneous 1D-1D distribution using the powers of machines computed considering the `dgemm` speed (*1D-1D* in green). These baselines always use the same distribution for the factorization and generation phases. Our proposed distributions, using the linear program and the workflow to compute two distinct distributions for each phase with the factorization distribution using the normal 1D-1D (*LP HetDist* in purple) or the 1D-1D Constrained distribution (*LP HetDist Constrained* in orange). The orange bar also presents an inner white bar inside to represent the ideal makespan obtained by the linear program. The last group of machines presented is usually the fastest homogeneous subset of nodes (used in the block-cyclic distribution in blue). However, in cases with only one Chifflot, the single Chifflot machine cannot perform this workload well because of high GPU memory utilization. For these cases, the *BC on Fast* result indicates the second most powerful partition.

We can observe in Figure 11 that the block-cyclic distributions are never the best result, neither using all the resources (red) nor the fastest homogeneous subset of nodes (blue). Overall, the linear program distribution (purple and orange) performs very well. Compared to the homogeneous cases (*BC on Fast*), using the LP is always beneficial: the most significant gain is 69.8% on the 4 Chiclet + 1 Chifflot 101 case, while the lowest gain is 6.5% on the 6 Chiclet + 4 Chifflot 101.

Compared to the single heterogeneous distribution (1D-1D), using the LP is most of the time beneficial or ties, but there are a few situations (6 Che + 6 Chi + 1|2 Cho) where it slightly degrades performance. This last situation occurs because the gains by correctly balancing the load are small, and the imbalance compensates for the differences when phases overlap in the heterogeneous nodes. Also, when only using one distribution, there is no cost for redistribution. The minimal difference between the ideal execution time obtained by the linear program and the actual makespan for the good cases shows that the

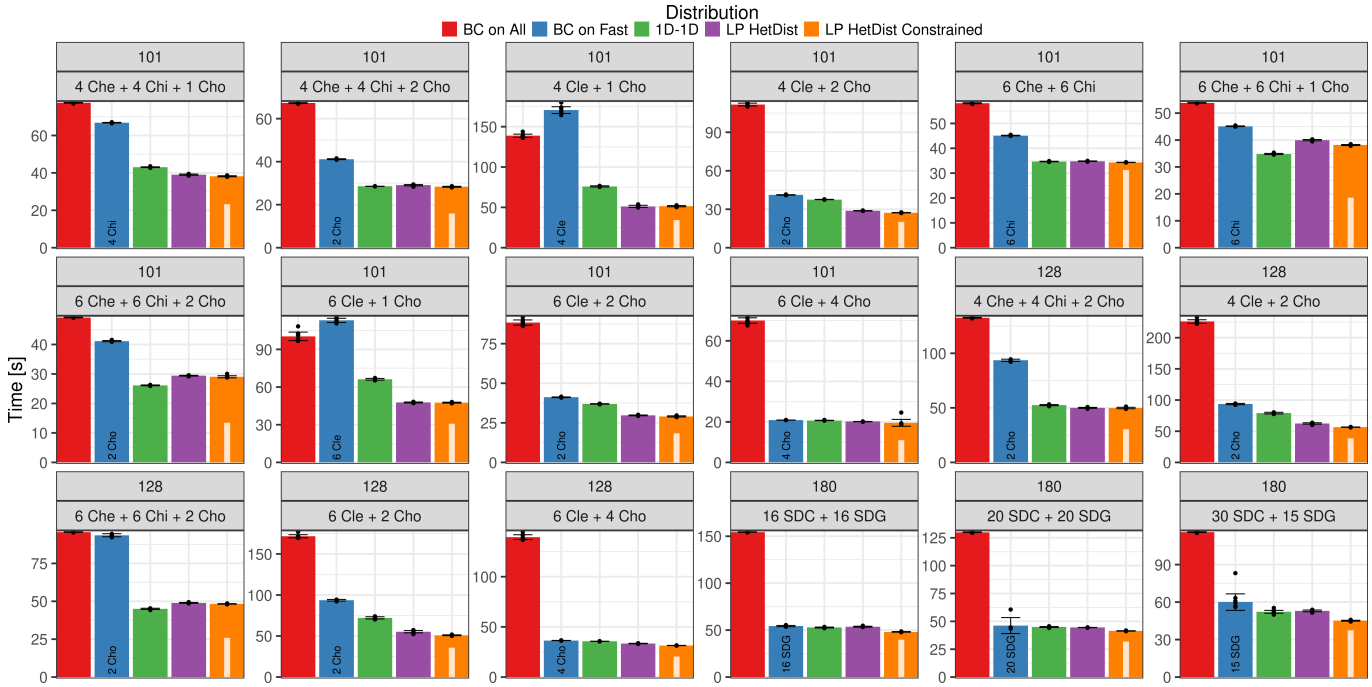

Figure 11: ExaGeoStat makespan for homogeneous and heterogeneous distributions in 18 machine configurations.

redistribution communication overhead is wholly overlapped. However, when using the three levels of heterogeneity with the Chifflot nodes (the cases where the LP distributions are worse than the pure 1D-1D), the execution time is much larger than the LP solution. Further investigation indicates that the problem is the communication of the fast nodes with the slow ones and the ratio of work and power in this configuration. The following section analysis this situation.

5.4.1. Analysis of a case when using too many fast nodes

Using more nodes may not always improve performance. Sometimes it can impact negatively. Considering this, Figure 12 presents the iteration (top) and Node occupation (bottom) panels for three different cases: 6 Che + 6 Chi (left), 6 Che + 6 Chi + 2 Cho using all the resources in the factorization (center), and 6 Che + 6 Chi + 2 Cho using only nodes with GPUs in the factorization (right). Both the execution with two types of machines and the one shown in Figure 8 (right) have low idle times. Moreover, the transition in Figure 12 (left) from generation to factorization is smothered in CPU-only nodes (A.1 annotation) and then in GPU nodes (B.1 annotation). In GPU nodes, the high-priority tasks will execute on the faster resource, while in CPU, they begin to share the resources with some generation tasks. In this execution, it is also possible to see the effect of the constrained version, as the work ends earlier in the CPU-only nodes and then in the fastest ones (C.1).

When adding two Chifflot nodes (represented in the very bottom of the center of Figure 12), the P100 GPU computes the `dgemm` task 10× faster than the Chifflet nodes, adding extra heterogeneity, and the need for faster communication. While the overall makespan decreases, much idle time is visible (D.2 annotation). Further investigation revealed that communication along the critical path and the small workload for this set of machines are responsible for such high idle times. One problem is that the factorization load is highly different in the nodes; the two very fast nodes receive most of the computation and are aid slight by the slower nodes. A lot of communication is necessary to propagate all the computation done by the GPU nodes, and even with their faster network, it's not enough, mainly because these nodes share a different subnet in the Lille site. This problem is also why cases 4+4+2, 6+6+1, and 6+6+2 of Figure 11 have a distant LP prediction to the result. Another evidence for this network problem is that the pure 1D-1D uses a single distribution across all phases and then requires less communication, trading distribution balance for communication improvements. One possible technique to circumvent the communication problem is limiting the number of nodes during the factorization, which is the phase causing most communication operations. We can easily set such a limit by excluding the nodes without GPUs from the factorization in the LP constraints. The case in the right of Figure 12 depicts the resulting behavior. The idle time decreases (D.3 annotation), indicating a better usage of the nodes, leading to a slight decrease in the makespan with fewer resources. This case provides a mean makespan of ≈26s. Remaining a difference of 50% between the actual makespan and the LP result, so enhancements in communication, or a better selection of machines for this workload, should improve this even further. Overall, comparing this result against the original synchronous 2 Chifflot homogeneous execution (≈41s), we have a performance improvement of 41%. This result reinforces the necessity of having strategies to decide the correct number of nodes considering communication [26], and after this inquiry, using such distributions on the correct number of nodes.

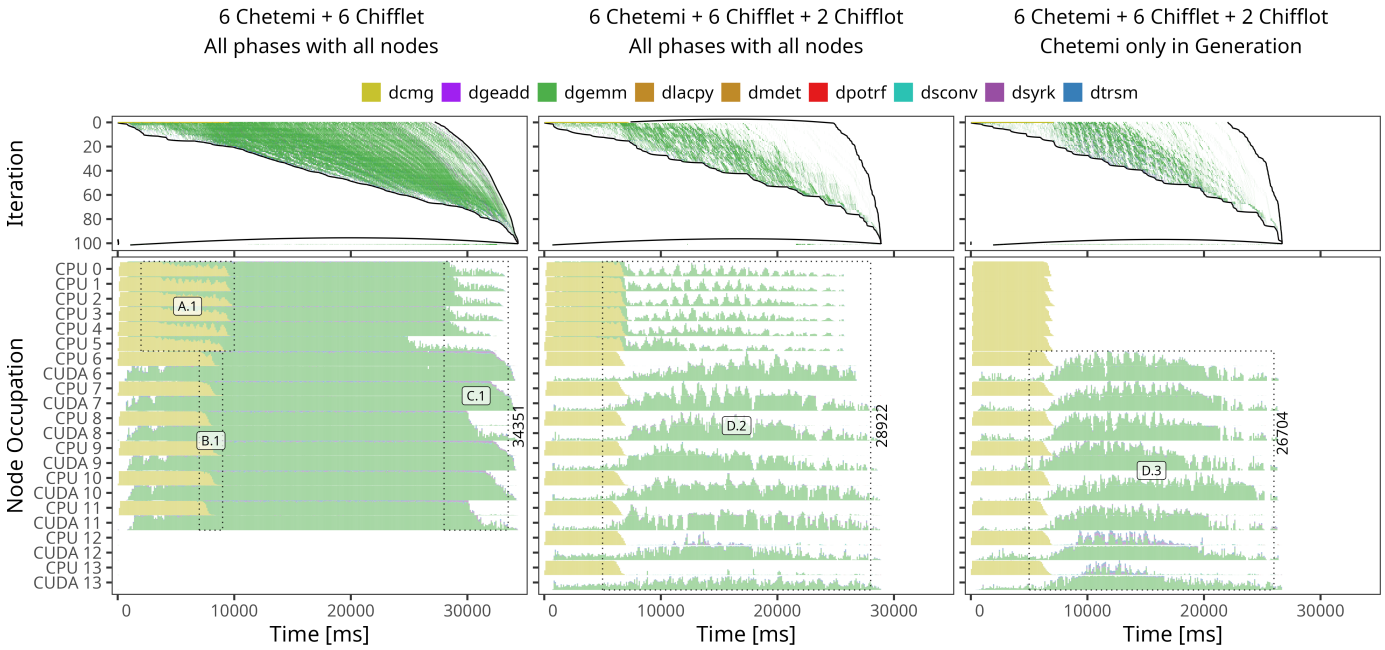

Figure 12: Cholesky Iteration, Node occupation, and Memory utilization panels of the ExaGeoStat iteration using the LP HetDist Constrained distribution for three sets of machines: 6+6, 6+6+2, and 6+6+2 restricting factorization to GPU-only nodes.

5.5. Diodon phases partitioning in heterogeneous clusters

The experiments use the 75, 100, and 200 workloads for the matrix D and $r = 1000$ for Ω. Figure 13 shows the makespan (application start to the last `gram` task) in 18 machine scenarios with five different distributions. As before, the white bar inside shows the LP ideal makespan.

Once again, the first three bars of each panel are our baselines: 1) *BC on All* uses all the available machines on that configuration but with the traditional block-cyclic distribution (red); 2) *BC on Fast* shows the case of using only the fastest nodes (with GPUs) for both phases with the original homogeneous distribution (blue); 3) The *1D* is the version with the heterogeneous distribution (with one dimension) that considers node heterogeneity for the factorization while the generation phase uses the standard round-robin distribution (green). In all these cases, the read phase does not require heterogeneous awareness as the relative read power is very similar across such nodes. Our proposed distributions are the *LP HetDist* (purple) and *LP HetDist Constrained* (orange). The *LP HetDist* version describes the case when using the LP to compute the ideal capacity of each machine for each phase, considering the overlap. It uses the 1D heterogeneous distribution algorithm for the computational intensive phase and the derivative fewer communication one for the read phase. The *LP HetDist Constrained* version is similar to the *LP HetDist*, but instead of the original 1D heterogeneous distribution for the computation phase, it uses the constrained version that will only assign rows to a node if the CPB is higher than the application expected duration.

The results of Figure 13 show the improvements when using heterogeneous resources and adequate distributions over them considering phases overlap. In most cases, using the baseline strategy *BC on Fast* presents the worst results, as the reduction in computational power, especially in the read phase, is considerable. The performance sometimes improves by adding different extra nodes using the original homogeneous distribution (*BC on All*). There are positive and negative cases when transitioning to heterogeneous independent distributions (*1D*). However, the performance improves in all cases only when using the LP-based versions. In some cases, especially where the workload ratio to machines is lower (less workload per node), our constrained version (*LP HetDist Constrained*) improves considerably. Also, another advantage of using more heterogeneous nodes is that they may handle larger problems if the application fails to handle large workloads in fewer nodes. This is the situation of the case 12 SDC + 4 SGD, where the workload of 200 was unable to run using only 4 SDG. When comparing the *LP HetDist Constrained* version with the block-cyclic one on the fast nodes only, the greater performance increase was in case 4 Cle + 4 Chi + 1 Cho with 73.1%, and the lesser performance increase was in 4 Cle + 4 Chi with 24.6%.

6. Applying the Proposed Approach to other Applications

This Section summarizes how to apply the techniques to an existing task-based application that already contains a series of operations potentially asynchronous. The general idea comprises three parts. **Part 1:** Characterize the application execution, identify asynchronous operations, and improve overlap. Trace visualizations like the ones in this article can help in this step. The user should check possible optimizations described in the main categories at the beginning of Section 4.2. These optimizations should focus on modifying the DAG and operations to increase the possible phase overlap. It includes changes in the operations' tasks, critical path improvements, and giving the correct task priorities to the scheduler. A good overlap

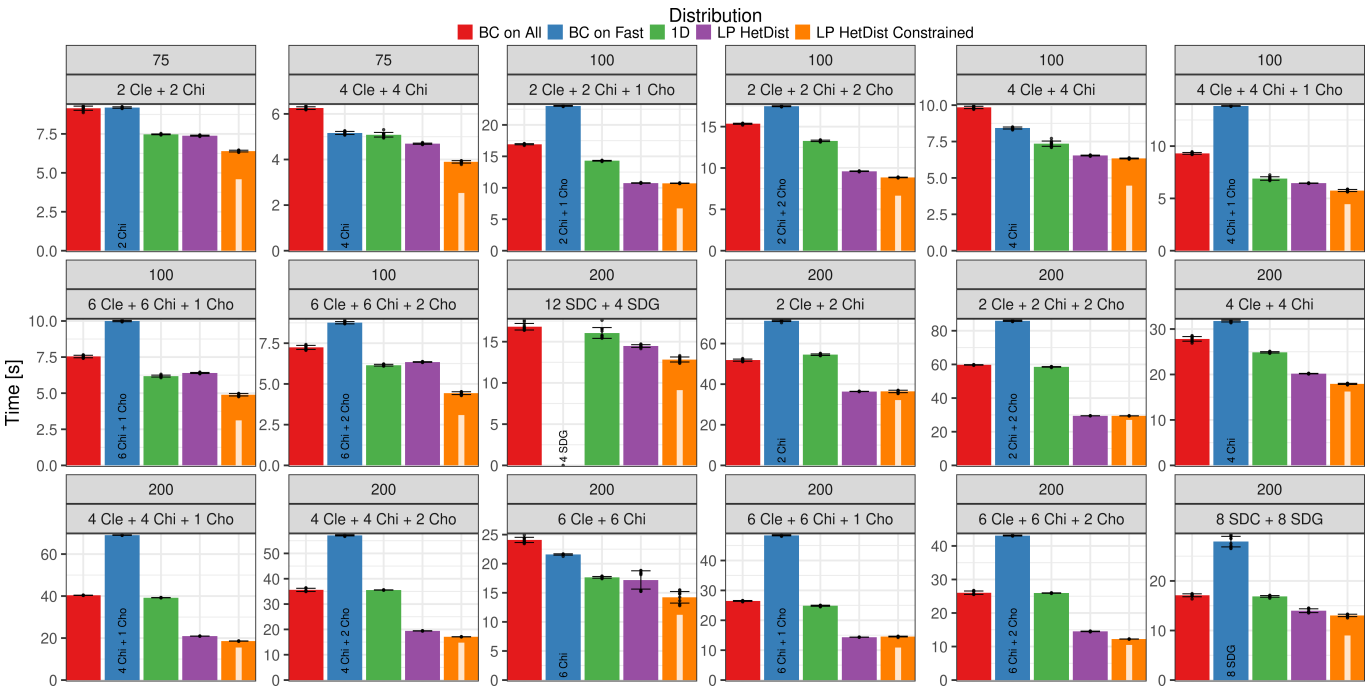

Figure 13: Diodon time to Gram operation for homogeneous and heterogeneous distributions in 18 machine sets configurations.

would mean that tasks of latter operations can start earlier and use their best resource. Also, the user should check idle times, correlating it to possible memory or communication problems and potentially correcting them. These problems are usually application-specific. The phase characterization must provide the performance models of the selected operations' tasks, which the LP will use. **Part 2:** Once the application has good behavior, it is necessary to define the phases and steps to apply the techniques. A phase is a significant operation or multiple operations with similar behavior, where larger operations should incorporate the smaller ones. A step is a set of blocks that their tasks (independent of the operation) could and should run simultaneously. In the Cholesky case, we define it as the anti-diagonal, as the general flow is one anti-diagonal per time, whereas, in the matrix-matrix multiplication, we define it as per row because of the reading part. This notion is not a strict definition, where tasks from different steps could not run in parallel, but an intuitive understanding of the data flow. **Part 3:** With the performance models and the step definition, the user can use the LP to model the application and get the ideal number of tasks per type and node. Equation 13 computes the relative powers of each phase for each node. These powers are the standard input for many distribution algorithms, including the 1D-1D constrained. The distribution of the compute-intensive phase should have precedence and have its distribution computed first using the traditional distribution algorithms. If there are limited dependencies, the other phase could use the algorithm discussed in Section 4.4 to compute a distribution while minimizing redistribution cost. Ultimately, the application will have optimized asynchronous operations with substantial overlap and distributions for each phase considering resource heterogeneity, phase overlap, and redistribution communication.

7. Discussion and Conclusion

As heterogeneity becomes even more prominent at a system level, when there are distinct nodes, correctly distributing applications in such an environment enables them to exploit it and better accommodate their load. This paper presents optimizations for improving the overlap of asynchronous multi-phase task-based applications and strategies for distributing them over heterogeneous system-level resources. Two real scientific applications, ExaGeoStat and Diodon, provide genuine scenarios for this research. First, we demonstrate three general optimization categories that improve the overlap of asynchronous phases. These optimizations enable the latter phases' critical tasks to execute earlier and reduce resource idleness, enhancing performance on homogeneous scenarios from 29% to 46% (see Section 5.2). When considering the heterogeneous case, we present strategies that compute the ideal relative power for each phase on each group of nodes. A linear program models the application's flow considering tasks and resource heterogeneity. The relative power is extracted from the linear program and later used in traditional distribution algorithms for some computationally intensive phases. Also, we provide an algorithm to design a distribution of a previous generation phase that maintains data cyclically, reduces redistribution communications, and balances the load for that particular phase. This new distribution is distinct but tightly coupled to the next one. All those heterogeneous distribution strategies enhance the performance (see Section 5.4) up to 69% compared to a simple homogeneous setup in ExaGeoStat, and 24% to 73% in Diodon.

Future works include adding communication in the model, which would improve the decisions when using too many nodes for a particular workload. The excessive use of nodes is ex-

pensive and not necessarily valuable, as performance usually deteriorates because of communication overheads. However, modeling when communication occurs exactly in a dynamic runtime is complex, especially with system-level heterogeneity involved. The StarPU-Simgrid [19, 27] simulations may be an essential ally when trying to achieve this goal. Another possibility is the use of execution data combined with machine learning algorithms. Some works consider this problem dynamically during application execution [26], and use online execution information. Moreover, another opportunity is the investigation of situations where tasks could have different precision in the blocks (multi-precision or low-rank), like in ExaGeoStat. The LP modeling in such cases seems reasonable, but the final mapping can present challenges on heterogeneous configurations.

Software and Data Availability. We endeavor to make our analysis reproducible. A public companion (`https://gitlab.com/lnesi/companion-fgcs`) contains the data and instructions to reproduce our results.

8. Acknowledgments

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), grant no 141971/2020-7 to the first author, and the projects: FAPERGS (Data Science – 19/711-6), Inria (Associated Team ReDas), CAPES (Cofecub 04/2017). Experiments were carried out using Grid'5000, 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`). The authors acknowledge the National Laboratory for Scientific Computing (LNCC/MCTI, Brazil) for providing HPC resources of the SDumont supercomputer, which have contributed to the research results reported within this paper (`http://sdumont.lncc.br`). The authors would like to thank Hatem Ltaief for his work in ExaGeoStat; Mathieu Faverge, and Florent Pruvost for their work in Diodon and Chameleon; and the fruitful discussions about this work with all of them.

References

[1] J. J. Dongarra, et al., With extreme computing, the rules have changed, Computing in Science & Engineering 19 (3) (2017) 52.

[2] U.-U. Haus, E. Laure, S. Narasimhamurthy, E. Suarez, Heterogeneous high performance computing, ETP4HPC White Paper (2022).

[3] J. J. Dongarra, H. W. Meuer, E. Strohmaier, et al., Top500 supercomputer sites, Supercomputer 13 (1997) 89–111.

[4] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, J. Planas, OmpSs: a proposal for programming heterogeneous multi-core architectures, Parallel Processing Letters 21 (02) (2011) 173–193.

[5] G. Bosilca, et al., Parsec: Exploiting heterogeneity to enhance scalability, Computing in Science Engineering 15 (6) (2013) 36–45.

[6] C. Augonnet, et al., StarPU: A unified platform for task scheduling on heterogeneous multicore architectures, Concurrency and Computation: Practice and Experience, SI: EuroPar'09 23 (2011) 187–198.

[7] S. Abdulah, H. Ltaief, Y. Sun, M. G. Genton, D. E. Keyes, Exageostat: A high performance unified software for geostatistics on manycore systems, IEEE Transactions on Parallel and Distributed Systems 29 (12) (2018) 2771–2784.

[8] P. Blanchard, O. Coulaud, E. Darve, A. A. Franc, FMR: Fast randomized algorithms for covariance matrix computations, Platform for Advanced Scientific Computing (PASC), poster (Jun. 2016).

[9] E. Agullo, et al., Task-based randomized singular value decomposition and multidimensional scaling, Research Report RR-9482, Inria Bordeaux - Sud Ouest; Inrae - BioGeCo (2022).

[10] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, S. Tomov, Faster, cheaper, better – a hybridization methodology for high-performance linear algebra software for GPUs, in: GPU Computing Gems Jade Edition, Elsevier, 2012, pp. 473–484.

[11] L. Leandro Nesi, A. Legrand, L. Mello Schnorr, Exploiting system level heterogeneity to improve the performance of a geostatistics multi-phase task-based application, in: Proceedings of the 50th International Conference on Parallel Processing, ICPP '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 1–10.

[12] C. Augonnet, J. Clet-Ortega, S. Thibault, R. Namyst, Data-Aware Task Scheduling on Multi-accelerator Based Platforms, in: 2010 IEEE 16th International Conference on Parallel and Distributed Systems, 2010, pp. 291–298.

[13] L. S. Blackford, et al., ScaLAPACK User's Guide, Society for Industrial and Applied Mathematics, USA, 1997.

[14] R. B. Gramacy, Surrogates: Gaussian Process Modeling, Design, and Optimization for the Applied Sciences, Chapman & Hall/CRC Texts in Statistical Science, CRC Press, United States, 2020.

[15] M. Folk, G. Heber, Q. Koziol, E. Pourmal, D. Robinson, An Overview of the HDF5 Technology Suite and Its Applications, in: Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases, AD '11, Association for Computing Machinery, New York, NY, USA, 2011, p. 36–47.

[16] A. Kalinov, A. Lastovetsky, Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers, Journal of Parallel and Distributed Computing 61 (4) (2001) 520–535.

[17] O. Beaumont, V. Boudet, F. Rastello, Y. Robert, Matrix multiplication on heterogeneous platforms, IEEE Transactions on Parallel and Distributed Systems 12 (10) (2001) 1033–1051.

[18] O. Beaumont, A. Legrand, F. Rastello, Y. Robert, Static LU decomposition on heterogeneous platforms, International Journal of High Performance Computing Applications 15 (2001) 310–323.

[19] L. L. Nesi, L. M. Schnorr, A. Legrand, Communication-aware load balancing of the LU factorization over heterogeneous clusters, in: 26th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2020, IEEE, Hong Kong, 2020, pp. 54–63.

[20] L. Prylli, B. Tourancheau, Efficient block cyclic data redistribution, in: L. Bougé, et al. (Eds.), Euro-Par'96 Parallel Processing, Springer Berlin Heidelberg, Berlin, Heidelberg, 1996, pp. 155–164.

[21] M. Gates, J. Kurzak, A. Charara, A. YarKhan, J. Dongarra, Slate: Design of a modern distributed and accelerated linear algebra library, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 1–18.

[22] J. Herrmann, et al., Assessing the Cost of Redistribution Followed by a Computational Kernel, Parallel Comput 52 (C) (2016) 22–41.

[23] V. Garcia Pinto, et al., A visual performance analysis framework for task-based parallel applications running on hybrid clusters, Concurrency and Computation: Practice and Experience 30 (18) (2018) e4472.

[24] L. Nesi, S. Thibault, L. Stanisic, 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, pp. 142–151.

[25] A. Denis, Scalability of the NewMadeleine Communication Library for Large Numbers of MPI Point-to-Point Requests, in: 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), IEEE, Cyprus, 2019, pp. 371–380.

[26] L. L. Nesi, L. M. Schnorr, A. Legrand, Multi-Phase Task-Based HPC Applications: Quickly Learning how to Run Fast, in: 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2022, pp. 357–367.

[27] L. Stanisic, et al., Faithful performance prediction of a dynamic task-based runtime system for heterogeneous multi-core architectures, Concurrency and Computation: Practice and Experience 27 (16) (2015) 4075–4090.